Preserving Knowledge in Design Projects: What Designers Need to Know

James D. Herbsleb
The University of Michigan
701 Tappan Street
Ann Arbor, MI 48109-1234, USA
herbsleb@csmil.umich.edu

Eiji Kuwana
NTT Corporation
1-9-1 Kohnan Minato-ku
Tokyo 108 JAPAN
kuwana@mickey.ntt.jp

ABSTRACT
In order to inform the design of technology support and new procedural methods for software design, we analyzed the content of real design meetings in three organizations, focusing in particular on the questions the designers ask of each other. We found that most questions concerned the project requirements, particularly what the software was supposed to do and, somewhat less frequently, scenarios of use. Questions about functions to be performed by software components and how these functions were to be realized were also fairly frequent. Rationales for design decisions were seldom asked about. The implications of this research for design tools and methods are discussed.

KEYWORDS: Design tools, design methods, design rationale, user scenarios.

INTRODUCTION

The difficulty, expense, and unpredictability of large software development projects are so well known and so widely discussed that the term "software crisis" has become passé. This is not because the difficulties have been overcome -- there has been no silver bullet [1] -- but rather, we suspect, because the difficulty of the task is no longer a surprise. Empirical research has a major role to play in the process of bringing about incremental improvements. As we attain a better understanding of the cognitive and organizational demands of large software development projects, we are in a better position to introduce methods and tools which are precisely tuned to the biggest problems.

One of the suggestions most often heard is to provide developers with access to more knowledge about various aspects of the development project. But the views about precisely what knowledge to provide are many and diverse. Here are a few of the major contenders:

Rationale for Design Decisions

Much attention is currently focused on methods, notations, and tools for recording rationales for design decisions.

What is represented in this approach is not primarily the application domain or the system design itself, but rather the space or history of arguments surrounding the actual decisions made as development progresses (see [17]). The most commonly advocated framework for selecting and organizing this kind of data is argument structure (e.g., gIBIS, [3], SIBYL [14], and QOC [16]). It typically includes nodes such as *issue*, *alternative*, *argument*, *criterion*, *goal*, and *claim*. These are linked up into structures by relations like *achieves*, *supports*, *denies*, *presupposes*, *subgoal-of*, and *subdecision-of*. The most expressive language to date is Decision Rationale Language (DRL)[15], which includes all of these and more. What is represented is the "rhetorical" space around decisions, and structure is created by links which have strictly rhetorical significance. If this sort of information is found to be sufficiently useful, it could be maintained independently or integrated with traditional design representations (e.g., [20]).

Knowledge of application domain

In a major study of software development projects, Curtis, Krasner, & Iscoe [4] found that one of the problems that was most salient and consistently troublesome was "the thin spread of application domain knowledge." Particularly rare and important was command of the larger view, i.e., the integration of all the various and diverse pieces of domain knowledge. This was essential for creating a good computational architecture, and for forging and communicating a common understanding of the system under development.

Recently, there has been increased attention to analysis of problem domains and representing domain knowledge (see, e.g., [5]). Methods using such notations support the representation of the problem domain in terms of nodes like *entities*, *objects*, *processes*, or *data structures*, and links such as *data flow*, *control flow*, *relations*, *inherits*, *subclass-of*, and so on. The basic idea is to represent the domain and the system, generally in terms which domain experts would understand.

Scenarios of use

Closely related to application domain knowledge is knowledge of scenarios of use. In contrast to general domain knowledge, knowledge of scenarios of use concerns the ways in which the system will need to fit into the dynamic flow of activities in its environment. As noted by Guindon [7], scenarios of use are one of the major kinds of knowledge developers bring to bear in designing software. These scenarios are very important for understanding the requirements, and appeared to play a role in the sudden unplanned discovery of partial solutions. In a similar vein, Curtis et al. [4] also concluded from their extensive interviews with software developers that scenarios of use were very important for understanding the behavior of the application and its relation to its environment. Yet they observed that while it is common for customers to generate scenarios as they are determining their requirements, they very seldom pass them on to the developers. As a consequence, the developers had to generate their own scenarios, and could only predict the obvious ones and not ones which created unusual conditions. There is also anecdotal evidence that scenarios of use are very helpful in the user interface design process [12].

Scenarios of use could be made available to designers in several ways. At least one software engineering method, *Objectory* [10] explicitly incorporates scenarios of use ("use cases") as a central part of the method. There are also other, less formal techniques, (e.g., [11]) for making this kind of knowledge available during design. Finally, techniques that get users actively involved in the design process (e.g., [18]) may serve, among other purposes, to inject knowledge of user scenarios into the design process.

Knowledge generated by design methods

Finally, there are many software design methods, each with an associated notation, and embedded in rules-of-thumb, principles, and a development philosophy. They fall within several broad categories, including structured analysis and design, entity-relation modeling, and object-oriented design. There are many claims by advocates of these techniques, and also some empirical evidence, e.g., from research on software errors [19], that these methods can have a significant positive impact on the development process. It is unclear how much of this effect is attributable to an improvement in the ongoing design process and in the quality of the design decisions made, and how much is attributable to capturing knowledge in the system's notation so that the knowledge can be used at later stages. But it seems very plausible that capturing this sort of knowledge could significantly impact the later stages of development.

What is really needed?

Each of these ideas for capturing project knowledge and making it available for later use embodies empirical hypotheses about the knowledge needs of the software design process. Testing these claims should be given top priority, since they determine the potential of various classes of tools to make positive contributions to the design process.

Unfortunately, it is very difficult to test these hypotheses directly, i.e., by building an appropriate tool then designing a software system and assessing the results. The expense, risk, and the difficulty of interpreting the results of complex processes in the real world make this option untenable. Laboratory studies solve some of these problems by isolating the effects of selected variables, but they do not provide the opportunity to take advantage of many of the potentially most beneficial features of knowledge-preserving tools and techniques, since there is generally no realistic organizational or project history in a laboratory context. Preserving small quantities of knowledge for the duration of a typical experiment, i.e., an hour or two, is radically different from preserving potentially enormous quantities of knowledge for more realistic time periods of months to years.

This research attempts to inform this issue by taking a different approach. To begin to assess the basic knowledge needs in upstream software development, we examine the *questions* that arise in actual requirements specification and design meetings among software engineers. The central assumption is simply that the questions asked in these meetings by experienced, professional software designers are a reasonably good indicator of the kinds of knowledge that an ideal method should make available. It is certainly not a perfect indicator, since designers may be unaware of their lack of information, or they may be asking a question just to test their understanding. We assume, however, that asking a question very often indicates that the asker believes the answer contains knowledge important in the immediate context, and that the asker does not currently possess this knowledge.

In a previous paper [13], we briefly described this method, and presented basic frequency data on questions concerning software requirements. Here we provide a more detailed description of our method, present data concerning questions about all of the software development stages, and draw out the implications of our findings for design tools and methods.

METHOD

Data Profile.

We use two basic kinds of data in this study. The first is a set of minutes from 38 design meetings at Nippon Telegraph and Telephone Corporation (NTT) Software Laboratories that took place over an eight month period. The task was to specify requirements and design for version 1.5 of an existing software development environment. The meetings from which our data are drawn involved external behavior analysis and preliminary design. Individual members of the team wrote the minutes, generally a day or two after the meeting, using their notes and documents from the meeting. The chore of taking minutes rotated through the development team.

This corpus of data covers a substantial continuous period of time on a large re-design project. One potential weakness of this data stems from the fact that it is filtered through and reconstructed by the individual taking minutes. Presumably, this will not cause too much distortion, since minutes customarily capture the most important points, and the minute-takers were experts in the software design domain. But the second data source was included, in part, to compensate for these possibilities.

The second type of data we used is videotape protocol data gathered in the United States from three software requirements and preliminary design meetings. Each meeting had software requirements and/or preliminary design as its primary activity, had either four or five participants, and lasted from slightly under one hour to slightly over two hours. These particular meetings were selected, in part, to span early requirements through preliminary design phases of development.

Two of the meetings were teams at Andersen Consulting (AC). One was a preliminary design meeting concerned with specifying a client-server architecture to be used by Andersen to build systems for a variety of customers. The other AC meeting, involving a different team, was concerned with detailed requirements of "reverse engineering" software which would heuristically identify and describe structure in large, old, unstructured, assembly-language programs. In the third meeting, a team at Microelectronics and Computer Corporation (MCC) was an early discussion of the requirements for a knowledge-base editor, trying to determine its basic functionality.

As one would expect, the three organizations from which the data are taken differ with respect to development methods. NTT's development process was governed by internal NTT guidelines similar to those published by IEEE (e.g., [8, 9]. These guidelines spelled out what documents must be created and what each should contain. The development style was based on Composite Design Methods and SA/SD. The Andersen Consulting projects made use of Method/1, a proprietary method with very detailed specification of required documents and deliverables. The style tended to be process-oriented, postponing consideration of data structures. Development on the MCC project was in the context of a research-oriented artificial intelligence project, and was thought to be much less structured than in the other two settings.

These two data sets complement each other. The videotape data are unfiltered and unreconstructed, and so do not suffer from those potential sources of distortion. The chief disadvantages of the videotapes are first, that we have no real way of knowing which of the questions we identify would be considered important by the software engineers themselves; and second, these are only three brief snapshots of three different projects, a sample with many potential biases. The NTT minute data compensates for these weaknesses, since it is a continuous eight month sample of questions deemed important enough to record.

Data Analysis.

As we mentioned above, our basic assumption is that the questions software engineers ask provide a good heuristic for identifying knowledge that should be preserved and made available to designers. We extracted from our data not only explicit questions, but also implicit requests for information, including statements of ignorance that were interpreted as questions. We excluded such things as rhetorical questions, questions intended as jokes, questions that were embedded in digressions and clearly bore no relationship to the task, requests for action that were worded as questions, and questions that asked for a restatement of something that was badly worded or just not heard clearly.

Once we had identified the questions, we coded them according to the following scheme.[1] First, we identified one or more **targets** for each question. A **target** is simply the thing, happening, or task that the questioner was asking about. So, for example, if a question asked about a particular component of the design, that component is a **target**. Many questions had more than one **target**.

Second, we coded each target according to the **attribute** which the question referred to. We adopted a simple classification of target attributes into **who**, **what**, **when**, **why**, and **how**. This turned out to be a simple, yet meaningful and comprehensive set of categories. In brief, we used the following criteria to determine the attribute: Questions about who built a target or performed a task, or about skills needed, were coded as **who**. **What** questions concerned the external behavior or function of a target, i.e., what it was or what it did, without regard to how that function was actually carried out. **How** questions focused on the particular way that a target carried out its function or the way a task was performed. For example, a question about how a user would accomplish a user task with the functionality described in a particular software requirement, would be coded **how**. Questions about deadlines and scheduling were coded as **when**. Finally, questions asking why some decision was made, or about an evaluation that was assigned or might be assigned to some alternative, or soliciting a comparison of alternatives, or arguments about alternatives were coded as **why**. If a question referred to two or more attributes of a single target, each was coded separately and is reflected in our results.

Next, we categorized the target according to the **stage** in the traditional software life cycle in which the target was (or would be) created. We used a scheme which included **requirements specification**, **design**, **implementation**, **testing** and **maintenance**. We used software engineering textbooks (e.g., [6] and IEEE guidelines [8, 9] to help define these stages. In general, descriptions of what the

[1] Additional details are available from the authors.

software system, as a whole, is supposed to do are **requirements**. **Design**, on the other hand, concerns determining the modules into which the system will be decomposed and the interfaces of these modules (preliminary design), and the ways in which their functionality is to be realized (detailed design). **Implementation** was defined just as writing and compiling statements in a programming language, and was relatively easy to identify. **Testing** was also straightforward. The date the software was released marked the beginning of the **maintenance** phase. See Table 1 for some example questions, the targets we identified from the questions, and the attributes and target creation stages.

Example Question	Target(s) for that example	Attribute(s) asked about	Stage when target would be created
What kinds of high level interfaces do we want to support?	specifications for high-level interfaces	What	Requirements
You want the user to specify either that he wants everything or a range of the diagram, right?	specification for the way a user designates part of a diagram	How	Requirements
What is the correct behavior [of a component] in [a given situation]?	behavior of a software component	What	Design
And that's [data structure] used by context management?	use of data structure	How	Design
Why should I have two tasks running simultaneously when I want to get to local data?	simultaneous tasks	Why	Design
If I [i.e., a user] have a diagram on the screen, what do I need to do to print it?	specification for printing a diagram	What & How	Requirements

Table 1. *Typical questions from **Design** and **Requirements** stages. The last question mentions both the **what** and the **how** attributes of the target. The relation between these attributes is **realize**, since it is asking about the way some functionality will be accomplished, or **realized**.*

We also wanted to see how the knowledge needs of a software design team changed over time. As mentioned above, the videotapes were selected in order to have an example of a meeting in early requirements specification, late requirements specification, and preliminary design. The minutes were taken from 38 meetings which spanned these same stages. In order to divide the questions from these meetings (to a rough approximation) into these same three stages, we simply put the questions in temporal order and divided them into thirds. In this way, we were able to look at how distributions of target and question types changed over these early project stages.

In order to establish the reliability of our coding, we independently coded the **attributes** and **target creation stages** of three samples of questions, and obtained interrater agreement rates from 68-73%. As we discussed our differences, we discovered that they were nearly always due to a failure of the person less familiar with a dataset to understand the terms or the context of the question, or to language problems in translating between Japanese and English. Upon discussion, we agreed in virtually every case. We each then coded the dataset with which we were most familiar, so we believe the agreement rates substantially underestimate the accuracy of the coding, and are acceptable for data of this type.

As mentioned earlier, many questions had more than one target. Targets were not randomly bundled in a single question, but rather the targets were generally related in some way, and the relation was an important, often the central, aspect of the question. In order to investigate these relations, we categorized them into one of five categories: 1) **evolve** is the relation between an earlier and later version of a component, 2) **task assignment** is the relation between persons and tasks they are performing, 3) **interface** is the relation between communicating components or systems, 4) **realize** is the relation between a higher-level function or behavior and the lower-level pieces which actually carry it out, and 5) **same** is a question about whether targets are identical in some way. In order to establish reliability of this coding, we separately coded a sample of questions and achieved an agreement rate of over 90%.

RESULTS

One of the most interesting and surprising findings is the extraordinary degree of similarity in our results between the two datasets. Table 2 gives the correlations between the videotape and minute data for the basic frequencies we report. This degree of similarity was quite unexpected, given the enormous differences between the projects from which the data were drawn.

Target Creation Stage Frequencies	.98
Target Attribute Frequencies	.97
Relation Frequencies	.92

Table 2. *Correlations between the basic frequencies for the two datasets.*

Target Characteristics.
In both datasets, as one would expect, targets created in the **requirements** stage were by far the most frequently asked about. (61%), and **design** was a distant second (36%). None of the other stages exceeded 1.5%. This is not terribly surprising, since the projects themselves were in the requirements and early design stages. On the other hand, it is a little surprising that targets which would be created during the later stages were almost **never** asked about.

As the projects themselves moved from early requirements into the design phase, the percentage of **requirements** targets declined linearly from 81% to 48%, while the **design** targets rose from 19% to 52% (chi-squared = 89.48, df=8, p=.0001).. The direction of change was expected, but it is significant that even well into the design stage, nearly half the targets asked about were **requirements**.

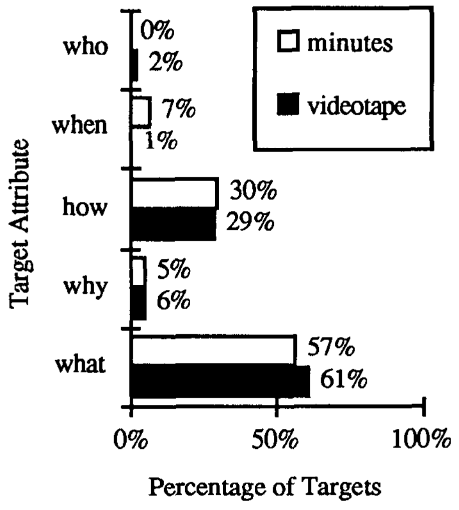

Figure 1. *Percentage of targets for which the given attributes were asked about.*

Figure 1 shows that the **what** attribute was asked about much more often than any other, with **how** also at a relatively high frequency. So the engineers asked about twice as many questions about the basic functionality or external behavior of a target as they did about the details of **how** it would work. This would certainly seem to support the notion that understanding what the software is supposed to do is a bigger problem than figuring out how to make it behave properly once "properly" is understood.

These values changed somewhat over time. **What** targets increased from 55% to 69%, while **how** declined from 39% to 25% (chi-squared = 34.96, df=8, p=.0001).. **Why** remained at a constant 6%. So **how** questions were generated most often in the requirements stage of the project, asking, for example, how a user would do X with a given set of system functions.

One of the biggest surprises here is the relatively low frequency of **why** questions. This is the sort of knowledge that design rationale notations are designed to capture, and given the very high level of interest and expected benefits from such systems, we anticipated that we might see a great many **why** questions.

Table 3 shows the most frequently occurring pairs of **attributes** and **target creation stages**. By far the most frequent is the **requirements-what** combination, with **requirements-how, design-what,** and **design-how** each around one-third as frequent.

	Requirements	Design
what	404 / 43%	153 / 16%
why	33 / 4%	20 / 2%
how	118 / 13%	156 / 17%

Table 3. *The frequencies / percentages of the six most frequent combinations of* ***attributes*** *and* ***target creation stages*** *(out of 940 total targets). No omitted cell contains more than 1.5% of the targets.*

Relations Between Targets.
About half (48%) of the questions in our sample had multiple targets. Nearly all of these (97%) had two targets, a few had three, and one had four. In all, nearly two-thirds (65%) of our targets appeared in multiple-target questions.

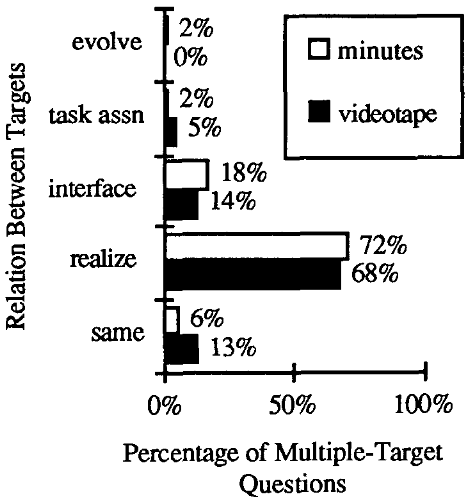

Figure 2. *Distribution of relations among targets in multiple-target questions.*

By far the most frequent relation among targets, as shown in Figure 2, was **realize**, with a significant portion of **interface** and **same**, but very few **task assignment** and **evolve** relations. Clearly, **realize** is a very broad category, including, e.g., the relation between an external behavior and software components, a module function and an algorithm, a function and OS calls, and so on. The extremely high frequency of **realize** relations is perhaps best illustrated by the fact that 30% of *all* targets in our data (278 out of 940) enter into a **realize** relation. We take this very high frequency of questions concerning the **realize** relation as an indicator of its importance, so we decided to examine the attributes of the targets that enter into this relation more closely.

In order to perform this additional analysis, we extracted only those questions which involved the **realize** relation. Disregarding the very few questions involving more than two targets, each of the two-target questions involves a pair of attributes, one for each target, e.g., **what-how**. Examining the frequencies of these pairings gives us an indication of the kinds of **realize** questions most often asked. It is also instructive to look at the creation stages of the targets, to see, for example, if the designers are asking most frequently about **realizing** some **requirement** in the **design**, or **realizing** a user **requirement** with given system functions.

Tables 4 and 5 show the results of these analyses. Table 4 reveals that over 90% of the questions involving the **realize** relation, targets have one of three pairs of attributes: **what-how**, **what-what**, and **how-how**. (Each of the other pairings accounts for less than 3% of the total.)

What-How	121	59%
What-What	57	28%
How-How	13	6%
All Others	13	6%

Table 4. *The frequencies and percentages of attribute pairs for targets joined by **realize*** relation.

Table 5 shows that most of these relations join targets created in the requirements stage. In particular, by far the most common occurrence of a **realize** relation is in questions

Requirements-Requirements	107	52%
Requirements-Design	47	23%
Design-Design	44	22%
All Others	6	3%

Table 5. *The frequencies and percentages of target creation stage pairs for targets joined by the **realize** relation.*

with **requirements-how** and **requirements-what** targets (33%, or 69 of 208 total questions with **realize** relations). These questions asks about the particular ways (**how**) a user would accomplish goals using some particular function of the system under design (**what**). For example, "How would you [i.e., a user] use it [some functionality to be provided by the system]?" These data clearly show that user scenarios are a frequently asked about type of information in software design.

Summary of results.

- Different types of data from software design meetings in different corporations and even different countries showed an astonishing degree of similarity in the frequency with which different types of questions were asked.
- Most questions in our sample of software design meetings concerned the **requirements**. In particular, developers tended to ask questions about **what** the **requirements** are, and this continued to be the most frequent sort of question as the project progressed from early requirements definition through preliminary design.
- User scenarios were frequently asked about. This is shown both by the significant overall percentage of **requirements-how** targets, and by the high proportion of multi-target questions which ask how a user will make use of some particular functionality of the system.
- Most questions concerned **what** function the target was to perform and **how** it would be performed.
- Very few questions asked **why** a decision was made, or solicited evaluations or comparisons of alternatives.

DISCUSSION

This degree of similarity between the questions taken from the minutes of design meetings at NTT and from videotaped design meetings at AC and MCC is quite startling. The questions in the minutes were filtered through a scribe, and represent an extended and continuous sample of a single subgroup on a single project. The videotaped data is unfiltered and unreconstructed, and is taken from three unrelated meetings. The data come from different projects, different corporations, and even different countries.

This similarity is important for two reasons. First, it greatly strengthens the findings. Any single data set is subject to many biases, and may be atypical with regard to software design in general. But similar results with widely different kinds of data suggest that the findings have considerable generality. Second, we think it is very important to establish a baseline against which questions from meetings supported with different sorts of tools, or using different methods, can be compared. The uniformity in our results gives us considerable confidence that they will be useful for this purpose.

As we mentioned earlier, a result that was particularly unexpected is the low frequency of **why** questions. There are several possible explanations for this finding. One is that the kind of information elicited by **why** questions, i.e., the rationale behind decisions, is simply relatively unimportant.. This certainly runs counter to the intuitions of many individuals experienced in software development, but it is not ruled out by our data. A variation on this theme is that this information is simply *perceived* to be unimportant, and perhaps even actively avoided by designers wishing to escape the overhead of becoming domain experts. A second possibility is that **why** questions and the information they elicit are very important, but they are relatively unlikely to arise in meetings as compared with other settings in which design work is done. One plausible line of reasoning is that in meetings, the context, as well as the content, is generally clear to all the participants. **Why** questions may often be used to establish this context when it is unclear. A third possibility is that the information that could be directly elicited with a **why** question is often elicited with **how** or **what** questions. If one knows enough about the possible rationales behind a decision, one may be able to infer the correct rationale by using clues obtained in this indirect way. If this turns out to be the case, it suggests that there is considerable overlap between design rationale tools (focusing on **why** questions) and other design tools which focus on creating the design itself. In other words, a good representation of the **what** and **how** of the design may enable one to infer many of the **whys** Finally, it may be that **why** questions are seldom asked in meetings because the participants realize that they cannot generally be answered in current practice, with current tools. This interpretation, of course, suggests that representations of design spaces or histories would often be consulted if available.

One suggestion concerning the **why** questions that we find somewhat less plausible than the ones just discussed is that although **why** questions are low in frequency, they are more important than other kinds of questions. Our skepticism stems from the observation that the percentage of **why** questions is nearly identical in the minutes and the videotapes. If the **why** questions tended to be more important than the other types of questions, one would expect to see them represented more often in the minutes, since the questions recorded there have been filtered by a scribe and selected for their importance. The nearly identical frequencies imply that the **why** questions in our sample were not more important than the other questions, at least as importance was judged by the scribes.

In any case, it is clear that more research is needed to sort out all of these importantly different possibilities. Given the extremely high level of interest in design rationale notations and tools, it is critical to begin to look at how, when, and in what settings such representations might be most useful. Without such research, there is a grave risk of building tools that provide the answers to the wrong questions.

The fact that the **requirements** are very often asked about supports those who have suggested that particular attention should be paid to tools, methods, and notations for this part of the software life cycle, e.g., [4, 5]. The most frequent single type of target asked about (43% of all targets) is simply **what** the system is supposed to do, i.e., **what** the **requirements** are.

The data also strongly suggest that the scenario of use is an extremely important type of information (see, e.g., [11]). What makes this finding particularly significant is that with only a few exceptions that we are aware of (e.g., [10]), software design methods and notations do not provide rich facilities for representing user scenarios. Data-flow diagrams, for a typical example, represent users as a simple node, a "terminator" (see, e.g., [22] pp. 64-73), which functions as a source and a destination for flows of data. There is typically no simple way to represent expected sets of interactions with users. We suggest that this is a relatively neglected area of potentially great importance. In the area of user interface design, there are a number of notations which can be used for expressing scenarios of use (e.g., GOMS [2] and UAN [21]). Although for questions that arise in upstream software design, these notations are often too fine-grained, but extensions or analogs might be very useful.

The high frequency of questions about **realize** relations also suggests that notations and tools for design should optimize for retrieval and display of this relation and the objects (or components or functions, etc.) that enter into **realize** relations. This property is often called *traceability*, and the high frequency of **realize** relations supports those who stress its importance.

ACKNOWLEDGMENTS
This work has been supported by the National Science Foundation (Grant No. IRI-8902930), and by the Center for Strategic Technology Research (CSTaR) at Andersen Consulting, and by a grant from the Center for Japanese Studies at the University of Michigan. We would particularly like to thank Libby Mack, Nancy Pennington, Barbara Smith, and Bill Curtis for their help in the collection and analysis of the data. We also wish to acknowledge the important contribution of the researchers at NTT Software Laboratories who made data available to us. We would also like to thank Kevin Crowston, Michael Knister, Gary M. Olson, Judith S. Olson, and Atul Prakash for their valuable comments and suggestions.

REFERENCES

1. Brooks, F.P., *No silver bullet.* IEEE Computer, 1987. **20**: p. 10-19.

2. Card, S.K., T.P. Moran, and A. Newell, *The psychology of human-computer interaction.* 1983, Hillsdale, NJ: Erlbaum.

3. Conklin, E.J. and K.C.B. Yakemovic, *A process-oriented approach to design rationale.* Human-Computer Interaction, 1991. **6**: p. 357-391.

4. Curtis, B., H. Krasner, and N. Iscoe, *A field study of the software design process for large systems.* Communications of the ACM., 1988. **31**: p. 1268-1287.

5. Davis, A.M., *Software requirements: Analysis and specification.* 1990, Englewood Cliffs, NJ: Prentice Hall.

6. Ghezzi, C., M. Jazayeri, and D. Mandrioli, *Fundamentals of software engineering.* 1991, Englewood Cliffs, NJ: Prentice Hall.

7. Guindon, R., *Knowledge exploited by experts during software system design.* International Journal of Man-Machine Studies, 1990. **33**: p. 279-304.

8. IEEE. *Guide for software requirements specifications,* 1984, Std 830-1984.

9. IEEE *Recommended practice for software design descriptions,* 1987, Std 1016-1987.

10. Jacobson, I., *Object-oriented software engineering.* 1992, Reading, MA: Addison-Wesley.

11. Karat, J. and J.L. Bennett, *Using scenarios in design meetings -- a case study example,* in *Taking software design seriously,* J. Karat, Editor. 1991, Harcourt Brace Jovanovich: Boston. p. 63-94.

12. Karat, J. and J.L. Bennett, *Working within the design process: Supporting effetive and efficient design,* in *Designing interaction: Psychology at the human-computer interface,* J.M. Carroll, Editor. 1991, Cambridge University Press: New York. p. 269-285.

13. Kuwana, E. and J.D. Herbsleb. *Representing knowledge in requirements engineering: An empirical study of what software engineers need to know.* in *IEEE International Symposium on Requirements Engineering.* 1993.

14. Lee, J. *SIBYL: A tool for managing group design.* in *CSCW '90.* 1990. Los Angeles:

15. Lee, J. and K.-Y. Lai, *What's in design rationale.* Human-Computer Interaction, 1991. **6**: p. 251-280.

16. MacLean, A., *et al., Questions, options, and criteria: Elements of design space analysis.* Human-Computer Interaction, 1991. **6**: p. 201-250.

17. Moran, T. and J. Carroll, ed. *Design Rationale.* in press.

18. Muller, M.J. *Retrospective on a year of participatory design using the PICTIVE technique.* in *CHI '92.* 1992.

19. Nakajo, T. and H. Kume, *A case history analysis of software error cause-effect relationships.* IEEE Transactions on Software Engineering, 1991. **17**: p. 830-837.

20. Potts, C., *Supporting software design: Integrating design processes, design methods, and design rationale,* in *Design Rationale,* T. Moran and J. Carroll, Editor. in press,

21. Siochi, A.C., D. Hix, and H.R. Hartson, *The UAN: A notation to support user-centered design of direct manipulation interfaces,* in *Taking software design seriously: Practical techniques for human-computer interaction design,* J. Karat, Editor. 1991, Academic Press: Boston. p. 157-194.

22. Yourdon, E., *Modern structured analysis.* 1989, Englewood Cliffs, NJ: Yourdon Press.