Design Critiquing Systems

Jason E. Robbins

jrobbins@ics.uci.edu

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92697-3425

ABSTRACT
Design critiquing systems are a type of intelligent user interface used to support human designers in decision making. This paper places design critics in the larger context of intelligent user interface approaches and surveys several critiquing systems. Each approach and system is evaluated with respect to a five-phase design improvement process. This paper concludes with a summary of the state of the art in critiquing systems and recommendations for future research directions.

1. Introduction

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 making design decisions.

The goals of this survey are to:

- describe the motivations for critiquing systems and their range of application,
- clarify the concepts involved in design critiquing,
- define a design improvement process model that describes interaction between designers and design critiquing systems,
- situate the critiquing approach in the larger context of intelligent user interface techniques,
- review and evaluate existing critiquing systems,
- guide the reader to critiquing systems literature,
- summarize the state of design critiquing today, and
- recommend new directions for research and development of critiquing systems.

1.1. Motivation for Design Critiquing Systems

Brooks outlines some of the accidental and essential difficulties of software development [7]. Designers in all fields face many accidental and essential difficulties during design. Accidental difficulties are those that are somewhat arbitrary and usually result from the historical actions of other people in the field. For example, designers need to deal with the idiosyncrasies of the particular design notation they use. Often accidental difficulties can be eliminated with better tool support. Essential difficulties of design are those that are inherent to the nature of design and cannot easily be resolved or changed. Essential difficulties of design arise from essential activities of design: learning about the problem and solution domains, decision making, managing interactions between decisions, ordering decisions in a design process, communicating with other stakeholders, and evaluating the design. Design critics primarily address the essential decision-making challenges faced by designers, however they can help with accidental difficulties as well.

The inclusion of critics in a design tool is motivated by the following observations of the decision-making needs of human designers.

1. Designers' limited domain knowledge. In complex domains, no single designer has all the knowledge needed to make a complete design. Instead, most complex systems are designed by teams of stakeholders with each stakeholder providing some of the needed knowledge and imposing their own goals and priorities. Even experienced designers need knowledge support in complex domains or when working with unfamiliar design elements. The "thin spread of application domain knowledge" has been identified as a general problem in software development [2]. This problem has been worsened by software's newest crisis: a shortage of trained workers [6]. Design critics can address this by supplying design knowledge at the time when it is needed.

2. Low relative cost of immediate revision. Sound decision making is especially important in the early phases of the software life-cycle. Errors and oversights introduced in high-level design become much more expensive to remove as development proceeds. Typical estimates put the cost of fixing an error during the testing phase at more than ten times the cost of fixing the same error in high-level design [8, 9]. Design critics can address this by identifying errors early.

3. Continuous learning. Designers must continuously learn new skills, design methods, technologies, and design elements [98-104]. For example, designers of component-based software systems must frequently learn new integration technologies and the characteristics of new components. Design critics can address this by identifying problems as they arise and providing explanations of the underlying issues.

4. Cost of failures arising from design errors. In safety-critical systems, financial systems, and other critical systems the potential cost of design errors is great [66]. Design critics provide one form of support for detecting and removing these errors.

5. Time-to-market. The time needed to complete a design task can be substantial. Industry is placing increasing emphasis on development methods that reduce time-to-market. Pressure to reduce design time may result in poor design decisions, fewer design reviews, and ultimately more design errors that offset the benefits of bringing a product to market quickly. Design critics can reduce design time by automatically detecting problems that might otherwise require manual effort to detect. Furthermore, critics can help designers make decisions that help speed other phases of development. For example, a testability critic applied during software design might help reduce the testing effort needed to bring a product to market.

6. Risk management. Iterative development has several advantages over the waterfall development life-cycle, notably risk management [11]. Many of these advantages depend on the software product progressing through a series of intermediate states that are stable and allow evaluation. Design critics can help reduce risk by reducing the number of undetected errors present in the design at any given time.

1.2. The Nature of Design

Design critiquing systems are intended to support designers, so any definition of "design critiquing system" must rely on definitions of "design" and "designer." We use the word "design" to refer to both an artifact and a process.

A design artifact can be generically thought of as a set of interrelated design elements. For example, a design for a computer program might be represented as a set of procedures, classes, variables, source files, user interface screens, and software components. The relationships between these design elements include inheritance, calls, containment, and uses. The main purpose of the design artifact is to guide future implementation, without completely determining all implementation details.

The design process is a series of design decisions made over a period of time. These decisions include choice of design elements, their characteristics, and relationships. The design process is typically both a learning process and an iterative one. As designers progress through the series of decisions, they come to understand more of the implications of the problem and the relevant features of the solution domain; this new understanding allows them to reevaluate previous decisions. Schoen's theory of *reflection-in-action* indicates that designers cannot make design decisions in isolation, instead they must make design decisions in the context of a design process that considers multiple interrelated decisions [108, 109]. Designers must construct a partial design, evaluate, reflect on, and revise it, until they are ready to extend it further. Simon states that "[One can] think of the design process as involving, first, the generation of alternatives and, then, the testing of these alternatives against a whole array of requirements and constraints" [10]. Guindon, Krasner, and Curtis note the same effect as part of a study of software developers [105]. 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.

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 [105, 107, 110]. 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 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 [111]. If they are lacking knowledge about how to structure a solution or proceed with a particular task, they are likely to delay this task. Accessibility of information may also cause a deviation in planned order. If designers must search for information needed to complete a task, that task might be deferred. Complexity of a task roughly corresponds to the number of smaller tasks that comprise it.

Priority or importance of a step is the primary factor that supersedes cognitive cost in decision ordering. 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 attending to overlooked steps or fixing identified errors [107].

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

> The main breakdowns observed are: (1) lack of specialized design schemas; (2) lack of a meta-schema about the design process leading to poor allocation of resources to the various design activities; (3) poor prioritization of issues leading to poor selection of alternative solutions; (4) difficulty in considering all the stated or inferred constraints in defining a solution; (5) difficulty in performing mental simulations with many steps or test cases; (6) difficulty in keeping track and returning to subproblems whose solution has been postponed; and (7) difficulty in expanding

or merging solutions from individual subproblems to form a complete solution. [105]

Furthermore, since designers cannot make all decisions simultaneously at the beginning of the design process, they are likely to use simplifying assumptions and place-holder elements that serve to structure the design and narrow the set of options considered. These initial decisions are tentative and must be reconsidered later.

Designs are typically evaluated with respect to many interdependent criteria. For example, a software system can be evaluated with respect to usability, performance, maintainability, or functional completeness. For each criterion the design may be classified as acceptable or unacceptable. Designs within the acceptable range may usually be compared to determine which is better or worse. For example, a source code file may be syntactically legal or illegal, and within the legal range the code may be more or less readable. Some of these evaluations are objective, while others are subjective. It is often difficult to predict the impact of a particular design decision on a particular evaluation criterion. For example, flattening a class inheritance hierarchy may make the design more maintainable or less maintainable, depending on the nature of future modifications. Since these evaluations involve multiple design elements, decisions based on them must be reconsidered if any of the elements are changed.

1.3. What are Critiquing Systems?

In this subsection we discuss the elements of the critiquing approach, give a very brief scenario of interaction with a critiquing system, review definitions of critiquing systems found in previous work, and present our revised definition.

The elements of the critiquing approach are an artifact being constructed, a designer making decisions or choices about that artifact, and timely feedback from design critics to the designer. One central idea of the critiquing approach is that analysis should be used to improve designers' task performance and that it is most helpful when its results are provided to designers in the context of their decision making. Traditional analysis approaches also seek to inform designers' decision making. However, critiquing differs from traditional analysis approaches in that the designer's cognitive needs are made central. Specifically, we include the designer's mental context as part of the designer's decision making context. This places additional emphasis on delivering feedback quickly, before the designer's short-term memory fades, and on supplying contextual information in the feedback that helps the designer reconstruct prior mental states.

Traditional approaches to design analysis tools follow the *authoritative assumption*: they support design evaluation by proving the presence or absence of well defined properties. This allows them to give definitive feedback to the designer, but limits their application to late in the design process after the designer has formalized substantial parts of the design and may have lost the mental context of the problematic decisions. The critiquing approach follows the *informative assumption*: designers are assumed to normally make design decisions on their own, and analysis is used to support designers by informing them of potential problems, possible corrective actions, and pending decisions. Critics are written to pessimistically detect potential problems. They need not go so far as to prove the presence of problems; in fact, formal proofs are often not possible, or meaningful, on partially specified designs. This approach aids designers in reviewing their decisions, avoids the need to assume that critics have complete knowledge, and facilitates incremental development and improvement of critics.

For example, a software designer using Argo/UML might experience the following scenario. The designer uses Argo/UML in much the same way that other object-oriented design tools are used: he or she places classes and associations in a design diagram. Upon placing each class, several critics fire to indicate that part of the design has been started, but not yet finished: the new class has not yet been given a name and it lacks instance variables, associations, and methods. As the designer works to further define the design these incompleteness critics withdraw their criticism, but new criticism may be raised. For example, a correctness critic fires if the designer specifies circular inheritance. Design feedback from critics is always available during normal tool use, but it is presented so as not to disrupt the designer's train of thought. In Argo/UML small visual indications of errors are placed directly on the design diagram, and a hierarchical list of all outstanding feedback items is always available (Figure 1). At any time the designer may view design feedback, improve his understanding of the state of the design and domain knowledge, and move forward with the design by resolving identified problems. A more detailed usage scenario is presented in the next section, and brief scenarios are presented as needed to explain the systems reviewed.

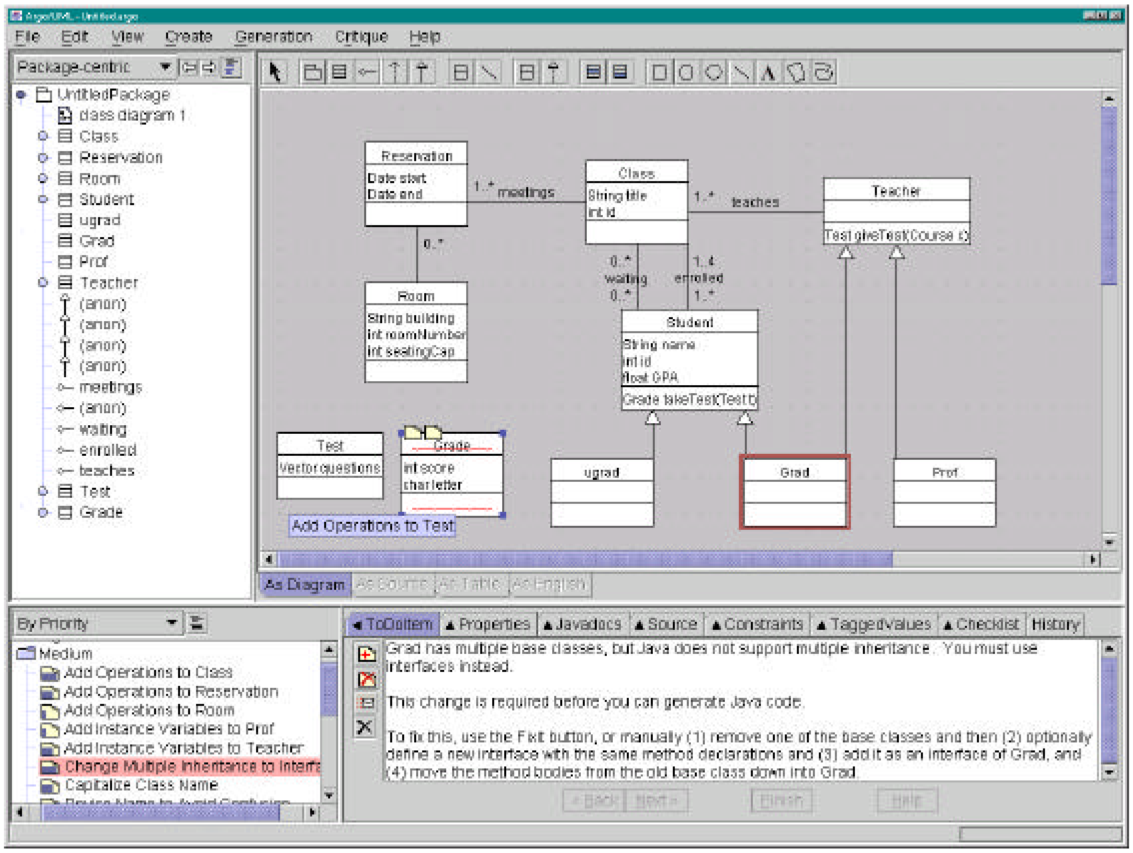
Table 1 shows some of the definitions of critiquing systems found in the literature. We have added italics to each definition to highlight key phrases that differentiate it from the others.

The definition in Table 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. However, 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 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

Figure 1: Screenshot of Argo/UML



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 FixIt 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.

Table 1: Selected Definitions of Critiquing System

Langlotz and Shortliffe 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.” [36]
Miller on ATTENDING: “A critiquing system is a computer program that *critiques human generated solutions*.” [39]
Fischer et al. 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.” [52]
Silverman and Mehzer describing their theory of error and critiquing: “Expert critiquing systems are a class of program that receive as input the statement of the problem and the user-proposed solution. They produce as output a *critique of the user’s judgment and knowledge* in terms of what the program thinks is wrong with the user-proposed solution.” [66]
Sumner, Bonnardel, and Kallak 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.” [69]

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 our own.

Silverman’s definition is mainly concerned with the nature of human error and categories of errors. His definition of critiquing implies that critics not only help to correct the errors at hand, but that they identify and may even cure designers of certain human failings. Silverman’s survey of critiquing systems is very inclusive: “For example, source code debuggers and language sensitive editors criticize the syntactic properties of the user’s task result. Code skeleton graphers and dataflow analyzers critique completeness and clarity and code optimizers critique the workability of programs” [4]. To Silverman, the nature of the interaction that the designer has with the system is not the main point. Instead, critics are any system that evaluates the designer with the intent of changing his or her behavior. “The quality of the human’s solution is a sign of the level of task performance he or she is capable of” [4]. In describing how grammar checkers detect the use of passive voice, Silverman writes that “curing technical writers of this class of deeply ingrained errors probably requires... multiple strategies of criticism”[4].

Each of the first four definitions attempts to define or redefine the concept of critiquing. 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.

Our definition of critiquing differs from those in Table 1 in several ways. We 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. Unlike Silverman, we do not desire the system be critical of the designer's abilities. Opportunities for improvement are simply that; we do not interpret them as signs of underlying limitations of the designer. Like Fischer's definition, we require that critics provide cognitive support for human decision-making, but we do not limit that support to a single theory of design. All of the definitions in Table 1 stop at informing the designer of the existence of problems; *we go a step farther by defining the goal of critiquing as helping to carry out design improvements*. We 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.

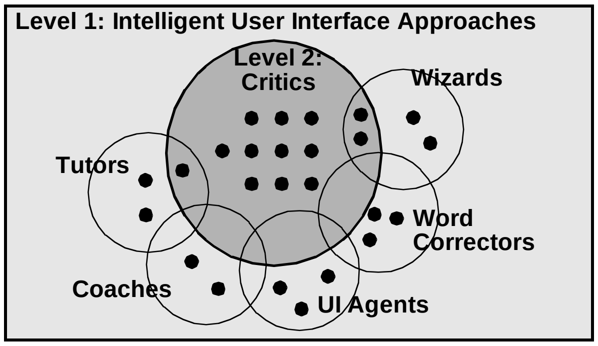
1.4. Scope of the Survey

This survey covers work related to critiquing systems at two levels. First, it describes, compares, and evaluates six approaches to intelligent user interfaces. Then it focuses on systems that follow the critiquing approach. Figure 2 shows a space of intelligent user interfaces, the sets of systems that fall under each approach reviewed in Section 3, and the individual critiquing systems reviewed in Section 4.

The quality of the knowledge contained in these systems plays a large role in determining the quality of their design support. However, this survey does not attempt to evaluate the domain-specific knowledge content of the diverse systems reviewed. Instead, we assume that the embedded knowledge is, for the most part, correct and useful. We focus on the features of each approach or system and evaluate them in terms of their potential support for design tasks.

The preceding subsection gave several definitions of the term "design critiquing system." These definitions are the basis for the desired critiquing system capabilities listed in Table 2. Not all the systems reviewed provide all of these capabilities; unsupported capabilities are discussed for each system in the appropriate subsection of Section 4. Some of these capabilities overlap the others, but they are listed explicitly for clarity.

Figure 2: Survey Scope



Each dot is a system reviewed.

Table 2: Desired Critiquing Capabilities

C1. Identifies opportunities for design improvement
C2. Provides knowledge to support designers in making design decisions
C3. Supports reflection-in-action
C4. Provides feedback about design problems while the designer is still in the mental context of the design decision that caused or exposed the error
C5. Operates on partially specified designs
C6. Distinguishs between good and poor designs, not just legal and illegal designs
C7. Allows heuristic rules
C8. Provides support for carrying out design improvements
C9. Provides feedback in a useful and usable format

In Section 4 we review several examples of critiquing systems. Here are some things that are *not* critiquing systems.

1. Compilers. The primary purpose of a compiler is to convert a program from one language to another (usually machine code). Compilers generate error messages, but they do so in a batch process after the source files have been prepared. Compilers only differentiate between legal and illegal programs; they lack knowledge needed to differentiate good legal programs from poor ones. Any program is accepted if it has a legal interpretation, even if it is unlikely that the user intended that particular interpretation. In contrast, critics differentiate between legal designs and may pessimistically identify likely errors or other improvement opportunities in perfectly legal designs. Furthermore, the feedback produced by compilers is for the most part uncontrolled and unorganized. Compilers fail to provide capabilities C1, C3-C9. Compilers can also act as program checkers (see below) to the extent that they provide warning messages.

2. Program checkers. Program checkers such as “lint” are similar to critiquing systems in that they have knowledge of common errors and can provide advice on potential improvements. Application of heuristic rules can be controlled through a vast array of command line options and preprocessor directives. However, since they are not integrated into the tools that programmers use to enter design decisions (i.e., text editors) and they must be explicitly invoked, they do not provide our desired capability of providing feedback while designers are in the mental context of a decision (C4).

3. Syntax directed editors. Syntax directed editors prevent the construction of syntactically invalid programs by constraining what can be entered [33]. Programs can be in incomplete states, but they can never be in states that are syntactically complete and invalid. Syntax directed editors typically provide little explanation of why a particular design manipulation is illegal (missing C2), and they do not distinguish among good and poor legal programs (missing C6).

4. Organizational memories. Organizational memory systems, such as AnswerGarden [32], can contain heuristic knowledge that differentiates good designs from poor ones. However, the designer must proactively search for this knowledge; it is not automatically delivered at the time that it is needed. A designer would have to suspect that an error has been committed before they would have any reason to check the organizational memory. Furthermore, the designer would have to understand the design guidelines in the organizational memory and correctly apply them to his design. Organizational memory systems provide weak or no support for capabilities C1, C3, C4, and C8.

5. Expert systems. Critiquing systems may be implemented with expert system techniques, but the traditional expert system user interface is not a critiquing system. Traditional expert systems take a problem specification as input, apply domain-specific heuristics to solve the problem, and output the solution. It is the designer’s responsibility to accept or decline the computed solution. Most expert systems use heuristic rules and many deal with incomplete specifications and provide some explanation of their solution. Expert systems are not critiquing systems because they replace rather than aid the designer. Furthermore, since the designer is not directly engaged in decision making, it can be very difficult for the designer to recognize that part of the generated solution is incorrect, and override the expert system’s decisions to repair the failure. Traditional expert systems can succeed in domains where they can cost-effectively achieve a high degree of certainty. Critiquing systems are likely to succeed in more open-ended domains where it is impractical to embed complete knowledge, or where professional designers are unwilling to cede decision-making. Section 4 explains that some of the first critiquing systems were developed to replace medical expert systems.

Many of these other types of systems can be made to fit the definition of critiquing systems by adding new features and integrating them with design tools. For example, the Unix “lint” utility could be the basis of a critiquing system if it were integrated into a text editor, applied automatically as the programmer edits the program, and if its feedback was better controlled and presented in a more organized form. Likewise, organizational memory systems could be made into critiquing systems by formally representing some of the situations under which a specific part of the memory should be retrieved, and then proactively delivering that knowledge to designers when the situation is detected.

1.5. Organization of the Paper

Section 2 defines a five phase design improvement process called the ADAIR critiquing process. Section 3 covers the first level of the survey by describing several approaches to intelligent user interfaces and evaluating them with respect to the criteria for design support introduced in Section 2. Section 4 covers the second level of the survey by narrowing the discussion to critiquing systems and applying the same design support evaluation criteria. Section 5 concludes the paper by summarizing the state of the art in critiquing systems and identifying opportunities for future research.

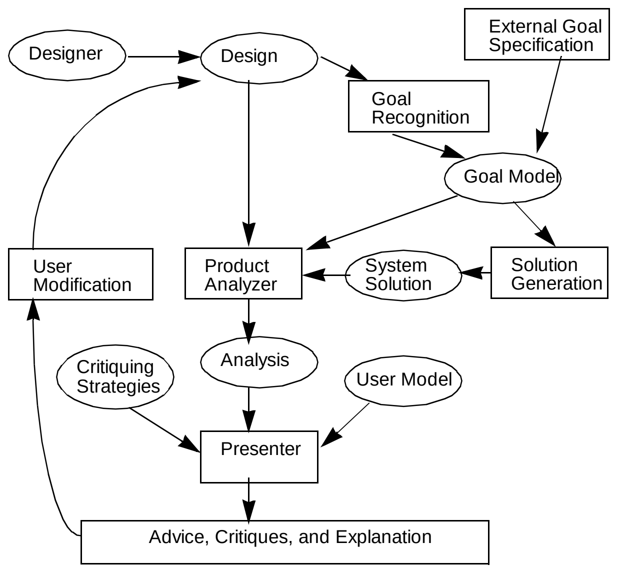
2. The ADAIR Critiquing Process

It is common for a survey to define a set of evaluation criteria. These criteria often relate to one another rather than being completely orthogonal. Our main contribution is the identification of a relationship among the evaluation criteria that serves to better organize our evaluation and guides our recommendations for future research.

We structure our evaluation criteria around the phases of a process (Figure 7) to highlight a temporal dependency among the areas of design support provided by the approaches and systems reviewed. For example, a critiquing system cannot provide strong support for detecting problems if does not first activate appropriate critics, and it cannot advise the designer of problems if has not first detected the problems.

The definitions of critiquing systems given in Table 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. This process is repeated for each design problem detected and multiple instances of this process may be active concurrently. 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 design feedback to the designer’s current task, and support for guiding or making design improvements. Below we describe five more sophisticated processes that include additional phases and then we describe our own critiquing process model and give an example usage scenario.

Figure 3: Critiquing Process Described by Fischer et al. (adapted from [49])



2.1. Previous Work on Critiquing Processes

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. When using Janus, the designer enters a design into the system, but also fills out a form to specify the design goals. For example, if the designer specifies that he is working on a kitchen for use by a person living alone, then critics that support family kitchens are deactivated. If the designer later specifies that the resale value of the house is very important, critics relevant to resale value will be activated. The observation that designs and specifications co-evolve is central to the Janus family [53, 60]. Figure 3 shows a critiquing process proposed in 1991 by Fischer et al.

Sumner, Bonnardel, and Kallak [69] define the critiquing process shown in Figure 4. In this process the critiquing system performs 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. Specifically, in the final step, the designer is expected to modify the design to resolve problems or modify the rationale related to the design decision at hand.

Gertner uses the critiquing process in Figure 5 to describe TraumaTIQ, a medical critiquing system that emphasizes real-time interactions with a doctor during emergency medical treatment [70]. Like Janus, this system deals explicitly with the user's goals. However, in TraumaTIQ, the goals are inferred from the user's actions rather than stated directly.

Silverman's survey of critiquing systems [4] describes the critiquing process shown in Figure 6. Much like the detect-advise process model, this process model centers on the detection of errors and the generation of textual feedback to the designer. Silverman adds discussion of influencing and repairs. Influencers are graphical annotations on the design that help the designer avoid mistakes from the start. For example, if a new design element cannot be placed near an existing one, the "off-limits" area is highlighted in red before the designer attempts the placement. Influencers act before and during the time when the designer works on a particular design subtask.

As described below, we have attempted to merge and extend these process models to clarify our own understanding of the role of critics and document the functionality of our own critiquing system. The resulting process model is the described in the next subsection.

2.2. 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 7 as a linear sequence, however some phases may be skipped in certain situations, and multiple instances of the process may

Figure 4: Critiquing Process Described by Sumner, Bonnardel, and Kallak (from [69])

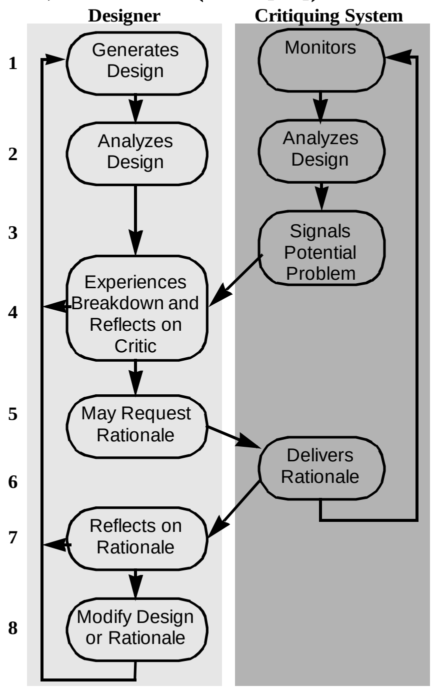


Figure 5: TraumaTIQ's Critiquing Process (from [70])

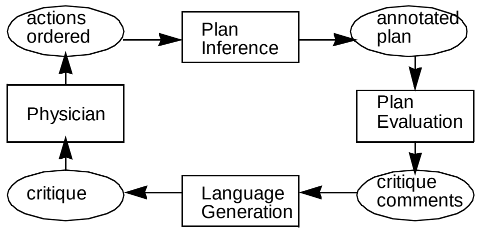
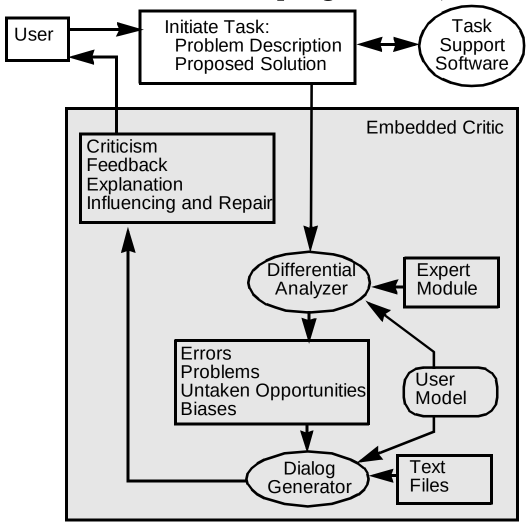


Figure 6: Silverman's Critiquing Process (from [4])



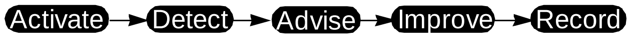
be concurrently active at any given time. The ADAIR phases are not necessarily contiguous: other work often intervenes.

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 paper 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.

We believe ADAIR to be a complete model in the sense that it contains elements that can be mapped to all of the elements of previous critiquing processes. ADAIR contains the detect-advise process as its core. The specification sheets and critiquing perspectives of Janus map to ADAIR's Activate phase. Steps 1 through 7 in Figure 4 map to the Detect and Advise phases of ADAIR, and step 8 maps to the Improve phase. The activities in Gertner's process model (Figure 5) map to ADAIR's Activate, Detect, and Advise phases. The majority of the boxes in Figure 6 map to the Detect and Advise phases. The word "repair" in Figure 6 maps to part of ADAIR's Improve phase. Furthermore, we believe that the ADAIR critiquing process model is complete in the sense that it has been useful in characterizing every critiquing system that we have found in the literature.

For clarity, we describe the process in terms of critics, but in Section 3 we demonstrate that it is also applicable to coaches, tutors, agents, and other forms of intelligent user interfaces.

Figure 7: The ADAIR Critiquing Process



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. A critiquing system may have hundreds of critics that address a wide range of possible design issues. Only some of these critics will be appropriate to the designer's current task. Activating all critics would unnecessarily waste available machine resources and would de-emphasize those critics that are truly timely and relevant. Some of the reviewed systems rely on the designer to explicitly specify the critics that should be activated, other systems attempt to infer relevance and timeliness from the state of the design and various user models. 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. Many of the approaches and systems reviewed assume that detection is done using a rule-base. Increasing support for detection by adding new types of detection mechanisms broadens the range of advice that the system can offer.

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; the presence of this phase differentiates critiquing systems from automated problem solving systems. 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). Two events must occur for the designer to benefit from design feedback: the feedback must be presented and the designer must understand it. 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 that have fewer errors and better conformance to stylistic conventions. Realizing these benefits requires effective means for designers to manage feedback and careful phrasing of problem descriptions and suggestions. Increasing support for the presentation of advice helps designers make more effective use of the feedback that they are given.

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, or a change might be made to some aspect of the design that is outside the representation used by the design tool. Design support systems can aid designers in making improvements by providing suggestions for improvements or corrective automations that fix the identified problem (semi-)automatically. Increasing support for the Improve phase reduces the procedural knowledge and effort needed for designers to make improvements and increases the likelihood that designers will follow through on the advice of critics.

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. Often a design change that fixes one problem causes another problem. When a new problem is identified designers often need to know why the problem arose, or they risk reintroducing problems that had previously been resolved. 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 [69]. Furthermore, feedback items can also be resolved based on action taken outside of the design tool and these actions must be recorded to avoid presenting the feedback again. For example, one critic might warn that using a "beta" software component in a design requires a commitment from the quality assurance manager; once that commitment is obtained the issue is resolved and the feedback should no longer be presented, even though the design artifact is still in the state where the problem was detected. Increasing support for the Record phase provides designers with more design context on which to base decisions and improves the usefulness of feedback over time. Furthermore, recording the outcomes of criticism and collecting metrics on the impact of individual critics is key to effective maintenance of critiquing systems.

The Activate and Detect phases are similar in that they both apply predicates to the design artifact and take action if the predicate is satisfied: in the Activate phase the action is to activate critics and in the Detect phase the action is to generate feedback. It would be possible to simplify the process model by combining these phases, however we have separated them because they can occur at different times and because they use algorithms with different reuse characteristics. The results of activation predicates typically can only change when part of the user model changes, rather than in reaction to design changes which occur more frequently. The strategies used to activate critics primarily compare attributes of the critics to attributes of the design process, user model, and goals model. These strategies can be domain specific, but they are likely to be based on reusable, domain-independent critiquing strategies. In contrast, the algorithms and predicates used in the detection phase consider attributes of the design itself, are much more domain specific, and are much less reusable across critiquing systems.

1. "Foil" is a playwright's term for a minor character who's main purpose is to help define a major character through dialog.

In the ADAIR critiquing process generation of feedback is done in the Detect phase, rather than in a separate Generate phase. We choose to model these two activities as one phase because our experience in building a family of critiquing systems has shown that much of the work of detecting problems is in gathering specific pieces of information from the design representation and computing intermediate results. This same information is needed during the generation of feedback. Some of the critiquing systems described in Section 4 have a dialog generation component in their architectures, however this text generation is done based on a data structure that is produced during problem detection. The presentation of this feedback data structure as English text falls under the Advise phase.

2.3. ADAIR Scenario

In this scenario we illustrate the ADAIR process by describing how critiquing features in Argo/C2 support each phase.[2] Specifically, the software architect using Argo/C2 is prompted to improve a decision about component selection. To emphasize the fact that the architect maintains control, we structure the scenario as an initial situation and a branching sequence of steps leading to six alternative conclusions. The relationship between these steps is summarized in Figure 11.

Argo/C2 is a software architecture design tool with critics, a "to do" list, and a process model and goal model that are used to keep criticism timely and relevant to the design task at hand [87]. Argo's *decision model* is derived from the state of its process model and consists of a list of domain-oriented decision types, each of which is prioritized (Figure 9). A C2-style software architecture specifies a set of software components, the messages that each sends and receives, and the communication relationships between components.

Step 1. The software architect is working on designing an HTML editing tool (Figure 8). White boxes in the diagram represent software components, black boxes represent buses that broadcast messages from one set of components to another, and arcs represent communication relationships. The architecture is in an early stage of completion: some components are already chosen, configured, and connected, although many other aspects of the functionality of the system have not yet been addressed.

Step 2: Activate. The user and goal models indicate that the architect is willing to consider alternative component choices, i.e., alternative critics are active. The state of the decision model is shown in Figure 9. Argo activates all critics that are timely and relevant, including an alternative component selection critic.

2. Not all of the functionality used in this scenario is implemented in Argo/C2. The infrastructure for providing wizards has been implemented in Argo/UML but has not be transitioned to the earlier Argo/C2 tool. Rather than use two partial scenarios with examples from two different domains, we explain all the features in the context of a single example.

Figure 8: Argo/C2 Modeling an HTML Editor

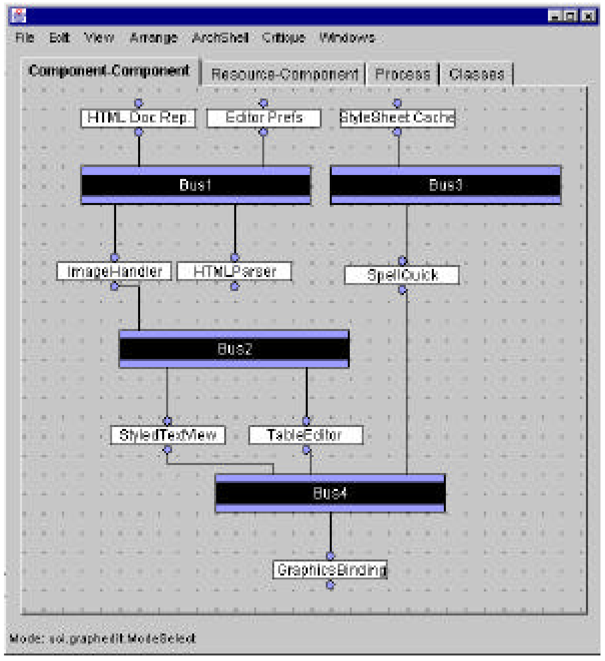
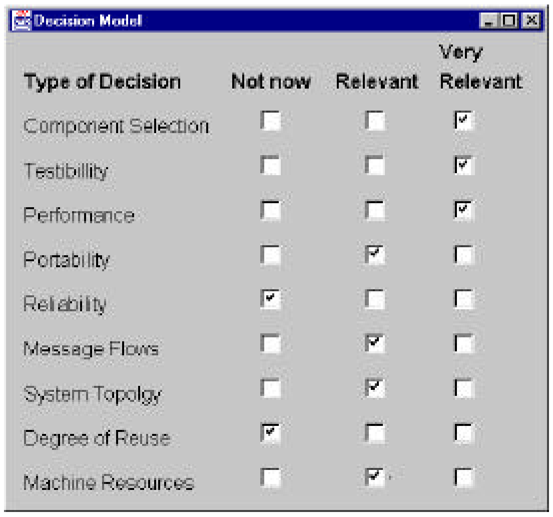


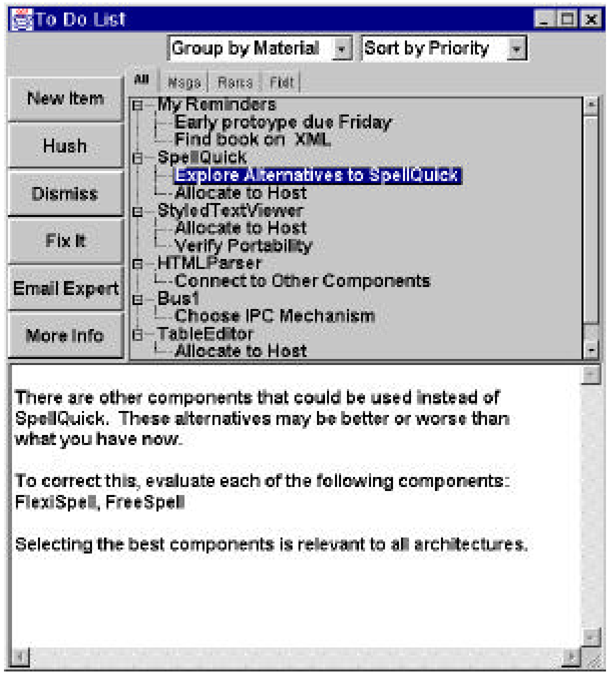
Figure 9: Argo/C2's Decision Model



Decision Model

Type of Decision	Not now	Relevant	Very Relevant
Component Selection			✓
Testibility			✓
Performance			✓
Portability		✓	
Reliability	✓		
Message Flows		✓	
System Topolgy		✓	
Degree of Reuse	✓		
Machine Resources		✓	

Figure 10: Argo/C2's "To Do" List User Interface



- My Reminders
  - Early protoype due Friday
  - Find book on XML
- SpellQuick
  - Explore Alternatives to SpellQuick
  - Allocate to Host
- StyledTextViewer
  - Allocate to Host
  - Verify Portability
- HTMLParser
  - Connect to Other Components
- Bus1
  - Choose IPC Mechanism
- TableEditor
  - Allocate to Host

There are other components that could be used instead of SpellQuick. These alternatives may be better or worse than what you have now.

To correct this, evaluate each of the following components: FlexiSpell, FreeSpell

Selecting the best components is relevant to all architectures.

Step 3: Detect. Argo/C2 applies all active critics. An alternative critic detects that the SpellQuick spell-checking component has the same interface as two other library components: FlexiSpell and FreeSpell. This critic determines component substitutability by finding equivalent messages in the components' interfaces. It represents a component's interface as a set of messages that can be sent or received. Messages are considered equivalent if they have the same name and the same number and types of arguments. If all messages used in the current component have equivalents in another, then the critic will suggest considering the other component. Since message semantics are not considered, the architect will need to apply his or her own knowledge. The critic constructs a feedback item data structure, and the fact that the feedback was produced is recorded in the design history.

Step 4: Advise. Argo/C2's "to do" list displays the new feedback item as shown in Figure 10. The architect continues working on the architecture uninterrupted. Eventually he or she reaches a reflective point and looks at outstanding design criticism to evaluate the state of the design and decide what should be done next. While browsing the list of "to do" items, the architect sees the headline of the alternative critic's feedback.

Step 5: Advise. The architect reads the problem description and realizes that choosing SpellQuick was fairly arbitrary. Here the architect's understanding of the state of the architecture is improving, even though the design artifact has not changed yet.

Step 6: Improve. In this case, the architect decides to follow the critic's advice and manually performs several manipulations to achieve the desired state. The architect looks at the alternatives and chooses FlexiSpell. He or she deletes SpellQuick; this also deletes any relationships between SpellQuick and other components in the architecture. The architect then inserts FlexiSpell, parameterizes it, and connects it to the same components. During this process several other critics may raise or withdraw their criticism as the state of the partially specified architecture changes.

Step 7: Record. Each of the actions performed by the architect is individually added to the design history. Without explicit rationale from the architect, the alternative critic that fired initially can, at best, withdraw its criticism because one of its suggested alternatives was considered. The same critic is likely to fire again with the criticism that

Figure 11: Forking Timeline of Scenario Steps

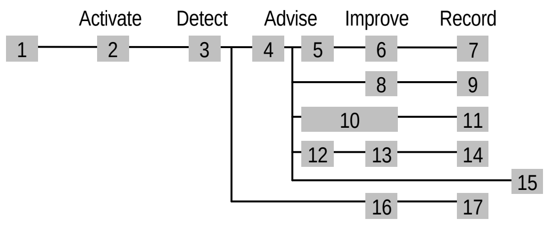
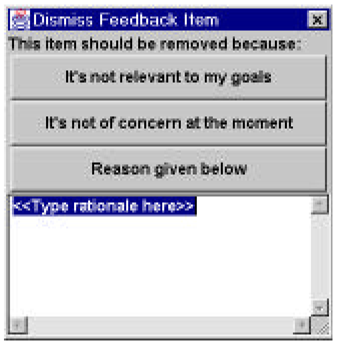


Figure 12: Argo/C2's Feedback Dismissal Dialog


Dismiss Feedback Item
This item should be removed because:
It's not relevant to my goals
It's not of concern at the moment
Reason given below
<<Type rationale here>>

there is still another alternative component that has not been considered.

Step 8: Improve. We now move to the second of the six branches shown in Figure 11. In this branch of the scenario the architect looks at the suggested alternative components and decides to keep the current component. This improves the architect's understanding of and confidence in the current design, but it does not modify the design artifact itself.

Step 9: Record. The architect presses the "Dismiss" button to indicate that the matter should be considered resolved. Argo prompts the architect to enter rationale. He or she optionally types a description of why the suggested alternatives were not selected and presses "Reason Given Below" (Figure 12). A new record is added to the design history with the annotation that it was explicitly resolved by the architect along with the rationale. This particular item will not be posted on the "to do" list again.

Step 10: Advise/Improve. In this branch of the scenario we assume that the feedback item references a wizard that can be used to swap an alternative component for the current component. The architect activates the wizard by pressing "Fix It" and works through a sequence of dialog boxes. At each step the wizard explains the step and prompts the architect to make or confirm specific design decisions, such as choosing FlexiSpell as the alternative components to use. The wizard offers to automatically parameterize FlexiSpell with some of the same parameter values used for SpellQuick and then offers to connect it to the same surrounding components. As with manual changes, other critics may raise or withdraw criticism based on the state of the design.

Step 11: Record. Wizards are task-specific user interfaces. Here, the task is the resolution of a specific feedback item. The multiple changes made during the wizard interaction can be added to the design history as a logical group and marked as the resolution to the original criticism.

Step 12: Advise. In this branch of the scenario the architect reads the design feedback headline and possibly the problem description and decides not to address issues of component selection right now. He or she may simply read past this feedback item and move on to other items. This would leave any other feedback about alternative component choices in the "to do" list.

Step 13: Improve. If the architect wants to focus the "to do" list on other issues he or she may update the decision model to indicate that component choice issues are not of interest at the moment. This action improves the state of the user model. This causes all alternative component choice critics to be deactivated and their criticism withdrawn.

Step 14: Record. The fact that feedback items were withdrawn because of a change in the decision model is recorded in the design history. The new record consists of the new state of the changed part of the decision model and references to the creation records of all feedback items that were withdrawn.

Step 15. In this branch the architect decides that the criticism is currently not of interest, but also declines to take the time to update the user model. Pressing the "Hush" button temporarily disables the critic and hides all "to do" items raised by that critic. After several minutes the critic is automatically re-enabled and its outstanding feedback items reappear on the "to do" list. Hushing is only a feedback management operation, not a step in the critiquing process, and no feedback items are being raised or resolved.

Step 16: Improve. In the final scenario branch the architect never looks at the alternative critic's feedback. However, in the normal course of working on the architecture, he or she decides to replace SpellQuick with a new custom component that does not have the same interface as any component in the library. The alternative critic's feedback is automatically withdrawn as soon as Argo/C2 determines that it is no longer valid.

Step 17: Record. Argo/C2 records each change made to the architecture as the architect works. At some point Argo/C2 determines that the alternative critic's feedback is no longer valid. Argo/C2 then searches its recent design history for the operations that affected the items offenders. One of those operation records is selected as the resolution to the problem and annotated with a reference to the item's creation record.

In sum, Argo/C2 provides a variety of levels of support for improvement and recording design changes. The architect receives feedback about possible improvement opportunities and is free to act on them according to his or her own initiative. If the architect chooses to make a suggested design change, a wizard may be provided to guide and partially automate the task.

2.4. Relation between ADAIR and the Broader Software Process

The design process is only one part of a complete software development process. ADAIR focuses on critiquing and design improvement and does not address the usual concerns of software process modeling, such as project planning, work-flow, or scheduling. Instead, ADAIR provides a simple and useful description of a small, recurring process fragment.

Even though design is only one part of the overall software development process, it is an important one. Many features of the product being built are determined during design and much of the rest of the product development process is affected by design. For example, a complex design calls for much more implementation effort and possibly a larger development organization than would be needed for a simple design; also, a design that anticipates future adaptations correctly is much easier to maintain than one that assumes that no requirements will change.

The phases of ADAIR do not correspond to phases or stakeholders in the broader software process, but critics may supply knowledge that does. One type of design critic represents the interests of a certain project stakeholder, for example a quality assurance manager. Other critics may be used as conditions on progress through the development progress, for example in a waterfall process all criticism related to incompleteness in the design must be resolved before implementation can begin. Furthermore, the state of the broader software process may be used to determine which critics should be active, for example one set of critics is relevant during early conceptual design while another set is relevant during low-level design that is oriented toward a specific implementation language.

3. Comparison of Approaches to Intelligent User Interfaces

This section addresses the first, more general, level of the survey. Here we situate the critiquing approach in the space of intelligent user interface approaches. Intelligent user interfaces are systems that structure or modify their interface elements based on implicit or explicit models of the user or the user's task [15]. Table 3 characterizes several intelligent user interface approaches with respect to the phases of the ADAIR process shown in Figure 7. The rating in each cell of the table is a subjective score from zero to three points based on how well each approach supports each ADAIR phase. We discuss each of these approaches below.

3.1. Tutoring Systems

Tutoring systems, such as Geometry Tutor [16] or the legal argumentation tutor of Ashley and Aleven [17], are used in classrooms or other training environments by students trying to master the material in a lesson. A typical session with a tutor starts with the student reading new lesson material that is presented by the system. Then students solve problems that test their understanding of the material. The tutor tracks the student's progress, provides explanations that reinforce topic areas that the student might not fully understand, and selects new problems to further probe the student's understanding.

Tutoring systems use detailed user models to implicitly activate tutoring rules and to customize the advice given to students. Tutoring systems evaluate a student's work and provide feedback to help the student achieve specific lesson objectives. Tutoring systems do not improve the artifact themselves, that is the task of the student. However, they are constructive in that they give procedural hints to students that are having difficulties with the assigned task. Recording may be done for the purpose of grading, but the most direct use of recording is to update and refine the user model as the student makes progress.

Tutors differ from critics in that tutors are intended to help students with artificial educational exercises, whereas critics aid professional designers in actual work situation. Tutors must be programmed with detailed knowledge of the objectives of the exercise and overall lesson plan. Furthermore, they must possess or be able to generate, a complete solution. This requirement limits application of tutoring systems to domains that are very well understood and where the number of users justifies the cost of authoring. In contrast, critics must function with partial knowledge as the problem and solution co-evolve [52].

3.2. Critiquing Systems

Critiquing systems are design tools intended for use by professional designers in their normal practice, however they may also be useful in classroom or "on the job" training. A typical session with a critiquing system consists primarily of the designer using the design tool to construct or modify

Table 3: Summary Comparison of Intelligent User Interface Approaches

Intelligent User Interface Approach	ADAIR Critiquing Process Phase				
	Activate	Detect	Advise	Improve	Record
Tutoring systems	★★★	★★★	★★★	★★	★
Critiquing systems	★★★	★★	★★	★	★
Coaching systems		★★	★★	★★	★
Wizards		★	★★	★★★	★
Automatic Word Correction		★★★	★★	★★★	
User Interface Agents	★★★	★★	★★	★★	★★

a design. Only when specific improvement opportunities, for example design errors, are detected do critics provide feedback.

Critiquing systems can choose which critics should be activated based on a specification of design goals or the design process. Critics detect problems in the proposed solution and provide advice to the designer. The advice provided by critics usually includes an explanation of the problem and suggestions for improvement. Most critiquing systems do not play an active role in the improvement of the design or the recording of design activities. Detailed examples of critiquing systems are the focus of this survey and provided in Section 4.

3.3. Coaching Systems

Coaching systems, such as COACH [20] and Lumire (Microsoft Office Assistant) [18, 19], assist users by watching their actions and suggesting help topics. Coaching systems can be deployed in desktop applications and used by a broad class of users. Users proceed to use the tool as normal, until the system detects a specific assistance opportunity, for example misusage of one of the application's features. At that time the coach suggests a better way to use the tool.

COACH is a dedicated system that is always activated. Lumire is also always active; although it may be hidden, in which case it only reappears to provide high priority feedback. COACH detects assistance opportunities with rules and an adaptive user model, while Lumire uses a Bayesian network. Feedback primarily consists of a set of help topics or suggested actions to perform next. Improvement cannot be made automatically because the objective is to cause the user to learn and change their future behavior. However, the advice given is rich in procedural knowledge. COACH records interactions by updating its user model.

The primary difference between this approach and the critiquing approach is that coaches assist in tool usage while critics assist in design decisions. Furthermore, feedback presentation is more intrusive in coaching systems than in critiquing systems because the goal of teaching tool usage demands immediate feedback when breakdowns are detected. Since the two approaches supply complementary types of knowledge, it seems reasonable for a given system to include both kinds of support. Furthermore, both coaches and critics are intended for day-to-day professional use.

3.4. Wizards

Wizards, such as TaskGuide [21] and those found in Microsoft Office [22], are user interface dialogs that guide the user through a sequence of steps or decisions. Wizards typically appear when an application is launched or they are explicitly invoked. Each step of the wizard is shown in dialog box that provides or requests information and allows the user to move on to the next step. Most wizards are modal: the user cannot pause the wizard to manually change the design and then continue the wizard. Usually the wizard performs an automation once all needed information has been entered and then disappears until it is invoked again.

Wizards are not concurrent processes and need not be activated by the system. Detection of assistance opportunities is the user's responsibility. The primary strength of wizards is their procedural knowledge that allows them to explain and perform the steps needed for specific design improvements. For the most part, recording in wizards is limited to the ability to backtrack through steps to revise information; however, some wizards record decisions in the artifact being designed.

Wizards differ from critics in the type of knowledge that they provide to the designer and that they require from the designer. Wizards provide procedural knowledge about how to make a complex change to a design. While wizards are gathering information needed for the change, they may prompt the designer with explanations of some of the issues involved in the change. Wizards typically require designers to know that the supported design change would improve the design, that the tool includes a wizard to support that change, and how to invoke the wizard. In contrast, critiquing systems provide much stronger support for automatic activation and detection of assistance opportunities, but little or no support for carrying out design improvements.

3.5. Automatic Word Correction

We consider two types of intelligent user interface for automatic word correction: dynamic revision and spell checking. Both are features commonly found in word processors such as Microsoft Word and text editors such as Emacs. Interlisp's DWIM (Do What I Mean) feature is an example of dynamic word correction applied in a programming environment [26]. Dynamic revision systems watch what users type and automatically replace some strings with others. For example, in Microsoft Word "teh" is replaced with "the," and "HEllo" is replaced with "Hello." Spelling checkers are also familiar to users of word processors. Early spelling checkers simply produced a list of the words in a document that were not found in a dictionary; the actual correction of the errors was left entirely to the user. Recent spelling checkers have sophisticated rule bases, natural language models, and user interfaces that automatically highlight probable spelling errors as they are typed and offer suggested corrections.

Simple dictionary look-up certainly does not require intelligence. However, the best spelling checkers suggest replacement words based on heuristic rules that consider common causes of spelling errors, knowledge of word frequency in business writing, and the relationship of the suspect word to the surrounding text. Kukich's survey of techniques for word correction covers the historical development of these approaches and their application to word processing, optical character recognition (OCR), and speech recognition [24]. For example, Grudin found that many spelling errors resulted from users accidentally pressing two keys at once [25]. One heuristic rule often used in offering suggestions is that misspelled words usually have the first letter correct. The substitution tables used in dynamic revision are also based on knowledge of capitalization rules and word frequencies in natural language.

Most dynamic revision and spell checking systems limit user modeling to customizable dictionaries and require explicit activation. However, they are strong on detection and improvement in the limited domain that they address. Dynamic revision is effective at making improvement, however since it does not advise the user or ask for confirmation, these improvements are sometimes undesired. In fact, this word correction feature is most noticeable when it has performed an undesired modification. Recent spell checkers, in contrast, do visually advise the user and ask for confirmation before making a change (e.g., a red, wavy underline). This takes extra effort on the part of the user, but gives a sense of control. New words can be recorded in the dictionary when a valid word is encountered that is outside the knowledge of the system.

Automatic word correction systems emphasize detection and improvement, whereas critics focus on detecting potential problems and advising the designer. Spelling errors occur at a single point in the document and can be explained trivially. In contrast, the broader class of design errors that critics can detect may involve design elements from different parts of the design and explaining the errors may require the tool to convey knowledge of the problem and solution domains. Word correction systems do not need sophisticated support for the Record phase of the ADAIR process, in part, because spelling decisions are not interrelated, i.e., other writing decisions are not likely to depend on the spelling of individual words.

3.6. User Interface Agents

User interface agents assist users primarily by continuously retrieving or filtering information. For example, interface agents can categorize and prioritize one's email, filter news streams, or recommend entertainment [27, 28, 29]. Agents are directable: the user specifies a set of goals for the agent, and the agent tries to fulfill those goals over a period of time, with a minimum of further interaction with the user. In some cases, the goals can be inferred from the user's own actions and reactions rather than stated explicitly.

Most user interface agent systems built to date have few supported goals and thus keep all agents active. Agents detect improvement opportunities based on models of the user and goals. Once an improvement opportunity is detected, agents may either advise the user of the opportunity or, if confidence in the goal model is high, take immediate improvement actions [27]. Agents do not record histories, however they do learn rules and refine their user and goal models by analyzing interactions with users and other agents.

When a user specifies the goals for a user interface agent, they know of the existence of the agent and request that it do something on their behalf. In contrast, designers may not know of the existence of a particular critic until it has performed its analysis task and delivered feedback. Critics are implicitly activated based, in part, on the designer's own stated goals for the design. Ideally, user interface agents learn from their interactions with the user and gain more confidence in their own ability to carry out the user's wishes. Learning agents are most useful when personalized to a particular user and are most reliable when the user performs repetitive actions. In contrast, knowledge-based approaches (such as critiquing, coaching, and tutoring) are more suitable for design support where the designer may lack needed knowledge.

4. Comparison of Critiquing Systems

This section addresses the second, more specific, level of the survey. Our survey of critiquing systems literature revealed over fifty articles describing thirteen different critiquing systems. In Table 4 we characterize 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. The scoring rules are presented in Table 5. We do not attempt to evaluate the knowledge embedded in each system, instead we focus on how that knowledge is applied, presented, and ultimately used to improve the design. Many of these critiquing systems are integrated into design tools, while others are stand-alone tools.

Each system uses comparative critiquing, analytic critiquing, or both. *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 this respect, comparative critiquing systems are similar to tutoring systems and suffer from the same authoring costs and domain size limitations. Pointing out differences can lead designers to make their design more like the generated design or cause them to re-examine their reasons for making different decisions. Comparative critiques can be confusing when multiple good solutions exist that are very different from each other. In contrast, *analytic critiquing* uses rules to detect assistance opportunities, such as problems in the design. This aids designers by guiding them *away* from recognized problems rather than guiding them *to* known solutions. In general, analytic critics can be built incrementally and applied throughout the design process. Substantial domain knowledge is needed to implement analytic critics, but they need not have access to a generated solution. This allows analytic critics to be applied to a broader range of domains. Avoidance of detectable errors is necessary but not sufficient for good design.

We can categorize critics based on the type of domain knowledge that they provide. *Correctness critics* detect syntactic and semantic flaws. *Completeness critics* remind the designer to complete design tasks. *Consistency* critics point out contradictions within the design. *Optimization critics* suggest better values for design parameters. *Alternative critics* prompt the architect to consider alternatives to a given design decision. *Evolvability critics* address 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 critic categories are descriptive rather than definitive. Some critics may belong to multiple categories, and new categories may be defined, as appropriate for a given application domain. Table 6 shows some examples of Argo/C2's architecture critics and the type of knowledge that they can provide.

Fischer offers the following critic classification dimensions: active vs. passive, reactive vs. proactive, positive vs. negative, global vs. local [45]. 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.

In the summary comparison of critiquing systems (Table 4) we score critiquing systems based on the features they implement. We have not attempted to define a complete set of possible features in a top-down fashion, instead we have used a bottom up approach by looking at the features that have been implemented in the systems reviewed and scoring them on a scale of zero to three points. The rubric is shown in Table 5.

In the following subsections we discuss each of these critiquing systems in roughly chronological order. The date of first publication on each system is shown in Table 4.

4.1. ONCOCIN

In 1980, Teach and Shortliffe conducted a survey of doctors' attitudes regarding computer based clinical consultation systems [35]. 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. Previous medical consulting systems such as MYCIN and MV are reviewed in [34]. 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

Table 4: Summary Comparison of Critiquing Systems

System	Year of First Publication	ADAIR Critiquing Process Phase					Provides All Capabilities
		Activate	Detect	Advise	Improve	Record	
ONCOCIN	1981		Comparative	★			✓
ATTENDING family	1983		Both	★★	★		✓
Janus family	1989	★★	Analytic	★★	★	★	✓
Framer	1990	★★	Analytic	★★	★★		✓
KRI/AG	1992		Analytic	★★	★		
CLEER	1992		Analytic	★			✓
VDDE	1993	★	Analytic	★		★	✓
TraumaTIQ	1993	★★	Comparative	★★			✓
AIDA	1995	★	Both	★			✓
UIDA	1995	★	Both	★	★		
SEDAR	1995	★★★	Analytic	★★	★★		✓
Argo family	1996	★★★	Analytic	★★★	★★★	★★	✓
ICADS	1997	★	Analytic	★	★		

override the systems advice when they did not agree with the generated treatment plan [36]. 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.

ONCOCIN generated exactly one solution and offered exactly one piece of advice for each patient, thus there was no need for an activation strategy. Detection consisted of simply comparing dosage levels and some other aspects of the treatment plan. The criticism produced was more of a directed explanation of the system's solution than an actual critique of the doctor's treatment plan. For each significant difference between the doctor's planned dosage and the system's plan, ONCOCIN reported the difference and offered an explanation for ONCOCIN's decision. A series of prompts allowed doctors to traverse ONCOCIN's goal hierarchy to view explanations of intermediate steps in the system's reasoning. The system did not explicitly offer advice on how to improve the doctor's plan; the implication was that the doctor's plan could be improved by making it more like ONCOCIN's. ONCOCIN recorded a history of treatment administered, but it did not record any proposed plans that were later changed or the impact of its critiques.

Table 5: Critiquing System Rubric

Activate. Each system earns one point in the Activate column for each of the following features: • having any activation strategy, • offering multiple activation strategies, • employing a goal model for activation, or • employing a process model for activation.
Detect. Since we are not evaluating the knowledge content of critiquing systems, they are not given points for their detection ability. Instead, we indicate the type of detection mechanisms employed.
Advise. Critiquing systems earn points in the Advise column by: • providing explanations of problems detected, • allowing the designer to conveniently browse feedback, • graphically indicating feedback in the design document, • providing supporting design context, e.g., the email address of a person to contact for more information, or • generating context sensitive, natural language text.
Improve. One point for the Improve column is given to each critiquing systems for each of the following: • including improvement instructions in the feedback, • proving graphical cues that indicate what the designer should do to make improvements, or • providing automation to (semi-)automatically fix identified problems.
Record. Points for support in the Record column are earned by: • updating user models, design process models, or design goals models over time, • supporting the designer in entering design rationale, and • automatically recording how criticism was resolved.

The desire for a human-like user interfaces resulted in the use of natural language for presenting advice. Emphasis on natural language interfaces is seen in many medical systems, including all of the medical critiquing systems reviewed. In contrast, all of the non-medical critiquing systems reviewed use graphical indications, constant strings, or simple textual templates; none of them use sophisticated natural language generation.

4.2. The ATTENDING Family

At about the same time that ONCOCIN was being developed at Stanford, Miller was developing the ATTENDING system at Yale. The ATTENDING paper appeared months before the ONCOCIN paper, and Langlotz and Shortliffe cite Miller's work. However, we place ONCOCIN earlier in the time-line of critiquing systems because the sequence of publications that lead to ONCOCIN was begun before the ATTENDING publication and ONCOCIN provided a more primitive form of critiquing than did ATTENDING.

Like ONCOCIN, much of the emphasis of ATTENDING was on the avoidance 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" [39].

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. ATTENDING can function as a design critic or as a tutor. When used in actual practice it functions as a critic, although its limited knowledge made it ineffective in this role. Alternatively, it can function as a tutor, in which case, the system poses a problem from a library of lessons, then prompts the student to enter a proposed treatment plan and critiques it. 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. ATTENDING has mainly been used in training [39, 40].

Other members of the ATTENDING family are HT-ATTENDING [40], a critiquing system for management of hypertension; DxCON [41], a critiquing system for the radiological workup done in obstructive jaundice cases; and, E-ATTENDING [42] a critiquing system shell for building similar critiquing systems.

A sample session with HT-ATTENDING is shown in Figure 13. The system's knowledge-base includes two treatment plans for managing hypertension with drugs (one is shown in Figure 14) and several rules about drug interactions and contra-indications of drugs. HT-ATTENDING performs comparative criticisms by determining the patient's current step in the treatment plan, and comparing the proposed treatment to the next step. In the example ATTENDING recognizes cholrthalidodone as a diuretic and infers that the patient has reached step one of the standard treatment plan; the proposed use of guanethidine does not match step two, so the system suggests a beta blocker or sympathetic blocker and warns against the use of a step four drug before step three drugs have been tried. HT-ATTENDING can also produce analytic criticism by applying its contra-indication rules. For example, the use of beta blockers with a patient who has had prior heart problems is contra-indicated.

None of the systems in the ATTENDING family have user models or goal models. All critics are always active. This approach to critic activation is reasonable for these systems because the number of critics is limited and all criticisms support the implicit goal of improving the patient's health. Detection is done by comparative and analytic rules. These systems do not generate a new solution. Instead, they are programmed with knowledge of standard treatment plans. Advice is presented in high-quality natural language text that is organized to present the most important criticisms first, it also includes references to further information on drugs and medical studies. However, the advice is organized by the system and cannot be browsed or managed by the user, making it unscalable. Weak support for improvement results from the constructive phrases used in text generation. The ATTENDING-family systems treat each case independently and do not prompt the user to enter a revised treatment plan, and thus cannot record whether their criticism was followed or ignored.

4.3. The Janus Family

The Janus family consists of several versions of a household kitchen design environment, named successively Crack, Janus, Hydra, and KID [44-60]. 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 [52]. Goal specification sheets prompt the designer to provide information through a structured set of choices, for exam-

Figure 13: A Sample Critique by HT-ATTENDING (from [40])

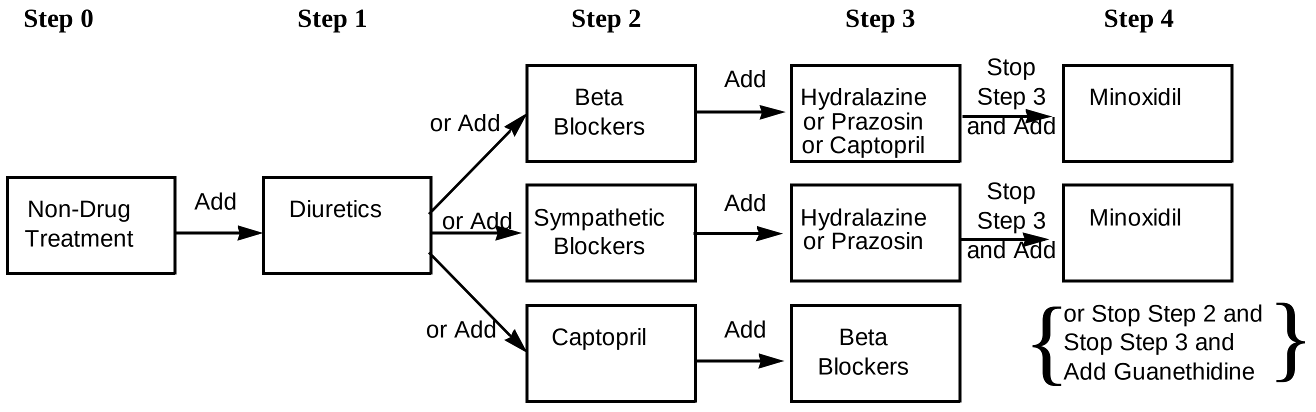
The patient is a 59-year-old woman with a history of congestive heart failure.
Current regimen: cholrthalidodone 50 mg qd, propranolol 160 mg bid
Proposed regimen: cholrthalidodone 50 mg qd, propranolol 160 mg bid, guanethidine 20 mg qd

In the United States, diuretics are recommended by most authorities as an initial or step one therapy. A thiazide diuretic or similar acting agent is the drug of choice for most patients. Cholrthalidodone has the advantage that it is long-acting and can be given once a day.

A beta blocker is, in general, our preferred step two agent. Clearly, in a patient with decompensated congestive heart failure, however, all beta blockers should be avoided in favor of alternatives, such as central sympathetic blockers or possibly captopril. On the other hand, in certain patients with well compensated congestive heart failure, a beta blocker may be useful. Here pindolol, which has intrinsic sympathomemetic activity (ISA) may well be the beta blocker of choice.

Guanethidine should be reserved for patients who have not responded to other agents since it is often poorly tolerated and difficult to administer. Thus for this patient, we would suggest first trying a step three drug, such as hydralazine, prazosin, or possibly captopril before using guanethidine which we reserve for use as a step four agent. When treatment with guanethidine is begun, we recommend that all other treatments, except diuretics be stopped. If you do use guanethidine, we would suggest starting treatment with a lower dose that you propose. We recommend that guanethidine be given once a day starting with a dose of 10 mg and increasing slowly until the desired therapeutic effect is achieved or intolerable toxicity occurs.

Figure 14: HT-ATTENDING's "Preferred" Treatment Approach (from [40])



ple, “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.

Janus’s goal model and critiquing perspectives are used to implicitly activate critics relevant to stated design goals. Janus automatically applies analytic critics that detect problems and produce brief descriptions with links into a hypermedia argumentation database. Janus’s critics primarily advise the designer of problems, while the argumentation describes both problems and possible solutions. In KID, feedback items are sorted by priority. Corrective automations that improve the design are not supported. Decisions can be recorded as additions to the argumentation database, but these are not interpreted by the system and do not represent the history of a single design.

4.4. Framer

The Framer design environment [61-64] supports user interface window layout done 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. Another panel describes what the designers should do during the current task and lists relevant commands. 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. Framer uses a checklist interface metaphor to group feedback according to suggested steps in a prespecified process model. This process model is also used to activate critics when timely. Analytic critics in Framer detect incompleteness in the design and incorrect use of user interface elements. Framer’s advice is presented in constructive terms that match the checklist paradigm; for example, “Add a menu bar” is presented instead of “Menu bar is missing.” Framer strengthens support for improvement by associating corrective automations with feedback items. For example, once the lack of a menu bar has been identified, a single button press will direct Framer to automatically add a menu bar. No support is provided for recording design decisions or the resolutions of critic’s feedback items.

4.5. KRI/AG

Like Framer and UIDA, KRI/AG [65] is intended to support designers of graphical user interfaces. The knowledge-base used in KRI/AG is taken from published guidelines on designing Motif user interfaces. The benefit of KRI/AG is that it provides this knowledge to designers in a more effective way than do printed style guides. Designers use KRI/AG by first entering their user interface layouts into the TeleUSE GUI builder, then they save their work in a file and invoke KRI/AG on it. Lowgren and Nordqvist do not discuss the presentation of advice to the designer, but they do give examples such as the following:

> “There is no Help menu in the menu bar. Every application should have a help menu. The recommended standard menus in the menu bar are File, Edit, View, Options, and Help, in that order. (Motif Style Guide p. 7-42)” [65]

> “None of the items in the menus of the menu bar have accelerators. It is a good idea to use accelerators for the most frequently used items. (Motif Style Guide, 3.3.2, 4.2.3, pp. 7-3, 7-4)” [65]

KRI/AG does not have a user or task model: once the designer invokes the system, all critics are applied. KRI/AG detects improvement opportunities by means of approximately 70 rules like the one shown in Figure 15. Advice from KRI/AG consists of simple textual templates that describe the problem in somewhat constructive terms. KRI/AG does not provide direct support for improving the design or recording the resolution of the criticism it raises.

Figure 15: Example KRI/AG Rule (from [65]).

```
Rule PopupMenuTitle in OSF_MOTIF Is
ForAll ?inst WhichIs Motif$XmOioulMenu;
   If
      Not (Class(MenuItem(?inst, 1) = “XmLabel”))
   Then
      MakeComment(“The popup “, ?inst.Name, “does not have a
         title. Every menu should have a unique title placed at the
         top. (Motif Style Guide 4.2.3)”);
End;
```

One result of the KRI/AG research is that many published user interface guidelines cannot be evaluated solely with respect to static information in the design. Often guidelines depend on assumptions about how an interface will be used. For example, the second feedback item quoted above can only be accurately detected if run-time monitoring is performed to determine which menu items are accessed most frequently.

KRI/AG does not satisfy all of our requirements for a critiquing system because it is not integrated into a design tool and it is not tightly integrated into the designer’s task. While KRI/AG may help designers better evaluate their designs, as does any useful analysis technique, it is not likely that designers will run the tool while they are still in the mental context of the decisions being critiqued.

4.6. 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 [66]. The placement of antennas on ships affects the performance of the antennas, the radar profile of the ship, and the function of other ship-board 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. CLEER primarily informs the designer using negative feedback. Positive feedback is also included, but Silverman and Mehzer report that it is largely ignored by designers. The system does not constructively aid the designer in improving the design or recording a design history.

Silverman and Mezher propose an enhanced version of CLEER that would use decision networks to add support for activation, advisement, and improvement [66]. Decision networks are discussed more in Section 5.2.

4.7. VDDE

The Voice Dialog Design Environment (VDDE) [67-69] is a design environment for voice dialog systems, for example the menu structure of a voice mail system. Good voice dialog design, like any form of user interface design, can improve productivity and reduce the opportunity for errors. Stylistic guidelines help to keep multiple voice dialog systems consistent with standards, for example, every menu should allow the user to exit by pressing the star (“*”) key. Furthermore, two voice dialog designs can be compared for consistency with each other, for example, if one system uses the “2” key for the “send” function, then the other system should also use “2” for “send.”

Designers use VDDE by placing menu nodes in a connected graph. Each menu node consists of several key-pad choices, and each choice is linked to an action or another menu. Designers may also specify voice prompts, voice recording actions, and numeric entry actions including valid ranges. In addition to visualizing the menu structure, designers can listen to a simulation of what a user of the system would hear. Critics 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. Critics detect incompleteness and style guideline violations. Analytic critics are used to detect incompleteness and style guideline violations, while comparative critiquing is used to evaluate consistency between two particular designs. VDDE’s advice is prioritized and consists of a brief description of a problem, a set of offending design elements, and a link into an argumentation database. Problem descriptions are generated from textual templates that are filled in with details of the problem situation. VDDE does not provide corrective automations or phrase its criticism in constructive terms. Support for recording design decisions is similar to that found in the Janus family.

Sumner, Bonnardel, and Kallak did an exploratory study of four professional voice dialog designers using VDDE [69]. All four designers were observed while they used 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 designer’s 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. Another interpretation is that criticism should be limited to clear-cut problems and phrased persuasively so that they are not so easily argued against. However, doing so would abandon much of the potential range of application of critiquing systems.

4.8. TraumaTIQ

TraumaTIQ is a stand-alone system that critiques plans for treatment of medical trauma cases, such as gunshot wounds [70-78]. One emphasis of TraumaTIQ is the time-critical nature of its domain: the patient’s health depends on getting the correct treatment, and on getting it as soon as possible.

A doctor or scribe nurse enters treatment orders into the system as they are performed. Recording of treatment is traditionally done on paper “trauma flow sheets.” 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.

TraumaTIQ determines which criticisms are relevant by analyzing the doctor’s treatment plan to infer its goals. Each action can support multiple goals in TraumaTIQ’s knowledge-base; a greedy algorithm is used to select those goals that are best supported. In comparison, ONCOCIN and ATTENDING address much narrower domains and support only a single goal. The system then generates its own treatment plan and detects assistance opportunities by comparing it to the doctor’s plan. Each difference results in a feedback record with an expected disutility value and is classified as tolerable, non-critical, or critical.

Advice is presented in the form of English text generated from the feedback records, textual templates, and a domain-specific language model. Each piece of advice contains a brief explanation and is sorted by urgency in the output window. Of all the systems reviewed, TraumaTIQ has the most sophisticated support for natural language text generation, and is the only one that can combine multiple, related feedback records into a single, brief piece of advice. The following example demonstrates the quality of the generated text and the system’s inference of a goal based on treatment actions: “Getting a chest x-ray seems premature at this point. There is not enough information to justify ruling out a simple left or right pneumothorax." Here, TraumaTIQ has been told that the doctor ordered a chest x-ray and can infer that the only reason for an x-ray is to check for air in the chest cavity. However, an x-ray is not sufficient to make a conclusion and other treatment issues are more pressing.

TraumaTIQ does not provide automated improvement of the treatment plan; the doctor must perform the suggested treatment and enter it into the system manually. Likewise, the system does not record the resolution of criticisms, however if the doctor does take action in response to the criticisms, those actions will be used to refine the system's goal model.

4.9. AIDA

The Antibody IDentification Assistant (AIDA) is a tool intended for use by medical laboratory technicians to categorize blood samples [78]. Despite the fact that misidentification can potentially result in the death of a patient, most practitioners learn this skill "on the job."

The antibody identification task is primarily a problem solving task: the technician must interpret a panel of tests done 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. We consider this task to be substantially similar to problem solving sub-tasks of the design tasks supported by other reviewed systems.

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.

AIDA assumes a simple task model that recognizes three types of events: (1) the designer makes any change to the design, (2) the designer switches from one screen to another, and (3) the designer declares that the task is complete. AIDA associates critics with each of these three event types and activates timely critics when each event occurs. Correctness critics are applied after each change to the design. Completeness critics are applied only when switching between screens. All other types of critics are applied when the designer declares the design complete. Analytic critics are used to detect implausible data in the design. Comparative critics are used to detect differences between the designer's solution and an automatically generated solution, for example, if the designer's solution rules out a type of antibody that the system's solution does not. Advice is generated from simple textual templates and each piece of feedback is presented immediately in a dialog box. AIDA does not offer to automatically correct identified problems and it does not record design histories or criticism resolutions.

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 a 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" [5].

Guerlain et al. evaluated AIDA by asking thirty-two professional laboratory technicians from seven different hospitals to solve four difficult problems [78]. 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 in comparison to the eight errors produced by the control group. This is the most in-depth system evaluation reported for any of the systems reviewed, and the only one with overwhelmingly positive results.

4.10. UIDA

The User Interface Design Assistant (UIDA) is a standalone system that critiques user interface window layouts for compliance with Motif style guidelines and consistency with other window layouts in the same application [79]. A designer works with a standard window layout editor, then saves the design to a file, then invokes UIDA to critique and modify the design. When UIDA detects a style violation, it asks the designer to confirm its suggested improvement, and then automatically changes the design.

UIDA has no user model to implicitly activate critics, but the designer may explicitly activate groups of style rules. UIDA performs analytic critiquing by applying 72 style rules written in an OPS5-like language. UIDA performs comparative critiquing by recording and comparing the particular set of rules satisfied by each layout. Comparative critiquing of this type does not require a generated solution. Relative to other systems reviewed, UIDA is weak in advisement and strong in improvement. Advice is limited to a sequence of brief prompts asking the designer to confirm suggested changes. Corrective automations are provided with the rules and result in a new version of the window layouts at the end of the critiquing session. A history of rule applications is kept only for the duration of the session to support comparing different layouts.

Like KRI/AG, the UIDA system does not satisfy the requirement that critiquing systems provide advice to designers while they are in the mental context of making design decisions.

4.11. SEDAR

The Support Environment for Design and Review (SEDAR) is a critiquing system for civil engineering. Specifically, it supports the design of flat and low-slope roofs [80-83]. 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 intended to support two distinct groups of people: designers and reviewers.

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. In some cases SEDAR can visually suggest a design improvement by drawing a new design element in one corner of the screen and draws an arrow to the general area where the new element should be placed.

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. Within the error detection strategy, critics are activated based on the state of a process model called the Designers' Task Model (DTM) and a Requirements Hierarchy Model (RHM). The DTM is essentially a work-breakdown structure for the roof design process. Additional relationships in the DTM indicate suggested temporal relationships and logical dependencies between steps. For example, decisions made during the drainage system layout step are known to be likely to interact with decisions made during the chimney layout step. A subset of all tasks in the DTM are marked as "focus" or "active" based on the designer's recent interactions with the tool. Each design requirement in the RHM can be marked "on" if critiquing relevant to that goal is desired, or "off" if it is not. The SEDAR system activates only those critics that support process steps in the "focus" or "active" state and requirements in the "on" state.

Assistance opportunities are detected by applying a set of design rules. Advice takes the form of brief error messages that are phrased somewhat constructively. Feedback items are sorted based on the type of design knowledge provided (physical violations, specification violations, and preference violations) and the activation (focus or active) of the DTM step that the feedback supports. Improvement is supported for some criticisms by displaying the type of design element that should be added to resolve the problem and a general location where it 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 does not record design histories.

SEDAR is unique among the critiquing systems reviewed here in that it identifies two classes of project stakeholders: designers and reviewers. Research on Argo acknowledges the possibly disparate interests of different stakeholders, but it does not name specific types of stakeholders. Every other system reviewed is intended to be used by designers only. 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's 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 reviews.

4.12. The Argo Family

We have built on the previous work described above and implemented specific support for each phase of the ADAIR critiquing process. The Argo family [84-87] of design environments consists of three tools: Argo/C2, Prefer, and Argo/UML. Software architects use Argo/C2 to enter a high-level specification of the structure of a software system. Prefer is used to model state-based requirements documents in the CoRE notation [89], a derivative of the SCR requirements method [90]. Argo/UML uses the Unified Modeling Language, a standard notation for object-oriented design [88, 91]. Some examples of critics found in Argo/C2 are found in Table 6. All of these systems use a common infrastructure which supports critics that are implicitly activated, analytically detect improvement opportunities, and provide feedback which is presented in a dynamic "to do" list.

The Argo infrastructure has both a user model and a goal model to support activation. Argo's decision model is automatically updated when the architect works with Argo's process model, however Argo does not infer goals from the partially specified design as does TraumaTIQ or infer the current process step as does SEDAR. Argo's critics analyze the design and produce feedback items with more kinds of design context than those produced by other systems; specifically, it provides a problem description, a suggestion of how to solve the identified problem, and contact information for relevant experts and stakeholders. Feedback management in Argo is more flexible than that of other systems reviewed. Improvement is supported by constructive advice and corrective automations, which may take the form of wizards. Argo records design activities done in the environment and some of their relationships, but it does not yet make much use of this information.

4.13. ICADS

Intelligent Computer Aided Design System (ICADS) supports architects in designing residential apartments [92]. Two critiquing modules are provided. A Floor Plan Design eXpert (FPDX) reminds designers of building codes related to fire exits and ventilation. An Interior Design eXpert (IDX) provides somewhat heuristic advice about the style and comfort of the apartment design, for example people should not have to pass through the kitchen on the way to the bathroom. ICADS is loosely integrated with a CAD system: a designer edits a design in the CAD system, saves the design to a file, invokes ICADS on that file, and reviews the design feedback produced by ICADS.

Figure 16: An Example ICADS Rule (from [92])

Condition: The master bedroom should be behind the central lines of the house, i.e., away from the main entrance.
Reason: People prefer a more private and quite location for the bedroom.
Suggestion: Move master bedroom away from front part of house.

ICADS does not have a user or task model and does not implicitly activate design critics. One simple activation rule found in ICADS is that the interior design expert is only applied if no problems were detected by the floor plan design expert. ICADS's critics are analytic and are implemented as a PROLOG program that uses forward-chaining to detect rule violations. An example ICADS critic is described in Figure 16. Each feedback item provides a short textual suggestion for solving the identified problem. ICADS does not record design histories.

ICADS is not integrated into a design tool, and thus cannot provide designers with feedback while they are still in the mental context of the decisions that are being critiqued.

5. Conclusions

5.1. 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 done 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 try 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. This is in part because little work has been done on the software engineering issues of developing reusable infrastructures, development methodologies, or authoring tools for creating critiquing systems.

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.

Table 6: Example Critics in Argo/C2

Name of Critic		Explanation
Critic Type	**Decision Category**	
Interface Mismatch		This component needs certain messages be sent or received, but they are not present.
Correctness	Message Flows	
Direct Connection		Violation of C2 style guidelines. Consider using a message bus to allow new components later.
Correctness	System Topology	
Component Choice		Here are other components that could "fit" in place of what you have: <<*list of components*>>.
Alternative	Component Selection	
Too Much Memory		Calculated memory requirements exceed stated goals. Try adjusting individual components, or moving some components to other hosts.
Correctness	Machine Resources	
Too Many Components		There are too many components at the same level of decomposition to be easily understood. Remove components or group components into a subsystem.
Evolvability	System Topology	
Hard Combination to Test		If you need to use these components together, please make arrangements with the testing manager.
Organizational	Component Selection	
Generator Limitation		The default code generator cannot make full use of this component. Consider using a different component or code generator.
Tool	Component Selection	
Portability Questionable		Your colleague, <<*name of person*>>, has reported difficulty using this component under <<*name of operating system*>>.
Experiential	Portability	

Detecting and highlighting errors is itself a useful form of support.

5.2. Research Directions Suggested in Previous Work

One difficulty with using critics is that designers are often made uncomfortable by the critic user interface metaphor. The metaphor is that of a critical person always watching over one's shoulder and finding fault with every decision. Designers using systems that follow this metaphor would face constant challenges to their authority, feel the need to guard against criticism, and rarely accept suggestions. Sumner, Bonnardel, and Kallak found that designers using the Voice Dialog Design Environment (VDDE) often changed their behavior to avoid situations where critics might fire and rarely followed the critic's advice [69]. TraumaTIQ lessens this effect by limiting provided feedback and focusing on urgent problems [77]. Framer and ATTENDING attempt to counter the negativity of critics with small amounts of praise [62, 39]. In building Argo we have tried to mitigate this effect by making critics constructive and by replacing the critic metaphor with that of a dynamic "to do" list. Critics in Argo are constructive whenever possible: their feedback explains the problem and its relevance to stated goals, suggests resolutions, and offers corrective automations in some cases. In fact, designers may find that design feedback provides more direct access to these automations than do standard command menus. The dynamic "to do" list metaphor does not challenge designers authority, instead it reminds them of pending decisions. We expect experienced designers to be familiar and comfortable with having many pending decisions.

Silverman and Mezher [66] propose decision networks as a new direction for critiquing systems that will give more complete design support. Decision networks are related groups of agents that together provide support for most phases of the ADAIR process. A dialog generation component activates appropriate agents based on a task model. Influencers provide tutoring before or during a task, usually by coloring regions of the design diagrams to indicate legal choices for the current design decision. Debiasers provide negative feedback when mistakes are made, as do most critics. Clarifiers present feedback to designers in graphical or textual form. Directors provide task-specific support for carrying out improvements, as do wizards. Silverman and Mezher suggest that debiasers should learn from interactions with designers; specifically, it should suppress criticism that the designer had previously rejected. In Argo, critics and wizards are used in combination. Possible uses of machine learning in design support are discussed below.

5.3. Trends in Intelligent User Interfaces

Comparing critiquing systems to more mature forms of intelligent user interfaces suggests how critiquing systems themselves might mature.

1. Integration into design tools and design tasks. Early spelling checking systems were stand-alone programs that only identified errors without offering fixes, in contrast modern spelling checkers are tightly integrated with word processors and provide sophisticated correction capabilities. Critiquing systems should also be tightly integrated into design tools and provide automation to ease correction tasks.

2. Improvement without requiring confirmation. When confidence in the assistance opportunity and the corrective action is high enough, corrective action can be taken without consulting the user. This occurs in dynamic revision and user interface agents under certain conditions. Likewise, some design critics may have enough confidence to make a correction without notifying the designer. Doing so relieves the designer of the burden of interacting with critics for every error, however it runs the risk of changing the design in ways that the designer did not intend. One conservative way to manage the problem of unintended design changes is to only automate the most obvious corrective actions. User preferences, design histories, and support for out-of-order undo may allow wider application of automatic improvements to while still keeping designers informed and in control.

3. Combination with other approaches. Most wizard systems require explicit activation and do not proactively detect when they are applicable to the current design, but they strongly support design improvements. Tools such as Microsoft Office'98 have shown that wizards combined with coaches, which are strong on activation and detection, are more effective than either feature alone.

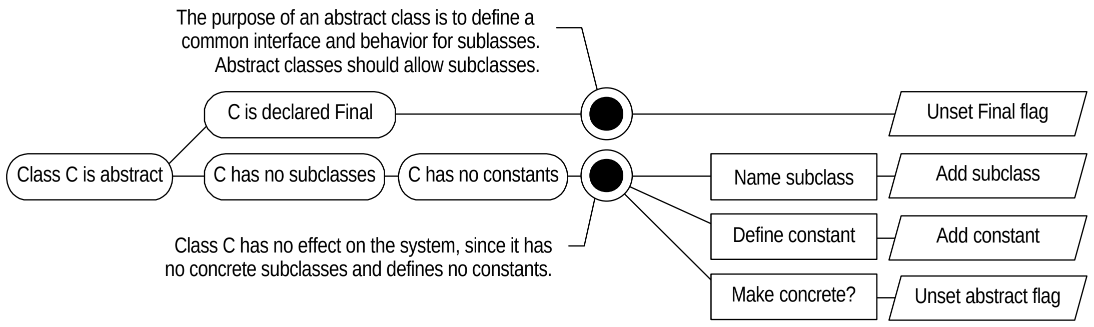
5.4. Our Recommendations for New Research Directions in Critiquing Systems

We recommend that future research on critiquing systems attempt to achieve more comprehensive support for identified challenges of design, and provide more automation to support identified design tasks. Furthermore, experience with larger critiquing systems is needed to uncover issues of scale and practical use. Specifically, we recommend the following.

1. Combination with other approaches. Critics should be combined with other types of intelligent user interfaces that provide complementary strengths. For example, coaches can be embedded in design tools alongside critics to deliver two complementary types of knowledge. Also, critics can be combined with wizards to make wizards' procedural automation more accessible and critics' feedback more constructive.

2. Organization-specific critics. More research emphasis should be placed on the use of critiquing systems in the context of a development organization. Most critiquing work to date has provided support for a specific design domain, e.g., low-slope roof design, or a specific design notation, e.g., the Unified Modeling Language. Little work has been done on critics that advise designers about the interests of other stakeholders in the development organization. Typically, organizational policies are conveyed to designers in memos or employee manuals. These paper-based communications are often not effective; this is true, in part, because the communication happens outside the context of day-to-day

Figure 17: Diagrammatic Representation of Two UML Critics and Associated Wizards.


The purpose of an abstract class is to define a common interface and behavior for sublasses. Abstract classes should allow subclasses.
C is declared Final
Unset Final flag
Class C is abstract
C has no subclasses
C has no constants
Name subclass
Add subclass
Define constant
Add constant
Class C has no effect on the system, since it has no concrete subclasses and defines no constants.
Make concrete?
Unset abstract flag

design work. We expect that critics can be an effective means of making designers aware of the organizational consequences of design decisions. Fulfilling the potential of the research direction will likely require integration with organizational memory systems or adoption of some of their concepts.

3. Reusable critiquing infrastructure. A reusable, scalable, domain-independent critiquing infrastructure should be developed based on the ADAIR process. To date, each critiquing system implementation has essentially been started from scratch. This has limited the scale and depth of these systems and the research topics that have been explored. Without an infrastructure that supports authoring and application of thousands of critics, critiquing systems will not provide enough knowledge to make a significant impact on design practice. Without addressing the finer points of critiquing strategies, feedback presentation and management, and constructiveness, critiquing systems will not provide their knowledge effectively enough to make a significant impact on design practice. Without systems large enough to make a significant impact on practice, research efforts have been limited to laboratory studies.

4. Critic development life-cycle. Little work has been done on the maintenance and management of critics, since most critiquing systems have only reached the prototype stage and have never been expanded to large numbers of critics or deployed to large numbers of users. One exception to this is the work done by Fischer et al. on “seeding, evolutionary growth, and reseeding” [57]. Critic development proceeds through a life-cycle in which useful critiques are identified, implemented as critics, refined as the result of use, and may eventually be retired. Most critics in the systems reviewed arose from domain analysis by tool builders rather than being nominated by practicing designers. Argo allows designers to email their comments on the usefulness of particular critics to critic maintenance staff. More infrastructure should be put in place to facilitate communication, collect statistics that measure the return on investment in critics, and that guide maintenance and development of critics. Features from organizational memory systems, usage monitoring, and software metrics may prove useful in maintaining and managing critics.

5. Critic implementation language. No widely useful critic implementation language has been proposed to date. Some of the systems reviewed use expert system shells to implement critics, while others implement critics in third-generation programming languages. Future research should explore the possible advantages of special purpose critic languages as compared to the use of more general-purpose languages. Many researchers have investigated end-user programming, e.g., [112, 113]. However, none has been successfully demonstrated as useful to practicing designers. One key point of comparison is the capability for practicing designers to easily specify improvements to critics. Figure 17 shows our 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 dialog windows presented to the user as a step in the wizard. Parallelogram nodes are actions that modify the design or user model. We hope that practicing designers 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.

6. Limited machine learning. Machine learning techniques could be used to fine-tune the activation of critics, priorities for design feedback, and the organization of feedback into categories. The main knowledge content provided by critics cannot be effectively learned from designers, because many types of design decisions happen rarely and the designer’s own knowledge may be lacking. However, some of the interactions between designers and critics are repetitive and could be made more effective by using machine learning. For example, if a critiquing system allows the designer to categorize feedback items into folders (much like email systems do), then the same techniques used in email sorting interface agents could be used. Furthermore, programming by demonstration may allow designers to make critics more constructive by repeatedly demonstrating how a certain type of design error can be corrected, and then allowing the critic to perform that correction automatically of after a brief confirmation dialog. User interface agents capable of this form of learning are described in [27] and [31].

7. Integration with design rationale systems. Design rationale systems support designers by capturing and making available information about the reasoning behind design decisions [93-97]. We feel that critics can help elicit design rational by prompting designers to make, change, or explain their decisions. Also, the knowledge contained in design rational is an important form of design context that can be used by critics when presenting identified problems to designers. The Janus and Argo families of design environments each support design rationale in different ways. Janus emphasizes delivery of rationale to explain criticism but does not provide much support for the task of building a hypermedia argumentation database. Argo captures a log of how design criticism is resolved, including any rationale entered by the designer when feedback is dismissed; however, it does not provide support for retrieving rationale when needed. To better serve designers, design support systems should include design rationale features that aid in both the collection and retrieval of rationale.

We have shown that the phases of the ADAIR critiquing process can serve as check-list items when evaluating the completeness of support provided by critiquing systems, and intelligent user interfaces in general. We also believe that ADAIR can be used constructively in the design of new critiquing systems. Furthermore, combining critics with corrective automations makes them constructive, which not only provides more complete support for designers' tasks, but also mitigates the negative connotations of the critic metaphor.

REFERENCES

1. Fischer, G. Domain-Oriented Design Environments. *Proc. of The 7th Knowledge-Based Software Engineering Conference.* 204-213.
2. 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.
3. 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.
4. Silverman, B. G. Survey of expert critiquing systems: practical and theoretical frontiers. *Communications of the ACM*. vol. 35. no. 4. April 1992. pp. 106-127.
5. 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.
6. Fox, R. News track. *Communications of the ACM.* vol. 40. no. 5. May 1997. pp. 9-10.
7. Brooks, F. P. No silver bullet: essence and accidents of software engineering. *IEEE Computer*. vol. 20. no. 4. April 1987. pp. 10-19.
8. Boehm, B. W. Software Engineering. *IEEE Transactions on Computers*. vol. C-25. no. 12. Dec. 1976. pp. 1226-1241.
9. Dunn, R. H. Software Defect Removal. New York: McGraw-Hill. 1984.
10. Simon, H. A. *The Sciences of the Artificial*. 3rd ed. 1996. MIT Press. Cambridge MA.
11. Boehm, B. W. A spiral model of software development and enhancement. *IEEE Computer*, vol. 21. May 1988. pp. 61-72.
12. Stacy, W. and MacMillian, J. Cognitive bias in software engineering. *Communications of the ACM*. June 1995. pp. 57-63.
13. Fischer, G. and Reeves, B. Beyond intelligent interfaces: exploring, analyzing, and creating success models of cooperative problem solving. In R. Baecker, J. Grudin, W. Buxton and S. Greenberg: *Reading in Human-Computer Interaction: Toward the Year 2000*. Morgan Kaufmann, 1995, pp. 822-831.
14. Thomas, C. G. and Fischer, G. Using agents to personalize the Web. In *Proceedings of the 1997 International Conference on Intelligent User Interfaces*. 1997. pp. 53-60.
15. Hook, K. *Designing and Evaluating Intelligent User Interfaces*. Tutorial at the 1998 International Conference on Intelligent User Interfaces. San Francisco CA. January 1998.

Tutors

16. Anderson, J. R., Corbett, A. T., Koedinger, K. R., and Pelletier R. Cognitive tutors: lessons learned. *Journal of Learning Sciences*. vol. 4. 1995. pp. 167-207.
17. Ashley, K. D. and Aleven, V. Toward an intelligent tutoring system for teaching law students to argue with cases. *Proc. Third International Conference on Artificial Intelligence and Law*. 1991. pp. 42-52.

Coaches

18. Horvitz E. Agents with beliefs: reflections on bayesian methods for user modeling. *Proc. Sixth International Conference on User Modeling*. Chia Laguna, Sarinia. June 1997.
19. Microsoft Corporation, *Getting results with Microsoft Office 97*. 1997.
20. Selker T. Coach: a teaching agent that learns. *Communications of the ACM*. vol. 37. no. 7. July, 1994.

Wizards

21. Dryer, D. C. Wizards, guides, and beyond: rational and empirical methods for selecting optional intelligent user interface agents. *Proc. 1997 International Conference on Intelligent User Interfaces*. Orlando FL. January 1997. pp. 265-268.
22. Microsoft Corporation, *Getting results with Microsoft Office 97*. 1997.

Automatic Word Correction

23. Microsoft Corporation, *Getting results with Microsoft Office 97*. 1997.
24. Kukich, K. Techniques for automatically correcting words in text. *ACM Computing Surveys*. vol. 24. no. 4. Dec. 1992. pp. 377-439.
25. 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.
26. Teitelman, W. and Masinter, L. The Interlisp Programing Environment. *IEEE Computer*. vol. 14. no. 4. April 1981. pp. 25-33.

User Interface Agents

27. Maes, P. Agents that reduce work and information overload. *Communications of the ACM*. vol. 37. no. 7. July 1994. pp. 31-40.
28. Kautz, H. A., Selman, B. and Coen, M. Bottom-up design of software agents. *Commununications of the ACM*. vol. 37. no. 7. July 1994. pp. 143-146.

29. Boden, M. A. Agents and creativity. *Communications of the ACM*. vol. 37. no. 7. July 1994. pp. 117-121.
30. Kaye, A. R. and Karam, G. M. Cooperating knowledge-based assistants for the office. *ACM Transactions on Information Systems*. vol. 5. no. 4. Oct. 1987. pp. 297-326.
31. Cypher, A., editor. *What What I Do: Programming by Demonstration*. 1993. MIT Press.

Non-Critiquing System Examples

32. 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.
33. Ballance, R. A., Graham, S. L., and Van De Vanter, M. L. The Pan language-based editing system for integrated development environments. *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*. Irvine, CA. 3-5 Dec. 1990. pp. 77-93.

ONCOCIN

34. 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.
35. 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.
36. 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.
37. Bischoff, M. B., Shortliffe, E. H., Scott, A. C., and Carlson, R.W. Integration of a computer-based consultant into the clinical setting. In *Proceedings of the Seventh Annual Symposium on Computer Applications in Medical Care*. Silver Spring, MD, USA: IEEE Computer Society Press. 1984. pp. 149-152.
38. Shortliffe, E. H. Update on ONCOCIN: a chemotherapy advisor for clinical oncology. In *Proceedings of the Eighth Annual Symposium on Computer Applications in Medical Care*. Silver Spring, MD, USA: IEEE Compuer Society Press. 1984. pp. 24-25.

ATTENDING Family

39. 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.
40. Miller, P. L. and Black, H. R. Medical plan-analysis by computer: critiquing the pharmacologic management of essential hypertension. *Computers and Biomedical Research*, Feb. 1984. vol. 17. no. 1. pp. 38-54.
41. Rothschild, M. A., Swett, H. A., Fisher, P. R., Weltin, G. G., and Miller P. L. Exploring subjective vs. objective issues in the validation of computer-based critiquing advice. *Computer Methods and Programs in Biomedicine*. Jan. 1990, vol. 31. no. 1. pp. 11-18.
42. Miller, P. L. Expert Critiquing Systems: *Practice-based Medical Consultation by Computer*. Springer-Verlag. New York, 1986.

Janus Family

43. Fischer, G. Cognitive view of reuse and redesign. *IEEE Software*. July 1987.
44. 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.
45. Fischer, G. Human-computer interaction software: lessons learned, challenges ahead. *IEEE Software*. January 1989. pp. 44-52.
46. Fischer, G., McCall, R., and Morch, A. Design environments for constructive and argumentative design. *Proceedings of Conference on Human Factors in Computing Systems*. Austin, TX, USA, 30 April-4 May 1989. pp. 269-75.
47. Fischer, G., McCall, R., and Morch, A. JANUS: integrating hypertext with a knowledge-based design environment. *Hypertext ‘89 Proceedings*. Pittsburgh, PA, USA, 5-8 Nov. 1989. pp. 105-17.
48. Fischer, G., Lemke, A. C., Mastaglio, T., and Morch, A.I. Using critics to empower users. Empowering People, Seattle, WA, USA, 1-5 April 1990). SIGCHI Bulletin, April 1990 (special issue CHI ‘90 Conference Proceedings). pp. 337-47.
49. Fischer, G., Lemke, A. C., Mastaglio, T., and Morch, A. I. The role of critiquing in cooperative problem solving. *ACM Transactions on Information Systems*. April 1991. vol. 9. no. 2. pp. 123-151.
50. Fischer, G., Mastaglio, T. A Conceptual Framework for Knowledge-Based Critic Systems. *Decision Support Systems*. Elsevier Science Publishers B.V. 1991. pp. 355-378.
51. 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.* Nov. 1991. vol. 35. no. 5. pp. 695-721.
52. Fischer, G., Lemke, A. C., McCall, R., and Morch, A. I. Making argumentation serve design. *Human-Computer Interactions.* 1991. vol. 6. no. 3-4. pp. 393-419.
53. 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.
54. 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.* 1993. Amsterdam, The Netherlands. pp. 157-164.
55. Fischer, G. and Eisenberg, M. Programmable Design Environments: Integrating End-User Programming with Domain-Oriented Assistance. *Human Factors in Computing Systems, CHI’94 Conference Proceedings*. pp. 431-437.
56. Fischer, G., Nakakoji, K., Ostwald. J., Stahl, G., and Sumner, T. Embedding Critics in Design Environments. *The Knowledge Engineering Review Journal, Special Issue on Expert Critiquing*. vol. 8. no. 4. 1993. pp. 285-307.
57. 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 24-28, 1994. pp. 292-298.
58. 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.
59. 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, USA. October, 1995. pp. 7-15.

60. Fischer, G. Domain-Oriented Design Environments. *Proceedings of the 18th International Conference on Software Engineering*. Berlin, Germany. March 25-29, 1996. pp. 517-520.

Framer

61. Fischer, G. Human-computer interaction software: lessons learned, challenges ahead. *IEEE Software*. January 1989. pp. 44-52.
62. Lemke, A. C., Fischer, G. A cooperative problem solving system for user interface design. *AAAI-90*. pp. 219-240.
63. Lee, E. User interface development tools. *IEEE Software*. May 1990. pp. 31-36.
64. Rettig, M. Cooperative Software. *Communications of the ACM*. April 1993. pp. 23-28.

KRI/AG

65. Lowgren, J. and Nordqvist, T. Knowledge-based evaluation as design support for graphical user interfaces. In *Conference proceedings on Human factors in Computing Systems*. 1992. pp. 181-188.

CLEER

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

VDDE

67. Harstad, B. *New Approaches to Critiquing: Pluralistic Critiquing, Consistency Critiquing, and Multiple Intervention Strategies*. Masters thesis. University of Colorado at Boulder. Dept. of Computer Science. 1993.
68. 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.
69. Sumner, T., Bonnardel, N., and Kallak, B. H. The cognitive ergonomics of knowledge-based design support systems. Conference *Proceedings on Human Factors in Computing Systems*. 1997. pp. 83-90.

TramaTIQ

70. Gertner, A. Real-time critiquing of integrated diagnosis/therapy plans. *Proc. AAAI Workshop on Expert Critiquing Systems.* Washington D.C. August 1993.
71. Gertner, A. Ongoing Critiquing During Trauma Management. *Proc. AAAI Spring Symposium on Artificial Intelligence in Medicine: Interpreting Clinical Data*. Stanford, CA. March 1994.
72. Gertner, A. Responding to Users' Informational Needs in Time-Critical Situations. *Proc. 4th International Conference on User Modeling.* Hyannis, MA. August 1994.
73. Gertner, A., Webber, B. L., and Clarke, J. R. Upholding the Maxim of Relevance during Patient-Centered Activities. *Proc. 4th Conference on Applied Natural Language Processing*. Stuttgart, Germany. October 1994.
74. Gertner, A. and Webber, B. L. Recognizing and Evaluating Plans with Diagnostic Actions. In *Working notes of the IJCAI-95 workshop on plan recognition*. Montreal, Canada. August 1995.
75. Gertner, A. and Webber, B. L. A Bias towards Relevance: Recognizing plans where goal minimization fails. *Proceedings of the 13th National Conference on Artificial Intelligence*. Portland, OR. August 1996.
76. Gertner, A., Webber, B. L. and Clarke, J. R. On-Line Quality Assurance in the Initial Definitive Management of Multiple Trauma: Evaluating System Potential. *Artificial Intelligence in Medicine*. vol. 9. 1997. pp. 261-282.
77. Gertner A. S. and Webber B. L., TraumaTIQ: online decision support for trauma management. *IEEE Intelligent Systems*. January/February 1998. pp. 32-39.

AIDA

78. 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.

UIDA

79. Bolcer, G. A. User interface design assistance for large-scale software development. pp. 203-218. *Automated Software Engineering*. vol. 2. no. 3. September 1995. pp 203-218.

SEDAR

80. Fu, M. C. *A Task-Based Model of Design for Flexible Control in an Expert Critiquing System*. Masters thesis, University of Illinois at Urbana-Champaign. 1995.
81. Fu, M. C. An automated design and review assistant: SEDAR. *Proceedings of the Third Congress on Computing in Civil Engineering*. Anaheim, California, June 17-19, 1996. pp. 118-125.
82. Fu, M. C. and East, E. W. The support environment for design and review (SEDAR) for flat and low-slope roofs. U*.S. Army Corps of Engineers Construction Engineering Research Laboratories Technical Manuscript 96/99*. September 1996.
83. 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.

Argo family

84. 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.
85. Robbins, J. E., Hilbert, D. M., and Redmiles, D. F. Argo: a design environment for evolving software architectures. In *Proceedings of the 1997 International Conference on Software Engineering*. Boston, MA, USA, 17-23 May 1997. pp. 600-601.
86. Robbins, J. E., Hilbert, D. M., and Redmiles, D. F. Extending design environments to software architecture design. *Automated Software Engineering*. 1998. In press.
87. Robbins, J. E. and Redmiles, D. F. Software architecture critics in the Argo design environment. *Knowledge-Based Systems*. 1998. In press.
88. Robbins, J. E., Medvidovic, N., Redmiles, D. F., and Rosenblum, D. S. Integrating architecture description languages with a standard design method. *In Proceedings of the 1998 International Conference on Software Engineering*.

Kyoto, Japan. 19-25 April 1998. pp. 209-18.

89. 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.
90. 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.
91. Rational Partners (Rational, IBM, HP, Unisys, MCI, Microsoft, ObjecTime, Oracle, i-Logix, and other CASE vendors and customers). *UML notation guide*. Object Management Group document ad/97-07-08. Available from http://www.omg.org/ docs/ad/97-07-08.

ICADS

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

Design Rationale

93. Jarczyk, A., Loeffler, P., and Shipman, F. Design rationale for software engineering: a survey. *Proceedings of the 25th Hawaii International Conference on System Sciences*. pp. 577-586.
94. Reeves, B. *The Role of Embedded Communication and Embedded History in Collaborative Design*. Ph.D. Dissertation. Department of Computer Science, University of Colorado, 1993.
95. Reeves, B. N. and Shipman, F. Supporting communication between designers with artifact-centered evolving information spaces. *Proceedings of the Conference on Computer-Supported Cooperative Work*. November 1992. pp. 394-401.
96. Lee J. Design rationale systems: understanding the issues. *IEEE Expert*. 1997. 78-85.
97. Wong, S. T. C. Preference-based decision making for cooperative knowledge-based systems. *ACM Transactions on Information Systems*. vol. 12. no. 4. Oct. 1994. pp. 407-435.

Training on Demand

98. Bailin, S. C., Gattis, R. H., and Truszkowski, W. A learning-based software engineering environment for reusing design knowledge. *International Journal of Software Engineering and Knowledge Engineering*. Dec. 1991. vol. 1. no. 4. pp. 351-371.
99. Fischer, G. Conceptual Frameworks and Computational Environments in Support of Learning on Demand, in A. DiSessa, C. Hoyles and R. Noss (eds): *The Design of Computational Media to Support Explanatory Learning*. Springer Verlag, Heidelberg. 1995. pp. 463-480.
100. Fischer, G. Distributed Cognition, Learning Webs and Domain-Oriented Design Environments. In *Proceedings of the Conference on Computer Supported For Collaborative Learning (CSCL'95)*. Indiana University. October 1995. pp. 125-129.
101. Fischer, G. Supporting Learning on Demand with Design Environments. *Proceedings of the International Conference on the Learning Sciences*. 1991. pp. 165-172.
102. Eden, H., Eisenberg, M., Fischer, G., and Repenning, A. Domain-Oriented Design Environments: Making Learning a Part of Life. *Communications of the ACM*. vol. 39. no. 4. April 1996. pp. 40-42.
103. Fitzgerald, E. P. and Cater-Steel, A. Champagne training on a beer budget. pp. 49-60. *Communications of the ACM*. vol. 38. no. 7. July 1995.
104. Carroll, J., Aaronson, A. Learning by doing with simulated intelligent help. *Communications of the ACM*. vol. 31. no. 9 Sept. 1988. pp. 1064-1079.

Cognitive Theories of Design

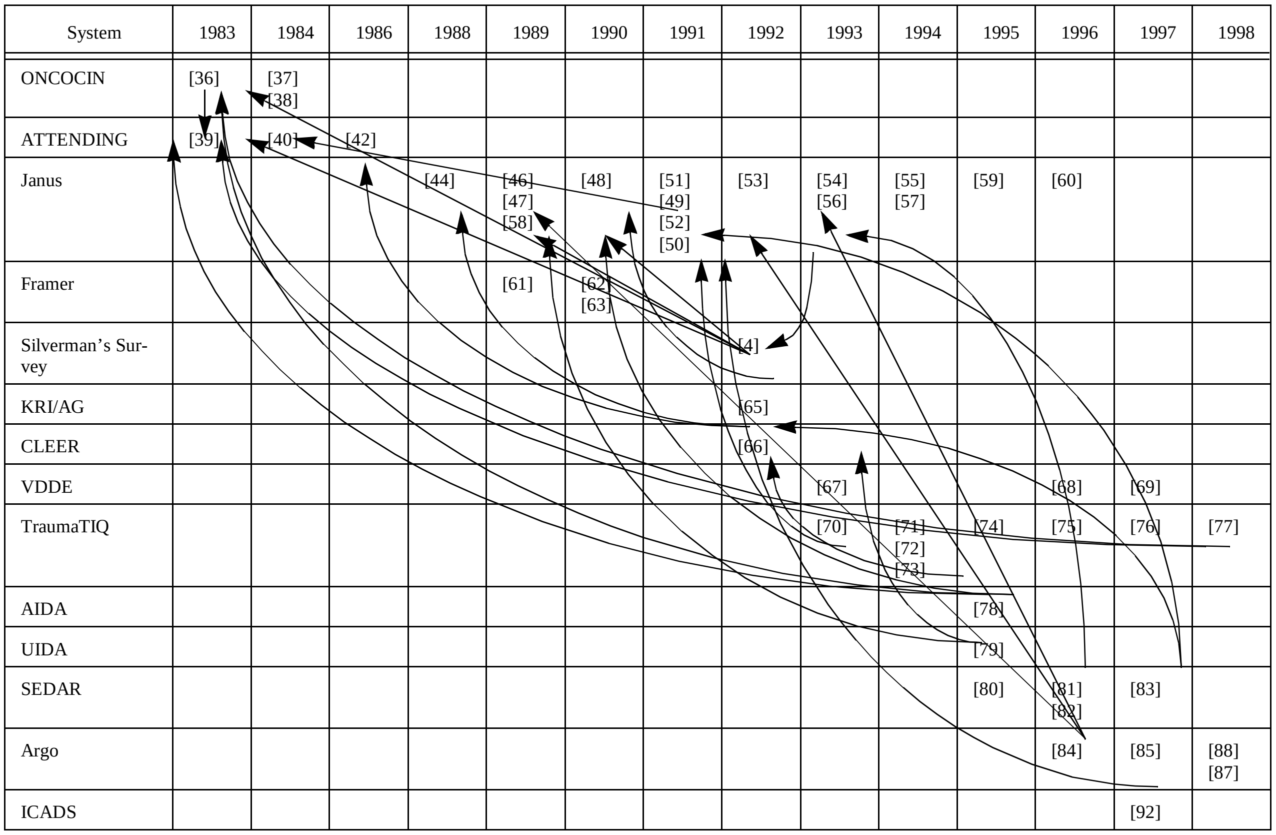
105. Guindon, R., Krasner, H., and Curtis, W. Breakdown and processes during early activities of software design by professionals. In: Olson, G. M. and Sheppard S., eds. *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ: Ablex Publishing Corporation. 1987. pp. 65-82.
106. Stacy, W. and MacMillian, J. Cognitive Bias in Software Engineering. *Communications of the ACM*. June 1995. pp. 57-63.
107. Visser, W. More or Less Following a Plan During Design: Opportunistic Deviations in Specification. *Int. J. Man-Machine Studies*. 1990. pp. 247-278.
108. Schoen, D. *The Reflective Practitioner: How Professionals Think in Action*. 1983. New York: Basic Books.
109. Schoen, D. Designing as reflective conversation with the materials of a design situation. *Knowledge-Based Systems*. 1992. vol. 5, no. 1. pp. 3-14.
110. Hayes-Roth, B. and Hayes-Roth, F. A Cognitive Model of Planning. *Cognitive Science*. vol. 3, no. 4. 1979. pp. 275-310.
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.

Critic Authoring Languages

112. Girgensohn, A. *End-user modifiability in knowledge-based design environments*. Ph.D. Thesis. University of Colorado at Boulder. Dept. of Computer Science. June 1992.
113. Riesbeck, C. K. and Dobson, W. Authorable critiquing for intelligent educational systems. *Procedings of the 1998 International Conference on Intelligent User Interfaces*. Sanfrancisco CA. January 6-9, 1998. pp. 145-152.

Appendix A: Citation Graphs

Table A-1: Citation Graph by System and Year

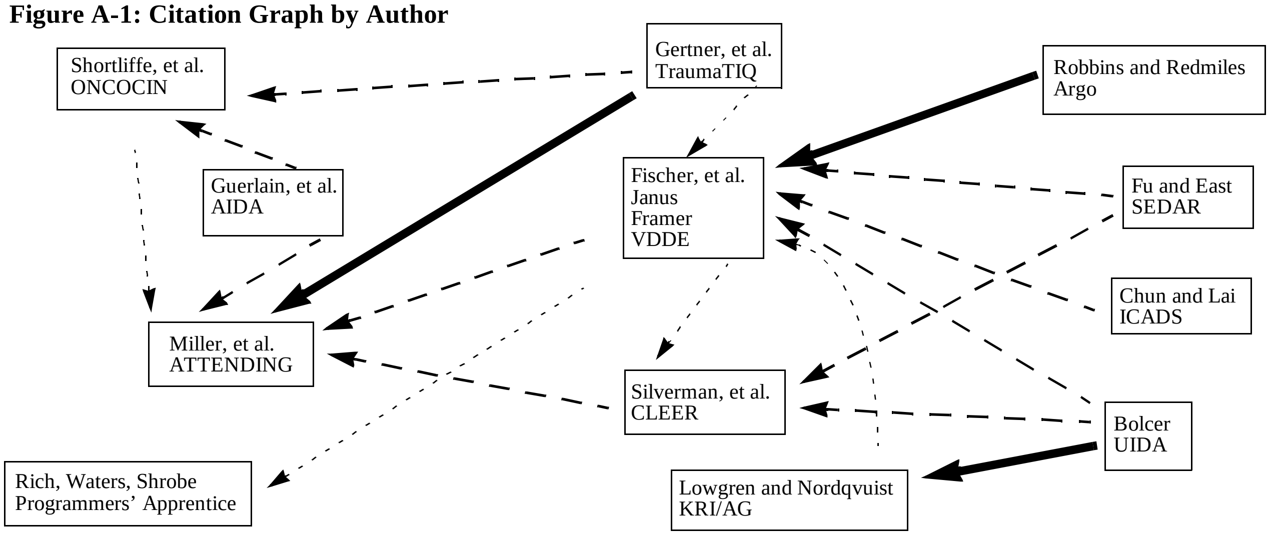


System	1983	1984	1986	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998
ONCOCIN	[36]	[37] [38]												
ATTENDING	[39]	[40]	[42]											
Janus				[44]	[46] [47] [58]	[48]	[51] [49] [52] [50]	[53]	[54] [56]	[55] [57]	[59]	[60]		
Framer					[61]	[62] [63]								
Silverman's Survey								[4]						
KRI/AG								[65]						
CLEER								[66]						
VDDE									[67]			[68]	[69]	
TraumaTIQ									[70]	[71] [72] [73]	[74]	[75]	[76]	[77]
AIDA											[78]			
UIDA											[79]			
SEDAR											[80]	[81] [82]	[83]	
Argo												[84]	[85]	[88] [87]
ICADS													[92]	

Table A-1 shows the critiquing papers referenced in this survey, grouped by year and system. Arrows indicate that one paper at the base of the arrow cites a paper indicated at the head of the arrow. Arrowheads do not always refer to papers referenced in this survey, instead they indicate the group of authors and year of the publication cited.

Figure A-1generalizes the information in the table. Nodes in the diagram indicate groups of collaborating authors. Arrows indicate that one or more papers by the authors at the tail of the arrow cite one or more papers by the authors at the head. The line thickness indicates our subjective evaluation of the degree to which one body of work is based on the other.

Figure A-1: Citation Graph by Author



Shortliffe, et al. ONCOCIN
Gertner, et al. TraumaTIQ
Robbins and Redmiles Argo
Guerlain, et al. AIDA
Fischer, et al. Janus Framer VDDE
Fu and East SEDAR
Chun and Lai ICADS
Miller, et al. ATTENDING
Silverman, et al. CLEER
Bolcer UIDA
Rich, Waters, Shrobe Programmers' Apprentice
Lowgren and Nordqvuist KRI/AG