From "Folklore" To "Living Design Memory"

Loren G. Terveen
600 Mountain Avenue
AT&T Bell Laboratories
Murray Hill, NJ 07974
(908) 582-2608
terveen@research.att.com

Peter G. Selfridge
600 Mountain Avenue
AT&T Bell Laboratories
Murray Hill, NJ 07974
(908) 582-6801
pgs@research.att.com

M. David Long
2000 N. Naperville Road
AT&T Bell Laboratories
Naperville, IL. 60566
(708) 979-5648
mdlong@ihlpb.att.com

ABSTRACT
We identify an important type of software design knowledge that we call *community specific folklore* and show problems with current approaches to managing it. We built a tool that serves as a *living design memory* for a large software development organization. The tool delivers knowledge to developers effectively and is embedded in organizational practice to ensure that the knowledge it contains evolves as necessary. This work illustrates important lessons in building knowledge management systems, integrating novel technology into organizational practice, and managing research-development partnerships.

KEYWORDS: organizational interfaces, organizational design, knowledge representation, software productivity

INTRODUCTION

Developing and maintaining large software systems is notoriously difficult and expensive. Several factors contribute to this situation. Software development is a new discipline; this leads to rapid change in languages, tools, and methodologies. Relatively simple 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 raises the possibility of change, and market pressures ensure the necessity of change.

A crucial implication of this picture is that 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 application domain, the existing software base, and local programming conventions. This knowledge typically has the status of *folklore*, in that it is informally maintained and disseminated by experienced developers. This process is (1) ineffective – not everyone gets the knowledge they need, (2) inefficient – communication of knowledge, whether in formal meetings or informal consulting, comes to take up more and more time, and (3) fragile – loss of key personnel can mean loss of critical knowledge.

We addressed the problem of managing design knowledge in a software development organization in AT&T. 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. We describe this type of system as a *living design memory*. The term "living" emphasizes that the system must be embedded in the organization's normal design process, in particular, that it must evolve in response to problems detected with it or changes in the knowledge situation of the organization. We constructed a system according to these guidelines and deployed the tool in the organization.

We have learned important lessons from this work, starting with technical issues involved in developing a system for managing design knowledge (see [22] for details). Here we emphasize more general lessons that concern embedding any technical solution in organizational practice and carrying out a successful research-development partnership. The two most important lessons are:

- knowledge of *facts* is not enough; it also is necessary to know *how the knowledge is be used* – where in the process it fits, how to access only relevant knowledge, and how to allow for change;
- the members of the community in which a system is to be deployed must *own* the system – they must be able to use it effectively and modify it when necessary; this leads us to reject the metaphor of "technology transfer" in favor of "knowledge communication" as an appropriate paradigm for a research-development partnership.

The focus of this paper is to elaborate the context in which these lessons were learned. We first explore the knowledge management problem in more detail and discuss challenges to acquiring, maintaining, and disseminating design knowledge. We then describe a framework for integrating a design memory tool into an existing software development process, making it living. We next present the implemented tool that instantiates the framework. We emphasize how the tool evolved in response to the expectations of the user community and the shifting of "ownership" from the research members of the team to the developers. We

conclude by comparing our work to other approaches, bringing out unique characteristics of our work through the comparison, and describing areas for future work.

THE KNOWLEDGE MANAGEMENT PROBLEM IN LARGE SCALE SOFTWARE DEVELOPMENT

The work described here was a collaboration between AT&T research (represented by the first two authors, LGT and PGS) and a large AT&T software development organization (represented by the third author, MDL). The collaboration was initiated to address problems in managing design knowledge in the development organization, which consists of several thousand people who maintain and enhance a large telecommunications software system. It is important to realize that software development in this organization never begins "from scratch"; it always involves enhancing or repairing an existing, deployed software system.

Our work has focused on the design process. This process starts with a specification document, originating from either a customer request or an internal source. The specification document describes a new feature of the software in customer (behavioral) terms. This document 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 necessary, feedback is incorporated into the design and the process iterates. Once the design document is complete and approved, it is passed to a coding phase. This process is shown in figure 1.

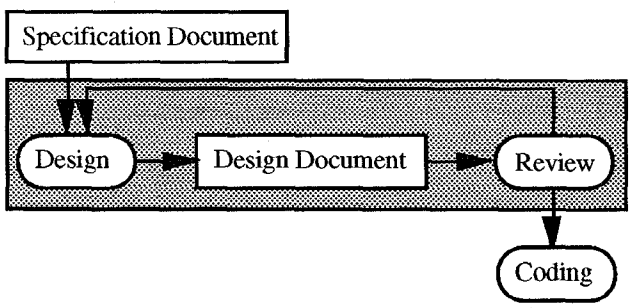

Figure 1: The Design Process

One major problem in the design process is the lack of accessible general design knowledge. This knowledge involves such things as *real-time and performance constraints* ("one real-time segment shouldn't take more than 200 milliseconds 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 get a bad message"), and *personnel and organization* ("ask Nancy about that; she knows about local stack space"). This kind of knowledge usually is not written down, rather, it is part of the community specific folklore that is maintained and disseminated by experienced individuals in the organization. This form of knowledge maintenance and dissemination is unsatisfactory. First, not only are experts difficult to locate when needed, but individuals must know who the expert is 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. Second, experts can spend more time disseminating knowledge than solving problems relevant to their jobs. Third, knowledge often is generated (e.g., in design, review, testing, fault analysis) only to be lost, thus depriving the organization of a valuable resource and leading to potential duplication of effort in the future. Finally, since key knowledge often is only known to a few individuals, loss of personnel can mean loss of knowledge. Failure to manage design knowledge effectively can result in sub-optimal designs, late and costly detection of errors, long delivery times, and personal frustration.

AT&T has been trying to improve its software development productivity for some time. A major emphasis has been to institute various quality initiatives aimed at formalizing the process to make it measurable, repeatable, and more easily managed [6]. As in the design process shown in figure 1, sequences of steps have been defined, suppliers and customers identified, and inputs and outputs specified. Any solution to the problem of managing design knowledge will be deployed in the context of the existing design process.

The organization tried to address the problem of managing design knowledge by documenting as much knowledge as possible 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 be incomplete and incorrect, and the programming constructs, requirements, constraints, and methodologies they describe all will change over time.

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. For a design memory tool to be adequate, developers must use the tool consistently and at appropriate points in the development process. Then they have to incorporate knowledge from the tool into their designs (the fundamental purpose of the tool is to produce *better* software designs).

In addition, there should be an organizational method for encouraging tool use and checking whether the tool was used and the advice followed. Finally, exceptions and modifications to the advice need to be captured, both for maintenance and to assure credibility with the developers. The maintenance issue is critical: improperly maintained knowledge will, rightfully, go the way of improperly maintained documentation.

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 above requirements. The framework is based on two components. First, a *design knowledge base* records relevant information. Second, a *Designer Assistant* program provides access to the knowledge base, following the general paradigm of interactive assistance for software development [21]. Our framework assumes the Designer Assistant augments the existing development process. This process uses *informal* design artifacts, i.e., text documents. Therefore, the Designer Assistant provides textual advice to developers, and it is their responsibility to incorporate the advice into their designs or explain why the advice does not apply to their designs. Attempts to formalize design artifacts through the use of knowledge-based tools [1, 16, 19, 20] are complementary to our approach.

However, this framework is flawed; it would be adequate only if all relevant design knowledge could be captured completely, once and for all. This clearly is an unlikely and unrealistic assumption. As Clancey states particularly well [5], a knowledge base is always subject to additional refinement and re-interpretation. More important, the world changes: the software base changes (indeed, this is the goal of the design activity), the hardware and software technology changes, protocols and conventions change, customer requirements change, faults are observed in the running software, and all the other assumptions and constraints are subject to continual, if slow, evolution. This places some additional requirements on our framework, in particular, that it support (1) the elaboration and evolution of design knowledge as the tool is used and evaluated; and (2) the addition of new knowledge generated during design activities.

To support the first requirement, we record a trace of user interactions with the Designer Assistant and annotate the Design Document with this trace. This allows those aspects of the design that were influenced by the advice to be traced during design review. In addition, we modify the review process slightly to make the *advice itself an object of review*. To support the second requirement, note that the problem is not to *produce* new knowledge, but rather to ensure the new knowledge already generated during normal development activities is *captured* in the knowledge base. We do this in three ways. First, the Designer Assistant elicits comments from developers concerning revisions or additions they think should be made to the knowledge base. Second, when a fault is observed in the running software system, and analysis shows that it was due to a design error, a report detailing the problem and its solution is generated to be encoded in the knowledge base. Finally, we add a knowledge base maintenance activity to the design process. This activity takes as input information to be added to the knowledge base, in particular, fault reports and the annotated design document and reviewer comments; it produces changes or updates to the design knowledge base as necessary. Figure 2 shows the complete framework.

To summarize the framework, the design knowledge base contains information relevant to design tasks in the application domain. As developers design, the Designer Assistant program helps them to access relevant knowledge. The result of the design process includes the design document, feedback from the developer about updates to the design knowledge base, and a trace of the interactions of the developer with the Designer Assistant. At the review, reviewers examine the design and identify issues, some of which result from the advice of the design assistant. Such issues lead to (proposals for) modifications of the design knowledge base. Other issues lead to (proposals for) addition of new knowledge to the knowledge base. All pro-

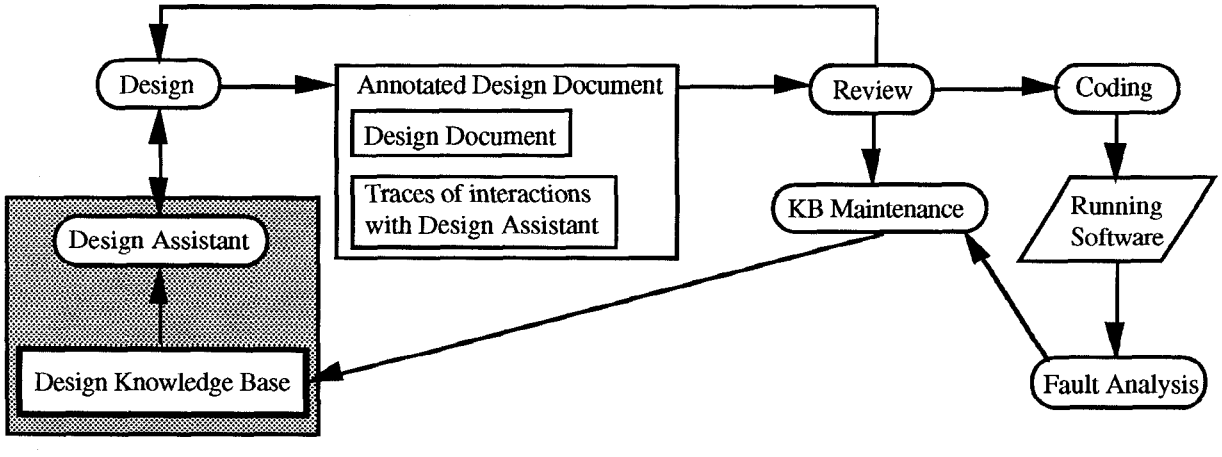

Figure 2: The Living Design Memory Framework

posals for updates to the knowledge base that are generated during design, review, or fault analysis are collected and sent to a knowledge base maintainer. The knowledge base maintainer then updates the knowledge base. The Designer Assistant is embedded in organizational practice so that it evolves in response both to shortcomings in its own knowledge and to changes in the external situation.

EVOLVING A DESIGN MEMORY TOOL

We developed a design memory tool within our framework that contains significant "folklore" knowledge in the telecommunications software domain. The knowledge has been acquired by engineering several domains and analyzing several fault reports. The tool works by leading developers through a dialogue concerning characteristics of their design, then providing advice based on the answers. The tool 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. We focus here on tracing the evolution of the tool, emphasizing that making the tool successful involved a process of *transfer of ownership* to the development organization.

Constructing a Prototype

We first identified a design knowledge sub-domain in which to construct a prototype. We selected an error handling mechanism that is critical to the system's fault-tolerance. This mechanism is used if the code being designed reaches an illegal state. The mechanism 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 that is 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, when developers decide to use this error mechanism, they have to make a series of decisions about exactly how to invoke it. Some of these decisions are quite complicated and interact in various ways.

This domain, while limited, 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 this mechanism. However, this knowledge is managed as folklore, as we described it earlier: experts disseminate this information in a frustrating and inefficient manner, i.e., one-to-one communication with individual developers. Finally, discussions with developers showed that this domain is typical in all of these respects, and that many other domains within the organization share these problems.

Once the domain was chosen we spent dozens of hours interviewing several domain experts about the knowledge needed to use this mechanism and studying the existing written documentation. We took a set of examples of the mechanism in the current code base and asked the experts to categorize the examples in terms of *design attributes*, the features of a design that all the uses of the mechanism in each category responded to. This was a very important abstraction step because it meant that the tool interaction could use terms that are familiar to developers, rather than refer to syntactic features of the construct. Presumably developers won't be familiar with the latter vocabulary, since it is precisely this they are getting help about.

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 could be 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 de-synchronized?". Then, for each design attribute, we elicited from the domain experts advice about how to use the error handling mechanism in this situation. Some advice might apply only to very specific design situations (the leaf nodes in the attribute hierarchy), while 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. (See [22] for details about the representation of design attributes and advice items.)

The next task was to construct a prototype design memory tool. To do so, we had to represent the information we had acquired and construct an interface to allow developers to access the information. We used the language CLASSIC [3,4] to represent the information. We then 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, independent of any specific window or platform features. The prototype simply used basic C input and output routines. The interaction basically consisted of the tool asking a developer yes/no questions to guide the developer down the 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 [22]). When the interaction was complete, the tool asked several evaluation questions – "Was this session useful?" and "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 about the interaction with the tool.

The output of the tool is a script of the interaction. The script is added to the formal design document for two reasons. First, the advice becomes part of the document and gets reviewed during the formal review process. The reviewers have a chance to verify that software developers did receive information relevant to their design situation and either followed the advice of the tool or determined that their situation was an exception to the situation anticipated by the tool – in such cases, the exception itself is worth noting, discussing and acquiring. Second, the advice itself can be reviewed, and changes and modifications can be directed to the maintenance process illustrated in figure 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 section of design involving the error handling mechanism. They used the tool to get advice on how to do so. Their reaction was highly favorable; in some cases, it was asserted that the 20 minutes spent using the tool saved from 4 to 8 hours of their time! The reason for this is that the only other way to find out 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 to be very positive, we deployed the Designer Assistant in the design process. That is, the formal process definition was modified to include the steps "run the Designer Assistant and append the resulting transcript to your Design Document." We sent email to all the developers in the organization announcing the availability of the tool. Within 15 days, more than 85 developers had used the DA. Their feedback was unexpectedly negative. Only 38% judged the interaction "useful" and 58% said the level of detail was "too little". These rating and detailed comments from the users indicated that the DA had two major problems. First, developers would not consider it useful until it had more knowledge – it was more trouble than it was worth to run a tool that contained information about just one small domain. Second, the interface did not match user expectations, e.g., it gave no guidance in entering legal answers, and the overall structure of the dialogue was unclear.

Thus far in the project, the research members of our team, LGT and PGS, had designed and implemented the prototype. However, responsibility for the tool now was shifting to MDL. He was faced with the two problems of improving the interface and adding more knowledge. To do so, he took a major step, re-implementing the Designer Assistant 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 provided much useful functionality, including input type checking, ability to revisit and re-answer any previous question, text and cursor-based menus, automatic recording of user responses, and output of a transcript in a convenient database format, so it solved many of the user complaints. The re-implementation illustrates several interesting points:

(1) MDL now "owned" the re-implemented Designer Assistant – it was a facile tool for him. This meant that he could modify or add knowledge easily.

(2) DCS was a known resource in the repertoire of the developers but not the researchers. Once the researchers moved out of their world of workstations and X Windows, their knowledge of system development environments was limited. 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* [14], and that the type of learning required may be impossible to anticipate.

(3) Changing the underlying technology from CLASSIC to DCS involved some tradeoffs. Gains in ease of interaction and ownership by MDL came at the expense of great loss of representational power. However, where the CLASSIC prototype focused on elegant representation and efficient computation of advice, experience showed that it was much 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 tradeoffs. Those members of the design team who actually implement the design have great influence in deciding how the tradeoffs are resolved [9]. Research and development members of a joint design team might well judge how to resolve particular tradeoffs differently. Thus, tradeoffs that have not been articulated and discussed may well become the occasion for re-design or re-implementation as ownership of a system shifts from researchers to developers.

(4) The re-implementation 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 new partnership undertaken by LGT and PGS, all parties have agreed up front that this is what the researchers will produce.

When the re-implemented Designer Assistant was deployed, complaints about the interface ended. However, user feedback was only marginally more positive. Of 67 users who provided feedback, 42% rated the interaction "useful", and 46% now said the level of detail was "about right". More knowledge still had to be added before a majority of developers would consider the tool useful.

Since then, much new knowledge has been added, including one large domain – impacts of design decisions on customer documentation – and several smaller domains. In addition, rules derived from analysis of several fault reports have been encoded. The knowledge can be divided into three categories, (1) *expert knowledge* – like the error handling mechanism, specialized areas of design knowledge that most designers are not familiar with, (2) *impact knowledge* – how characteristics of a design impact another area of the

software, and (3) *fault prevention knowledge* – how characteristics of a design could lead to a fault, and how the fault can be avoided. To facilitate adding knowledge, MDL designed a simple rule language tailored for representing hierarchies of design attributes and a compiler that produces DCS code from the rules. The Designer Assistant now contains more than 250 rules. The ongoing process of knowledge acquisition has shown that one of the most useful results of the project has been the principle of organizing knowledge in terms of questions that index from characteristics of a design situation (that are familiar to developers) to advice about a particular domain (e.g., use of the error handling construct, impact on customer documentation, necessary changes to the database). This heuristic is useful for acquiring domain knowledge from experts, organizing the knowledge, and providing efficient access to the knowledge.

User satisfaction with the current version of the Designer Assistant is significantly higher. Since it has been deployed, 354 developers have used the system and 256 have provided feedback. Of those who provided feedback, 61% judged the interaction "useful" and 63% said the level of detail was "about right". Table 1 summarizes user judgements of the three versions of the Designer Assistant that have been in use.

metric	Classic	DCS 1	DCS 2
number of users	≥ 85	95	354
users giving feedback	85	67	256
% judging "useful"	38	42	61
% judging "about right"	41	46	63

Table 1: Usage statistics for the Designer Assistant

User evaluation of the current tool is positive, but not highly so. However, the development organization considers the Designer Assistant to be very successful since it does record knowledge that previously was available only as informal folklore or in inefficient documents. Developers are using the system on a daily basis and are able to access information effectively. Further, it is clear that what is required to increase user satisfaction is still more knowledge, and progress on this front is very good. Several additional domains and fault analyses are ready to be encoded. The knowledge base should grow by 50% within the next few months. Finally, there is managerial commitment to the Designer Assistant approach to managing design knowledge and to providing the resources necessary to increase its effectiveness. We discuss additional ways of evaluating the success of the Designer Assistant in the next section.

To summarize, the research members of our team constructed a prototype Designer Assistant that served the role of a "running specification." Ownership of the prototype was transferred to the development organization, with successful transfer requiring a re-implementation. A knowledge acquisition and organization heuristic (indexing by design attribute) developed in building the prototype has proved extremely useful in ongoing knowledge engineering.

DISCUSSION

We next 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, e.g., relating patient symptoms to disease classifications [23] or customer computer orders to a configuration diagram [18]. Expert systems typically are intended to *automate* a task, i.e., to perform *like an expert* in their domain. Expert systems work to the extent that knowledge in a domain can be completely formalized; this requires both articulating the knowledge precisely and encoding it in a format that allows the system to compute the proper inferences. This type of approach applied to software design might involve deriving a program from some (formal or informal) specification of a problem to be solved.

However, we see limits on the applicability of the formalization that expert systems require. First, various researchers have advanced strong theoretical arguments that many domains cannot be completely formalized and that AI-style knowledge bases are inherently incomplete [5, 24, 26]. Second, attempts to construct automated software design systems [2] have proved unsuccessful, leading the field to shift its emphasis to assisting people in designing software (e.g., 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. Our experience has shown the effectiveness of giving software designers efficient access to relatively informal information, like "ask Nancy; 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 the Designer Assistant and guiding developers directly to the relevant expert when additional communication is required. We also have addressed not just acquiring factual knowledge, but also how the knowledge is to be used – for example, at what point in the process it should be consulted, how it should be indexed for efficient access, and when and how it should evolve. Thus, our focus is on helping a software development organization manage its knowledge effectively, rather than automating parts of the software development process.

Second, there has been previous work that has called for a design memory (10, 19) and even suggested that a design memory must evolve over time. However, this work has used formal representations of design artifacts and components, e.g., code modules. Much power is gained from this formal representation, e.g., a design assistant system can help in fitting new designs into a library of existing designs, acquiring and organizing the rationale for designs, and can use rules to evaluate designs, judge trade-offs, etc.

We view this work as complementary to ours (in fact, we have done similar work in the past [25]). However, there are several differences. First, in the development organization we have worked with, design artifacts are English documents; thus, an approach that required a formal representation of design artifacts was not possible immediately. However, we see this as a promising area of future work, and our success to date increases the likelihood that this more radical innovation will be considered. Next, the "folklore" knowledge we have captured is a much broader class of knowledge than these systems have handled, and we deliver it in different ways. Finally, 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 co-designed to ensure a *living* design memory. In particular, while 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 is it left up to individual designers. There are a number of reasons for this, chiefly that it is been our experience that systems must have "owners" (in our case, the knowledge maintenance group), since if no one is responsible for the integrity of a system, it decays rapidly over time.

Third, work on design rationale [1, 7, 11, 12, 17, 20] also is relevant. 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 [12, 20] to relatively informal text or hypertext representations [7]. Work reported by Conklin and Yakemovic [7] in applying the IBIS [17] methodology to a software development project is most relevant to the concerns of this paper. They developed a simple textual form of IBIS 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 consult information relevant to their evolving design. One important reason why the IBIS technology was effective was that it incrementally improved an existing task (that is, 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 we were concerned with 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 AT&T development organization. We focused not just on integrating the Designer Assistant program 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 Yakemovic's commitment to cost-benefit analysis [15] concerning the adoption of new technology. The Designer Assistant is quick and easy for developers to use – no training is required, and the average session lasts about 10 minutes – and it offers substantial benefits, e.g., they do not have to read long documents, waiting time is reduced, review meetings can be smaller, thus scheduled more quickly, and reviewers can be satisfied more easily. Domain experts have a larger burden, since they are asked to cooperate with the knowledge maintenance group to encode their domains. However, we have found that experts are eager to do so because it reduces the amount of time they have to spend on consulting and lets them accomplish other work, and it helps to ensure that design rules they think are important are communicated throughout the development community.

Finally, our framework responds to many of the points made by Curtis, Krasner, and Iscoe [8] in their discussion of large software development projects. They state that software development must be seen as a learning and communication process. They recommend that software development tools facilitate the enterprise wide sharing of knowledge, accommodate change as a normal and expected process, and serve as mediums 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.

We conclude by discussing areas for future work. First, we will continue to encode new knowledge in the Designer Assistant. Thus far, indexing by design attributes has proved effective; however, we are looking for types of knowledge that cannot be coded adequately in this way. Second, we will do more evaluation of the system. We will track the metric *number of faults per line of source code due to the design process*. As the Designer Assistant contains more and more knowledge, including knowledge of past design faults, we expect this number to decrease by at least 25%. We will examine feedback from developers who have used the system and interview individual developers to identify "success stories" of the form "I would have made the following design error if not for advice from the Designer Assistant..." Third, we will improve the knowledge maintenance process by defining the process more precisely and by developing a tool to assist knowledge maintainers in updating the design knowledge base. We postponed work on such a tool until we gained experience with how the process of updating the knowledge base was working in practice. We now can begin to state requirements for this tool. Finally, several other organizations within AT&T are very interested in using the Designer Assistant tool and methodology to manage their knowledge, and we have begun working with them to enable them to do so.

REFERENCES

1. Bailin, S.C., Moore, J.M., Bentz, R., & Bewtra, M. 1990. KAPTUR: Knowledge Acquisition for Preservation of Tradeoffs and Underlying Rationale. *Proc. 5th Annual Knowledge-Based Software Assistant Conference.* (Syracuse, NY, Sept. 1990), pp. 95-104.

2. Barstow, D.R. 1979. An Experiment in Knowledge-Based Automatic Programming. *Artificial Intelligence.* 12(2): 73-119.

3. Borgida, A., Brachman, R.J., McGuinness, D.L, & Resnick, L.A. 1989. CLASSIC: A Structural Data Model for Objects. *Proc. ACM SIGMOD Int'l. Conf. on Management of Data.*

4. 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 Sowa, J., Ed. *Formal Aspects of Semantic Networks.* Morgan Kauffman.

5. Clancey, W. 1991. The Frame of Reference Problem in the Design of Intelligent Machines. In vanLehn, K., Ed. *Architectures for Intelligence: The Twenty-Second Carnegie Symposium on Cognition.* Hillsdale, NJ: Lawrence Erlbaum Associates.

6. Colson, J.S. & Prell, E.M. 1992. Total Quality Management for a Large Software Project. *AT&T Technical Journal.* 71(3): 48-56.

7. Conklin, E.J. & Burgess Yakemovic, KC. 1991. A Process-Oriented Approach to Design Rationale. *Human-Computer Interaction.* 6 (3-4): 357-391.

8. Curtis, B., Krasner, H., & Iscoe, N. 1988. A Field Study of the Software Design Process for Large Systems. *CACM.* 31(11): 1268-1287.

9. Ehn, P. & Kyng, M. 1991. Cardboard Computers: Mocking-it-up or Hands-on the Future. In [13].

10. Fischer, G., Grudin, J., Lemke, A.C., McCall, R., Ostwald, J., & Shipman, F. 1992. Supporting Indirect, Collaborative Design with Integrated Knowledge-Based Design Environments. To appear in *Human-Computer Interaction.* 7(3).

11. Fischer, G., Lemke, A.C., McCall, R., & Morch, A.I. 1991. Making Argumentation Serve Design. *Human-Computer Interaction.* 6 (3-4): 393-419.

12. Franke, D.W. 1991. Deriving and Using Descriptions of Purpose. *IEEE Expert.* April 1991.

13. Greenbaum, J. & Kyng, M. 1991. *Design at Work: Cooperative Design of Computer Systems.* Hillsdale, NJ: Lawrence Erlbaum.

14. Greenbaum, J. & Kyng, M. 1991. Introduction: Situated Design. In [13].

15. Grudin, J. 1988. Why CSCW Applications Fail: Problems in the Design and Evaluation of Organizational Interfaces. *CSCW-88.* 85-93.

16. Johnson, W.L., Feather, M.S., & Harris, D.H. 1991. The KBSA Requirements/Specification Facet: ARIES. *Proc.s 6th Knowledge-Based Software Engineering Conference* (Syracuse, NY. Sept. 1991), pp. 57-66.

17. Kunz, W., & Rittel, H. 1970. Issues as Elements of Information Systems. Working Paper 131. Center for Planning and Development Research. The University of California at Berkeley.

18. McDermott, J. 1982. R1: A Rule-Based Configurer of Computer Systems. *Artificial Intelligence.* 19: 39-88.

19. Mark, W., et al. 1992. Commitment-Based Software Development. *IEEE Transactions on Software Engineering.* October 1992.

20. Ramesh, B. & Dhar, V. 1991. Representation and Maintenance of Process Knowledge for Large Scale Systems Development. *Proc. 6th Annual Knowledge-Based Software Engineering Conference.* Syracuse, NY. Sept. 1991), pp. 223-231.

21. Rich, C.H., & Waters, R.C. 1990. *The Programmer's Apprentice.* Reading, MA: Addison-Wesley.

22. Selfridge, P.G., Terveen, L.G., & Long, M.D. 1992. Managing Design Knowledge to Provide Assistance to Large-Scale Software Development. *Proc. 7th Knowledge-Based Software Engineering Conference,* (McLean, VA, Sept 1992).

23. Shortliffe, E.H. 1976. *Computer-Based Medical Consultation: MYCIN.* New York: American Elsevier.

24. Suchman, L.A. 1987. *Plans and Situated Action.* Cambridge: Cambridge University Press.

25. Terveen, L.G. & Wroblewski, D.A. 1991. A Tool for Achieving Consensus in Knowledge Representation. *AAAI-91.*

26. Winograd, T. & Flores, F. 1986. *Understanding Computers and Cognition.* Norwood, NJ: Ablex.