Chapter 4.1

The Psychology of Programming in the Large: Team and Organizational Behaviour

Bill Curtis[1] and Diane Walz[2]

[1] *Microelectronics and Computer Technology Corporation (MCC), 9430 Research Boulevard, Austin, TX 78759, USA*
[2] *Assistant Professor, Department of Accounting and Information Systems, The University of Texas at San Antonio, San Antonio, TX 78285, USA*

In its century-old quest to explain human behaviour, psychology has spawned many subfields to study different phenomena. Several of these psychological specialties, such as social, organizational, ecological and interactional psychology, focus on how humans behave in groups and how situational conditions affect them. Disappointingly little theory and few research paradigms from these fields have been imported into the psychological study of programming. Most of the empirical research on software development has been performed on individual programming activities (Curtis, 1985; Curtis *et al.*, 1986). This orientation toward programming as individual activity occurred because:

(1) most psychologists studying programmers were not social or organizational pychologists;

(2) experiments on team and organizational factors are difficult and expensive to conduct.

Psychology of Programming
ISBN 0-12-350772-3

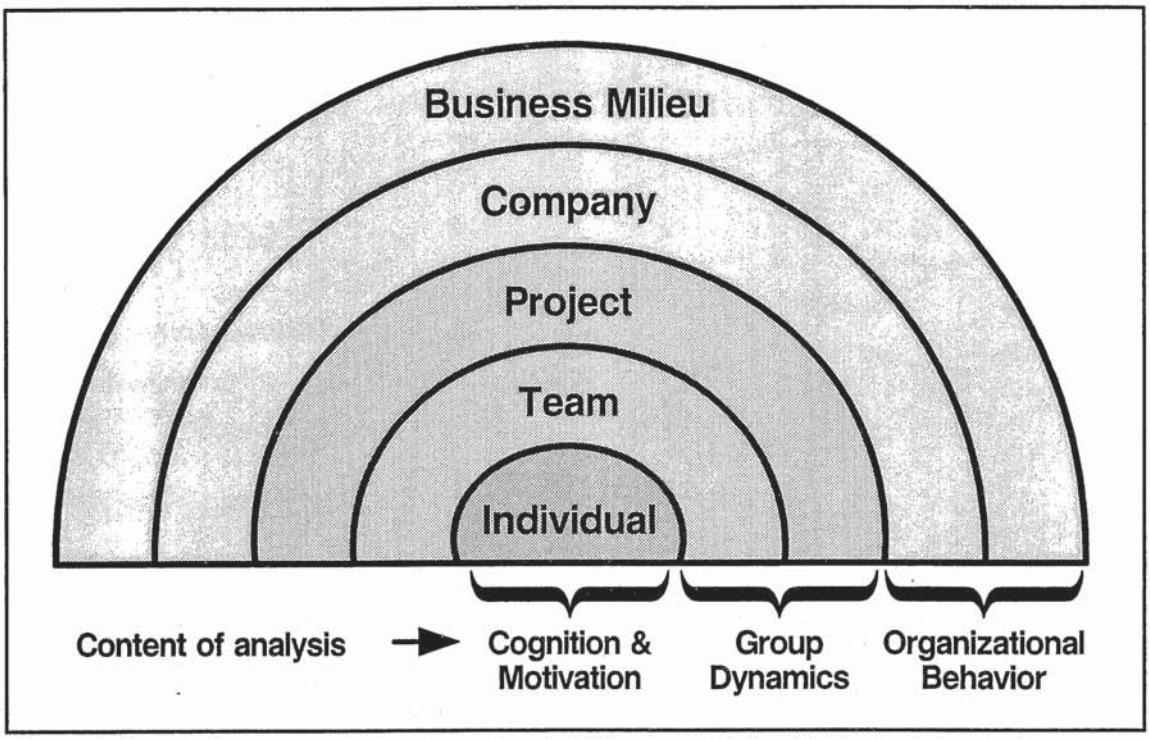

Figure 1: The layered behavioural model of software development.

Since software tools and practices are usually designed for individual problem solving, their benefits often do not scale up on large projects to overcome the impact of design processes that emerge from the social behaviour of the development team or the organizational behaviour of the company. Software development must be studied at several behavioural levels, as indicated in the *layered behavioural model* in Figure 1 proposed by Curtis *et al.* (1988). This model emphasizes factors that affect not merely the cognitive processes of software development, but also the social and organizational processes.

The layered behavioural model is an abstraction for organizing the behavioural analysis of large software projects. This model is orthogonal to traditional process models by presenting a cross-section of the behaviour on a project during any selected development phase. The layered behavioural model encourages researchers to extend their evaluation of software engineering practices beyond individuals, to determine if their effects scale up to an impact on the performance of an entire project (Kling and Scacchi, 1982; Scacchi, 1984). In order to develop a psychology relevant to programming in the large, we must take the *systems* view of software development activities encouraged by this model.

Contrary to most software process models, the layered behavioural model focuses on the behaviour of the humans creating the artifact, rather than on the evolutionary behaviour of the artifact through its developmental stages. At the individual level, software development is analysed 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 behaviour 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. The size and structure of the project determine how much influence each layer has on the development process.

Curtis (1988) described five psychological paradigms that have been used in studying programming. In the following sections we will review contributions to the empirical study of software development from two of these paradigms: *group dynamics* and *organizational behaviour*. Unfortunately, there is little research guided by these paradigms compared to the literature on the cognitive aspects of programming. Each of these two paradigms will be explored in later sections of this chapter. Although the senior author has conducted controlled experiments on programming phenomena (cf. Curtis *et al.*, 1989), the study of team and organizational behaviour has required different empirical methods, some more characteristic of sociology or anthropology. In subsequent sections we will describe some of the recent exploratory studies at MCC of software development projects from which we have tried to identify important problems that should become the focus of future empirical research on programming in the large.

1 Group dynamics

1.1 Programming team structure

Organizing programmers into teams superimposes a layer of social behaviour on the cognitive requirements of programming tasks. Two structures have been proposed for programming teams based on the centralized versus decentralized team organizations often studied in group dynamics research.

Mills (1971) and Baker (1972) designed a centralized organization for programming teams that placed primary responsibility for programming on a *chief programmer*. Other team members such as backup programmers, the program librarian and technical writers were organized as a support team for the chief programmer. Technical communication was *centralized* through the chief programmer. The success of this approach is generally believed to depend on the availability of a stellar technician to take the role of chief programmer.

In contrast to the chief programmer model, Weinberg (1971) proposed a decentralized team structure. In Weinberg's *egoless* team no central authority is posited in any team member. Different members take leadership responsibility for those project tasks that match their unique skills. The communication network in this team structure is *decentralized*, with technical information flowing freely among all team members. The key to egoless teams is that no single individual feels private ownership of any piece of the program. The program is a shared work product and decisions concerning it are reached by consensus. Weinberg recommended that the maximum size for this team was about ten members.

Social psychologists have established several results about centralized versus decentralized team structures relevant to differences between chief programmer and egoless teams. Shaw (1971) concluded that empirical evidence generally supported the following principles of team behaviour:

Table 1: Favourable conditions for different team structures.

Condition	Chief programmer	Egoless
Difficulty of problem	simple	complex
Size of program	large	small
Creativity required	low	high
Reliability requirements	low	high
Modularity requirements	high	low
Schedule	tight	relaxed
Duration of project	short	long
Team morale	low	high
Risk taking	low	high

* Groups usually produce more and better solutions to problems than individuals working alone, and their judgements are usually better on tasks involving error.

* A decentralized communication network is more effective for solving complex problems, whereas a centralized network is better for solving simple ones.

* Leaders emerge more often and organizational development is more rapid in centralized teams.

* A centralized network is more vulnerable to the saturation of its communication channels than is a decentralized network.

* Greater conformity and higher morale occur in decentralized teams.

Mantei (1981) used these and other principles to suggest conditions under which chief programmer and egoless teams would be most effective. An augmentation of her analysis is presented in Table 1. For instance, chief programmer teams should be most effective on large, simple, tightly schedule projects of short duration which do not require highly reliable or creative solutions. Egoless teams, on the other hand, should be most effective on small, complex projects requiring highly reliable and creative solutions with some risk performed under relaxed schedules over a long duration. Rather than being an either-or choice, chief programmer and egoless teams appear suited for different types of programming projects.

Unfortunately, few programming projects fit neatly into one of the two categories described above. For instance, many aerospace projects can be described as large, complex, tightly scheduled efforts spread over a long duration with high reliability requirements and sections requiring creative solutions. However, in such projects large portions of the system are not complex and do not require creativity. These latter components have been developed by the organization on previous projects and the structure of their solution is familiar to the project team.

A hybrid approach to structuring programming teams might be taken on large projects that have characteristics favourable to different types of programming teams.

Portions of the system whose solution does not present a new technical challenge might be programmed by chief programmer teams. Critical path or innovative portions covered by stringent reliability requirements or requiring creative solutions would be programmed by egoless teams. Within a single project, tasks would be assigned to the type of team best suited for them, thus matching the structure of the team and the task (von Mayrhauser, 1984). This matching is characteristic of the *interactional* approach that Sells (1963, 1966) has argued is necessary in making recommendations for real-world activities based on psychological theory.

In simulating team programming performance, Scott and Simmons (1975) found that as the amount of communication increased among the members of a five-person team, the chief programmer team became less productive because of the communication bottleneck which developed around the chief programmer. When programming team membership was varied between three and eighteen people, productivity leveled off between nine and twelve people, a level consistent with Weinberg's (1971) recommendation for the size of an egoless team.

1.2 The interaction of methodology and team process

Basili and Reiter (1981) were among the first to study actual programming teams experimentally. They wanted to determine the effects of programming discipline on team performance. Their experiment involved forty-five advanced students assigned to one of three conditions. These conditions were seven three-person teams trained in a disciplined team methodology, six three-person teams provided with no methodological training, and six individuals working alone. The task was to develop a compiler for a small, high-level language that was estimated to require two person-months and 1200 lines of Simpl-T.

Basili and Reiter used automated data collection techniques to amass measures on both the product and the process. They found almost no differences between the three development approaches on the product measures at traditional ($p < 0.05$) levels of statistical significance. However, differences among the approaches emerged on the process measures. The disciplined teams required fewer computer job steps, fewer compilations, fewer executions, and fewer changes than the unorganized teams or individuals.

Based on these results, Basili and Reiter concluded 'that methodological discipline is a key influence on the general efficiency of the software development process'. They believed there was evidence, although weaker, 'that mental cohesiveness is a direct influence on the general quality of the software development product...and that the disciplined methodology offsets the mental burden of organizational overhead [of working in teams] and enables a disciplined programming team to behave more like an indiviudal programmer relative to the developed software product'.

Boehm *et al.* (1984) studied seven teams of programmers using either a prototyping or a top-down specification-based methodology for producing a moderate sized application. Although the prototyped application contained less code and required less development effort, the traditionally specified packages exhibited a more coherent design and were easier to integrate. The prototyped packages were easier to learn and use, but were less functional and robust. These two approaches appeared to focus on different attributes of the problem during design. Prototyping focused on the user, while traditional specification-based approaches focused on the

structural integrity of the program. Deciding which approach is more appropriate depends on an analysis of various trade-offs during the design process.

Some programming team research has studied technical reviews. Myers (1978) found that team walkthroughs consumed twice as many minutes per defect found as did individual execution testing. However, walkthroughs exhibited much less variability in results, since the large differences among individuals yielded large differences in the effectiveness of individual testing sessions. In a further analysis, Myers found that pooling the results of independent testing sessions was more effective than team walkthroughs in detecting defects. He observed differences in the focus of these different testing approaches. Individuals focused too much on normal rather than abnormal conditions, while walkthrough teams focused too much on logic rather than input/output problems. A fruitful area of future research concerns how to use a mix of individual and team processes to focus attention on different aspects of a problem.

Programming team activities offer many opportunities for peer review activities that may be formal or informal components of the development process. Shneiderman (1980) reported several brief experiments in having programmers provide anonymous feedback on the design of each other's programs. Most programmers indicated that the experience was educational, although it was naive to expect that anonymity could be maintained when programmers knew each others' coding habits from previous exposure. Similar educational benefits for team walkthroughs were observed by Lemos (1979).

1.3 The MCC object server study

Although existing studies help us begin to build a psychology of programming team behaviour, we need deeper insight into the actual processes that occur as teams develop programs over extended periods of time. The existing research on programming teams either studied the quantitative results of a task completed in a short time, or assessed the output of an extended programming assignment performed by students. In order to study the behaviour of an actual programming team, we collected longitudinal data on an MCC team that was building an object server to provide a repository for the persistent objects created during object-oriented programming. For a period of three months, we videotaped every group meeting held among customers, developers, and combinations of both groups. We transcribed seventy-two hours of videotapes covering thirty-seven different meetings that ranged from one to two hours.

The analysis of these videotapes emphasized the project's information requirements and their effect on the group process, especially its information sharing activities (Walz *et al.*, 1987). We began by assuming that the conflicts within a software design team were not solely the result of incompatible goals and/or opinions, but also represented the natural dialectic through which knowledge was exchanged. A scheme was developed for coding each utterance by a team member into one of several categories that reflected the unique characteristics of software design meetings. Analyses of these utterances indicated that the design meetings were generally dominated by a few individuals on the team to whom fellow participants attributed the greatest breadth of expertise. These individuals appeared to form a coalition that controlled the direction of the team.

An important issue in group dynamics that has not been discussed in the context of programming teams is the formation of coalitions. A small subset of the design

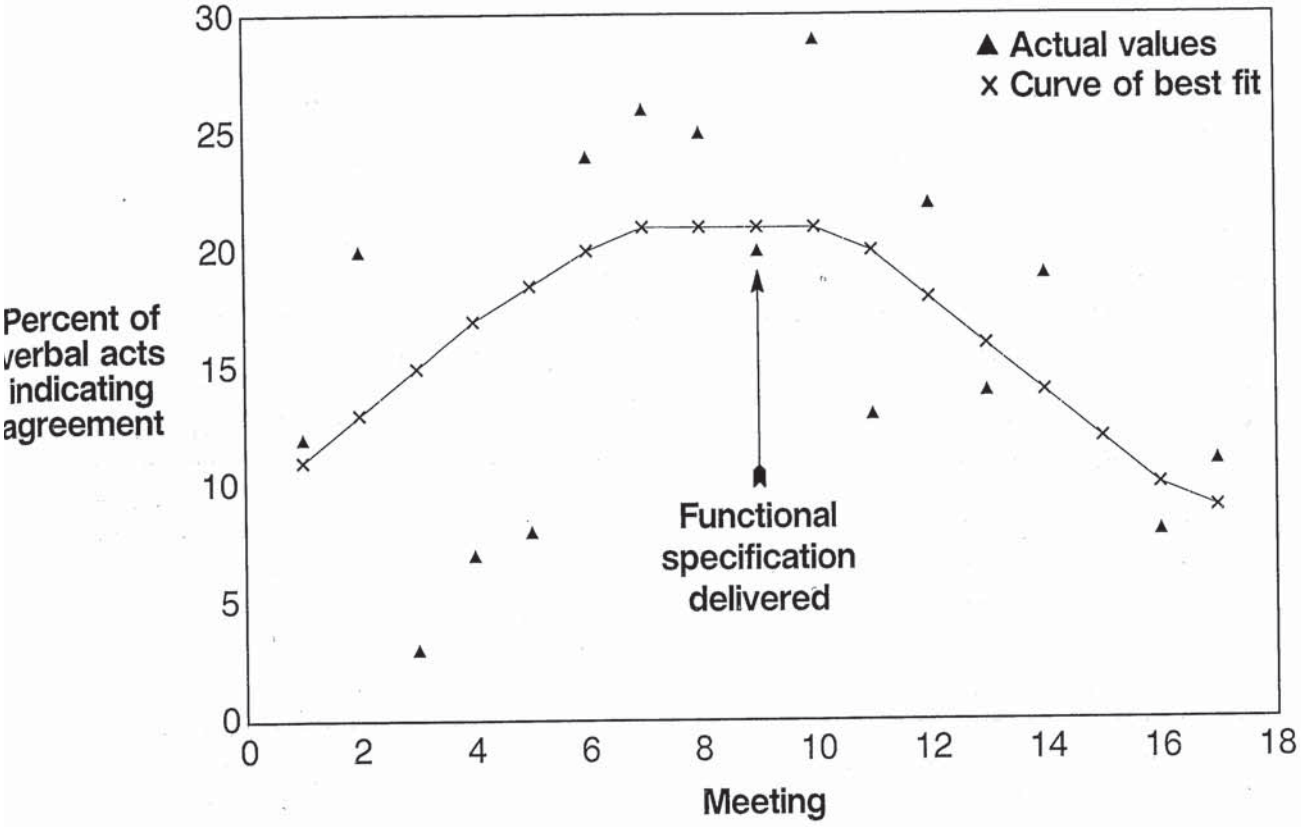

Figure 2: Level of agreement among designers working together on subtasks.

team with superior application domain knowledge often exerts a large impact on the design. Dailey (1978) found that collaborative problem solving was related to productivity in small, rather than large, research and development teams. Similarly, the small, but influential, design coalitions we have seen in other design projects represent the formation inside the larger team of a small team within which collaboration was more effective. Exceptional designers were often at the heart of these coalitions. Although teams usually outperform the average team member, the phenomena of exceptional designers is important because there is little evidence that teams can outperform the best team member (Hastie, 1987; Kernaghan and Cooke, 1986).

A scheme was developed to categorize each verbal act in the meeting protocols according to its role in the meeting (Walz, 1988). An interesting pattern in the verbal acts representing agreement among participants was observed across the seventeen meetings in the design phase of this project. A simplistic model assuming that cooperative design activity requires agreement among team members would hypothesize that such agreement should increase monotonically across meetings. However, as the data in Figure 2 demonstrate, there was a surprising inverted U-shaped curve (verified through logistic regression) that characterized verbal acts of agreement. This plot presents the percentage of agreement among team members working on the same subtask; a conservative analysis that should produce more consistent and lasting patterns of agreement than analyses based on the entire team. However, the plot indicates that agreement increased until the period (meetings 7 to 10) when the design team released a document presenting its response to customer requirements. In subsequent meetings the level of agreement began to decrease.

Since we analysed data from only a single team, it is difficult to draw conclusions that we would generalize to other software design episodes. However, there are some intriguing possibilities that lead to radically new models of how design teams operate. For instance, we may have observed a phenomena in which a team must reach some level of agreement in order to produce an artifact, but then reopens conflicts that were suspended in order to meet a milestone. On the other hand, we may have observed a phenomena where reaching consensus at one level of abstraction in a top-down design process does not relieve the conflicts that will result from issues that emerge at the next level down. Alternatively, we may have observed the impact of a small coalition that took control of the team during a period of deadline pressure and forced the level of consensus necessary to produce an artifact, but then dissolved when the deadline pressure passed, opening the door to renewed disagreement. Finally, we may have been observing a phenomena isolated to the unique characteristics of this particular team. We will not be able to make a definitive assessment of conflict resolution in design teams until we have studied other teams and compared our data to those produced by other researchers who study design teams longitudinally.

The design behaviours we observed in the meetings we videotaped suggested a three-stage process. During the first stage the team focused on determining requirements through learning about the application. The second stage involved communicating requirements among team members and with customers in order to develop a common understanding of the system to be built. In the third stage the group focused on creating artifacts that satisfied the requirements. Although the first and third stages are typical in most models of the software development process, the second stage involving a dialectic to surface misunderstandings is rarely made explicit.

Our observational study raised important questions that we need to study further in order to improve software development methods and process models. Consensus decisions are best achieved when a team reaches common understandings about their disagreements on interpreting requirements and how different architectural models of the system might operate. Our observations suggest the importance of explicit stages for training in the application domain and surfacing assumptions about the design. In these stages group conflict may be an important precursor to establishing group consensus. Formal requirements analysis methods may need process components that indicate when to enter and exit these various stages.

1.4 Team behaviour summary

Unfortunately, there has been too little research on software development teams in relation to their impact on software productivity and quality. If we assume that the conceptual unity of the program design is critical to the success of the software project, then teams must co-ordinate their work to make it appear as the work of one individual. The advantage of teams is their ability to bring divergent perspectives to bear in designing a program architecture. The process of synthesizing diverging design opinions on a programming team into a consensus involves conflict. Far from being a destructive force within the team, conflict can be a creative force if managed properly. Team methodologies must focus on co-ordinating the tasks and product concept. The structure of programming teams should: (1) reflect the nature of the task rather than the organization, (2) allow members to speak as if with one mind, and (3) determine the tasks the team can effectively handle.

2 Behaviour in programming organizations

2.1 The impact of organizational factors on programming

Although the primary focus for increasing software productivity and quality has been on improved methods, tools and environments, their impact in empirical studies of software productivity and quality on large projects from several industrial environments has been disappointing compared to the impact of factors characterizing the behaviour of the organization. For instance:

* In IBM Federal Division, Walston and Felix (1977) found that the complexity of the customer interface, the users' involvement with requirements definition, and the experience of the project team had more productivity impact than the use of software methods and tools.

* In the Defense and Space Group at TRW, Boehm (1981) found that the capability of the team assigned to the project had twice the productivity impact of the complexity of the product, and four times the impact of software tools and practices.

* In identifying a broad set of factors that acounted for two-thirds of the variation in software productivity, Vosburgh *et al.* (1984) argued that half of this variation was affected by factors over which project management had little control (factors other than the use of software engineering practices and tools).

In these studies human and organizational factors presented boundary conditions that limited the situations in which methods, tools and environments could increase software productivity and quality.

Many of the technologies purported to improve software productivity and quality only affect a few of the factors that exert the most influence over outcomes on large projects (Brooks, 1987). As the size of the system increases, the social organization required to produce it grew in complexity and the factors controlling productivity and quality may change in their relative impact. Technologies that enhance productivity on medium-sized systems may have less influence on large projects where factors that, while benign on medium systems, ravage project performance when unleashed by a gargantuan system that may involve the co-ordination of many companies. A great danger on large projects is that management will be deceived by the simplicity of the prescribed processes and will not understand what pitfalls are likely to await them (Fox, 1982).

Most of the organizational research on computing organizations has been on the effect of computing systems on the structure and functioning of the organizations that use the systems. Far less research has been performed on how organizational behaviour affects those who produce the systems. In part, this is because the cost of conducting experimental studies on these whole divisions is prohibitive. One of the few topics investigated has been the power and influence of information services departments in large organizations. Lucas (1984) reported several reasons that information services departments are perceived to have lower power than other departments in a company and carry less influence in decision making than might be expected. Srinivasan and Kaiser (1987) warn about the level of exposure that the programming team is allowed from outside sources. However, these studies have

given us little insight into the problems that corporations experience in designing large, complex computer systems.

2.2 The MCC field study

In order to obtain insight into large system development problems, members of the software empirical research team at MCC conducted a field study of large software development projects (Curtis *et al.*, 1988). The field study was designed to provide detailed descriptions of development problems in such processes as problem formulation, requirements definition and analysis, and software architectural design. We sought to study projects that involved at least ten people, were past the design phase, and involved real-time, distributed or embedded applications. We interviewed projects from nine companies in such businesses as computer manufacturing, telecommuncations, consumer electronics and aerospace. Our objective was to get software development personnel to describe the organizational conditions that affected their work. From their descriptions we hoped to gain insight that could lead to better models of actual development processes at several levels of analysis, especially the team and organizational levels of the layered behavioural model presented in Figure 1.

In this study we employed *field research* methods characteristic of sociology and anthropology (Bouchard, 1976). This field study consisted of interviews with ninety-seven project team members from seventeen large system development projects. We conducted hour-long structured interviews on-site with systems engineers, senior software designers, the project manager, and occasionally the division general manager, customer representatives, or the testing/QA manager. Participants were guaranteed anonymity, and the information reported was 'sanitized' so that no individual person, project, or company could be identified. The data we collected lend themselves to the creation of the case studies that Benbasat *et al.* (1987) and Swanson and Beath (1988) recommended for use in research on information systems development and project management.

Three of the most important problems we uncovered in this study were the thin spread of application domain knowledge on the software development staff, fluctuating and conflicting requirements, and communication and co-ordination breakdowns. We will describe one of the problems we investigated – breakdowns in communication and co-ordination – as a fertile source of opportunities for psychological research on programming in the large at the organizational level.

2.3 Communication and co-ordination breakdowns

One of the major problems in organizational behaviour that we identified in the field study was breakdowns in project communication and co-ordination. A large number of groups had to co-ordinate their activities, or at least share information, during software development. Figure 3 presents some of the groups mentioned during field study interviews, clustered into behavioural layers according to their remoteness from communication with individual software engineers (cf. Tushman, 1977). Remoteness involved the number of nodes in the formal communication channel that information must 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.

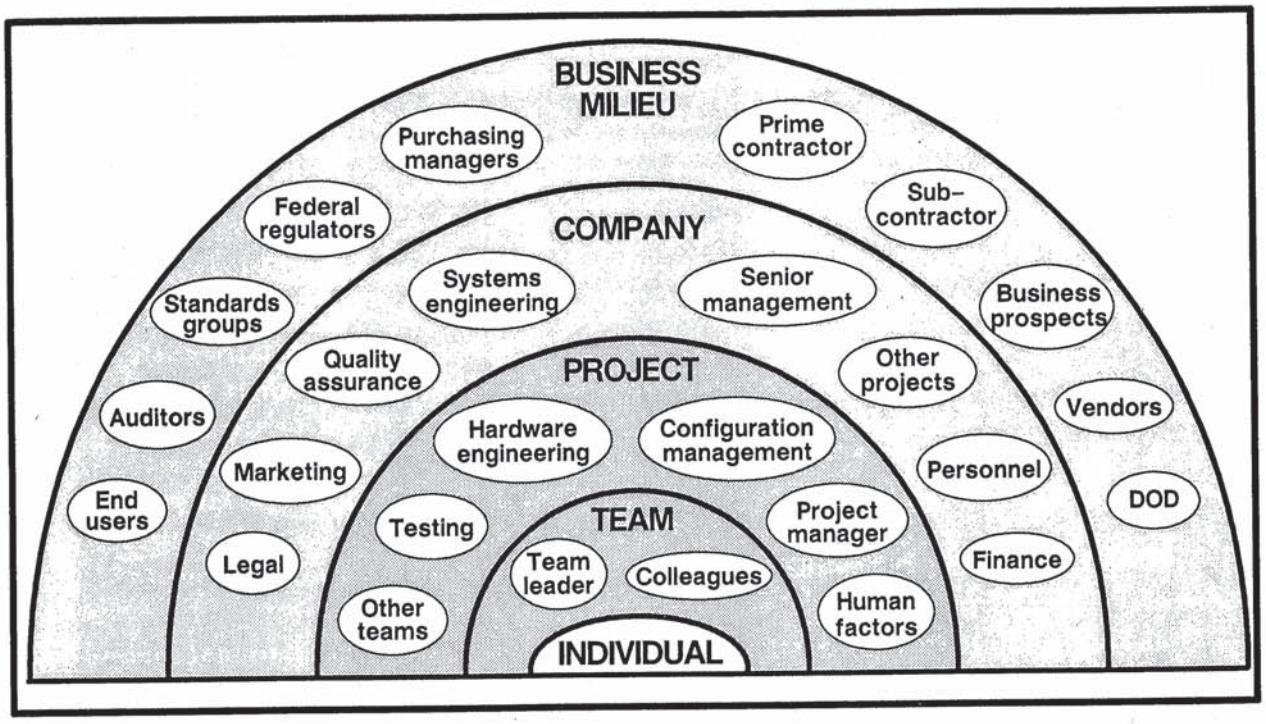

Figure 3: Remoteness of communications expressed in the layered behavioural model.

The model in Figure 3 implies that a software engineer normally communicated 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 cross 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 the geographic separation, to cultural differences, and to environmental factors.

Organizational boundaries to communication among groups both within companies and in the business milieu inhibited the integration of knowledge about the system. These communication barriers were often ignored since the artifacts produced by one group (e.g. marketing) were assumed to convey all the information needed by the next group (e.g. system design). However, designers complained that constant verbal communication was needed between customer, requirements and engineering groups. For instance, organizational structures that separated 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.

Most project members had several networks of people they talked with to gather information on issues affecting their work. Similar to communication structures observed by Allen (1970) in research and development laboratories, each network might involve different sets of people and cross organizational boundaries. Each network

supported a different flow of information; for example, information about the application domain, the system architecture, and so forth. When used effectively, these sources helped co-ordinate dependencies among project members and supplemented their knowledgte, thus reduced learning time. Thus, integrating information from these different sources was crucial to the performance of individual project members (Allen, 1986). These networks lend themselves to the network analysis described by Rogers and Kincaid (1981).

Some communication breakdowns between project teams were avoided when one or more project members spanned team or organizational boundaries (Adams, 1976). 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 implementation teams.

On most large projects, the customer interface was also an *organizational communications* issue and this interface too often restricted opportunities for developers to talk with end users. The formal chain of communication between the developer and the end user was often remote because the communication chain traversed through the programmer's team leader, the team leader's project manager, the project manager's marketing organization, marketing's contact with the purchasing manager, and ultimately the end user for which the system was being purchased (Figure 4). Notice that four nodes have been placed between the programmer and the end user.

This tortuous chain of communication links was usually set up to solve two communication problems. First, the marketing group wanted to control the customer's access to the developers so that any attitudes or personal habits of the programming staff that would shake the customer's confidence in the development organization are not observed. Secondly, the marketing group wanted to establish a single point of contact for the customer, so that consistent messages would be sent to the customer through a single source. However, at the same time the interface was often cluttered with communications from non-user groups such as auditors, finance and standards groups in the customer's organization, each with its particular concerns. Typically, development organizations could not get a single point of customer contact for defining system requirements. 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.

Designers needed operational scenarios of system use to understand the application's behaviour and its environment. Unfortunately, these scenarios were too seldom passed from the customer to the developer. Customers often generated many such scenarios in determining their requirements, but did not record them and abstracted them out of the requirements document. Lacking good scenarios of intended system use, designers worked from the obvious scenarios of application use and were unable to envision problematic exception conditions or subtleties the customer would ultimately want. In some cases, classified documents contained operational scenario information, but the software designers could not obtain the documents from the customer, because developers were not considered to have a need to know. There

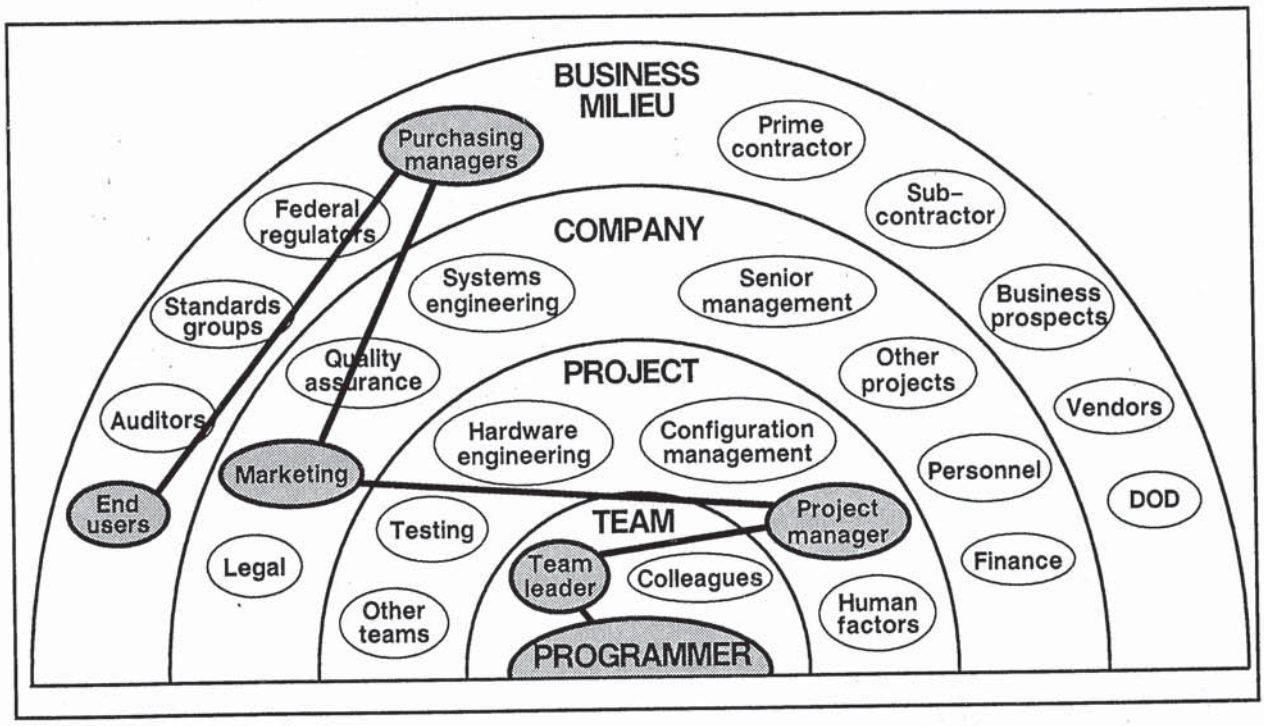

Figure 4: The long path from the programmer to the end user.

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.

Large projects required extensive communication that was not reduced by documentation. 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. However, even on commercial projects information was occasionally withheld from a development group for reasons ranging from political advantage to product security.

Project staff found the dialectic process crucial for clarifying issues. Particularly during early project phases, teams spent considerable time defining terms, co-ordinating 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. The complexity of the customer interface hindered the establishment of stable requirements, and increased the communication and negotiation costs of the project.

2.4 Software development questions for organizational psychologists

The paradigm of organizational behaviour provides multiple levels for analysing external impacts on programming teams. These levels include the parent organization, the local division, the customers, the business marketplace, the management structure, the administrative procedures, the physical environment, the psychological environment, the professional environment, etc. Although there is a large body of empirical research on organizational behaviour (Dunnette, 1976), little of it has been performed in programming organizations. Most current thinking on organizational processes in software development comes from accounts in books like those by Weinberg (1971), Brooks (1975), Kidder (1981), Fox (1982) and DeMarco and Lister (1987). This area badly needs new research, especially with the growing size and complexity of software products, and their effect on the size and complexity of the organizations that build them. Some of the most important questions concerning how organizational processes affect software development are:

* What is the most effective way to structure a programming organization, tall (many layers of management) or flat (large span of management control)?

* Does matrix management provide greater control over programming projects or does it reduce motivation by denying programmers a sense of ownership and pride in the software product?

* Under what conditions does a technical career path which parallels that of the management path help retain top programming talent?

* What are the major factors affecting the morale and climate of programming organizations?

* Which programming functions (e.g. testing, quality assurance, documentation, etc.) should be managed in groups separated organizationally from the software development group?

* How should the physical environment be arranged to maximize performance?

* Under what conditions will flex hours, quality circles, and other quality-of-work-life techniques be effective in software oganizations?

* How should development organizations interact with their customers, especially when the customer is a complex organization like the US Department of Defense?

Introductions to these issues can be found in many books on organizational behaviour and industrial psychology, and the results of current research can be perused in such journals as *Administrative Science Quarterly,* the *Journal of Occupational Psychology,* the *Journal of Applied Psychology, Personnel Psychology, Organizational Behavior and Human Decision Processes, Human Relations,* the *Journal of Management,* the *Academy of Management Journal* and the *IEEE Transactions on Engineering Management.*

3 Conclusion

Programming in the large is, in part, a learning, negotiation, and communication process. These processes have only rarely been the focus of psychological research on programming. The fact that this field is usually referred to as the 'psychology of programming' rather than the 'psychology of software development' reflects its primary orientation to the coding phenomena that constitute rarely more than 15% (Jones, 1986) of a large project's effort. As a result, less empirical data has been collected on the team and organizational aspects of software development.

Although there are cognitive questions involved in programming in the large (Curtis *et al.*, 1988), they must be investigated with an understanding of the social and organizational processes involved. For instance, given the amount of knowledge to be integrated in designing a large software system, and given the inability of current technology to automate this integration (Rich and Waters, 1988), the talent available to a project is frequently the most significant determinant of its productivity (Boehm, 1981; McGarry, 1982). But contributions by good people come not just from their ability to design and implement programs. Good people must become involved in myriad social and organizational processes such as 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.

The constant need to share and integrate information on a software project suggests that just having smart people is not enough. The communication needed to develop a shared vision of the system's structure and function, and the co-ordination needed to support dependencies and manage changes on large system projects are team and organizational issues. Individual talent operates within the framework of these larger social and organizational processes. The influence of exceptional designers is exercised through their impact on other project members, and through their ability to create a shared vision to organize the team's work (Brooks, 1975). Recruiting and training must be coupled with *team building* (Thamhain and Wilemon, 1987) to translate individual talent into project success. Thus, the impact of processes at one level of the layered behavioural model presented in Figure 1 must be interpreted by their impact on processes at other levels.

The requirements levied on advanced software systems are growing rapidly (Boehm, 1987). Larger organizations – often webs of organizations – are required for producing systems that can require well over 10 000 000 lines of code. Companies often bet their future in a particular market on their ability to create a large software system rapidly. For instance, in 1986 ITT Corporation sold off their traditional telecommunications business because of difficulties in developing System 12, a fully distributed digital switch. The issues of how to organize programmers into teams and how to manage large programming organizations are crucial to many companies' success. Unfortunately most of the information available for guiding decisions about social and organizational structure in software organizations is anecdotal or is an extrapolation of results found with student teams in laboratories. The scientific community performing empirical research on programming must approach software development from more than a cognitive paradigm, and should begin performing research on the social and organizational aspects. At the same time corporations must make their software development organizations available for research, so that they

are not forced to discount results obtained on undergraduate student teams. Better research on team and organizational factors may increase our ability to account for variation in software productivity and quality, and thus our ability to manage large systems development.

References

Adams, J. S. (1976). The structure and dynamics of behavior in organizational boundary roles. *In* M. D. Dunnette (Ed.), *Handbook of Industrial and Organization Psychology.* Chicago: Rand-McNally, pp. 1175-1199.

Allen, T. J. (1970). Communication networks in R&D laboratories. *R&D Management,* **1(1)**, 14-21.

Allen, T. J. (1986). Organizational structure, information technology, and R&D productivity. *IEEE Transactions on Engineering Management,* **33(4)**, 212-217.

Baker, F. T. (1972). Chief programmer team management of production programming. *IBM Systems Journal,* **11(1)**, 56-73.

Basili, V. R. and Reiter, R. W. (1981). A controlled experiment quantitatively comparing software development approaches. *IEEE Transactions on Software Engineering,* **7(3)**, 299-320.

Benbasat, I., Goldstein, D. K. and Meand, M. (1987). The case research strategy in studies of information systems. *MIS Quarterly,* **11(3)**, 369-386.

Boehm, B. W. (1981). *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall.

Boehm, B. W., Gray, T. E. and Seewaldt, T. (1984). Prototyping versus specifying: a multiproject experiment. *IEEE Transactions on Software Engineering,* **10(3)**, 290-302.

Boehm, B. W. (1987). Improving software productivity. *IEEE Computer,* **20(9)**, 43-57.

Bouchard, T. J. (1976). Field research methods. *In* M. D. Dunnette (Ed.), *Handbook of Industrial and Organization Psychology.* Chicago: Rand-McNally, pp. 363-413.

Brooks, F. P. (1975). *The Mythical Man-Month.* Reading, MA: Addison-Wesley.

Brooks, F. P. (1987). No silver bullet. *IEEE Computer,* **20(4)**, 10-19.

Curtis, B. (Ed)., (1985). *Human Factors in Software Development,* 2nd edn. Washington, DC: IEEE Computer Society.

Curtis, B. (1988). Five paradigms in the psychology of programming. *In* M. Helander (Ed.), *Handbook of Human-Computer Interaction.* Amsterdam: Elsevier North-Holland, pp. 87-105.

Curtis, B., Soloway, E., Brooks, R., Black, J., Ehrlich, K., and Ramsey, H. R. (1986). Software psychology: the need for an interdisciplinary program. *Proceedings of the IEEE,* **74(8)**, 1092-1106.

Curtis, B., Krasner, H. and Iscoe, N. (1988). A field study of the software design process for large systems. *Communications of the ACM,* **31(11)**, 1268-1287.

Curtis, B., Sheppard, S. B., Kruesi-Bailey, E., Bailey, J. and Boehm-Davis, D. (1989). Experimental evaluation of software documentation formats. *Journal of Systems and Software,* **9(1)**, 1-41.

Dailey, R. C. (1978). The role of team and task characteristics in R&D team collaborative problem solving and productivity. *Management Science,* **24(15)**, 1579-1588.

DeMarco, T. and Lister, T. A. (1987). *Peopleware.* New York: Dorset.

Dunnette, M.D.(Ed)., (1976). *Handbook of Industrial and Organization Psychology.* Chicago: Rand-McNally.

Fox, J. M. (1982). *Software and Its Development.* Englewood Cliffs, NJ: Prentice-Hall.

Hastie, R. (1987). Experimental evidence on group accuracy. *In* G. Owen and B. Grofman (Eds), *Information Processing and Group Decision-Making.* Westport, CT: JAI Press, pp. 129-157.

Jones, C. (1986). *Programming Productivity.* New York: McGraw-Hill.

Kernaghan, J. A. and Cooke, R. A. (1986). The contribution of the group process to successful group planning in R&D settings. *IEEE Transactions on Engineering Management,* **33(3)**, 134-140.

Kidder, T. (1981). *The Soul of a New Machine.* Boston: Little, Brown.

Kling, R. and Scacchi, W. (1982). The web of computing: Computer technology as social organization. *Advances in Computers,* vol. 21. Reading, MA: Addison-Wesley, pp. 1-90.

Kraft, P. (1977). *Programmers and Managers: The Routinization of Computer Programming in the United States.* New York: Springer-Verlag.

Lemos, R. S. (1979). An implementation of structured walkthroughs in teaching Cobol programming. *Communications of the ACM,* **22(6)**, 335-340.

Lucas, H. C. (1984). Organization power and the information services department. *Communications of the ACM,* **27(1)**, 58-65.

Mantei, M. (1981). The effect of programming team structures on programming tasks. *Communications of the ACM,* **24(3)**, 106-113.

McGarry, F. E. (1982). What have we learned in the last six years? *Proceedings of the Seventh Annual Software Engineering Workshop* (SEL-82-007). Greenbelt, MD: NASA-GSFC.

Mills, H. D. (1971). *Chief Programmer Teams: Principles and Procedures.* Technical Report IBM-FSC 71-5108. Gaithersburg, MD: IBM Federal Systems Division.

Myers, G. J. (1978). A controlled experiment in program testing and code walkthroughs/inspections. *Communications of the ACM,* **21(9)**, 760-768.

Rich, C. and Waters, R. C. (1988). Automatic programming: Myths and prospects. *IEEE Computer,* **21(8)**, 40-51.

Rogers, E. M. and Kincaid, D. L. (1981). *Communication Networks: Toward a New Paradigm for Research.* New York: Free Press.

Scacchi, W. (1984). Managing software engineering projects: A social analysis. *IEEE Transactions on Software Engineering,* **10(1)**, 49-59.

Scott, R. F. and Simmons, D. B. (1975). Predicting programming group productivity: A communicatons model. *IEEE Transactions on Software Engineering,* **1(4)**, 411-414.

Sells, S. B. (1963). An interactionist looks at the environment. *American Psychologist,* **18 (11)**, 696-702.

Sells, S. B. (1966). Ecology and the science of psychology. *Multivariate Behavioral Research,* **1**, 131-144.

Shaw, M. E. (1971). *Group Dynamics: The Psychology of Small Group Behavior.* New York: McGraw-Hill.

Shneiderman, B. (1980). Group processes in programming. *Datamation,* **26(1)**, 138-141.

Srinivasan, A. and Kaiser, K. M. (1988). Relationships between selected organizational factors and systems development. *Communications of the ACM,* **30(6)**, 556-562.

Swanson, E. B. and Beath, C. M. (1988). The use of case study data in software management research. *Journal of Systems and Software,* **8(1)**, 63-71.

Thamhain, H. J. and Wilemon, D. L. (1987). Building high performance engineering project teams. *IEEE Transactions on Engineering Management,* **34(3)**, 130-137.

Tushman, M. L. (1977). Special boundary roles in the innovation process. *Administrative Science Quarterly,* **22(4)**, 587-605.

von Mayrhause, A. (1984). Selecting a software development team structure. *Journal of Capital Management,* **2(3)**, 207-225.

Vosburgh, J., Curtis, B., Wolverton, R., Albert, B., Malec, H., Hoben, S. and Liu, Y. (1984). Productivity factors and programming environments. *Proceedings of the Seventh International Conference on Software Engineering.* Washington, DC: IEEE Computer Society, pp. 143-152.

Walston, C. E. and Felix, C. P. (1977). A method of programming measurement and estimation. *IBM Systems Journal,* **16(1)**, 54-73.

Walz, D. (1988). A longitudinal study of group design of computer systems. Unpublished Doctoral Dissertation. Austin: Department of Management Science and Information Systems, The University of Texas.

Walz, D., Elam, D., Krasner, H. and Curtis, B. (1987). A methodology for studying software design teams: An investigation of conflict behaviors in the requirements definition phase. *In* G. Olsen, E. Soloway and S. B. Sheppard (Eds), *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex, pp. 83-99.

Weinberg, G. M. (1971). *The Psychology of Computer Programming.* New York: Van Nostrand Reinhold.