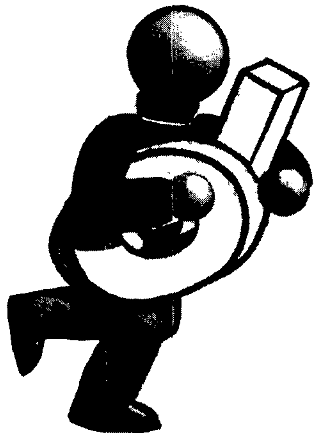
The problems of designing large software systems were studied through interviewing personnel from 17 large projects. A layered behavioral model is used to analyze how three of these problems—the thin spread of application domain knowledge, fluctuating and conflicting requirements, and communication bottlenecks and breakdowns—affected software productivity and quality through their impact on cognitive, social, and organizational processes.

A FIELD STUDY OF THE SOFTWARE DESIGN PROCESS FOR LARGE SYSTEMS



BILL CURTIS, HERB KRASNER, and NEIL ISCOE

THE NEED FOR ECOLOGICAL DATA IN TECHNOLOGY RESEARCH

MCC, The Microelectronics and Computer Technology Corporation, is a research consortium whose Software Technology Program was tasked by its member companies to create technology that dramatically improves software productivity and quality. This program has focused its research on the *upstream* portion of the software development process, since the empirical literature suggests that requirements and design decisions exert tremendous impact on software productivity, quality, and costs throughout the life cycle [35]. From the beginning, the program was committed to *problem-driven*, rather than technology-driven, research [46]. To pursue problem-driven research, an empirical studies group was established to assess the upstream factors in our member companies' development environments that reduced software productivity and quality.

Some members of our team have been proponents of quantitative and experimental methods in software engineering research [18, 20–22]. We judged these methods insufficient, however, for providing insight into our member companies' problems early enough to support a large, focused technology research program. Accordingly, we employed *field research* methods characteristic of sociology and anthropology [12]. The need for expedient results dictated the short, intensive study of a broad cross-section of projects, rather than the longitudinal study of a single project. In a similar field study, Zelkowitz, Yeh, Hamlet, Gannon, and Basili [63] identified discrepancies between the state of the art and the state of practice in using software engineering tools and methods. The data we collected lend themselves to creating the case studies often recommended for use in research on software development projects [8, 54].

This field study of the software design process consisted of interviews with personnel on large system development projects. The interviews revealed each project's design activities from the perspectives of those whose actions constituted the process. Our interviews provided detailed descriptions of development problems to help identify high-leverage factors for improving such processes as problem formulation, requirements definition and analysis, and software architectural design. We focused on how requirements and design decisions were made, represented, communicated, and changed, as well as how these decisions impacted subsequent development processes.

A LAYERED BEHAVIORAL MODEL OF SOFTWARE DEVELOPMENT PROCESSES

Studies by Walston and Felix [58], Boehm [9, 10], McGarry [43], and Vosburgh, Curtis, Wolverton, Albert, Malec, Hoben, and Liu [57] have demonstrated the substantial impact of *behavioral* (i.e, human and organizational) factors on software productivity. The effects of tools and methods were relatively small in these studies. For instance, rather than the sizable gains often promised, Card, McGarry, and Page [16] found that applying a collection of software engineering technologies to actual projects had only a 30 percent impact on reliability and none on productivity. To create software development technology that dramatically improves project outcomes, Weinberg [61], Scacchi [51], and DeMarco and Lister [24] argue that we must understand how human and organizational factors affect the execution of software development tasks. Nevertheless, Weinberg warned that "the idea of the programmer as a human being is not going to appeal to certain types of people" [61 p. 279].

For instance, software design is often described as a *problem-solving* activity. Nevertheless, few software development models include process components identified in empirical research on design problem-solving [2, 31, 32, 34, 36, 42]. Even worse, software tools and practices conceived to aid individual activities often do not provide benefits that scale up on large projects to overcome the impact of team and organizational factors that affect the design process.

Our study differs from the quantitative studies we cited earlier by describing the processes and mechanisms through which productivity and quality factors operate, rather than developing a quantitative assessment of their impact. These descriptions support our need to understand how different tools, methods, practices, and other factors actually affect the processes that control software productivity and quality. Since large software systems are still generated by humans rather than machines, their creation must be analyzed as a *behavioral* process. In fact, software development should be studied at several behavioral levels [40], as indicated in the *layered behavioral model* presented in Figure 1. This model emphasizes factors that affect psychological, social, and organizational processes, in order to clarify how they subsequently affect productivity and quality.

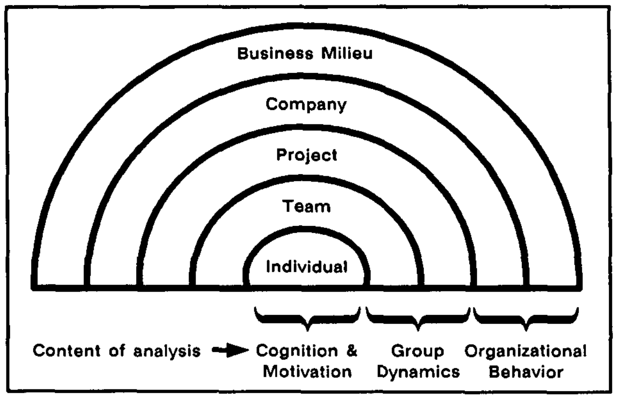


FIGURE 1. The layered behavioral model of software development.

The layered behavioral model focuses on the behavior of those creating the artifact, rather than on the evolutionary behavior of the artifact through its developmental stages. At the individual level, software development is analyzed as an intellectual task subject to the effects of cognitive and motivational processes. When the development task exceeds the capacity of a single software engineer, a team is convened and social processes interact with cognitive and motivational processes in performing technical work. In larger projects, several teams must integrate their work on different parts of the system, and *inter*team group dynamics are added on top of *intra*team group dynamics. Projects must be aligned with company goals and are affected by corporate politics, culture, and procedures. Thus, a project's behavior must be interpreted within the context of its corporate environment. Interaction with other corporations either as co-contractors or as customers introduces external influences from the business milieu. These cumulative effects can be represented in the layered behavioral model. The size and structure of the project determines how much influence each layer has on the development process.

The layered behavioral model is an abstraction for organizing the behavioral analysis of large software projects. It encourages thinking about a software project as a system with multiple levels of analysis. This model does not replace traditional process models of software development, but rather organizes supplementary process analyses. This model is orthogonal to traditional process models by presenting a cross-section of the behavior on a project during any selected development phase. Describing how software development problems affect processes at different behavioral levels indicates how these problems ripple through a project [51]. The layered behavioral model encourages researchers to extend their evaluation of software engineering practices from individuals to teams and projects, to determine if the aggregate individual level impacts scale-up to an impact on programming-in-the-large.

SAMPLE AND ANALYSIS

Sample and Study Procedures

Candidate projects were identified by each company's liaison to the MCC Software Technology Program in conjunction with company management. These industries were in businesses such as computer manufacturing, telecommunications, consumer electronics, and aerospace. Originally, we wanted to study projects that:

- involved at least 10 people
- were past the design phase but not yet delivered
- involved real-time, distributed, or embedded applications

Most projects selected conformed to some, but not all, of these criteria, and the deviations provided a richer set of project types to study. Rather than provide only successful projects, companies were willing to let us interview a project that had been terminated and several others that had been resurrected from failures. Nevertheless, we make no claim that this is a random sample.

From May through August, 1986, we visited 19 projects from nine companies. Two projects were actually programming teams embedded in larger projects and were dropped from this analysis. Prior to each site visit the project manager had completed a brief form describing project characteristics, and these are summarized in Table I. These projects varied in the:

- stage of development (early requirements definition through maintenance)
- size of the delivered system (24K to an estimated 100M lines of code)
- application domain (operating systems; transaction processing; communications, command, and control [C^3]; avionics)
- key project/system attributes (e.g., real-time, distributed, embedded, and defense)

We conducted structured interviews approximately one hour long on site with systems engineers, senior software designers, and the project manager. On about one-third of the projects, we were able to interview the division general manager, customer representatives, and the testing or quality assurance team leader. Participants were guaranteed anonymity, and the information reported has been sanitized so that no individual person, project, or company can be identified. The methods we used in creating questions and conducting these interviews with participants are described in Appendix A, along with a discussion of salient methodological issues regarding interview data.

Although all interviews were recorded, we offered to turn off the tape recorder any time the participant wished. One participant requested that the interview not be recorded, and several others requested that the recorder be turned off briefly while describing supervisors. Several participants, most often senior system engineers, requested that their tapes be played for senior management. Tape recordings of the 97 interviews yielded more than 3,000 pages of transcripts.

TABLE I. Characteristics of the 17 Field Study Projects

			Characteristics				
Project	**Stage of Life Cycle**	**KLOC**	**Real-time**	**Distributed System**	**Embedded System**	**Defense**	**Application**
1	Terminated	—					Support Software
2	Development	24	✓		✓		Radio Control
3	Development	50	✓	✓	✓		Process Control
4	Development	50	✓				Operating System
5	Design	70				✓	CAD
6	Development	130					CAD
7	Development	150+	✓		✓	✓	Avionics
8	Maintenance	194	✓			✓	C^3
9	Development	200					Compiler
10	Maintenance	250					Run-time Library
11	Development	350+					Compiler
12	Maintenance	400					Transaction Proc.
13	Design	500	✓	✓			Telephony
14	Maintenance	725					Operating System
15	Development	1000	✓	✓			Telephony
16	Maintenance	50K+	✓	✓	✓	✓	Radar, C^3
17	Requirements	100K	✓	✓	✓	✓	C^3, Life Support

Analysis of the Interviews

Analysis of the interview transcripts revealed the processes that underlie a number of classic software development problems. We took a two-pronged approach in our analysis. In a top-down approach, we built models of the important processes described in the interviews. In a bottom-up approach, using projects that presented particularly crisp case studies, we wrote summaries of process-related issues from individual interviews and then synthesized summaries for the project. We clustered the problems into several areas we heard repeatedly across different projects. The three most salient problems, in terms of the additional effort or mistakes attributed to them, were:

(1) the thin spread of application domain knowledge
(2) fluctuating and conflicting requirements
(3) communication and coordination breakdowns

We distinguished among these three problems because they operate through different mechanisms and may require different solutions. Each problem typically emerged from processes at one level of the layered behavioral model, but affected processes at several levels. For instance, the thin spread of application knowledge was a cognitive issue, while fluctuating requirements normally resulted from conditions in the business milieu. Communication breakdowns, however, could occur at any process level. The effects of these problems were not independent. For instance, fluctuating requirements increased a development team's need for communication both with customers and with the project's other teams.

A section on each problem will begin with discussion at the behavioral level whose processes formed the problem's primary mechanism. We will then describe how it rippled through a software project by affecting processes at other levels. We will illustrate these descriptions with sanitized quotes from the field study transcripts.

THE THIN SPREAD OF APPLICATION DOMAIN KNOWLEDGE

The deep application-specific knowledge required to successfully build most large, complex systems was thinly spread through many software development staffs. Although individual staff members understood different components of the application, the deep integration of various knowledge domains required to integrate the design of a large, complex system was a scarcer attribute. This problem was especially characteristic of projects where software was embedded in a larger system (e.g., avionics or telephony), or where the software implemented a specific application function (e.g., transaction processing). These systems contrast with applications currently taught in computer science departments, like single processor operating systems and compilers. Although most software developers were knowledgeable in the computational structures and techniques of computer science, many began their career as novices in the application domains that constituted their company's business. As a result, software development required a substantial time commitment to learning the application domain.

> System engineer: Writing code isn't the problem, understanding the problem is the problem.

Many forms of information had to be integrated to understand an application domain. For instance, project members had to learn how the system would behave under extreme conditions such as a jet fighter entering battle at night during bad weather, a telephone switch undergoing peak load on Mother's Day, or an automated factory with machines running at different speeds. Software developers had to learn and integrate knowledge about diverse areas such as the capabilities of the total system, the architecture of a special-purpose embedded computer (often a microprocessor), application-specific algorithms, the structure of the data to be processed and how it reflected the structure of objects and processes in the application domain, and occasionally even more esoteric knowledge about how different users performed specific tasks.

Individual Level

Project managers and division vice presidents consistently commented on how differences in individual talents and skills affected project performance. These observations were consistent with earlier differences observed in software productivity studies [9, 19, 43]. Individual performance is a combination of motivation, aptitude, and experience; where experience often consists of disorganized education acquired on-the-job.

Some performance differences were determined by how deeply programmers understood the application for which they were writing programs. Specification mistakes often occurred when designers did not have sufficient application knowledge to interpret the customer's intentions from the requirements statement. Customer representatives and system engineers complained that implementations had to be changed because development teams had misconceptions of the application domain.

> Customer representative: They didn't have enough people who understood warfare to assess what a war actually meant. When we say we're going to use this system to . . . search areas, . . . [they] thought you do it with a fixed geometric method. Whereas I had to explain you don't, . . . it's always relative to the kind of force you are protecting. Suddenly, that becomes a whole different problem.

Many projects had one or two people, usually senior system engineers, who assumed prime responsibility for designing the system. On about one-third of the projects we studied, one of these individuals had remarkable control over project direction and outcome, and in some cases was described by others as the person who "saved" the system. Since their superior application domain knowledge contrasted with that of their development colleagues, truly *exceptional designers* stood out in this study, as they have elsewhere [15, 17], as a scarce project resource. Thus, the unevenness with which application-specific knowledge was spread across project personnel was a major contributor to the phenomena of *project gurus*. Although our primary orientation in the field study had been to study organizational processes, we could not escape the impact of these differences in individual design talent.

Exceptional designers performed broader roles than design [44], and were recognized as the intellectual core of the project (i.e., the keeper of the project vision) by other project members. As part of this central role, exceptional designers provided us with the richest insight into the design process. Their understanding of both customers and developers allowed them to integrate different, sometimes competing, perspectives on the development process.

> System engineer: The people that seem to be really gifted at this sometimes seem to have . . . an understanding of the market voice—even though they're not always in touch with the customer—and can understand what makes sense. . . . Lots of what we work with is a hundred million ordinary people out there. I sort of relate to them and how they'll react.

Three characteristics appeared to set exceptional designers apart from their colleagues. First, exceptional designers were extremely familiar with the application domain. Their crucial contribution was their ability to map between the behavior required of the application system and the computational structures that implemented this behavior, shown in Figure 2. In particular, they envisioned how the design would generate the system behavior customers expected, even under exceptional circumstances. Yet exceptional designers often admitted that they were not good programmers, indicating they did not write optimized code, if they wrote code at all.

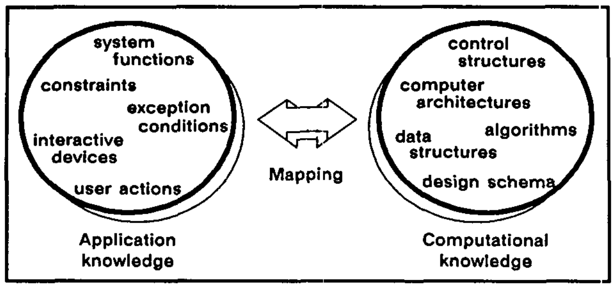


FIGURE 2. The expertise of exceptional designers.

Exceptional designers were often described as interdisciplinary, since they integrated several knowledge domains that constituted the application domain. The volume of application domain knowledge and lack of good domain models are serious obstacles in current automatic programming systems [6]. For large embedded systems these problems are complicated by the number of domains that must be integrated. For instance, designing military avionics software might require expertise in flight control, navigation, sensor-data processing, electronic countermeasures, and target acquisition.

System engineer: It is one of the underlying main problems, . . . not having enough of the system-level thinkers . . . to coordinate the thinking of the people who don't think on a system level. . . . There aren't enough system-level thinkers to go around, even to do the quotes. . . . It's what people are paying attention to, what are their hot-buttons. You get the bit bangers who are only interested about bits. . . . [Systems thinkers] are not looking at the computer as the end-all and be-all of the problem. It's just one more of the objects that they have to deal with.

Although a project might have experts in each of its functional domains, these experts were often unable to model the effect of component integration on processing or storage constraints. Exceptional designers were skilled at modeling the interaction of a system's different functional components, and occasionally developed notations for representing them. Exceptional designers were also adept at identifying unstated requirements, constraints, or exception conditions.

System engineer: One of the things I do best is model the real world within our database, . . . and we always have the same problem, "This is what you want to model? Well, you've got this little hickey they didn't tell you about." . . . Most people cannot model, . . . it just requires an ability to abstract.

Second, exceptional designers were skilled at communicating their technical vision to other project members. They usually possessed exceptional communication skills [30] and often spent much of their time educating others about the application domain and its mapping into computational structures. In fact, much of their design work was accomplished while interacting with others. Weinberg suggests that the integrative role of an exceptional designer compounds itself. This happens because those perceived as most knowledgeable will become communication focal points, providing them more knowledge about the system to integrate into a more comprehensive model.

Third, exceptional designers usually became consumed with the performance of their projects. They were a primary source of coordination among project members and assumed, without formal recognition, many management responsibilities for ensuring technical progress. They frequently internalized the pressures of the project because of their identification with its success. Although not part of our original focus, we became sensitive to the health risks of stress on crucial project personnel and the business risks that can result.

Conventional wisdom on software development often argues that no software project should rely on the performance of a few individuals. The experience of many successful large projects, however, indicates why this reliance is more troublesome in theory than in practice. An exceptional designer represents a crucial depth and integration of knowledge domains that are arduous to attain through a group design process. Under severe schedule constraints, groups may be unable to achieve the level of knowledge integration required to develop a cohesive architecture and design strategy [14].

Broad application knowledge was acquired more through relevant experience than through training, since little training was provided for integrating technical domains. Developing design skill required the right project assignments, since some large system development lessons could not be acquired through classroom instruction or on small projects. Thus, the substantial cost of developing talented large system designers is part of the cost of developing large systems.

System engineer: Someone had to spend a hundred million to put that knowledge in my head. It didn't come free.

Team Level

Although the thin spread of application domain knowledge is a cognitive issue, it had impact on processes, such as decision-making, occurring at the team level and above. Owing to the broad skill ranges on design teams, expert power [28], meaning the ability to influence a group through superior knowledge, appeared to be the most effective means of exercising authority during many parts of the design process. Group decision-making researchers have generally not studied teams on long duration activities like system design, where the quality of the result is difficult to measure [26, 33]. Therefore, theoretical models of group decision-making may not describe the behavior of large system design teams.

If we were to construct a simple participative, consensus-oriented model of the team design process, we might begin with team members holding their own, often partial, models of the system's structure. These individual models usually differ in their representation of factors such as the application system's external behavior, the environmental context in which it will operate, or the most appropriate computational model. In the second stage, individuals sharing similar models would form *coalitions* to argue for their architectural position. In the final stage, the technical differences between coalitions would be resolved into a *team consensus.* Belady [7] observed similar processes within Japanese design teams.

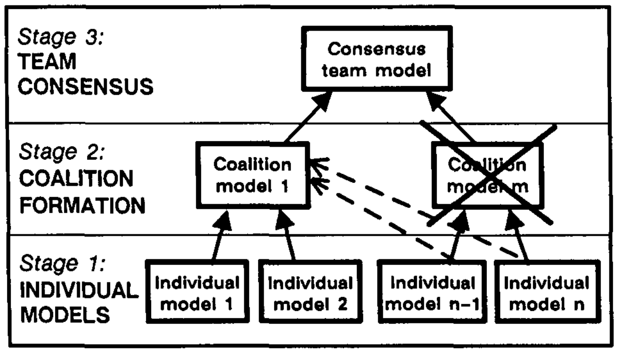


FIGURE 3. Small coalitions often coopt the design process.

In contrast to the model just outlined, the early phases of most projects in our study were dominated by a small coalition of individuals, occasionally even a single individual (the exceptional designer) who took control of the project's direction. Members of the dominant coalition usually knew most about the application, or had previous experience that made them quick studies. When all team members were from the same corporate division, competing coalitions were reported much less often than we had expected. As Figure 3 shows, competing coalitions were difficult to form because a dominant coalition's speed in formulating a design made catch-up by late-forming coalitions difficult. Further, alternatives were usually debated in terms of the architectural foundation already proposed.

> System engineer: I tried an experiment last summer and said, "What would happen if I just sort of was agreeable to a certain extent with [a colleague]." Ever noticed in a meeting where there's 15 people and there's 15 points of view, a majority is only two. Two people say the same thing and everything moves forward. . . . I think we pulled off an incredible project in a very short time by that relationship. He lets me win sometimes and I let him win sometimes, and the game goes on.

Competing coalitions occurred more often on teams formed with representatives from several different companies. For single company projects, competing coalitions formed primarily when the design team consisted of members from different organizational divisions. Coalitions based on organizational allegiances often resulted from differences in each organization's model of the application (discussed under Business Milieu).

These observations do not imply that teams are unimportant during design. Videotaped observations of a design team in our laboratory [59] suggested that teams composed of members from different technical areas were better at exploring design decisions in breadth, rather than depth, by posing alternatives and constraints and by challenging assumptions. Thus, design directions set by a small coalition may benefit when challenged by colleagues who may never gather enough support to form a coalition. Forming a competing coalition requires considerable effort to generate support for an alternate proposal among colleagues. Rather than being only a matter of technical argumentation, forming an alternate coalition requires a social process of mobilizing support.

Project Level

When application knowledge was thinly spread, it was necessary to ensure that the design and development teams shared a model of the system's operation. A system model is actually an integrated collection of models. One potential set of relationships among components of a system model for a hypothetical project is illustrated in Figure 4. The relevant components and their relationships may vary by system. Most project personnel were knowledgeable in one or two of the areas represented by circles in the diagram. Those who focused on the system architecture, however, were best positioned to integrate application and computational

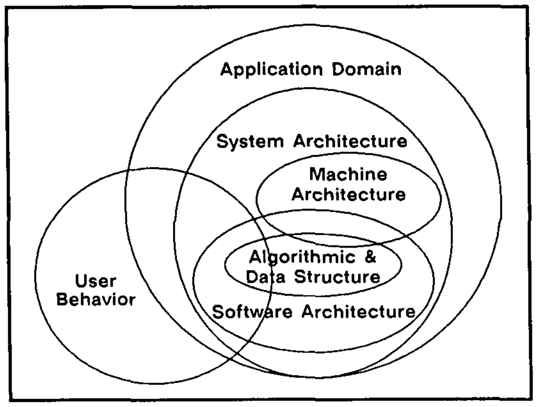


FIGURE 4. Knowledge domains involved in system building.

knowledge, and to oversee the functional integration of the design. System engineers used many methods to integrate a project, ranging from gentle persuasion to aggressive steamrolling.

System engineer: We create a project management group with about five or six people that can do anything. When somebody's not doing something, we roll in there and start doing it and get it structured just the way we want it. Then they get mad and say, "It's our job," but by then it's all structured and we back out and throw our resources somewhere else, and get something else going just the way we want it.

The time devoted to learning and coordinating application-specific information was initially buried within the design phase and could not be accounted for separately. Learning costs were paid for in several ways: in planned training, in exposure to customers, in prototypes and simulations, in defects, in budget or schedule overruns, and in canceled projects. Customers were usually unwilling to pay for training since they believed the contractor should already have the required knowledge. Thus, the time required for design was often seriously underestimated, since these estimates were usually based only on the time actually spent designing. The time spent educating project personnel about the application domain and coordinating their understanding of the system was overlooked.

We were tempted to conclude that the best prototype was a failed effort. We interviewed several highly productive projects that had emerged from the ashes of failed architectures, and heard several citations of Brooks' [14] admonition to "plan to throw one away." These *phoenix projects* occurred when exceptional designers had immersed themselves in enough of the application and computational problems of their architecture to recast their vision of the system. A rapidly developed prototype that missed the problems uncovered in an unsuccessful architecture would not have provided the required insight. To be effective, prototypes must be sufficiently comprehensive for misunderstood requirements or subtle system problems to present themselves.

Company Level

The cost of learning an application area was a significant corporate expense. The time estimated for a new project assignee to become productive ranged from six months to a year. Major changes in the business application or in the underlying technology required additional learning. As the technical staff's application knowledge matured, however, the organization usually increased its ability to reduce project cost and schedule, and increase productivity and quality.

System engineer: If you look at the evolution of this place . . . over the course of three or four years—at the beginning the most important thing you could be . . . as an engineer was somebody who knew the operating system internals. We're now making the transition to the most important thing . . . is understanding the application. That's really where our bread and butter is. For a long time we could never keep engineers focused on what they were supposed to be doing here.

Companies were affected by the migration of technical talent into management and by whether management decisions were based on current knowledge of technical issues. If a business' software applications and related technology were stable, a manager's previous technical experience provided an adequate basis for decisions. However, major changes eroded the value of a manager's technical knowledge for making decisions, especially those that involved technical tradeoffs. Some managers were frequently unable to participate in the technical meetings (e.g., requirements analysis, design reviews) that provided training for their project team. The contribution of previous technical knowledge grew more remote as managers were promoted beyond first line management.

Although most managers had developed progress tracking schemes, many were less aware of system status than were their system engineers. On extremely large projects, middle managers expressed frustration at being removed both from the technical decisions made by engineers and from the strategic decisions made by executives. Some software managers had difficulty articulating their role in the project and had no company source for advice or training on better development tools and practices.

Programmer: The way the managers are getting trained is that the engineers are coming back [from software engineering courses] and are fighting to keep using some of the tools and techniques they've learned; and fighting against the managers to let them use them; and that's really how the managers are getting their experience.

A major challenge to most managers was to assess the limits of their staff's capability and its impact on producing a successful system. An implicit component of their job was to close the gap between the technical challenges of the system and their staff's capability for solving them. They also had to assess the claims made by staff members about their own abilities and about how long it would take them to perform a task.

Business Milieu

When several companies cooperated in building a system, the separation imposed by organizational boundaries hindered their shared understanding of the application and the system architecture. Competing coalitions in multicompany design teams formed along company boundaries and clashed over assumptions about market applications or system functionality that were unique to their business or product lines. These differences frequently caused co-contractors to try to push the hard problems into each other's component as they negotiated the requirements and specifications.

System engineer: You . . . minimized your own problems and maximized theirs. What it boiled down to was . . . a big finger pointing contest.

The coordination process was more complicated on multicompany projects than on single company projects, because each company understood the application domain in the context of its own product lines. Software contractors often took responsibility for coordinating design decisions because they had to architect the system's behavior. Technical coordination required a long dialectic among co-contractors both for surfacing assumptions and for resolving misunderstandings.

System engineer: We had three different simulators all coming up with different answers. . . . None of them reflected the same reality because they were all using their own preconceived notions. . . . The human factor definitely played a role. . . . We spent many a day in trying to figure out what the assumptions were of the three different simulations, saying, "No, you can't do it that way, go back and do it this way."

Customers often believed that the software contractor should be the prime contractor for a system since the software team had the greatest need to understand the details of the customer's application environment. Yet the software contract might involve as little as one-tenth of the total project cost, since the largest cost involved building multiple versions of the hardware. Software contractors were unwilling to assume the financial risk of the total project when they received such a small percent of the contract's value. Forcing the customer and the software designer to communicate through the hardware contractor limited the software team's ability to learn about the application domain. It also hindered the customer's ability to negotiate small, but necessary, corrections to the software requirements.

Application Domain Knowledge Summary

Our interviews revealed that the thin spread of application knowledge among the project staff was a significant problem on many software development projects. This problem initially manifested itself at the individual level and underlay the phenomenon of the project guru, an exceptional designer who could map deep application knowledge into a computational architecture. Those with this skill exerted extraordinary influence over the direction of the design team, and the formation of effective coalitions supporting alternate proposals happened less often than expected. Substantial design effort was spent coordinating a common understanding among the staff of both the application domain and of how the system should perform within it. Periodic changes in the application domain or in the supporting technology reduced a company's technical maturity and weakened its foundation for sound management decisions. Multicompany development efforts had to overcome company-specific models of the application domain and their translation into system functionality. Aggregating these issues across behavioral levels points to the importance of managing learning, especially of the application domain, as a major factor in productivity, quality, and costs.

FLUCTUATING AND CONFLICTING REQUIREMENTS

Fluctuation or conflict among system requirements caused problems on every large project we interviewed. For example, we visited one gargantuan system that was being acquired in separate components, each involving competitive bidding among corporations. On the day we interviewed the proposal team, the customer announced a realignment of functional components across different bidding competitions. We found team members gathered around a newspaper clipping and other, more official postings on the bulletin board, in an attempt to determine which of their designed artifacts could still be included in their proposal. On another project, we were told that hardware changes could cause a redesign of the software every six months.

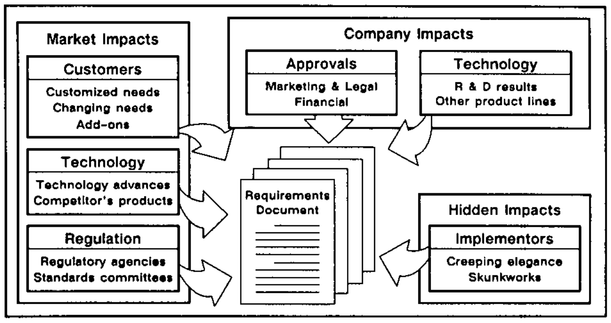


FIGURE 5. Sources of fluctuating and conflicting requirements.

Requirements will appear to fluctuate when the development team lacks application knowledge and performs an incomplete analysis of the requirements. Now we will concentrate on sources of fluctuation and conflict that were external to the design team. A variety of events caused volatility and conflict in product requirements, as shown in Figure 5, including such market factors as technological advances, competitive products, regulatory constraints, standards committees, and such internal company factors as corporate politics, marketing plans, research results, and financial conditions. Less visible within the project were the hidden effects on the requirements such as skunkworks (work hidden by managers) and creeping elegance. Since the primary sources of fluctuating and conflicting requirements existed in the company and the business milieu, we begin our discussion at these levels.

Business Milieu

Product requirements fluctuated most frequently when different customers had separate needs or when the needs of a single customer changed over time. Analyzing requirements for commercial products was difficult without an explicit statement of at least one customer's needs. The requirements were often defined for the first customer to place an order, even though project personnel knew that other customers would state different requirements. During development, designers tried to raise the product specification from the specific (driven by a single customer) to the general (driven by a market of customers), although it often continued to evolve from the specific to the specific.

> Software architect: The whole software architecture, to begin with, was designed around one customer that was going to buy a couple of thousand of these. And it wasn't really designed around the . . . marketplace at all. Another . . . customer had another need, so we're trying to rearrange the software to take care of these two customers. And when the third one comes along, we do the same thing. And when the fourth one comes along, we do the same thing.

Even when a customized system was developed for one client, the requirements often provided a moving target for designers. During system development, the customer, as well as the developer, learned about the application domain. The dialectic through which the developer generated the requirements revealed new possibilities to the customer [29]. As customers learned more about the system's capability and understood their application better, they envisioned many features they wished they had included in the requirements.

> Project manager: Planned is probably a generous term, . . . an englightenment occurs as they move forward.

Many customers misunderstood the tradeoffs between requested functions, the capabilities of existing technology, the delivery schedule, and the cost. They learned of these tradeoffs through an iterative negotiation with the system design team, as the requirements were translated into a design and costs could be estimated. Each cycle is driven by trying to balance and integrate technical and non-technical constraints into the product requirements.

> System engineer: The original proposal was rejected because it was not as all-encompassing as they had originally perceived [the] system ought to be. So we made it bigger. Then it was too costly. So we scaled it down. It went through over 20 versions. It keeps expanding and contracting until it cools. It's like the earth.

Customers rarely understood the complexity of the development process and often requested frequent changes to the requirements. They underestimated the effort required to re-engineer the software, especially when the system involved tight timing or storage constraints. They rarely understood the impacts that rippled through the software when changes were made and the coordination required to document and test these changes. As a result, customers could not understand why changes to the requirements were so costly.

When customers had access to the development team, they often requested additions to the requirements without going through a formal change review process. Thus, the requirements were often unstable in ways that were not visible to project management.

> Customer representative: We like to be in among the contractors, assisting where we can, getting early decisions where necessary, and at the same time trying to talk them into enhancements we didn't pay for.

Government customers used the requirements statement as the basis for obtaining competitive bids. They tried to ensure that all competitors received identical information, regardless of whether it was in the requirements statement or in answers to questions. Making the competition fair to all bidders often clashed with the need to clarify ambiguities or omissions, and answers to questions might be oblique. As a result, bidders were forced to make assumptions about requirements that might later have to be changed.

Requirements also fluctuated when approvals had to be obtained from a government regulatory agency. An agency could create design constraints in the form of new requirements that differed from, and occasionally contradicted, those received from customers. The requirements could also change, based on regulatory evaluations of a completed design.

> Vice president: There were changes being driven by [a government agency's] considerations. . . . We were so used to working hard on a technical decision, . . . and here you had all of a sudden [a government agency] being your sounding board and you couldn't go anywhere until you heard from [a high-ranking government official].

Company Level

On projects producing commercial products, internal company groups, such as the marketing department, often acted as a customer. They could add conflict into requirements definition since their requirements occasionally differed from those of potential customers. A common tension occurred, for instance, when marketing wanted to redesign a system to take advantage of new technology, while existing customers did not want to lose their investment in software that ran on the current system. On several projects, the requirements—and even the understanding of the product—varied among strategic planning, marketing, and product planning groups. The design team had to reduce the conflict between these contending forces in their design. This conflict varied with how deeply groups such as marketing understood the customer's application and the limits of existing technology. Marketing groups understood why customers (who were not necessarily the users) would buy the system, but this often differed from the application-specific information about product use that was needed for design.

Software architect: Marketing came out with a description that had every single feature of every similar product and said, "Here do this," and they expected us to start writing software.

Resolving the conflicts among system requirements created a feedback cycle in which many groups provided inputs or constraints that had be negotiated into a design. Some of the toughest decisions involved tradeoffs between system features and the current market trends. Technical requirements were traded off against business decisions involving delivery dates and other marketing and sales issues.

Vice president: Even though quality and performance may suffer, it's better to have people using your stuff and complaining than to have them using somebody else's.

Project Level

Unstable requirements, when caused at the project level, usually resulted from the absence of a defined mission. Without a sense of mission the motivation for the project could not be translated into clear product requirements. When projects were started for political reasons rather than market demands, requirements fluctuated with the prevailing attitudes of those who approved funds. Such projects often reflected senior management's desire for large organizations under their authority. In such cases, product requirements were initially defined as those that would garner company funds, and the market's requirements were added retrospectively and had to be updated to justify the project.

System engineer: There's a big game that goes on to get giant projects started. You've got to figure out a way that everybody wins. I mean, development people want resources and big projects and long-term stability and something new and high-tech. The manager wants something with large revenue potential and something new and exciting to talk about. . . . You just have to know how to play all the angles. . . . We know how to do it, but we never wrote it down. We don't want to write it down.

Some large projects were started to exploit new, sophisticated technology in order to create a market. The requirements for supporting a new technology often conflicted with the needs of existing customers. In such cases, the project became the source for conflicting requirements that had to be resolved through managing product lines. In other cases, technical advances often threatened to make a technology obsolete before the system was delivered. Additional requirements were, therefore, levied on a product during development to compensate for the technology's growing market weaknesses. In such cases, developers were forced to emulate a post-release enhancement process before the product had been delivered.

Requirements were unstable when the initial project team was more interested in winning a procurement than in accurately estimating required costs and resources. In competitive procurements, some requirements analyses were driven toward producing a winning proposal rather than toward accurately portraying the size of the system and the effort required to build it. Requirements had to be readjusted when the technical and financial risk in developing the system became apparent.

Test engineer: They knew [the requirements were] inaccurate. . . . They were trying to competitively win, . . . so the requirements document looked an awful lot like a proposal. It was not adequate in any fashion to design from. . . . If that level of detail were opened, the customer would have understood and I don't think [we] would have won the follow-on.

A frequent conflict among requirements occurred when the functionality required of the system outstripped the processing or storage capacity of the specified hardware. In such cases, the *software crisis* was actually a symptom of a deeper crisis in the mismatch between the often explosive growth of requirements [10] compared to the limitations of available hardware. This crisis was accentuated when the risk and difficulty of resolving this conflict in architecting the software was not fully understood or accepted by either management or the customer.

Team Level

The design team had to clarify the conflicts among requirements and constraints generated both inside the company and in the marketplace. Resolving some conflicts required knowledge of actual user behavior that was scarce on some design teams. One solution was to design a flexible system that could be easily modified to accommodate future changes and technologies. To produce a flexible product on schedule and within hardware constraints, the requirements were rewritten by the design team to eliminate a smorgasbord of features and to require multiple alternatives for a few features.

Software architect: One of the pitfalls in our process occurs when . . . marketing, engineering, [and] development say, "Do we have to make the decision on how it's going to operate? Could you write it both ways?" We say, "Well, it's going to cost some resources but we could." The tendency is to not make the decision. . . . This leads to thinking that we can make everything flexible. In implementation we can do fewer things, because we are going to do each thing eight different ways.

Another solution to conflicting requirements was to prioritize them and include as many as possible in the specification in order of importance. This technique was effective when conflicts resulted from problems such as storage limitations. Even so, a consensus on the specifications was often difficult to build among development groups that had to accept and abide by the rankings. Even after priorities were negotiated, a consensus was hard to maintain without strong leadership to oversee adherence to priorities. When the primary constraint was schedule, the conflicts might be resolved by developing an implementation plan that phased in features across system releases.

> System engineer: The most difficult thing was allocating the features into memory, prioritizing and making the decisions, getting people to agree to what we are and are not putting in.

Even when the requirements were stable, specifications occasionally fluctuated because designs for different components were not tightly coordinated. In making a design decision, designers often made incorrect assumptions about how another group had interpreted a requirement. In such cases, a requirement was unstable not over time, but over different components of the system. Without tight coupling of interface decisions among components, inconsistencies became apparent only at integration time.

> Project manager: When we see problems it's often because they don't understand that you don't go build computer programs and build hardware and someday at the waterfront integrate them.

Unresolved design issues were a great concern for system engineers who lamented having no tools for capturing issues and tracking their status. The ratio of unresolved issues to the number of issues recorded may be a valuable indicator of design stability and actual progress in the design phase. Failure to resolve issues frequently did not become obvious until integration testing.

Individual Level
New requirements frequently emerged during development since they could not be identified until portions of the system had been designed or implemented. The need for some requirements could only be determined after the relevant questions had been posed. Designers also realized that many stated requirements were open to interpretation, and therefore, it was difficult to agree on the proper level of detail for specifying either the requirements or the design.

Many designers thought that requirements should act as a point of departure for clarifying poorly understood functions interactively with the customer. They argued that specifications should not be hardened while still learning about the application domain or the capabilities of the proposed architecture. That is, specification should not be formalized any faster than the rate of uncertainty about technical decisions is reduced.

> Customer representative: You will never really be able to specify enough detail. It doesn't matter how. You can even take the actual system and write the specs around it and still come out wrong. . . . The specifications are something you've got to take on trust.

A hidden source of instability in the requirements was the creeping elegance that occurred when programmers went beyond the stated requirements and continued to add system features. Even with strict controls on the growth of new code, managers were frustrated in trying to slow the spread of creeping features. These features constituted new requirements and were the bottom-up, programmer-driven counterpart to customer-driven requirements fluctuation. Most disturbing, their impact on project schedule and performance was often hidden from view.

> Quality assurer: We've had cases where people will fake an error in the system in order to be able to pull the code . . . so that they could replace it with a whole new implementation.

Fluctuating and Conflicting Requirements Summary
Fluctuation and conflict among requirements usually resulted from market factors such as differing needs among customers, the changing needs of a single customer, changes in underlying technologies or in competitors' products, and, as discussed earlier, from misunderstanding the application domain. Requirements problems could also emerge from such internal company sources as marketing, corporate politics, and product line management. When presented with the requirements statement, the design team often negotiated to reduce conflicts and limit requirements to those that could be implemented within schedule, budget, and technical constraints. Nevertheless, it was difficult to enforce agreements across teams, and programmers often created a hidden source of requirements fluctuation as they added unrequired enhancements. Although requirements were intended as a stable reference for implementation, many sources conspired, often unwittingly, to make this stability illusory. The communication and coordination processes within a project became crucial to coping with the incessant fluctuation and conflict among requirements.

COMMUNICATION AND COORDINATION BREAKDOWNS
A large number of groups had to coordinate their activities, or at least share information, during software development. Figure 6 presents some of the groups mentioned during interviews, clustered into behavioral layers according to their remoteness from communication with individual software engineers [56]. Remoteness involved the number of nodes in the formal communication channel that information had to pass through in order to link the two sources. The more nodes that information had to traverse before communication was established, the less likely communication

was to occur. This model implies that a software engineer would normally communicate most frequently with team members, slightly less frequently with other teams on the project, much less often with corporate groups, and, except for rare cases, very infrequently with external groups. Communication channels across these levels were often preconditioned to filter some messages (e.g., messages about the difficulty of making changes) and to alter the interpretation of others (e.g., messages about the actual needs of users). In addition to the hindrances from the formal communication structure, communication difficulties were also due to geographic separation, cultural differences, and environmental factors.

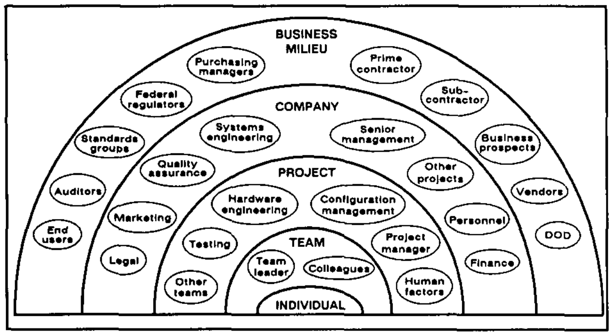


FIGURE 6. Remoteness of communications expressed in the layered behavioral model.

For example, communication at the team level mostly concerned system design, implementation, or personal issues. At the project level, proportionately more of the communication was related to coordinating technical activities and discussing constraints on the system. Communication at the company level generally concerned product attributes, progress, schedules, or resources. Communication with external organizations involved user requirements, contractual issues, operational performance, delivery planning, and future business. Thus, communication to each higher level involved a change in the content of the message, a different context for interpreting the message, and a more restricted channel for transmission (e.g., the more remote the level, the less the opportunity for face-to-face transmission).

Individual Level

Documentation is one form of communication among project members. Most interviewees, however, indicated frustration with the weakness of documentation as a communication medium. We found little evidence that documentation had reduced the amount of communication required among project personnel. Tardiness and incompleteness were not the only problems with documentation. Many required formats were insufficient for communicating some of the design information needed throughout the life cycle.

> Programmer: Our documentation is intended to be read; it is not "MIL-standards-like."

Documentation practices were usually vulnerable to other project pressures. For instance, as the size of the project grew, project members had to make a tradeoff between time devoted to communicating verbally with colleagues and time for recording written information for future project members. Many communication related activities that appeared to be good software engineering practices were almost unworkable when scaled up to support communication on large projects with deadline pressures.

> Programmer: I think this is the way it always turns out with this stupid design of large systems. In the beginning . . . there were 3 of us. How many lines of communication are there, 1, 2, 3? But once you go to 15 people it can get out of hand. . . . In the beginning, it was easy to keep track of what was going on. It was only after reaching the critical mass . . . that things began falling into the cracks, and we were losing track. . . . I used to religiously keep track of the [change notices], but now I don't think I've looked at them in 6 months. I just couldn't keep up with everything else going on. There was just so much going on.

Most project members had several nets of people they talked with to gather information on issues affecting their work [50]. Similar to communication structures observed in R&D laboratories [3, 4], each net might involve different sets of people and cross organizational boundaries. Each net supported a different flow of information, as shown in Figure 7. When used effectively, these sources helped coordinate dependencies among project members and supplemented their knowledge, thereby reducing learning time. Integrating information from these different sources was crucial to the performance of individual project members.

> System engineer: I get my requirements . . . by talking. I spend a third of my time talking with requirements people and helping them negotiate.

Team Level

The communication needs of teams were poorly served by written documentation since it could not provide

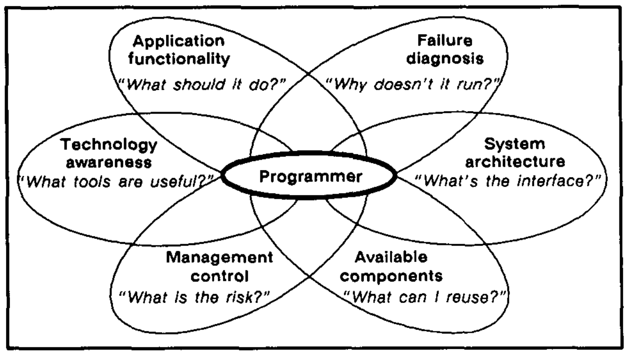


FIGURE 7. Examples of a programmer's communication nets.

the dialectic necessary to resolve misunderstandings about requirements or design decisions among project members. Rather, forging a common understanding of these issues required interaction.

System engineer: In the dynamics of the team there is only one way—verbal. . . . Paper disappears, it gets in a stack. I'm sure people read it, . . . but the ultimate method for managing requirements level activity with a small group of 10 or 20 people is 10 hours of meetings a day. And then you go work 5 hours.

Many techniques were used to organize and communicate a shared system model. Successful projects usually established common representational conventions to facilitate communication and to provide a common reference for discussing system issues. From a team perspective, this sort of representation was valuable as a common dialect for project argumentation, rather than as a basis for static documentation.

System engineer: The ER diagram means that everybody speaks the same language. Developers, designers, human performance people, we all use the same language. . . . It was 6 months or so before it settled down, but once it did, we could resolve all problems in terms of the diagram.

Once the development team had accepted common representational conventions (a process that could take six months or longer), its members could resolve disagreements and misunderstandings by referencing the structures in a diagram. System engineers were usually adamant about having the freedom to select a representational format that matched the application domain's structure. After selecting and tailoring the format, considerable effort was spent to establish agreement on diagrammatic conventions. In the early stages, disagreements over naming conventions could take as much time as did system decomposition.

System engineer: At least they know to carry around their dictionary when they talk to us. Being done with a phase of development, . . . what does "done" mean? We could never settle on that, so we settled on what "done done" means. The first "done" means internal done, and the second "done" means external done.

Project Level

Project managers often found it difficult to establish communication between project teams unless communication channels opened naturally. Since documentation did not provide sufficient communication, reviews were often the most effective channels. In fact, communication was often cited as a greater benefit of formal reviews than was their *official* purpose of finding defects. At other times, communication among teams was thwarted by managers for reasons that ranged from the politics of forging a lead over other teams to a lack of appreciation for coordination requirements.

Some communication breakdowns between project teams were avoided when one or more project members spanned team or organizational boundaries [1]. One type of *boundary spanner* was the chief system engineer who translated customer needs into terms understood by software developers. Boundary spanners translated information from a form used by one team into a form that could be used by other teams. Boundary spanners had good communication skills and a willingness to engage in constant face-to-face interaction; they often became hubs for the information networks that assisted a project's technical integration. In addition, they were often crucial in keeping communication channels open between rival groups.

System engineer: The parochial interest was a big deal. There was a lot of concern about loss of control over some aspect of the system and personality entered into that a lot. . . . Because I appeared relatively harmless to everybody in the organization, I didn't have any trouble moving back and forth from one group to the other. But there were times when [people would ask me], "When you're going to be talking to such-and-such would you please tell him to . . . ," that type of thing.

The social structure of the project was occasionally factored into architectural decisions. The system partitioning that reduced connectivity among components also affected communication among project personnel. Higher connectivity among components required more communication among developers to maintain agreed upon interface definitions. Occasionally, the partitioning was based not only on the logical connectivity among components, but also on the social connectivity among the staff.

System engineer: The real problem . . . was partitioning the system enough so we could minimize the interfaces required between people. In fact, it was more important to minimize the interfaces between system engineers than it was to make the system logical from the viewpoint of the user.

Company Level

Companies usually established formal processes for making and reviewing decisions about the design of large systems. These structures, however, were often ineffective for communicating design problems that arose in sections of the organization that were not part of the formal process. Rather, informal personal contacts were frequently the most effective way to transmit messages across organizational boundaries.

System engineer: The original impetus that I got to define something that could be used for all the machines came from, surprisingly enough, some member of the Board of Directors who is not an employee of the corporation, [and] who couldn't understand why we had different [computational platforms] . . .

Interviewer: You just knew him? How did you get the message from him?
System engineer: A dove descended with it. You know how it is.

When groups such as marketing, systems engineering, software development, quality assurance, and maintenance reported to different chains of command, they often failed to share enough information. This problem is not surprising in a government environment where security requires tactical information about a system to be classified. Yet, even on commercial projects information was occasionally withheld from a development group for reasons ranging from political advantage to product security.

Software architect: Even the product description for this project is a secret document for only people who need to know. I know that there were at least three revisions that I didn't hear about until 6 months after the fact. Some of the changes we are making now might have been avoided if we would have had earlier access to it.

Even when all of the interacting teams were in the same division, work on different phases (e.g., proposal, definition, development, delivery, and maintenance) of a large project was often performed by different groups. Figure 8 presents a typical work flow among teams responsible for different phases. Communication problems occurred in the transition between phases when groups transferred intermediate work products to succeeding groups. When a later redesign was undertaken and the previous design team had dispersed, these problems were exacerbated.

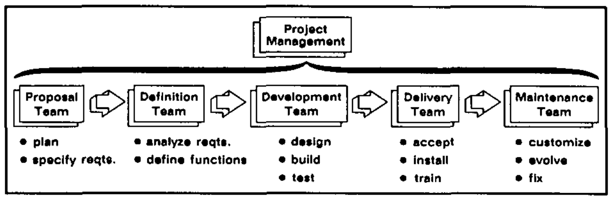


FIGURE 8. Teams responsible for different phases of the life cycle.

Although documentation was not accepted as an alternative to talking with colleagues, it was often the main source of communication between successive teams. Unfortunately, much of the needed information had not been recorded because of schedule pressures. Also, communicating system knowledge between these teams depended on continuing personnel from one team to the next.

System engineer: We didn't have enough documentation. We didn't have enough code review. We didn't have enough design review. . . . We're going to suffer because all the smart guys who developed the system are now going to leave . . . and what are the poor [expletive deleted] who have to maintain the system going to do? . . . How do you get our management to see that that's important and to give us the brownie points for doing it.

Business Milieu

Coordinating understanding of an application and its environment required constant communication between customers and developers. Developers had to clarify the meaning of terms and the associations between different objects or processes to avoid misinterpreting a requirement. Contact between customer and developer needed to be direct since intermediaries often had difficulty identifying the subtleties that had been misunderstood. This communication was required to establish a mutual frame of reference.

Customer representative: I think [we] had to learn as well as [the developers]. . . . At the time we wrote the specification, we did not appreciate that it could be interpreted any other way. . . . This particular thing was so obvious to me as an operator, you know, it's common knowledge. It's one of the basics you teach the uninitiated student. Everyone knows, . . . I should have known.

On most large projects, the customer interface was an *organizational communications* issue and this interface too often restricted opportunities for developers to talk with end users. For instance, the formal chain of communication between the developer and the end user was often made more remote by having to traverse two nodes involving the developer's marketing group and the end user's manager. At the same time this interface was often cluttered with communications from non-user components of the customer's organization, each with its particular concerns. Typically, development organizations said they would like to have, but could not get, a single point of customer contact for defining system requirements. None of the large projects we interviewed had a lone point of customer contact for defining requirements.

Often, the largest problem in managing a government contract involved the complexity of the customer interface [58]. This interface usually included many different agencies, each with a different agenda and each representing itself as *the customer.* Figure 9 depicts a

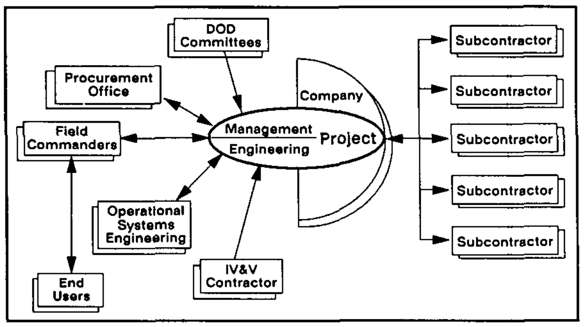


FIGURE 9. Customer interfaces on a DOD project.

simplified set of interfaces for a large contract with the Department of Defense (DOD). The interface might be comprised of organizations such as:

- senior DOD officials who championed the system
- the procurement office that tracks costs, schedules, and acceptance criteria
- an operational systems engineering group involved in specifying the system
- the commanders whose troops will use the equipment
- the actual operators of the equipment
- the Independent Validation and Verification (IV&V) contractor who inspects the software artifacts for DOD

The coordination of these projects became further complicated when the development organization was itself a customer for the components built by its subcontractors.

One of the most significant challenges to government or commercial development teams was to coordinate communications from different customer sources to develop a consistent understanding of the customer's requirements. When several customer sources gave inconsistent input, project personnel had to negotiate among them to clarify requirements. Conversely, different project members needed to provide consistent answers to the customer, and establishing a single point of contact for coordinating these communications was difficult.

When the communication channels to the customer were remote, necessary changes were often stifled. Communicating with DOD was especially difficult when the software project was subcontracted from a hardware company that held the prime contract. The approval cycle required for a change to the specifications was often too time-consuming to justify any but the most necessary modifications.

> Programmer: I got myself in trouble with a program manager one time because I said, "I know this [function] is wrong so we're just going to change it." We changed it and made it work. But he said, "No, no, no, go back and do it the way the spec says, because that spec came from the customer." . . . It is still not changed . . . to be correct. That particular function gives nonsense answers.

Designers needed operational scenarios of system use to understand the application's behavior and its environment. Unfortunately, these scenarios were too seldom passed from the customer to the developer. Customers often generated such scenarios in determining their requirements but did not record them and abstracted them out of the requirements document. Without deep application knowledge, designers worked from the obvious scenarios of application use and were unable to envision problematic and exceptional conditions the customer would ultimately want handled.

In some cases, classified documents contained operational scenario information, but the software designers could not obtain the documents from the customer. It was assumed that developers did not have a need to know. There might have been less need for system prototypes meant to collect customer reactions if information generated by potential users had been made available to the project team. That is, many projects spent tremendous time rediscovering information that, in many cases, had already been generated by customers, but not transmitted to developers.

Communications and Coordination Summary

Large projects required extensive communication that was not reduced by documentation. Project staff found the dialectic process crucial for clarifying issues. Particularly during early phases, teams spent considerable time defining terms, coordinating representational conventions, and creating channels for the flow of information. Artificial (often political) barriers to communication among project teams created a need for individuals to span team boundaries and to create informal communication networks. Organizational and temporal boundaries made some communication channels especially remote. Organizational boundaries hindered understanding the requirements, while temporal boundaries buried the design rationale. The complexity of the customer interface hindered the establishment of stable requirements and increased the communication and negotiation costs of the project. Since no single group served as the sole source of requirements in either commercial or government environments, organizational communications became crucial to managing the project.

CONCLUSIONS

The Behavioral Processes of Software Development

The problems elaborated in the preceding sections were described many times across projects that varied in size, technology, company, and customer. The way problems manifested themselves though, differed among projects. Chronicled by Weinberg [61], Brooks [14], Fox [27], and others, these problems have survived for several decades despite serious effort at improving software productivity and quality. We are not claiming to have discovered new insights for engineering management. Rather, we are trying to organize observations about the behavioral processes of large systems design to help identify which factors must be attacked to improve overall project performance. We are seeking to understand the mechanisms underlying these problems in order to design more effective software development practices and technology. The question is not whether we learned something new, but what did we observe that keeps us from acting on all those things we already knew.

Our interviews indicated that developing large software systems must be treated, at least in part, as a learning, communication, and negotiation process. Much early activity on a project involved learning about the application and its environment, as well as new hardware, new development tools and languages, and other evolving technologies. Software developers had to integrate knowledge from several domains before they could perform their jobs accurately. Further, as the project progressed they had to learn about design and implementation decisions being made on other parts of the system in order to ensure the integration of their components. Characteristically, customers also underwent a learning process as the project team explained the implications of their requirements. This learning process was a major source of requirements fluctuation.

A small subset of the design team with superior application domain knowledge often exerted a large impact on the design. Collaborative problem solving is related to productivity more often in small, rather than large, teams [23]. Similarly, the small, but influential, design coalitions that developed on numerous projects represent the formation of a small team in which collaboration was more effective. This decomposition of a large design team into at least one smaller coalition occurred when a few designers perceived their tighter, less interrupted collaboration would expedite the creation of a workable design. Exceptional designers, when available, were at the heart of these coalitions and accepted responsibility for educating the design team about the application and ensuring their technical cohesiveness.

Fluctuation and conflict among requirements afflicted large system development projects continuously. Whether they are called *ill-structured problems* [47] or *wicked problems* [49], the unique obstacles encountered in large software projects typically did not plague small, well-understood software applications with complete and stable specifications. These requirements problems emerged from the learning process at the heart of the dialectic between customers and developers. There was a natural tension between getting requirements right and getting them stable. Although this tradeoff appeared to be a management decision, it was just as often adjudicated by system engineers. Fluctuation and conflict among requirements were exacerbated when several organizational components presented themselves as the customer and the developers had to negotiate a settlement.

Organizational boundaries to communication among groups both within companies and in the business milieu inhibited the integration of application and computational knowledge. These communication barriers were often ignored since the artifacts produced by one group (e.g., requirements documents from marketing) were assumed to convey all the information needed by the next group (e.g., system designers). Designers complained that constant verbal communication was needed between customer, requirements, and engineering groups. Organizational structures separating engineering groups (hardware, software, and systems) often inhibited timely communication about application functionality in one direction and feedback about implementation problems that resulted from system design in the other direction. When coalitions formed around conflicting views of the design, they typically formed along organizational lines.

Although far from the only issues participants described, requirements issues were a recurring theme in our interviews. The three problems we described provide, among other things, three views of the requirements problem: how system requirements were understood, how their instability affected design, and how they were communicated throughout a project. Although a circumscribed requirements phase can be identified in most software process models, requirements processes occur throughout the development cycle.

Implications for Software Tools and Practices

The descriptions provided in our interviews indicate how productivity and quality factors influenced project performance. Three issues, in particular, must be addressed if software productivity and quality are to be improved. The first is to increase the amount of application domain knowledge across the entire software development staff. Designers of software development environments should discover ways for these environments to creatively facilitate the staff-wide sharing and integration of knowledge.

Second, software development tools and methods must accommodate change as an ordinary process and support the representation of uncertain design decisions. For instance, the essence of simulation and prototyping is a process of exploration, discovery, and change. Whether design decisions are delayed, or whether new requirements are negotiated among several customer components, change management and propagation is crucial throughout the design and development process.

Finally, any software development environment must become a medium of communication to integrate people, tools, and information. If information created outside of the software tools environment must be manually entered, developers will find ways around using the tools, and information will be lost. Lost information and poor communication facilities make the coordination task more difficult. Thus, three capabilities that we believe must be supported in a software development environment are knowledge sharing and integration, change facilitation, and broad communication and coordination.

Software development tools and practices had disappointingly small effects in earlier studies, probably because they did not improve the most troublesome processes in software development. Understanding the behavioral processes of software development allows us to evaluate the claims for software tools and practices. Conceptually, this understanding helps us reason whether a given tool or practice can affect the processes underlying the problem it claims to solve. Empirically, it helps identify which processes should be measured in evaluating whether the tool or practice can spark improvement. If a tool is used in individual activities by designers, and the benefits they experience individually do not scale up to reduce a project's effort or mistakes, then we should not be surprised when little impact shows up in productivity and quality data. If a tool or practice failed to impact at least one of the three problems we discussed in this article, we would be surprised if it had substantial impact on the performance of large projects.

Implications for Project Management

Although we initiated this project to study organizational level factors in software development, we were constantly confronted with the impact of individual talent and experience on a project. After observing similar effects in his productivity data, Boehm concluded: "Personnel attributes and human relations activities provide by far the largest source of opportunity for improving software productivity" [9 p. 666]. Brooks reiterated this point: "The central question in how to improve the software art centers, as it always has, on people" [15 p. 18]. This view was reflected with remarkable consistency in interviews with vice presidents from different companies.

> Vice president 1: I guess if you had to pick one thing out that is most important in our environment, I'd say it's not the tools that we use, it's the people.
> Vice president 2: The most important ingredient that was successful on this project was having smart people . . . Very little else matters in my opinion. . . . The most important thing you do for a project is selecting the staff. . . . The success of the software development organization is very, very much associated with its ability to recruit good people.
> Vice president 3: The only rule I have in management is to ensure that I have good people—real good people—and that I grow good people, and that I provide an environment where good people can produce.

Given the amount of knowledge to be integrated in designing a large software system and the inability of current technology to automate this integration [48], these opinions are not surprising. Contributions by good people do not come just from their ability to design and implement programs. A myriad of other processes—resolving conflicting requirements, negotiating with the customer, ensuring that the development staff shares a consistent understanding of the design, and providing communications between two contending groups—are crucial to project performance and require faculties that no tool or practice can provide.

The constant need to share and integrate information suggests that just having smart people is not enough. The communication necessary to develop a shared vision of the system's structure and function, and the coordination necessary to support dependencies and manage changes on large system projects are team issues. Individual talent operates within the framework of these larger social and organizational processes. The influence of exceptional designers was exercised through their impact on other project members and through their ability to create a shared vision to organize the team's work. Recruiting and training must be coupled with *team building* [55] to translate individual talent into project success. Thus, the impact of processes at one level of the layered behavioral model must be interpreted by their impact on processes at other levels.

Implications for Software Process Models

A typical statement that we heard from participants was that, you've got to understand, this isn't the way we develop software here. This type of comment suggested that these developers held a model of how software development should occur, and they were frustrated that the conditions surrounding their project would not let them work from the model. The frequency of this comment also suggested that the model most developers envisioned accounted poorly for the environmental conditions and organizational context of software development. The participants we interviewed were uniformly motivated to do a good job, but they had to mold their development process to navigate through a maze of contingencies.

These interviews provided a clearer understanding of such crucial processes as learning, technical communication, requirements negotiation, and customer interaction. These processes are poorly described in software process models that focus instead on how a software product evolves through a series of artifacts such as requirements, functional specifications, code, and so on. Existing software process models do not provide enough insight into actual development processes to guide research on software development technologies. Models that only prescribe a series of development tasks provide no help in analyzing how much new information must be learned by a project staff, how discrepant requirements should be negotiated, how design teams resolve architectural conflicts, and how these and similar factors contribute to a project's inherent uncertainty and risk. Boehm's spiral model is a promising attempt to manage these issues at a macro level [11].

The layered behavioral model must be integrated with evolutionary process models in order to create a comprehensive model of the software development process. When we overlay cognitive, social, and organizational processes on the phased evolution of software artifacts, we begin to see causes for bottlenecks and inefficiencies in development. The more deeply project managers understand these behavioral processes, the greater their insight into the factors that determine their success.

The layered behavioral model encourages greater focus on the human processes that exert so much influence on software productivity and quality. For this model to mature beyond its current descriptive state, rules of aggregation must be posed that provide the model with analytic power for at least some development processes. Aggregating behavior across layers in the model exposes the effects of new processes added at each layer. Aggregation also indicates how the impact of processes such as communication may not scale linearly across layers. Behavioral processes at each layer are useful analytically only if they make independent contributions to understanding software development processes. The relative importance of each layer's contribution will vary with the process or problem under analysis. Further work with this model may indicate analyses for which new layers need to be identified or existing layers combined. Our goal is to fashion a useful tool for analyzing how different factors in software development affect project behavior—and, ultimately, project outcomes.

Implications for Ecological Research on Professional Programming

This study provides an *ecological* perspective on software design, since software design problems were assessed against the backdrop of the working environment in which they occurred. Ours, however, was not a purely ecological analysis. Traditionally, the ecological perspective has only focused on how characteristics of the situation affected human behavior [5, 45]. Rather, the information in our interviews forced us to account for differences among individual project members and to determine how these differences interacted with variations among situations. Therefore, our analysis is more accurately characterized as *interactionist* [41, 52, 53] since we attributed variation in software productivity and quality to differences between both people and situations and their interaction.

The exploratory ecological research reported here exposed many of the processes that affect software productivity and quality. The MCC Software Technology Program is using these insights as problem-driven input to its research on advanced software design environments. As research artifacts are developed, the focus of our empirical research will shift from exploratory to evaluative. Evaluative research will investigate how the most important productivity and quality factors can be improved by changing either the process or technology of software development.

APPENDIX A: FIELD STUDY METHODS

Interview Format

This field study consisted of structured interviews [13, 25, 60, 62] with design team members who held different roles (e.g., system engineer, lead software architect, project manager). In designing these structured interviews, each member of our field study team independently generated a set of questions for each level in the layered behavioral model and indicated the project roles for each question. These questions focused on such upstream activities as customer interaction, requirements analysis, design meetings, and project communications. The questions were then reduced to a single set that was reviewed by representatives from each participating company to ensure their relevance across software environments.

The questions were open-ended and allowed participants to formulate answers in their own terms. Thus, the questions were points of departure for participants to describe their opinions about important events and challenges during software design, and their insights were explored in depth. Participants were encouraged to recall as much information as possible about the process of designing their system and the factors that affected its productivity and quality. Questions producing identical answers over a number of projects were eventually dropped from the interviews, and new questions were added when we learned of additional processes needing investigation.

Interviewers worked in pairs [38] with one interviewer taking the lead, while the other recorded notes about important points. This division of responsibilities increased rapport with participants, since they had the questioner's full attention. We found tandem interviewing had two additional advantages. First, interviewers often exchanged the lead role several times during the interview as topics changed, or as one interviewer began to tire. The ability to shift roles kept the pace of the interview lively and provided timely opportunities for shifts in focus. Second, the interviewer exercising the support role often requested deeper explanations of important points not pursued by the lead interviewer.

We piloted our field study methods on a project in our own laboratory and videotaped our interviews for study and critique. We also conducted a pilot field study on a participant company project. Further, prior to beginning formal data collection we worked with an anthropologist/psychologist team experienced in interviewing software development projects in order to refine our methods and enhance participants' willingness to reveal their experiences.

Interview Bias

The information gathered from these interviews was subjective. By interviewing numerous participants in varying positions (e.g., manager, designer), we attempted to balance the perspectives presented on each project. Nevertheless, bias can result from various interactions between the interviewers and respondents which can affect interview data. We will describe the most significant biases in our methods and explain how we minimized their impact.

Warwick and Lininger [60] warn of four interviewing mistakes that we attempted to minimize. First, reshaping questions to match the participant's role in the project presented few problems, since we were not attempting to derive quantitative data from the responses. Second, tandem interviews increased the probing necessary to obtain full explanations of answers. Third, tape recording eliminated data recording errors. Finally, we did not have to motivate participants, since most were anxious to discuss their work with people interested in listening. Some even returned after hours to complete interviews.

The bias introduced into the interview data by the participants was a more serious concern. Of the various types of participant bias discussed in the interviewing literature, three presented the greatest problems for interpreting our data. The *social desirability bias* occurred when participants constructed answers to conform to the norms of their location or professional group. The *self-presentation bias* occurred when participants described their role in past events in a more favorable or important light than was actually the case. The *plausibility bias* occurred when portions of an event had been forgotten and were reconstructed with plausible explanations that differed from the actual events. Recalling past events is a reconstructive process [39]. We attempted to detect these biases by deeply probing participant's answers and comparing explanations of the same events with answers provided by other participants to piece together the most likely sequence and explanation of events on a project.

Acknowledgments. We recognize strong contributions to this research from Vincent Shen, who assisted in the collection and summary of the interview data, and from Barbara Smith, who spent so many hours transcribing tapes. We thank the remaining members of the MCC-STP empirical studies team (Diane Walz, Raymonde Guindon, and Nancy Pennington) for their thoughtful contributions, and our colleagues who provided insightful reviews (Jim Babcock, Barry Boehm, Fred Brooks, Jack Carroll, Jonathan Grudin, Peter Marks, Colin Potts, Stan Rifkin, Ben Shneiderman, Elliot Soloway, and Jerry Weinberg). We also thank Dani and Jerry Weinberg for help in refining interview techniques and Les Belady for his continuing support of our Empirical Studies of Software Design project. We are indebted to the companies in MCC's Software Technology Program and their employees who candidly participated in this study, and we thank the company liaisons for arranging these interviews. Brief reports of this research have appeared in the proceedings of the Ninth International Conference on Software Engineering, the Eleventh NASA Software Engineering Workshop, and the Second Workshop on Empirical Studies of Programmers.

REFERENCES

1. Adams, J.S. The structure and dynamics of behavior in organizational boundary roles. In *Handb. Ind. Organ. Psychol.*, Ed. M.D. Dunnette. Rand-McNally, Chicago, (1976), pp. 1175–1199.
2. Adelson, B., and Soloway, E. The role of domain experience in software design. *IEEE Trans. Softw. Eng. 11*, 11 (Nov. 1985), 1351–1360.
3. Allen, T.J. Communication networks in R&D laboratories. *R&D Manage. 1*, 1 (Jan. 1970), 14–21.
4. Allen, T.J. Organizational structure, information technology, and R&D productivity. *IEEE Trans. Eng. Manage.* 33, 4 (Apr. 1986), 212–217.
5. Barker, R.G. *Ecological Psychology: Concepts and Methods for Studying the Environment of Human Behavior.* Stanford Univ. Press, Palo Alto, Calif., 1986.
6. Barstow, D.R. Domain-specific automatic programming. *IEEE Trans. Softw. Eng. 11*, 11 (Nov. 1985), 1321–1336.
7. Belady, L.A. The Japanese and software: Is it a good match? *IEEE Comput. 19*, 6 (June 1986), 57–61.
8. Benbasat, I., Goldstein, D.K., and Mead, M. The case research strategy in studies of information systems. *MIS Q. 11*, 3 (Mar. 1987), 369–386.
9. Boehm, B.W. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, N.J., 1981.
10. Boehm, B.W. Improving software productivity. *IEEE Comput. 20*, 9 (Sept. 1987), 43–57.
11. Boehm, B.W. A spiral model of software development and maintenance. *IEEE Comput. 21*, 5 (May 1988), 61–72.
12. Bouchard, T.J. Field research methods. In *Handb. Ind. Organ. Psychol.*, Ed. M.D. Dunnette. Rand-McNally, Chicago, (1976), pp. 363–413.
13. Brenner, M., Brown, J., and Canter, D. *The Research Interview: Uses and Approaches.* Academic Press, London, 1985.
14. Brooks, F.P. *The Mythical Man-Month.* Addison-Wesley, Reading, Mass., 1975.
15. Brooks, F.P. No silver bullet. *IEEE Comput. 20*, 4 (Apr. 1987), 10–19.
16. Card, D.N., McGarry, F.E., and Page, G.T. Evaluating software engineering technologies. *IEEE Trans. Softw. Eng. 13*, 7 (July 1987), 845–851.
17. Christiansen, D. On good designers. *IEEE Spectrum 24*, 5 (May 1987), 25.
18. Curtis, B. Measurement and experimentation in software engineering. *Proc. IEEE 68*, 9 (Sept. 1980), 1144–1157.
19. Curtis, B. Substantiating programmer variability. *Proc. IEEE 69*, 7 (July 1981), 846.
20. Curtis, B. *Human Factors in Software Development.* 2d ed. IEEE Computer Society, Wash., D.C., 1985.
21. Curtis, B., Sheppard, S.B., Kruesi-Bailey, E., Bailey, J., and Boehm-Davis, D. Experimental evaluation of software documentation formats. *J. Syst. Softw.* In press.
22. Curtis, B., Soloway, E., Brooks, R., Black, J., Ehrlich, K., and Ramsey, H.R. Software psychology: The need for an interdisciplinary program. *Proc. IEEE 74*, 8 (Aug. 1986), 1092–1106.
23. Dailey, R.C. The role of team and task characteristics in R&D team collaborative problem solving and productivity. *Manage. Sci. 24*, 15 (Nov. 1978), 1579–1588.
24. DeMarco, T., and Lister, T.A. *Peopleware.* Dorset, New York, 1987.
25. Fenlason, A.F., Ferguson, G.B., and Abrahamson, A.C. *Essentials in Interviewing.* Harper & Row, New York, 1962.
26. Fischer, B.A. *Small Group Decision-Making.* 2d ed. McGraw-Hill, New York, 1980.
27. Fox, J.M. *Software and Its Development.* Prentice-Hall, Englewood Cliffs, N.J., 1982.
28. French, J.R.P., and Raven, B. The bases of social power. In *Studies in Social Power*, Ed. D. Cartwright, Institute for Social Research, Ann Arbor, Mich., 1959, pp. 150–167.
29. Gould, J.D., and Lewis, C. Designing for usability: Key principles and what designers think. *Commun. ACM 28*, 3 (Mar. 1985), 300–311.
30. Guinan, P.J., and Bostrom, R.P. *Communication Behaviors of Highly-Rated Versus Lowly-Rated System Developers: A Field Experiment.* The Institute for Resesrch on the Management of Information Systems, Indiana Univ., 1987.
31. Guindon, R., and Curtis, B. Control of cognitive processes during design: What tools would support software designers? In *Conference Proceedings of CHI'88*, (Washington, D.C., May 1988). ACM, New York, 1988, 263–268.
32. Guindon, R., Krasner, H., and Curtis, B. Breakdowns and processes during the early activities of software design by professionals. In *Empirical Studies of Programmers: Second Workshop*, Ed. G. Olsen, et al., Ablex, Norwood, N.J., (1987), 65–82.
33. Hastie, R. Experimental evidence on group accuracy. In *Information Processing and Group Decision-Making*, Ed. G. Owen, and B. Grofman. JAI Press, Westport, Conn., 1987, 129–157.
34. Jeffries, R., Turner, A.A., Polson, P.G., and Atwood, M.E. The processes involved in designing software. In *Cognitive Skills and Their*

Acquisition, Ed. J.R. Anderson. Erlbaum, Hillsdale, N.J., 1981, pp. 255–283.
35. Jones, T.C. The limits to programmer productivity. In *Proceedings of the Joint SHARE/GUIDE/IBM Applications Symposium*, SHARE/GUIDE, Chicago, (1979), pp. 77–82.
36. Kant, E., and Newell, A. Problem solving techniques for the design of algorithms. *Info. Process. Manage. 28*, 1 (Jan. 1984), 97–118.
37. Kernaghan, J. A., and Cooke, R.A. The contribution of the group process to successful group planning in R&D settings. *IEEE Trans. Eng. Manage. 33*, 3 (Mar. 1986), 134–140.
38. Kincaid, H.V., and Bright, M. The tandem interview: A trial of the two-interviewer team. *Public Opin. Q.* 21, (1957), 304–312.
39. Klatzky, R.L. *Human Memory: Structures and Processes.* San Francisco, W.H. Freeman, 1975.
40. Kling, R. The web of computing: Computer technology as social organization. Vol. 21, *Adv. Comput.* Addison-Wesley, Reading, Mass., 1982, pp. 1–90.
41. Magnusson, D. *Toward a Psychology of Situations: An Interactionist Perspective.* Erlbaum, Hillsdale, N.J., 1981.
42. Malhotra, A., Thomas, J.C., Carroll, J.M., and Miller, L.A. Cognitive processes in design. *Int. J. Man-Machine Stud. 12*, (1980), 119–140.
43. McGarry, F.E. What have we learned in the last six years? In *Proceedings of the Seventh Annual Software Engineering Workshop* (Greenbelt, Md., Dec. 1982), NASA-GSFC, Greenbelt, Md., 1982.
44. Mills, J.A. A pragmatic view of the system architect. *Commun. ACM 28*, 7 (July 1985), 708–717.
45. Moos, R.H., and Insel, P.M. *Issues in Social Ecology: Human Milieus.* National Press Books, Palo Alto, Calif., 1974.
46. Myers, W. MCC: Planning the revolution in software. *IEEE Softw. 2*, 6 (Nov. 1985), 68–73.
47. Newell, A. Heuristic programming: Ill structured problems. Vol. 3, In *Prog. Oper. Res.*, Ed. J. Aronofsky. Wiley, New York, 1969, pp. 360–414.
48. Rich, C., and Waters, R.C. Automatic programming: Myths and prospects. *IEEE Comput. 21*, 8 (Aug. 1988), 40–51.
49. Rittel, H.W.J., and Webber, M.M. Dilemmas in a general theory of planning. *Policy Sci. 4*, 1973, 155–169.
50. Rogers, E.M., and Kincaid, D.L. *Communication Networks: Toward a New Paradigm for Research.* Free Press, New York, 1981.
51. Scacchi, W. Managing software engineering projects: A social analysis. *IEEE Trans. Softw. Eng. 10*, 1 (Jan. 1984), 49–59.
52. Sells, S.B. An interactionist looks at the environment. *Am. Psychol. 18*, 11 (Nov. 1963), 696–702.
53. Sells, S.B. Ecology and the science of psychology. *Multivariate Behav. Res. 1*, 2 (Feb. 1966), 131–144.
54. Swanson, E.B., and Beath, C.M. The use of case study data in software management research. *J. Syst. Softw. 8*, 1 (Jan. 1988), 63–71.
55. Thamhain, H.J., and Wilemon, D.L. Building high performance engineering project teams. *IEEE Trans. Eng. Manage.* 34, 3 (Mar. 1987), 130–137.
56. Tushman, M.L. Special boundary roles in the innovation process. *Adm. Sci. Q. 22*, 4 (Winter 1977), 587–605.
57. Vosburgh, J., Curtis, B., Wolverton, R., Albert, B., Malec, H., Hoben, S., and Liu, Y. Productivity factors and programming environments. In *Proceedings of the Seventh International Conference on Software Engineering* (Orlando, Fla., Mar. 1984). IEEE Comput. Soc., Washington, D.C., 1984, pp. 143–152.
58. Walston, C.E., and Felix, C.P. A method of programming measurement and estimation. *IBM Syst. J. 16*, 1 (Jan. 1977), 54–73.
59. Walz, D., Elam, D., Krasner, H., and Curtis, B. A methodology for studying software design teams: An investigation of conflict behaviors in the requirements definition phase. In *Empirical Studies of Programmers: Second Workshop*, Ed. G. Olsen, et al. Ablex, Norwood, N.J., 1987, pp. 83–99.
60. Warwick, D.P., and Lininger, C.A. *The Sample Survey: Theory and Practice.* McGraw-Hill, New York, 1975.
61. Weinberg, G.M. *The Psychology of Computer Programming.* Van Nostrand Reinhold, New York, 1971.
62. Whyte, W.F. Interviewing in field research. In *Human Organization Research*, Eds. R.N. Adams and J.J. Priess. 1960.
63. Zelkowitz, M., Yeh, R., Hamlet, R., Gannon, J., and Basili, V. Software engineering practices in the U.S. and Japan. *IEEE Comput. 17*, 6 (June 1984), 57–66.

CR Categories and Subject Descriptors: D.2.9 [**Software Engineering**]: Management; D.2.10 [**Software Engineering**]: Design; D.m [**Miscellaneous**]—*software psychology*; J.7 [**Computers in Other Systems**]; K.1 [**The Computer Industry**]; K.6.1 [**Management of Computing and Information Systems**]: Project and People Management; K.7.2 [**The Computing Profession**]: Organizations
General Terms: Software Management, Systems Design
Additional Key Words and Phrases: Case studies, domain knowledge, empirical studies of software development, large systems design, layered behavioral model, project communications, system requirements

ABOUT THE AUTHORS:

BILL CURTIS is a director in MCC's Software Technology Program where he has directed research on software process modeling and coordination, software design methods and tools, computer supported cooperative work, intelligent user interfaces, and empirical studies of software development. He is also an Adjunct Associate Professor in the Department of Management Science and Information Systems at the University of Texas. Present address: Microelectronics and Computer Technology Corp., P.O. Box 200195, Austin TX 78720.

HERB KRASNER manages the Software Process Research Group in Lockheed's Software Technology Center. He has experience in large systems development, industrial/academic research, and university teaching. His current research interests include: AI applied to software engineering, design teamware, process modeling and evaluation, decision-based design methods, and empirical studies. Present address: Lockheed Research Division, Organization 9601, Building 30E, 2100 East Elmo, Austin, TX 78744.

NEIL ISCOE is currently completing his Ph.D. in the Department of Computer Sciences at the University of Texas at Austin. His research interests include domain modeling and analysis, object-oriented design, and program generation. Prior to his work in the MCC field study, he served as president of a software development and consulting firm called Statcom Corporation. Present address: Department of Computer Sciences, University of Texas, Austin, TX 78712.