Recomposition: Putting It All Back Together Again

Rebecca E. Grinter
Bell Labs, Lucent Technologies
263 Shuman Boulevard, 2F-309
Naperville, IL 60566 USA

beki@research.bell-labs.com
http://www.bell-labs.com/~beki

ABSTRACT
Design and development work have become increasingly interesting to CSCW researchers. This paper introduces a new perspective for examining that work: recomposition. Recomposition focuses on the activities required to coordinate the assembly of an artifact. Using examples drawn from a study of three software development organizations, I show how recomposition is a form of articulation work. I describe how that articulation work influences the product produced, and how the product itself influences the coordination required. I discuss the implications of a recomposition view for CSCW research.

Keywords
Software design studies, articulation work, recomposition.

INTRODUCTION
In the last few years there has been an increasing interest in design among CSCW researchers. First, design is the activity that studies of work intend to inform [23]. As discussions about the relationship between studies of work and systems development continue, design will remain a focus of attention within the community. Second, design is a highly collaborative activity. It provides a good opportunity to study issues that are relevant to CSCW concerns.

This paper follows the tradition of examining the collaborative aspects of developing software. Development work includes design activities, because the process of shaping software never ends during development. However, this study differs from many other investigations of development work to date, that have focused on work that occurs in meetings [4, 18], or how individuals reflect on the problem [26]. It is also different from the smaller pool of studies of coding work, the development of the software itself [7]. Finally, it does not take the broader perspective of those studies that examine the overall project organization of development work [8, 12].

This paper takes a different perspective, one that looks at the process from its conclusion, a reassembled artifact. Specifically, I examine the process of designing and building software from the perspective of the coordination issues it creates when the product is put back together. I call this process of reassembling the product *recomposition.*

The paper begins with an examination of the process of decomposition, where the overall problem to be solved is divided into components. The process of recomposition is then defined. I discuss three different kinds of recomposition work, and the coordination they require after describing the methods and sites where this work was conducted. Finally, I outline the implications this perspective has for CSCW research. Specifically, considering the construction of software systems from this perspective yields two important findings for the CSCW research community. First, it grounds articulation work in the technical realities of assembling products. Second, it offers insights into the character of intergroup and interorganizational coordination.

DECOMPOSITION IMPLIES RECOMPOSITION
That software development involves coordination work is not a new observation. In 1968 North Atlantic Treaty Organization (NATO) convened a meeting in Garmisch, Germany that drew researchers from industry and academia. At this meeting the name software engineering was picked as the title for the emerging research discipline investigating the complexities of software development [19]. Already, the topic of coordinating the development of software was being discussed as one of their challenges.

Around the same time, Conway was writing an explanation of why this was the case [11]. Conway's thesis — subsequently this has become known as Conway's Law — was that the structure of the code mirrored the structure of the committee that had designed it.

By the mid-70's, the coordination involved in software development work was becoming increasingly important to researchers. Brooks observed that adding more people to a software development project does not make the process happen quicker [3]. Specifically, he argued that the coordination overhead of adding new people would slow down the project.

Like Brooks, David Parnas also recognized the importance of coordination in software development. Like Conway, Parnas recognized a relationship between the people working on the software and the product built. His attention was focused on the process of modular decomposition.

CSCW 98 Seattle Washington USA

Modular decomposition is the process of dividing the software problem into smaller tractable pieces to work on. Once those pieces have been defined, and the relationships among the components specified, the work of development commences. What was not well understood were the criteria to guide the decomposition itself. Parnas recognized that this process was not only a technical division of the product, but a division of labor among individuals [21]. Specifically, the smaller pieces typically would be assigned to the different individuals on the project, who would then know how their code fitted into the bigger software problem being addressed.

Parnas argued that the advantages of doing a good modular decomposition of a software problem were more than simply technical they were also managerial. Specifically he thought that good modular decomposition could reduce the communication required among different developers by eliminating some of their relationships with others.

Despite the widespread use of modular decomposition, recent studies of software developers suggest that they still spend over 50% of their time communicating with others [22]. Clearly, relationships exist among developers that need to be articulated. There are at least three reasons why these relations may exist. First, some relationships are defined at decomposition time. Second, legacy code complicates the decomposition process. The initial decomposition of new problems onto an existing base of code is analogous to retrofitting or extending an existing house rather than building a new residence. Some relationships may be maintained simply because it is easier in terms of putting new features on to the existing code base. Third, it is well known that requirements change during the development life cycle, and so as features are deleted, added and modified, those relationships are all subject to change.

This last reason is particularly significant when trying to understand how relationships in code affect coordination. Specifically, understanding the relationships among developers and the code that they produce in the decomposition phase provides only a limited insight into the social relationships that evolve and disappear during development. Instead, I chose the more unusual perspective of examining these relationships from the perspective of putting the system back together. I call the aggregate of activities seen from this perspective *recomposition.* This builds on the work of Tellioglu and Wagner [29], who observe that the relations among developers play a critical role in how products are assembled. It also follows in the spirit of Button and Harper [6] who argue that studies of work should go beyond theoretical and abstract notions of work, and examined the lived work of those studied.

Software recomposition occurs whenever the pieces of code are put together. Obviously the software is recomposed once to be released to the customer. However, the software needs to be recomposed many times before that, for testing purposes. Therefore, recomposition occurs far more times than the initial decomposition, and it is because it happens over and over again, I call it recomposition. It includes basic configuration management practices, release management work, and other integration work. The recomposition perspective focuses on the work that it takes to reassemble the product from its pieces.

It is at the point of recomposition when the relationships among the different system components become critical. The technical relationships among modules are known as dependencies, and they can take a number of forms. For example, if module A passes a variable to module B, then module B depends on A to give it that information. Another kind of dependency occurs if module A contains information that B needs in order to compile. Then A must be compiled before B. One of the hardest kinds of dependency to find is when module A generates a certain behavior during program execution which module B needs to function. This last case is known as a run-time dependency, and is hard to see because there may be scant evidence of this relationship in the written code itself.

Dependencies become interesting from a sociological perspective because of the potential division of labor among the components. When two modules are owned by different developers and share a dependency relationship then the individuals find themselves needing to maintain enough communication about the state of their work with the other person. If the communication fails the result is often that the modules will not work together — either at all or as intended. It is at the point of recomposition when developers are focused on making their code work, not only with other pieces they wrote, but with yet more components written by others' that these relationships become the acts by which software is recomposed. The strength of those relationships also becomes the ability of the developers to put all the pieces back together and make them work. In other words it is at the point of recomposition when a dependency among modules must be understood by all the developers that own the pieces involved.

The next section introduces the sites and methods used in this study. Then I describe three kinds of recomposition work that I discovered formed a routine part of development: module, subsystems, and institutional. Throughout the discussion I want to draw attention to the intimate link between the social and technical aspects of development that the developers find themselves managing. It is more than just grappling with an organizational context of design work but that social relationships are embedded into the software itself, and furthermore, can be driven and dictated by the technology.

SITES AND METHODS

I used two criteria for selecting sites for this study. First, and foremost I wanted the organization to be successful. I defined success simply as the ability to produce working software and sell it to their customers. Second, they also needed to be willing to host a researcher, sometimes for extended periods of time.

Tool Corp. is the vendor of a configuration management (CM) tool. At the time they were finishing a new release of the product as well as an upgrade to an existing version.

Both products contained code from prior systems. During my time there Tool Corp. grew from 14 to 18 developers and their product consisted of 1 million lines of code (LOC). I spent three months on site and conducted over 100 interviews there.

Computer Corp. is a large computer company that produces a real-time operating environment. They employ 700 developers who are distributed across a number of sites in the US and other continents. Their product suite consists of around 10 million LOC. I conducted 14 interviews with developers and spent three days on site.

Flow Corp. is the vendor of a suite of office technologies. The company employs over 1000 people spread out across different sites around the world. As a result of their product diversity it is hard to estimate the size of their code base. I conducted 6 interviews with one group working on one product within the suite.

Methods...

A qualitative approach was used to gather and analyze the data. In particular I used grounded theory, a method of developing substantive formal theories from qualitative data [14]. Grounded theory calls for a continual interaction between data gathering and analysis. As the researcher gathers new data they analyze it to see how it extends, modifies, or contradicts the existing theory being constructed. The result of grounded theory is an explanation of a set of practices.

Data were collected using observation and interviewing techniques. At all three sites I conducted non-participant observation, by watching people at work and maintaining diaries. I also used participant observation data gathering strategies at Tool Corp. and Computer Corp. Participant observation involves working in the environment being studied. The researcher learns — by doing — about the participants' work.

Two interview protocols, unstructured and semi-structured, were employed at the three sites. Unstructured interviews have little if any interview guide and present opportunities to gather information about the concerns of the participants. These were used extensively at the beginning of the study of Tool Corp. and towards the end to fill in any gaps in the theory.

Semi-structured interviews make use of an interview guide. The guide consists of a number of questions to ask the interviewee. The contents of the guide changed over the course of the study as the theory developed; however, I was careful to revisit old questions every so often to insure that the answers remained consistent among different developers and organizations.

Data analysis consists of three stages known as open coding, axial coding, and selective coding. Open coding consists of reducing raw data to categories that explain the behaviors reported. A category is a classification drawn from the field notes that describes an event. These categories have properties that can vary along dimensions. An example would be the category of pain, that can vary on the dimension of how intense the pain was. Axial coding involves developing these categories further, finding the conditions that lead to their emergence, and the consequences of their occurrence. Selective coding involves picking one category as the core category that forms the center of the theory. The three coding steps take individual cases of action reported and generate a formal theory that can be tested and revised in other settings. The value added in this approach is distilling what the people in the setting know and presenting it in an organized way.

...and Practice

The data were collected from the three sites using the techniques described above. Initially my interests focused on the coordination involved in the production of software. This led very naturally to the recomposition work. Specifically, I began examining configuration management practices in place.

Initial interviews and observations focused on the formal procedures and tools that developers used to perform configuration management activities. I have written about these practices and tool support elsewhere [15, 16]. However, the interviews soon broadened to encompass other activities described by developers: talking with people, the use of meetings, and diagrams. My interests also broadened to the sum of all the activities involved in reassembling software, from configuration management to larger integration efforts, and final release management techniques. This usually meant re-interviewing the initial participants in the study to find out whether they were using informal methods of coordinating work, and whether they were involved in integration and release management work. The advantage of qualitative interviewing is that it allows for this on-going revision of the study during the data gathering period.

The process of generalizing about how developers manage dependencies in their work involves making a shift from what Strauss calls substantive to formal theory [28]. This was accomplished in two ways. First, I used and refined the concepts generated at Tool Corp. in the organizations I subsequently visited. The discoveries at Tool Corp. helped to shape and refine the interviews and the observations that I made in the other organizations. Second, in each subsequent organization I revisited a set of core questions asked at the earlier organizations. This combination of refining parts of the interviews while holding other questions constant allowed me to extend the theory through the collection of new facts and data, while insuring the reliability of previous data collected. Thus, the process of generalization consists not of counting instances of occurrence, but through systematic collection of data that elaborates on the patterns of behavior found in multiple organizations, both extending and refining previous understandings. Finally, I took the results of my fieldwork and presented them to some of the developers I had interviewed and asked them for comments.

MODULE RECOMPOSITION

The division of labor in software development work usually means that small teams of developers — often collocated — work on related parts of the system. Within

large subsystems, this division is further applied until small groups have established responsibility for a small component. Recomposition work at this level involves assembling modules into pieces of the subsystem. In this section I describe three kinds of recomposition work: merging, single module over time, and change sets.

Merging

When two or more developers work on the same piece of code simultaneously it is known as parallel development. When this happens the first stage of recomposition involves making a unified module out of the versions that exist. Tools exist that automatically merge code; however, in many cases the developers find that they need to get together with their colleagues and manually merge the versions.

Tools fail to support merging well as the complexity of what has to be merged increases [15]. Most tools manage to merge versions automatically if the components do not depend on each other in any way. For example, one person changed the comments and another altered a fragment of the code in an unrelated part of the module. However, some changes can not be handled automatically because they involve unraveling and sorting out dependencies between the multiple versions of the module.

Some developers related their merging difficulties with the process of decomposition itself. As one developer put it:

> Perhaps the module itself needs to be broken up ... usually this set of functionality belongs to me, other people working on the project are working on a different functionality in the same module, therefore the modules doing too much.

The complexity of merging increases when the developers have simultaneously altered the same lines of code or algorithm to address different problems. There are now technical dependencies among the different changes that need to be understood and combined to produce one aggregated version of the module. Consequently, developers have to find everyone else, discuss what changes they each worked on, and how they altered the code. They work together to develop a shared understanding of all the versions, 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.

Parallel development involves multiple developers working on the same module at the same time. The developers depend on the same module for the work that they need to be done, and they end up depending on each other. Merging is the resolution of the technical dependencies that exist between the versions of the module. It also involves managing the social dependency among the developers working on that module.

Single Module Over Time

Merging focuses on the relations among multiple versions of a module that coexist at the same point in time. Another set of dependencies exists among versions of the same module over time. At all the sites I noticed that developers often examined the previous versions of a module before making decisions about how to proceed.

Developers described this behavior as necessary as part of the process of designing the best solution for their current problem. Previous versions of a module represent an evolution of implementation decisions and a set of technical constraints and opportunities for future work. One developer justified why he went back to previous versions of code as seeing:

> why this particular change was made um how it does or doesn't affect this next set of changes or you understand why this person made this change and hadn't considered this other problem and that sort of thing.

The developer is clear about the technical implications of changing the module without considering what has happened previously. However, he also identifies another person, the individual who made those changes in the past.

Dependencies with code in the past involve individuals who previously worked on that code. These kinds of relationships can get very complicated in established organizations. As the software and organization grow older, more employees move among projects. Other developers leave the company, and new people arrive to take their place. Consequently, many developers become responsible for code they did not originally design and the person who did is not available to explain how it evolved. The new owners are left trying to understand the character of the software from the code itself. New developers described situations where they needed to learn about what problems previous developers had been trying to fix and the reasons behind the choice of solution.

Some configuration management tools provide information that forms a kind of organizational memory, giving developers some clues about who changed code, and why. The more readily this information is available the more likely developers are to use it in their work, because without it tracking down documentation or other people who remember the person and their development work is a time consuming and sometimes inconclusive activity.

When developers do choose to ignore the past they are not in any doubt of the potential problems that it could cause:

> If you have to go through a lot of steps to figure out the history and do comparisons and things like that you tend not do that. You know cause I don't have time to do this, I'll just kind of blast ahead and cross my fingers and hope that I haven't screwed up.

Historical dependencies arise because the approach to changing the present version of a module may depend on how a developer changed that module in the past. This becomes challenging when the module has changed ownership, and one developer needs to understand the work of another. When the previous developer can not be contacted, then the current developer depends on information stored inside tools, organizational repositories and people who remember that development time.

Change Sets

The previous two cases have focused on relationships that exist between versions of the same module. Another relationship developers manage derives from the fact that their work involves changing the behavior of the system. One logical change — a request for a new, enhanced or corrected feature — often results in more than one module being altered. Making the logical change involves aligning all the physical changes to the components so that the desired outcome is produced. The modules involved in the logical change are known as a change set. They depend on each other to produce the feature requested. Furthermore, usually the change set must be introduced into the system when all the physical changes have been made otherwise any system build will probably fail.

A consequence of change sets is that the developers responsible for making the necessary alterations to their code become involved in a relationship with the others working on the same logical change. Synchronizing all the alterations and putting the updated versions of the change set back into the main line development requires coordination. At all the organizations I studied coordinating change sets was a combination of manual and automated procedures.

Developers at all the sites had access to some systems —often problem tracking facilities — that supported some of this change set work. The developers would usually begin by identifying who they were working with on the change set, which involved extrapolating back from the physical change to the statement of the logical problem to be fixed. Typically the systems provided some clues about who was working on related parts of the problem.

Then the developers would get together and discuss how they were going to implement the change as a group. Finally, one person would take responsibility for tracking how all the physical changes were progressing, with the view to informing the person responsible for the system build when that change set was ready to be compiled.

This sequence of events is managed more or less formally depending on the size and span of the change to be made. A small change involving two or three developers from the same team was usually handled informally with the developers organizing themselves. When a change spans a number of subsystems and involves more developers then some levels of management may formally orchestrate the process to ensure that the work remains aligned.

Change sets begin with the dependencies among modules as they are simultaneously altered to meet new system requirements. The technical component of the dependency involves ensuring that all the altered code functions together, produces the desired behavior, and works with the rest of the system. At the same time, the individuals responsible for the physical changes, and sometimes their management, need to form and maintain relationships to ensure that everyone's efforts are synchronized.

SUBSYSTEM RECOMPOSITION

The previous section described the coordination required to align a few modules together. This section focuses on the work necessary to assemble collections of modules into parts of the product — subsystems — and the entire system. This kind of recomposition work also involves modules of code, but because it involves increasingly more parts of the system so the processes required by an organization to coordinate and accomplish this work become increasingly formalized and complex. I begin by examining the complexities of recomposing a subsystem: the build. Then I discuss managing multiple builds of a single subsystem, and finally describe the processes behind assembling the entire system.

Build

Groups of developers working on a subsystem often want to test their changes — before submitting them to an official test group — against a working version of the subsystem. A build is process for pulling together all the modules and components that make up the subsystem. The build gives developers something to test their next changes against, and provides them with feedback on the changes that they made that are inside the build.

I have described the advantages of using a tool to automate this process elsewhere [16]. Instead, I want to describe how the technical dependencies among the pieces of code form social relationships that have to be maintained, in varying degrees, by the person responsible for the build. This person is known as the build manager.

The three organizations varied considerably on their automated support for build management. Among the organizations where there were manual procedures in place the role of the build manager involved maintaining important relationships that allowed them to build the code. Specifically, the build manager needs to know about at least three things.

First, they must be aware of any change sets in development. A person responsible for managing a change set needs to coordinate with the build manager to let them know when to put all the revisions of a module into the build. Second, the build manager also needs to know about any compile-time dependencies that exist among pieces of code. Compile-time dependencies specify that one module must be processed before another because it provides information to the latter one. Third, they need to understand the dependencies that exist among the components in the subsystem as they were defined at decomposition and have evolved. By this I mean that they need to know how the subsystem produces its functionality and what kinds of run-time dependencies exist among the code within the system. This piece of information becomes particularly useful when the build breaks and they need to identify the source of the problem.

For these reasons build managers try to have close communications with all the members of the development team. Depending on the degree of tool support for the build process, build managers interact with their colleagues to find out what code the developers are working on, what impacts that has on the subsystem, and where the components reside. The work of the build manager involves maintaining up to date information about all the

technical dependencies in a subsystem; consequently, they find themselves sustaining relationships with everyone involved in the development effort.

The degree of formalization of the build manager role varied among the three organizations. At Tool Corp. and in the part of Flow Corp. I studied, the role of build manager was an additional responsibility on top of one developer's work. The build managers at these sites were responsible for doing build work within fairly small teams of developers. Computer Corp. was a much bigger organization, and some of the subsystems were big enough that they had full-time staff that would perform the build function for their systems. Computer Corp. also had some teams that were small enough to provide their own build function internally, in much the same way that Tool Corp. and Flow Corp. did.

Platforms Creating Multiple Life Cycles

The need to compete for market share creates another dependency. It is only applicable to those development efforts that span multiple hardware and software platforms. It is a dependency when code is shared and developed for multiple platforms simultaneously.

When the companies I studied made a decision to offer their product on a new platform they did not double the development effort. Often parts of the code remain the same across platforms and new platform-specific pieces are created in the development environment to accommodate the differences. This approach means that the system now depends on the platform-specific components to provide the functionality of the product to the new platform. At all the companies I studied, some developers become responsible for working on the platform-specific pieces of code in the system.

Typically with a new development effort, some initial work needs to be done before anything is ready to compile. One solution the developers used was to copy a version of the system into their workspace and develop the variants from that. As changes carried on to the main development effort, the copied code became increasingly misaligned to the main development effort. So, at some point the new platform variants need to join the main development effort and be realigned.

The technical dilemma faced is that if they are introduced too early then they are unstable and under tested and as a consequence likely to break the build. However, if the developers wait too long the main system will have changed so significantly that the build may break due to the fact that the platform variant modules assume a set of behaviors that the system no longer produces. One solution developers used was to recopy the system to their private workspace with increasing frequency as they approached the point when they wanted to put their platform-specific changes into the central development effort. This prevented the gross misalignment from occurring towards the latter stages of platform variant development.

By reusing as much of the system as possible, the developers created a dependency between their platform specific code and the rest of the modules. They had also created an important relationship that they needed to maintain, one that allowed them to time when their pieces would go into the system. The relationships would need to provide them with enough information to plan how to prepare to fit their work into the development effort. This coordination requires working with people who are familiar with the current state of the system and can suggest when to put certain pieces back into the system.

Who these individuals happen to be is highly contingent on the specific arrangement of work within the development organization. I found that the developers sought out people who had detailed knowledge of the current state of the system development. In some cases that was the person responsible for managing the integration of the code. In other cases, developers turned to managers for help finding out when introducing their code would not do too much harm to others. The common feature here is that developers who create relationships between platform-specific components and the main system find themselves needing to maintain strong ties to the rest of development or risk mistiming the introduction of their work into the central effort.

Assembling the Entire System

The product also needs to be integrated as a whole, which involves making sure all the subsystems work together. At Tool Corp. assembling the entire system was equivalent to assembling two groups' work, as the organization was small. The coordination work involved in assembling the system was much greater for larger organizations such as Computer Corp.

At the sites studied, systems assembly work consisted of two stages. First, the individual teams needed to align their work with other groups with whom they shared dependencies. Once that was done the entire system needed to be assembled. At the time of the study, neither organization had any tools that automated the entire function. Instead, the companies used a combination of technical and organizational solutions to making the product work as a whole.

Tool Corp. was able to align the two subsystems it had fairly easily. Experts for each team were well known to all the developers there so they could check individual problems throughout development. The build managers for each system also worked closely to maintain alignment.

Computer Corp. used two strategies to manage the alignment issues. Some pieces of the system are more central than others. Changes to those pieces of code had impacts across the organization because many subsystems depended on that code behaving a certain way. Computer Corp. had established a repository and an organizational defined unit — the shared resources group — to manage the problems of commonly used code. The shared resources group had the responsibility of maintaining the code in a publicly accessible repository.

The shared resources group also standardized ways of accessing the common code. They categorized the code by what part of the system it came from, and then informed all

the teams where the code could be found inside the repository. This process formalized the ways that other teams accessed the code by removing the individual communication required and replacing it with a formal procedure. It also helped to prevent the problems of having multiple versions of a single piece of common code being used by different teams in their development work.

However, not all code was managed by the shared resources group. In the absence of the shared resources group taking care of their code, teams had to find out who was using their code. One mechanism some developers used was to broadcast information about their systems electronically, in the hope that people who were dependent on their code would reply. The weakness of this approach was that the dependency relationships were identified by the individuals who owned each component. When the developer who owned the remote code left or moved on to a new project the connection was lost. If the relationship spanned large sections of the organization the developer who needed to reestablish contact with the other side of their dependency would likely not know anyone to ask.

Computer Corp. was in the process of implementing technology that would sit on top of the CM tools and store versions of subsystems that other groups could access. Part of the functionality of the system was to depersonalize the code, remove individual's names from the modules that they worked on, and replace it with the team name. In this case depersonalization was an effort to make the subsystems more recognizable by assigning ownership at a higher level of abstraction.

At Computer Corp. the actual act of systems assembly was handled by another separate organizational unit responsible for releases. All the groups working on the product suite would hand over a copy of their code to the release group at fixed points in the development life cycle. The release group would then test to see whether the code from different groups worked together as expected.

Assembling the entire system is a coordination intensive activity. It is the time when all the subsystems must work with each other. This drives a complex set of relationships among developers and other people responsible for elements of this process. The management of this relationship requires the construction and maintenance of complex documented procedures, organizational units, and some technologies.

INSTITUTIONAL RECOMPOSITION

The development environments I have described have appeared relatively autonomous. Most of the dependencies and relationships discussed so far have been created by decisions taken within companies. However, outside influences also create complex relationships that managers and developers have to manage as part of their work. In this section I will describe just one of those relationships brought about by open systems.

Open Systems

The dependencies created by platforms I described earlier focused on fitting development efforts in different stages of being finished together. However, platforms and other kinds of software create another set of dependencies that forced all the companies I studied to create and maintain more complex relationships with vendors of other hardware and software products. All of the companies I studied, built products that needed to work with other systems and as a consequence found themselves managing these relationships.

One way that other vendors influence the development strategies followed by these companies is through their own releases. For example when a company released a new version of its operating system, Tool Corp. found itself having to make important development decisions. Specifically, Tool Corp. had to decide whether enough potential and existing customers of their tool would upgrade to the new operating system and whether that warranted starting a new development effort to make their own product compatible.

Sometimes the companies I studied wanted a tighter integration between their products and those of other vendors. Then some code must be shared between two independent vendors. At Computer Corp. I found developers that work with other companies' code, and who find themselves dependent on how those companies choose to make their own system revisions.

I spoke to one developer who was hired into the organization particularly for his experiences with a software package that Computer Corp. wanted to integrate with their software. His own development, the integration between the two products, was routinely impacted by changes made to the vendor code, as he explained:

> basically you have to take their base and compare it to what they've given you now and see the differences there and then you have the choice of either implementing it from their new code or putting their changes into your new code.

These vendors change their code, not only in terms of functionality of individual modules, but at higher levels, in the architecture of the overall product, over time. However, software development organizations who depend on other vendors, pay a high price for that dependency. The development organization must adjust their code every time a vendor's code changes. Quite frequently, the development organizations need to ensure that they support both new and older versions of the vendor's code, allowing customers the option of upgrading or not.

Developers who have code that depends on other vendors' systems find themselves spending time reworking their own modules to accommodate new changes from other vendors. Sometimes the connections between vendors let developers interact with their counterparts in the other organization; sometimes managers meet and negotiate for information about product changes. Critically, for these developers part of the job of making their code work depends on maintaining enough of a connection to the other vendor.

IMPLICATIONS FOR CSCW RESEARCH

In the last three sections I have described specific instances of the coordination work required to produce software. Studying the coordination from the recomposition perspective has helped to identify the occasions that create coordination work, and what work individuals and organizations have to do. Although I have focused on the recomposition work necessary to assemble software systems from their parts, recomposition work occurs in other settings.

Studies that include hardware design suggest that the same issues arise there [5, 20]. For example, in his studies of engineering design, Bucciarelli notes that synthesis of the product even at early stages of design is not simple. It also seems probable that this kind of work exists in the assembly of other artifacts, such as large documents written by multiple participants. In their study of the IMF, Harper and Sellen [17] observe that professionals working on documents spend time corroborating their sections with their colleagues to make sure that they fit together. In this section, I examine what the recomposition perspective has to offer for our understandings of coordination work and organizational CSCW.

Recomposition as Articulation Work

Recomposition work is a specific form of the broader concept of articulation work. Articulation work is all the coordination and negotiation necessary to get work done [27]. Recomposition work is the coordination required to assemble artifacts from their parts. Schmidt and Bannon [25] have argued that articulation work focuses on work practices that reach beyond small group settings and provides a useful conceptual tool for broadening the scope of CSCW. Recomposition work includes local work practices, but must also span the entire organization, and even brings different companies together.

This study has illuminated four features of this kind of articulation work. First, dependencies among the pieces — in this case system components — create a host of complex social relationships that need to be articulated and coordinated over long periods of time. Furthermore the strength of these relationships influences the organizations' ability to assemble its artifact. In other words, if these relationships can not be maintained then it is likely that the artifact will not fit together, as pieces will drift away and become increasingly misaligned.

Second, these relationships change over time as the product changes its technical character and the organization changes its social makeup. Knowing what they were when the artifact was originally designed is not enough, because the requirements change, the people change, and so the product is transformed. People need to revise their understandings of where the artifact is headed continually in order to understand what it has become.

Third, any existing artifacts — in this case code — creates even more complications. It creates difficulties for the initial decomposition effort in ways that potentially create more dependencies among various pieces than desired. For example, the choice not to create an entirely new and separate product for each platform means that all new development must fit with all the existing work going on.

Fourth, these dependencies are not entirely under the control of the organization. If another organization — a company, a federal agency, or regulatory body — makes revisions that impact the organization's work, the organization must realign their efforts again. In other words, the recomposition perspective situates the production of artifacts in a broader institutional context.

In summary, this study ties the coordination work required to produce objects to the artifact that is developed. Specifically, recomposition grounds one type of articulation work in the technical details of what is produced. This study demonstrates the importance of articulation work, the variety of types that occur simultaneously during the production of an artifact, and clarifies the character that these relations take during the process.

Button and Sharrock [9] observe that a relationship exists between technology and an organization. In this case, it is a relationship between the technology under production and the organization shaping it. Recomposition work provides a mechanism for understanding how coordination influences the outcome of any artifact produced, and in turn how that artifact directs the coordination required. Furthermore, it suggests that there is a three way relationship: among the artifact — whether it be a technology or not — produced, the organization building it, and any technology introduced to facilitate the development process.

Organizational CSCW

Recently, CSCW researchers have begun to explore intergroup and interorganizational forms of coordination [2]. This study shows that many of the technical solutions that support recomposition work exist for small groups trying to coordinate their development efforts locally. The amount of technical support decreases when multiple teams or organizations are involved. At the same time, and not unsurprisingly, the number of organizational "fixes" increases. The establishment of departments, steering committees, and other policies, all step in to address the coordination needs that their current technologies do not [10, 24].

This study has outlined some of the ways that organizations coordinate their development efforts. However, the relationship between the artifact and organization that I have identified also offers opportunities for examining potential organizational and technological solutions for organizational CSCW. In the rest of this section I discuss two occasions when recomposition provides insight into organizational CSCW issues.

Loosely Coupled Work and Architectures

Olson and Teasley [20] describe one organizational solution to the challenges of aligning geographically distant design work. The solution that two teams — one in the USA and the other in France — took involved slowly uncoupling their work over time. Initially they had an architecture that required them to work closely. As the project evolved it

became easier to redivide the work in ways that reduced the coordination required. They call this kind of work loosely coupled.

This study suggests that when work is redivided to accommodate distance then a corresponding change in the architecture of the artifact built must also occur (which Olson and Teasley suggest happened in their study). Work can still proceed in a loosely coupled fashion without changing the architecture of the artifact, but serious difficulties will arise. Although it may save time initially, allowing different groups to proceed more independently, if they have to reassemble their product and the dependencies have gotten misaligned, it will take longer, and require reestablishing the lost connections, and forgotten context.

This suggests that the architecture constrains the ability of developers to choose tight or loose coupling independently of the artifact. So, the artifact itself can provide a source of information about what kinds of work can be distant and uncoupled, and those which if loosely coupled create recomposition problems. This information could be used to determine opportunities for groupware, and other settings where those same technologies would be less helpful. Furthermore, by comparing the architecture with charts that identify whether work is loose or tightly coupled we can find places where work has become loosely coordinated when it should not be. Finally, the recomposition perspective provides insight into the specific activities that developers will find hard if the work has been loosely coupled while the architecture demands tight coordination.

Organizational Awareness

In their study of design work Bellotti and Bly [1] describe how individuals watched and talked with their colleagues as a way of maintaining enough awareness of what others were doing. This study confirms that developers engage in the same practices to maintain this information about what their team mates are working on. In these activities, developers associate information about other software components with the individuals building them.

Much of the awareness literature to date makes this tie between individuals and artifacts. The three types of awareness mechanisms Dourish and Bellotti [13] describe all connect information about work states with an individual responsible. Informational awareness allows individuals to describe what they did and role restrictive awareness lets other deduce what work has happened based on the role of the individual. Their shared feedback approach provides awareness of others within a commonly shared electronic workspace. Changes to an artifact are associated with the individual that made them in all these approaches.

This study suggests that awareness at an organizational level is subtly different. Awareness — what people need to be aware of — changes when organizations attempt to assemble artifacts that span many divisions of the company. In the organizations I studied, people involved in subsystems and systems recomposition work found it more useful to have components associated with the team that produced it. In fact, developers at all of the companies I studied often referred to subsystems by the name of the team that built them. Often the subsystem name was the team name (i.e., the kernel group built the kernel). For systems development this means providing ways to aggregate and display awareness information about teams. Furthermore, it raises questions about how to represent abstractions of team activity.

The notion of awareness changes radically when companies create alliances with other vendors. Companies are interested in making their products compatible, but not giving away proprietary information. What it means to be aware, to maintain enough context to align products, becomes the discussion of management meetings and the content of legal documents.

CONCLUSIONS

The predominant way of thinking about design and development work focuses on the work to break the problem into smaller components. This view permeates the literature about design as well as the techniques for organizing project activities. Systems assembly seems to play a minor role in this view.

Reversing our thinking from decomposition to recomposition places systems assembly at the center of the development effort. For CSCW researchers it highlights how critical and just how difficult individuals find maintaining congruency among all the parts. Furthermore, it establishes a foundation for explaining exactly how coordination work directly impacts the artifact produced, and in turn how that artifact affects the coordination work required. The recomposition perspective also situates this coordination in the local and global contexts of production. Finally, it offers some insights into technological and social solutions for organizational CSCW. Recomposition provides a new tool for CSCW researchers to explore the coordination involved in the production of large collaborative artifacts.

ACKNOWLEDGMENTS

This study is a product of the collective wisdom of those who manage these dependencies in their daily work. Mark Ackerman, Lisa Covi, Jonathan Grudin, Jim Herbsleb, John L. King, Tom Rodden, and Larry Votta provided support and helped me to articulate these ideas. I thank the reviewers for their comments. None of this would have been possible without the financial support of the Engineering and Science Research Council of the United Kingdom.

REFERENCES

1. Bellotti, V. and S. Bly. Walking Away from the Desktop Computer: Distributed Collaboration and Mobility in a Product Design Team. In *Proceedings of ACM Conference on Computer Supported Cooperative Work CSCW '96* (Cambridge, MA, November 1996), ACM Press, 209-218.
2. Bowers, J., G. Button, and W. Sharrock. Workflow from Within and Without: Technology and Cooperative Work on the Print Industry Shopfloor. In *Proceedings of European Conference on Computer-*

Supported Cooperative Work ECSCW '95 (Stockholm, Sweden, September 1995), Kluwer Academic Publishers, 51-66.

3. Brooks Jr., F. P. The Mythical Man-Month. *Datamation. 20*, 12 (1974), 44-52.
4. Bucciarelli, L. L. An Ethnographic Perspective on Engineering Design. *Design Studies. 9*, 3 (1988), 159-168.
5. Bucciarelli, L. L., Designing Engineers MIT Press, Cambridge, MA, 1994.
6. Button, G. and R. Harper. The Relevance of 'Work-Practice' for Design. *Computer Supported Cooperative Work: The Journal of Collaborative Computing. 4*, 4 (1996), 263-280.
7. Button, G. and W. Sharrock, The Mundane Work of Writing and Reading Computer Programs. In *Situated Order: Studies in The Social Organisation of Talk and Embodied Activities*, P. ten Have and G. Psathas, Editors. 1992, The University Press of America: Washington D.C.
8. Button, G. and W. Sharrock. Project Work: The Organisation of Collaborative Design and Development in Software Engineering. *Computer Supported Cooperative Work: The Journal of Collaborative Computing. 5*, 4 (1996), 369-386.
9. Button, G. and W. Sharrock. The Production of Order and the Order of Production. In *Proceedings of Fifth European Conference on Computer-Supported Cooperative Work ECSCW '97* (Lancaster, UK, September 1997), Kluwer Academic Publishers, 1-16.
10. Carstensen, P. H. and C. Sørensen. From the Social to the Systematic: Mechanisms Supporting Coordinating in Design. *Computer Supported Cooperative Work: The Journal of Collaborative Computing. 5*, 4 (1996), 387-413.
11. Conway, M. E. How Do Committees Invent? *Datamation. 14*, 4 (1968), 28-31.
12. Curtis, B., H. Krasner, and N. Iscoe. A Field Study of the Software Design Process for Large Systems. *Communications of the ACM. 31*, 11 (1988), 1268-1287.
13. Dourish, P. and V. Bellotti. Awareness and Coordination in Shared Workspaces. In *Proceedings of ACM Conference on Computer-Supported Cooperative Work CSCW '92* (Toronto, Canada, October 31 - November 4, 1992), ACM Press, 107-114.
14. Glaser, B. G. and A. L. Strauss, The Discovery of Grounded Theory: Strategies for Qualitative Research Aldine de Gruyter, Hawthorne, N. Y., 1967.
15. Grinter, R. E. Supporting Articulation Work Using Configuration Management Systems. *Computer Supported Cooperative Work: The Journal of Collaborative Computing. 5*, 4 (1996), 447-465.
16. Grinter, R. E. Doing Software Development: Occasions for Automation and Formalisation. In *Proceedings of Fifth European Conference on Computer-Supported Cooperative Work ECSCW '97* (Lancaster, UK, September 1997), Kluwer Academic Publishers, 173-188.
17. Harper, R. and A. Sellen. Collaborative Tools and the Practicalities of Professional Work at the International Monetary Fund. In *Proceedings of ACM Conference on Human Factors in Computing Systems CHI '95* (Denver, CO, May 1995), ACM Press, 122-129.
18. Herbsleb, J. D., *et al.* Object-Oriented Analysis and Design in Software Project Teams. *Human-Computer Interaction. 10*, 2-3 (1995), 249-292.
19. NATO Science Committee. *Working Conference on Software Engineering*. NATO Scientific Affairs Division. 1969.
20. Olson, J. S. and S. Teasley. Groupware in the Wild: Lessons Learned from a Year of Virtual Collocation. In *Proceedings of ACM Conference on Computer Supported Cooperative Work CSCW '96* (Cambridge, MA, November 1996), ACM Press, 419-427.
21. Parnas, D. L. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM. 15*, 12 (1972), 1053-1058.
22. Perry, D. E., N. A. Staudenmayer, and L. G. Votta. People, Organizations, and Process Improvement. *IEEE Software. 11*, 4 (1994), 36-45.
23. Plowman, L., Y. Rogers, and M. Ramage. What Are Workplace Studies For? In *Proceedings of Fourth European Conference on Computer-Supported Cooperative Work ECSCW '95* (Stockholm, Sweden, September 1995), Kluwer Academic Publishers, 309-324.
24. Schmidt, K. Of Maps and Scripts: The Status of Formal Constructs in Cooperative Work. In *Proceedings of International ACM SIGGROUP Conference on Supporting Group Work GROUP '97* (Phoenix, AZ, November 1997.), ACM Press, 138-147.
25. Schmidt, K. and L. Bannon. Taking CSCW Seriously: Supporting Articulation Work. *Computer Supported Cooperative Work (CSCW): An International Journal. 1*, 1-2 (1992), 7-40.
26. Schön, D. A., The Reflective Practitioner: How Professionals Think in Action Basic Books Inc., New York, NY, 1983.
27. Strauss, A. Work and the Division of Labor. *The Sociological Quarterly. 26*, 1 (1985), 1-19.
28. Strauss, A., Qualitative Analysis for Social Scientists Cambridge University Press, New York: N.Y., 1987.
29. Tellioglu, H. and I. Wagner. Negotiating Boundaries: Configuration Management in Software Development Teams. *Computer Supported Cooperative Work: The Journal of Collaborative Computing. 6*, 4 (1997), 251-274.