Vol. 39 No. 10 Transactions of Information Processing Society of Japan Oct. 1998

Regular Paper

An Empirical Study of Information Needs in Collaborative Software Design

JAMES D. HERBSLEB† and EIJI KUWANA††,☆

The increasing scale of software structures together with current social and technical factors all point to a new software development style featuring collaboration. The difficulty of software development is greatly increased by the social and organizational context in which it takes place. One type of solution often suggested to help alleviate these difficulties is providing tools, methods, or techniques which give developers access to more information about the project in the form of design rationale, knowledge about the problem domain, user scenarios, or the software design. In order to try to determine the kinds of information software developers in a team *actually* need, we examined questions experienced developers asked each other in actual development *meetings* during the upstream activities in several projects. We found that developers most often ask about what the requirements are, how users will interact with the system, what the functional definitions and interfaces of the software modules are, and how the functionality of these modules will be realized. In addition, the mix of questions changed substantially as the projects moved from early requirements definition to preliminary design. Based on these results, we propose a set of empirically-based suggestions for the kinds of assistance *design teams* need in getting the information they require.

1. Introduction

To begin by stating the obvious, group software design is a hard problem. Not only is it intellectually challenging[3),10),15)], but the social, organizational, cultural, and business contexts in which the work inevitably takes place add enormously to the complexity of the task[2),7)]. In particular, communication and coordination, adapting to changing requirements, and disseminating sufficient knowledge of the application domain are pervasive sources of difficulty in real world collaborative software design.

It is often suggested that tools which facilitate recording, searching, and retrieving important information would provide valuable assistance in overcoming these context-generated problems. But there exists a wide variety of opinions on precisely what sort of information would be most useful. The most frequently discussed possibilities seem to be *design rationale*, *knowledge of the application domain*, *user scenarios*, and *knowledge generated by design methods*.

1.1 Rationale for Design Decisions

In Computer Supported Collaborative Work (CSCW), Software Engineering, and HCI (Human Computer Interaction) areas, 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 Ref. 22)). A prototypical sort of question that such representations are designed to answer is a **why** question about a design alternative and the basis for choosing or rejecting it. The actual evaluations of the alternatives, the criteria for evaluating them, and their organization into design issues are additional layers of structure that may also be explicitly represented[19)].

The most commonly advocated framework for selecting and organizing this kind of data is argument structure (e.g., gIBIS[6)], SIBYL[18)], and QOC[20)]). It typically includes nodes such as *issue*, *alternative*, *argument*, *criterion*, *goal*, and *claim*. **Figure 1** shows QOC[20)] vocabulary. *Questions* are key issues which shape the argument spaces, *Options* are alternative solutions and *Criteria* are for comparing and evaluating *Options*. These are also linked up into structures by relations. For instance gIBIS[6)] has links such as *achieves*, *supports*, *denies*, *presupposes*, *subgoal-of*, and *subdecision-of*. So far, some tools such as gIBIS and rIBIS, which provide a functionality of argument structure, have been provided and evaluated. The most expressive language to date is Decision Rationale

† Lucent Technologies, Bell Laboratories
†† NTT Software Laboratories
☆ Presently with NTT Multimedia Service Promotion Headquarters

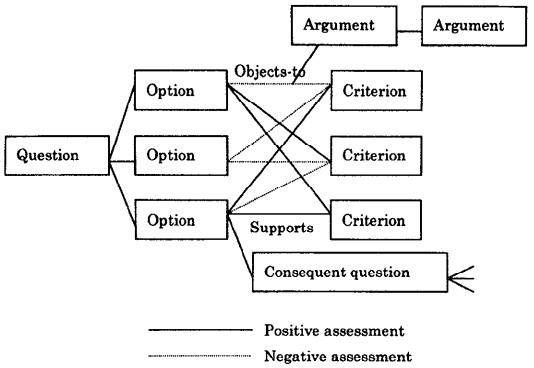

Fig. 1 Argument structure example: QOC[20] vocabulary.

Language (DRL)[19], 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 in parallel with some more traditional representation, or information about the rationale could be integrated with a design notation (such as data-flow diagrams) by adding appropriate links and nodes (e.g., Ref. 28)).

1.2 Knowledge of Application Domain

In a major study of collaborative software development projects, Curtis, et al.[7] 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, communicating, and sharing 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., Ref. 8)). Typical methods support the representation of the problem domain in terms of nodes like *entities*, *objects*, *processes*, or *data*, 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. A prototypical sort of question an analysis model is designed to answer would be *what* the system is actually supposed to do.

1.3 Scenarios of Use

Closely related to application domain knowledge are scenarios of use. In contrast to general domain knowledge, scenarios of use concern the ways in which the system will need to fit into the dynamic flow of activities in its environment. As noted by Guindon[11], 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 and sharing the requirements between designers and end users, and appear to play a role in the sudden unplanned discovery of partial solutions. In a similar vein, Curtis, et al.[7] also concluded from extensive interviews with software developers that scenarios of use were very important for understanding and sharing 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. In other words, scenarios of use are very useful methods as a supporting cross cultural communication between designers and end users. 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[17]. A prototypical sort of question for a representation of a user scenario would be *how* a user would actually perform some task, given a system with some specified functionality.

1.4 Knowledge Generated by Design Methods

Finally, there are many software design methods, with associated notations, rules-of-thumb, principles, and development philosophies. They fall within several broad categories, including structured design (e.g., Ref. 21)) , entity-relation modeling (e.g., Ref. 4)), and object-oriented design (e.g., Refs. 1), 30), 31)). There are many claims by advocates of these techniques, and also some empirical evidence, e.g., from research on software errors[24], that design 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 it 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. The basic units of design notations vary considerably, and so the kinds of questions they could answer vary accordingly. But in general, prototypical questions concern *what* the various software components are and what they do, as well as *how* they generate this external behavior.

1.5 What Is Really Needed?

Unfortunately, despite a proliferation of tool-building in labs around the world, there are currently very few data about the kinds of information software engineers need as they design software. So, in effect, each tool-building effort embodies a set of largely untested assumptions and predictions about what will prove helpful. Testing these assumptions should be given top priority, since they determine the potential of various classes of tools to assist in the design process.

It is very difficult, however, to test these assumptions directly with studies in the field. The expense, risk, and the difficulty of interpreting the results of complex processes in the real world make this option untenable. Tools can fail to be helpful or fail to be used at all for many reasons which have nothing to do with the value of their basic functionality. Laboratory studies, on the other hand, are generally uninformative since the problems that these tools are designed to address are the result of scale. Getting the information one needs is a major problem only in projects which involve a significant number of people working together for a substantial period of time on a problem of realistic complexity. Laboratory studies involving one or a very few people for an hour or two on a relatively simple task with no history or context cannot generally produce comparable needs for information.

This research attempts to inform this issue by using an indirect measurement of the kinds of information that groups of software developers need. 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 should be made available during these collaborative development stages. Asking a question generally indicates that the questioner believes the answer contains knowledge important in the immediate context, and that the questioner does not currently possess this knowledge.

The research we present in this paper is an empirical study of questions in group software design. We have data from software development meetings in two countries. We examined all of the questions in our sample, and developed a classification scheme for them. From our results, we have attempted to determine what kinds of information the questions asked for, and we offer empirically-based suggestions for the kinds of assistance group software developers need.

2. Method

2.1 Data Profile

We use two basic kinds of data in this study. The first is a set of minutes from 38 design meetings held at the Nippon Telegraph and Telephone Corporation (NTT) Software Laboratories that took place over an eight month period. The minutes track meetings of one group of developers in a single project, the task of which was to specify requirements and design for a new version of an existing software development environment. The meetings from which our data are drawn spanned the requirements definition and preliminary design phases of development (see Refs. 12), 13)). 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 among the development team.

This body of data covers a single team over a substantial continuous period of time on a major re-design project. One potential weakness of this data stems from the fact that it is filtered through and reconstructed by each minute taking individual. 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. However 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, and had either four or five participants, and lasted from slightly under one hour to slightly over two hours. (See

Refs. 26), 27) for a detailed description and analysis of these data.) Two of the meetings were teams at Andersen Consulting. One was concerned with specifying a client-server architecture to be used by Andersen to build systems for a variety of customers. The other, involving a different team, was concerned with 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 discussing a knowledge-base editor, trying to determine its basic functionality. Like the NTT data, these projects involved both the requirements specification and design phases. These three particular meetings were chosen from a larger body in an attempt to find meetings as diverse as possible, in terms of personnel and organization, and to span the early software development stages.

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., Refs. 12), 13)) and ISO 9001. 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 (rather than data-structure oriented. Development on the MCC project was in the context of a research-oriented artificial intelligence project, and appeared 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, 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.

2.2 Data Analysis

As we mentioned above, our basic assumption is that the questions software engineers ask in a meeting provide a good heuristic for identifying knowledge that should be made available to collaborative software developers. 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. In general, these distinctions were quite easy to make.

Once we had identified the questions, we categorized them according to the following scheme. 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. Many questions had more than one **target**, in which case each target was included in the data.

Second, we categorized 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. We used the following criteria to determine the attribute: Questions about who built an object or performed a task, or about skills needed, were categorized 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 the way a user would accomplish a task by interacting with the software, would be coded **how**. Questions about deadlines and scheduling were categorized 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 categorized as **why**. If a question referred to two or more attributes of a single target, each was categorized separately and is reflected in our results.

Next, we categorized the target according to the **stage** in the traditional software life cycle in

Table 1 Examples of (paraphrased) questions and how the targets of each question were categorized according to attributes and stages.

Question	Attribute	Stage
What kinds of *user interfaces* will we support?	What	Requirements
How will the user specify a *range*?	How	Requirements
What is the correct behavior for this *module*?	What	Design
Does *context management* access this data structure?	How	Design
Why should I have *two tasks* running at once?	Why	Design

Table 2 Examples of (paraphrased) two-questions and how each relation was categorized.

Question	Relation
Will the *application* have a single *I/O layer*?	Realize
To remove things from the *agenda*, must the *user take them off*?	Realize
Does *I/O* always go through the *message manager*?	Interface
Is the *hint window* the same as the *message area*?	Same

which the target was (or would be) created. We used a simple scheme which included **requirements specification**, **design**, **implementation**, **testing** and **maintenance**. We used the IEEE standards for software requirements and design descriptions[12),13)] and software engineering text books (e.g., Refs. 8), 9)) to develop our criteria for these stages. The thrust of this classification, of course, is that descriptions of what the 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. **Table 1** shows some example questions and how we categorized the targets of the questions.

As mentioned earlier, many questions had more than one target. In these questions, the relation between the targets was central to what the question was asking about. In order to investigate these relations, we categorized them as follows: **Realize** is a relation between a function and the means of carrying it out. For example, the functionality might be "editing a document," and the means for carrying it out might be things like cutting, pasting, and so on. **Same** concerns whether targets are identical in some way. **Interface** is a relation between targets communicating with each other. **Evolve** is the relation between an earlier and later version of a target. Finally, **task assignment** is a relation between persons and tasks they are performing. **Table 2** has examples of some multiple-target questions and how we categorized the relations in them.

Table 3 Correlations between the basic frequencies for the two data sets.

Target Creation Stage Frequencies	0.98
Target Attribute Frequencies	0.97
Relation Frequencies	0.92

Finally, we wanted to see how the knowledge needs of a software design team changed over the early software development activities. 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 the minutes (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 targets and relations changed over these early project stages.

3. Results

3.1 Results and Findings

One of the most interesting and surprising findings is the extraordinary degree of similarity in our results between the two data sets. **Table 3** 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 two data sets. Recall that

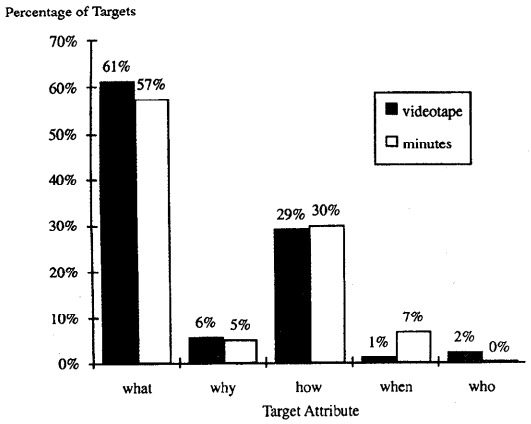

Fig. 2 Percentages of attributes. The difference between the two data sets, though small, is statistically reliable ($X^2 = 27.10$, $p < 0.001$).

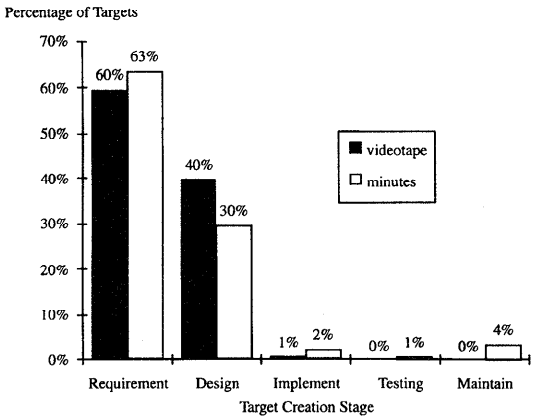

Fig. 3 Percentages of targets created in the software development stages. The small differences across the two data sets are statistically reliable ($X^2 = 40.35$, $p < 0.001$).

the minutes were filtered and reconstructed, while the videotapes were not; the minutes were from a re-design project while the videotapes were all from original design projects; and the data sets are from different countries and cultures. Yet all the correlations are 0.92 or higher. This degree of similarity is really quite astonishing.

Figure 2 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 approximately twice as often about the basic functionality or external behavior of a target as they did about the details of **how** it would be realized. This clearly supports the widely-held belief that understanding *what the software is supposed to do* is the biggest problem in upstream software development.

One of the biggest surprises here is the relatively low frequency of **why** targets. 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** targets. But in fact they are only about 6% of targets in both data sets.

In both data sets, as one would expect, targets created in the **requirements** stage were by far the most frequently asked about, and **design** was a distant second (see **Fig. 3**). The percentage of targets created in other stages is vanishingly small by comparison. It is a little surprising that targets which would be created during the later stages, such as potential future modifications, test suites and testing procedures, implementation issues, and so on, were almost **never** asked about.

Table 4 Cross-tabulation of two most frequent **target creation stages** and three most frequent **target attributes**.

	Requirements		Design	
	Frequency	% all	Frequency	% all
what	404	43%	153	16%
why	33	4%	20	2%
how	118	13%	156	17%
total	555	60%	329	35%

Table 4 shows a cross-tabulation of the three most frequent **attributes** and the two most frequent **creation stages**. By far the most frequent sort of target, accounting for nearly half of all targets, was **requirements-what**. This sort of question asks such things as what is this requirement, what does it mean, what is the software actually supposed to do. Three other types of targets were also relatively high in frequency. **Requirements-how** targets are basically asking about user scenarios, or the particular way in which the user (or other external system) will interact with the software. **Design-what** targets generally concern what software modules are supposed to do, how their functions and interfaces are defined. Finally, **design-how** targets concern the particular way in which functions will be carried out in the software, often asking about things such as algorithms and data structures. There is also a sprinkling of **why** targets from both **requirements** and **design**. The six categories defined

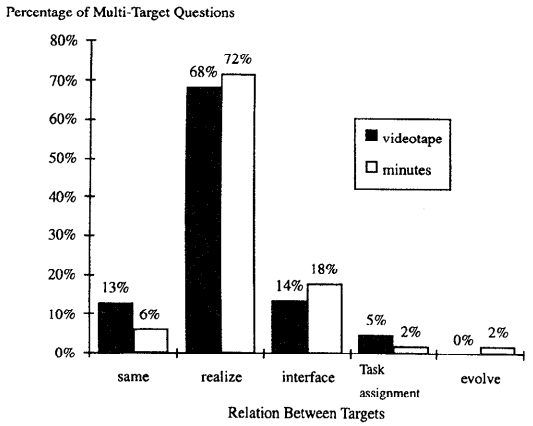

Fig. 4 Distribution of relations in multiple-target questions. The small difference between the data sets is marginally significant ($X^2 = 9.21$, $p < 0.06$)

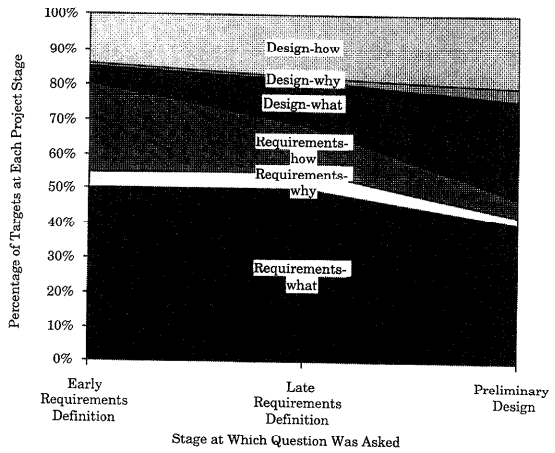

Fig. 5 Changes in the most frequent types of targets over time. The horizontal axis represents the stage at which the question was asked, while the areas represent the six most frequent types of targets. The differences across stages are statistically reliable ($X^2 = 127.4$, $p < 0.0001$).

by this cross-tabulation account for about 95% of all targets in our sample.

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. By far the most frequent relation among targets, as shown in **Fig. 4**, was **realize**, with a significant portion of **interface** and **same**, but very few **task assignment** and **evolve** relations.

As mentioned earlier, we have data from each of three early stages of software development: early requirements definition, late requirements definition, and preliminary design. **Figure 5** shows the changes, over these early development stages, in the six most frequent kinds of targets. Two kinds of targets are consistently highly frequent throughout. The single most frequent kind of target *at each stage* concerns **what** the **requirements** are. So the developers seem primarily concerned with understanding what the software is supposed to do, well into the design phase. The other consistently frequent type of target is **design-how**, asking about the particular way in which the functionality will be realized. The relatively high frequency of these targets during the very early stages is somewhat surprising. After all, these targets occur in relatively detailed questions about the design, concerning things such as algorithms, data structures, and the like. Presumably the fact that they are very frequent even in early requirements definition indicates a concern with assessing the feasibility or difficulty of various requirements.

In contrast with these targets which are consistently frequent, there are two types which change dramatically in frequency over time. One is **requirements-how**, or, roughly, user scenarios. These are very frequent during early requirements definition, but taper off substantially by early design. So it appears that user scenarios provide information which is very important during requirements definition, but less useful in design. The second type of target which changes in frequency over time is **design-what**, which typically occur in questions about such things as functional definitions of modules and interfaces between modules. These targets are rare in requirements definition, but quite frequent by the time preliminary design is reached.

Finally, **why** targets remain consistently *in*frequent. As the project progresses, **requirements-why** targets are traded off, as one would expect, for **design-why** targets. But at each stage, the total of **why** targets is a consistent 5–6%.

The relations between targets remain fairly consistent over time. The **realize** relation is at every point the most frequent by a very large margin.

3.2 Summary of Results

- Questions taken from very different sources of data (minutes versus videotapes, design versus re-design, different countries and cultures) showed an astonishing degree of

similarity in the frequency with which different types of questions were asked.

- Several kinds of questions were consistently frequent throughout the upstream development activities we observed:
  - Questions aimed at understanding what the software is supposed to do.
  - Questions aimed at understanding the detailed design.
  - Questions aimed at understanding the particular way some functionality will be **realized**, or carried out.
- Two kinds of questions changed radically in frequency over time:
  - Questions asking about user scenarios decreased from early requirements to preliminary design.
  - Questions about modules and interfaces increased from early requirements to preliminary design.
- Questions about rationales, issues, arguments, evaluations, and criteria were consistently infrequent.

4. Conclusions: What Is Needed?

We began this paper by asking what kinds of information collaborative software designers need in order to do their jobs in a team. Empirically-based suggestions about the kinds of assistance that would prove most beneficial follow rather directly from our results. The fact that the two very different sources of data produced such extraordinarily similar results greatly strengthens these conclusions. 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, at least within the bounds we discuss below.

Insofar as we can judge from these data, group software developers primarily need assistance with the following:

(1) Understanding the problem domain

We are not the first, of course, to observe that developers generally do not know enough about the problem domain in which they are working (e.g., Ref. 7)), and what the software is actually supposed to do in that domain. This difficulty could be addressed in a number of different way, including such things as domain analysis and modeling (e.g., Refs. 25), 29)), employing techniques for including users in the development process (e.g., Ref. 23)), or perhaps just reading about, taking classes in, or observing people working in the domain (as suggested, e.g., by Ref. 5)).

Our data lead us to two further observations about this problem. First, the need for knowledge about the domain apparently does *not* include a need for information about reasons, rationale, issues, alternatives, evaluations, criteria, and so on which are supported in design rationale tools. At least, we see no indication of this in our data. Second, whatever form the assistance takes, it will very likely be needed not just during requirements definition, but at least through preliminary design, and perhaps beyond.

(2) Exploring detailed design

The many questions about detailed design indicate a need for tools such as rapid prototyping environments, or CASE tools that support appropriate representations such as state diagrams, interaction diagrams, and the like in collaborative software development environments such as CSME (computer-supported meeting environment). Surprisingly, the need for this sort of support is likely to begin at the outset of the project, in early requirements specification, and continue throughout the upstream stages. This, of course, is a major departure from the kinds of recommendations one would derive from a sequential waterfall-style model, according to which these questions would not arise until the detailed design stage.

(3) Tracking *realize* relations

Throughout the upstream activities, we found many questions about the way some functionality would be carried out. The property of being able to assist in answering this sort of question is often called *traceability*, and it is already widely believed to be important. Our findings can perhaps add another increment of credibility to those asserting the importance of this capability.

(4) Capturing and communicating user scenarios

Scenarios of use could be made available to developers in several ways. At least one software engineering method, *Objectory*[14] explicitly incorporates scenarios of use ("use cases") as a central part of the method, and it will be useful for group software designers to provide an user scenarios sharing environment by utilizing asynchronous and synchronous communication support tools. There are also other, less formal techniques, (e.g., Ref. 16)) for making

this kind of knowledge available during design. It seems likely, given the sharp decrease over time in questions about user scenarios, that any tool or technique that is adopted will have its primary benefit in requirements definition, and will be substantially less useful in group design.

(5) Functional definitions and interfaces

Tracking and sharing the functional definitions of modules and interfaces among modules is an area where developers could use help in the design stage. In fact, there are data which strongly suggest that misunderstanding the functions that modules are supposed to carry out and misunderstanding interfaces between modules are the two most frequent sources of software errors[24].

5. Open Questions

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 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 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. Our data do not allow us to distinguish among these possibilities.

These considerations hint at a broader need to assemble the rest of the puzzle of which this work is but one piece. Future work should be aimed at looking at additional settings in which work is done in group software development. We have focused on meetings among developers, but work also gets done in meetings with clients, casual conversations around the water cooler, individuals working by themselves, and so on. We need to fill out the picture by examining what kinds of information are important in these other settings. The information needs may or may not differ substantially from what we found here. We also need to look at the full software development life cycle. Our data come from upstream activities, and the needs for information in the later stages might be different.

Finally, different development methods may give rise to different kinds of information needs. The data we report provide a baseline of information needs using traditional methods, which could be used in order to detect and evaluate changes introduced by new technology. In fact, one of the authors (JDH) is currently examining questions from development meetings using object-oriented methods in order to make such a comparison.

We conclude by noting the importance of empirical studies of software development groups. While we agree that building prototype tools and trying them out is indispensable, unless such efforts are complemented by careful empirical analysis, they run the risk of solving the wrong problem.

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, Atul Prakash, Seishiro Tsuruho, Hironobu Nagano,

and Koichi Matsuda for their valuable comments and suggestions.

References

1) Booch, G.: Object-oriented development, *IEEE Trans. Softw. Eng.*, Vol.12, pp.211–221 (1986).
2) Brooks, F.P.: *The mythical man-month: Essays on software engineering*, Addison-Weslsy, Reading, MA (1975).
3) Brooks, F.P.: No silver bullet, *IEEE Computer*, Vol.20, No.4, pp.10–19 (1987).
4) Chen, P.: The entity-relationship model – Toward a unified view of data, *ACM Trans. Database Syst.*, Vol.1, No.1 (1976).
5) Coad, P. and Yourdan, E.: *Object-oriented analysis*, 2d ed, Prentice Hall, Englewood Cliffs, NJ (1990).
6) Conklin, E.J. and Yakemovic, K.C.B.: A process-oriented approach to design rationale, *Human-Computer Interaction*, Vol.6, pp.357–391 (1991).
7) Curtis, B., Krasner, H. and Iscoe, N.: A field study of the software design process for large systems, *Comm. ACM*, Vol.31, pp.1268–1287 (1988).
8) Davis, A.M.: *Software Requirements: Analysis and Specification*, Prentice Hall, Englewood Cliffs (1990).
9) Ghezzi, C., Jazayeri, M. and Mandrioli, D.: *Fundamentals of software engineering*, Prentice Hall, Englewood Cliffs, NJ (1991).
10) Guindon, R.: Designing the Design Process: Exploiting Opportunistic Thoughts, *Human-Computer Interaction*, Vol.5, Nos.2&3, pp. 305–344 (1990).
11) Guindon, R.: Knowledge exploited by experts during software system design, *Int. J. Man-Mach. Stud.*, Vol.33, pp.279–304 (1990).
12) IEEE: Guide for software requirements specifications, Std 830-1984 (1984).
13) IEEE: Recommended practice for software design descriptions, Std 1016-1987 (1987).
14) Jacobson, I., et al.: *Object-oriented software engineering: A use case driven approach*, Addison-Wesley, Reading, MA (1992).
15) Jeffries, R., Turner, A.A. and Polson, P.G.: The processes involved in designing software, *Cognitive skills and their acquisition*, Anderson, J.R. (Ed.), pp.255–283, Erlbaum, Hillsdale, NJ (1981).
16) Karat, J. and Bennett, J.L.: Using scenarios in design meetings – A case study example, *Taking software design seriously*, Karat, J. (Ed.), pp.63–94, Harcourt Brace Jovanovich, Boston (1991).
17) Karat, J. and Bennett, J.L.: Working within the design process: Supporting effective and efficient design, *Designing interaction: Psychology at the human-computer interface*, Carroll, J.M. (Ed.), pp.269–285, Cambridge University Press, New York (1991).
18) Lee, J.: SIBYL: A tool for managing group design, *CSCW '90*, Los Angeles (1990).
19) Lee, J. and Lai, K.-Y.: What's in design rationale, *Human-Computer Interaction*, Vol.6, pp.251–280 (1991).
20) MacLean, A., et al.: Questions, options, and criteria: Elements of design space analysis, *Human-Computer Interaction*, Vol.6, pp.201–250 (1991).
21) Marca, D. and McGowan, C.: *Structured analysis and design technique*, McGraw-Hill, New York (1988).
22) Moran, T. and Carroll, J. (Eds.): *Design Rationale: Concepts, Techniques, and Use*, Lawrence Erlbaum Associates (1996).
23) Muller, M.J.: Retrospective on a ycar of participatory design using the PICTIVE technique, *CHI'92*, ACM press, Monterey, CA (1992).
24) Nakajo, T. and Kume, H.: A case history analysis of software error cause-effect relationships, *IEEE Trans. Softw. Eng.*, Vol.17, pp.830–837 (1991).
25) Nerson, J.-M.: Applying object-oriented analysis and dcsign, *Comm. ACM*, Vol.35, pp.63–74 (1992).
26) Olson, G.M., et al.: Small group design meetings: An analysis of collaboration, *Human-Computer Interaction* (1992).
27) Olson, G.M., et al.: The structure of activity during design meetings, *Design Rationale*, Moran, T. and Carroll, J. (Eds.) Lawrence Erlbaum Associates (1996).
28) Potts, C.: Supporting software design: Integrating design processes, design methods, and design rationale, *Design Rationale*, Moran, T. and Carroll, J. (Eds.) Lawrence Erlbaum Associates (1996).
29) Rubin, K.S. and Goldberg, A.: Object Behavior Analysis, *Comm. ACM*, Vol.35, No.9, pp.48–62 (1992).
30) Rumbaugh, J., et al.: *Object-oriented modeling and design*, Prentice Hall, Englewood Cliffs, N.J. (1991).
31) Wirfs-Brock, R., Wilkerson, B. and Wiener, L.: *Designing object-oriented software*, Prentice Hall, Englewood Cliffs, NJ (1990).

(Received March 2, 1998)
(Accepted June 5, 1998)

James D. Herbsleb is a member of the Software Production Research Department of Bell Laboratories, Lucent Technologies. He has a Masters Degree in computer science from the University of Michigan and a Ph.D. in psychology from the University of Nebraska. Prior to joining Bell Labs, he conducted empirical studies of software engineering as a post-doctoral fellow at the University of Michigan, and at the Software Engineering Institute at Carnegie Mellon University. His published work includes studies of object oriented technology, information needs in software engineering, results of software process improvement efforts, software metrics, and the use of collaborative technology in software development.

Eiji Kuwana received his B.S., M.S. in computer science from the University of Electro-Communications, in 1982, 1984, respectively. He joined NTT Laboratories in 1984. From 1991–1992, he worked for the University of Michigan as a visiting research scientist. He is currently a senior manager of NTT Multimedia Service Promotion Headquarters. His major research interests include multimedia services, human-computer interaction, computer network, and CSCW. He served as a program committee member of ACM DIS'95, CSCW'96, IEEE CoopIS'95, ICPAD'96, APCHI'96, '97, '98. He is a member of ACM and IPSJ.