In Proceedings, International Conference on Software Engineering, 1999, Los Angeles, CA, pp. 85-95

Splitting the Organization and Integrating the Code: Conway's Law Revisited

James D. Herbsleb and Rebecca E. Grinter
Bell Laboratories, Lucent Technologies
263 Shuman Blvd
Naperville, Illinois 60566. USA

+1 630.713.1869
{herbsleb, beki}@research.bell-labs.com

ABSTRACT
It is widely acknowledged that coordination of large scale software development is an extremely difficult and persistent problem. Since the structure of the code mirrors the structure of the organization, one might expect that splitting the organization across time zones, cultures, and (natural) languages would make it difficult to assemble the components. This paper presents a case study of what indeed turned out to be the most difficult part of a geographically distributed software project, i.e., integration. Coordination problems were greatly exaggerated across sites, largely because of the breakdown of informal communication channels. The results imply that multi-site development can benefit to some extent from stable plans, processes, and specifications. The inherently unpredictable aspects of projects, however, require communication channels that can be invoked spontaneously, by developers, as needed. These results shed light on the problems and mechanisms underlying the coordination needs of development projects generally, be they co-located or distributed.

Keywords
Coordination, collaboration, systems integration, qualitative research.

1 INTRODUCTION

Geographically distributed software development has become a business necessity for many global corporations. This imperative is being driven by the need to locate resources in a country for marketing purposes, the acquisition of foreign divisions in mergers and buy-outs, and the promise of global round the clock development. Despite the necessity, and perhaps even desirability of geographically distributed development, it is extremely difficult to do successfully (see, e.g., [10]). Unanticipated coordination breakdowns appear frequently to lead to delay, inefficiency, and frustration.

In this paper we report research that begins building a better understanding of why geographically distributed development is so difficult to coordinate. Specifically, we present a case study of the integration phase of a geographically distributed development effort. By "integration" we mean all the work that is necessary to assemble the product from its components. We were pointed towards integration as a topic by an initial series of interviews that focused on occasions where the multi-site character of the project was most disruptive.

The next section reviews the literature on coordination in software development and the various kinds of mechanisms that serve to maintain coordination. Section 3 describes the sites involved and our empirical research methods. Our results are presented in section 4, and our conclusions in section 5.

2 COORDINATION, PREDICTION, AND COMMUNICATION

The coordination of software development work has been a focus of attention within research for a long time. One of the first people to suggest its significance was Melvin Conway [3]. Specifically, what Conway said was that the structure of the system mirrors the structure of the organization that designed it. Conway's Law — as it has become known — was the first explicit recognition that the communications patterns of an organization left an indelible mark upon the product built.

Parnas went on to clarify how the relationship between organization and product occurs within software development. His definition of a module as "a responsibility assignment rather than a subprogram" clearly shows that dividing a software system is simultaneously a division of labor [12]. It is the division of labor, among different individuals, that creates the need for them to coordinate, to align their efforts, in the production of software.

Both Parnas and Conway focused on the structure of the product, which provides an important foundation for the coordination of work. Product structure addresses the question of *what* is to be developed by individuals or small groups. This, however, is only one of several dimensions on

which development projects must be coordinated. A project must also, for example, determine *how* the product is to be developed (i.e., the development process). Processes can also determine work assignments, as when an intermediate product is handed off between two groups, e.g., for different stages of design, coding, or testing. Again, well-defined boundaries, in this case between process steps, are critical in breaking up work so that it can be assigned to different groups. In addition to what is developed, and how the work is done, *when* various project milestones are to be achieved (usually laid out in a plan), and *who* will do the work (often contained in a staffing profile) are also critical to project coordination. Projects can struggle badly or fail altogether if these other critical dimensions of project coordination are not attended to.

Around the same time that Parnas and Conway were describing how coordination was fundamentally part of software development work, Brooks was already noticing how difficult it was in practice. Brooks' Law says that if a project is late then adding more people to the development will slow the work down further [2]. Brooks argues that one of the reasons this law holds is that the addition of more people creates communications overhead, because of the project coordination required. A few years later, Curtis [4] documented the fact that communication and coordination was one of the most difficult and pervasive problems in large-scale software engineering.

The challenge of coordinating development projects has not gone unnoticed by the software development researchers and practitioners who have primarily worked on solutions that provide explicit and visible mechanisms for coordination. These solutions include activities such as planning, defining and following a process, carefully managing requirements and design specifications, measuring process characteristics, regular status meetings to track progress, implementing a workflow system, and so on. They are generally imposed by management, although they require cooperation from everyone to succeed. The software industry has increasingly recognized the importance of these sorts of practices, as evidenced by such things as the increasing adoption of the Capability Maturity Model for Software [6, 13] and increasing attention to software processes [1, 11].

These kinds of solutions facilitate the coordination of development by providing a shared understanding of what the purpose, process and desired outcomes are. They give all the members of the team a common direction. However, plans and processes are most useful only to the limits of one's ability to anticipate. One can certainly plan for various risks and contingencies, and can have different flavors of the process for different circumstances. But there will always remain many decisions that cannot be made ahead of time, unanticipated problems, details to be filled in, mistakes to be corrected and recovered from [18, 19].

The ability to predict will nearly always be much better developing the *n*th version of a product than developing the first version, but predictability will always have its limits, and various coordination-preserving adjustments will always be needed. For this reason, a number of vital elements of software development work such as the exercise of individual skill, habitual but unrecorded patterns of human activity, creative handling of unanticipated conditions and events, and the use of informal personal networks, are hard to represent explicitly in plans and processes [9, 16].

Work that is unpredictable to some degree, and hard to represent explicitly in plans and processes, is impossible to avoid in software development. The usual and perhaps most effective approach to dealing with the limits of predictability is to avoid “over-engineering” a process description or a plan. Beyond a certain level of detail, those doing the work must be trusted to have the skills to do it appropriately, handling exceptions and coordination issues as needed. Interference from plans and processes that anticipate incorrectly is thereby largely avoided.

Yet these essential "outside-the-plan" actions are a serious potential threat to project coordination for two reasons. First, they often need to take account of others' actions, so as not to interfere with them. Second, they need to be communicated to everyone potentially affected, in order to curtail ripple effects. Previous empirical studies of the coordination of software development suggest that developers use informal communication channels [4, 5, 7, 14, 15] to address this threat. Informal communications channels are outside the official reporting structure of a project. They are simply developers' access to other developers, managers, testers, and anyone else they need to interact with during the development process. Unlike the explicit mechanisms described above, they are usually invoked by those doing the work, without requiring management authorization, and perhaps without management's knowledge. These channels help developers fill in the details of work, handle exceptions, correct mistakes and bad predictions, and over time mange the ripple effects of previous decisions and actions. Using informal communications channels, or even unofficial roles such as boundary-spanner [4] is a critical complement to explicit coordination mechanisms.

These communications channels have not received the same kind of attention in software engineering as the more visible coordination mechanisms have. Nevertheless, previous research in other areas has shown their critical importance (e.g., [20]). Changes in the way work is done that disrupt these informal communications channels often have disastrous effects (e.g., [17]). What appear to be merely “casual conversations around the water cooler” often serve to informally exchange the kinds of information and experience that are critical to project coordination.

Since communication between distant sites is more difficult than communication within a single site, this analysis suggests that multisite development will disrupt informal communication channels to a greater or lesser extent. We would expect lapses in coordination to become visible at the point where the products of ongoing work are finally brought together, or recomposed [5]. In order to better understand the loss of coordination in multi-site development, we conducted a qualitative study of the integration process during the development of a large, software-intensive telecommunications product. The site and the empirical methods used are described in the next sections.

3 SITES AND METHODS

In this section we describe the sites of study, including some background on the products built. We also discuss how the work is divided among sites. We conclude with a description of the methods used to analyze and collect the data.

Site

Geographically distributed software development is pervasive among most large companies, including Lucent Technologies. We chose one division of the company for this study, which is part of a larger project to examine all aspects of coordinating multi- site development work. We are studying this department for three reasons.

First, the department was willing to host researchers and provide us with access to developers, documents and other resources. Second, they work in an area of telephony that is both technically complex and growing rapidly in market demand. The product that they build is a real-time system, and the software controls embedded hardware elements, making the design and development work challenging. In addition this product competes in an aggressive market which brings its own pressures to development work. We feel that if we can better support this kind of work, we will find insights into other complex domains. Finally, the department engages a number of cross-site collaborations, both within the development of their product as well with other divisions of the corporation, and other companies. We were interested in exploring the varieties of coordination required in the development process.

In this study we focus on two of those locations, one in the UK and one in Germany, where the department does a large share of its development work. These two sites exchange information frequently and make decisions that require cross-site synchronization. The German site had existed for a number of years, and the people there had considerable experience working together on similar systems. However, it had not previously participated in cross-site development where the architecture is split. The UK site was new, with no existing relationships to any other Lucent site.

The department also has interactions with other divisions of the company because the product must interact with other technologies. Many of these technologies are built in the United States so the developers coordinate work with these other sites. These US sites had not previously worked together, nor had they worked before with the UK or German sites. In all cases, the collaborations span different languages, cultures, and many time zones, making them more difficult.

Methods

The study began with an initial set of 10 interviews with managers and technical leads. The purpose of these interviews was to gather information about what people in the department felt the challenges of multiple site development are. They identified integration as one area where they had experienced difficulties, so we conducted a second round of 8 interviews where we focused on those problems explicitly.

Data analysis followed qualitative protocols [8]. We transcribed all the interviews and then reviewed the materials for specific events within the integration activities. We then looked for causes, and outcomes, as part of building up a rigorous explanation of what happened during integration.

In addition to the primary interview material we had a number of secondary sources available to us, that helped us to learn about the problems and verify some of the information we gathered. Specifically, we reviewed documents related to the development process. We were also given access to a retrospective of the development process that the sites carried out at the end of their first release. These documents and other archival sources helped us to learn about the development context in which the developers found themselves working on cross-site integration.

4 RESULTS

4.1 Limits of explicit coordination mechanisms

The means of coordination that figured most prominently in the integration phase of this project were

- the integration plan,
- component interface specifications,
- software processes, and
- documentation.

In this section, we explore how these were used to coordinate work, and how the limitations of these mechanisms were exposed in this project.

4.1.1 The Integration Plan

As with any development effort, a plan for integrating the system was devised. In this case it consisted of 40 steps, that would bring elements of the overall product together. The initial plan was not closely followed because it was based on several assumptions and predictions that turned out to be in error.

Dependence on the overall development plan. For the integration plan to succeed, the components to be integrated had to be available at the right time. The project suffered from many of the usual difficulties of planning software development, such as changing requirements, staff turnover, and extreme schedule pressure. Add to this the virtual impossibility of predicting the effort and timing of a new product being developed in a new organization, and it is no surprise that the components were not ready for integration on the schedule described in the plan. The original plan turned out, in retrospect, to be very optimistic.

As the developers strove to adjust to the realities of the project, they took an approach one developer described in this way, "We chopped and changed as things became ready." Developers reported that the plan changed weekly. As the project progressed, the developers augmented the documentation to help them deal with the unpredictability, keeping very detailed records, for example, of exactly what steps they had taken, and exactly what files went into each build so that they could quickly back them out if something went wrong.

Substrate environment. Some developers concluded, in retrospect, that the plan missed a critical pre-step, i.e., building the substrate environment on which the product sits. This product relies on a number of substrate software and hardware technologies, built within other divisions of the corporation, and by other companies. Making the product work involved ensuring that it ran on top of these substrate technologies, and so prior to testing the software, the environment had to be assembled. As it turned out, the plan did not adequately take account of the difficulty of this. In addition to slowing down the testing while some developers constructed the environment, it further compromised the integration effort because none of the developers were sufficiently familiar with these substrate technologies and had not given sufficient attention to aligning their efforts with them during development.

Perhaps the biggest uncounted for challenge of assembling the substrate environment was the new relationships that the development team had to forge. In their own work the development group spans two primary sites. The development of the substrate involved working with all the usual partners but also required the developers to get parts from the United States. These substrate technologies took time and energy to assemble, and often problems were very subtle and difficult to spot remotely. This added further time onto the integration process.

4.1.2 Interface Specifications

Specification of interfaces is a critical part of any development effort, especially when it is split among a number of development teams. This project used a contemporary solution to support the specification of their interfaces, ORBIX (based on CORBA) and OMT. Interfaces were specified primarily by event tracing, or "fence diagrams" that showed sequences of messages among processes.

The use of ORBIX/OMT allowed each development group to develop their own part of the product, reassured that the agreements among the different interfaces held. In the process of building their pieces the developers also built simulators to represent the other components that their code would need to interact with. Even before integration got under way, it became clear that the interface specifications lacked many essential details, such as message type, return types, and assumptions about performance. In many cases, the incomplete specifications allowed the developers to proceed with different assumptions about what the other components were doing, and it was not until the initial attempts to make the pieces work together that these alternate assumptions were exposed. By letting development groups write simulators to represent others' code the discrepancies among assumptions had remained hidden during unit testing.

Not all conflicts were hidden until integration. In many other cases, developers realized that conflicts among assumptions were likely to occur, and they contacted developers responsible for components with which they shared an interface, and worked out informal refinements to the specification. It is not at all clear that this sort of incremental refinement could be eliminated, since the refinements were often based on knowledge and understanding that came from the design work itself.

The informal refinements were sometimes, but not usually, recorded in the documentation. Only be examining the code could one infer the information. This caused difficulties on a number of occasions, particularly when the original developer left and a new developer, unaware of the informal specification refinement, took over. It was also very problematic in the test phase, when many bug reports were generated by tests that violated the informal agreement. One developer, for example, reported that there was an informal agreement that, for performance reasons, another component would verify all data it sent to his component. The testers, however, not knowing about this agreement submitted a series of bug reports based on tests which sent the component bad data. Much time was wasted with this and similar problems with undocumented interface refinements.

The developers also had to manage interfaces among the product and its substrate technologies. Differences in assumptions about what the product wanted and what the substrate could provide persisted in part because no-one was using the real substrate in their development work. When integration surfaced these differences the developers faced additional challenges of finding the right people to contact who were organizationally and physically remote. In many cases problems were solved by hosting a substrate developer on-site for a period of time. This not only

resolved problems during their stay, but gave the product developers vital contacts into these other divisions.

4.1.3 Process

The developers used a number of processes in their work to help organize and structure the development environment. These processes evolved as the project did. However, during integration a number of weaknesses with the processes were uncovered.

One response to the multiple site problem was to isolate each site from another in certain ways. For example, the architecture of the product was divided so that the components did not cross sites, each group worked on separate pieces of the product. This separation allowed two change management processes to evolve independently, one for each site. The advantages of working this way were obvious initially, changes got into local builds quickly, and during these initial phases developers at both sites got timely information about their code. Given the conditions that the project faced, for some time this was a sensible operating strategy to allow the rapid development of the product. It did not come without costs though.

When it came time to finally integrate both sites' work, the processes in place created additional effort for those responsible. For example, changes could be found at either sites and logged into both change management systems, fixed, and with a sequence where the code was merged together in between that would mean the same problem was fixed twice usually creating new bugs. To prevent this, changes needed to be logged at both sites, and then the responsibility assigned to one person. However, the numbers for the same change varied between the sites, so it became incumbent on the integration team to know both numbers for the same change. Finally, over time the processes for building the product also became different, so a build that worked at one of the sites, couldn't be made to compile at the other. The extent of these complications certainly was not foreseen, nor perhaps could have been foreseen, when the parallel databases were set up.

The obvious solution to this problem was to consolidate the process at one site, which happened. However, this in turn led to a series of challenges. Especially difficult was getting timely feedback about the results of the build at the other site. This was resolved in part by sending developers from the remote site to the integration site. However, whenever developers from the remote site came to the integration site, they lost their ability to work in their own development environment, which remained back at the old site. So the remote developers faced a choice, go to the central site and find problems, or stay remote and fix them. This slowed down the development effort quite seriously.

Change Control Board. A change control board (CCB) is another mechanism for controlling the changes made to software. Their role is to examine each change request that comes in and decide whether it should be fixed, review the problem to ensure its validity and sometimes to assign a person to work on the revision. The organization under study had a "local" CCB at each cite which coordinated with a project-level CCB. The discussion in this subsection concerns only the project-level board.

Initially, the CCB was almost exclusively managed at one of the two sites. This meant that the members of the CCB were familiar with their code, but collectively knew much less about the work at the other site. In fact, it was hard to know anything about the remote sites' software in advance. The members knew about their software because it had existed as a product before this particular development effort. The software at the other site was completely new. So at first, the CCB did not need to span the sites, and their lack of knowledge was not an issue because there was nothing to know.

Over time as the site began to build their software and the product evolved from a one-site legacy system to a multiple site revision, the lack of knowledge became a problem. When change requests were made for the new code, no one on the board had enough familiarity to make decisions about how best to proceed with the change. Furthermore they could not predict how changes in the code that they did know would impact the software they did not understand. So the developers of the new code found that they got hit with problems stemming from changes made to other pieces of software that probably would not have been implemented in that way if anyone had understood their code better. Since the problem had a gradual onset, the CCB was for a considerable time unaware of the extent of the problems their decisions were causing.

The solution to the problem was straightforward. An architect from the other site was added to the CCB. He was able to bring a broad and deep knowledge of the design of code developed at the other site to bear on CCB decisions, and the problems were largely alleviated.

Sharing and evolving processes. Another process challenge involved evolving practices that both groups of developers shared collectively. One case was a system that one site devised for debugging their code. They used a series of numbers that represented different kinds of problems in the code, so that when the system broke the developers understood why. However, to the other site of developers this system of numbers was as incomprehensible as the bugs in the code. The process was not understood at all, and it took considerable time to become familiar with how the numbers worked and what they were telling the developers about the state of the code.

4.1.4 Documentation

Documenting the software process is something that a number of researchers continue to argue for. The rationale in this case is that reliable and accurate documentation can help the process by supporting access to accurate and reliable information. This project experienced a number of

challenges in documenting their process and decisions rationally though.

First, the technologies that were supposedly there to help with documentation did not met expectations. Specifically, the technologies to support the development of code from design documents could not handle the complexities of the application domain. Eventually this problem was solved by abandoning the tools, and thus the first break between the designs and the code was taken. From this point on developers would have to manually update the design documentation as well as their code to reflect their changes.

Under time pressure to build the system, the developers proceeded with coding, and slowly the code diverged from design. However other people were still relying on that documentation to design their own components and test suites. Over time, and especially in integration, all these inconsistencies came to light. Testers pointed to documentation as demonstration of why code was failing certain tests, other developers pointed to documents that described behavior that they had assumed was still exhibited by the code. All of these inconsistencies had to be resolved as part of building a working product.

This was further compounded by a shift in the role of documentation. In the beginning documents had been focused on recording the design of the system. In later phases of the product documentation became oriented around the change management process. So instead of having descriptions of the components and their behavior, the documentation was organized around the changes and updates to the system. This kind of documentation does not reveal the overall architecture and behavior of a component let alone the product easily. The developers were left with one accurate source of documentation -- the code itself. And as the original designers of the system left, the software itself became increasingly the source of their system understanding.

Yet, omissions in documentation, which can simply arise from a careless mistake, or because it is almost impossible to completely describe a system, can create serious problems in development. This project faced one such problem with some documentation from the one of the substrate technologies' provider. In this case the omission involved ensuring that header files contained certain non-obvious details. As this information was not included in the documentation, the headers were not correct, and it took weeks to figure out that this was the problem, simply because it was something that would never have occurred to anyone to think of in absence of the documentation.

4.1.5 Summary

In hindsight it seems easy to see that any of these things might have been problems that could have been resolved with better plans. This is not however the case. The essence of the difficulties with these plans, specifications, processes, and documentation, is the fundamental challenge of predicting the highly variable process of software development. Requirements change, other related development efforts proceed and impact the work that you do, systems seem to provide support but present subtle challenges to work with, and so forth. All projects manage these variabilities as part of day to day software development work by ad hoc communication, informal agreements, testing assumptions, and so on. As we argued in section 2, none of these coordination mechanisms can work without allowing for the filling in of details, handling exceptions, coping with unforeseen events, and recovering from errors. This section has presented a number of examples of how plans, specifications, and processes were modified or augmented in order to allow the work to proceed. What perhaps makes this project more challenging than others is the fact that multiple sites have to potential to disrupt the conditions necessary for these "adjustment" techniques. This is the topic of the next section.

4.2 Barriers to informal communication

It was clear from the interviews that splitting the organization across sites presented barriers to informal communication during the project, causing serious problems. The primary barriers which led to coordination breakdowns were

- Lack of unplanned contact,
- Knowing who to contact about what,
- Cost of initiating contact,
- Ability to communicate effectively, and
- Lack of trust, or willingness to communicate openly.

Our findings in each of these areas are discussed in the following sections.

4.2.1 Lack of unplanned contact

In a typical co-located development effort, project members run into each other in the hallway, at the coffee machine, in the cafeteria, and elsewhere, on a frequent basis. They discuss many things, including many that have nothing to do with project work. But they also bring up the project as a topic of discussion, especially things that are currently on their minds, or are causing them some concern. This is not necessarily explicitly to get help, or to formally notify others of specific events, but rather just the usual sort of friendly exchange, common in the workplace.

These sorts of unplanned contacts seem to be surprisingly important in keeping projects coordinated. For example, one interviewee told us of an incident in which, during the course of such an unplanned discussion, it came to light that he and a co-worker were making contradictory assumptions about what board would have a particular digital signal processing chip. They were able to resolve the issue in a matter of a few minutes. But had they not discovered this

difference in assumptions as early as they did, this minor problem could have become extremely costly.

What makes this and similar incidents significant is that the participants were not aware of any need to communicate. There was absolutely no reason either one knew of, even if communication could be initiated very easily, to contact the other person to discuss the project. Many conflicts in assumptions follow this pattern, because project members are often unaware of the assumptions they are making, or that others might be making conflicting assumptions. This form of coordination is predicated on relatively frequent unplanned communication, during the course of which relevant information and important issues may come to light.

In our interviews, there were a number of problems that seem to be the result of a general lack of information. Developers at one site, for example, had developed a very handy step tracing tool that providing information on memory usage, CPU usage, and so on. While developers at the same site used the tool extensively, those at the other site did not know of the tool's existence for months. It often took weeks to deal with problems that could have been solved very quickly with the tool. The fact that developers at one site were aware of the tool and those at the other site were not, makes it seem quite likely that unplanned contacts played a significant role in disseminating awareness of the tool.

Overall, several consequences seemed to flow from the relative lack of unplanned contact. One was to make it much less likely for conflicts and issues to be recognized. If a developer is aware that an issue exists, it is possible to take action to correct it. But since unplanned contact seems to be one of the primary mechanisms for bringing issues to light, many more conflicts went unrecognized until later in the development. Another consequence was just the general lack of background information across sites, i.e., how they work, what issues are most pressing to them, how they typically communicate with each other, site-specific vocabulary, and the responsibilities, expertise, and relationships among those at the other site.

4.2.2 Knowing who to contact about what

Developers often reported great difficulty in determining the appropriate person to contact at the other site. If there was a need to coordinate on an interface specification, for example, there was no straightforward way to find out who was responsible for the component on the other site.

Developers found several workarounds for this problem, although none was entirely satisfactory. One was to look for clues buried in the documentation, such as names at the bottom of web pages containing specifications. Often those whose names were listed as authors were the correct person, or could point to the correct person. Another frequently used workaround was to contact a system architect at the other site. Architects were known to have a very broad knowledge of the system and who was doing what.

Once some of the developers had spent a significant amount of time at the other site, these individuals became "contact people" or "liaisons." A visitor from the UK, for example, would often be used by those at Germany to help them to figure out who they should get hold of. When these people returned to their own sites, they also often acted as the first point of contact to people at the other site. In addition, people at their own site regarded them as something of an expert on the other location, and would often come to them with questions about who was doing what, and about how things worked at the other site. This, of course, imposed a significant cost on the liaisons, particularly in the earlier days when there were very few people with cross-site experience.

4.2.3 Cost of initiating contact.

When developers are co-located, contact can generally be initiated quite easily. There are many cues about who is around, how available they are (e.g., is the door open?), and so on. If someone's office is only a few feet away, one need expend little effort to stroll down the hall for a chat. Perhaps more significantly, it is socially comfortable to do so, since you know them well, know how to approach them, and have a good sense of how important your question is relative to what they seem to be doing at the moment. For developers at different locations, the cost of initiating contact was often much higher.

Who is available. One difficulty is simply determining if someone is available. If they do not, e.g., answer the phone, they could be momentarily tied up, or in the midst of a crisis, or it could be a holiday, or they could be on a vacation (of many weeks). Unlike the same-site case, it is not easy to determine if the person being called is likely to be available. Our interviewees reported it often took many attempts, over many days and often involving several people, to contact someone at the remote site. In the mean time, progress was often held up.

Time difference. There is only one hour time difference between the two sites, so one would not think that would make much difference. But this can be very deceptive. There is an hour lost at the beginning (or end) of each day, but since typical lunch time was displaced by an hour, that meant that another hour for each site was unavailable. Add to that the fact that developers at the German site tended to start earlier and finish earlier in the day, and an additional hour or two of potential time overlap was lost. These time differences meant that something that could be handled in a matter of minutes for a same-site development would often have to wait at least until the next business day.

Reduced responsiveness. People who know each other relatively well, and know that they will be dealing with each other frequently in the future tend to be more responsive to each other's needs and requests. Individuals at different

sites often seemed relatively unresponsive, e.g., not answering e-mail or voice mail promptly. This again makes it much more costly to initiate contact, since a single message is less likely to be effective.

Several of our interviewees believed that it was difficult to accurately gage the importance of a message from another site. Often, they did not understand the context well enough to determine why the question was being asked, or to see why it was an important request and not merely arbitrary or irrelevant.

Consequences: Three consequences of the high cost of initiating contact were mentioned. First, developers did not try to communicate as frequently as they would have had they been co-located. They were more inclined to take the risk that significant coordination issues would not arise if they did not check assumptions, etc. People also reported that they were not consulted on decisions made at the other site that affected them. Second, cycle time was increased. Even when messages were answered promptly, developers believed that resolution was far more likely to stretch into the next day. Worse, it often took several days, rather than minutes or hours, to make the right contact. Finally, issues had to be escalated more often, if an adequate and timely response was not forthcoming.

4.2.4 Ability to communicate effectively

Once the right person has been identified and communication is initiated, information must be conveyed in a relatively complete and undistorted way for communication to effectively support coordination.

What can't be seen. The most obvious obstacle to communication is simply not being able to see the same things, have access to everything in the environment. This problem took a variety of forms. For example, one lengthy problem involved a hardware and software component from an internal contractor. It was not behaving properly, and the supplier could not duplicate the problem, even though they had identical hardware and could access the software remotely to ensure it was correct. However, they could not "see" a faulty firmware chip, and the problem was very difficult to solve.

Another example illustrates a very different form of this problem, in a developer at one site could not see what a tester at another site was doing. As the developer explained it, the documentation essentially said, "leave blank if you want to get all the faults on the system and this tester was typing in 'b-l-a-n-k,' the actual word 'blank' into the system and that's it, a lot of problems occurred on that end." The developer could not duplicate error, naturally, and finally, after several weeks of trying to straighten it out, went to the other site, where the problem was spotted immediately.

Communication technologies. Face-to-face meetings, planned and unplanned, are the most common form of communication among co-located developers. When this communication channel is removed or greatly attenuated by geographically distributed developments, developers are forced to fall back on other communication media which they used with varying degrees of success.

Most of the native English speaking developers found the phone to be a useful medium for one-to-one communication. It often required considerable patience, but they generally found that issues could be resolved. It was particularly effective for asking very specific questions, less so for reasoning about hypotheticals. The non-native English speakers, on the other hand, seemed to regard these same telephone conversations as much less effective. As one described it, "it's hard to explain something to someone you don't know in your second language." They also found that conversations frequently became very emotional, and took a great deal of time and energy.

Some developers reported that it was very difficult to get everyone together who was needed in order to solve a problem. If everyone was at the same site, the people involved could gather in an office or conference room and reach a conclusion quickly. But when they were distributed, someone generally had to contact people one at a time, and get "bounced around" from one to another. Problems such as determining which component generated an error that propagated through a number of other components were very hard to resolve this way.

Conference calls, on the other hand, i.e., calls involving more than two people, and often 6-10 or even more, tended to be less than satisfactory. They were adequate for relatively simple discussions, and for status meetings, but were thought to be inadequate for contentious issues or for substantial technical discussions. As one developer said, "every conference call I walked out of, if I ask somebody what do you understand from it, and they say, 'I don't know'."

E-mail was the preferred means of communication for many, especially the non-native English speakers. The advantage of e-mail was that one could take time to think, to be very careful in the language one used, and could do research if necessary before responding. It seemed to take much of the pressure off non-native speakers communicating in a less-familiar language. They were able to overcome some of the limitations of this text-only medium by constructing "text diagrams," or simple diagrams built from text characters. They also attached other text documents, such as log files, to messages.

There were a number of difficulties surrounding the distribution and use of various kinds of documents. Distribution was made particularly difficult by the fact that there were both UNIX and PC platforms in use, and a variety of applications used for editing documents. Even when developers used the same platform and applications, there was often a version mismatch so that, as one developer reported, they usually ended up in a secretary's

office faxing a document to someone at every multi-site meeting.

Using documents, or collaboratively viewing files, presented many problems. Developers generally found it very difficult to look at code together with someone on the telephone. They could not point to places in the code, or scroll the other person's screen to a point of interest.

Different cultures and languages. The sites represented different cultures in at least two senses, and both of these influenced how they interacted. Most obviously, the sites are in different countries with distinct national cultures. These cultures differ in many subtle ways. One that was mentioned very frequently was the more direct communication style of the Germans as compared to the British. A German interviewee mentioned that Germans are accustomed to calling someone up and immediately saying, e.g., there is a problem with your code. The British, on the other hand, tend to expect more in the way of a greeting, and more of an indirect "polite form" for suggesting there might be an error in their code. Such differences made it difficult for the Germans and British to work together without confusing or irritating each other.

There are also differences in how Germans and British customarily regard hierarchy and process. While this is clearly an oversimplification, one might say that the Germans are more comfortable with a detailed process that specifies the steps in a development fairly completely and precisely. The British, on the other hand, are more comfortable with processes specified only at a relatively high level. Similarly, the Germans have a greater tendency to take hierarchical relationships a bit more seriously, expecting and receiving a greater degree of direction from managers and supervisors. These differences often led to misunderstandings about, for example, whether a developer could simply take the initiative to change something, or whether a change required a managerial decision or a process exception. Discussions around such issues often led to feelings that the person on the other end was either being obstinate and rule-bound, on the one hand, or a bit of an anarchist and disrupter on the other. This did not help to foster cooperation.

The sites also represented two distinct engineering cultures. The German technical staff was experienced in real-time systems and telecommunications, while the British staff, in addition to being generally younger and less experienced, tended to be more UNIX-oriented. These differences often made it difficult to communicate, since they thought about problems differently and used different vocabularies.

Consequences. One developer estimated that any necessary discussions for a small change involving only one site could generally be resolved within an hour. The same change, if trivial or nearly so, would probably take a day if two sites were involved, and several days or more if the change was non-trivial. Although the primary consequence seemed to be cycle time, some developers mentioned that the difficulty of communication also influenced the way in which they went about modifying the code. They strove to make absolutely minimal changes, regardless of what the "best" way to make the change would be. This was because they were so worried about how hard it would be to repair the problem if they "broke the system."

4.2.5 Willingness to communicate openly: Trust

As the two primary sites began to work together, there was initially little trust between people at different locations. There was concern about holding onto work, worries that one site might be closed down and everything moved to the other. There was also little sense that they were partners, cooperating toward the same ends. This manifested itself in "uncharitable" interpretations of behavior, as when, for example, someone would say, "we can't make that change," it was often interpreted as "we don't want to make that change," whether it would benefit the overall project or not.

The situation improved considerably over time, and visits across sites seem to have been pivotal. As one developer noted, they just did not seem to be able to make progress until they had worked together face to face. Primarily because of difficulties encountered during integration, about a half-dozen developers traveled from the UK to Germany (where the central test site was located and where the build was done) for significant periods of time, often months. After working together, the relationships between the sites began to change. As one developer said, "things eased a lot when we met these people face to face instead of over telephones and e-mail. We worked much closer, and resolved things much quicker as well."

The change seems to have arisen from several sources. For one thing, the differences in communication style, e.g., the relative directness of the Germans, was seen in context. The British developers saw that they spoke to each other this way, and it was not intended to be, nor interpreted as, rude or insulting. In a similar fashion, they became accustomed to other cultural differences, and were less mystified or offended by behavior that had seemed strange or out of place.

Other factors leading to these changes seemed to be the perception that "Now there's less of a wall between the UK and Germany, and we can see that they're in basically the same boat as us." By working together, face to face on a daily basis, a sense of common goals and purpose was eventually established. People began to give more charitable interpretations of ambiguous behavior. Rather than assuming that disagreements were arbitrary, each side was more likely to assume that the other side was acting out of a genuine concern about the welfare of the project. Disagreements still arose, but the context of the discussion was achievement of a common goal. As one developer said, "I learned to have a lot of patience, and we got through it quite well."

Time spent at the other site familiarized each party with the terminology and problem-solving style of the other. They learned to communicate much more effectively, and understood the context underlying the concerns expressed by people at the other site.

As mentioned above, the visits also created a number of de facto "liaisons" who understood the other site very well, and could act as points of contact for interactions between sites. They could provide information about who was responsible for what, how to get things done, could interpret information that seemed cryptic, and so on. This continues to be a very important role, but it was especially so when only a few people had spent appreciable time working face to face with people at the other site.

Consequences. The lack of trust led to a reluctance to share information, for fear that if work was able to move between sites, one site might be closed down. It also caused very uncharitable attributions to be made whenever disagreements arose. This initially caused hard feelings, and probably introduced considerable delay in resolving problems that spanned sites, since each side tended to assume the other was just being difficult, rather than trying to understand the concerns behind their position.

5 CONCLUSIONS

In this paper we have shown that multiple site development works against informal communication channels by creating geographic boundaries among developers who need to work together. Simple things like meeting in hallways and knowing who to ask about a problem, are more than simple pleasantries of development, they are a vital part of the coordination of this kind of work. Since designs never exhibit perfect modularity and are never error-free, process execution is rarely flawless, and the world is never completely predictable, informal communication will be essential to maintain project coordination.

There are, however, some sound ideas on how to keep the need for cross-site communication to a minimum. It was pointed out many years ago [12]that a good design is one in which design decisions about each component can be made in isolation from decisions about other components. As we noted in the introduction, it follows that good organizational design should mirror product structure [3, 12] in order to minimize the need for communication and coordination among groups. Geographic distribution of those groups is just an extreme case where it is even more important to reduce the need for communication, and correspondingly more important to have a good design.

As fundamentally sound as this analysis is, we believe there is ample evidence indicating it is necessary to extend it in two major ways in order to account for the complexities of communication and coordination of real-world development projects. First, product structure is only one of several domains in which dependencies arise, and hence is only one of the domains in which coordination is required. When, for example, intermediate work products are handed off between groups, it is necessary for both to have a clear idea of what steps have and have not been carried out at that point. This generally requires a common understanding of a defined development process. Large projects are also replete with temporal dependencies that have major implications for resource planning and on-time delivery. Good design is a powerful coordination mechanism, but it is by itself insufficient. In addition to coordination on the basis of *what* is being developed, it is also essential to coordinate *when*, *how*, *who*, and *where*. Things like project plans, defined processes, staffing profiles, and so on, serve as coordination mechanisms in these domains.

The second extension was anticipated by Conway (although it seems to have received very little attention in the last 30 years). Even though communication needs are the primary consideration in organizing a design project, Conway adds "this criterion creates problems because the need to communicate at any time depends on the system concept *in effect at that time*" [3] (p.31, emphasis added). In other words, the "volatility" of the design over the course of the project limits the degree to which it is possible to optimize the project organization. This astute observation applies to all of the coordination mechanisms, not just product design. To the extent the original design, plan, development processes, and so on are unstable, substantial communication between teams and across organizational boundaries will be required. This is precisely what geographically distributed organizations do least well, since, as we have seen, communication is greatly attenuated across sites.

Several lessons for multi-site development follow. First, reduce the need for cross-site communication as much as possible:

- Attend to Conway's Law: To the extent possible, assign work to different sites according to the greatest possible architectural separation in a design that is as modular as possible.
- To the extent possible, only split the development of well-understood products (or parts of products), where plans, processes, and interfaces are established and likely to be very stable. Instability will greatly increase the need for communication.

Second, take all possible steps to overcome the barriers to informal communication.

- Front load travel, i.e., don't postpone using the travel budget, bring people who need to communicate together early on. All other means of communication will work better once developers, testers, and managers have some face-to-face time together.
- Plan this travel in order to create a pool of liaisons. Give the early travelers the explicit assignment of

meeting people in a variety of groups at the other site, and learning the overall organizational structure. Try to send gregarious people who will enjoy this role. When they return, make it known they can help with cross-site issues, and free up some of their time to do so.

- Invest in tools to make it easier to find organizational information, check availability of people, and to have more effective cross-site meetings, both planned and spontaneous.

ACKNOWLEDGMENTS

We would like to thank all the developers who gave their time willingly to answer our questions about integration. We would also like to thank Dorene Brummel for the timeliness of her help with this study.

REFERENCES

[1] B. Boehm, "Anchoring the Software process," *IEEE Software,* Vol. 13, No. 4, 1996, pp. 73.

[2] F.P. Brooks Jr., "The Mythical Man-Month," *Datamation,* Vol. 20, No. 12, 1974, pp. 44-52.

[3] M.E. Conway, "How Do Committees Invent?" *Datamation,* Vol. 14, No. 4, 1968, pp. 28-31.

[4] B. Curtis, H. Krasner and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM,* Vol. 31, No. 11, 1988, pp. 1268-1287.

[5] R.E. Grinter, "Recomposition: Putting It All Back Together Again," *Proc. ACM Conference on Computer Supported Cooperative Work (CSCW '98)*, ACM Press, 1998, pp.

[6] J. Herbsleb, D. Zubrow, D. Goldenson, W. Hayes and M. Paulk, "Software Quality and the Capability Maturity Model," *Communications of the ACM,* Vol. 40, No. 6, 1997, pp. 30-40.

[7] R.E. Kraut and L.A. Streeter, "Coordination in Software Development," *Communications of the ACM,* Vol. 38, No. 3, 1995, pp. 69-81.

[8] M.B. Miles and A.M. Huberman, *Qualitative Data Analysis: An Expanded Sourcebook*, Sage Publications, Inc., Thousand Oaks, California, 1994.

[9] R.R. Nelson and S.G. Winter, *Evolutionary Theory of Economic Change*, Harvard University Press, Cambridge, MA, 1985.

[10] M. O'Hara-Devereaux and R. Johansen, *Globalwork: Bridging Distance, Culture, and Time*, Jossey-Bass, San Francisco, CA, 1994.

[11] L. Osterweil, "Software Processes are Processes Too," *Proc. 9th Conference on Software Engineering*, 1986, pp.

[12] D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM,* Vol. 15, No. 12, 1972, pp. 1053-1058.

[13] M. Paulk, B. Curtis, M. Chrissis and C. Weber, *Capability Maturity Model for Software (Version 1.1)*, Technical Report, CMU/SEI-93-TR-024, Pittsburgh, Software Engineering Institute, Carnegie Mellon University, February, 1993.

[14] D.E. Perry, N.A. Staudenmayer and L.G. Votta, "People, Organizations, and Process Improvement," *IEEE Software,* Vol. 11, No. 4, 1994, pp. 36-45.

[15] J.M. Pickering and R.E. Grinter, "Software Engineering and CSCW: A Common Research Ground," *Software Engineering and Human-Computer Interaction: ICSE'94 Workshop on SE-HCI Joint Research Issues*, R.N. Taylor and J. Coutaz ed., Springer-Verlag, Heidelberg, 1995, pp. 241-250.

[16] M. Polanyi, *Tacit Dimension*, Peter Smith Publications, 1983.

[17] P. Sachs, "Transforming Work: Collaboration, Learning, and Design," *Communications of the ACM,* Vol. 38, No. 9, 1995, pp. 36-44.

[18] K. Schmidt, "Of Maps and Scripts: The Status of Formal Constructs in Cooperative Work," *Proc. International ACM SIGGROUP Conference on Supporting Group Work GROUP '97*, ACM Press, 1997, pp. 138-147.

[19] L. Suchman, *Plans and Situated Actions: The Problem of Human-Machine Communication*, Cambridge University Press, Cambridge, UK, 1987.

[20] L.A. Suchman, "Office Procedure as Practical Action: Models of Work and System Design," *ACM Transactions on Office Information Systems,* Vol. 1, No. 4, 1983, pp. 320-328.