From Local to Global Coordination: Lessons from Software Reuse

Rebecca E. Grinter
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304 USA

grinter@parc.xerox.com
http://www.parc.xerox.com/grinter

ABSTRACT
Software reuse offers the promise of reducing product costs and increasing system reliability by making it possible to share code. However, software reuse in practice has proved much harder. This paper examines three cases of software reuse to understand why reuse remains elusive. The findings show that reuse encounters three coordination problems: the work required to traverse boundaries, the effects of organizational and environmental changes, and the coordination required to align and assemble multiple pieces of software.

Keywords
Software reuse, recomposition.

INTRODUCTION
Software reuse has long promised to reduce costs and increase reliability of development by making code sharing possible. Specifically, code stored within a reuse database allows individuals to discover previously written solutions to current development problems. In principle, reuse seems intuitive; however in practice it creates code sharing challenges.

Studies of work offer us insights into why reuse may be hard to achieve in practice. For example, studies of people sharing information show that context affects how and whether information can be shared (see for example [1, 13, 15]). However, while these and other studies offer rich insight, they have tended to focus on groups of individuals in close physical and organizational proximity.

Software reuse offers the opportunities to revisit questions about how context influences the meaning and consequently usability of information. However, many corporations have attempted large reuse programs that span multiple divisions of the organization. These larger reuse programs offer new opportunities to study how people manage to share and reuse information — in this case code and other development artifacts — across wide physical and organizational divisions. Moreover, since many corporations have attempted long-term reuse programs it also provides an opportunity to understand how an evolving corporate context affects the ability to share and reuse software.

In this paper, I present three cases of software reuse. The first case is a local effort that illustrates just how much work it takes to share information even when employees share a common work environment. The second and third cases look at broader reuse efforts and highlight how organizational and external forces make sharing information much harder. All three cases emphasize the work that individuals have to do to make code sharing possible.

I begin by describing software reuse. Then I describe the sites and methods used to collect and analyze the data used in this study. I present three cases of reuse, which range from reuse in the small to reuse in the large, and discuss their implications for CSCW research. Finally, I revisit issues of the work required to span boundaries, the impact of organizational and environmental changes on these collaborations, and how reuse efforts require significant recomposition work to become products [4].

SOFTWARE REUSE
The idea of software reuse is as old as the idea of software engineering itself. The idea of building shared libraries of components that could be used among different programs was proposed by Doug McIlroy at the first NATO on software engineering [9, 11]. The argument was simple: by reusing components developers would not spend time writing code that already existed, and the existing code would already have been tested for accuracy and completeness. From the beginning, software reuse was a compelling vision of building systems quickly and with bug-free components.

Today software reuse is an approach to developing systems where artifacts that already exist are used again [16]. The types of artifacts that could be reused have evolved from the initial NATO proposal. In addition to code, today's software reuse programs reuse requirements, analysis models, design, and test artifacts. However, the rationale behind software reuse remains the same: using existing components can help develop better, faster and cheaper software systems [9].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GROUP'01, Sept. 30-Oct. 3, 2001, Boulder, Colorado, USA.

Copyright 2001 ACM 1-58113-294-8/01/0009…$5.00.

Software reuse has three steps: abstraction, storage and recontextualization of the components to be reused. Abstraction focuses on designing a reusable artifact, often known as an asset. Software reuse advocates draw a strong distinction between designing reusable code as opposed to salvaging code from current systems [16]. Their concerns with code salvaging arise because that code was not designed to be abstract and eventually may not be reusable in all cases. While this may be true, in practice little development begins from scratch and reuse sometimes begins with existing code. Either way, the process of reuse must begin by abstracting what is to be reused.

The second concern of software reuse is making these assets available to others to use. Reuse researchers and practitioners have discovered that storing and retrieving components creates challenges. Even in simple cases, storing and retrieving assets proves difficult due to the multiple ways that software can be indexed [2]. However, when many assets are being reused, or when they are diverse — software, design diagrams, software processes, associated development tools — a simple storage mechanism is not possible.

The final challenge for software reuse is recontextualization: making what is reused understandable to those who will incorporate it into their systems. At this point, software reuse advocates have found that they need to motivate others to use reusable assets and ensure that the assets are technically compatible. Reuse of software components raises more than cognitive issues of understanding. It raises issues of incentives that managers often need to address and technical issues of compatibility [12, 16].

The challenges associated with abstraction, storage and recontextualization have led reuse advocates to a realization that software reuse requires considerable organizational effort to implement [3]. For example, Jacobson, Griss and Jonsson [9] argue that the benefits of systematic software reuse can only be realized by an organization that:

"produces related applications (or significant subsystems) that are members of a product line or product family; is willing to make a significant investment to build up reusable architectures, components, processes, and tools; and is willing to make certain process and organizational changes." [9] Page xiii.

Despite the multiple challenges of reuse programs, work continues in the area. Researchers have focused on designing technical and procedural solutions that facilitate reuse. More importantly in this context, corporations continue to advocate software reuse initiatives in order to remain competitive. This desire to reuse software is unlikely to go away in the face of increasing market pressures to release high-quality products quickly. Consequently, there are many opportunities to study reuse programs in progress. These programs, while reformulated over time as research continues to enhance our understanding of reuse, have often been in place for a considerable period of time. Individuals who are engaged in reuse often have considerable knowledge of the practical challenges of reuse and have seen difficulties unfolding over the course of several or many years.

In the rest of this paper, I describe three cases of software reuse. These three cases vary considerably in scale and scope of the reuse attempted. These examples provide good opportunities to examine the breadth and depth of reuse as it occurs in practice. I begin by describing the sites and methods used to gather and analyze data.

SITES AND STUDY METHODS

The cases reported are drawn from numerous studies of software development (see [4-6] for more details). Specifically, the data are drawn from three sites: Comms Corp, Computer Corp and Tool Corp. In this section, I briefly describe each site and how the data was gathered and analyzed.

Comms Corp. is a telecommunications equipment vendor. They develop broadband and narrowband, wireless and wired telecommunications products. The corporation's products usually contain hardware and software. The software varies in lines of code (from under 1 million to over 50 million) depending on the application. I conducted interviews and observed projects at Comms Corp.

Computer Corp. is a large computer company. They build highly reliable computer hardware and operating systems. Their suite of products contained around 10 million lines of code. I conducted 14 interviews with developers and spent three days on site.

Tool Corp. is a small development company that builds software configuration management systems. Their product contains about 1 million lines of code. There were between 14 and 18 developers working on the product. The product was a system designed to run on multiple hardware platforms and be compatible with different operating systems and database technologies. I spent three months on site and conducted over 100 interviews there.

Software reuse was not always the primary focus of these studies. However, given these corporations' interest in reusing artifacts in the software development process, it was something I encountered at each site. It was discussed during interviews with developers and managers as well as being a topic of conversation among developers.

As part of a qualitative data gathering strategy, I was able to explore how software reuse affects development. I asked questions about reuse strategies when the topic arose by using an unstructured approach to interviewing. Analysis consisted of examining the data for information about the challenges of software reuse. I began with examples of reuse drawn from data I collected. I used these examples to build an understanding of the practical difficulties involved with software reuse. Where gaps remained, I asked people involved in current reuse efforts. These interviews helped to round out my understandings of the work it takes to reuse software.

Finally, I also used other sources of data including project web sites and documentation describing procedures. These sources, along with the interview data, were especially

useful for developing an understanding of the history of reuse efforts. This history helped to ground the reuse effort in an organizational context, which often revealed insights about what factors led to the current state [10].

CASE 1: ARCHITECTS REUSING DESIGNS

One of the earliest stages in systems design focuses on describing the product architecture. This involves defining functionality, describing behaviors, and producing high-level and low-level specifications that detail the hardware and software solution. Comms Corp calls the people who do this work architects. In this section, I describe the architects' reuse practices.

The architects have the responsibility for producing the initial designs for either new products or enhancements to existing systems (see [6] for a detailed description). One aspect of their work involves making presentations of their current design ideas to various groups within the corporation. Specifically, the architects present their current solutions to different constituencies within the corporation including developers, funding bodies, standards groups, and management. The purpose of these presentations is to secure continued support for their proposed solution. These presentations often include people attending via telephone or videoconference given the size and distribution of Comms Corp.

In addition to making presentations to other corporate constituencies, the architects present to colleagues working on the project. Architects rarely work alone because the products they build require expertise that spans a range of disciplines and skills. Typically, they need help from people with knowledge of hardware, software, standards and marketing. Since this knowledge is spread around the corporation, the architects often find that they work in distributed teams that include people from Asia, Europe and North America.

An important implication of making presentations and working in distributed teams has been the use of the World Wide Web (WWW) and PowerPoint™. The architects were among the first people to experiment with the WWW and find a use for it within the company. They adopted it as part of their general interest in new technologies. However, they began to use it as a mechanism for communicating with the people in their project teams. They use the web to display PowerPoint slides during meetings for physically remote team members attending via telephone. They also developed web sites for their projects that included copies of their slides, meeting notes and other documentation on those sites. Over time, these practices standardized the use of WWW and PowerPoint among the architects.

The use of the WWW and PowerPoint fostered another practice**:** that of borrowing diagrams to reuse in other presentations. They adopted the practice of borrowing from other presentations for several reasons. First, PowerPoint makes borrowing diagrams very simple. Copy and pasting others' diagrams into their own is easy in comparison with generating the pictures from scratch.

Second, if the architects are working on an extension or enhancement to an existing product line, it often makes sense for them to reuse drawings and add their own pieces as necessary. However, even when the architects are working on entirely new products, they find that they can reuse others' diagrams. One reason is that new products usually need to interoperate with other existing systems. The architects take existing network configurations and add their piece in to show where their solution fits.

Third, when they architects work on new products they may reuse prior designs. The ability to reuse diagrams in new products arises because projects can be terminated after design for reasons including prohibitively high technology costs or lack of market demand. As technologies and economic forces change, something that was not feasible before becomes attractive to redesign and implement. In these cases, the architects working on the new version can partially reuse the previous materials.

Typically some rework needs to be done to incorporate new technologies and standards. However, an underlying design may still be partially reusable. Since the architects do not typically delete cancelled projects' web sites, many repositories exist for potential reuse. The incentive to reuse previous work comes from the time saving made by not having to rework the architecture both in terms of exploring the design space as well as the effort to draw the diagrams again.

Finally, architects often act as technical representatives of the company to customers. In this role, they can be called on to give presentations to customers about different products. Reusing components of presentations ensures that a customer, who might see several presentations, has a consistent and integrated view of the company's products.

All these reasons make the practice of reusing diagrams and other artifacts extremely desirable. However, the information does not come packaged for reuse. Instead, the diagrams are contained inside presentations on the project web site. Moreover, these web sites are typically organized for the people on the project, which may not be the most useful order for someone trying to borrow diagrams. To find the diagram the person browsing the site has to have enough knowledge of the when designs were drafted and what they were called.

Often the search for web sites is made more complex by the evolutionary nature of the architects' work. Architect's projects change as new technologies come and old ones go. In order for architects to reuse others' work, they have to remember when these changes happened. For example, in addition to remembering the current name of the project, they often have to remember the previous names of a project. A project with a long life may change names several times, which is often associated with significant changes in technological and managerial direction. The architects may have to browse several different web sites, each associated with one project name, to find the right slide. The architects avoided a long search when they knew a project's history and could use that information to narrow their search. In other words, part of

recontextualizing slides is being able to locate them within this corporate history.

Luckily, these architects typically had this corporate knowledge. In addition to usually having worked for corporation for a decade, they also shared a common method of work and mechanisms for sharing project information within their group. Most of the architects were familiar with their colleagues' projects. Typically, they knew the various names and the corporate history as well as having a technical overview. The architects also knew the areas of the Intranet where their colleagues put project information. Using their knowledge of their colleagues' work and where that work resides on the network, they usually found the materials they wanted to reuse.

The architects also share a common architecture process methodology. Most of the architects used the nomenclature of the process to describe presentations and structure their project web sites. Architects who understood the process could often use that knowledge to deduce where certain diagrams and slides resided in a project web site. For example, the architects often produced low-level and high-level specifications for their projects. Other architects, who understood the difference between the two documents, knew which would contain potentially reusable materials for their own work.

The knowledge of common work processes that allows architects to find relevant diagrams also helps them recontextualize those slides with respect to the project. In other words, the architects can use steps in the process and the resulting documentation to deduce what state the project is in.

These architects reuse each other's diagrams as part of their need to communicate within and outside the corporation. All the architects also contribute to the store of knowledge by producing materials during the course of their work. They organize these materials in project web sites that are typically organized by work process. Since they have a common process, the architects know how to navigate their colleagues' web sites looking for reusable assets. They also use their knowledge of corporate project history to locate materials related to a certain technological "era" of the project. Finally, this common knowledge allows the architects to recontextualize what they find correctly.

CASE 2: A TALE OF TWO PRODUCTS

The previous example might not be considered reuse by some definitions since it was informal and did not begin with artifacts designed for reuse. Most software companies that begin a reuse program create a formal reuse asset creation plan at the beginning of a development effort. In this section, I describe one corporation's early effort to experiment with reuse.

In this case there were two products: Alpha and Beta. Both Alpha and Beta needed to be built for two markets. The company designed Alpha so that there was a common piece that worked in both markets. In order to accommodate the differences between Alpha's two markets other groups would take the common piece and customize it for a specific market. Beta was planned identically. This plan led to both Alpha and Beta having a largely reused common core that was customized for local differences.

Alpha and Beta were selected because the market differences were minor at the time of design. The markets differed in the implementation of a single low-level protocol. From the design perspective, it was easy to isolate the variability between the two protocols and create a market specific set of components to plug into a larger common piece. However, the common parts of Alpha and Beta were not very reusable despite the initial design plan.

Efforts to reuse Alpha stopped after only five months for two reasons. First, Alpha's two markets diverged in terms of their needs for Alpha-type products because they took different technological evolution paths. To compete in both markets, Alpha had to follow both technological trajectories and incorporate all the required functions. Over time, Alpha's market-specific components grew and the common part shrank to the point where it took more resources to develop from the common base than build separately. So Alpha was divided and reuse stopped. Both separated Alpha's went on to be successful products.

The second reason exacerbated the first reason Alpha diverged. In one of the two markets, the company became involved in a partnership with another vendor of similar products. This partnership was designed to help both companies ensure success in the marketplace. However, the partnership also came with agreements for collaborative development of products. Specifically, the other vendor had product like Alpha and it was agreed that these products should be developed in concert. This led to a situation where three sets of technical requirements needed to be accommodated by the design. This dramatically reduced the commonality between the two versions of Alpha. It soon made more sense to align the two market specific versions and stop attempting across-market reuse.

The second product — Beta — lasted longer, the hardware remained common for six months and the software for six years. A difference in markets caused the hardware in the two products to diverge. Specifically, it turned out that one market would bear more hardware costs (for an associated increase in reliability) than the other would. The choice to have common hardware presented a dilemma. If they used the more expensive hardware in Beta one market would not purchase it. The other market would not buy it if they used the cheap hardware.

After six months, it was decided to separate the hardware but in a way that would still allow common software development. In this revised manner, the software development was able to last another six years until both versions of Beta entered the latter phases of development where only modifications and support would be offered. The corporation decided to offer modifications and supports for each market separately and development on common Beta ceased.

The evolution of requirements makes these two cases of software — and hardware — reuse interesting. Both efforts had been well planned in terms of what would be reused and how. However, what they did not account for was the

pressure that external forces would exert on the development of Alpha and Beta. Both Alpha and Beta suffered from changes in market demands that pulled the common pieces apart. Alpha had trouble maintaining any commonality at all in the face of changing market demands. Beta fared better because the changes in markets were isolated in the hardware and the software design remained unaffected. Alpha also suffered from another change that affected its reusability. The strategic alliance between vendors created considerable design complexity for the common part of Alpha. In the end, Alpha's designers felt that it was easier to separate the product rather than attempt to accommodate all the technical functionality required.

Alpha and Beta were designed for reuse. This design took the form of delivering an architecture that isolated the market-specific components from the common pieces for Alpha's and Beta's developers. In practice, their designs for reusing Alpha and Beta did not predict the external forces that would undermine the value of what was common.

CASE 3: REUSE IN THE LARGE

Reuse in the small consists of using small entities such as diagrams or code fragments. It often takes place inside a department or on a single project. For example, the architects who reuse pictures and text in their own work carry out reuse in the small. Alpha and Beta were formal reuse efforts but they were also small efforts since they only consumed the attention of a small division of the corporation. However, reuse has also been tried at a much broader scale, which I call *reuse in the large*.

Reuse in the large efforts often begin with a corporate initiative to support and encourage reuse across an organization.[1] Market pressure to produce high quality systems quickly often leads corporations to implement reuse in the large. These efforts typically begin by identifying products that have a common functional need. Reuse in the large programs then proceed to restructure development so their common pieces are built once and then customized for specific products. Significantly, the products that consume these reusable assets may reside in organizationally distinct divisions of the corporation. In other words, customers may be organizationally independent of each other and the reusable asset. In this section, I describe Project Shape that was part of a corporate-wide reuse program involving several reusable systems to be used throughout the organization.

Project Shape began in the early-90's as a reusable system. Specifically, Shape would provide a function that many of the corporation's systems used. Shape's initial plan was to provide a system for a few customers to ensure that the code was reusable. Today, internal customers use Shape in the products that they sell externally. The corporation views Shape as a successful example of reuse.

In this section, I will not relate the entire history of Project Shape. Instead, I report the features Shape's experience that are salient to understanding the challenges of corporate-wide reuse.

Experienced Developers

Building reusable software requires developers who have general development experience and know the corporation. While it is possible to staff a project with developers who are new to this work and the company, experienced developers bring additional skills. This was the case for Shape where experienced developers gave the project two distinct advantages.

First, the experienced developers understood the technical domain well. Shape was a subsystem within a highly reliable real-time product and it took new developers a long time to learn how to design these types of system. The experienced developers understood how to design reusable functions for Shape's customers.

Designing reusable assets requires the ability to abstract to the most generic system that remains usable to potential customers. This skill comes with experience of designing many similar systems and is hard to teach to those without much practice. Experienced developers had typically seen several implementations of the types of functions Shape required. Using this knowledge, experienced developers could decide what designs worked well and how generically reusable those implementations would be in the future.

Second, the experienced developers came from departments inside the corporation. In addition to acquiring technical domain expertise, they understood what these departments' development priorities were. Some of these departments became customers for Shape. The experienced developers knew how Shape fit into these departments' projects and what functionality Shape would have to provide to be compatible.

Sharing a Subsystem

The vision of Shape was to provide a complete subsystem that would fit into other development efforts. Obviously that required building the subsystem itself. However, giving the system to other development groups was not all Shape's staff found that they needed to do to be reusable.

Project Shape found that if they delivered just the subsystem to their customers, their customers could not reuse the components. Specifically, the subsystem alone did not fit into the customer's development environment. A development environment consists of more than just the code being written. Tools that help developers debug the system, compile the code and provide versions management surround that system. These tools constitute the development environment and help developers the ability to work with the code itself.

For example, Shape used a configuration management system to track problems, organize code, and produce builds. Configuration management systems contain data such as who worked on different components, how old code elements are and what problems remain outstanding. This information helps developers and managers understand the state of the code and monitor its evolution.

[1] See [9] for a description of reuse in the large programs at companies including HP and AT&T.

When Shape provided only the subsystem, their customers lost all that extra data about the code. Shape's staff found that the loss of configuration management data made customers less interested in using Shape. To resolve this problem, Shape staff began encouraging their customers to use the same configuration management system. Customers who used the same configuration management system got Shape code and the extra information. This helped the customers reuse Shape by helping them recontextualize Shape in their own development environments.

Another example of recontextualization happened with versions of compilers. Different compilers and even different versions of the same compiler can significantly alter a system's performance and behavior. With many projects this does not matter because as long as they always run in their home environment the code works. However, Shape developers found that when they ported their system to customer environments compiler differences could change the behavior of Shape or break it completely.

As a result of these difficulties, Shape's project management team initiated a common development environment project to support their subsystem. The environment provided customers with tools to view, compile and run Shape. The common development environment also provided information about how Shape had evolved.

Providing the environment changed the scope of Shape. Shape had changed from the original vision of being a reusable subsystem. In order to meet the challenges presented by recontextualization, Shape had become a subsystem and supporting reuse technologies in the form of a common development environment.

Providing this environment was critical to make the system work in another development organization. However, Shape management also discovered that it was important to share their technical expertise along with their system. Specifically, they found that customers had much better results reusing Shape when a technical expert accompanied the latest release. The Shape expert would spend time at the customer's site installing the subsystem and answering questions about how it operated.[2]

The original vision of Shape underestimated the work required to reuse the subsystem. Customers needed additional tools to successfully reuse Shape. Shape's project management realized that customers needed supporting tools, and they decided to make that a part of Shape. Consequently, the scope of Shape changed to include a common development environment as well as the original subsystem. Shape also sent an expert to the customer site along with the subsystem and common development environment. Shape experts helped customers recontextualize the subsystem. Supplying a common development environment and sending an expert allowed customers to reuse Shape.

Alignment Across Organizational Divisions

In the last two sections, I discussed how Shape developers used their knowledge of customers to design the subsystem. Also, I described how the staff helped customers reuse Shape by sending an expert to deploy the subsystem and environment. However, while it was important for Shape to coordinate with their customers during design and deployment, Shape needed to communicate throughout development.

Shape needed to coordinate with customers regularly because their customers' needs changed as the technologies and organization evolved. Shape staff needed constant access to the development directions of their customers so that they could predict whether Shape would be reusable. The customer's system and Shape would drift apart without regular contact.

Shape staff also needed regular contact to ensure that their product's interfaces matched their customer's systems. Interfaces among system components can present hidden problems. Interface design agreements, even those produced by tools that require detailed specifications, can have different assumptions embedded in them [7]. When interface disagreements occur among peer-projects they can negotiate a unified approach. Shape was not a peer project but a project with customers. If Shape's interfaces were incompatible with its customers' there was an expectation that Shape would amend theirs.

Circumstances can make regular communication difficult to maintain for at least two reasons. First, Shape was geographically separated from some of its customers. Learning about physically remote customers' work schedules and pressures could only be achieved by a significant amount of travel to customer sites. Moreover, in line with findings from other studies of geographically distributed development, Shape suffered from being almost invisible to its remote customers during times when visits were infrequent [8].

Second, customers had their own development priorities. Sometimes schedules created sufficient pressure that Shape's customers ignored Shape. Typically this was the same time that significant changes would be occurring to the customer's product. Shape developers realized that they needed to understand changes to their customer's system or risk not being able amend Shape. This could lead to Shape being unusable by the customer.

Shape needed to regularly coordinate with its customers but faced difficulties doing so. Infrequent communications with customers could lead to Shape becoming unusable instead of reusable. This danger was exacerbated by the fact that Shape, as a reuse project, often carried the burden of ensuring that their code fit with their customers' systems.

Economic Alignments and Reorganizations

The customer-producer relationship described above illustrates another challenge for reusable assets. Internal corporate financial arrangements affect who can afford

[2] Other studies of development show that developers often follow their code to another site to help others work with it [7].

reuse. In this section, I examine two different economic arrangements that Shape experienced.

When Shape started it was organizationally distinct from its customers. The corporation levied a corporate wide "tax" on development projects to pay for the costs of developing reusable assets. This motivated projects to attempt reuse. However, the arrangement also created an organizational distance between the reusable projects including Shape that made coordinating schedules and technical goals difficult.

The organizationally distinct reuse division did not survive the first large corporate reorganization. Shape moved into the same division as its largest customer and the economics changed. Shape went from receiving part of the global tax to seeking financing from its customers. In other words, clients paid for a percentage of Shape development. The advantage of this arrangement was that customers cared about the reusable assets they paid for. However, the danger was that a single customer would end up paying for a considerable percentage of Shape's costs. If that happens, a reuse project can become subordinated to the large customer's will making it impossible for other clients to use.

This reorganization affected Shape in several distinct ways. It made one customer so organizationally distant that the financing was too difficult to arrange. When the new economic arrangements became clear, this customer stopped using Shape and turned to an alternative solution. Other customers became organizationally closer to Shape and found that they could easily finance some part of Shape development. Finally, some completely new customers decided that they could afford to use Shape. It was a significant change for Shape, one which profoundly influenced its architecture and functionality.

Shape went through several reorganizations that moved it closer to and further away from different customers. What remained constant is the commitment of the staff of Shape to working out how to fit their subsystem into their current customers' development efforts. This involved knowing who their clients could potentially be after each reorganization and then establishing relationships with these customers.

Internal economic arrangements affected Shape and its customers. Changes in economic arrangements were typically associated with corporate reorganizations. The original vision of Shape did not account in any way for the financing of Shape since that was assumed to come from corporate tax. The first reorganization changed Shape's income revenue profoundly and the project learned that part of making software reusable was surviving and adapting to economic changes. Their adaptation relied on many of the skills that Shape's staff were become adept at such as networking and organizational awareness.

IMPLICATIONS FOR CSCW

The three cases illustrate software reuse in practice. The cases show that much collaborative work is required to make reuse work irrespective of whether the reuse is small or large.. In this section, I examine three coordination challenges for reuse: traversing boundaries, the effects of organizational and environmental changes, and the coordination required to align and assemble multiple pieces of software.

Traversing Boundaries

Modern corporations comprise many distinct divisions. The distance between divisions can be measured in at least two ways. First, we can use the first common point of management between the two divisions to see how close they are. The higher up the formal chain of management the first common manager resides the further apart the divisions are. Second, the more complex the economic arrangements need to be to transfer money among divisions the further apart they are. Reuse projects encounter both kinds of distance and need to traverse these boundaries to be successful.

This study suggests that there are at least two challenges that producers of reusable assets have to overcome when they span organizational divides. First, they have to have visions of both the production and consumption of the components. Second, they have to have institutional support of the corporation and then be willing to encourage the reuse effort as individuals. I will discuss these in turn in the remainder of this section.

Developers find it technically challenging to build reusable systems. It requires the ability to abstract to the most general case that for complex technologies often requires input from many experts. Despite this, developers still find it easier to formulate reuse as a production problem. In fact, production has to precede consumption or there is nothing to use let alone reuse. However, since the design work is so challenging it is often impossible to remember to ask how it will be reused.

Cases 1 and 3 show that the developers had to adopt a consumer perspective to make their systems reusable. In other words, one of the boundaries reuse projects must cross is from their vision of how to produce the product to an understanding of how their customers consume it. In all three cases, components were only reused when the customers could understand the purpose and value of the reusable assets. These cases also show the further apart the producer from the consumer the more difficult reuse becomes. Any work associated with crossing the divide typically rested with the producers because the customers never had to reuse the product.

The architects and Shape developers found different ways of spanning the producer-consumer divide. The architects found it relatively easy to use and produce usable materials because of their shared working context as both users and consumers of architectural design materials. The developers and managers of Project Shape had to go to considerable lengths to understand their customers and maintain that awareness. In fact, the members of the Shape project found themselves simultaneously working two problems: producing a usable Shape and understanding their customers' needs.

Understanding their customer's technical needs is only one aspect of promoting reusable systems. A related problem for reuse efforts is making customers want to reuse a

system rather than build it themselves. Building it themselves has the advantage of ensuring that the development schedule is under local control. The three cases presented highlight institutional and individual mechanisms for encouraging development projects to use reusable assets.

Institutional support involved creating incentives and rewards for those who consumed reusable assets. Cases 2 and 3 illustrate the kinds of institutional support possible. The manager who had responsibility for Alpha and Beta made reuse possible and desirable. In his position, he could see some of the common elements of the products, and consequently he supported designing a reusable core in each case. Decisions from management promoted and fueled the process of thinking about sharing reusable components.

Project Shape began as one of several efforts in a corporate-wide reuse program. As a result, Shape benefited from being widely known throughout the company as a reusable system. Having support from top management generated interest in Shape. In other words, institutional support helped create an environment where people were at least willing to try it out rather than build their own.

However, institutional support alone is probably not enough. Shape illustrates this very well. The efforts of Shape's staff at spanning organizational divides helped make their system a success. Shape staff's constant communications with customers that promoted reusing Shape were critical throughout development. For example, Shape developers built the development environment and went to their customers' sites to make sure that Shape worked despite the ever present organizational divides.

Individual support for reuse became even more critical as institutional support changed. Shape outlasted the corporation's commitment to reuse as a top priority. As those priorities shifted, Shape's developers and managers picked up the work of promoting their reusable system. I will return to the challenges created by change in the next section.

Successful reuse efforts focus on customers' needs as well as design issues. They consider what consumers will need technically, and they also attend to how to fit a system into their clients' environments. Although a corporation can encourage reuse, it is also necessary to have articulate individuals who facilitate the process by listening and understanding customers' requirements and practices.

Reuse repositories are boundary objects that require articulation to be both useable and useful. Some boundary objects may function simply by their existence. In some cases, boundary objects work better when their meaning is ambiguous and open to interpretation. For software reuse this does not seem to work. In the case of software reuse, both institutional and individual commitment to explaining the system makes it possible for clients to use it.

In other words, producers of reusable assets carry the burden of articulating their systems through verbal explanation and tool support. It is this articulation in both verbal and technical forms that allows others to recontextualize the system. When that reuse occurs locally, users — like the architects — can easily interpret the data and make sense of what they find. When organizationally distant reuse occurs, the process of recontextualization becomes more complex. It becomes a translation between producer and consumer that is facilitated by the company and the individuals who develop the assets. Further, this recontextualization can be subject to internal corporate economics and reorganizations that change those economics. To survive and be shared, a reusable system has to be valued in ways that producers can afford to build their systems and customers can afford to use them.

Recontextualization and Change

The three cases presented here suggest that both recontextualization and the preceding decontextualization are made more complex by change. In all three cases, the producers and consumers of reusable assets had to cope with changes from the organization and environment. Both kinds of changes led to significant reprioritizations of effort within the reuse projects.

Project Shape illustrates the challenges presented by corporate reorganizations. Developers on Shape found that after any reorganization that some of their customers would come and go. This resulted from several distinct problems. First, the technical direction of the customers might change to reflect the new corporate priorities. Second, the technical direction of Shape might change to reflect their new corporate circumstances. Third, the organizational distance among producers and consumers might change, which would make it easier or harder to collaborate. Finally, the reorganization might change the internal financing structures leading to reprioritizations of resources by both customers and producers. The results of any of these changes — or a combination of several of them — led to the process of alignment beginning again.

Case 2 illustrates the difficulties of coping with changes from the environment. For both pairs of products, it was diverging markets that pulled the different versions so far apart that the common pieces became smaller and less significant. Alpha's project trajectories separated because the two markets wanted different functions. Alpha also experienced difficulties in maintaining common code because of a strategic corporate alliance. Beta's hardware trajectories separated because the two markets would bear different costs. Both Alpha and Beta illustrate how evolving market demands can slowly destroy reuse efforts by eroding the common base until it is not worth the development effort.

Change from inside and outside the corporation situates the value of building and consuming reusable assets in time. To survive change a reusable asset must remain valuable. For Shape, that typically meant losing some customers and gaining others after internal reorganizations. Alpha and Beta found it harder to remain valuable as their markets changed.

Reuse: an Artful Integration literally

In a previous study, I introduced the concept of recomposition. Recomposition is all the coordination work

that developers do to assemble a working system from its components [4]. In this section, I show how reuse recomposition is an example of an *artful integration*.

Recomposition highlights how the ability of an organization to assemble its product was its ability to manage social relationships over long periods of time. An example of recomposition is all the coordination among a team of developers to ensure that their code changes work together. Another type of recomposition relationship focuses on the coordination required to assemble large software systems from small subsystems, which typically requires dedicated personnel such as those in a release group. Recomposition highlights the time and complexity of all kinds of coordination associated with constructing a working software product.

Reuse — and particularly reuse in the large — challenges recomposition because it separates development groups across the organization and simultaneously requires the same tight coupling of systems. As I have argued in previous sections, successful reuse requires system producers to coordinate with their customers. These complex relationships are recomposition.

In her opening plenary at ECSCW 99, Lucy Suchman described a central project within CSCW, which she called artful integration [14]. Artful integrations place technologies into the social and material world in ways that enable users in their work. Integration also happens to be the name of a development activity: the process of assembling components into the software system.

Reuse is traditional conceived in the integration sense. The goal of reuse is to create subsystems that can be easily integrated into a final software product. This limits an understanding of the work required to make reuse work.

Reuse is surrounded by recomposition. Recomposition helps individuals understand what it means to fit into another context. For reuse, recomposition means understanding their customers by assessing their needs, motivating them to reuse rather than build, understanding the organizational and external contexts they operate in, and constantly readjusting all of that to account for any changes. These relationships are the process of making the reusable asset "fit" into another development context. In other words, recomposition is the process that allows a reusable asset to become an artful integration in addition to a system integration.

Reuse requires a day-to-day understanding of the social and material world into which the reusable system must fit. This artful integration does not happen at the end of a long period of isolation but by maintaining on-going relationships. This type of artful integration cannot occur without recomposition because otherwise the organizational divisions and the evolution of technical and economic priorities will slowly erode the effort.

CONCLUSIONS

In this paper, I described three cases of software reuse. I showed that taking a perspective of understanding collaborative work reveals three challenges for software reuse in practice. These challenges are the need to traverse boundaries, the effect of organizational and environmental change on what can be reused, and the need for recomposition. As well as illuminating some of the difficulties with reuse, these insights add to our knowledge of collaborative work.

ACKNOWLEDGEMENTS

Mark Ackerman, Jim Coplien, Martin Griss, Jim Herbsleb, Rachel Jones, Tave Lamperez, Phil Polli, Tom Rodden, and Doug Schmidt provided me with different perspectives on reuse as well as comments on previous formulations of these ideas. Mistakes and omissions remain mine.

REFERENCES

1. Ackerman, M.S. and Halverson, C., Considering an Organization's Memory. in *ACM Conference on Computer Supported Cooperative Work (CSCW '98)*, (Seattle, Washington, 1998), ACM Press, 39-48.
2. Curtis, B. Cognitive Issues in Reusing Software Artifacts. in Biggerstaff, T.J. and Perlis, A.J. eds. *Software Reusability Volume II Applications and Experience*, ACM Press, New York, NY, 1989, 269-287.
3. Fafchamps, D. Organizational Factors and Reuse. *IEEE Software, 11* (5). 1994. 31-41.
4. Grinter, R.E., Recomposition: Putting It All Back Together Again. in *ACM Conference on Computer Supported Cooperative Work (CSCW '98)*, (Seattle, Washington, 1998), ACM Press, 393-403.
5. Grinter, R.E. Supporting Articulation Work Using Configuration Management Systems. *Computer Supported Cooperative Work: The Journal of Collaborative Computing, 5* (4). 1996. 447-465.
6. Grinter, R.E., Systems Architecture: Product Designing and Social Engineering. in *ACM Conference on Work Activities Coordination and Collaboration (WACC '99)*, (San Francisco, California, 1999), ACM Press, 11-18.
7. Herbsleb, J.D. and Grinter, R.E., Splitting the Organization and Integrating the Code: Conway's Law Revisited. in *21st International Conference on Software Engineering (ICSE 99)*, (Los Angeles, CA, 1999), ACM Press, 85-95.
8. Herbsleb, J.D., Mockus, A., Finholt, T.A. and Grinter, R.E., Distance, Dependencies and Delay in a Global Collaboration. in *ACM Conference on Computer Supported Cooperative Work (CSCW 2000)*, (Philadelpia, PA, 2000), New York, N.Y.: ACM Press, 319-328.
9. Jacobson, I., Griss, M. and Jonsson, P. *Software Reuse: Architecture, Process and Organization for Business Success*. ACM Press, New York, NY, 1997.
10. Latour, B. and Porter, C.t.b. *Aramis or the Love of Technology*. Harvard University Press, Cambridge, MA, 1996.

11. Naur, P. and Randall, B. Working Conference on Software Engineering, NATO Science Committee, 1969.

12. Schmidt, D.C. Why Software Reuse has Failed and How to Make It Work for You. *C++ Report*, *11* (1). 1999. 46-52.

13. Star, S.L. Cooperation Without Consensus in Scientific Problem Solving: Dynamics of Closure in Open Systems. in Easterbrook, S. ed. *CSCW: Cooperation or Conflict?*, Springer-Verlag, Heidelberg, Germany, 1993, 93-106.

14. Suchman, L., Artful Integration as CSCW's Project: Reflections and Reveries. in *Sixth European Conference on Computer-Supported Cooperative Work ECSCW '99*, (Copenhagen, Denmark, 1999).

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

16. Tracz, W. *Confessions of a Used Program Salesman: Institutionalizing Software Reuse*. Addison-Wesley, Reading, MA, 1995.