HUMAN-COMPUTER INTERACTION, 1995, Volume 10, pp. 1-37

Living Design Memory: Framework, Implementation, Lessons Learned

Loren G. Terveen, Peter G. Selfridge, and M. David Long
AT&T Bell Laboratories

ABSTRACT

We identify an important type of software design knowledge that we call *community-specific folklore* and discuss problems with current approaches to managing it. We developed a general framework for a *living design memory,* built a design memory tool, and deployed the tool in a large software development organization. The tool effectively disseminates knowledge relevant to local software design practice. It is embedded in the organizational process to help ensure that its knowledge evolves as necessary. This work illustrates important lessons in building knowledge management systems, integrating novel technology into organizational practice, and carrying out research–development partnerships.

Loren G. Terveen is a computer scientist with interests in organizational memory, human–computer collaboration, and cooperative work; he is a Member of Technical Staff in the Human–Computer Interface Research Department of AT&T Bell Laboratories. **Peter G. Selfridge** is a computer scientist with interests in organizational knowledge, effective knowledge delivery, and graphical tools; he is a Member of Technical Staff in the Artificial Intelligence Principles Research Department of AT&T Bell Laboratories. **M. David Long** is a computer scientist with interests in the design process and design support tools; he was a Member of Technical Staff in the International Switching Customer Business Unit of AT&T Bell Laboratories at the time this work was done and now is a Senior Systems Engineer for Cadre Technologies.

CONTENTS

1. INTRODUCTION
2. THE KNOWLEDGE MANAGEMENT PROBLEM IN LARGE-SCALE SOFTWARE DEVELOPMENT
3. A FRAMEWORK FOR LIVING DESIGN MEMORY
4. THE DESIGNER ASSISTANT–A DESIGN MEMORY TOOL
   - 4.1. Evolving a Prototype
   - 4.2. From Prototype to Deployed Tool
   - 4.3. Technical Description of DA
   - 4.4. How DA Knowledge Has Evolved
5. RELATED WORK
6. DISCUSSION
   - 6.1. Lessons Learned
   - 6.2. DA Status and Evaluation
   - 6.3. DA Limitations and Future Work
7. CONCLUSIONS

1. INTRODUCTION

Our research is aimed at improving the effectiveness of large-scale software development, a notoriously difficult and expensive activity. Several factors contribute to this situation. Software development is a new discipline; this leads to rapid change in languages, tools, and methodologies. Many individual software constructs and components can be composed to build large systems; this leads to systems that perform very complex tasks, are built by many people, and are beyond the understanding of any single person. Software is a highly malleable medium; this makes change possible, and market pressures make change likely.

Large-scale software development is a knowledge-intensive activity. The knowledge required for effective software development is vast, complex, heterogeneous, and evolving. Much of the knowledge required to be a successful developer in a particular organization is *community specific,* concerning the existing software base, the application domain, local programming conventions, and the expertise of particular individuals. This knowledge typically is managed as *folklore*–it is informally maintained and disseminated by experienced developers. This process is ineffective (not everyone gets the knowledge he or she needs, inefficient (communication of knowledge, whether in formal meetings or informal consulting, comes to take up more and more time), and fragile (loss of key personnel can mean loss of critical knowledge).

We addressed the problem of managing design knowledge in the context of a large AT&T software development organization. The goal of our work was to construct a system for recording and effectively disseminating folklore design knowledge throughout the organization. We aimed to improve both the software product and the software development process and to achieve a better understanding of the interplay and potential synergy between technology and organizational processes.

We developed a general framework for providing folklore knowledge, constructed a system that instantiates the framework, and deployed the system in the organization. Our system is called the *Designer Assistant (DA)*. It has been in use since October 1992. It has been used hundreds of times by software developers and has gone through several versions; most important, its knowledge has evolved as usage revealed incomplete or incorrect knowledge. It thus is an important example of a *living design memory*. Integrating the system into the organization's design practice was key in making it living. Keeping organizational memory up-to-date and relevant is a big challenge in today's competitive business climate; however, many approaches to managing information neglect the critical factor of knowledge evolution. Our approach also significantly extends previous approaches for delivering design assistance, capturing design rationale, and representing organizational information; detailed comparisons are made in Section 5.

The primary lesson of this work is that technology and organizational processes are mutual, complementary resources. We have found that successful integration of new technology into an organization depends on developing the technology in coordination with existing practice and processes. An organization's internal design, patterns of coordination, information flow, local culture, and technology must be integrated into a coherent, overall solution. Technology–process integration is a key enabler of knowledge evolution. Other related lessons include:

- The pragmatics of knowledge use are critical. Simply recording facts is not enough; issues such as where in the process knowledge is to be accessed, how to access relevant knowledge from a large information space, and how to allow for change also must be addressed for a knowledge management system to be successful.
- Addressing organizational problems while offering direct benefit to individuals is key. Although we focus on helping an organization manage its knowledge effectively–thus alleviating mainly problems that manifest themselves on an organizational level (e.g., excessive communication and coordination overhead, duplicated effort, long product delivery times)–we know that it is individuals who implement new practices and use new technology, and they need proper incentives to cooperate with such initiatives (Grudin, 1988).

- Computer information delivery and computer mediation of human collaboration must be tightly interwoven. Our approach integrates the perspectives of *cooperative problem solving* (Fischer, 1990; Fischer, Lemke, Mastaglio, & Morch, 1991; Silverman, 1992; Terveen, 1993, in press), in which a computer system assists a person in performing a task, and *computer-supported cooperative work,* in which computer technology is used to mediate collaboration among humans.
- A research–development partnership is a mutual learning process. We discovered the limits of the "technology transfer" metaphor; instead of engaging in a discrete act of transfer, we engage in an ongoing cycle of problem–solution coevolution that involves research, rapid prototyping, user testing, deployment, and user feedback; issues such as access to expertise, knowledge of local culture and technology, credibility, and ownership become crucial.

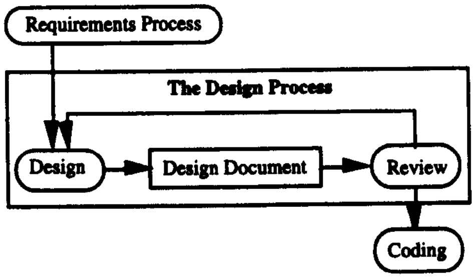
In this article, we first explore the knowledge management problem in more detail and discuss challenges to acquiring, maintaining, and disseminating design knowledge. Then we describe a framework for integrating a design memory tool into a software development process. Next we describe the implemented tool that instantiates the framework, including the prototype version, its evolution, a technical description of the current tool, and a discussion of how its knowledge has evolved. Then, after discussing related work and comparing our approach to other approaches, we discuss the lessons learned, the current status and evaluation of the tool, and its limitations and our plans for the future. We conclude with a short summary.

2. THE KNOWLEDGE MANAGEMENT PROBLEM IN LARGE-SCALE SOFTWARE DEVELOPMENT

The work described here is a collaboration between an AT&T computer science research organization (represented by Terveen & Selfridge) and a large AT&T software development organization, the International Switching Customer Business Unit, or ISCBU (represented by Long). ISCBU consists of several thousand people who maintain and enhance a large telecommunications software system, the 5ESS^{TM}. The goal of the collaboration was to address problems in managing design knowledge.

It is important to realize that software development in this organization never begins "from scratch." From a high-level perspective, the "same" hardware and software system has been delivered to many customers over the last 10 years. In reality, however, there is intensive ongoing development centered around maintaining and enhancing the software. Each new customer has somewhat different requirements; new services and features are planned and developed; bugs must be repaired. This means that development always must take into account the existing software base and

Figure 1. **ISCBU design process before introduction of DA.**



thus must be based on knowledge of how things have been done, why they were done that way, and who did them. A design memory would be particularly helpful in this type of development situation.

Our work has focused on the design process. This process starts with a *requirements document,* originating from a customer request or an internal source. The requirements document describes a new feature of the software in terms of customer needs. It is used by a software developer to produce a *design document,* which describes how that new feature will be implemented and added to the existing software architecture. This design document then is formally reviewed by a committee of experts. If significant feedback is generated in the review, the design process iterates. After the design document is complete and approved, it is passed to a coding phase. This process is outlined in Figure 1.

One major problem in the design process is the lack of access to design knowledge. A particularly important type of knowledge is what we call *community-specific folklore*–information that is relevant to the local design practice and that "you can't learn in school." This knowledge involves things such as real-time and performance constraints ("One real-time segment shouldn't take more than 200 msec, or overall performance will suffer"), properties of the current implementation ("The terminating terminal process is already close to its memory limitation, so you can't add much to it"), impact of design decisions on other aspects of the software ("If you modify the automatic testing process, you'll need to update the customer documentation"), local programming conventions ("Call the central error-reporting mechanism if your function gets a bad message"), and individual and organizational expertise ("Ask Nancy about that; she knows about local stack space"). This kind of knowledge usually is not written down; rather, it is maintained and disseminated informally by experienced individuals.

This form of knowledge maintenance and dissemination is unsatisfactory for many reasons. First, not only are experts difficult to locate when needed, but individuals also must know who the experts are for their particular problems. Studies in this organization have shown that successful developers are those who have effective "expertise networks" and thus know who to ask about particular problems. This valuable knowledge is a scarce resource that is acquired only through experience (Curtis, Krasner, & Iscoe, 1988). Second, experts can spend more time disseminating knowledge than applying their expertise to developing software–laboring under a sort of expertise tax. Third, knowledge often is generated (e.g., in design, review, testing, or fault analysis) only to be lost, thus depriving the organization of a valuable resource and leading to potential duplication of effort in the future. Fourth, because key knowledge often is known only to a few individuals, loss of personnel can mean loss of knowledge. Poor management and delivery of design knowledge can cause suboptimal designs, late and costly detection of errors, long delivery times, and personal frustration.

ISCBU has taken various steps to improve its software development process, including instituting various quality initiatives (Colson & Prell, 1992). One aspect of this approach is the precise description of software processes in terms of sequences of steps, suppliers and customers, and inputs and outputs (as in the design process shown in Figure 1). The process movement is relevant to our project because any technological solution to the problem of managing design knowledge will be deployed in the context of the existing design process and must be appropriately integrated with the process. Another benefit for our project is that technical people are empowered to improve their processes, and Long was the technical leader of the design process management team.

We note one caveat about defining organizational processes–the danger of overregimentation. Suchman's (1983) analysis of office work showed that procedures can be defined in too much detail and that such procedures are less help than hindrance to people in performing their work. ISCBU has moved away from its initial tendencies toward excessive regimentation. Processes define macro-scale patterns of coordination, milestones, and deliverables–the what, not the how. Designers have great discretion in how they accomplish their work; for example, no single design methodology or computer-aided software engineering (CASE) tool is mandated.

ISCBU tried explicitly to address its knowledge management problem by documenting knowledge in structured text files. Even if all relevant facts could be captured in this manner, this approach still is inadequate for three reasons:

- The documents are not organized for efficient access. Without adequate indexing, the resulting information base is simply too large to be very useful (busy people, including software developers, will not read large documents that are not immediately relevant to their current task).

- There is no way to ensure compliance. That is, it is impossible to be sure that developers and reviewers have consulted all the information that is relevant for a particular design problem.
- There is no natural way to ensure evolution of the documents. Documents will become incomplete and incorrect over time, and the programming constructs, requirements, constraints, and methodologies they describe all will evolve.

From our perspective, the crux of the problem is that the on-line documents are not a living design memory. They are not well integrated into organizational practice and do not address how knowledge is to be used and changed.

We can build on the preceding discussion to state some requirements for a design memory tool:

- The pragmatics of tool use must be specified, including the points in the design process at which it is to be used and the purposes for which it is intended.
- Designers must be able to access task-relevant knowledge efficiently.
- Designers must be able to apply the knowledge they get from the tool, incorporating it into their designs easily and quickly.
- The organizational processes should be structured to encourage tool use and check whether the tool was used and the advice followed.
- Additions and modifications to the advice need to be captured. Advice never is complete, circumstances of the organization change, and new knowledge is generated.

3. A FRAMEWORK FOR LIVING DESIGN MEMORY

We have developed a framework for integrating a design memory tool into a software development process that addresses the requirements just stated. The framework includes both technical and organizational aspects in a tightly interwoven manner.

Technically, our framework specifies a design knowledge tool, following the paradigm of interactive assistance for software development (Rich & Waters, 1990). The tool has two components–a *design knowledge base* that records relevant information and an *interface* that provides access to the knowledge base. Such a tool (ours is DA) must be designed by researchers, domain experts, and potential users working together. In our framework, we assume that the design memory tool augments an existing process that uses informal design artifacts (i.e., text documents). Therefore, the tool provides textual advice to developers, and it is their responsibility to modify their designs in accordance with the advice or to explain why the advice does not apply to their designs. Attempts to formalize design artifacts through the use of knowledge-based tools (Bailin, Moore, Bentz,

& Bewtra, 1990; Johnson, Feather, & Harris, 1991; Mark, Tyler, McGuire, & Schlossberg, 1992; Ramesh & Dhar, 1991) are complementary to our approach.

One might assume that a complete knowledge base can be built up-front, before system use begins. Although certainly the knowledge base must contain some information before system use can begin, we claim that in a real-world, large-scale design memory, knowledge engineering must occur throughout the life span of the memory. There are three reasons we make this claim. First, recent work in both social science (Lave, 1988; Suchman, 1987) and artificial intelligence (AI; Clancey, 1991; Gaines, 1989; Winograd & Flores, 1986) has argued that the "knowledge" encoded in AI knowledge bases is at best a partial, finite rendition of human knowledge. Thus, a knowledge base is always subject to additional refinement and reinterpretation, and users of a knowledge base might need to contact the builders of the knowledge base to understand fully the information it contains. Second, the circumstances of the software organization that are modeled in the knowledge base change. The software base changes–indeed, this is the goal of the design activity–as new customer requirements are met. Hardware and software technology advances. Faults are observed in the running software and must be fixed. All the other assumptions and constraints are subject to continual, if slow, evolution. Third, different knowledge domains will "mature" at different times; for example, the architecture team might be ready to encode its knowledge a year before the database team is. Large-scale, real-world knowledge engineering is inherently incremental (Shipman, 1993; Shipman & McCall 1994).

In response to these issues, we extend our framework in three ways. First, it must support two types of "knowledge evolution"–*knowledge update,* which is the elaboration and evolution of the design knowledge base as the tool is used and evaluated, and *knowledge addition,* which is the addition of new knowledge generated during development activities and as new domains mature. Second, computer assistance must be backed up with human collaboration; when developers access knowledge in the design memory, it should be easy for them to contact the people responsible for encoding that knowledge.

We consider the second point first. We take a simple approach to integrating computer assistance and human collaboration, based on the notion of knowledge "ownership." We require that every piece of information in the knowledge base be tagged with its *owner*–the individual who articulated the information and who is responsible for it being in the knowledge base. Then, when advice is delivered to a developer, the developer can find out the advice owner, including e-mail address and phone number. If developers do not understand the advice or need more information, they are directed to the person best able to help. This begins to turn important networking knowledge–which currently is a scare, informally managed resource–into an organizational asset. Notice that the ownership relationship is itself knowledge that must evolve over time as people leave the organization or their responsibilities shift. We next describe techniques that can be used to maintain this and other types of knowledge.

Our first step to support knowledge evolution is to add a *KB* [knowledge base] *update-and-maintenance* activity to our framework. This process institutionalizes knowledge engineering as an ongoing activity in the design process. It takes as input requests and suggestions for modifications or additions to the knowledge base, and it produces changes or updates to the design knowledge base in response. The person or persons who perform this process are *DA knowledge engineers*. They do their work in close collaboration with developers, domain experts, and customers.

We take several steps to support knowledge update. First, DA collects comments from users at the end of each session. Second, DA automatically produces a trace of each session, and designers annotate their design documents to include the trace. This allows those aspects of the design that were influenced by the advice to be commented on (and disagreed with) during design review, thus making the advice itself an object of review. This also means that, if designers choose not to follow advice, they must explain why not. Their reasoning can be judged and, if valid, will lead to capturing exceptions to or modifications of existing advice. Notice that traces do not become part of the permanent knowledge base; rather, examining the traces, which record developers' interactions with DA, might lead to changes to the knowledge base. To sum up, designers and reviewers encounter the knowledge encoded in DA in the context of actual design situations. These situations are likely to trigger tacit knowledge that might modify or extend the information in DA. We provide several means by which designers and reviewers can articulate this information and input it to the KB update-and-maintenance process.

Next we consider how to support the addition of new knowledge to DA. We must first define potential sources of new knowledge. So far, we have identified the following:

- *Fault analysis*. When a fault is observed in the running software system, and it is determined that the fault was due to a design error, *root cause analysis* is performed to determine the underlying cause of the fault; this information then can be encoded in DA so that the situation that led to the fault can be detected and the fault avoided in the future.
- *Process improvement*. As part of the ongoing quality efforts, the design process management team continuously analyzes data about the process; they can identify "opportunities for improvement," which can be encoded as information in DA.

- *Expert initiative.* As already mentioned, knowledge acquisition in a large organization is inherently incremental; as experts in a particular domain articulate their knowledge to a suitable point, or as they become aware that DA is an appropriate mechanism for communicating their knowledge, they can initiate the process of engineering their domain.
- *Customer impact.* Downstream processes that are design customers can identify effects that design decisions have on their work (e.g., certain types of design decisions can lead to difficulties for software testing); these customers can initiate a knowledge engineering process to encode these effects.

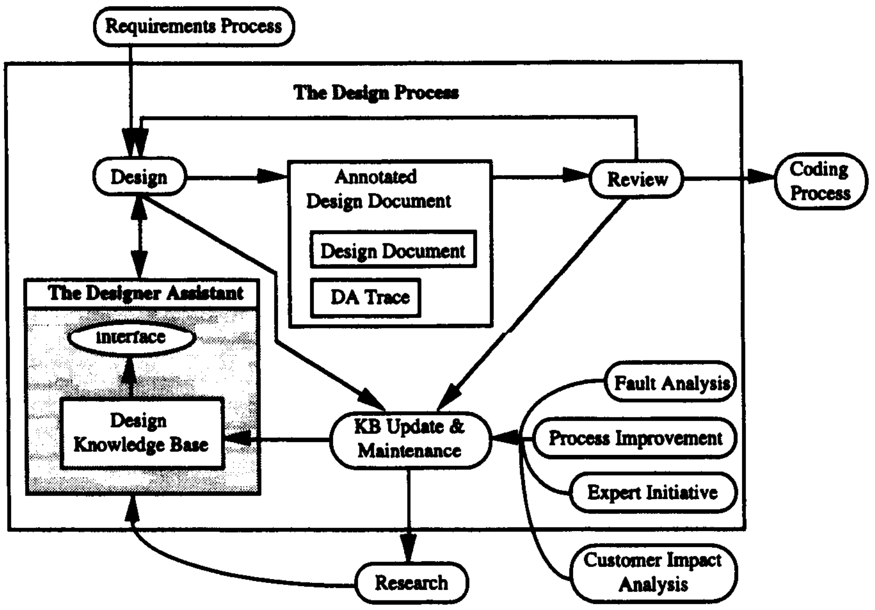
In any of these cases, knowledge evolution can be triggered through informal communication with the DA knowledge engineer or by the use of standard process improvement mechanisms such as *quality improvement projects* or *opportunities for improvement* (two local mechanisms for continually improving processes). Again, we find that exploiting mechanisms and procedures that are familiar to developers in the organization helps increase the likelihood that knowledge evolution will occur, making the design memory living. The entire KB update-and-maintenance process is described precisely in an ISO 9000–compliant process description document.

Finally, we learned that a living system can lead to a living research project. Although the KB update-and-maintenance process takes care of updates to the contents of the design knowledge base, use of DA can reveal shortcomings in the structure of the knowledge base or in the means by which the knowledge is accessed. New research is then required to address these problems, leading to a modified representational framework and interface.

Figure 2 shows the complete living design memory framework.

We conclude this section by summarizing our framework. It is useful to note that it does not commit us to how information is to be represented or what sorts of information access and interface techniques should be used. Rather, it makes clear how the technological aspects of a design memory must be integrated with and extend existing organizational processes. The framework specifies the point at which DA is to be consulted and the components of DA (a knowledge base of information relevant to the design task and an interface that helps designers access relevant knowledge). The framework specifies a KB update-and-maintenance process that provides for ongoing knowledge engineering, thus making the design memory living. Developer and reviewer comments organized around an *annotated design document* suggest updates to existing knowledge. Fault analysis, process improvement opportunities, expert initiative, and customer impact analysis lead to the encoding of new knowledge. All the

Figure 2. **Living design memory framework–ISCBU design process with DA.**



various types of feedback and the means by which feedback can be communicated are specified in a formal description of the KB update-and-maintenance process. DA is embedded in organizational practice so that knowledge is captured in a public repository, is disseminated effectively, and evolves as necessary.

4. THE DESIGNER ASSISTANT–A DESIGN MEMORY TOOL

Our design memory tool, DA, contains significant folklore knowledge about design in ISCBU. The initial "seed" knowledge base of DA has grown to include about 10 main domains and the results of several fault analyses and has evolved in response to many user comments. DA guides developers through a dialogue concerning characteristics of their design and provides advice based on the answers. DA is integrated into the development process, and all levels of the organization, from high-level management to developers, view the tool favorably. Work continues on engineering new domains, and user feedback constantly is received. In short, DA is living. In this section, we trace the evolution of the tool, give a technical description of its current state, and describe how its knowledge has evolved.

4.1. Evolving a Prototype

One important factor in both the design and deployment phases of our project was Long's organizational standing. He was an experienced developer and the engineer responsible for the design process management team. This ensured our access to domain experts, potential users, and management.

First we identified a knowledge domain in which to construct a prototype. We selected an error-handling mechanism, "asserts," that is critical to the fault tolerance of the software system. An *assert* is used to signal an illegal state and is implemented as a macro call with a number of arguments that have various effects on the system. For example, one value of one argument initializes the processor running the current process. Other arguments cause the dumping of different kinds of data needed to diagnose the problem or schedule a data-checking audit. Thus, a developer has to make a series of decisions in order to use an assert appropriately. Some of these decisions are quite complicated, and there are dependencies among the decisions.

Although limited, this domain still has the following important features. First, it is a difficult domain. Developers typically do not know when to use the mechanism, how to use it, or even how to find out about it. This is especially true of novices in the organization, but even experienced developers commonly misuse the construct (in fact, many of the existing uses in the code base are incorrect). Second, there are local experts who have extensive knowledge about asserts, and this knowledge is managed as folklore. The experts disseminate this information in a frustrating and inefficient manner (i.e., one-to-one communication with individual developers). Third, attempts to document asserts were unsuccessful due to the compliance and indexing problems mentioned earlier. The asserts domain is typical of design knowledge in ISCBU, so we believed that focusing on it in the prototyping process would allow us to address general issues in design knowledge management.

After choosing the domain, we spent dozens of hours interviewing domain experts and studying the existing written documentation. We took a set of several hundred examples of asserts from the code base and asked the experts to categorize the examples in terms of ***design attributes,*** the design features to which all the uses of the mechanism in each category responded. The experts did so in two stages, first sorting the examples into categories, then articulating explicitly what each category had in common.[1] This was a very important abstraction step because it meant that the

1. We are grateful to Mike Wish of AT&T Bell Laboratories for first suggesting this approach.

tool interaction could use terms familiar to developers (the design attributes) rather than refer to syntactic features of the construct. Presumably, developers would not be familiar with the latter vocabulary, as it is precisely this for which they are getting help.

After several iterations, the experts succeeded in generating a small number of design attributes with almost complete domain coverage. Each attribute could be expressed as a yes–no question (e.g., "Does your design update data in the database?"). Attributes were arranged in a generalization hierarchy, with more general attributes subsuming more specific attributes. Thus, under the previous question might be the additional question, "Is it possible for dynamic and static data to become desynchronized?" In other words, the attributes (and corresponding questions) were arranged in a decision-tree structure. The next, crucial step was to elicit advice from the domain experts about how to use the assert mechanism in the situation covered by each design attribute. Some advice might apply only to very specific design situations (the leaf nodes in the attribute hierarchy), whereas other advice could apply to more general situations (interior nodes). The advice was distilled into small units of text that we call *advice items.* Indexing by design attribute has proved to be a very useful way of acquiring and organizing knowledge in this and other domains. (Selfridge, Terveen, & Long, 1992, gave details about the representation of design attributes and advice items in the prototype.)

The next task was to construct a prototype design memory tool. We used the language CLASSIC (Borgida, Brachman, McGuinness, & Resnick, 1989; Brachman, McGuinness, Patel-Schneider, Resnick, & Borgida, 1990) to represent the information we had acquired and developed a simple dialogue-based interface. Because of the wide variety of terminals used within the target organization, the tool had to be ASCII based and independent of any specific window system or platform. The prototype simply used basic C input and output routines. The interaction consisted of the tool asking yes–no questions to guide a developer down the design attribute hierarchy (an interactive classification task); when a developer responded "yes" to a leaf node in the attribute hierarchy, the system presented several paragraphs of advice about how to use the error-handling mechanism for this situation. The advice was computed by collecting the advice items associated with the leaf attribute description and all generalizations of that attribute description. (In addition, several mechanisms for overriding and ordering advice were applied; Selfridge et al., 1992.) One advantage of having the system direct the dialogue was that it increased the chances of compliance (i.e., that designers would be exposed to all relevant information). When the interaction was complete, the tool asked several evaluation questions (e.g., "Was this session useful?" "Was the level of detail *about right, too much,* or *too little?*") and gave the user a chance to enter more detailed comments and suggestions. The technical details of information representation and dialogue management are slightly different in the current implementation of DA. We present the details of the current implementation in Section 4.3.

To summarize, the results of the prototype process were:

- The representational framework organizes information in hierarchies of design attributes.
- This framework serves as a useful organizing principle for engineering new domains. Experts can be guided to formulate the key design attributes that organize their domain and the advice that should be delivered for the situation covered by each attribute.
- The framework suggests an interaction technique–organizing dialogue as interactive classification, in which designers (a) categorize their design situations by answering questions posed in terms that are likely to make sense to them and (b) receive advice about specialized topics.

4.2. From Prototype to Deployed Tool

We next carried out informal user tests on the prototype. We created a realistic software design problem and asked half a dozen developers to write a design section involving the error-handling mechanism. They used the tool to get advice on how to write the design section. Their reaction was highly favorable; in some cases, designers asserted that the 20 min spent using the tool saved from 4 to 8 hr of their time. The reason for this is that the only other way to access the knowledge presented by the tool would have been to track down the local experts or search through large documents, both notoriously time-consuming activities.

Considering these results very positive, we deployed DA in the design process. That is, we modified the formal process description to include the steps in "Run the Designer Assistant and append the resulting transcript to your design document." We sent e-mail to all the developers in the organization announcing the availability of the tool. Within 15 days, more than 85 developers used DA. Their feedback was unexpectedly negative. Only 38% judged the interaction useful, and only 42% said that the level of detail of information was ***about right***. These ratings and detailed comments from the users indicated that DA had two major problems. First, it did not contain enough knowledge–running a tool that contained information about just one small domain was more trouble than it was worth. Second, the interface did not match user expectations–for example, it gave no guidance in entering legal answers, and the overall structure of the dialogue was unclear.

Up until that point in the project, our team's research members, Terveen and Selfridge, worked on designing and implementing the prototype. However, responsibility for the tool was shifting to Long, who faced the problems of improving the interface and adding more knowledge. To do so, he reimplemented DA in an internal AT&T system development environment, the Data Collection System (DCS). DCS is a tool for constructing screen-based, menu-driven interfaces. DCS provides much functionality that helped address user complaints, including input type checking, ability to revisit and reanswer any previous question, text- and cursor-based menus, automatic recording of user responses, and output of a transcript in a convenient database format. The reimplementation illustrates several interesting points:

- Long now "owned" the reimplemented DA–it was a facile tool for him. This meant that he could maintain both the behavior of the tool and the contents of the knowledge base easily. To further ease the task of adding new knowledge, Long designed a simple rule language tailored for representing hierarchies of design attributes and a compiler that produces DCS code from the rules.
- DCS was a known resource in the repertoire of the developers but not the researchers. We learned that making a new tool meet user expectations is facilitated by exploiting the system development environments available in the target user community. More generally, this highlighted for us that a research–development partnership involves mutual learning (Greenbaum & Kyng, 1991) and that the details of the learning required might be impossible to anticipate.
- Changing the underlying technology from CLASSIC to DCS involved some trade-offs. Gains in ease of interaction and ownership by Long came at the expense of great loss of representational power. However, the CLASSIC prototype focused on elegant representation and efficient computation of advice, whereas experience showed that it was more important to be able to manage structured hierarchical dialogues effectively, and DCS was much better at this. This illustrates another general lesson. Any design project inevitably involves trade-offs. Those members of the design team who actually implement the design have great influence in deciding how the trade-offs are resolved (Ehn & Kyng, 1991). Research and development members of a joint design team might well judge how to resolve particular trade-offs differently. Thus, trade-offs that have not been explicitly articulated and discussed might well become the occasion for redesign or reimplementation as ownership of a system shifts from researchers to developers.
- The reimplementation shed new light on what had been accomplished thus far in the project. We came to see the CLASSIC prototype as a "running specification" of the behavior for an adequate tool. In a subsequent partnership undertaken by Terveen and Selfridge with ISCBU (Terveen & Selfridge, 1994), the participants agreed up-front that a running specification would be the result.

- Our view of our partnership and our work changed somewhat–we came to see that we had engaged in design after use. The difficulty of testing new technology under conditions that approximate the everyday working conditions of a large software development organization was highlighted. So, we will plan for some amount of redesign in future partnerships. This gives us more reason to reject the technology transfer metaphor–we see ourselves engaged in a living research project paradigm, in which actual system use reveals new problems.

When the reimplemented DA was deployed, complaints about the interface decreased significantly, and user feedback began to improve (see Figure 10). In the remainder of this article, we discuss the current implementation–its management, cost, status, and evaluation.

4.3. Technical Description of DA

To this point, we have concentrated on tracing the development history of DA and pointing out lessons we have learned along the way. We now present a more detailed technical description of DA in its current form.

The representation of information in DA can be seen either as an extended form of decision trees or as a simplified version of production rules. We use the first perspective to explain the representation. Each design attribute describes a category of design situations and can be expressed as either a yes–no or multiple-choice question. Each answer can have an associated piece of advice[2] and can lead to another design attribute. Figure 3a illustrates a design attribute with two values. The prototypical case is a yes–no question. We use one other construct to form dialogues, iterate (Figure 3b). An iterate construct forms a sequence of design attributes for presentation. Last, we note that information in general is structured not as trees but as directed acyclic graphs. We use the term *shared attribute* (or *shared question*) to refer to an attribute that has more than one parent.

DA interaction consists of the system guiding the user down the design attribute hierarchy. For an iterate node, the system visits each child in left-to-right, depth-first order. For all other design attributes, the system presents the attribute's question and possible answers and then receives the user's answer. Any advice associated with that answer is given, and then the dialogue continues down the children of the selected answer.

2. In the currently deployed version of DA, advice consists only of text. However, in our new research prototype (to be discussed), the notion of "advice" is expanded to include arbitrary system actions (e.g., sending e-mail, starting execution of a tool, and displaying graphics or images).

Figure 3. **Basics elements of DA information representation.**

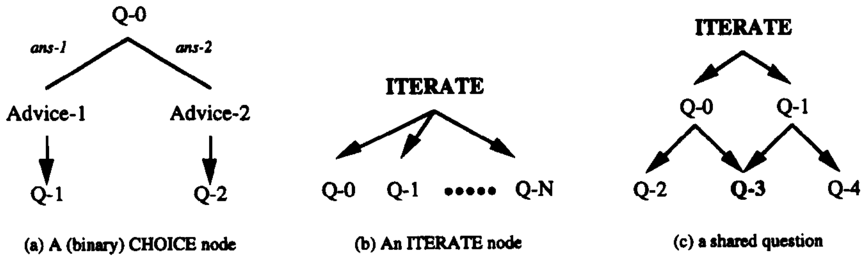
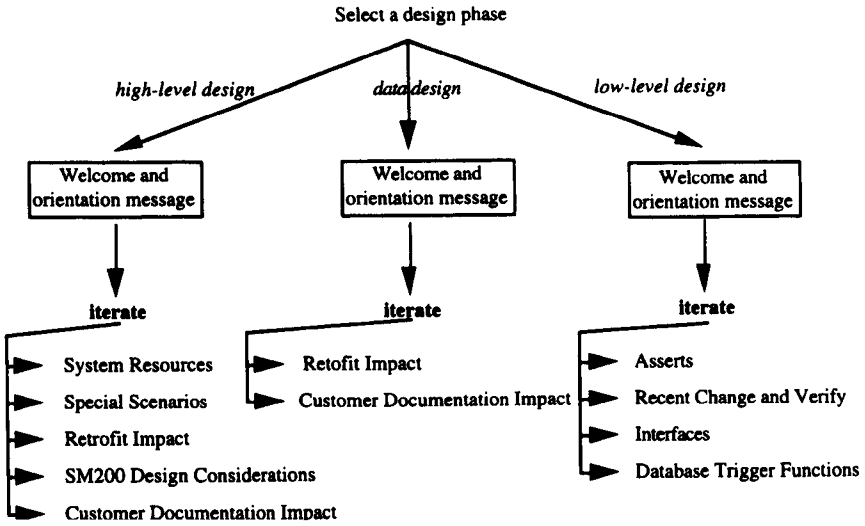


Figure 4. **Top-level structure of DA knowledge base.**


Select a design phase
high-level design
data design
low-level design
Welcome and orientation message
Welcome and orientation message
Welcome and orientation message
iterate
iterate
iterate
System Resources
Special Scenarios
Retrofit Impact
SM200 Design Considerations
Customer Documentation Impact
Retofit Impact
Customer Documentation Impact
Asserts
Recent Change and Verify
Interfaces
Database Trigger Functions

When a shared question is encountered a subsequent time, the system uses the previous user answer to that question. There are several features for controlling the dialogue, including ***recapping,*** which is viewing a list of all the questions and answers in the session and going back to any point and revising the answer, and ***backing up,*** which is returning to the previous question and revising the answer.

The top-level division of knowledge in DA is by the type of design in which the developer is currently engaged. Then, a series of relatively independent domains is represented. Figure 4 gives an overview of the top-level structure of the knowledge base.

Figure 5 shows the system resource domain (under the high-level design phase) in more detail, giving some idea of the size and complexity of a typical domain. (Figure 5 still simplifies the domain slightly for ease of presentation; e.g., only a summary of each piece of advice, not the actual text, is presented.) The system resource domain contains 16 design attributes, with a maximum depth of 4, and 6 pieces of advice. The terms in

Figure 5. **Example DA knowledge domain–system resources.**

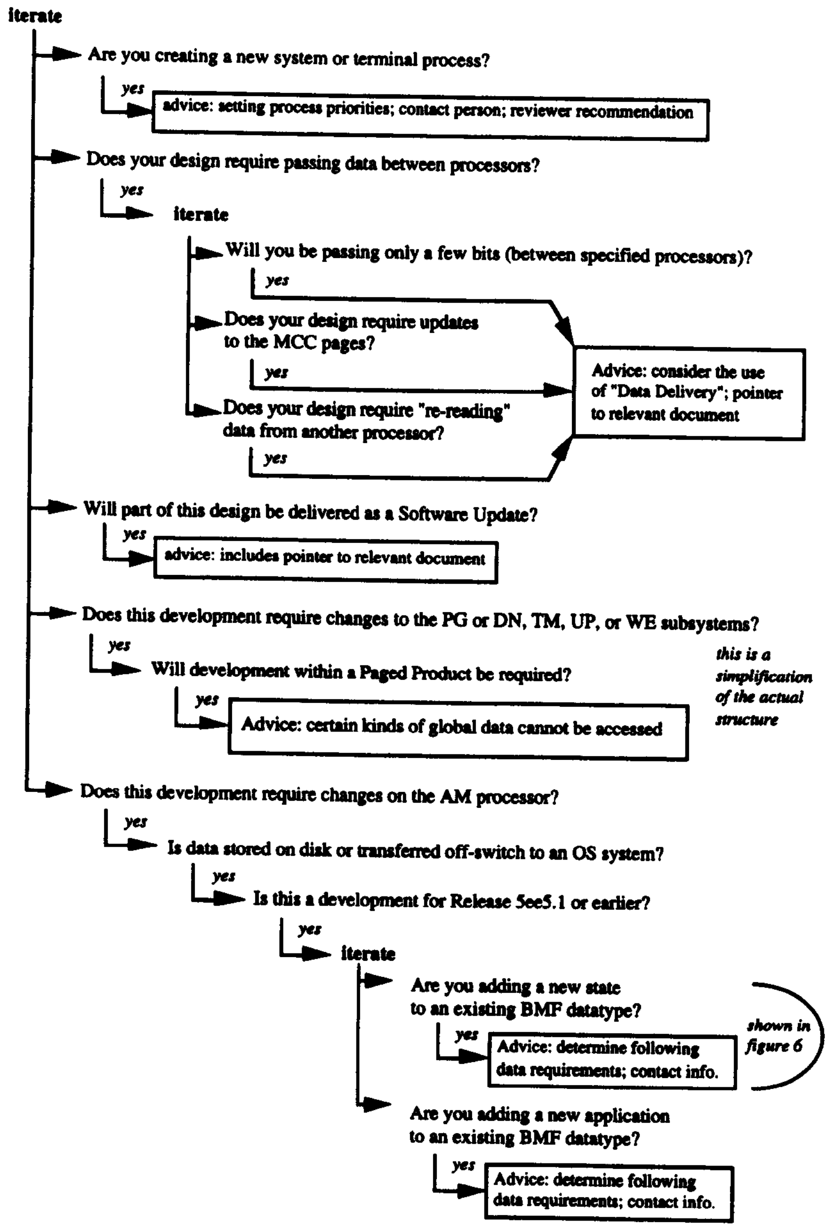
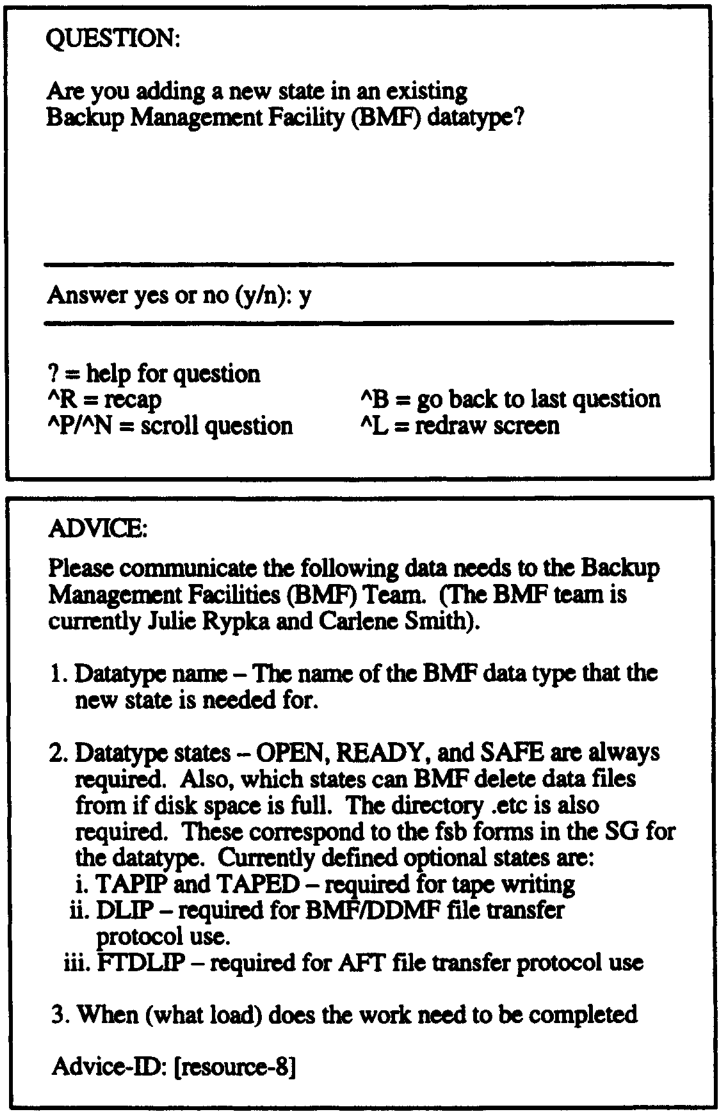

iterate
Are you creating a new system or terminal process?
yes
advice: setting process priorities; contact person; reviewer recommendation
Does your design require passing data between processors?
yes
iterate
Will you be passing only a few bits (between specified processors)?
yes
Does your design require updates to the MCC pages?
yes
Does your design require "re-reading" data from another processor?
yes
Advice: consider the use of "Data Delivery"; pointer to relevant document
Will part of this design be delivered as a Software Update?
yes
advice: includes pointer to relevant document
Does this development require changes to the PG or DN, TM, UP, or WE subsystems?
yes
Will development within a Paged Product be required?
yes
Advice: certain kinds of global data cannot be accessed
this is a simplification of the actual structure
Does this development require changes on the AM processor?
yes
Is data stored on disk or transferred off-switch to an OS system?
yes
Is this a development for Release 5ee5.1 or earlier?
yes
iterate
Are you adding a new state to an existing BMF datatype?
yes
Advice: determine following data requirements; contact info.
shown in figure 6
Are you adding a new application to an existing BMF datatype?
yes
Advice: determine following data requirements; contact info.

Figure 6. **Example DA question and advice.**



QUESTION:

Are you adding a new state in an existing
Backup Management Facility (BMF) datatype?

Answer yes or no (y/n): y

? = help for question
^R = recap ^B = go back to last question
^P/^N = scroll question ^L = redraw screen

ADVICE:

Please communicate the following data needs to the Backup Management Facilities (BMF) Team. (The BMF team is currently Julie Rypka and Carlene Smith).

1. Datatype name – The name of the BMF data type that the new state is needed for.

2. Datatype states – OPEN, READY, and SAFE are always required. Also, which states can BMF delete data files from if disk space is full. The directory .etc is also required. These correspond to the fsb forms in the SG for the datatype. Currently defined optional states are:
   i. TAPIP and TAPED – required for tape writing
   ii. DLIP – required for BMF/DDMF file transfer protocol use.
   iii. FTDLIP – required for AFT file transfer protocol use

3. When (what load) does the work need to be completed

Advice-ID: [resource-8]

the questions (e.g., *software update, PG or DN subsystem,* and *the AM processor*) are familiar to ISCBU developers (however, the wording of the questions has evolved in response to user feedback). Advice is a combination of specific information (e.g., that in certain design situations, certain types of global data cannot be accessed) and pointers to documents and individuals. Figure 6 illustrates the appearance of DA screens and shows a system question and the advice that is presented when a user answers "yes" to the question.

4.4. How DA Knowledge Has Evolved

We now consider in a bit more detail how the knowledge evolution process has worked in practice. To limit the scope of the discussion, we summarize feedback received between two relatively mature DA versions. We also present a few specific examples of feedback.

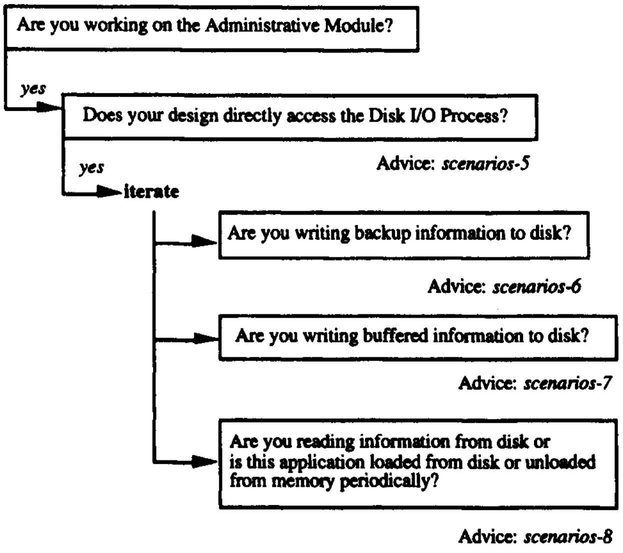
The KB update-and-maintenance team identified more than 25 issues that came either from DA users or from experts who suggested new knowledge domains to be encoded. Of these issues, 10 were rated as the highest priority. These issues ranged from extremely general ("Improve help messages" and "Address how to use the tool for a design with multiple authors") to very specific ("Clarify ambiguous use of the term *data*" and "Update the members of the GFS/BMF team"). Of the high-priority issues, 8 came from DA user feedback, and 2 came from domain experts, who wrote "opportunities for improvement" that proposed knowledge they thought should be included in DA. Some issues required adding new knowledge to DA; others required modification of existing knowledge.

Knowledge in DA can be divided into three categories: *expert knowledge,* specialized areas of design knowledge with which most designers are unfamiliar, like asserts; *impact knowledge,* how characteristics of a design affect another area of the software or another process; and *fault prevention knowledge,* how characteristics of a design could lead to a fault and how the fault can be avoided. This division is motivated by differences in how the knowledge is acquired–the different types of knowledge are used in the same way. Expert knowledge is acquired from design process subject matter experts, impact knowledge from downstream customers of design, and fault prevention knowledge from root cause analysis of software faults.

One expert domain that was added was *DIOP* [disk input/output process] *scenarios.* Considering the process by which this domain was added nicely illustrates the knowledge evolution process. First, several developers wrote an "opportunity for improvement" specifying that information on this domain should be added. Next, the developers met with the DA knowledge engineer to discuss how the domain should be encoded. The DA knowledge engineer then proposed a representation of the domain. The representation specified the location of the domain in the existing knowledge base (i.e., under which design attribute its root design attribute should be placed). The structure of the DIOP scenarios domain was roughly as shown in Figure 7. (We only indicate where advice is delivered; we do not show the actual advice. Each piece of advice consists of one or two sentences.)

The DA knowledge engineer communicated this structure to the domain experts and assigned one of them to be the owner of each advice

Figure 7. **DIOP scenarios domain.**


Are you working on the Administrative Module?
yes
Does your design directly access the Disk I/O Process?
yes
Advice: *scenarios-5*
iterate
Are you writing backup information to disk?
Advice: *scenarios-6*
Are you writing buffered information to disk?
Advice: *scenarios-7*
Are you reading information from disk or is this application loaded from disk or unloaded from memory periodically?
Advice: *scenarios-8*

item. The advice owner responded by suggesting some clarifications in the language used in the questions and advice, pointing out several modifications to the content of the advice, and identifying some new information to be communicated. Finally, the revised knowledge was encoded in DA.

This example illustrates important properties of the DA approach. First, knowledge engineering has been institutionalized as an ongoing part of the design process, and experts are willing to work on engineering their domains. Second, knowledge engineering is a collaborative process between the DA knowledge engineer and domain experts. Third, it is important that the representation of advice used in DA is relatively simple (e.g., can be communicated in e-mail messages in the form of nested if–then statements). This makes it a good medium for communication between domain experts and the DA knowledge engineer–they understand easily what advice means and how to modify and add to it.

5. RELATED WORK

In this section, we compare our living design memory approach to other related work, bringing out important characteristics of our framework by contrast. First, AI and expert systems work also is aimed at capturing knowledge in particular domains. These systems contain a knowledge base and a reasoning component that computes the desired inferences–for example, relating patient symptoms to disease classifications (Shortliffe, 1976) or customer computer orders to a configuration diagram (McDermott, 1982). The original goal of expert systems work was to produce automated systems. However, automation requires a complete formalization of knowledge in the target domain, which is rarely possible. Thus, most existing expert systems function as interactive assistants. This change of emphasis also is true for the application of AI to software design. The early focus was on automatic programming (Barstow, 1979). Despite some interesting successes (Kant, Daube, MacGregor, & Wald, 1991; Smith, 1991), the field of knowledge-based software engineering has shifted its emphasis to creating systems that assist people in designing software (see the annual Knowledge-Based Software Engineering Conference); this trend also appears in AI approaches to design in general (e.g., the annual International Conference on AI in Design). Our work is part of this trend. We also have exploited the power of delivering informal information like "Ask Nancy about that; she knows about local stack space." This sort of information helps to reduce the high communications overhead in large organizations by making some information available through DA and guiding developers directly to the relevant expert when additional communication is required. We also have integrated the use of knowledge into an organizational process by, for example, specifying points in the process when DA should be consulted, defining feedback sources, and institutionalizing knowledge engineering in the design process.

Second, our framework responds to many of the points made by Curtis et al. (1988) in their discussion of large software development projects. Curtis et al. stated that software development must be seen as a learning and communication process; they recommended that software development tools facilitate the enterprise-wide sharing of knowledge, accommodate change as a normal and expected process, and serve as media of communication for integrating people and information. We identified a particular type of knowledge to be managed, developed a tool for managing and a heuristic for acquiring it, and constructed a framework for integrating the tool into the software development process.

Software process modeling research (Curtis, Kellner, & Over, 1992; Krasner, Terrel, Linehan, Arnold, & Ett, 1992) has addressed some of the issues raised by Curtis et al. (1988). Processes are represented formally, often through the use of a process modeling environment. After a process is represented, automated support can be provided for carrying it out, for keeping track of progress toward completion, and for analyzing the process for potential improvements. This work is complementary to ours. An organizational process can be improved both by formal representation and computational support for enactment (as in the process modeling approach) and by better management of the knowledge needed to perform the process (as in the DA/organizational memory approach). In addition, because introducing an organizational memory system might require changes to a process, an explicit representation of the process can make such changes easier.

We devote the remainder of this section to comparing our work to other approaches to organizational memory. One idea that has been the focus of much research attention is design rationale (Bailin et al., 1990; Conklin & Burgess Yakemovic, 1991; Fischer, Lemke, McCall, & Morch, 1991; Franke, 1991; Kunz & Rittel, 1970; Ramesh & Dhar 1991). *Design rationale* captures the reasons behind the design, including issues that were considered, alternative resolutions of these issues, and arguments for and against the different resolutions. Many approaches to capturing and using design rationale exist, ranging from very formal AI-type representations (Franke, 1991; Ramesh & Dhar 1991) to relatively informal text or hypertext representations (Conklin & Burgess Yakemovic, 1991). Work reported by Conklin and Burgess Yakemovic (1991) in applying the IBIS (Kunz & Rittel, 1970) methodology to a software development project is most relevant to the concerns of the present article. Conklin and Burgess Yakemovic developed a simple textual form of IBIS (itIBIS) that was suitable for use on the technology of the development organization with which they were working. The development project successfully used this method to record and present information relevant to their evolving design. One important reason why the IBIS technology was effective was that it incrementally improved an existing task (i.e., participants in a design project already kept track of relevant information with handwritten notes) instead of creating a whole new task. The new technology offered some payoff with minimal cost of adoption and disruption of existing organizational practice.

We addressed the same general issue–how to develop a system for managing design knowledge that could be integrated into existing organizational practice. However, the folklore knowledge with which we were concerned is more general than design rationale, and it must be used and modified not primarily in one design project, but over time in many projects in the ISCBU organization. We focused on integrating DA not just into existing practice, but also into existing organizational processes, modifying these processes as necessary. Specifically, we instituted knowledge review and maintenance processes to ensure the evolution of the knowledge. We also share Conklin and Burgess Yakemovic's (1991) commitment to analyzing incentives for and obstacles to adopting new technology (Grudin, 1988). DA is quick and easy for developers to use–no training is required, and the average session lasts about 10 min–and it offers substantial benefits (e.g., developers do not have to read long documents, waiting time is reduced, and review meetings can be smaller and thus scheduled more quickly; reviewers can be satisfied more easily). Domain experts have a larger burden, because they are asked to work with the KB update-and-maintenance process to encode their domains. However, we have found that experts are eager to do so because it reduces the amount of time they spend consulting, lets them accomplish other work, and helps to ensure that design rules they think are important are communicated throughout the development community.

At Hewlett–Packard, Berlin, Jeffries, O'Day, Paepcke, and Wharton (1993) developed a group memory system called *TeamInfo,* containing information relevant to a small group of researchers and software developers. Information in TeamInfo consists of e-mail messages grouped into one or more categories. The categories were developed by the group and included one category for each of the group's two main projects and other categories such as events, people and projects, topic tracking, and "technology hacks." TeamInfo was seeded initially with messages from the individual mail folders of group members. Group members extend the memory by carbon-copying TeamInfo on an e-mail message. Messages are classified through a combination of methods–manual (the sender selects at least one category to which the message belongs) and automatic (the system does pattern matching on words and phrases that indicate particular categories). Messages also are organized in a conversational structure, with each message pointing to (a) the message to which it responded and (b) all messages that responded to it. Users retrieve information by browsing and querying the memory.

Berlin et al. (1993) offered an interesting analysis of problems that arise as people change from using individual filing (e.g., in their personal mail folders) to group memory. For example, "purists" want a small number of very general categories, whereas "proliferators" prefer a large number of very detailed categories. A related distinction involves whether people want to invest in more work at message storing time (giving a very accurate and detailed categorization of a message) or at retrieval time (constructing a very precise query and being ready to try different queries and browse through messages). Such differences in individual preferences can make the use of group memory difficult. For example, people who prefer to put more work into retrieval might not put as much into categorizing messages at storage time–frustrating people who want to find these messages but who prefer to put more work into initial categorization.

There are several notable differences between TeamInfo and DA. First, in TeamInfo, information access is user driven (users query and browse the information base); in DA, access is system guided (the system asks users questions and gives advice based on their answers). Each method has its pluses and minuses. We decided on system-guided dialogues for one main reason–to try to achieve compliance (i.e., to increase the chances that each designer will be exposed to relevant information). Although we remain convinced that system-guided dialogues are appropriate in some use situations–most important, in checking that completed designs do not violate any design norms–user feedback has shown that there are circumstances in which users need to be able to explore the information base (e.g., early in design, users sometimes want to find out all the information DA contains about a particular topic). Second, the representation of information in DA is more highly engineered. The DA knowledge engineer and domain experts work together to clarify the structure of a domain, selecting and arranging design attributes, determining understandable questions, and creating useful advice. Finally, evolution in TeamInfo is mostly an individual process (group members send messages to TeamInfo as they see fit); in DA, users, domain experts, customers, and the design process team all can ask specific questions or suggest specific facts to be added, propose new domains to be engineered, and identify problems and solutions to be captured. All such suggestions go through the KB update-and-maintenance process.

There have been several approaches to design memory organized around design artifacts–for example, by Bill Mark and colleagues at Lockheed (Mark et al., 1992) and by Gerhard Fischer and colleagues at the University of Colorado (Fischer et al., 1992). They offer *design environments* that include a library of existing design artifacts (formally represented) and construction mechanisms (e.g., a palette of tool objects) for building new designs. Prototype systems have been developed for software, network, and architectural design. Designers construct artifacts by reusing and modifying existing artifacts and adding new design components. Much power is gained because artifacts are represented formally: Systems typically can assist designers by maintaining consistency of the design, suggesting issues to consider, and helping to recover from breakdowns. Fischer's systems also provide design rationale, indexed by features of artifacts and the design situation.

Recently, Fischer and colleagues (Fischer et al., 1992; Fischer, McCall, Ostwald, Reeves, & Shipman, 1994) augmented their model to address the issue of the evolution of design knowledge. The key phases of their model are *seeding, evolution,* and *reseeding.* Initially, knowledge engineers work with domain experts to develop a design environment containing an initial "seed" of domain knowledge. Then, as designers use the system to design artifacts, they can add new information as they work (e.g., they can add notes that describe problems they encountered and how they solved them), which forms the process of evolution. This information almost certainly will be informal (e.g., textual notes) or perhaps semiformal (e.g., hypertext). Periodically, reseeding–the process in which knowledge engineers analyze and reorganize the knowledge base–will be necessary. The goal is to increase the formality of information added by designers (e.g., organizing textual information into a hypertext structure of issues, alternatives, and arguments or even creating rules that can be used to critique a design).

We view the design environment approach as complementary to ours (in fact, we have done similar work ourselves; Terveen & Selfridge, 1994; Terveen & Wroblewski, 1991). More specifically, the seeding–evolution–reseeding model is quite similar in spirit to our living design memory framework. However, there are several differences. First, in the ISCBU development organization, design artifacts are English documents; thus, an approach that required a formal representation of design artifacts was not immediately possible. Designers use DA to get information about their design situations, rather than to do design. However, we consider formalization of design artifacts as a promising area for future work, and our success to date increases the likelihood that this more radical innovation will be considered. Second, the folklore knowledge we have captured is a broader class of knowledge than that handled by these systems, and we deliver it in different ways. Third, we go beyond these approaches to focus on how any design memory must be integrated into organizational processes–we view the design memory and organizational processes as mutual resources that must be codesigned to ensure a design memory that is living. In particular, although previous proposals for supporting evolution have focused on the individual designer, in our approach, knowledge evolution is a formal organizational process–it is neither fully automated nor left up to individual designers. It is driven by feedback from users, experts, and customers, but it is carried out by the DA knowledge engineer. The motivation for this is our experience that systems must have "owners" (in our case, the KB update-and-maintenance group), because, if no one is responsible for the integrity of a system, it decays rapidly over time. And, because our system is in daily use in a development organization, quality control of information is crucial. Fourth, in contrast with Fischer's model, we cannot separate evolution and reseeding. The various forms of feedback to the KB update-and-maintenance process can be seen as evolutionary pressures. The DA knowledge engineer updates the design knowledge base in response to these pressures, in a (potentially) continuous process of reseeding. Finally, the ongoing involvement of research in the project is a kind of "meta-reseeding": In addition to the content and structure of the design knowledge base, the very representational framework and interface mechanisms might have to change based on user feedback.

The final group memory system we consider is Answer Garden (Ackerman, 1994; Ackerman & Malone, 1990). Answer Garden is intended to capture recurring questions and their answers in a central database, thus easing the task of information seeking, reducing the burden on experts, and turning knowledge into an organizational asset. The database is designed to grow "organically" in response to new user questions.

Answer Garden organizes information (in the prototype, about the X Window system) in a branching network of multiple-choice questions and answers. Users traverse the network by answering questions until they reach the specific question that they need answered, where they receive an answer. Answers might include text or graphical images, or they might be "active nodes" that cause a certain action to be performed at run time (e.g., querying a database). If users come to the end of a path and do not find their question, or they do not understand a question or answer, they can enter a new question. This question is routed automatically to the appropriate expert. When the expert answers the new question, the answer is sent to the user, and the network is updated with the new question and answer. Therefore, Answer Garden's database evolves directly in response to user questions. In addition, experts can analyze the history of users' interactions with the system to determine whether the network needs to be reorganized. For example, the authors (Ackerman, 1994; Ackerman & Malone, 1990) suggested that, if very long paths are traversed frequently, this might indicate that the network should be revised to place certain of the questions higher in the tree.

Answer Garden is very similar to DA in its motivation, techniques used, and accompanying analysis. We too are interested in capturing knowledge that is maintained and disseminated informally–in capturing folklore, as we call it. Our representation of design attribute hierarchies and our interaction style are quite similar to those of Answer Garden. We too are beginning to explore the use of "active nodes" to perform actions such as sending e-mail or initiating execution of a tool. Ackerman and Malone's (1990) analysis of incentives for using Answer Garden is similar in the issues it addresses and the answers it offers; for example, users might be able to access authoritative information at any time, experts will spend less time answering routine recurring questions, and organizations can hope for more effective and efficient information management and dissemination. Both projects also found that relatively simple technologies can offer significant organizational benefits, when combined properly and, as DA emphasizes, integrated appropriately with organizational processes. In a complicated, information-rich environment, simplicity and ease of use can be crucial to the acceptance of a system. If a system poses few burdens on its users (e.g., little training is required, documentation is unnecessary, and sessions are brief), it can offer relatively modest benefits and yet still have an attractive cost–benefit ratio.

There also are several differences between Answer Garden and DA. We note what seems to us the three most significant. First, knowledge in DA is somewhat more structured. Second, growth in Answer Garden is largely bottom-up (i.e., driven by user questions). Although this has the advantage that growth is in areas in which at least some users are interested, it has its limits in the world of DA. DA also must evolve as design faults are discovered and analyzed, experts become ready to engineer their domains, process improvement projects initiated by the design process management team identify new information to be communicated to designers, and downstream customers identify effects of design on their processes–effects of which designers must be made aware. Thus, user feedback is only one of the evolutionary forces to which DA responds. And the unit of evolution in DA tends to be much larger than a single question and answer (e.g., a whole new domain might be added). Finally, evolution in the Answer Garden is decentralized (each responsible expert can add his or her answer to a new user question), whereas updates to DA are handled by a central KB update-and-maintenance process. The DA knowledge engineer is responsible for working with users, experts, and customers to respond to their suggestions for modifications and updates to the knowledge base. As we have mentioned before, this seems necessary in order to try to preserve the integrity of the knowledge base. Further, because growth can occur in fairly large chunks, the DA knowledge engineer is responsible for managing each particular knowledge engineering project.

Figure 8 summarizes the points of comparison between DA and the most closely related systems and approaches.

6. DISCUSSION

6.1. Lessons Learned

The primary lesson from this work is that technology and organizational processes are mutual, complementary resources. We have come to see that many technological approaches to improving individual or group work ignore a powerful resource, organizational processes. An organization's internal design, patterns of coordination, information flow, and technology must be integrated into a coherent overall solution. We integrated DA into the ISCBU design process using well-known quality mechanisms. We use the existing design review process to ensure that information in DA is modified as necessary. We defined a new process, the KB update-and-maintenance process, to manage changes and additions to the DA knowledge base. This process is defined in a formal process description that specifies its customers, suppliers, inputs, and outputs. Existing quality mechanisms, such as "opportunities for improvement" and "quality improvement projects," are used to produce feedback to DA.

Now we restate the lessons appearing in Section 1 and, for each, summarize how the DA project illustrates the lesson:

- "The pragmatics of knowledge use are critical. Simply recording facts is not enough; issues such as where in the process knowledge is to be accessed, how to access relevant knowledge from a large information space, and how to allow for change also must be addressed for a knowledge management system to be successful."

  We identified particular points in the ISCBU design process at which DA should be used. The currently deployed DA is best used as an "after-the-fact" design checker with a system-guided dialogue structure appropriate. User feedback has indicated that user-directed exploration is more appropriate early in the design process, and our new research prototype supports this. The design attribute hierarchies that structure information in DA serve as an index that enables

Figure 8. **Comparison of DA and other organizational memory systems.**

System/Approach	System Purpose	Reported Use	Representation	Access	Evolution
DA	Deliver advice	Used by large development organization daily for almost 2 years	Design attribute hierarchies	Top-down traversal; browsing and search available in new prototype	Various feedback sources and mechanisms; evolution mediated by update-and-maintenance process
Answer Garden	Deliver advice	Used by several medium-size groups for a number of months	Branching network of questions and answers	Top-down traversal; "jumping" to any node	Answers to new user questions; expert analysis of network usage statistics
TeamInfo	Information system–store and retrieve e-mail messages	Used by small group for about 6 months	E-mail messages placed in group-defined categories	Browsing and querying	E-mail messages sent to TeamInfo; automatic and manual categorization
Fischer's systems	Design environment	Various prototypes	Formal artifacts-design rationale; critics	System delivers advice in context; users can browse and explore	Annotations during use; restructuring by knowledge engineers
Design rationale (itIBIS)	Capture design reasoning	Used by small group for 18-month project and for other, smaller projects	Structured hypertext; issues, alternatives, arguments, resolutions	Hypertext browsing	Project members raise new issues and deliberate them

designers to locate information relevant to their design tasks. The design attribute hierarchy representation also has proved very useful for ongoing knowledge acquisition, serving as a schema for the DA knowledge engineer and domain experts to formulate knowledge. It encourages the identification of important design states (i.e., those that require designer action) and the particular advice relevant in these states. We identified various sources of and mechanisms for feedback to DA and instituted a process to respond to the feedback. By having an "owner" for DA (the DA knowledge engineer) and for each information item (the "knowledge owners") we seek to ensure the integrity of knowledge in DA.

- "Addressing organizational problems while offering direct benefit to individuals is key. Although we focus on helping an organization manage its knowledge effectively–thus alleviating mainly problems that manifest themselves on an organizational level (e.g., excessive communication and coordination overhead, duplicated effort, long product delivery times)–we know that it is individuals who implement new practices and use new technology, and they need proper incentives to cooperate with such initiatives."

DA does offer benefits to ISCBU. It captures design knowledge that previously was managed and disseminated only informally, and it has the potential to reduce software faults due to design. Individuals also receive benefits from DA. Developers do not have to wade through large documents or spend time "blocked" while attempting to locate someone who can help them with their problem, and they can satisfy reviewers more easily if they show they have complied with DA advice (or explain why they have not). DA is fairly simple and quick to use; this is an important benefit in an already highly complicated, information-rich work environment. It also means that a fairly modest amount of usefulness is required for DA to have a favorable cost–benefit ratio. Experts can reduce the amount of time they spend answering routine questions and can ensure that important design norms are disseminated throughout the organization.

- "Computer information delivery and computer mediation of human collaboration must be tightly interwoven. Our approach integrates the perspectives of *cooperative problem solving* . . . , in which a computer system assists a person in performing a task, and *computer-supported cooperative work,* in which computer technology is used to mediate collaboration among humans."

DA captures expertise that is communicated to designers. It also records the owner for each piece of advice; thus, designers can be referred to specific experts when they have additional questions. In some cases, advice consists only of pointers to appropriate experts. Although not sufficient in the long term, this itself is a significant improvement. First, it turns valuable and scarce networking knowledge into a public resource. It can decrease the total amount (and thus cost) of communication while focusing and making more efficient the communication that still is required. Second, it offers a path to formalization of expertise: A next logical step for experts to whom DA advice refers is to work with the DA knowledge engineer to engineer their expertise.

- "A research-development partnership is a mutual learning process. We discovered the limits of the 'technology transfer' metaphor; instead of engaging in a discrete act of transfer, we engage in an ongoing cycle of problem-solution coevolution that involves research, rapid prototyping, user testing, deployment, and user feedback; issues such as access to expertise, knowledge of local culture and technology, credibility, and ownership become crucial."

  For us, the most dramatic illustration of this lesson was the reimplementation of the initial CLASSIC version of DA using DCS. It showed the importance of local ownership and the use of local technology. It illustrated that research and development members of the design team have different knowledge that is difficult to articulate but that is deployed effectively on actual problems encountered during design and use. We learned that researchers can achieve a valuable result–a "running specification" for a new system–without any of their specific technology going into general use. Finally, we learned that many new, interesting research problems arise only after a new technology is deployed. By participating in the development of the new technology, we (Terveen & Selfridge) as researchers gained credibility with ISCBU. As a result, we now have access to the new research problems and guaranteed interest for new technology that we develop–we have an ongoing relationship that we call a living research project.

6.2. DA Status and Evaluation

DA has been part of the software design process at ISCBU since October 1992. It has been used hundreds of times by dozens of developers at the average rate of about 20 times a week. Figure 9 illustrates usage from the week of October 1, 1992, to June 27, 1994 (except for Week 24, for which usage statistics are not available).

DA collects general feedback at the end of each session and gives users the opportunity to express detailed comments. DA has grown significantly during its lifetime as more information is added to it. Figure 10 shows a table of DA versions and some of the statistics collected.

After initial deployment and evaluation with primarily the asserts knowledge, the amount of information in the tool was expanded dramatically in Version 3.1. From then on, the amount of knowledge has continued to grow steadily. The degree of satisfaction of the users, at least as

Figure 9. **Weekly use of DA.**

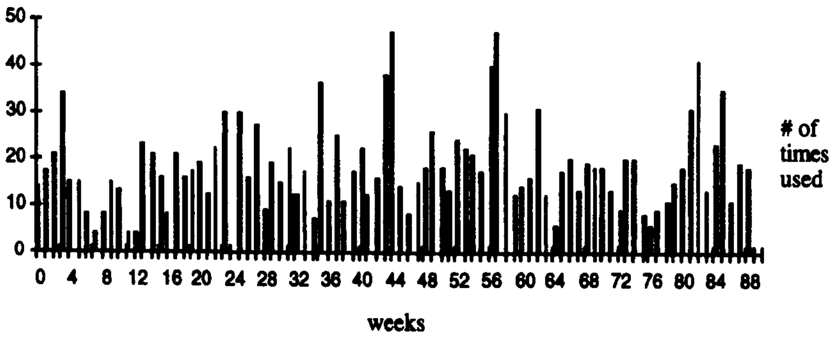


Figure 10. **DA growth and user evaluation statistics.**

Metric	System Version					
	2.4	3.0	3.1	4.0	4.1	4.2
Number of advice items	7	10	214	244	246	259
Number of times used	85	95	169	522	224	329+
Percentage of users judging the interaction useful	38	42	61	61	69	60
Percentage of users judging the level of detail *about right*	41	46	65	66	69	72

measured by the gross statistics just mentioned, has remained in the 60% range for the past four releases. The more detailed feedback has been used to generate a set of requirements for the next version of the tool.

ISCBU determined the cost savings of this tool over the first year and for each year thereafter. Because the value of the tool will be reflected in better software designs, the benefits of which will be evident only over time, the cost savings are only estimates. However, they are on the order of $1 million a year. These savings are derived from hard data on the number of faults found in the field and on the amount of money spent correcting them and from soft data on the degree to which DA will reduce these faults. (Because many process improvements are going on simultaneously within the organization, it is very difficult to assign credit to one particular activity.) These cost savings, of course, must be balanced against the cost of developing and maintaining the tool–in particular, against the cost of funding the knowledge maintenance activity. This activity officially is estimated at 30% of one technical head count; however, this figure might be misleading because much of the process-improving activity would be taking place anyway.

Perhaps even more important is the perception of ISCBU developers and management that this knowledge delivery tool works. It has become one of the primary mechanisms for fault reduction and process improvement. As shown in Figure 5, the tool now holds a significant amount of knowledge of several different kinds, and requests for additional knowledge come in regularly. Although issues of scale might arise as the amount of knowledge increases, DA is currently perceived in a very positive manner.

6.3. DA Limitations and Future Work

In this section, we review some of the problems with the current DA implementation revealed by feedback from users and from the knowledge base maintainer. We then describe a new graphical interface that we developed to alleviate some of these problems. Here we list some of the main problem areas of the current tool:

- *Different types of knowledge.* We have begun work with other AT&T organizations to apply DA to their processes. They have expressed the need to represent design knowledge expressed as checklists and tables, to filter knowledge based on the design level specified by the user, and to obtain a list of unresolved design issues.
- *Different means to access the knowledge.* Many users complained about having to step down the question hierarchies in a fixed order. Others commented that the structure of the dialogue was not clear. One request was for some sort of search or browsing functionality. Some users essentially said, "I simply want to find out what information there is about subject X. Why do I have to answer a bunch of questions?" Another request was that the tool should be able to generate a list of all the questions it would ask. The users who made this request said that they would like to use this as a checklist as they worked on their designs, answering questions as they became able to do so.
- *Better "Help."* Users sometimes commented that they could not tell what a question really meant until they answered it and considered subsequent questions and advice. Certain terms were ambiguous or unclear. Users wanted to be able to clarify the meaning of questions and specific phrases within questions and advice.
- *Better support for group use.* Users who were part of a group working on a large design commented that they were unable to answer all the questions asked by DA. They suggested that it would be effective if the notion of a DA session were extended to take on an incremental and persistent character. The goal would be to allow each member of the group to answer questions as he or she was able, with the system maintaining a unified history of the questions and advice.
- *System responses should be more than just textual advice.* Some users pointed out that, rather than telling them to contact a certain person or read a certain document or run a certain tool, the system should

be able to initiate action on their behalf. For example, DA might be able to compose an e-mail message, give the user a chance to edit the message, and then send the message to the right person with a single keystroke. Or, DA might be able to begin execution of a particular tool, initializing it with data based on user responses to the system's questions.

We developed a new graphical interface to DA that responds to many of these issues. It displays dialogue as a tree of typed nodes, reflecting the structure of the underlying representation (see Figure 3). As the user navigates the tree, questions are presented, answers collected, and advice delivered in a separate window. The tree is highlighted as interaction proceeds so that it is always evident where the user is in the dialogue. The interface also provides different types of user-directed search and browsing techniques, a notion of a "session" that the user can store and return to at a later date or share with other collaborators, a better and more accurate feedback mechanism, and the ability to generate nontextual advice and take actions on behalf of the user.

The new interface also addresses some problems of knowledge maintenance. Because the dialogue structure is explicitly represented as a tree and supports string search, it is easy to find and make small changes in the underlying text. More important, the tool provides a graphical editing ability to create new dialogue domains, "splice" them into the complete dialogue, and make changes in the existing structure (e.g., by collapsing a subtree, merging it with another part of the tree, or creating a subtree shared among different domains). We also have improved the ability to save interaction traces, making it easy to determine what parts of the dialogue are executed most often and what parts are relatively unvisited. This capability, coupled with an improved mechanism for getting user feedback, will greatly improve our ability to understand and modify the overall dialogue structure.

7. CONCLUSIONS

To summarize, we carried out a research and development partnership project that has resulted in a deployed design memory system. The system is in daily use, enjoys significant organizational support, and continues to grow–it is a living design memory. The primary lesson is the importance of achieving synergy between technology and organizational processes, but we have also learned important lessons about effective knowledge delivery, focusing on benefits to individuals, mediating collaboration, and carrying out collaborative design projects. But the story does not end there. We are still in active collaboration with the development organization and have designed a new interface that we hope will enhance the usability and maintainability of DA, generate a next set of research challenges, and further our understanding of providing effective and living organizational memory.

NOTES

Acknowledgments. We thank all the people at AT&T who helped make this work possible, especially Mary Zajac, Ron Brachman, Mike Wish, and Eric Sumner.

Authors' Present Addresses. Loren G. Terveen, AT&T Bell Laboratories, 2C-401, 600 Mountain Avenue, Murray Hill, NJ 07974. E-mail: `terveen@research.att.com`; Peter G. Selfridge, AT&T Bell Laboratories, 2B-425, 600 Mountain Avenue, Murray Hill, NJ 07974. E-mail: `pgs@research.att.com`; M. David Long, Cadre Technologies, Inc., Suite 800, 425 North Martingale Road, Schaumburg, IL 60173. E-mail: `dlong@cadre.com`.

HCI Editorial Record. First manuscript received September 2, 1993. Revision received March 23, 1994. Accepted by Ruven Brooks. Final manuscript received July 12, 1994. – *Editor*

REFERENCES

Ackerman, M. S. (1994). *Answer Garden: A tool for growing organizational memory.* Unpublished doctoral dissertation, Massachusetts Institute of Technology, Sloan School of Management, Cambridge.

Ackerman, M. S., & Malone, T. W. (1990). Answer Garden: A tool for growing organizational memory. *Proceedings of the Conference on Office Information Systems,* 31–39. New York: ACM.

Bailin, S. C., Moore, J. M., Bentz, R., & Bewtra, M. (1990). KAPTUR: Knowledge acquisition for preservation of tradeoffs and underlying rationale. *Proceedings of the 5th annual Knowledge-Based Software Assistant Conference,* 95–104. Liverpool, NY: Rome Air Development Center.

Barstow, D. R. (1979). An experiment in knowledge-based automatic programming. *Artificial Intelligence, 12,* 73–119.

Berlin, L. M., Jeffries, R., O'Day, V. L., Paepcke, A., & Wharton, C. (1993). Where did you put it? Issues in the design and use of a group memory. *Proceedings of the INTERCHI '93 Conference on Human Factors in Computer Systems,* 23–30. New York: ACM.

Borgida, A., Brachman, R. J., McGuinness, D. L, & Resnick, L. A. (1989). CLASSIC: A structural data model for objects. *Proceedings of the SIGMOD International Conference on Management of Data,* 59–67. New York: ACM.

Brachman, R. J., McGuinness, D. L., Patel-Schneider, P. F., Resnick, L. A., & Borgida, A. (1990). Living with CLASSIC: When and how to use a KL-ONE-like language. In J. Sowa (Ed.), *Formal aspects of semantic networks* (pp. 401–456). Los Altos, CA: Morgan Kaufman.

Clancey, W. (1991). The frame of reference problem in the design of intelligent machines. In K. VanLehn (Ed.), *The twenty-second Carnegie Symposium on Cognition: Architectures for intelligence* (pp. 357–424). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Colson, J. S., & Prell, E. M. (1992). Total quality management for a large software project. *AT&T Technical Journal, 71*(3), 48–56.

Conklin, E. J., & Burgess Yakemovic, S. C. (1991). A process-oriented approach to design rationale. *Human–Computer Interaction, 6,* 357–391.

Curtis, B., Kellner, M. I., & Over, J. (1992). Process modeling. *Communications of the ACM, 35*(9), 75–90.

Curtis, B., Krasner, H., & Iscoe, N. (1988). A field study of the software design process for large systems. *Communications of the ACM, 31,* 1268–1287.

Ehn, P., & Kyng, M. (1991). Cardboard computers: Mocking-it-up or hands-on the future. In J. Greenbaum & M. Kyng (Eds.), *Design at work: Cooperative design of computer systems* (pp. 169–195). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Fischer, G. (1990). Communication requirements for cooperative problem solving systems. *International Journal of Information Systems, 15*(1), 21–36.

Fischer, G., Grudin, J., Lemke, A. C., McCall, R., Ostwald, J., & Shipman, F. (1992). Supporting indirect, collaborative design with integrated knowledge-based design environments. *Human–Computer Interaction, 7,* 281–314.

Fischer, G., Lemke, A. C., Mastaglio, T., & Morch, A. I. (1991). The role of critiquing in cooperative problem solving. *Transactions on Information Systems, 9,* 123–151.

Fischer, G., Lemke, A. C., McCall, R., & Morch, A. I. (1991). Making argumentation serve design. *Human–Computer Interaction, 6,* 393–419.

Fischer, G., McCall, R., Ostwald, J., Reeves, B., & Shipman, F. M. (1994). Seeding, evolutionary growth and reseeding: Supporting the incremental development of design environments. *Proceedings of the CHI '94 Conference on Human Factors in Computer Systems,* 292–298. New York: ACM.

Franke, D. W. (1991). Deriving and using descriptions of purpose. *IEEE Expert, 6*(2), 41–47.

Gaines, B. (1989). Social and cognitive processes in knowledge acquisition. *Knowledge Acquisition, 1*(1), 39–58.

Greenbaum, J., & Kyng, M. (1991). Introduction: Situated design. In J. Greenbaum & M. Kyng (Eds.), *Design at work: Cooperative design of computer systems* (pp. 1–24). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Grudin, J. (1988). Why CSCW applications fail: Problems in the design and evaluation of organizational interfaces. *Proceedings of the CSCW '88 Conference on Computer-Supported Cooperative Work,* 85–93. New York: ACM.

Johnson, W. L., Feather, M. S., & Harris, D. H. (1991). The KBSA requirements/specification facet: ARIES. *Proceedings of the 6th Knowledge-Based Software Engineering Conference,* 48–56. Los Alamitos, CA: IEEE.

Kant, E., Daube, F., MacGregor, W., & Wald, J. (1991). Scientific programming by automated synthesis. In M. Lowry & R. McCartney (Eds.), *Automating software design* (pp. 169–205). Menlo Park, CA: AAAI Press/MIT Press.

Krasner, H., Terrel, J., Linehan, A., Arnold, P., & Ett, W. H. (1992). Lessons learned from a software process modeling system. *Communications of the ACM, 35*(9), 91–100.

Kunz, W., & Rittel, H. (1970). *Issues as elements of information systems* (Working Paper

131). Berkeley: University of California, Center for Planning and Development Research.

Lave, J. (1988). *Cognition in practice.* Cambridge, England: Cambridge University Press.

Mark, W., Tyler, S., McGuire, J., & Schlossberg, J. (1992). Commitment-based software development. *IEEE Transactions on Software Engineering, 18,* 870–885.

McDermott, J. (1982). R1: A rule-based configurer of computer systems. *Artificial Intelligence, 19,* 39–88.

Ramesh, B., & Dhar, V. (1991). Representation and maintenance of process knowledge for large scale systems development. *Proceedings of the 6th Knowledge-Based Software Engineering Conference,* 223–231. Los Alamitos, CA: IEEE.

Rich, C. H., & Waters, R. C. (1990). *The programmer's apprentice.* Reading, MA: Addison-Wesley.

Selfridge, P. G., Terveen, L. G., & Long, M. D. (1992). Managing design knowledge to provide assistance to large-scale software development. *Proceedings of the 7th Knowledge-Based Software Engineering Conference,* 163–170. Los Alamitos, CA: IEEE.

Shipman, F. M. (1993). *Supporting knowledge-base evolution with incremental formalization.* Unpublished doctoral dissertation, University of Colorado, Department of Computer Science, Boulder.

Shipman, F. M., & McCall, R. (1994). Supporting knowledge-base evolution with incremental formalization. *Proceedings of the CHI '94 Conference on Human Factors in Computer Systems,* 285–291. New York: ACM.

Shortliffe, E. H. (1976). *Computer-based medical consultation: MYCIN.* New York: Elsevier.

Silverman, B. G. (1992). Human–computer collaboration. *Human–Computer Interaction, 7,* 165–196.

Smith, D. R. (1991). KIDS: A knowledge-based software development system. In M. Lowry & R. McCartney (Eds.), *Automating software design* (pp. 483–514). Menlo Park, CA: AAAI Press/MIT Press.

Suchman, L. A. (1983). Office procedures as practical action: Models of work and system design. *ACM Transactions on Office Information Systems, 1,* 320–328.

Suchman, L. A. (1987). *Plans and situated action.* Cambridge, England: Cambridge University Press.

Terveen, L. G. (1993). Intelligent systems as cooperative systems. *International Journal of Intelligent Systems, 3,* 217–249.

Terveen, L. G. (in press). An overview of human–computer collaboration. *Knowledge-Based Systems.*

Terveen, L. G., & Selfridge, P. G. (1994). Intelligent assistance for software construction: A case study. *Proceedings of the 9th Knowledge-Based Software Engineering Conference,* 14–21. Los Alamitos, CA: IEEE.

Terveen, L. G., & Wroblewski, D. A. (1991). A tool for achieving consensus in knowledge representation. *Proceedings of the 9th National Conference on Artificial Intelligence, AAAI-91,* 74–79. Menlo Park, CA: AAAI Press/MIT Press.

Winograd, T., & Flores, F. (1986). *Understanding computers and cognition.* Norwood, NJ: Ablex.