Coordination and collective mind in software requirements development

by K. Crowston
E. E. Kammerer

The purpose of this study was to understand how the group processes of teams of software requirements analysts led to problems and to suggest possible solutions. Requirements definition is important to establish the framework for a development project. Researchers have proposed numerous requirements development techniques, but less has been done on managing teams of requirements analysts. To learn more about group processes within such teams, we studied two teams of analysts developing requirements for large, complex real-time systems. These teams had problems ensuring that requirements documents were complete, consistent, and correct; fixing those problems required additional time and effort. To identify sources of problems, we applied two theories of collective action, coordination theory and collective mind theory. Coordination theory suggests that a key problem in requirement analysis is identifying and managing dependencies between requirements and among tasks. Most requirements methods and tools reflect this perspective, focusing on better representation and communication of requirements. The collective mind perspective complements these suggestions by explaining how individuals come to understand how their work contributes to the work of the group. This perspective suggests that deficiencies in actors' representations of the process and subordination to collective goals limit the value of their contributions.

One of the hardest parts of system development is deciding what the system should do, that is, in determining the system requirements. In his classic essay "No Silver Bullet," Frederick Brooks[1] noted that:

> The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines and to other software systems. No part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

A Standish Group report[2] identified clear requirements as the third most important factor for successful project development; incomplete and changing requirements as the second and third most important factors leading to unsuccessful projects; and incomplete requirements as the number-one factor for canceled projects. Meyers[3] suggested that more than half the cost of developing systems could be attributed to decisions made during the development of their requirements. Once one knows exactly what the system should do and how it should behave, implementation is often simple by comparison.

For small projects, requirements analysis and development are relatively unproblematic: an individual analyst working with users can specify system requirements reasonably completely and consistently. This person may also implement the system, so even if there are problems with the specification, he or she

©**Copyright** 1998 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

 0018-8670/98/$5.00 © 1998 IBM

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

can simply fix them as they arise with little danger of damaging the integrity of the system.

Large systems, however, pose greater challenges. Researchers have identified numerous problems that arise for projects large enough to require a team of requirements analysts who are unlikely to be the users or implementers of the system. These problems include ensuring that the requirements:

- Fit customers' needs, even though it is difficult (and sometimes impossible) for customers to understand the requirements specifications as written by the requirements analysts
- Specify a system that can actually be built by the developers
- Are complete and consistent

These problems arise primarily because large systems require knowledge from more domains[4] and involve many more requirements than can be managed by a single person, no matter how talented. Furthermore, it is nearly impossible to create part of the specification without interaction with other parts. Designers attempt to decompose systems into pieces that are not tightly coupled, but it is difficult to create pieces both small enough for a single individual to work on and having only limited interactions with the rest of the system.

Therefore, large projects will always require coordinated group effort. Requirements analysts must be able to share their knowledge of the problem and individual parts of the system with customers, developers, and other analysts to generate a complete, correct, and consistent set of requirements. Numerous techniques have been proposed to guide and structure the development process. However, less attention has been given to the processes within groups of requirements developers.[5] In this study, we show how two theories could be used to identify the causes of some problems that arise in requirements analysis and development and to suggest possible strategies to eliminate or minimize them.

In our study we started with coordination theory, which suggested ways to manage dependencies in processes.[6] Coordination theory provided some strategies for managing requirements analysis, but our research also probed the limitations of coordination theory. We therefore turned for additional insight to collective mind theory.[7]

Research setting

Before discussing the theories, however, we introduce our research setting in more detail to explain why these two theories seemed appropriate. We studied requirements development in two companies, which we will refer to by the pseudonyms LGC and TC. LGC was a large government information systems contractor that developed requirements for air traffic control systems; TC was a large multinational telecommunications company at which we studied a division that developed control software for telecommunications switches.

We chose these two sites because the requirements problems they faced were particularly difficult.

- Both companies were actively involved in requirements definition for very large complex systems requiring large development teams. The telephone switching software at TC had about five million lines of code, and 200 people were involved in requirements analysis. LGC had 300–400 employees in the division we studied, and the government agency for which the requirements were developed employed many more.
- Both developed real-time software, which is particularly complex to specify because of the need for strict timing and the unpredictable effects of interactions among elements.
- The systems included both software and nonsoftware components such as humans, hardware, etc. In Davis's terms,[8] they were systems rather than pure software.
- The systems comprise some new and many already developed components, which constrained the development process. Functional needs had to be squared with requirements imposed by the existing technology and architecture.
- The systems had to be tailored to provide the desired functionality in various customers' environments and to work with customers' existing equipment and processes.

About LGC. The group at LGC was involved in a single project: integrating several existing and new prototypes of future air traffic control systems and an air traffic simulation into a single interactive simulation. The overall result of the project was to be a set of requirements for the real systems. In other words, the prototypes were a way to test and refine requirements rather than an end in themselves. However, the prototypes did have an immediate customer: experimenters who were to test various system

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

configurations and capabilities to ensure that they worked together and were useful. The final requirements document was then to be written based on experience with the prototypes and given to another contractor as a basis for systems development.

The group inherited much of the prototype code from other groups and developed the remaining prototype code themselves. Developing the requirements for a particular prototype involved determining which overall system requirements engaged that component, specifying which services the component would provide to satisfy the requirements, and determining the other components with which the prototype interacted and exchanged data. For existing prototypes, the newly specified services would then be mapped to the already written code. As a particular prototype was specified, changes might be made to the system-level requirements or to other components.

One problem this group had was that the requirements documents inherited and developed needed extensive editing for consistency when they were integrated. For example, different components made different assumptions about when shared data would be provided. The individuals developing parts of the specification were not able to anticipate all the opportunities for inconsistency. Yet, it was not feasible for a single person to write the entire specification.

About TC. The second company, TC, developed telephone switching systems, which are essentially large special-purpose computers. The division we studied developed high-capacity switches for an international market. The telephone switch hardware and software were extensively customized to meet each customer's unique needs. The content of a particular switch was expressed as a list of the features it included, such as interfaces to the rest of the network (e.g., a particular communications line and signaling protocol), customer services (e.g., call waiting) and operational support (e.g., customized reports or diagnostics). Features were described in detail in separate specification documents and implemented in hardware and software.

Unlike at LGC, the requirements process at TC was executed for each customer. Every customer was different, and the group had several projects at various stages of work at any time. Work was largely reactive: typically, customers submitted requests for contract proposals, usually through the sales force, and the company would respond with a bid. The "front-end" group that we studied used information in the request to determine which features were needed in the delivered product to satisfy the stated needs. A typical product included many previously developed features that were simply retested, some extensions to existing features, and a handful of entirely new features (e.g., 10 new features out of 200 total). For each new feature, a specification document was written to guide the development of new hardware and software.

Large systems require knowledge from more domains and involve many more requirements than can be managed by a single person.

Generation of feature lists was hampered by the complexity of the products. There were literally thousands of possible features, some required, some optional, and many mutually exclusive. The primary difficulty experienced by the front-end group was feature churn. Feature churn happened when the list of required features turned out to be incorrect in some way and had to be changed, e.g., by adding missing features or replacing one feature with another. Most significant were missed new features, since these required additional programming work that had to be fit into the development schedule. One study done by the company identified 170 changes to the feature lists for six projects in a six-week period. The same study showed that changes could cost up to 500 times as much when made late in a project as in the beginning, and late changes were unfortunately not uncommon—some coming as late as customer delivery.

Table 1 summarizes the comparison of the two research sites. In many ways our sites were atypical. The projects were larger than most, the constraints of the underlying technology were more important, and user involvement was difficult to obtain, as discussed below, limiting the use of user-driven development methodologies such as joint application development technique (JADT). Furthermore, our study is based on only two sites, further suggesting that gen-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 1 Comparison of key features of research sites

Dimension	TC	LGC
Project	Telephone switching software	Integrating simulations of and human-interface prototypes for air traffic control systems
Size of project	5 million lines of code	
Size of group	200 people involved in requirements analysis	300–400 employees in division
Requirements analysis problem	Determining existing and new features to meet a customer request	Determining which existing or new module would meet system requirement and how data would be exchanged
Specific problems with process	"Feature churn"	Inconsistent assumptions across modules

eralizations should be made with caution. Nevertheless, we feel that the enormous complexity of these systems makes these sites an extremely rich source of data about the requirements analysis and development process. Both companies were interested in participating in this research because they felt that requirements development was a fundamental problem that they had to learn to manage more effectively.

Coordination theory

From our initial examination of the problem, it was apparent that a major problem in developing requirements was coordination. Curtis, Krasner, and Iscoe[9] identified this problem in a field study done to identify problems affecting software productivity and quality in 17 large-systems development projects (some successful and some not). The three problems they considered most important because of the additional effort or mistakes attributable to them were "the thin spread of application knowledge," "fluctuating and conflicting requirements," and "communication and coordination breakdowns." They concluded that large projects have extensive communication and coordination needs that are not mitigated by documentation. They also found that breakdowns were likely to occur at organizational boundaries, but that coordination across these boundaries was often extremely important to the success of the project. These results suggested that it would be valuable to study the kinds of coordination problems that arise in software requirements development and the mechanisms available to address these problems.

About coordination theory. Coordination theory provides a theoretical framework for analyzing complex processes such as requirements analysis. We used the model presented by Malone and Crowston,[6] who define coordination as "managing dependencies." They analyzed group action in terms of *actors* performing *interdependent tasks*. These tasks might also require or create *resources* of various types. For example, in the case of software requirements development, actors included the customers and various employees of the software company. Tasks included translating aspects of a customer's problem into system requirements and checking requirements for consistency against other requirements. Finally, resources included the information about the customer's problem, existing system functionality, and analysts' time and effort.

In this view, actors in organizations face *coordination problems* arising from dependencies that constrain how tasks can be performed. Dependencies can be between tasks, between tasks and the resources they need, or between the resources used. Dependencies may be inherent in the structure of the problem (e.g., components of a system may interact with one another, constraining how a particular component is designed) or they may result from the assignment of tasks to actors and resources (e.g., two engineers working on interacting components face constraints on the designs they can propose without interfering with each other).

To overcome these coordination problems, actors must perform additional work, which Malone and Crowston called *coordination mechanisms*. For example, if particular expertise is necessary to perform a particular task (a task-actor dependency), then an actor with that expertise must be identified and the task assigned to him or her. Important for this process, dependencies between requirements (resource-resource dependencies) must be actively identified and their implications for the design assessed and managed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Coordination theory suggests that, given an organization performing some task, one way to generate alternative processes is to first identify the particular dependencies and coordination problems faced by that organization and then consider what alternative coordination mechanisms could be used to manage them. Problems with the process may also be caused by dependencies that are not managed.

Data collection and analysis. In each site, we wanted to document the kinds of dependencies that constrain the requirements development process and document the coordination mechanisms that were used to manage these dependencies. Dependencies are conceptualized as arising between tasks and resources, so we started by ascertaining the tasks and resources used in requirements analysis, then identifying dependencies and related coordination mechanisms. Since the tasks in requirements analysis are entirely information-based, we adopted the information processing view of organizations.[10-12]

Given this perspective, the goal of the data collection was, in the terms of March and Simon,[10] to uncover the programs used by the individual requirements analysts in the groups. March and Simon suggest three methods for data collection to uncover these programs: (1) interviewing individuals, (2) examining documents that describe standard operating procedures, and (3) observing individuals at work. Although all three techniques were used, we relied most heavily on semi-structured interviews. As March and Simon point out, "most programs are stored in the minds of the employees who carry them out, or in the minds of their superiors, subordinates, or associates. For many purposes, the simplest and most accurate way to discover what a person does is to ask him."[13]

We started the data collection by identifying different kinds of actors in the groups. This identification was done with the aid of a few key informants and refined as the study progressed. Formal documentation of the process was used as a starting point when available. However, it was expected that the process performed would differ from the formally documented process. It was the informal process (as well as the formal process surrounding it) that we sought to document.

Interview subjects were identified by the key informants, based on their job responsibilities. Interviews were generally one to two hours long. When possible, both authors took part, one leading the interview while the other took notes. Because of concerns about the confidentiality of the products being developed, few interviews were tape-recorded. Most interviews were held at the companies' engineering headquarters, with some follow-up interviews conducted by telephone.

At LGC we interviewed 14 individuals, including managers and systems developers. At TC, we held 19 interviews with 22 individuals, including group managers, product managers, project managers, and application engineers. We also sat in on several work meetings and collected examples of documents created and exchanged during the requirements development process.

Since tasks in requirements analysis are information-based, we adopted the information processing view of organizations.

The initial focus of individual interviews was to identify the type of information received by each kind of actor and the way each type was handled. For example, we asked subjects: (1) what kinds of information they receive; (2) from whom they receive it; (3) how they receive it (e.g., from telephone calls, memos, or computer systems); (4) how they process the different kinds of information; and (5) to whom they send messages as a result. When possible, these questions were grounded by asking interviewees to talk about items they had received that day, an "in-basket methodology."[14] Meetings were also held with three different groups at TC (numbering 5, 7, and 15 members, respectively) during which they identified responsibilities of their group and the various roles they performed or with which they interacted.

In addition to interviews, we collected data from participant observation. One of the authors participated in the development process at LGC for a period of four months, which included the latter part of the requirements analysis process and initial attempts to use the requirements for further development. Additionally, a former member of the software devel-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 2 Comparison of experiences with coordination mechanisms in research sites, organized by type of dependency

Dependency	Applicable Coordination Mechanisms	TC	LGC
Producer-consumer (usability)—Requirements had to solve customer's problem	Standardization	Requirements are stated in a standard form to ensure usability by developer Specifications of certain components meet standards, but system is customized	System is being developed for the first time, so no standards exist
	Ask customers	Primary source of information, but information was often delayed or incomplete due to bidding process	Lack of communication, even though immediate users were local
	Developer expertise	Requirements developers often knew customer requirements better than customer	Requirements developers understood air-traffic control, but not needs of experimenters
Task-subtask dependencies—Process of developing requirements decomposed into developing individual requirements	Planning subtasks to accomplish task	Analysts read specification to determine which features customer needed Could use database to determine features, but reportedly out-of-date and hard to search Not infrequently, missed or misspecified a feature	Analysts wrote specifications for existing modules
Task-actor dependencies—Given a requirement, had to find person with expertise to work on it	Determine expertise needed	Discuss in group meetings Most analysts seemed to know whom to ask	Discuss in group meetings Most analysts seemed to know whom to ask
Dependencies between requirements—Requirements might interact	Modules interact only via well-defined interfaces Manual review of features	Database existed, but reportedly unreliable Most interactions caught based on analyst's knowledge	Approach missed data dependencies

opment group at TC assisted in the data collection and analysis.

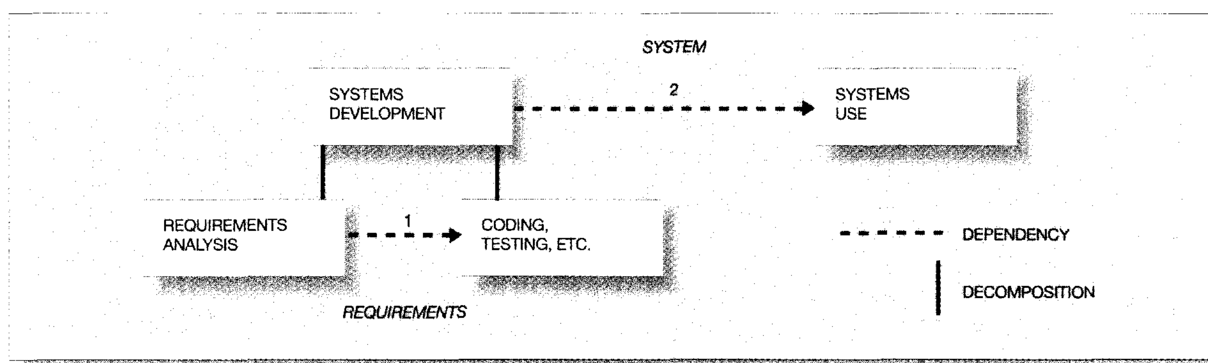
Notes from the interviews and participant observation were analyzed by the authors to identify evidence supporting or contradicting the theories used. Although the process of moving from theory to data and back was somewhat cyclical, in that the data were used to critique the theoretical models, the analysis was primarily deductive because the purpose was to evaluate the applicability of the theories to the requirements analysis process.

Coordination in requirements development. In this subsection, we present the dependencies and coordination mechanisms identified in our sites. This analysis is summarized in Table 2. Malone and Crowston listed several types of dependencies.[6] Our research built on and contributed to this work by identifying additional coordination mechanisms that can be used to manage the dependencies we identified. We discuss four kinds of dependencies in the context of software requirements analysis: producer-consumer dependencies, task-subtask dependencies, task-actor dependencies, and dependencies between requirements.

Producer-consumer dependencies. Dependency analysis can be done at different levels of abstraction. At a high level, a producer-consumer dependency existed among requirements analysis and tasks further downstream, such as coding and testing, as shown by the dashed lines in Figure 1. (The process representation used in Figure 1 is modeled after that used in the process handbook.[15] The solid vertical lines indicate decomposition of a task into subtasks.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 1 **Dependencies between high-level tasks in software development**



Dashed horizontal lines indicate flow of a resource from one task to another and thus a producer-consumer dependency.) In other words, requirements analysis produced an output—the requirements (shown in italics in Figure 1)—that was used by coders and testers. The flow of this resource indicates a producer-consumer dependency between the two tasks (dashed line number 1). At a higher level of abstraction, the requirements analysts were part of the process of creating a system for the customer. In other words, both requirements analysis and coding are subtasks of the systems development process (indicated by solid lines in Figure 1), and systems development creates an output—the system—that is used by some other task, creating a higher-level producer-consumer dependency (indicated by dashed line number 2). Further distinctions could be made similarly if desired, e.g., between the needs of the customers who commission and pay for the system and the eventual end users of the system.

Malone and Crowston noted that producer-consumer dependencies impose several constraints on the producer,[6] in particular *usability*—ensuring that the output is of a form useful to the consumer, and *transfer*—ensuring that the output is available to the consumer when needed. In this case, usability seemed to pose the key problem: requirements analysis is essentially a coordination mechanism for ensuring that the system created is useful, that is, a way to manage flow dependency number 2. Malone and Crowston suggested alternative approaches[6] to satisfying usability dependencies including standardization (i.e., producing the output in an expected form), asking the consumer for input, or involving the consumer in a participatory design process. Each mechanism will be considered in turn.

Standardization. If a standard product is satisfactory for the user, requirements analysis is unnecessary after the product has been designed once. This approach is used for most packaged software, for example. For LGC, however, it was the initial system being developed; for TC, customers required highly customized products. In both cases, therefore, standardization was not an appropriate way to manage the usability dependency between systems development and use (dependency number 2 in Figure 1). Therefore, requirements analysts had to determine from the users of the system what kind of system would satisfy their needs.

Nonetheless, requirements were provided to developers in a standard format, e.g., using agreed-upon representations for system functionality. In other words, standardization was used to manage dependency number 1 in Figure 1. In the companies we studied, it seemed to be assumed that requirements analysts understood the capabilities of the technology and the development group because of training or experience with the development process. Therefore, communication among analysts and developers about the needs of the developers was indirect.

User involvement. A second strategy for managing usability constraints is to ask consumers to state their requirements or to somehow involve the consumer in the design process. Indeed, many who have studied this problem in recent years have emphasized the overwhelming importance of user involvement. In both companies, requirements analysts did solicit user input, e.g., by asking questions (sometimes through an intermediary or by arranging face-to-face meetings) or, in the case of LGC, by having air traffic controllers work with the prototypes. Other input was

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

also used; for example, in TC, if a system was required to interface with unfamiliar equipment, a planner might attempt to borrow samples of that equipment to check the implementation.

Unfortunately, getting input from the end users was quite difficult in both companies, and in neither case was user input sufficient to solve the requirements definition problem. At TC, users and developers worked for different organizations and were typically separated geographically and linguistically, since the division specialized in international sales. The bidding process also imposed further barriers, since information from the customer had to be provided to all bidders equally, and interactions were typically funneled through the sales staff as a check on what the company promised. As a result, detailed technical information might become available only after a contract was signed and requirements definition was partially completed. Finally, because of the large number of developers involved in creating requirements in both companies, it would have been impossible for all of them to meet a user anyway (nor guarantee that they would all interpret the users in the same way). Although these problems were more extreme in our two sites, Davidson[16] notes comparable problems applying joint application development technique (JADT) in three financial service companies.

More importantly, user input alone was insufficient to ensure that the requirements were correct. For air traffic control and telephone switching, making a useful system required the application of specialized domain knowledge[4] as well as information about the particular user context. For example, at TC, customer representatives did not always appreciate the detail required (e.g., which of 24 kinds of call waiting service was wanted, which standard functionality was required, or the precise specification of the trunk interfaces). At LGC, the ultimate users of the system were air traffic controllers who are easily able to provide input on how the system should interact with them, but who did not have any information about how the many systems that support their tasks might interact.

In both companies, this kind of specific domain knowledge was a central part of the developers' competency and, in a sense, was the product they offered. Many members of the requirements development groups were experts in their fields, in some ways more expert than the customers. Developers used this expertise to make sense of the incomplete and ambiguous input they received from customers and translated it to their own domains. In other words, in this site a new coordination mechanism was used to manage usability, namely knowing as much or more about the problem domain than the eventual user.

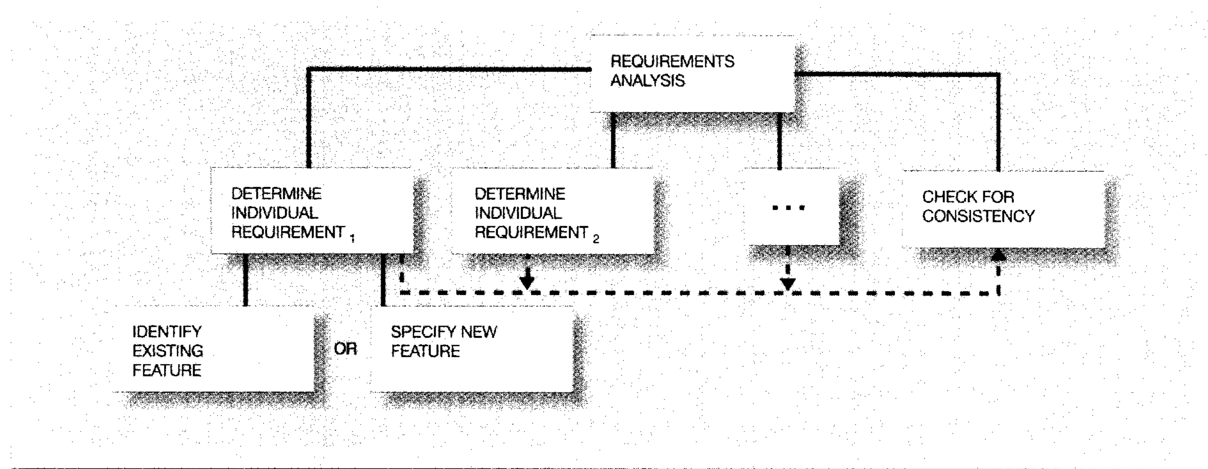
However, it should be noted that over-reliance on this mechanism can easily lead to failure to listen to the customer and development of the wrong product. Tellingly, failures of this type of expertise were at the root of a problem encountered by the development group at LGC. The prototypes developed failed to fully meet the needs of the immediate users, the experimenters, because the developers did not understand what experimenters needed from prototypes (as opposed to what the eventual users of the system needed from the system itself). In previous projects, requirements were derived from study and experience; this project was one of the first to use prototypes.

Task-subtask dependencies. The previous subsection considered dependencies with requirements analysis taken as a whole. Of course, the process comprised many detailed tasks; a possible partial decomposition is shown in Figure 2, using the same notation as in Figure 1, where vertical solid lines indicate decomposition of a task into subtasks, and horizontal dashed lines indicate a flow of resources and therefore a producer-consumer dependency. The determination of individual requirements corresponds roughly to what Davis[8] called the problem analysis stage, the goal of which is a "relatively complete understanding of the problem." It is followed by the problem description stage, during which analysts "resolve conflicting views and eliminate inconsistencies and ambiguities"[17] as they write a final requirements document.

At TC, analysts created requirements by adding features to the feature list. If an appropriate feature could not be found, a detailed specification of a new feature would have to be written. Alternatively, the analyst might argue that the feature might not be worth implementing. At LGC, the process was similar, except that analysts created specifications for an individual prototype by locating a piece of code that already did what was required or by specifying new functionality. Again, the high-level requirements might be revised instead. Finally, the requirements created were checked for consistency. (Of course, individual requirements could also be checked for consistency as they were developed.) As a result, there are producer-consumer dependencies between

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 2 Decomposition of requirements analysis into specification of individual requirements and checking for consistency



the creation of each requirement and the final check, as indicated by the dashed lines in Figure 2.

Between the high-level task of performing requirements analysis and the lower-level task of writing individual requirements were task-subtask dependencies. These dependencies were managed in this case by selecting subtasks that accomplish the desired task. In other words, requirements analysis is in essence a planning task, in which the high-level task is clear (write the requirements) and even the range of possible lower-level tasks is largely known (e.g., at TC, most features already exist), but an appropriate set of lower-level tasks must be chosen to achieve the high-level task. We were particularly interested in analyzing planning mechanisms, since they had not been discussed in detail by Malone and Crowston.[6]

Davis claimed that the difficulty in problem analysis is "organizing all the information, relating different people's perspectives, surfacing and resolving conflicts,"[18] i.e., developing a consistent set of requirements. However, according to our interviews, a large part of the problem was simply determining what the often incomplete and ambiguous customer statements meant for the requirements. (As discussed above, it was not always possible to simply ask users for clarification, nor would this always have been sufficient to resolve the issue.)

At TC, analysts read customers' proposal requests and determined which features were required; at LGC, analysts read the overall system requirements and determined which were related to their particular prototype. Both required an understanding of the goals of the system and knowledge about the capabilities of the technology. An analyst who did not know which features already existed, for example, would have been hard-pressed to understand what a customer was asking for, much less how to accomplish it.

At TC, analysts had access to a database of features, although it was difficult to search and was reported to be out-of-date. They could also ask engineers for advice in picking features. In some cases, however, they would simply miss a requirement or pick the incorrect feature, misunderstanding the customer's needs. Fixing these errors caused feature churn. In most cases, however, analysts seemed able to make sense of the task and decompose it to specifics. Unfortunately for our goal of characterizing the steps in coordinating these requirements, analysts were not able to easily articulate how they did this decomposition.

Task-actor dependencies. The next class of dependencies we consider are task-actor dependencies. Tasks require resources, if only the effort of an actor to perform the task. An important class of coordination processes manages the assignment of re-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sources to tasks. Crowston[19] suggested four steps to such resource assignments: identifying the type of resources needed, identifying available resources, selecting the resources, and making the assignment. Most work on resource assignment (e.g., in economics, organization theory, or computer science) has focused on the middle steps, that is, techniques for identifying or selecting available resources. By contrast, in the requirements analysis process, the first step—identifying what resources are needed to perform a particular task—seems at least as important.

Analysts could easily refer problems to one another and for the most part seemed to know who had expertise in which particular areas. Before they could refer a problem, however, they had to first determine what expertise was required to solve the problem, or even what the problem was. One approach is to consult everyone about everything that might be relevant, whether one-to-one, in small groups reviewing a section of the requirements, or in large status meetings. All three techniques seem to be used at both LGC and TC, although analysts for the most part seemed to know to whom to talk.

Dependencies between requirements. The final class of dependencies are those between requirements. Interactions among requirements are often a problem in large real-time systems. In a telephone switch, for example, call waiting and call forwarding on busy specify different actions for an incoming call when the person being called is already on the phone. If both features are active, the requirements must give one or the other priority, or the system behavior will be undefined. This interaction is fairly obvious; much more subtle interactions are possible.

To manage these dependencies, they first had to be identified, and harmful dependencies had to be eliminated by changing one or the other of the requirements. For the processes we studied, identifying dependencies seemed to be a key factor. Analysts had to determine how the features they specified interacted with every other feature in the system to ensure that they were complete and consistent with one another and with prior decisions about the system. Because there were hundreds of features, each supported by different people, these interactions were difficult to detect. Modifying the software for one feature could affect other features in ways that were unanticipated by the analyst specifying the modification. Those in charge of the affected feature might not become aware of the changes until their feature "broke."

LGC attempted to control these interactions by having prototypes interact only through message passing. Where dependencies were necessary, analysts would negotiate an acceptable interface from one prototype to another. As it turned out, however, the prototypes (some of which had been created separately and for different purposes) still interacted in unexpected ways, often because they made different assumptions about such things as the situation being simulated or the order in which they would obtain data. For example, there might be a piece of information about the environment that one simulation needed to know when it started, but which the central simulation manager did not (at least originally) provide until later. Progress reviews focused on the functionality and status of each module, but as the project manager reported, "Those were the wrong questions." To catch these interactions, she suggested instead a focus on the flow of control between modules (e.g., as a simulated flight is passed from one system to another). Fixing these dependencies required extra work.

In TC, ensuring that requirements (i.e., feature lists) were consistent required considerable additional effort. Specified features might conflict in nonobvious ways, and these dependencies required additional work to detect and correct. (Detection of feature interactions in telecommunications systems has become an active research area, although most researchers are focusing on detecting or preventing interactions when features are executed rather than during design.) Various analysts reviewed the feature list at several different points in the process to uncover these dependencies, apparently based on their knowledge of the characteristics of the features. Because of time pressures, however, these checks were often cursory, since in practice most features did not interact. In addition, the feature database reportedly did not represent dependencies reliably. Both factors led to feature churn caused by missed dependencies (or in the worst case, a system that behaved incorrectly).

Tools and techniques for coordinating requirements development. Coordination theory suggests problems arise when dependencies go unmanaged. Implicitly, most published approaches to requirements analysis are consistent with a coordination theory approach, providing ways to communicate requirement information and to manage dependencies or even to reduce and eliminate them. The dependencies discussed above are: usability dependencies between requirements analysts and between users and re-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

quirements analysts and developers; task-subtask dependencies; task-actors dependencies; and dependencies between requirements. Of these, usability dependencies and dependencies between requirements seem to be addressed best.

Strategies for managing dependencies between analysts and requirements users include more efficient ways to communicate information from users to requirements analysts and from analysts to developers. Techniques for eliciting requirements have been surveyed by Powers et al.,[20] Davis,[21] and Birrell and Ould,[22] among others. Davis discussed numerous approaches to defining requirements, including listing all inputs and outputs, listing major functions, structured requirements definition (SRD), structured analysis and design technique (SADT), structured analysis and system specification (SASS), modern structured analysis, PSL/PSA, and object-oriented problem analysis. Davis also provided several extended examples of applying these techniques.[23] Many of these techniques also structure the process of decomposing a system into subunits, addressing task-subtask dependencies.

Most modeling methods allow the system to be described in several ways to provide a sufficiently comprehensive view of the system.

Although these techniques are certainly useful, they do not explicitly address how an understanding of the system is developed to address the higher-level usability dependency between systems development and system use. Instead, they provide a structured way to present and refine such an understanding. By contrast, Joint Application Development Technique (JADT), developed by IBM, provides a method for analysts to obtain information from and negotiate with clients in intensive workshop sessions. Unfortunately, JADT would have been difficult to apply in our sites because of restricted access to knowledgeable users, as discussed above.

Strategies for managing dependencies between requirements include techniques to reveal system interdependencies, such as formal representations or prototyping. Techniques for modeling systems are used to represent information gathered in a way that reveals potential problems. The methods are typically graphical, although physical and simulation models are occasionally used. Davis listed four notations for problem analysis[8]—data-flow diagrams, data dictionaries, entity-relation diagrams, and load object diagrams—and Birrell and Ould added structured English, Petri nets, and finite-state machines.[22]

Most modeling methods allow the system to be described in several ways to provide a sufficiently comprehensive view of the system. For example, a model might include interfaces to external entities, functions to be performed, data transformations, structure of input/output data, relationships among information, and system behavior.[24] An example of preventing dependencies is the decomposition of the system into decoupled components with well-defined interfaces or using design methodologies that only allow certain kinds of interactions, both with the goal of allowing dependencies only in well-known channels. As mentioned above, such techniques were tried in our sites, albeit with limited success.

Finally, none of the techniques seems to directly address the problem of finding an appropriate requirements engineer to work on a particular problem. Consideration of this issue led us to expand our study, as we discuss in the rest of this paper.

Limitations of coordination theory. After our initial analysis of data from our two sites, it seemed that coordination theory did illuminate some of the problems of requirements analysis on large projects, but it provided only one approach to the problems of requirements development. Better ways for analysts to coordinate were certainly important, but it seemed equally necessary for group members to develop shared understandings of customers' needs and to anticipate what actions would contribute to the process. In other words, the requirements analysis process seemed as much a matter of development of shared understandings and collective sensemaking[25] as of communicating and coordinating. The key to the successful coordination of the requirements development seemed to be that analysts mostly "just knew" which features were needed, whom they had to consult for advice on which features to pick, whom to ask to write a specification or check a dependency, etc. The question then became, "How did they know that?"—a question that coordination theory was not designed to answer.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Collective mind theory

As we were trying to understand how the analysts learned what they need to know to determine requirements and manage their dependencies, we were introduced to Weick and Roberts's theory of collective mind.[7] This theory describes how individual members of a group can act in ways that produce overall reliability in the face of complexity, especially when lack of comprehension can lead to disastrous consequences. The major claim of collective mind theory is that individuals develop shared understandings of the group's tasks and of one another that facilitate group performance. Collective mind theory was adopted because it addressed what had become for us the key question, namely how group members came to know how to contribute to the overall group performance.

Previous conceptions of group mind have been controversial because they seemed to imply the existence of some super-individual entity.[26] By contrast, collective mind is described as an individual's "disposition to heed," hence an emphasis on "heedful" behaviors. If each member of a group has the desire and means to act in ways that further the goals and needs of the group (i.e., "heedfully"), then that group would exhibit behavior that might be described as collectively intelligent, even though it is the individuals who are intelligent, not the group. However, various group processes are crucial for building and maintaining these "heedful" dispositions and capacities, and these processes are the foci of the theory.

We began using collective mind theory approximately one-third of the way into our data collection. Therefore, we were able to modify our interview outlines to cover aspects suggested by collective mind theory for the remaining two-thirds of our interviews. We also reanalyzed earlier interviews from this perspective.

Applicability of the collective mind to requirements definition. Weick and Roberts originally described the collective mind in aircraft carrier flight deck operations.[7] More generally, they listed three features of organizational environments that make the development of collective mind beneficial: (1) the need for high reliability, (2) nonroutine work, and (3) interactive complexity (the combination of complex interactions with a high degree of coupling). When some of the three are absent, it may be possible to satisfy organizational needs more easily, for example, through better training or increased specialization. However, when all three factors are present, it becomes essential for each individual to have a sophisticated conceptualization of the work, the people doing it, and his or her own place in the process. In the remainder of this section, we argue that these three features apply to software requirements development, making this theory applicable.

High reliability vs trial and error organizations. Errors early in the development process may not be as life-threatening as errors on an aircraft carrier, the original setting for collective mind theory, but they quickly become expensive and time-consuming to fix. In the case of mission-critical software, "minor" bugs and inconsistencies can create life-threatening hazards, such as malfunctioning 911 emergency telephone lines or air traffic control software that fails to adequately separate aircraft. Cost and schedule overruns are common in the software industry; these problems are often attributed to the costs of fixing problems that should have been caught during requirements definition. Therefore, software companies, and our two sites in particular, need to be highly reliable.

Nonroutine work. Early software development organizations do not routinely face unpredictable life and death crisis situations such as those studied by Weick and Roberts, but requirements definition does require a certain amount of creativity and improvisation. Each new software project has its own unique problems and characteristics, and the day-to-day ways in which developers respond to those can have enormous scheduling and budget implications.

Interactive complexity. The software systems developed by our sites were probably as complex as the aircraft carrier flight decks that served as the original setting for collective mind theory. Early software development for such software projects is too large a task for a single individual and yet too interrelated to be divided arbitrarily. Developers try to hierarchically decompose systems into pieces small enough to be handled by a single person and with minimal interactions with other pieces. Unfortunately, it is never possible to eliminate the need for interaction. Also, some types of software, including the products of the companies we studied, either require or would benefit from higher integration than is possible to develop under conditions requiring a minimum of interaction. For such organizations, a highly developed collective mind may permit better coordinated actions and thus better products.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 3 Comparison of collective mind features in research sites

Collective Mind Feature	TC	LGC
Contribution	Little contribution to building collective mind, due to segmentation of group and lack of recognition of need	Individuals worked to develop one another's understanding of who was doing what and how the pieces fit together, through group meetings, progress reports, working papers
Representation	Group members often did not understand whole process and reported conflicting ideas about responsibility for feature list	Interviewees had little or no trouble identifying who was responsible for various aspects of the tasks
Subordination	Difficult to assess	Group split into factions, with different goals and approaches
Socialization	Lack of opportunities	Strong, but factionalized
Conversations	Lack of opportunities; some accidental encounters had unexpected benefits	High levels within factions
Recapitulation	Difficult to assess	War stories and key working papers

Collective mind in requirements development. Weick and Roberts identified three individual behaviors[7] that epitomize group tasks: (1) contribution (an individual member of a group contributes to the group outcome), (2) representation (an individual builds internal models of the group), and (3) subordination (an individual puts the group's goals ahead of individual goals).

These actions go on in any group setting. The issue for collective mind is how "heedfully" (to use Weick and Roberts's term) they are done; are they done carefully, appropriately, intelligently? To the extent they are, the group will display collective mind. Although conceptualized separately, these three concepts overlap and reinforce one another to some degree. It is difficult to imagine heedful contributions from even highly talented and motivated individuals with weak representations of the group's needs and structure. Similarly, one cannot build an accurate representation without the contributions of others, nor can one heedfully subordinate without an accurate representation of the group's goals.

At TC, the collective mind seemed weak. In the interviews, we did not find robust understandings of how individual work fit the big picture, a failure of representation. At LGC, we found that requirements were determined by a smaller and much more tightly knit group. Most of the analysts had a fairly good understanding of the whole project and their place in it. However, disagreements about the approach to be used hampered individuals' subordination to the group, again resulting in a weak collective mind and reduced effectiveness. In the remainder of this section, we discuss the three aspects of collective mind at our sites. This analysis is summarized in Table 3.

Contribution. Contribution is the individual's input to the system. Individuals contribute when they perform a task, such as generating a requirement or interpreting a user's request, make a decision, or participate in the social processes that build the collective mind as discussed below. These contributions can be performed more or less heedfully in that individuals can perform tasks conscientiously, make decisions intelligently, coordinate their contributions with those of others, and so forth.

In TC, we found little contribution from individuals toward building the collective mind (as opposed to contributing toward the final product of the group). Such work was difficult to accomplish because of the segmentation of the organization and the lack of encouragement to overcome these obstacles or even recognition that doing so might be important or useful. Although meetings were held and information exchanged through many channels, this information appeared to focus primarily on the work products. (It is possible that efforts were made at times other than our visits, but results of such efforts were not apparent.) Since the collective mind was weak, contributions to the final product were lessened.

In the case of LGC, we saw a somewhat different situation. In addition to producing work products, individuals also worked to develop one another's understanding of who was doing what and how the pieces fit together. There was an organizational emphasis on sharing information among team members and throughout the organization through small and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

large meetings as well as widely disseminated progress reports and working papers. Members were encouraged to and frequently gave presentations on their work in progress, which were well attended both by group members and outsiders. Members frequently took the time to meet with other interested parties to explain what they were doing and how they were doing it. Although doing this extra work to develop the collective mind took time, the work enabled individuals to more efficiently direct their efforts when working on their projects.

Representation. Representation is the group's input to the system that is assimilated to varying degrees by each individual. As individuals do and say things, those actions are interpreted and synthesized by others who use that information to build their own internal model of the group. This model enables them to visualize how they fit in, how others will act, and how their actions will affect others. It embodies their ideas about the goals of the group and how they may be accomplished. Representations are created and acted upon more or less heedfully in that individuals' models can be more or less similar to others' models, individuals' models can be more or less similar to reality, and organizations can engage in processes that disseminate the information necessary to build the models more or less conscientiously.

The important point is that individuals need to develop models of what others do and a shared understanding of the problem they are working on. In particular, developers need to understand one another and the users' needs in order to be able to develop systems that solve their problems. Walz et al. similarly noted the importance of building "shared models of the problem under consideration and potential solutions."[27] Such representations might include what Fischer et al. described as domain models, which they noted were "socially constructed over time by communities of practice."[28]

As a result of the problems discussed above, the group at TC seemed to have undeveloped representations. In our interviews, for example, we found different understandings about how the feature list was developed. Different interview subjects had different ideas about who was ultimately responsible for the list, the steps necessary or desirable to ensure its completeness and consistency, or who would be affected by problems with the feature list. In one case, an engineer who prepared bids reported that he had learned from a more-or-less chance meeting that his work overlapped the work of planners later in the process. In this case, this lack of understanding resulted in duplicate work. The engineer knew how to do his job, but not how what he did could be reused by others.

In the case of LGC, we found that individuals generally had fairly well-developed representations. Interviewees had little or no trouble identifying who was responsible for various aspects of the tasks in which they were involved. Furthermore, although they could not describe the details of someone else's software, they frequently could give an accurate overview of its scope and approach. They also reported that they generally had little difficulty correctly identifying persons whose software would be affected by changes they wished to make in their own code.

Subordination. Finally, subordination involves the reliance of the individual on the system. Individuals subordinate when they trust others to provide needed information, when they obey superiors, and when they make decisions based on the needs of the system above and beyond their own personal needs. The act of heedful subordination separates organizations with highly developed collective minds from structures like markets in which the whole is held together by many individuals acting purely in their own best interests. Subordination can be performed more or less heedfully in that trust can be built on strong or shaky foundations, individuals' representations of the system to which they are subordinating can be more or less accurate, and individuals may or may not choose their own interests over those of the system.

At TC, it was difficult to assess the extent of subordination because of the problems with representation. Individuals did their jobs in the way they believed they contributed best, but they did not always know the overall goal. At LGC, what was particularly lacking from a collective mind perspective was subordination. Our interviews suggested that the group had split into four factions, which caused friction and delays within the group. One faction consisted of the "old-timers" who had been with the company for many years and had extensive experience with air traffic control systems but less with coding large software development projects, such as the prototypes. Another consisted of relative newcomers to the company, who had more experience and education in software development techniques, but less with air traffic control systems. Management had much in common with the old-timers, but was focused on management issues and dealing with the government sponsor. Finally, there was an experimenters' fac-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

tion that used the prototype system to run controlled experiments. Most experimenters had extensive experience with air traffic control systems and with analysis, but not with software development.

Even though the collective minds were quite well-developed within these groups, subordination seemed not to cross the borders. One example of this was when the newcomers wanted to use object-oriented methods to specify the system requirements. The old-timers did not like this idea because they were not familiar with the techniques and because it would be difficult to retrofit inherited software (the majority of the system) to the new paradigm. The newcomers, however, were a majority and were more cohesive as a group. Because of their software expertise, they managed to convince enough people that this approach was the right solution. The group struggled with the new methods, but in the end, without the full support and expertise of everyone, they gained few of the benefits while essentially wasting much time.

In this case, each faction was willing to understand what was happening, but only from the point of view of the faction. The old-timers were unable and unwilling to take advantage of the newcomers' expertise in new methods for large-scale software development, whereas the newcomers were unable and unwilling to take advantage of the old-timers' familiarity with the software and domain knowledge. Thus, representation was good, but it did not extend far enough. In addition, there were not enough individuals in either faction who were able to put aside their own assumptions and biases long enough to really examine the problem and what it demanded, a failure of subordination.

Tools and techniques for developing collective mind. As we mentioned above, many requirements analysis techniques and tools seem to have been designed to improve communications and coordination. However, collective mind theory suggests improving performance by actively supporting the development of representation, contribution, and subordination within the group. In both companies, we believe that a more developed collective mind could have solved some of the problems the group experienced as they developed system requirements.

First, tools and techniques might directly support the development of representation, provide a channel for contributions, or help users understand group needs. Walz et al. note many problems that arose because of difficulties creating a shared memory.[4] However, developing shared representations requires more than simple communications. As Boland and Tenkasi put it, "the problem of integration of knowledge in knowledge-intensive firms is not a problem of simply combining, sharing, or making data commonly available. It is a problem of perspective taking in which the unique thought worlds of different communities of knowing are made visible and accessible to each other."[29] They went on to describe several systems for building and sharing representations of work. Atwood et al. described a system, called Design Intent, for facilitating the development of, "shared problem understandings across problem stakeholders which is the source of the system requirements."[30]

Besides directly supporting these processes, systems might help by supporting the development of the collective mind. Weick and Roberts suggested three types of social processes that underlie the building of collective mind: socialization, conversation, and recapitulation.[7]

Socialization. Collective mind theory suggests that it is important to pay attention to how new members are socialized into a group. People joining a group need to understand how they fit into the process being performed (i.e., their contribution and subordination). They need to be encouraged and educated to interact with one another to develop a strong sense of "how we do things around here" (i.e., representation). The richer the social environment, the richer the understanding can be. The socialization of newcomers is especially important, because in the act of explaining the situation to others, veterans have an opportunity to critically reflect on that situation and change it, effectively resocializing themselves in the process.

Socialization seems obvious, but did not seem to be done well at TC. Individuals who had moved to different positions within the group reported that they encountered a whole new view of the process and their responsibilities within it and had to discover for themselves how best to contribute. In some cases, these roles were new, requiring the new people to define for themselves what they should do, rather than being socialized or trained.

Socialization at LGC, in contrast, was better developed. A strong, almost elitist sense of what it meant to work at LGC was promoted at virtually every level from the moment an employee entered the organi-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

zation. Document and presentation formats, even for internal distribution, were strictly defined and enforced—and they always included the LGC logo prominently displayed. Sharing of one's work and knowledge was not only encouraged, but required.

At a more detailed level, however, there were some breakdowns. As described above, we found that factions had developed. The separation was particularly evident in the socialization of new group members. If a new group member was younger or more software-oriented, he or she would be primarily taken under the wings of other young, software-oriented people (the "newcomers"). If a new group member was older or more domain-oriented, he or she would be primarily socialized by the "old-timers" and management. Thus, whatever differences existed among the factions were perpetuated and perhaps intensified by this process.

Socialization might be enhanced through organizational arrangements such as mentors or co-ops. LGC recruited many employees from its co-op program, which has the advantage that new employees arrive already knowing something about the organization. Lave and Wenger's[31] notion of legitimate peripheral participation may also be useful here. Socialization can also be promoted through computer tools. For example, the Design Intent system[30] allows designers to share information about the project and about themselves with the explicit goal of fostering socialization as well as communicating important design information.

Conversation. Second, Weick and Roberts stress the importance of conversation.[7] It is difficult to build a collective mind if people do not talk to one another somehow. Meetings, social events, hallway conversations, and electronic mail or conferencing are all ways in which group members can get in touch with what others are doing and thinking.

One problem with the groups in TC was simply that individuals had few opportunities to talk to one another because of the volume of work and the division of the process into discrete subtasks performed by different groups. Strikingly, one subject reported benefits simply from being on the same committee as others in the group, consistent with Walsh's observation that "some minimal level of social contact (about anything) in an organization may be necessary to maintain an alert organization mind."[26] A lack of conversation was a particular problem for the groups we were studying because they typically could not deeply involve the users in the design, a key success factor suggested by many requirements analysis techniques.

At LGC, there was a very high level of conversation within factions and some (though not nearly as much) across factions. Of the factions, the newcomers appeared to have much higher levels of conversation. Not only did they work together, often helping each other solve difficult software problems, they frequently had lunch together, played on departmental athletic teams together, and socialized together after work. There were even several relationships and a few marriages that came out of this group! Communication within other factions was also good, though it was somewhat more likely to be formal (e.g., through memo or meeting) and far less likely to include recreational tasks. Unfortunately, establishing communication between management and the other groups was difficult. Management was often off-site, in meetings with government sponsors, and therefore unavailable. Also, management often did not convey all the details of the interaction with the sponsor. Although this was a reasonable tactic given the sponsors' proclivity to frequently change requirements, it created a situation where nonmanagement personnel were unable to form a robust understanding of the sponsors' needs.

Conversation might be enhanced through various arrangements such as widely distributed progress reports or periodic group meetings. Allen[32] showed how the physical arrangement of office space and equipment can increase interaction. Conversation might also be supported with computer tools, such as computer conferences. For example, several computer companies support computer conferences on work and nonwork topics as a way to promote interaction. Walz et al. suggest training group members in dialectic techniques to promote fuller discussion of potentially problematic issues, again improving communications.[4]

Recapitulation. Finally, Weick and Roberts stress the importance of recapitulation.[7] To keep the collective mind strong and viable, important events must be "replayed" and reanalyzed and shared with newcomers. The history that defines who we are and how we do things in an organization must be continually reinforced, reinterpreted, and updated.

Of the three processes, this was the most difficult to detect and probably the least well-developed. At LGC there was some evidence of recapitulation. Individ-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

uals tended to know the histories of others they worked with, for example, knowing that one person had previously worked on a related project or that another had been a pilot or air traffic controller in a previous job. Certain stories were told over and over again, especially humorous anecdotes about life in control towers. Certain working papers and reports were also shared to communicate the core of the project. Some of these were out-of-date but still useful because they described the group's origins.

Recapitulation might be promoted by encouraging debriefing (or "bull") sessions, where individuals recount their perspectives on recent events.[33] Such sessions could prove valuable as a way to socialize newer members of the group, even if they do not directly educate listeners on how to behave. Recapitulation might also be supported by computer tools. Many design tools allow developers to replay decisions, to understand what has been done before and why, as suggested by Walz et al.[4] For example, Boland and Tenkasi describe what they call task narrative forums, in which members of the community can create and share narratives. As they note, "through narrative, the community constructs its practices and its social world by building and restoring its sense of the canonical."[34]

Conclusions

Our initial interviews suggest that the combination of coordination theory and collective mind illuminate some of the problems in requirements analysis. Coordination theory suggests that actors must identify and manage dependencies inherent in the process, whereas the collective mind describes how individuals may learn how to act in ways that enhance the reliability of the group.

Coordination theory seemed to be useful in focusing on the cause of some problems, although less useful in identifying solutions. For example, interdependencies between requirements caused problems, so it was necessary for analysts to identify dependencies or conflicts and manage them. To develop correct and complete requirements, information must be obtained about the needs of users and the abilities of developers, who may or may not be available for consultation. More specifically, we would recommend that TC ensure that the database of feature interactions is kept up-to-date to eliminate at least the most obvious conflicts.

Although coordination theory suggested that these were problems and described what analysts needed to know and do to manage them, it did not clarify for us how members of the team learned what they did. To be fair, sources of this "knowing how" are not a central concern of coordination theory; as Malone and Crowston pointed out,[6] coordination depends on underlying processes of decision-making, communication, and perception of shared objects, but these are not part of the theory itself.

At these levels, the collective mind theory provided a useful adjunct to coordination theory. In a sense, a well-developed collective mind is an alternative coordination mechanism. A group member who can successfully anticipate what others are likely to do can spend less time checking or asking. Without such an understanding, coordination becomes more difficult and time-consuming. Constantine described a similar phenomenon in what he calls the synchronous or harmonious alignment paradigm for group organization,[35] but noted that static performances "tend not to be highly responsive or adaptive to changing requirements." By contrast, a well-developed collective mind should allow group members to anticipate how to react even in novel situations, such as the crises studied by Weick and Roberts.[7]

More specifically, we would recommend that TC directly address the development of collective mind within and across the requirements development group. Opportunities for socialization, conversation, and recapitulation could be strengthened, for example, by providing time and space for social interactions and by consciously reflecting as a group on successes and difficulties near the end of each project. Such interactions might even be supported by a computer-conferencing system. LGC, where the collective mind was generally stronger, might benefit from the increased participation of two groups: management, who could provide a wider perspective on the goals of the project, and the experimenters, who were the immediate users of the simulation.

More generally, it may be that too much specialization may lead to less intelligent action. People need to understand what the other people do and how they do it. Perhaps it is not essential for them to be able to actually take over another position, but they do need to understand well enough to predict how their own actions will affect everyone else and how they are likely to be affected by others. Smaller groups are often thought to be more effective for software development but are not practical for the large

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

projects worked on by the companies we studied. However, coordination and collective mind theory taken together suggest what it is about small groups that makes them work better: small groups more easily develop a collective mind, thus facilitating the coordination of requirements development.

Our study also has implications for those hoping to develop distributed groups that interact primarily or only through electronic media. Because development of collective mind depends on processes of socialization, conversation, and recapitulation, it will probably be difficult for such a group to develop a strong collective mind. It may be that a strong collective mind is unnecessary for the group's tasks, e.g., if the information needed is all relatively unambiguous and at-hand. However, if a strong collective mind would be beneficial or necessary, managers may wish to ensure that the group has opportunities for socialization, conversation, and recapitulation, even if they meet face-to-face only rarely.

In this study, we used coordination and collective mind theory to diagnose problems in software requirements analysis and suggest possible remedies. We did not formally test either theory, and we do not claim to have "proven" either one. Rather, we have demonstrated that both can be useful in understanding these situations. Much work remains before these theories can be formally tested using standard research techniques. A necessary first step is to develop useful measures, e.g., for the various forms in which collective mind is displayed (contribution, representation, and subordination). Perhaps a more appropriate test, though, would be to apply the clinical research perspective.[36] We believe that the two theories provided suggestions for improving the performance of the organizations we studied. This claim can be tested by using our results as a basis for an intervention in these companies. Following this path, we hope at some future date to be able to report on the outcomes of our attempts to improve collective mind and coordination in organizations.

Acknowledgments

The authors wish to thank the two companies involved for their generous support of this research. The paper has also benefited from discussions with Karl Weick and comments from anonymous reviewers.

Cited references

1. F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley Publishing Co., Reading, MA (1975), p. 17.
2. The Standish Group, *Chaos*, The Standish Group, Dennis, MA (1995).
3. W. Meyers, "MCC: Planning the Revolution in Software," *IEEE Software* **2**, No. 6, 68–73 (November 1985).
4. D. B. Walz, J. J. Elam, and B. Curtis, "Inside a Software Design Team: Knowledge Acquisition, Sharing and Integration," *Communications of the ACM* **36**, No. 10, 63–77 (1993).
5. L. A. Macaulay, "Requirements as a Cooperative Activity," *RE'93: IEEE Symposium on Requirements Engineering*, IEEE, San Diego, CA (1993).
6. T. W. Malone and K. Crowston, "The Interdisciplinary Study of Coordination," *Computing Surveys* **26**, No. 1, 87–119 (1994).
7. K. Weick and K. Roberts, "Collective Mind in Organizations: Heedful Interrelating on Flight Decks," *Administrative Science Quarterly*, 357–381 (1993).
8. A. M. Davis, *Software Requirements Analysis and Specification*, Prentice Hall, Englewood Cliffs, NJ (1990).
9. B. Curtis, H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM* **31**, No. 11, 1268–1287 (1988).
10. J. G. March and H. A. Simon, *Organizations*, John Wiley & Sons, Inc., New York (1958).
11. J. R. Galbraith, *Organization Design*, Addison-Wesley Publishing Co., Reading, MA (1977).
12. M. Tushman and D. Nadler, "Information Processing as an Integrating Concept in Organization Design," *Academy of Management Review* **3**, 613–624 (1978).
13. J. G. March and H. A. Simon, p. 142.
14. J. M. Dukerich, F. J. Milliken, and D. A. Cowan, "In-Basket Exercises as a Methodology for Studying Information Processing," *Simulation and Gaming* **21**, No. 4, 397–410 (1990).
15. T. W. Malone et al., "Tools for Inventing Organizations: Toward a Handbook of Organizational Processes," *Proceedings of Second Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, IEEE Computer Society Press, Morgantown, WV (1993), pp. 72–82.
16. E. Davidson, *Joint Application Design (JAD) in Practice*, University of Hawaii (1997).
17. A. M. Davis, p. 20.
18. *Ibid*, pp. 54–55.
19. K. Crowston, *Towards a Coordination Cookbook: Recipes for Multi-Agent Action*, unpublished doctoral dissertation, MIT Sloan School of Management, Cambridge, MA (1991).
20. M. Powers, D. Adams, and H. Mills, *Computer Information Systems Development: Analysis and Design*, South-Western, Cincinnati, OH (1984).
21. W. S. Davis, *Systems Analysis and Design*, Addison-Wesley Publishing Co., Reading, MA (1983).
22. N. D. Birrell and M. A. Ould, *A Practical Handbook for Software Development*, Cambridge University Press, Cambridge (1985).
23. W. S. Davis, pp. 100–170.
24. J. W. Brackett, "Software Requirements," SEI Curriculum Module SEI-CM-19-1.2 (January 1990). Reprinted in *Standards, Guidelines, and Examples on System and Software Requirements Engineering*, M. Dorfman and R. H. Thayer, Editors, IEEE Computer Society Press, Los Alamitos, CA (1990).
25. K. E. Weick, *Sensemaking in Organizations*, Sage Publications, Thousand Oaks, CA (1995).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

26. J. P. Walsh, "Managerial and Organizational Cognition: Notes from a Trip Down Memory Lane," *Organization Science* **6**, No. 3, 280–321 (1995).
27. D. B. Walz et al., p. 63.
28. G. Fischer, S. Lindstaedt, J. Ostwald, M. Stolze, T. Sumner, and B. Zimmermann, "From Domain Modelling to Collaborative Domain Construction," in *DIS '95*, G. M. Olson and S. Schuon, Editors, ACM Press, Ann Arbor, MI (1995), pp. 75–85.
29. R. J. Boland and R. V. Tenkasi, "Perspective Making and Perspective Taking in Communities of Knowing," *Organization Science* **6**, No. 4, 350–372 (1995), p. 359.
30. M. E. Atwood, B. Burns, D. Gairing, A. Girgensohn, A. Lee, T. Turner, S. Alteras-Webb, and B. Zimmermann, "Facilitating Communication in Software Development," in *DIS '95*, G. M. Olson and S. Schuon, Editors, ACM Press, Ann Arbor, MI (1995), pp. 65–73.
31. J. Lave and E. Wenger, *Situated Learning: Legitimate Peripheral Participation*, Cambridge University Press, Cambridge (1991).
32. T. J. Allen, *Managing the Flow of Technology*, MIT Press, Cambridge, MA (1977).
33. J. Orr, "Narratives at Work," *Field Service Manager*, 47–60 (1987).
34. R. J. Boland and R. V. Tenkasi, p. 367.
35. L. L. Constantine, "Work Organization: Paradigms for Project Management and Organization," *Communications of the ACM* **36**, No. 10, 35–43 (1993).
36. E. H. Schein, "Lessons for Managers and Consultants," *Process Consultation*, Addison-Wesley Publishing Co., Reading, MA (1987), p. 64.

Accepted for publication January 6, 1998.

Kevin Crowston *Syracuse University, School of Information Studies, 4-206 Centre for Science and Technology, Syracuse, New York 13244-4100 (electronic mail: crowston@syr.edu).* Dr. Crowston recently joined the School of Information Studies. He received his Ph.D. in information technologies from the Sloan School of Management, Massachusetts Institute of Technology (MIT) in 1991. Before moving to Syracuse, he was a founding member of the Collaboratory for Research on Electronic Work at the University of Michigan and the Centre for Coordination Science at MIT. His current research focuses on new ways of organizing made possible by the extensive use of information technology.

Ericka Eve Kammerer *University of Michigan Business School, 701 Tappan, Ann Arbor, Michigan 48109-1234 (electronic mail: eek@umich.edu).* Ms. Kammerer is a doctoral student. In her dissertation research, she is using narrative techniques to study the development of community in Usenet discussion groups.

Reprint Order No. G321-5675.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.