ne organ there, flipping those keys up and down. There ought to be a etter way."

"There is."

"Yeah? What's that?"

"I'm working on it right now. We're doing it on the West Coast, but ne only machine is here, so we have to come here to debug. It's called Monitor System."

"Monitor System?"

"Yes, it's sort of an automatic operator. Takes the programmer away rom the machine. For example, I've got thirty jobs here which I'm going o run in my fifteen minute shot."

"Thirty? You're pulling my leg!"

"No, I mean it. By eliminating the operator, we achieve a fantastic peedup—and eliminate set-up and tear-down time, too."

"Well, good luck. But I don't see how I'm going to debug my programs f I can't be at the console. As it is I'm going crazy waiting for these—vhat was that name?—FORTRAN guys. See you later. I'm up now."

A surprising amount of useful information was transmitted in this way. Operating systems eliminated this social structure in the same way as automatic elevators eliminated the other. Still, if there is a common room right next to the place where computer output is returned, useful mixing can take place there. Personalized delivery services, however, tend to isolate the programmer from this type of interaction, and terminal systems for remote-job-entry and exit may make his isolation worse. This aspect of terminal operations is probably going to be a curse, not a blessing.

ERROR AND EGO

Many programmers who have read this far will be surprised at the emphasis placed on the social interaction among programmers. Programming—perhaps more than any other profession—is an individual activity, depending on the abilities of the programmer himself, and not upon others. What difference can it make how many other programmers you run into during the day? If asked, most programmers would probably say they preferred to work alone in a place where they wouldn't be disturbed by other people.

The ideas expressed in the preceding paragraph are possibly the most formidable barrier to improved programming that we shall encounter. First of all, if this is indeed the image generally held of the programming profession, then people will be attracted to, or repelled from, entering the profession according to their preference for working alone or working with others. Social psychologists tell us that there are different personality

types—something we all knew, but which is nice to have stamped with authority. Among the general personality traits is one which is measured along three "dimensions"—whether a person is "compliant," "aggressive," or "detached." The compliant type is characterized by the attitude of liking to "work with people and be helpful." The aggressive type wants to "earn money and prestige," and the detached type wants to "be left to myself to be creative."

Now, every person contains a mixture of these attitudes, but most people lean more heavily in one direction than the others. There is no doubt that the majority of people in programming today lean in the "detached" direction, both by personal choice and because hiring policies for programmers are often directed toward finding such people. And, to a great extent, this is a good choice, because a great deal of programming work is "alone and creative."

Like most good things, however, the "detachment" of programmers is often overdeveloped. Although they are *detached* from people, they are *attached* to their programs. Indeed, their programs often become extensions of themselves—a fact which is verified in the abominable practice of attaching one's name to the program itself—as in Jules' Own Version of Algol, better known as JOVIAL. But even when the program is not officially blessed with the name of its creator, programmers *know* whose program it is.

Well, what is wrong with "owning" programs? Artists "own" paintings; authors "own" books; architects "own" buildings. Don't these attributions lead to admiration and emulation of good workers by lesser ones? Isn't it useful to have an author's name on a book so we have a better idea of what to expect when we read it? And wouldn't the same apply to programs? Perhaps it would—if people read programs, but we know that they do not. Thus, the admiration of individual programmers cannot lead to an emulation of their work, but only to an affectation of their mannerisms. This is the same phenomenon we see in "art colonies," where everyone knows how to look like an artist, but few, if any, know how to paint like one.

The real difficulty with "property-oriented" programming arises from another source. When we think a painting or a novel or a building is inferior, that is a matter of taste. When we think a program is inferior—in spite of the difficulties we know lurk behind the question of "good programming"—that is a matter at least potentially susceptible to objective proof or disproof. At the very least, we can put the program on the machine and see what comes out. An artist can dismiss the opinions of a critic if they do not please him, but can a programmer dismiss the judgment of the computer?

On the surface, it *would* seem that the judgment of the computer is

indisputable, and if this were truly so, the attachment of a programmer to his programs would have serious consequences for his self-image. When the computer revealed a bug in his program, the programmer would have to reason something like this:

"This program is defective. This program is part of me, an extension of myself, even carrying my name. *I* am defective."

But the very harshness of this self-judgment means that it is seldom carried out.

Starting with the work of the social psychologist Festinger, a number of interesting experiments have been performed to establish the reality of a psychological phenomenon called "cognitive dissonance." A classical experiment in cognitive dissonance goes something like this:

> Two groups of subjects are asked to write an essay arguing in favor of some point with which they feel strong disagreement. One group is paid one dollar apiece to write this argument against their own opinions, the other is paid twenty dollars apiece. At the end of the experiment, the subjects are retested on their opinions of the matter. Whereas "common sense" would say that the twenty dollar subjects—having been paid more to change their minds—would be more likely to change their opinions, cognitive dissonance theory predicts that it will be the *other* group which will change the most. Dozens of experiments have confirmed the predictions of the theory.

The argument behind cognitive dissonance theory is quite simple. In the experiment just outlined, both groups of subjects have had to perform an act—writing an essay against their own opinions—which they would not under ordinary circumstances like to do. Arguing for what one does not believe is classed as "insincerity" or "hypocrisy," neither of which is highly valued in our society. Therefore, a *dissonance* situation is created. The subject's self-image as a sincere person is challenged by the objective fact of his having written the essay. Dissonance, according to the theory, is an uncomfortable and unstable state for human beings, and must therefore be quickly resolved in one way or another. To resolve a dissonance, one factor or another contributing to it must be made to yield. Which factor depends on the situation, but, generally speaking, it will *not* be the person's self-image. That manages to be preserved through the most miraculous arguments.

Now, in the experiments cited, the twenty dollar subjects have an easy resolution of their dissonance. "Of course," they can say to themselves or to anyone who might ask, "I didn't really believe those arguments. I just did it for the money." Although taking money to make such arguments is not altogether the most admirable trait, it is much better than actually holding the beliefs in question. But look at the quandry of the dollar group. Even for poor college students—and subjects in psychological

experiments are almost always poor college students—one dollar is not a significant amount of money. Thus, the argument of the other group does not carry the ring of conviction for them, and the dissonance must be resolved elsewhere. For many, at least, the easiest resolution is to come to admit that there is really something to the other side of the argument after all, so that writing the essay was not hypocrisy, but simply an exercise in developing a fair and honest mind, one which is capable of seeing both sides of a question.

Another application of the theory of cognitive dissonance predicts what will happen when people have made some large commitment, such as the purchase of a car. If a man who has just purchased a Ford is given a bunch of auto advertisements to read, he spends the majority of his time reading about Fords. If it was a Chevrolet he purchased, then the Chevrolet ads capture his attention. This is an example of anticipating the possibility of dissonance and avoiding information that might create it. For if he has just purchased a Ford, he doesn't want to find out that Chevrolet is the better car, and the best way to do that is to avoid reading the Chevrolet ads. In the Ford ads, he is not likely to find anything that will convince him that he is anything but the wisest of consumers.

Now, what cognitive dissonance has to do with our programming conflict should be vividly clear. A programmer who truly sees his program as an extension of his own ego is not going to be trying to find all the errors in that program. On the contrary, he is going to be trying to prove that the program is correct—even if this means the oversight of errors which are monstrous to another eye. All programmers are familiar with the symptoms of this dissonance resolution—in others, of course. The programmer comes down the hall with his output listing and it is very thin. If he is unable to conceal the failure of his run, he makes some remark such as

"Those keypunch operators did it again."

or

"The operator put my cards in out of sequence."

or

"When are we going to get that punch fixed so it duplicates properly?"

There are thousands of variations to these plaints, but the one thing we never seem to hear is a simple

"I goofed again."

Of course, where the error is more subtle than a complete failure to get output—which can hardly be ignored—the resolution of the dissonance can be made even simpler by merely failing to see that there is an error. And let there be no mistake about it: the human eye has an almost infinite capacity for not seeing what it does not want to see. People who have specialized in debugging other people's programs can verify

this assertion with literally thousands of cases. Programmers, if left to their own devices, will ignore the most glaring errors in their output—errors that anyone else can see in an instant. Thus, if we are going to attack the problem of making good programs, and if we are going to start at the fundamental level of meeting specifications, we are going to have to do something about the perfectly normal human tendency to believe that ones "own" program is correct in the face of hard physical evidence to the contrary.

EGOLESS PROGRAMMING

What is to be done about the problem of the ego in programming? A typical text on management would say that the manager should exhort all his programmers to redouble their efforts to find their errors. Perhaps he would go around asking them to show him their errors each day. This method, however, would fail by going precisely in the opposite direction to what our knowledge of psychology would dictate, for the average person is going to view such an investigation as a personal trial. Besides, not all programmers have managers—or managers who would know an error even if they saw one outlined in red.

No, the solution to this problem lies not in a direct attack—for attack can only lead to defense, and defense is what we are trying to eliminate. Instead, the problem of the ego must be overcome by a restructuring of the social environment and, through this means, a restructuring of the value system of the programmers in that environment. Before we discuss how this might be done, let us look at some examples of what has happened when it has been done—how it affects the programmers and their programs.

First of all, let no one imagine that such restructuring is the ivory tower dream of social theorists. Programming groups who have conquered the ego problem do exist and have existed from the earliest days of computing. John von Neumann himself was perhaps the first programmer to recognize his inadequacies with respect to examination of his own work. Those who knew him have said that he was constantly asserting what a lousy programmer he was, and that he incessantly pushed his programs on other people to read for errors and clumsiness. Yet the common image today of von Neumann is of the unparalleled computing genius—flawless in his every action. And indeed, there can be no doubt of von Neumann's genius. His very ability to realize his human limitations put him head and shoulders above the average programmer today.

Average people can be trained to accept their humanity—their inability to function like a machine—and to value it and work with others so as to keep it under the kind of control needed if programming is to be successful. Consider the case of Bill G. who was working in one of the early space tracking systems. His job was to write a simulator which would simulate the entire network of tracking stations and other real-time inputs. His system had to check out the rest of the system in real-time without having to have the worldwide network on-line. The heart of the simulator was to be a very small and very tight loop, consisting, in fact, of just thirteen machine instructions. Bill had worked for some time on this loop and when he finally reached the point of some confidence in it, he began looking for a critic—the standard practice in this programming group.

Bill found Marilyn B. willing to peruse his code in exchange for his returning the favor. This was nothing unusual in this group; indeed, nobody would have thought of going on the machine without such scrutiny by a second party. Whenever possible an exchange was made, so nobody would feel in the position of being criticized by someone else. But for Bill, who was well schooled in this method, the protection of an exchange was not necessary. His value system, when it came to programming, dictated that secretive, possessive programming was bad and that open, shared programming was good. Errors that might be found in code he had written—not "his" code, for that terminology was not used here—were simply facts to be exposed to investigation with an eye to future improvement, not attacks on his person.

In this particular instance, Bill had been having one of his "bad programming days." As Marilyn worked and worked over the code—as she found one error after another—he became more and more amused, rather than more and more defensive as he might have done had he been trained as so many of our programmers are. Finally, he emerged from their conference announcing to the world the startling fact that Marilyn had been able to find *seventeen* bugs in only thirteen statements. He insisted on showing everyone who would listen how this had been possible. In fact, since the very exercise had proved to him that this was not his day for coding, he simply spent the rest of the day telling and retelling the episode in all its hilarious details.

Marilyn, at the same time, did not feel any false confidence in her own work on the problem, for—she reasoned correctly—where there had been seventeen errors, there were probably a few more. In particular, she knew that after a certain amount of time working on the code, she had internalized it as much as had Bill, even though she had not written it originally. So she in turn went looking for a critic; and while Bill was giving everyone an enormous laugh at his expense, Marilyn and others managed to find three more errors before the day was over.

As an epilogue to this incident, it should be noted that when this code was finally put on the computer, no further errors were found, in spite

of the most diabolical testing possible. In fact, this simulator was put into use in more than a dozen installations for real-time operations, and over a period of at least nine years no other errors were ever found. How different might have been the story had Bill felt that each error found in that code was a wound in his pride—an advertisement of his stupidity.

This incident is not an isolated case, and this group is not unique. Why, then, are such groups not more conspicuous? Why is the practice of "egoless programming" not more widespread? A number of factors might be invoked to account for the impression that such groups are rare. First of all, many of the successful software firms are based on this type of interaction, and though they will admit to it under direct questioning, they often regard this knowledge as valuable proprietary information. Secondly, groups working in this way tend to be remarkably satisfied and stable, so that the programmers we find wandering from installation to installation are not likely to have come from such a group. Moreover, these gypsy programmers—to achieve a constantly escalating salary range—must encourage the myth that the best programming is the product of genius, and nothing else.

Another reason these methods are not better known is that nobody has ever experimented on the difference in quality of work produced by this method and the method of isolated individual programmers. Some experiments have been performed on factors affecting programmer productivity, but these have suffered first of all from emphasis on the mechanical aspects of programming, not the social. For example, a study will be made comparing time sharing with batch processing or language A with language B, because someone is trying to prove that he should be allowed to develop a time-sharing system or a compiler for language B. The people who run these experiments seem to take for granted the individual nature of programming effort—for that is probably the way they have always operated. Besides, things are complicated enough working with individuals. When you compare system X and system Y and find out that 90 percent of the variance in your experiment comes from individual programmer differences, who wants to add the complication and expense of studying group performance?

An interesting anecdote—which we mentioned briefly in the chapter on methods—can be told about one of our studies that tried to assess the difference in programming results obtained when different programmers were given slightly different impressions of what they were to achieve—efficient coding or quick completion. As usual, individual subjects were employed, but one of these subjects—they were all students on a special three-month course—happened to come from a group that practiced egoless programming. At a certain point, he came to me and said that he

had reached the point in his work where he needed someone to look over what he had done. As the object of the experiments was not to study differences between group work and individual work, I was forced—against my own beliefs—to request that the subject try to proceed without outside assistance, which would only add to the variance of the experiment.

As a sidelight to this incident, it should be noted that this programmer's work seemed to the evaluators to be better organized and better executed than the other four programmers working on the same problem. In discussing this question with him, he raised the point that he had worked throughout as he always did in his own group—always with an eye to making the program clear and understandable to the person or people who would ultimately have to read it. This insight indicates that all the advantages of egoless programming are not confined to the detection of errors, though that was perhaps the earliest and strongest motivation for adopting the technique. In fact, it might be useful to examine our four factors in good programming in the light of what effect this method would have on them.

For meeting specifications, the value is quite clear. On the matter of scheduling, the effect on the mean time to complete programs is not immediately evident, but the effect on the variation should be clear from our example of the bugged simulator. If it is true that programmers have bad coding days—and this seems supported from a number of sources—then a piece of code written on one of these days is going to have an extra long debugging cycle. In the case of Bill G.'s program, the twenty bugs might have taken several weeks to root out. Moreover, it seems likely that at least one of them might have survived in the system past the time when this piece was integrated with other pieces—in which case the schedule of other parts would have been adversely, or at least unpredictably, affected.

Not only is the variation in debugging time reduced, but since there is more than one person familiar with the program, it is easier to get realistic estimates on the amount of real progress that has been made. It is not necessary to rely on a single judgment—and of the person least likely to be unbiased, at that. The adaptability of programs is also improved, for we are assured that at least two people are capable of understanding the program. Under certain programming circumstances, this represents an infinite improvement. Also, the entire work is less susceptible to being disturbed if one of the involved programmers happens to be sick, or pregnant, or otherwise missing, as programmers are wont to be. This not only reduces variations in schedule, but also makes it more likely that at some time in the future, when the code must be modified, someone will be around who knows something about it.

On the question of efficiency, we can make no hard and fast statements. There certainly seems to be no reason why programs developed in this way should be *less* efficient than other programs. By having a second party look at the program, it would seem that we increase the possibility of eliminating at least the most obviously inefficient areas, although overall efficiency is usually going to be primarily influenced by the original structure chosen.

One final advantage of this method lies in the effect it has on the person reading the program of someone else, for, if we are correct in assessing the value of reading programs, he cannot help but become a better programmer for the exercise. We shall have more to say on this subject under the heading of programmer training, but it does seem that the general level of competence of such a group is likely to raise itself even in the absence of specific measures for education.

CREATING AND MAINTAINING THE PROGRAMMING ENVIRONMENT

The question of creating such a desirable environment as we have described is different from the question of maintaining such an environment once it exists. Maintenance is by far the easier task, for converting an existing group to this philosophy will usually run against the phenomenon of "locking" or "fixation" of social structures. Fixation occurs whenever a situation creates an environment favorable for maintaining that situation. For example, an FM tuner can be designed to center itself on the strongest signal in the vicinity of the tuning knob. Once a station is captured by this device, only a very strong change can get it off, because every small tendency to change is met by a compensating action by the tuner. Such locking occurs in all sorts of systems—physiological, electronic, biological, but especially for our immediate purposes, social.

One typical computing example of social fixation is the adoption of one programming language by an installation. Once the language has been adopted, a new language has more difficulty making an entry, because with most of the people using the old language, advantages accrue to following the beaten path. If one needs advice, it is more easily found. If one needs subroutines, they are more likely to exist. Scheduling of computer runs may favor the commonly used language, keypunchers will make fewer errors punching familiar coding, and procedures for using the old language will be smoother and better developed.

In the same way that an installation fixates on a programming language, it can establish a general social environment which either encourages or discourages egoless programming. When a new programmer

enters the mileu, his attitudes may be shaped by the reactions of the others already there. If he goes to somebody for advice and he is ridiculed for the stupidity of his errors, he is less likely to seek assistance the next time. If, however, someone comes to him and asks for help in looking over a program, he is flattered by the implied compliment to his ability and may not feel so threatened when he has to seek advice. To a large extent, we behave the way we see people behaving around us, so a functioning programming group will tend to socialize new members to its philosophy of programming.

Sometimes the group may have to maintain its philosophy in the face of a larger threat than the introduction of a new member or two. Adherents of the egoless programming philosophy are frequently subjected to threatening moves on the part of managers from higher levels in their organization. Managers tend to select themselves from the "aggressive" component of society and have difficulty appreciating the fact that other people do not completely share their goals of money and prestige. They are especially at a loss to understand the smooth functioning of a programming group based on mutual respect for individual talent and cooperation in the common cause. Instead, they tend to view people as working for money or under threat—as they themselves do.

A particularly pertinent example of the clash between aggressive management and a compliant-detached programming group occurred in a software section of one of the computer manufacturers. One group in the section, acting as a team, had been particularly successful at producing an entirely new system, which promised to have much market potential. The achievement of this group was so evident that the management of the company decided to give them a cash award. In typical management fashion, they gave the award to the person who had been designated as the group's manager. Imagine their bewilderment when he told them that he could not accept the award unless it was given to all.

His reaction, in view of the way the group shared its work, was perfectly correct, but was not understood by the management. Some managers thought that he was maneuvering for more money; others thought he was trying to set up a "prima donna" group. In any case, it was decided to force him to accept the award and also to break up the group—which seemed to have unhealthy ideas. He took the award and promptly split it equally among all the group members, after which the group left the company *en masse* and went to work for an independent software firm.

In this case, the group protected itself against a great outside threat by picking up and leaving. Had the management been more aware of what this group had to offer, and had they been more flexible, they might have worked out a solution that would have permitted this group to

influence the work of other groups in a favorable way. But this is not easily done by managers, who tend to feel that when work gets done it is the direct result of the actions of some leader of outstanding ability. Even when the manager appreciates the work of the group, it is not consistent with his own philosophy to see the productivity of the group as a property of the group, not as a sum of the contributions of the individual members.

In one interesting case, a group of ten programmers had come to work in the programming section of a large airframe manufacturer. They had worked together in another firm for about two years, but that firm had decided to centralize its data-processing facilities on the other side of the country. These programmers had been unwilling to move. After each had received attractive offers from several firms, they decided to go with the one that would hire them all. Mass movements of groups of this type are not rare, and though management tends to see them as some sort of conspiracy, they are usually motivated more by the desire to continue receiving the fulfillment of working together than any idea of enormous material gain—such is the powerful influence of a truly receptive working environment.

Some months after this group had moved to the airframe manufacturer, their new manager happened to meet their former manager at a computer conference and asked him if he had any more people like those. Upon being told that he had the entire group, he inquired about the secret that had been used in finding these people. Their old manager could think of nothing special—one girl had been a new college graduate majoring in Italian, one man had taught mathematics in a high school for seven years, one man was a professional engineer, one girl had been a business school graduate who had worked for several years as an executive secretary and accountant. Why, he wanted to know, had the other manager asked?

"I don't know what it is," came the honestly puzzled reply, "but since those people have come to our place, I've discovered that whenever there is a job that really has to be done right and on time, I give it to one of them. And I have 300 other programmers, but if I want it done right, that's who I ask to do it. They must be some kind of geniuses." His perception was so obviously colored by what he expected to see that he could not bring himself to understand that whenever he gave a job to one member of the group, it was worked on by all of them in their usual fashion. When they tried to explain their methods to him, he understood them to be covering up the fact that a few of the members were doing all the work and carrying the others. Fortunately, he was not so rigid that he had to break up the only satisfactory operation he had merely because he didn't understand how it functioned. Unfortunately, he was unable to

see how their successful methods could be transmitted to others in his group, so the group remained an isolated pocket until the time it moved on—again as a group—to a more understanding environment.

Of course, groups that follow the individualistic school of programming also have a way of preserving themselves—as did the remainder of the programming section at this airframe manufacturer. A single new member, or even a single new group within so large a group, really has no chance of converting the social system, even if he is firmly convinced of the correctness of his way of doing things. If he comes into an established group, he will probably change his ways to theirs, eventually, though after experiencing more psychological hardship than one might like. If the group is newly forming, however, as programming groups often are, and if it is forming from disparate elements, he may struggle to shape the group to his image and then leave if unsuccessful.

A case in point was Jim A., who was brought onto a newly forming project in Chicago from a programming center in New York. The group to which he was assigned was headed by two people who had been firmly brought up in the egoless programming tradition and who were determined to propagate that tradition in this project. The group consisted of these two, Jim, and four trainees. On the first day the group assembled, the group leaders began the indoctrination of the others into their method of working. It was decided at the beginning that each of the group members was to have the signature of one of the other members on his run request before going on the computer with any job. By this slightly formal method, they hoped to ensure that the group members would get in the habit of doing what they would come to do spontaneously later on.

During the meeting, Jim said nothing, but when the trainees had left, he approached the group leaders. "That's an interesting idea you have," he began, "to help those trainees learn the ropes."

"Well," it was patiently explained to him, "it's not just for the trainees. It's for all of us, so we don't start slipping into bad habits."

"You can't be serious," Jim laughed. "Why I have more than two years of experience. *I* certainly don't need anyone looking over *my* work. What could those trainees possibly teach me?"

Like most prophecies, this one had a way of fulfilling itself, and Jim managed to evade the falling of other eyes upon his sacred programs through one device or another. Before very long, his presence in the group became clearly counterproductive. As he saw the trainees advancing to do difficult and challenging assignments while he struggled on alone, he tried ridiculing them for their lack of independence and ability to think for themselves. His own programs were not up to the standard of quality which the rest of the group was producing. When

the group leader finally felt forced to turn over one of his programs—which Jim claimed was debugged, or "essentially debugged"—to one of the trainees to clear up, Jim had more than he could take and resigned.

In this case, the social environment of the group had been strong enough to shape the behavior of the trainees; but it was not strong enough to counteract Jim's "two years of experience." As his personality was not strong enough to carry the group in the direction he felt was correct, the situation eventually became intolerable. Perhaps if the group leaders had been more wise and experienced, they would have excused him from the group from the beginning, but the temptation of "two years of experience" proved their undoing—as it has to many others in a business which so sorely lacks experience.

SUMMARY

The environment in which programmers work is a rich and complex environment, full of human involvement, change, and misleading appearances. To understand that environment, one must understand the difference between formal and informal structures and the many factors that shape it, ranging from the physical surroundings to the individual ego. In an ongoing programming shop, the richness of this environment gives it a self-maintaining quality which resists changes imposed from the outside—especially changes imposed without an understanding of the difference between the formal and the informal. This self-maintenance is manifest on all social levels, and is neither inherently good nor inherently bad. It is merely a fact of programming life.

QUESTIONS

For Managers

1. Do you have an organization chart showing the organization below you and around you? Try taking a copy of this chart and marking—with wiggly lines—interactions that occur in your organization. Do the wiggly lines match the straight lines? If so, get out from behind your desk and find out what is really happening out there.
2. When was the last time you moved peoples' work locations? Can you recall any changed behavior from that move which was not part of the direct intention of the move? What would you have done differently if you were planning the move now?
3. Looking back over your interactions with programmers, can you think of things you have said that might have forced them into dissonant situations—situations in which their ego had to be defended? In those situations, was the resolution of the dissonance always in the direction you intended, or did you experience such reactions as covering up errors or schedule delays, rather than correcting for them? How could you have approached those situations so as to lessen the dissonance, or to direct the resolution of the dissonance in directions more useful to the overall goals of your organization?
4. What would you have to do to introduce egoless programming into your shop? What resistance would you expect to meet, and how would you deal with it? How long do you think it would take, and what are the chances of success?
5. What is your honest opinion of people who are not trying to "move up" in your organization, but who seem satisfied with the kind of work they do and the amount of money they get? To what extent is your view influenced by your own feelings for yourself?

For Programmers

1. If a computing center had perfectly consistent turnaround, there would be no need for an informal organization to produce information on when jobs are ready. In what other ways do the variations induced by the complexity of programming lead to the growth of informal social structures? Give some examples from your own experience.
2. If you use a terminal system regularly, how do you exchange information with other users of the terminal system? Does your terminal system have an operation which enables you to exchange messages with other terminal users? If so, how valuable is this facility for real communication, as opposed to the other methods you use?
3. Do you refer to your work as "my" program? Try passing one week without using the personal possessive in reference to programs, and take notes on the effects you observe.
4. Have you ever blamed other people for errors in "your" program? Have you ever blamed inanimate objects, such as keypunches or magnetic tapes? How many times were you right in blaming these people or things?
5. Have you ever blamed "bad luck" for errors in "your" program? How often? Are other programmers as unlucky as you? If not, why do you thinks the fates have singled you out for such ill treatment? What sort of rituals do you think you might follow to appease their anger with you?

BIBLIOGRAPHY

Lynch, Kevin, *The Image of the City*, Cambridge, M.I.T. Press, 1960.
In this small and insightful book, Lynch explores the ways in which our image of