Geographically distributed development teams face extraordinary communication and coordination problems. The authors' case study clearly demonstrates how common but unanticipated events can stretch project communication to the breaking point. Project schedules can fall apart, particularly during integration. Modular design is necessary, but not sufficient to avoid this fate.

Architectures, Coordination, and Distance: Conway's Law and Beyond

James D. Herbsleb and Rebecca E. Grinter, BELL LABORATORIES

Software engineering researchers have long argued that the architecture of a system plays a pivotal role in coordinating development work. Over 30 years ago, Melvin Conway proposed what has since become known as Conway's Law—that the structure of the system mirrors the structure of the organization that designed it.[1] This relation, Conway argued, is a necessary consequence of the communication needs of the people doing the work. David Parnas, in fact, defined a software module as "a responsibility assignment rather than a subprogram,"[2] driving home the idea that modular design enables decisions about the internals of each module to be made independently. Of course, the computer that runs the software doesn't care. The point of structure is to support coordination of the development work.

Architecture, however, addresses only one of the several dimensions on which we must coordinate development. To support efficient use of resources, projects require plans that specify when milestones must be completed and who will do the

0740-7459/99/$10.00 © 1999 IEEE

work. Moreover, to work together effectively, people must agree on how the product will be developed—that is, the project's process. Ideally, architectures, plans, and processes—coordination mechanisms—would be sufficient to establish effective coordination among teams.

In the real world, this ideal is seldom fully attained because unpredicted events occur that must be accommodated. Estimates are inaccurrate, process steps are executed imperfectly, requirements and technologies change, and people leave. The wise organization will anticipate this, making provisions for modifying designs, plans, and processes. But empirical studies suggest that developers also heavily rely on informal, ad hoc communication[3–6] to fill in the details, handle exceptions, correct mistakes and bad predictions, and manage the ripple effects of all these changes.

Empirical studies suggest that developers rely heavily on informal ad hoc communications.

In an organization where everyone is in a single location, this sort of informal communication is taken for granted and often goes almost unnoticed. People are frequently surprised that casual conversation at lunch, next to the coffee machine, or in a coworker's office is a critical means of coordination, because it operates "invisibly."

Empirical Methods

For our case study, we chose a Lucent Technologies department that develops real-time embedded systems for a rapidly growing market with extreme time-to-market pressures. The department engages in a number of cross-site collaborations within their own product group, with other corporate divisions, and with other companies. We focused on a new product release and collected data concerning the two locations (the UK and Germany) that did most of the software development work. In addition, both these sites had interactions with other departments, often in the US, to ensure the product successfully interacted with other systems. The different languages, cultures, and time zones complicated these collaborations.

We chose to study a geographically distributed project partly to improve our chances of observing coordination problems as they arose and partly to see if and how geographic separation places stress on informal communications; geographic separation is a pressing practical problem in its own right. Many companies are driven to distribute development resources around the globe for marketing purposes and because of acquisitions, cost considerations, and the availability of needed expertise. Geographic distribution challenges coordination mechanisms and informal communication by requiring that they be robust across distances.[7]

We interviewed 10 managers and technical leads who identified product integration (the work necessary to assemble the product from its components) as the activity that suffered the most from geographic distribution. We then conducted eight more interviews to focus specifically on integration. We transcribed and analyzed the interviews for specific events and then looked for causes and outcomes as we built a rigorous explanation of what happened during integration. We used several types of documentation to support and extend our analysis. We were also given access to a thorough retrospective the department conducted of their first product release, and we made extensive use of the report and the qualitative data it was based on.

In the results that follow, we strive to show the kinds of unpredicted events that caused this project's coordination problems. Because we studied the first release of a new product created by a new organization, it probably had more than its share of surprises. Nevertheless, we think they illustrate the kinds of unanticipated events that arise in any software engineering project.

Architecture-Based Coordination

The project generated the system architecture with Orbix—a commercial, Corba-based object-oriented application development product. This tool specified interfaces with event tracing, or *fence diagrams*, that showed message sequences among processes. The team expected the application development tool to support code generation, but the domain proved too complex for the system to model. Thus, they decided to develop the code manually based on the design agreements. From this point on, the design documentation competed with coding work for the developers' attention.

Following the design agreements, project teams

developed each component at a single site, in relative isolation from teams at the other sites. Each team built its own simulators to represent other components that their code would need to interact with. It turned out, however, that the interface specifications lacked essential details, such as message type, return types, and assumptions about performance. In many cases, developers proceeded unknowingly with incorrect assumptions about other components. Because development groups had written simulators to represent others' code, the discrepancies remained hidden during unit testing and were not exposed until integration.

There were times, of course, when a developer working on a particular component did recognize potential conflicts. In such cases, he or she generally tried to identify the people responsible for components that used the interface and then tried to work out specification refinements. These refinements were infrequently recorded in the documentation because that took time away from development. This caused difficulties on a number of occasions, particularly when the original developer left and a new person, unaware of the refinement, took over. It was also problematic during testing, when tests that violated these agreements generated bug reports. For example, one developer reported that there was an agreement that, for performance reasons, another component would verify all data it sent to his component. The testers, unaware of this agreement and working at a different site, submitted a series of bug reports based on tests that sent the component bad data. The issue proved difficult to resolve.

The developers also had to manage interfaces between the product and its substrate technologies. Important differences existed in assumptions about what the product wanted and what the substrate could provide. These differences surfaced during integration and took a long time to resolve because it was hard to find the right people to contact. The problems were often solved only by hosting a substrate developer on site.

Plan-Based Coordination

The project initially had a 40-step integration plan, which was not closely followed because it depended on the overall development plan and assumed that the substrate environments would be easy to assemble.

The integration plan relied on having components available for integration at certain times, which came from development plan dates. However, the project suffered from many of the usual difficulties and delays, such as changing requirements, staff turnover, and extreme schedule pressure. Compounding this was the virtual impossibility of predicting how long it would take a new organization to build a new product. In retrospect, it was not surprising that the components were not ready for integration on the schedule described in the plan.

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

Some developers concluded, in retrospect, that the plan missed a critical, initial step: building the product's substrate environment. The integration plan did not adequately account for the difficulty of assembling the substrate for testing with the product in two ways. First, the substrate was itself technically complex and took time to learn. Second, assembling the substrate required interacting with a new, remote site located in the US. Both problems slowed down testing and compromised the integration effort because, until the developers familiarized themselves with the substrate, they could not align their code with it.

Architecture, plans, and processes are all vital coordination mechanisms in software projects.

Process-Based Coordination

The developers also used a number of processes to help organize and structure the development environment. These processes evolved with the project, but it was not until integration that they discovered a number of weaknesses.

In the beginning, each development site had its own change-management process. Early on, this iso-

lation supported rapid development by allowing developers to get their changes into the local builds quickly. When it came time to finally integrate both sites' work, however, the separate processes were cumbersome. For example, changes were sometimes found and logged into the change-management system at both sites, and fixed twice, usually leading to new bugs. This was eliminated only with an awkward manual system for logging changes at both sites simultaneously. In addition, the build processes at the two sites diverged over time, so a build that worked at one site wouldn't compile at the other. The extent of these complications certainly was not foreseen when the parallel databases were set up.

The obvious solution to this problem was to consolidate the process at one site. This solution, however, led in turn to a series of new challenges. Especially difficult was getting timely feedback about the build results from the other site. When developers from the remote site came to the integration site to get feedback, however, they lost the ability to work in their own development environment. So, the remote developers faced a choice—go to the central site and find problems or stay remote and fix them—which significantly slowed the development effort.

The project used a change-control board approach for examining each change request and deciding whether, how, when, and by whom it should be fixed. Initially, the CCB was located almost exclusively at one site, which made sense because that site had existing software being adapted for the new product. The other location was starting from scratch and had no project-level changes for the CCB to review. Over time, as the second site began to build their software and the product evolved from a one-site legacy system to a multiple-site revision, the CCB's one-site focus became problematic. The new code developers got hit with problems stemming from changes made to other pieces of software that probably would not have been implemented that way if anyone had understood their code better. Because the problem had a gradual onset, the CCB was slow to recognize the extent of the problems. To solve this, an architect from the new code site joined the CCB. He added the broad and deep knowledge of the code design developed from his site to CCB decisions, which largely alleviated the problems.

Another process challenge involved evolving practices that both sites shared. One case was a system developed by one site for debugging 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, this system of numbers and the processes that generated and used them were incomprehensible to developers at the other site for a considerable time.

Distance and Flexible Ad Hoc Communication

Architectures, plans, and processes are all vital coordination mechanisms in software projects. However, their effectiveness extends only as far as our ability to see into the future. Handling the unanticipated both rapidly and gracefully requires flexible ad hoc communication. This need became clear as we examined how distance interfered in a variety of ways with the project teams' effective communication.

Unplanned contact

When developers work at the same location, project members run into each other frequently. These chance meetings are basically social; they are not necessarily intended to request help or to notify others of specific events. However, these unplanned contacts are surprisingly important in keeping projects coordinated. For example, one developer described a chance meeting where he discovered that he and his coworker had contradictory assumptions about which board would have a particular digital signal processing chip. They were able to resolve the issue quickly, but had they not discovered the difference, it could have been extremely costly. What makes this and similar incidents significant is that the participants were not aware of a need to coordinate, yet they exchanged critical information.

Collocated developers can initiate communication easily because they know who is around and if they are available.

Unsurprisingly, there were no chance discussions across sites. As a result, the developers didn't recognize and resolve many conflicts early on. It was also harder to pass general information across sites such

as how things work, what issues have priority, responsibility assignments, and who was an expert at what. The lack of chance encounters also inhibited the transfer of tools across sites. For example, developers at one site developed a useful step-tracing tool, which spread throughout the site by word of mouth. Developers at the other site did not know of the tool's existence for months and took weeks to solve problems that could have been handled quickly with it.

Knowing whom to contact

Developers often reported great difficulty in deciding who to contact at the other site with questions. They devised several workarounds for this, although none were entirely satisfactory. One way was to find the author of the relevant system part's documentation; he or she often knew the answer or could point to an individual who might. Another strategy was to contact a system architect or project manager at the other site because they had a broad knowledge of who was working on what.

Once some of the developers spent a significant amount of time at the other site, they became contact people or liaisons. A visitor from the UK, for example, would often be used by those in Germany to help them figure out who to contact. When these people returned to their own sites, they also acted as the first contact point for people at the other site. In addition, people would often come to them with a wide variety of questions 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 few people with cross-site experience.

The difficulty of initiating contact

Collocated developers can initiate communication easily because they know who is around and if they are available. For example, if someone's office is only a few feet away, it's easy to contact them. More significantly, it is socially comfortable to do so because you know them, 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 difficulty of initiating contact was often much greater. Developers found it hard to know whether someone was available. Unanswered phone calls, for example, could mean someone was in a meeting, working on a hard problem, in the midst of a crisis, away for a site-wide holiday, or on vacation.

Another problem was the time difference between the two sites. Although there is only an hour difference between the UK and Germany, it still led to many missed hours during the day. There was an hour lost at the beginning and end of each day and another hour lost due to different lunch times. The problem was compounded because the German site generally started earlier and left earlier, eliminating an additional hour or two of potential overlap time.

The most obvious obstacle to communicating across sites is the inability to share the same environment and to see what is happening at the other site.

We noticed that across sites, people seemed more unresponsive—not answering e-mail or voice mail promptly—which reduced the incentives to communicate because a single message was not likely to be effective. Furthermore, it was harder to assess the importance of a message from the other site because the receiver did not understand the context well enough to determine the question's importance. In general, the default assumption for messages sent by unknown people was that they were unimportant.

Initiating contact was particularly difficult in the frequent cases requiring more that two people to solve a problem. Issues that could be resolved very quickly just by gathering the right people in front of a whiteboard frequently stretched out for many days when multiple sites were involved.

We identified three consequences of the difficulty of initiating contact. First, developers did not try to communicate as frequently as they would have; they were more inclined to take the risk that problems would not arise if they did not check, so developers reported that they were not consulted on decisions made at the other site that affected them. Second, cycle time increased. Even when messages were answered promptly, resolution took far longer and stretched 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 to management more often.

The ability to communicate effectively

The most obvious obstacle to communicating

The Consequences of Distance

Based on our qualitative observations, the primary effect of distance is that it stretches out issue resolution. Even relatively simple issues that could be resolved in a few minutes or hours if all the needed parties were collocated and could gather around a white board often take days or weeks to resolve. When the issue arises, it is difficult to know who to contact, how to get and hold their attention, how to understand and be understood, and so on. Here is an example of how even the simplest and most straightforward things can be troublesome.

The documentation for a particular function directed the user to supply certain arguments to return particular parameters, or to "enter blank" to get all the parameters. The developer was not located at the site where the function was tested. The tester wrote a problem report, noting that the function did not return all parameters described in the documentation. The developer to whom the report was assigned, however, could not duplicate the error. They exchanged messages for three weeks, until the developer flew to the test site because the problem was holding up work. Within the first five minutes, the developer watched the tester type in "b-l-a-n-k" and then hit return. A misunderstanding that could have been resolved in a few minutes took three weeks.

across sites is the inability to share the same environment and to see what is happening at the other site. For example, developers found it hard to review documents with someone over the phone, because they couldn't point to specific items. We discuss other, more serious problems in the boxed text, "The Consequences of Distance."

Collaborative technologies offer the promise of supporting cross-site development; however, we found that they worked with varying degrees of success. The language on this project was English, and most of the native English speakers found the phone useful for one-to-one communication, especially when they had very specific questions. However, the nonnative English speakers found these same telephone conversations much less effective, because "it's hard to explain something to someone you don't know in your second language." They also found that conversations frequently became emotional and required considerable time and energy.

The nonnative English speakers preferred e-mail communication because it allowed them to spend time composing and translating their response. Unlike the phone, it relieved the pressure of communicating in a less-familiar language in real time about complex issues. Furthermore, the developers 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 to messages, such as log files. However, document distribution was still difficult, because the developers used different platforms, including Unix and PC machines, along with a variety of word processors.

The project had difficulties with cross-site meetings. At a single site, developers can gather to discuss a problem and reach a conclusion. Across sites, developers found it much harder because conference calls tended to be less than satisfactory for discussing technical issues (although they worked satisfactorily for simple issues and status reports). As one developer said, "every conference call I walked out of, if I asked somebody 'What do you understand from it?' they said, 'I don't know.'"

The different cultures also influenced the team's ability to communicate effectively. One difference was the more direct communication style of the Germans as compared to the British. A German developer mentioned that Germans are accustomed to calling someone up and immediately saying, for example, that there is a problem with their code. The British, on the other hand, tend to expect more of a greeting and an indirect "polite" form of suggesting possible errors. The German style initially seemed rude to the British, while the British style often confused the Germans, who wondered why the caller didn't come to the point.

Primarily, these communication problems led to lengthened cycle time to resolve systems issues. One developer estimated that any necessary, yet small, changes involving only one site were resolved within an hour. The same change, if trivial, probably took a day if two sites were involved, and several days or more for nontrivial changes. However, some developers mentioned that the communication difficulties also influenced the way in which they modified the code. They strove to make absolutely minimal changes, regardless of what the best way to make the change would be, because they were so worried about how hard it would be to repair the problem if they "broke the system."

Lack of trust

Initially, there was a lack of trust between the sites, because the developers worried that their site would be shut down, leading to a reluctance to

share information. The two sites did not see themselves as partners, cooperating toward the same end. This manifested itself in "uncharitable" interpretations of behavior. For example, if someone said, "we can't make that change," it was often interpreted as, "we don't find it convenient to make that change."

This started to improve when about six people from the UK visited the German site for significant periods of time to work on integration difficulties. After working together, the relationships between the sites began to change. One developer said, "things eased a lot when we met these people face to face, instead of over telephones and e-mail. We worked more closely and resolved things much quicker."

Several factors contributed to the change. The differences in cultures were seen in context and became less mystifying. It also let developers see that both sites were struggling to meet a very aggressive schedule. Working face to face let the developers establish a set of common goals and purposes. Finally, the time spent at the other site familiarized each party with the terminology and problem-solving style of the other.

As a result, the developers interpreted behavior more charitably. Rather than assuming the other site's position in a disagreement was purely arbitrary, each site was more likely to believe that others had reasons for their positions. Furthermore, when disagreements arose, the developers were more able to understand each other, and as a result, find common solutions.

Overcoming Distance

Studying the challenges of multisite development has revealed the key roles both of coordination mechanisms (such as architecture, plan, and process) and informal communication in coordinating software development work. The most effective approaches to overcoming distance will have to address both parts of the equation.

We believe that the qualitative evidence from our case study strongly supports Conway's and Parnas' positions that the essence of good design is facilitating coordination among developers. Geographic distribution is just an extreme case where coordination is more difficult, and good design is correspondingly more important.

Good design is vital, but it is not enough. It is also essential to coordinate when, how, where, and by whom the product will be developed. For example, when 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. Just as architectures play a vital role in coordination, so do project plans, defined processes, and staffing profiles.

Stability of the design is also important for multisite coordination. If you assign work to different teams and sites on the basis of an architecture that is constantly changing, the benefits of modularity might be completely lost as interfaces, functionality, and project commitments are continually renegotiated. In fact, Conway pointed this out long ago when he noted that you can only optimize the organizational arrangements with respect to "the system concept in effect at that time."[1] This observation applies not only to the architecture, but also to the plan, the processes, and all of the coordination mechanisms. Instability creates an enormous need for communication, which is precisely what distributed organizations do least well.

Good design is vital, but good design is not enough.

The results of this case study offer us several lessons for multisite development. The first three lessons will help to reduce the need for cross-site communication:

- Attend to Conway's Law: Have a good, modular design and use it as the basis for assigning work to different sites. The more cleanly separated the modules, the more likely the organization can successfully develop them at different sites.
- To the extent possible, only split the development of well-understood products where architectures, plans, and processes are likely to be stable. Instability will greatly increase the need for communication.
- Record decisions and make sure this documentation is easily available. In particular, documenting specification refinements and decisions reached in multisite meetings will save many troublesome misunderstandings.

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

♦ Front-load travel: Don't postpone using the travel budget; bring people who will 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 travel 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 and be effective in this role. When they return, make it known that they can help with cross-site issues, and free up some of their time to do so.

♦ Invest in tools that address the real problems. While video conferencing, desktop video, electronic bulletin boards, and workflow applications might add value in some circumstances, they do not directly address the core problems we observed. Distributed organizations desperately need tools that make it easier to find organizational information, to maintain awareness about the availability of people, and to have more effective cross-site meetings, especially spontaneous ad hoc sessions.

With vision obscured by perfect hindsight, it is easy to conclude that all of the coordination problems reported in this case study could have been prevented. Of course, drastically reducing unexpected events would be an enormous help, and much software engineering research is devoted to just that goal. But progress in cost and schedule estimation, software architectures, software processes, and verification techniques—disciplines that would make development more predictable—is slow at best, and project decisions must often be made very quickly, on limited information. For the foreseeable future, projects must continue to cope with unanticipated events that have significant consequences. Effective channels for informal, ad hoc communication will continue to be a critical organizational capability for adjusting quickly to the unexpected, recovering from errors, and managing the effects of change. ❖

Acknowledgments

We thank the Lucent Technologies department for their time and patience with our questions.

References

1. M.E. Conway, "How Do Committees Invent?" *Datamation*, Vol. 14, No. 4, Apr. 1968, pp. 28–31.
2. D.L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Comm. ACM*, Vol. 15, No. 12, 1972, pp. 1053–1058.
3. B. Curtis, H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Comm. ACM*, Vol. 31, No. 11, 1988, pp. 1268–1287.
4. R.E. Grinter, "Recomposition: Putting It All Back Together Again," *Proc. ACM Conf. Computer Supported Cooperative Work*, ACM Press, New York, 1998, pp. 393–403.
5. R.E. Kraut and L.A. Streeter, "Coordination in Software Development," *Comm. ACM*, Vol. 38, No. 3, 1995, pp. 69–81.
6. D.E. Perry, N.A. Staudenmayer, and L.G. Votta, "People, Organizations, and Process Improvement," *IEEE Software*, July/Aug. 1994, pp. 36–45.
7. J.D. Herbsleb and R.E. Grinter, "Splitting the Organization and Integrating the Code: Conway's Law Revisited," *Proc. Int'l Conf. Software Eng.*, ACM Press, New York, 1999, pp 85–95.

About the Authors

James D. Herbsleb is a member of the technical staff in the Software Production Research department at Bell Labs, Lucent Technologies. He is currently leading a Bell Labs research project that is developing tools, practices, and organizational models to address the problems of globally-distributed software engineering. In addition to geographically distributed software development, his research interests include computer-supported cooperative work and empirical software engineering. He has an MS in computer science from the University of Michigan and a PhD in psychology from the University of Nebraska.

Rebecca E. Grinter is a member of the technical staff in the Software Production Research department at Bell Labs, Lucent Technologies. Her research interests include empirical studies of software development and computer-supported cooperative work. She received a BSc in computer science from the University of Leeds, England, and an MS, and PhD in computer science from the University of California, Irvine. Contact her at beki@research.bell-labs.com.

Readers can contact Herbsleb at Bell Laboratories, Lucent Technologies, 263 Shuman Blvd., Naperville, IL 60566; herbsleb@research.bell-labs.com.