Managing Design Knowledge to Provide Assistance to Large-Scale Software Development

Peter G. Selfridge
AT&T Bell Laboratories
Murray Hill, NJ 07974
pgs@research.att.com

Loren G. Terveen
AT&T Bell Laboratories
Murray Hill, NJ 07974
terveen@research.att.com

M. David Long
AT&T Bell Laboratories
Naperville, Il. 60566
mdlong@ihlpb.att.com

Abstract

Developing and maintaining large software systems is notoriously difficult and expensive. One source of difficulty is that such projects involve large amounts of disparate knowledge about the domain, the hardware platform, the existing software architecture, the technical personnel and resources, etc. A critical problem is that a great deal of relevant knowledge is "folklore": it is not documented and remains accessible only through human experts. We propose to use knowledge-based technology to manage this kind of knowledge to increase productivity and product quality. To do so, we address three central issues. First, knowledge must be acquired from human experts. Second, the knowledge must be adequately represented and made accessible to users. Third, and most important, the knowledge must be maintained: just as code evolves, so will this knowledge. This paper addresses these issues in the context of providing relevant advice to developers during software design. It then describes an implemented design knowledge tool, augmenting an existing organizational design process, that provides such advice about a limited domain for a large-scale software development project.

1. Introduction

Large-scale software development is a complex engineering activity with some unique attributes. First of all, the nature of the final product is fundamentally different from physical design artifacts [4]. Software is more complex at a component level, more difficult to visualize, and is significantly more malleable, both during initial development and subsequent maintenance. Second, this capability for change leads to demands for new functionality, which in turn leads to the need for continual maintenance of the product; this maintenance looks much more like on-going development than maintenance of, say, a bridge. Third, it is a newer and less mature discipline than others, and thus its notations, conventions, and common practice for software development are less developed. This also applies to the practice of managing large software projects.

These factors contribute to making the process of software design (in this paper, we are concerned with *large-scale* software design) significantly more complex and difficult than other engineering design tasks. For example, consider a large body of software that controls a telecommunications switch. Such software has a multitude of interdependent functions: it allows people to make normal phone calls, provides dozens or hundreds of special telecommunications features, supports billing, and includes numerous components that make the system more fault-tolerant. Now consider the process of adding new functionality to the existing software, or designing changes to fix a bug or restructure the existing code. *The success of this process will depend on taking into account numerous general design constraints,* only partly reflected by the existing code. These general constraints may reflect real-time considerations, internal resource limitations, and personnel and logistics knowledge. Rarely are such constraints written down, partly because they are hard to capture and also because they are very likely to evolve as the system changes.

This paper identifies the problem of managing design knowledge as a crucial component in a large-scale software development project. We explore this design knowledge problem in more detail, describe both technical and non-technical challenges, discuss the maintenance of such knowledge, and briefly explore the issue of acquisition. We then describe a framework for providing knowledge-based assistance to software developers. This framework is integrated with and extends an existing design process and exploits that process to address the problem of knowledge maintenance. Then, we present an implemented design knowledge tool instantiating our framework that gives software developers access to knowledge about a particular error handling mechanism. We describe the organization of the knowledge and the design of the interface. Finally, we discuss the status of the implementation, areas for future work, and conclude.

2. General Design Knowledge for Software Development

We recently began a collaboration with a large software development organization in AT&T. This organization

0-8186-2880-4/92 $03.00 © 1992 IEEE

consists of several thousand people whose job it is to maintain and enhance a large telecommunications software system. An important part of the software development process is the design process. This process starts with a specification document, originating from either a customer request or an internal source. This 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 is then 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:

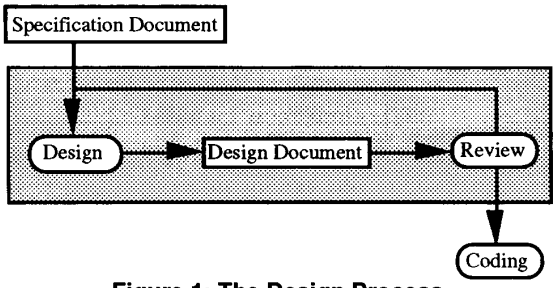

Figure 1: The Design Process

One major problem in the design process is the lack of accessible general design knowledge. This knowledge involves such things as real-time constraints ("one real-time segment shouldn't take more than 200 milliseconds or overall performance will suffer"), knowledge of the current implementation ("the terminating Terminal process is already close to its memory limitation, so you can't add much to it"), knowledge of local programming conventions ("call the central error reporting mechanism if you get a bad message"), and personnel and organization ("ask Nancy about that; she knows about local stack space"). (These examples of general design knowledge are diverse and informal, and this has some implications for our approach. In particular, these examples do not easily lend themselves to formal representation. They can, however, be disseminated by a computer-based tool if acquisition and maintenance issues can be addressed.) This kind of knowledge usually is not written down, rather, it is part of the organizational "folklore" that is maintained and disseminated by experienced individuals in the organization. This form of knowledge maintenance and dissemination is unsatisfactory: not only are experts difficult to locate when needed, but an individual must know who the expert is for their particular problem. In addition, expertise shifts among individuals over time; and experts can spend more time disseminating knowledge that solving problems relevant to their jobs. The failure of individuals to get access to appropriate design knowledge can result in incomplete designs, long delivery times, personal frustration, and a final product which is sub-optimal. We refer to this as the problem of *managing design knowledge*, and it is this problem we are trying to address.

AT&T has been aware of these problems for some time and has taken several steps to alleviate them. The first was to institute a number of quality initiatives to improve the overall software development process. The process was streamlined and a series of guidelines developed to specify the steps and milestones in the process. While substantially improving the process, these changes do not address the problem of managing design knowledge. (However, this did signal a willingness on the part of the organization to change its way of doing work and alerted us to the opportunities this project addresses.)

The second step taken by the organization was to try to document as much design knowledge as possible in structured text files. This approach is inadequate for three reasons. The first is the *acquisition and representation problem*: the amount of such knowledge is large, so capturing it is tedious and time-consuming. It also is unclear how to organize or index this knowledge. The second is the *access problem*: 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). Finally, there is the *maintenance problem*: the knowledge will change over time, just like the code of the system, and must be maintained if it is to remain useful.

These three problems represent the technical challenges to building a design knowledge tool. The other primary challenge is *organizational*: designing such a tool so that it can be integrated successfully with the current organizational processes. Such integration is surprisingly complicated; often, non-technical considerations are more difficult to overcome [10]. Individuals must use the tool consistently and successfully to gain the embedded knowledge. Then they have to incorporate this knowledge into their designs (the fundamental purpose of the tool is to produce *better* software designs); optimally, there should be an organizational method for encouraging tool use and checking whether the tool was used and the advice followed. In addition, exceptions and modifications to the advice need to be captured, both for maintenance and to assure credibility with the developers. Finally, the maintenance issue is a critical one: improperly maintained knowledge will, rightfully, go the way of improperly maintained documentation.

3. A Framework for Providing Knowledge-Based Assistance

Our first approximation of a framework for providing knowledge-based assistance supposes the existence of a *design knowledge base*, somehow acquired and adequately

represented and indexed. Then, we suppose a *design assistant* program that provides access to the knowledge base, following the general paradigm of interactive software assistance for different phases of software development [16]. Our framework assumes the design assistant augments an existing, informal process, and provides informal advice for the user. It is the user's responsibility to follow the advice or explain why the advice does not apply to his or her design. This contrasts with a design assistant that plays a more central role and provides advice about specific designs [20]. Similarly, attempts to formalize design artifacts through the use of knowledge-based tools [1, 11, 12, 14] are complementary to our approach. The augmented software design process is shown in figure 2:

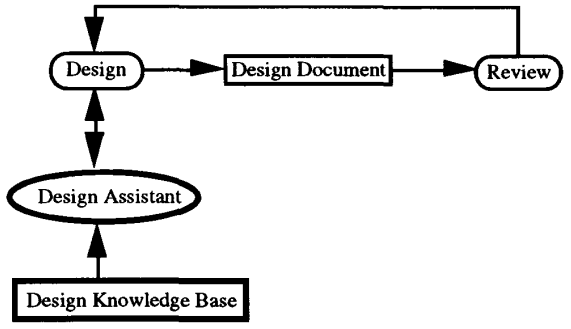

Figure 2: Design process with Knowledge-based assistance added

However, this framework has a fundamental flaw: it would be adequate only if all relevant design knowledge could be captured completely, once and for all. This is clearly an unlikely and unrealistic assumption. As Clancey states particularly well [6], 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, 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 Design 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 design and review activities is *captured* in the knowledge base. We do this in two ways. First, we modify design documents further by allowing designers to note new knowledge they would like to see added to the knowledge base. Second, we add a knowledge base maintenance activity to the design process. This activity takes as input the annotated design document and reviewer comments and generates changes or updates to the design knowledge base. The knowledge base maintenance process is responsible for integrating into the knowledge base the new knowledge produced during design, including designer/reviewer disagreements with or modifications of information already in the knowledge base. Figure 3 shows the complete framework.

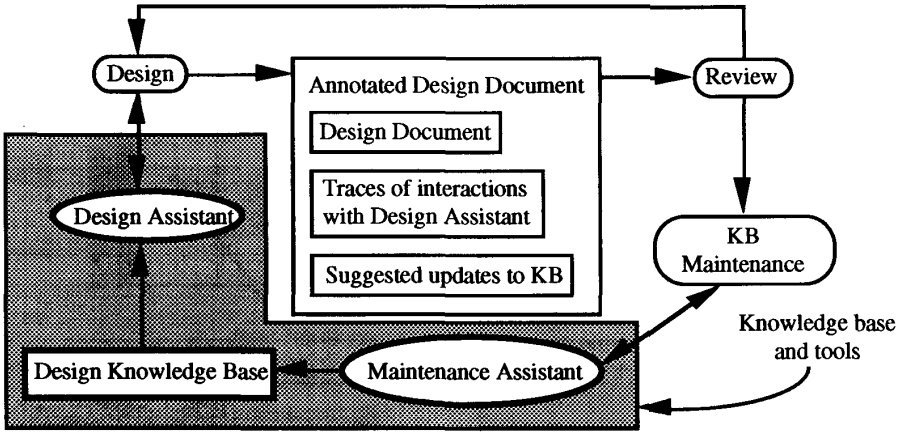

Figure 3: Design process with design assistance and knowledge base maintenance

To summarize this framework, the design knowledge base contains information relevant to design tasks in the application domain. As developers design, the design assistant program helps them to access relevant knowledge. The result of the design process includes the design document, suggestions the developer may have for updates to the design knowledge base, and a trace of the interactions of the developer with the design 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, for example, exceptions to rules or counter-arguments in a design rationale structure [1, 14]. Other issues lead to (proposals for) additions of new knowledge to the knowledge base. All proposals for modifications and additions to the knowledge base that are generated during design and review are collected and sent to a knowledge base maintainer. The knowledge base maintainer interacts with a maintenance assistant program – a knowledge acquisition tool tailored to the design knowledge base – to update the knowledge base, integrating the new design and any additional knowledge generated during design and review.

4. The Design Knowledge Tool

We have designed and implemented a design knowledge tool using the above framework and have integrated this tool into the design process of the software organization. Our first step was to identify a sub-domain of general software design knowledge in which to test our framework and approach. The sub-domain we chose was that of a particular error handling mechanism. This mechanism is part of a multi-layered system of error detection and handling procedures which is critical to the system's fault-tolerance. This error handling mechanism is used if the code being designed ever reaches an illegal state. The mechanism is implemented in the code as a macro call with a number of arguments. These arguments have various effects on the system. For example, one choice of one argument causes a so-called "selective initialization", initializing the processor that is running the process. Other arguments cause the dumping of different kinds of data needed to diagnose the problem or schedule a data-checking audit. Thus, after the user has decided to use this error mechanism, he or she then has 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: programmers typically do not know when to use this 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 mis-apply the construct. (In fact, many of the existing uses in the code base are incorrect.) Second, there do exist local experts who have extensive knowledge about this mechanism. These experts disseminate this information in a frustrating and inefficient manner, i.e., one-to-one communication with individual developers. Third, the domain has some underlying structure that could be used to advantage; i.e., the knowledge is more than a collection of ad-hoc rules. Finally, numerous discussions with software developers convinced us that this domain is very 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 the local domain experts about the knowledge needed to use this mechanism and studying the written documentation that did exist. We took a set of existing examples of this mechanism in the current code base and asked the experts to categorize these examples in terms of *design problems*. This is a very important abstraction step because tool interaction must be in terms that are familiar to the designer, rather than syntactic features of the construct. Presumably designers won't be familiar or comfortable with the latter vocabulary, since it is precisely this they are getting help about.

After several false starts, we succeeded in generating a small number of design problems with almost complete domain coverage. We then distilled the information needed to use the error handling mechanism into a small number of *decisions*. Each decision could be expressed as a succinct question with a small number of answers; for example: "What macro will be used, macro A, B1, B2, B3, C or D?" Then, for each design problem, we elicited from the domain experts advice about how to address each decision needed to use the error handling mechanism. For example, for a "bad parameter problem", how does one decide which macro to use? This may depend on whether the bad parameter is from a function call or a message and on the severity of the error. The experts' advice was distilled into small units of text that we call *advice items*. We indexed individual advice items by both problem and decision to create a *problem/decision grid* illustrated in figure 4:

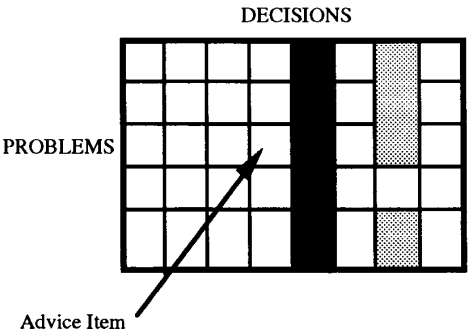

Figure 4: Problem/Decision grid

This graphical representation leads to an interesting observation that impacts the possible representation and organization of advice items in a knowledge base.

Consider the situation where the advice is the same for all design problems, shown by the dark column in figure 4. Instead of having identical advice items for each problem we can remove this redundancy by grouping the associated problems into a new problem category and associating a single advice item with all of the problems of that category. This is especially appropriate when the advice holds over all design problems, as shown by the dark column. Grouping problems based on the commonality of their advice items is an appealing idea and suggests a problem hierarchy based on this commonality. Using a formal knowledge representation [2, 3] this structure can be represented as a classification or description hierarchy, where a problem description at higher levels subsumes a lower level description. For example, the problem description "Data Problem" might subsume both "Bad Function Parameter Problem" and "Bad Value in Message Field Problem". Then, advice items can be associated with either a leaf in the hierarchy (an individual problem) or an internal node in the hierarchy, indicating the applicability of that advice item to all subsumed problems. We discovered some hidden structure in our problem set through the commonality of advice and structured our problem set appropriately.

In addition, consider a situation where an advice item holds over every problem but one, shown by the lighter column in figure 4. Instead of splitting the "exception" problem into a separate class, we can instead introduce the idea of "overriding advice" to handle these cases. That is, we can add an advice item to a node in the problem hierarchy and label it as overriding. What this means is that any advice items associated with any parents of this node and indexed by the same decision are suppressed in favor of this new, more specific item. This mechanism for overriding both further reduces redundancy and makes explicit the notion of "exception" advice, which we found relatively common in our domain.

Finally, we added three more fields to our representation of advice. The first responded to the observation that the advice elicited from the experts tended to come in one of several priority classes. For example, some advice was critical to a particular problem and decision, for example, which macro to use. On the other hand, some advice was general, background knowledge and thus secondary in importance. We use these priorities to order the output of the tool so that primary advice is presented first and secondary advice is presented second, serving to highlight the primary advice. Second, we added an advice identification number. This number is used in maintenance so that an individual advice item can be referred to by a distinct number. Third, we also added an explicit advice item *owner*, which we discuss later. The representation of an advice item is now:

```
Advice-item:
        advice: STRING
        problem: PROBLEM
        decision: DECISION
        override: {TRUE, FALSE}
        priority: {PRIMARY,
                SECONDARY}
        id: INTEGER
        owner: STRING
```

During execution the tool runs the following general algorithm, where A(P) is the set of advice items associated with problem P, Decision(A) is the decision associated with Advice Item A, and the hierarchy is rooted at THING:

```
For each design problem P
    ask whether the user's design
    anticipates this problem;
    if yes
        AdviceSet = NULL;

go up the problem hierarchy and examine all possibly
relevant advice; add new advice that is not overridden by
more specific advice

        while P ≠ THING
            for each AdviceItem A in A(P)
            unless there exists A' in AdviceSet
            such that Decision (A) =
            Decision (A')
                put A in AdviceSet;
            P = PARENT (P);

order advice according to the priority and output

        OrderByPriority (AdviceSet);
        OuputAdvice (AdviceSet);
```

In summary, advice items are represented as structured objects with a number of fields. The actual advice is a text string. Two fields index the advice: the Problem and the Decision fields. The Problem field associates the set of advice with a Problem Hierarchy such that the advice associated with a node applies to all problems subsumed by that node's Problem. The decision field is used to group related advice to see if an override situation exists. When a user indicates a problem is relevant to his or her design, all possible advice items are examined. More specific advice in the Problem Hierarchy may complement or override more general advice. The final set of advice items is ordered by priority and presented to the user.

Having described the acquisition and representation of the knowledge in the tool, we now briefly discuss issues of access and maintenance.

One of the interesting constraints put on this project at the beginning concerned the interface. Because of the

wide variety of terminals used within the target organization, it became clear that the tool should be ASCII-based, i.e., not depend on any specific window or platform features. For this reason we designed both the appearance of the tool and the interaction with the tool to be as simple as possible. When the tool is run, it asks a few introductory questions to identify the user and the design under consideration. It then gives the user several high-level options to choose from. One option begins a dialog with the user about his or her design. It asks a series of yes/no questions about the design, such as: "Does your design anticipate a situation where local and global parameters will become unsynchronized?" If the user responds "yes" to a question, the system will output a formatted list of advice about how to use the error checking mechanism for this situation. The list is grouped by decision and by priority, and is usually about half a page in length. (We also use the size of the screen, both the line length and the number of lines, to format the output properly and control a simple paging mechanism that keeps the advice on the screen until the user is ready to see more.) Finally, when the user is done, the tool asks some simple evaluation questions and gives the user a chance to enter some comments and suggestions about his or her interaction with the tool.

We discussed the overall issue of maintenance in section 3; here, we summarize and make a few more comments. The output of the tool is a script of the interaction with the user. This script is added to the formal design document for two reasons. First, the advice becomes part of the document and will get reviewed during the formal review process. The reviewers will have a chance to make sure that the software developer either followed the advice of the tool or determined that his or her situation was an exception to the situation anticipated by the tool – 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 3.

5. Discussion

This paper is about managing design knowledge to provide knowledge-based assistance for software development. It describes the problem of providing general design knowledge to individual software developers and maintaining that knowledge in an organizationally effective way. After acquiring design knowledge about the use of a specific error-handling mechanism, we represented that knowledge by creating a taxonomy of design problems and associating advice items with nodes in that taxonomy. The taxonomy removes redundancy and facilitates an advice exception mechanism. This knowledge base is accessed by a design assistant program which asks the user questions about his or her design and provides advice as textual output. Maintenance is addressed by having the design advice reviewed as part of the design itself, using an existing organizational process.

This tool has been implemented and tested at AT&T and has been initially deployed as part of AT&T's software development process. This means that all developers involved in software design in one particular large organization use this tool as part of their software development activities. We anticipate a good deal of feedback during the early phases of use. However, we have already tested the tools with a number of individuals in the following way. We created a realistic software design problem and asked people 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.

This tool will provide several benefits. First, the designs produced by software developers will be of higher-quality: the tool's advice will enable developers to know when to choose the error handling construct and use it properly. A related but less tangible benefit is lowering the frustration of developers when trying to find out information. Now, instead of trying to track down a local expert or wade through volumes of written or on-line documentation, the tool gives developers a focused, single source of knowledge. In addition the review process itself will be streamlined. Currently, a developer scheduling a review tends to try to get all potentially relevant human experts to attend this review. This scheduling can often take weeks. Because the tool will guarantee that the developer has at least been exposed to guidelines on error handling, some experts will no longer be required, and scheduling the review will be easier. Finally, demands on the experts' time will decrease. All of these benefits are especially important in very large organizations, where even a small process improvement can make a big difference to organizational output.

The key to the success of a tool of this kind is maintenance. We have taken a number of steps to assure that the maintenance issue is addressed. First of all, we designed the tool with maintenance in mind; thus the representation of the knowledge facilitates maintenance. Redundancy is minimized and because each advice item has a unique identifier and an owner, questions or proposed modifications can be intelligently resolved, i.e. directed to the right person. Second, we have addressed the integration of this tool with current practice in such a way that the output of the tool, and thus the knowledge base itself, is reviewed.

Our tool and approach is consistent with other approaches to introducing knowledge-based technology to support software development. For example, the idea of a "knowledge-based software assistant" [9, 16] proposes

that automation play a supportive role in development; it also suggests that different knowledge-based assistants support different phases of software development (although coordinated through a central repository or project management database). Our tool has exactly this flavor, but it is somewhat narrower in scope than other projects such as the Requirements Assistant [15], the ARIES project [11], and the KBSA Concept Demo [5]. However, it is also deeper in the sense that it both tackles a real-world domain and *addresses the issues of integrating tool use with current organizational practice.* Our tool currently is oriented towards providing direct advice for the use of a limited but important code construct as part of the software design process; it does not assist in general software design. We anticipate looking at the problem of general design assistance, as others have done [8, 17, 18, 19, 20].

We have several plans for the future. First of all, we are exploring the application of our general framework to other software development problems within AT&T. One aspect of our approach which enhances its generality is that the nature of an "advice item" is not constrained: as a small piece of text, it can contain technical advice, suggestions to run another tool or investigate another source of information, or even a suggestion to go talk to a specific individual! In general, we anticipate various knowledge-based advice tools within the software development process. Second, the issue of an adequate maintenance tool will become more and more important as our current knowledge base needs maintenance and as the technology gets used more within the software development organization. A variety of issues arise in designing such a tool. First of all, given that the purpose of such a tool is to add, remove, or modify knowledge, the knowledge must be both modular and highly connected to the context to which it is relevant. This means the tool must support both the modification of knowledge as such and also the "editing" of its context. For our tool, a maintainer must be able to change the binding of a piece of advice to a position in the problem hierarchy, and easily add new advice and manipulate the priority and overriding fields of existing advice items.

Envisioning the existence of multiple tools such as ours raises some other difficult issues of maintenance and coordination. It will almost certainly be the case that knowledge within one tool will be relevant to the domains of other tools; i.e., they will not be fully separable. Keeping multiple knowledge bases synchronized and up-to-date is similar to the analogous problem in databases [7]. Sharing knowledge, either in a distributed fashion or through a central repository, is a difficult problem which we anticipate addressing in the future.

6. Conclusions

We have identified the crucial problem of managing design knowledge in large scale software development; described a particular approach to this problem; and presented an implemented tool for solving this problem in one particular domain. The solution involves acquiring and representing design knowledge, providing appropriate access to the knowledge for those who need it, and ensuring that the knowledge is adequately maintained. Our approach emphasizes providing design advice to humans at appropriate times; in order to do this, we had to both integrate the tool with the existing software process and present the advice in an appropriate way. Maintenance of the knowledge is facilitated by the underlying representation and its integration with an existing organizational review process.

7. References

1. Bailin, S.C., Moore, J.M., Bentz, R., and M. Bewtra, KAPTUR: Knowledge Acquisition for Preservation of Tradeoffs and Underlying Rationales, *Proceedings of the 5th Conference on Knowledge-Based Software Assistant,* pp. 95-104, 1990.

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

3. Brachman, R.J., McGuinness, D.L., Patel-Schneider, P.F., Resnick, L.A., and A. Borgida, Living with CLASSIC: When and How to Use a KL-ONE-Like Language, in: *Formal Aspects of Semantic Networks*, J. Sowa, Ed., Morgan Kauffman, 1990.

4. Brooks, F.P., No Silver Bullet: Essence and Accidents of Software Engineering, *IEEE Computer Magazine*, April, 1987.

5. Cabral, G. and M. DeBellis, Domain-Specific Representations in the KBSA Concept Demo, *Proceedings of the 6th Annual Knowledge-Based Software Engineering Conference,* Syracuse, NY. September 1991, pp. 97-106.

6. Clancey, W., The Frame of Reference Problem in the Design of Intelligent Machines. In van Lehn, K., ed. *Architectures for Intelligence: The Twenty Second Carnegie Symposium on Cognition*, Lawrence Erlbaum Associates, 1991.

7. Elmasri, R. and S.B. Navathe, *Fundamentals of Database Systems*, Benjamin-Cummings, 1989.

8. Fischer, G., Grudin, J., Lemke, A., McCall, R., Ostwald, J., & Shipman, F. 1991. Supporting Asynchronous Collaborative Design in Integrated Knowledge-Based Design Environments. Department of Computer Science, University of Colorado.

9. Green, C., Luckham, D., Balzer, R., Cheatham, T., and C. Rich, "Report on a Knowledge-Based Software Assistant", *Rome Air Development Center report RADC-TR-83-195*, August 1983.

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

11. Johnson, W.L., Feather, M.S., and D.H. Harris, The KBSA Requirements/Specification Facet: ARIES, *Proceedings of the 6th Annual Knowledge-Based Software Engineering Conference* (Syracuse, NY. September 1991), pp. 57-66.

12. Mark, W., & Schlossberg, J. 1990. Interactive Acquisition of Design Decisions. Proceedings of the 5th *Knowledge Acquisition for Knowledge-Based Systems* workshop. Banff, Canada.

13. Musen, M.A., Conceptual Models of Interactive Knowledge Acquisition Tools, *Knowledge Acquisition* 1: 73-88, 1989.

14. Ramesh, B. and V. Dhar, Representation and Maintenance of Process Knowledge for Large Scale Systems Development, *Proceedings of the 6th Annual Knowledge-Based Software Engineering Conference* (Syracuse, NY, September 1991), pp. 223-231.

15. Reubenstein, H.B. and Waters, R.C., The Requirements Apprentice: an initial scenario, *Proceedings of the 5th International Workshop on Software Specification and Design* (Pittsburg, Penn., May 19-20, 1989) In ACM SIGSOFT Engineering Notes 14(3), May 1989.

16. Rich, C.H., & Waters, R.C., *The Programmer's Apprentice*. Addison-Wesley, 1991.

17. Silverman, B.G. and T.M. Mezher, Expert Critics in Engineering Design: Lessons Learned and Research Needs, *AAAI Magazine*, Spring, 1992.

18. Terveen, L.G. 1992. Intelligent Systems as Cooperative Systems. In Thomas, P., Ed. Special Issue of the *International Journal of Intelligent Systems* on *The Social Context of Intelligent Systems*. (forthcoming).

19. Terveen, L.G., & Wroblewski, D.A. 1991. A Tool for Achieving Consensus in Knowledge Representation. Proceedings of *AAAI-91*, pp. 74-79,. 1991

20. Waters, R.C. and Y.M. Tan, Toward a Design Apprentice: Supporting Reuse and Evolution in Software Design, submitted to the 1991 IFIP TC2 Working Conference on Constructing Programs from Specifications.