Using a Configuration Management Tool to Coordinate Software Development

Rebecca E. Grinter
Computers, Organizations, Policy, and Society
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425

rgrinter@ics.uci.edu
http://www.ics.uci.edu/dir/grad/CORPS/rgrinter

ABSTRACT

I describe a naturalistic study of one organization's use of a configuration management tool to coordinate the development of a software product. In this organization, the developers use the tool routinely to reduce the complexities of coordinating their development efforts. I examine how the tool provides mechanisms of interaction that let the developers work with each other. I identify four aspects of these mechanisms: difficulties of representing work, the multiple levels that they operate at, the possibilities for coordination they provide, and their role in supporting a model of work.

KEYWORDS: configuration management (CM), computer-supported cooperative work (CSCW), organizational memory, articulation work, mechanisms of interaction.

INTRODUCTION

In the last ten years there has been an explosion in the amount and kinds of software. A number of factors have contributed to this explosion, including: the multitude of hardware platforms that need to be supported, market competition pressures, and the ability to build more functionally complex software. Factors such as these, as well as the increasing demand for new innovative software, have encouraged the industry to grow. In response the industry has employed more people to build software quickly and reliably.

However, as Fred Brooks (1974) observed, adding more people to a software project does not necessarily decrease development time. What software project managers like Brooks discovered were problems of coordinating groups of developers working on the same project. Software engineers have typically addressed the difficulties of group work by developing formal procedures that structure the work of building software (Pickering and Grinter, 1995). These formal procedures include: modularization, process models, and formal methods.

COOCS 95 Milpitas CA USA © 1995 ACM 0-89791-706-5/95/08..$3.50

Software engineers have also built systems that provide automated support for some of these formal procedures. These technologies are groupware aimed at supporting software development work. Yet despite their existence few researchers outside of the software engineering community have explored how these systems support group work in practice (but see Orlikowski, 1991; Hughes et al., 1994).

In this paper I describe one system designed to resolve some of the challenges of coordinating group work. This particular technology, a configuration management (CM) tool, incorporates a configuration management approach to handling the complexities of managing software development. The paper begins with a description of configuration management, and how it involves coordinating the work of multiple developers. Next, the paper focuses on the types of formal procedures embedded into CM tools and how developers use those procedures in their work. I also describe the times when formal procedures do not help the developers and what they do in order to maintain the coordination required to develop a software system. Finally implications for research in supporting the work of groups are discussed.

CONFIGURATION MANAGEMENT AND COORDINATION WORK

Managing the Evolution of Software

Configuration management addresses the problems of managing the evolution of software (Bersoff et al., 1979). Software is hard to manage for three reasons. First, developers can easily change code. Second, the modifications can affect the behavior of the entire system because of the interdependencies among modules. Third, because teams develop software the changes one person makes often impact the work of others. Configuration management procedures focus on controlling developers' abilities to alter code. By controlling the changes, configuration management tries to ensure that the evolution of the software product goes steadily.

Small development teams often manage the evolution of software by communicating with each other. As one informant in this study explained:

> We used to have this really cool way of handling it, (software development) where we'd do the whole release in a single version. Everybody just, it was public and everybody modified it, well we had a very small team so it was manageable because we could talk over the walls to each other.

When teams grow larger they can no longer communicate at that level of detail about the changes made to the system. Instead they use procedures such as reports and announcements to notify everyone about the status of the software. These procedures are intended to keep software from being changed in ways that adversely affects the work of others.

First generation configuration management tools automated these mechanisms by using access controls.[1] Using a library metaphor for programming, these tools used "check-out" and "check-in" states to control changes to software. To make any modifications to a module of software developers had to check out the code. When a developer checked-out a module, the tool made a new version of the code and prevented others from checking out the same software. When changes had been completed, the developer checked in the code. A checked-in module was stable and usually working. Other developers could read and execute it with their own modules. By checking-out and checking-in code developers created successive versions of the module that the system stored.

These systems supported software development by providing code versioning. However, they had two disadvantages. First, they only worked for code. Software systems contain more than just code, including: libraries, test suites, makefiles, and documents. Modern configuration management systems use database technology to manage large data repositories that store all the types of artifacts that make up a system.

Second, the checked-out state turned out to be very limiting because it prevented others from changing the same module at the same time that slowed down developers' ability to get their work done. Modern systems solved this problem by allowing parallel development. If one developer checks out a module they create a new version of that code to work on. If a second developer also checks out the same module, from the last checked-in state, then they create another new checked-out version of the code. Both developers can now make their changes and then integrate them.

Modern CM tools also support three other layers of functionality on top of the check-out/check-in model (Caballero, 1994). The configuration control layer maintains information about the artifacts that form a software product. It knows which versions comprise a specific system and how they relate to each other. This has a number of advantages in modern software development contexts. Typically a software development organization builds a family of products for different hardware platforms. The configuration control level allows developers to find out exactly which artifacts belong to a certain hardware platform. Often development organizations support older products as well as developing new releases. The configuration control layer allows developers to recreate both previous and current releases of any software stored inside the CM data repository.

The process management layer provides a "life cycle" for each type of artifact stored in the system. A life cycle consists of a number of states. For example a typical life cycle for a software module consists of the checked-out, checked-in, quality-tested, and released states. While the developers are most concerned with the checked-out and checked-in states, testers of the software use the quality-tested state to signal that a particular version of a software module has passed rigorous system testing.

Finally, the problem reporting layer supports bug and enhancement tracking. Modifications to the artifacts in the system occur as a result of problems with the function of the system or enhancements requested for future products. The problem reporting layer provides a way of linking the bugs or enhancements to the changes themselves. Modern CM tools either have built in process management and problem reporting, or provide the necessary connections to allow users to build it themselves or purchase another off-the-shelf system and integrate it into the CM tool.

Configuration Management and Articulation Work

Configuration management specialists do not describe difficulties of group work in ways that seem familiar to information systems researchers. They focus on the challenges of managing projects rather than the social dynamics of teams. However, they recognize that group work creates added complexity for managing software development. For example, Babich (1986) describes configuration management as:

> ...the day-to-day and minute-by-minute evolution of the software inside the development team. Controlled evolution means that you not only understand what you have when you are delivering it, but you also understand what you have while you are developing it. Control helps to obtain maximum productivity with minimal confusion when a group of programmers is working together on a common piece of software. (Babich, 1986 p. vi.)

Clearly he believes that configuration management can reduce the complexities of coordinating developers.

One stream of research has characterized the coordination of work as *articulation work* (Strauss 1985, 1988). Strauss defines articulation work as:

> Articulation work amounts to the following. First the meshing of the often numerous tasks, clusters of tasks, and segments of the total arc. Second, the meshing of efforts of various unit-workers (individuals, departments, etc.) Third, the meshing of efforts of actors with their various types of work and implicated tasks. (The term "coordination" is sometimes used to catch features of this articulation work, but the term has other connotations so it will not be used here.) (Strauss, 1985. p 8)[2]

Articulation work is all the coordinating and negotiating necessary to get the work at hand done. In this case software developers work on building systems. However because they work in groups they must also coordinate their changes with other people's modifications. Because CM tools attempt to support their articulation work electronically they are groupware systems.

---

[1] The early systems are also known as version control systems. Two popular systems are Revision Control System (RCS) and Source Code Control System (SCCS).

[2] The total arc Strauss refers to "consists of the totality of tasks arrayed both sequentially and simultaneously along the course of the trajectory or project." (Strauss, 1985. p4)

Mechanisms of Interaction

Schmidt and Bannon (1992) have applied the concept of articulation work to the research problems in the computer supported cooperative work (CSCW) community. They describe how individuals engage in articulation work as part of their daily routines. They say:

> However in 'real world' cooperative work settings ... the various forms of everyday social interaction are quite insufficient. Hence articulation work becomes extremely complex and demanding. In these settings, people apply various *mechanisms of interaction* so as to reduce the complexity and, hence, the overhead cost of articulation work ... These protocols, formal structures, plans, procedures, and schemes can be conceived of as *mechanisms*... And they are *mechanisms of interaction* in the sense that they reduce the complexity of articulating cooperative work. (Schmidt and Bannon, 1992 p. 18-19, italics in original)

Examples of these mechanisms of interaction include plans, and standard operating procedures. These mechanisms supplement forms of social interaction like e-mail, video conferencing, and other forms of communication.

Although Schmidt and Bannon do not explicitly say that group size might render "the various forms of everyday social interaction" insufficient for articulation work but this must be a factor. Large groups of "project size," as Grudin (1994) (based on Curtis et al., 1988) calls them, can not find out what the status of the project is by social interaction alone. The overhead of finding people, speaking with them, and having it happen so quickly that nothing has changed is unrealistic.

From experiences of managing software projects, configuration management specialists devised mechanisms of interaction to organize the development process. They also developed computer systems to support configuration management. However, configuration management specialists did not adopt solutions to increase the communications bandwidth such as e-mail. Instead they embedded mechanisms of interaction, the configuration management procedures, into a CM tool.

CM tools provide an opportunity to examine computerized mechanisms of interaction. In this study I discuss four aspects of the mechanisms of interaction supplied by a CM tool: difficulties of representing work, multiple levels of mechanisms, new possibilities for coordination provided by mechanisms of interaction, and models of work supported in these mechanisms. For each of the aspects I explore the interplay between the mechanisms of interaction and the social interaction necessary to coordinate software development work.

ORGANIZATIONAL SETTING

The CM tool market has grown rapidly in the last few years and market analysts estimate that it will be worth approximately $1 billion worldwide by 1998 (Ingram, 1994).[3] This study focused on the development division of one CM tool vendor who I call "Tool Corporation," that competes in an oligopoly for this market.

Specifically I studied how the developers responsible for building the CM tool actually use their CM tool to manage their work. The group consisted of 14 members, including the manager, and software testing group, who also use the tool in their daily work. Because the developers use the CM tool to build the latest version of CM tool itself, they are experts in using it.

Obviously, studying expert users of the CM tool affects the conclusions that I can draw, but it also offers several advantages. While there are some studies of electronic mail usage few researchers have studied the use of more sophisticated CSCW systems. There are even fewer studies of the use of advanced CSCW systems in the workplace. Studies of CSCW technologies in workplaces have focused on the adoption of CSCW technologies (Orlikowski, 1992; Bowers, 1994). These studies, and others such as (Grudin, 1989), point out the difficulties that users had adapting to the new technologies.

Orlikowski's (1992) study of the adoption of Lotus Notes™ in Alpha Corp. revealed that the users of the system did not have cognitive frames that matched the technology. Specifically, the users did not completely understand the group features of the system and typically used single user applications. By studying a group of experts who have used the technology for some time I did not encounter problems of cognitive frames.

The users of a CSCW network that Bowers studied understood the purpose of groupware technologies. They were actively engaged in examining the potential uses for CSCW technology in the British government. They developed and used the network to learn more about the technologies. However, Bowers found that the users had trouble establishing new ways of working with the network. In this study the software developers had progressed beyond the initial difficulties of customizing their work routines to work with a new technology.

The ways in which the developers used the CM tool and the problems that they had are reminiscent of Suchman's (1983) study of purchasing staff. She showed how the staff understood the formal procedures governing their work. She also demonstrated how those formal procedures did not recognize the contingencies in their day to day activities. One distinction between Suchman's study and this one is that the configuration management procedures used to coordinate software development are computerized.

METHODS

I conducted a three and a half month on-site interpretive study of the company in mid-1994. I adopted participant, non-participant observation, and interviewing strategies to collect data (see Jorgenson, 1989). Supplemental material was gathered by reading journals, reports, electronic discussion lists, and company documents.

My participant observation included: helping with development activities, including usability testing, multiple

[3] Drives for bringing quality into the software development process include the Software Engineering Institutes' (SEI) Capability Maturity Model (CMM) and the ISO 9000 series of standards.

user testing, reviewing documentation, and attending meetings. I also had full access to the development environment created by the CM tool so I could watch the work in progress. I used two interview gathering strategies, informal interviewing and semi-formal interviewing. The interviews lasted anywhere from 20 minutes to 2 hours. The semi-formal interviews were taped and transcribed and the informal interviews were written up after they took place. In total 20 semi-formal interviews and 80 informal interviews took place.

Initially, data analysis concentrated on understanding how the developers used the CM tool to coordinate their work. The initial informal interviews were used to confirm and detail usage patterns. At the same time, the distinction between CM tools and more traditional forms of groupware such as e-mail and video-conferencing started to emerge, that led me towards a conception of this tool supporting mechanisms of interaction, rather than social interaction. This observation was used to develop the interview guide for the semi-formal interviews. Developers were encouraged to discuss their use of the CM tool and what they do when the tool does not support them. The final stages of data gathering and analysis focused heavily on fleshing out the concepts.

THE ROLE OF CM TOOLS IN COORDINATING WORK

The Case of Parallel Development

The developers call the times when more than one person has the same module checked out, "parallel development." This happens when different developers have changes that require them to work on the same module. Despite having mechanisms of interaction to support the activity the developers find parallel development difficult.

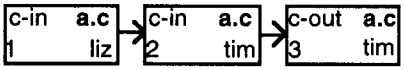

Figure 1: The time line view of an artifact

The tool provides a mechanism of interaction that helps the developers to choose whether to engage in parallel development. The CM tool maintains a time line view of the evolution of every artifact in the data repository (see Figure 1). The time line shows the history of an artifact's development as a series of boxes and lines that chart its evolution over time. Each box represents a version of the artifact and that corresponds to a time in the development of the artifact. The boxes show the name of the artifact (a.c), the version number (1,2, and 3), the person who worked on the artifact (Liz, and Tim), and the state of development then (checked in, and checked out).

Developers use the time-line views of modules to find out whether anyone else is currently working on the code they need to alter. In the time line view shown in Figure 1, Tim has the latest version of a.c checked out for changes. All the developers working on this project can also see that Tim has the module checked out, because they have access to the same time line view. This allows them to make decisions about whether they want to engage in parallel development. Often if developers see that someone has the latest version checked out, they either ask the person working on it to incorporate their changes into that version, or try to work on some other task.

However, sometimes the developers can not avoid parallel development. Their changes may be too complex to ask another person to work on, or they may be too critical to postpone until parallel development can be avoided. So the developers check out another version of the module. At this point, even if they have looked at the view, the system flags them with a message telling them that they have made a parallel version.

When the developers have completed their changes they usually have to merge their code with the changes made by the other person.[4] The person who finished last takes responsibility for merging their work with the other person's.[5] The tool supports merging by providing a facility that compares the two files and displays the lines that differ. The developer responsible for merging selects the lines that need to appear in the integrated module.

Merging can be easy when the developers have changed different parts of the module, for example if someone has changed the comments and another person has altered the functionality. Developers find cases like this easy because the changes involve distinct parts of the module and that shows up clearly in the merge display. In these easy cases the developer simply merges the modules without consulting anyone.

However, sometimes merging becomes too difficult for a developer to do without communicating with the other person who worked on the module. The times when this happens usually occur when both the developers have modified the same lines of code or algorithm. When this happens the complexity of merging rises because suddenly differences become embedded in the context of how a module works, what problems and enhancements the developers were working on, and which solution developers chose to implement.

At this point the developer responsible for merging finds the other person who also modified the module. They discuss what they did, explaining their programming strategies, the problems they solved, and the functionality that they believe the module possesses. They work together to develop a shared understanding of both modules, and determine the functionality of the merged module. This activity often takes place as a joint merging effort. The developers sit around one terminal and select the lines that should go into the final merged module.

Developers avoid parallel development because of the potential complexities of merging. The difficulty of coordinating the efforts of multiple developers in a single module cuts into their development time. The mechanism of interaction that formalizes merging tries to eliminate some of these difficulties, but clearly it breaks down when the developers make changes that interact with those made by their co-workers.

---

[4] Sometimes it is not necessary to merge modules at all, for example if the changes are hardware platform specific.

[5] Although there may be many parallel versions the CM tool can only merge two versions of the module at a time. In UNIX™ parlance it has a graphical "diff" facility.

Levels of Mechanisms of Interaction

The mechanisms of interaction embedded in the CM tool support coordination among developers by providing information about the work of others. For example, developers use the time line view to find out whether anyone is working on a certain module. Without that facility developers must coordinate their actions (engage in parallel development or defer because someone else has the code checked out) by communicating with all the other developers to find out what they are currently working on.

The developers also have access to information about the status of all artifacts related to the one that they are working on. Each developer has a view of the module that they are presently working on and all the artifacts that it relates to.[6] For each related artifact this view provides information that includes the name of the artifact, the developer who has most recently worked on the artifact, and the state of the artifact.

Developers have the option to "reconfigure" this view of related modules. A reconfigure causes the system to update the view, potentially revealing changes in an artifact's name, a new developer working on an artifact, or a change in the state of an artifact. Reconfiguring the view provides important information to developers because of the changes it shows. For example, if a developer sees that a related module has been checked in they know their software must work with that code. This view enables developers to continually coordinate their efforts as they alter related modules simultaneously. Often developers find after reconfiguring their view that their module does not work with some of the latest changes made to related modules and they must fix their work.

While the tool provides mechanisms of interaction to help coordinate development it did not support higher levels of system understanding. During discussions with developers I noticed that they often referred to a lack of a software architecture for the product being developed. Software architectures:

> permit designers to describe complex systems using abstractions that make the overall system intelligible. Moreover, they provide significant semantic content that informs others about the kinds of properties that the system will have... (Garlan and Perry, 1994 p. 363)

Software architectures represent another mechanism of interaction supporting collaborative work at a higher level of systems abstraction.[7] The collaborative work that software architectures support differs from the development work I have described. Instead of being concerned with the details of development work, the emphasis is on locating your work in the "bigger picture."

In this case, some developers had problems explaining the architecture of the tool they were building. While they understood their own sub-systems well, and many had worked on different sub-systems, the developers did not know the conceptual structure of the product being developed. This problem was exacerbated by the changes in design that occurred throughout its development that altered the architecture. The developers were unable to establish that shared understanding through communication as a large group.

The system had no mechanisms of interaction for showing developers how their individual work fitted into the "bigger picture." Without them, and unable to communicate the vision, these developers felt that they could not fully understand the work of developers who worked on sections of the system remote to theirs. This raised concerns amongst these developers that they could miss opportunities to share solutions.

Many levels of mechanisms of interaction must exist to support collaborative work. In this study, the CM tool supplied mechanisms of interaction to help developers coordinate developers their daily work. However, developers also wanted mechanisms of interaction that abstracted away the details of particular artifacts, showing higher level views of the system, providing information about the connections between sub-systems so they could share solutions and locate their own efforts.

Organizational Memory: Creating New Forms of Articulation Work

Developers often rework existing code, modifying it to fix repairs and add new functionality (Lubars et al., 1993). When developers reuse old code they often find themselves trying to work with code that someone else wrote. The job of development then becomes the task of aligning your efforts with the work of the previous developer. The complexity of working with other's code increases when the developer who originally wrote the code has left the organization or is assigned to a different project (Fischer et al., 1992).

The CM tool attempts to solve problems associated with reusing code by maintaining a record of changes made to the artifacts using the problem reporting facility. Developers use the problem reporting system to log problems and enhancements. When developers alter any artifact the tool forces them to link the new version of the artifact to one of the problems or enhancements in the problem reporting facility. The CM tool stores these links, and over time they build into a memory of which artifacts changed as a result of a certain problem or enhancement. In this organization the memory has been growing for 2 years.

These links are augmented by a free form comment field where developers can describe their changes. The CM tool stores the comments so the organizational memory contains problems and enhancements, the artifacts changed, and often descriptions by developers of how they implemented the solutions.

Developers use the organizational memory in a number of ways depending on their experience. Those developers who have worked at Tool Corp. for several years often do not need to consult the memory to remember why a change was made, but other developers do not have this personal experience. Two

---

[6] Artifacts relate to each other in different ways. Modules "call" each other, invoking the actions of others during program execution. Libraries relate to modules by providing collections of callable routines to the software.

[7] Of course software architectures have important technical properties, such as supporting the reuse of software, and comparing and contrasting systems to learn more about software systems.

cases emphasize the different uses made of the organizational memory.

About two thirds of the way through the study the company decided to change a naming convention used throughout the system. The name was embedded in the product, appearing on screens and named in commands. The manager assigned a number of developers the task of going through the system and changing all instances of the old name to the new name. Fortunately for the developers this name changing had already happened once before, and the code changes were linked to one problem describing the previous name change.

Instead of searching through all the artifacts by hand the developers used the organizational memory. They began with the problem and found all the artifacts that changed: those containing the name. This found most of the instances of the name, only excluding modules created after the last name change. The developers also used the free form comments to find out whether the previous developers had experienced any difficulties when they did the earlier name change.

The person in charge of interface development also uses the organizational memory. He described his role as a code maintainer, rather than developer, emphasizing that he primarily worked on amending and expanding existing code written by another developer. Because he often found himself editing parts of the interface code that he did not write himself, he consulted the organizational memory to see what the developer who wrote the code had said about the task at hand. This provided information about which kind of solution to pick.

I have only discussed two examples of the use of the organizational memory at Tool Corp. but they illustrate its main use: to learn about what previous developers did. The memory allows developers to coordinate with others over longer periods of time, such as months and years. The memory gives them the ability to leverage from the experiences of others. The organizational memory is a mechanism of interaction that creates these coordination possibilities.

However the organizational memory has a limitation. When development proceeds at a relaxed pace people usually take the time to explain what they did in the comment field. The pressure of tight project deadlines encourages developers to write less in the comment field. When other developers review these comments they do not understand what happened in detail that makes the comments almost meaningless.

Despite this limitation, the organizational memory supports some coordination between actions in the present and work done in the past. This mechanism of interaction can not be replaced by communication when the authors of the original software have left the organization. Even when the original developers have not left, the organizational memory provides developers some starting points for learning about the artifacts they must change. However, the organizational memory suffers from the difficulties of conveying enough context about code changes. When the demands of the organization do not leave the developers with time to write useful and insightful comments it limits the usefulness of the organizational memory.

Mechanisms and a Model of Work

The previous sections described mechanisms of interaction provided by the CM tool and how they support work. This section focuses more broadly on how these mechanisms constitute a model of how software development work should be done. The model allows developers to make assumptions about their environment and constrains their actions.

The mechanisms of interaction create a model of software development because the procedures support certain ways of working. For example, I described the view of relations between artifacts. This provides a standard way of understanding the current work environment that contrasts with conventional file arrangements like directories. Directories give developers complete discretion about how to arrange the artifacts they create and use, and people organize files differently. This system supports one arrangement, *is_related_to,* which means that when a developer works on one artifact all the artifacts related to it appear. Developers find this standard file structure particularly useful when they work on unfamiliar sections of the system. Instead of having to figure out how the artifacts relate, by looking at arbitrary arrangements of directories, developers automatically know what dependencies' artifacts have.

However, sometimes these models of work do not recognize all the details of software development. This happens when the real work does not match the model of software development embedded in the procedures. When this happens developers must choose to bend the rules of work around them.

Developers at Tool Corp. have special privileges to create and assign themselves problems using the problem reporting facility. This was not the intention of the tool that had a concept of a managerial group entering problems and developers responding to them. However the group tasked with creating and assigning problems only met every couple of days. If developers finished their assignments before the next meeting then they would be unable to work because the system would not accept module changes without an associated problem. The developers used system privileges reserved for system administrators and worked around the constraints of the CM tool.

While the rapid development times formed the main reason for giving developers these problem reporting privileges, they had an unintended payoff for managers. Often developers needed to make small changes to one artifact and so they created and assigned problems to themselves. Managers benefited because they did not have to read through and assign all these small problems leaving them to concentrate on the major system problems.

In another case a few developers violated the procedures imposed by the system. One example of this related to the difficulties of testing software. Testing software requires developers to run numerous tests that not only ensure the integrity of the artifact, but the reliability of its interactions with other artifacts. Developers have to test all the possible interactions that a module has with other artifacts that means generating tests for all the permutations of run-time behavior.

Sometimes after checking the module in developers realize that their code may not work when a certain sequence of actions occurs. Other times they have worked on a problem that spans

many artifacts and they realize after checking in all the modules that they forgot to make one part of the change.

If the developers discover that they need to make further changes they have two options: check out another version of the code, create a problem to assign to that code, and fix the problem, or cheat the system. Often developers go to the trouble of making another version, but occasionally they edit the artifact in the checked-in state. They do this despite knowing that the changes they make in the checked-in state might affect the work of other developers who assume that checked-in files do not change.

The CM tool creates both opportunities and constraints for developers. In this case the tool provides them with mechanisms for sharing work, and coordinating their software development efforts with others. When the processes do not match the realities of the work then two results may occur, developers may bend the model to work with it or reject it entirely.

IMPLICATIONS FOR SUPPORTING GROUP WORK

Articulation work has much promise in studies of CSCW, and this research represents an initial step in this direction. I explored the mechanisms of interaction concept in the study of developers using computer support to coordinate their work. I identified four aspects of these mechanisms of interaction: difficulties in representing work, the different levels, the possibilities they create for new types of coordination, and how the model embedded in them operates. In this section I discuss the impacts of these discoveries for research investigating the role of technology in coordination work.

Difficulties of Representation

In his study of navigation, Hutchins (1990) described how a team of ship navigators worked together to guide vessels into harbor. Part of his study examined the role of navigational instruments. Specifically, he discussed how the instruments provide representations of the navigation problems the team faced. He argued that these instruments supported the work of navigators because of the way that they represent the problems. Much navigation work involves mathematical relations and instruments that have these formulae built into them allow navigators to find the solutions easily. In this way the instruments reduce the complexity of the problems of navigation work.

The configuration management tool supports mechanisms of interaction that create visibility into the development process. Without these mechanisms the evolution of software has little tangible form other than hundreds of files, in many directories, stored on several machines. These mechanisms reduce the complexity of software development by providing views that show the current state of development.

However, in the case of parallel development I saw the difficulties of finding adequate representations. The work it takes to merge modules depends on the complexity of the changes made to both versions of the module. Sometimes merging goes smoothly, the tool support suffices, and the developer can merge without any communication. However, when the same lines of code or the same algorithm changed the merging representation does not enough support to allow one developer to merge alone. At this point the developer needs to find and communicate with the other developers who worked on that module.

This tool provides representations that often support the coordination work of developers. However, this study reveals the challenge of finding a representation of a certain problem that works for all cases. In the case of the merging problem, the representation failed to support the developer's coordination work when the complexity of the changes increased.

In a study of CSCW tools and concepts, Robinson (1991) makes a distinction between the formal and cultural levels of language. The formal level consists of elements that can be discussed by multiple participants without requiring interpretation. The tool provides developers with information that clearly shows which modules need merging and what the associated problem reports are. The cultural level focuses on the remaining elements, those requiring explanations to become meaningful to other participants. For the developers this level involves understanding how the modules were changed, and how those changes interact with each other, and what the combined functionality needs to be. Robinson (1991) claims that when a system does not support both levels of language then it becomes unusable.

When merging reached the cultural level the representation provided by the tool did not help developers. Because developers needed to know much more about intent, meaning, and interpretation of someone else's changes, the view of the different lines of code did not suffice. At these times the strength of the tool came from its proximity to a variety of alternative technologies, such as electronic mail and the telephone.

Multiple Levels of Articulation Work

Previous research has shown that individuals help others coordinate with them by making their work activities' public (Heath and Luff, 1991). Researchers have drawn mixed conclusions about the benefits of using computers to make people's work visible to others (Bowers, 1994; Sommerville et al., 1993; Zuboff, 1991). However, these authors agree that to work well, these systems must be supported by appropriate organizational policies that explain the role of the technology.

The software developers had grown accustomed to working with this visibility into their work and that of others. Specifically, the system provided low level visibility into the current actions of others through the reconfigure and time-line views. It also allowed developers to see what other developers did to artifacts in the past. However, the CM tool did not create visibility into the system as a whole. At a higher level of system abstraction, removed from the details of individual changes, the developers could not see how their work, or other people's, fitted together.

Tools designed to support collaborative work through visibility need to develop mechanisms of interaction that span multiple levels of system abstraction. This is especially true when the group cannot coordinate in other ways. In this case there were 14 software developers, and communicating enough information to establish and maintain an on-going shared understanding of the new product as it emerged in development had enormous costs, which they could not pay.

In a discussion of the articulation work involved in a project, Strauss (1988) describes a concept he called the articulation process that is:

> The overall process of putting *all* the work elements together *and* keeping them together represents a more inclusive set of actions that the acts of articulation work. (Strauss, 1988 p. 164, italics in original.)

The CM tool may have supported the articulation process by visibility into the overall product structure. It is unlikely that computer technology can support the articulation process in absence of other forms of articulation work. However, this study asks whether computers can provide some of the visibility into the overarching process.[8]

Technological Possibilities for Articulation Work

Focusing on mechanisms of interaction allowed me to look at the role organizational memory plays in supporting collaborative work. The developers use it to align their efforts with the work other developers did in the past. Sometimes it facilitates a new possibility for collaborative work, coordinating with the legacy of another person who has since left.

The organizational memory differs from other ways that the organization could support this kind of collaborative work. A common alternative for software engineers is to put comments next to the lines of code they change explaining what it does. Code commenting is usually optional. This memory provides an analogy to commenting in the free form comment field where developers can describe the changes that they made. However, by forcing the developers to link all the changes they made, that may involve more than one artifact, to a single problem the memory provides an additional function. It gives developers a history of why design changes were made and which artifacts were involved, as well as what those changes were.

Because the organizational memory remains up-to-date it differs from internal documentation and specifications. In fact the organizational memory provides reporting features that generate internal documentation. However, by being on-line and up-to-date it maintains a higher degree of accuracy.

However, organizational forces influence the usage patterns of organizational memories. In this case, the need for efficiency collided with the documentation of how problems were fixed. Other researchers have pointed this out, for example Orlikowski's (1992) study shows how the up-or-out promotion system within Alpha Corp. had negative impacts on the usage of Lotus Notes™. Instead of wanting to share consulting techniques using the system, those caught in the promotion structure kept their strategies to themselves, rather than helping others and potentially ruining their own careers.

This study of software developers does reveal that organizational forces influence the usage of technologies long after their initial adoption. This organizational memory had been in place for 2 years. The problem was not that the organization needed to adopt policies that encouraged individuals to use the new technology, but that when project deadlines got tight then developers focused on development rather than documentation. In other words, the usefulness of technology remains contingent on the organization long after its initial introduction.

Models of Work

Gerson and Star (1986) observe that mechanisms of interaction require articulation work because of the unforeseeable contingencies of real work. For example, manufacturing schedules often require adapting due to delays that occur in the production cycles. In that case people have the option of working together and mutually adjusting the schedule. Mechanisms of interaction embedded into a CM tool can not be changed so easily, which is reflected in developer strategies for dealing with the model of work imposed on them.

The procedures allow the developers to make assumptions about their working environment. For example, the developers know who is working on artifacts related to their own. This saves them from the work required to find the latest changes of the software.

However, these procedures do not work when the contingencies of their work deviate from the model. One case of this involves the special privileges for creating and assigning themselves problems. They use this to circumvent the procedure for problem assignment, the meeting where managers and testers assign problems. In this case the system provides an acceptable work around for its own limitations.

Sometimes developers deviate from the model because it does not support their work. In a study of the use of a CASE tool, Orlikowski (1991) describes how the model of work built into the tool reinforced the organizational status quo. Importantly, she observed that all the rules embedded into the tool were subject to interpretation by the people using it. The developers at Tool Corp. have a high degree of latitude in their interpretation, sometimes because the system supports them bending the rules, and other times because they have the expertise to break the mechanisms. The former was viewed by management as a necessary safety valve, but the latter was condoned and caused problems for other developers. More research is needed to examine the limits of models of work, and how they can become flexible enough to accommodate more of the real work of software engineering.

CONCLUSIONS

This paper described how one tool supported the collaborative work of a team of software developers. This case was unusual because the software developers knew the tool intimately and had incorporated it into their everyday working routine. However, this study provided a rare chance to look beyond the challenges of groupware adoption to discover the issues surrounding routine usage.

I found that mechanisms of interaction play a critical role in supporting the coordination of software development. Further research characterizing how mechanisms of interaction do support coordination work, and how organizations develop them, remains to be done. This study offers important starting points for that work.

---

[8] Since the completion of the study the organization has taken steps to resolve the issue of developing an understanding of the software architecture. Currently they have individuals who act as the software architects.

This study suggests that mechanisms of interaction do not support all the articulation work required to build software. Despite well-defined policies surrounding tool usage and a good cognitive understanding of the tool, coordinating software development remains difficult. The developers still use other communicative solutions to overcome challenges of coordinating work.

Currently no groupware solution offers the total solution to the complexities of coordination work. Much work remains to be done exploring how technologies can support the channel changing which allows software developers to get their work done. Learning about the limits of mechanisms of interaction, and how they can be extended, should be coupled with an examination people switch from one form of articulation work to another.

Acknowledgments

First, I would like to thank the Engineering and Physical Sciences Research Council (UK) for financial support. I am deeply indebted to Jonathan Grudin, Jeanne Pickering, and Jim Whitehead for their help and encouragement in formulating and pursuing these ideas. The Computers, Organizations, Policy, and Society research group provided an environment where these ideas grew through discussions and review. Lisa Covi, Dave McDonald, and Jonathan Allen helped me refine the ideas presented here.

References

Ackerman, M. (1994) "Augmenting the Organizational Memory: A Field Study of Answer Garden" In R. Furuta & C. Neuwirth (Ed.), *Proceedings of Computer Supported Cooperative Work 1994*, (243-252). Chapel Hill, North Carolina: ACM Press.

Babich, W. A. (1986) *Software Configuration Management - Coordination for Team Productivity*. New York, New York: Addison-Wesley.

Bendifallah, S. & Scacchi, W. (1987) "Understanding Software Maintenance Work" *IEEE Transactions on Software Engineering*, 13(3), 311-323.

Bersoff, E. H., Henderson, V. D., & Siegel, S. G. (1979) *Principles of Software Configuration Management*. Englewood Cliffs, N.J.: Prentice-Hall.

Bowers, J. (1994) "The Work to Make a Network Work: Studying CSCW in Action" In R. Furuta & C. Neuwirth (Ed.), *Proceedings of Computer Supported Cooperative Work 1994*, (287-298). Chapel Hill, North Carolina: ACM Press.

Brooks Jr, F. P. (1974) "The Mythical Man-Month" *Datamation* (December).

Caballero, C. (1994) "Life Cycle: Now the Focus in UNIX CM Market" *Application Development Trends* (August), 49-54,64,86.

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

Fischer, G., Grudin, J., Lemke, A., McCall, R., Ostwald, J., Reeves, B., & Shipman, F. (1992) "Supporting Indirect Collaborative Design with Integrated Knowledge-Based Design Environments" *Human-Computer Interaction*, 7(3), 281-314.

Garlan, D., & Perry, D. (1994) "Software Architecture: Practice, Potential, and Pitfalls" In *Proceedings of 16th International Conference on Software Engineering*. IEEE CS Press, Los Alamitos, CA.

Gerson, E. M., & Star, S. L. (1986) "Analyzing Due Process in the Workplace" *ACM Transactions on Office Systems, 4*(3), 257-270.

Grudin, J. (1989) "Why groupware applications fail: Problems in design and evaluation" *Office: Technology and People, 4*(3), 245-264.

Grudin, J. (1994) "Computer-Supported Cooperative Work: History and Focus" *IEEE Computer, May*, 19-26.

Heath, C., & Luff, P. (1991) "Collaborative Activity and Technological Design: Task Coordination in London Underground Control Rooms" In *European Conference on Computer Supported Cooperative Work.*

Hughes, J., King, V., Rodden, T., & Andersen, H. (1994) "Moving Out from the Control Room: Ethnography in Systems Design" In R. Furuta & C. Neuwirth (Ed.), *Proceedings of Computer Supported Cooperative Work* 1994, (429-439). Chapel Hill, North Carolina: ACM Press.

Hutchins, E. (1990) "The technology of team navigation" In J. Galegher,R. E. Kraut, & C. Egido (Eds.), *Intellectual Teamwork: Social Foundations of Cooperative Work* (191-220). Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Ingram, P (1994) "The Market for CM tools" In Proceedings of Unicom Conference London: England. October, 1994

Jorgenson, D. L. (1989) *Participant Observation*. Newbury Park, CA: Sage Publications.

Lubars, M., Potts, C. and C. Richter (1993) "A Review of the State of the Practice in Requirements Modeling" in *Proceedings of the International Requirements Engineering Symposium* IEEE CS Press, Los Alamitos, CA. 2-14.

Orlikowski, W. (1991) "Integrated Information Environment or Matrix of Control? The Contradictory Implications of Information Technology" *Accounting, Management and Information Technology, 1*(1), 9-42.

Orlikowski, W. J. (1992) "Learning from Notes: Organizational Issues in Groupware Implementation" In *Proceedings of ACM CSCW'92 Conference on Computer-Supported Cooperative Work* (362-369). Toronto, Canada: ACM Press.

Pickering, J. M., & Grinter, R. E. (1995) "Software Engineering and CSCW: A Common Research Ground" In J. Coutaz & R. N. Taylor (Eds.), *Software Engineering and Human-Computer Interaction: ICSE'94 Workshop on SE-HCI Joint Research Issues* (241-250) Lecture Notes in Computer Science, Vol. 896. Springer-Verlag.

Robinson, M. (1991) "Computer Supported Co-Operative Work: Cases and Concepts" originally appeared in Proceedings of Groupware '91 reprinted in R. M. Baecker (Eds.), *Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration* (29-49). San Mateo, C.A.: Morgan Kaufmann.

Schmidt, K., & Bannon, L. (1992) "Taking CSCW Seriously: Supporting Articulation Work" *Computer Supported Cooperative Work: An International Journal*, 1(1-2), 7-40.

Sommerville, I., Rodden, T., Sawyer, P., Bentley, R., & Twidale, M. (1993) "Integrating Ethnography in the Requirements Engineering Process" In A. Finkelstein & S. Fickas (Ed.), *Requirements Engineering 1993*. San Diego, California January 4-6, 1993: IEEE Computer Society Press.

Strauss, A. (1985) "Work and the Division of Labor" *The Sociological Quarterly, 26*(1),1-19.

Strauss, A. (1988) "The Articulation of project Work: An Organizational Process" *The Sociological Quarterly, 29*(2),163-178.

Suchman, L. A. (1983) "Office Procedure as Practical Action: Models of Work and System Design" *ACM Transactions on Office Information Systems,* 1(4),320-328.

Zuboff, S. (1988) *In The Age of The Smart Machine: The Future of Work and Power*. New York, New York: Basic Books Inc.