EDP ANALYZER

APRIL, 1974
VOL. 12, NO. 4

ISSUES IN PROGRAMMING MANAGEMENT

Some observers of the computer field have been predicting the demise of the programmer as a job title, with the application programmer being the first to disappear. They say that new technology will take over the programming function as it is now known. These seers may be right eventually but there isn't much evidence to support their views as yet. It appears that programmers—and the management of programmers—will continue to be needed as far ahead as our crystal ball can see. But the management of programmers has been a challenge since the days of UNIVAC I, and new theories continue to be proposed. In this report, we consider three types of proposals that are currently being widely discussed. These are: conducting programming as a *team* effort, performing programming as an engineering-type activity, and certifying programmers. They each deserve consideration and—depending upon the circumstances—support.

"There is a serious and widespread debate underway on how the computer programming effort can best be conducted and managed." One of the more interesting characteristics of our field is that this statement could just as well have been made in 1955 or 1965 or 1974; it probably can be repeated in 1980. The fact is, the management of the programming function continues to be almost the same challenge that it was in the early days of our field.

Yes, there have been some improvements. There are more tools and techniques available for use. Programming skills have been sharpened by training. But projects have grown larger and more complex. So projects have continued to be completed behind schedule and over cost—if they are even completed at all—and programming continues to be on the critical path of most projects.

Currently, there are three types of proposals that have been advanced for improving programming performance and that are receiving widespread debate. These are:

- The structure of the programming function
- The engineering of software
- The certification of programmers

Some of these proposals seem to fit together well, and others are definitely at odds. We will present the main points of each of the major proposals, and will give references to further detailed discussions. It is quite possible that data processing management will be pressured to take a for-or-against stand on some of these proposals in the not-distant future. If you have not yet studied these proposals, we hope this report will be helpful for beginning that study.

The structure of the programming function

Some of the proposers say that a significant improvement in the programming function can come about only by a restructuring of that function. They say that the conventional structuring of the programming group leads to wasted motions, frustrations, dissatisfactions with pro-

gramming management, and such.

We see three basic types of structures now being proposed for the programming function:

- Multi-level hierarchies
- Fixed structure teams
- Adaptive teams

Multi-level hierarchy of individuals

The term "individuals" in the above heading requires some explanation. We are discussing programmers as individuals here, as opposed to close-knit *teams* of programmers. Later, too, we will mention a multi-level hierarchy of *teams*. A hierarchy of individuals is pretty much the way that programming groups have been organized over the years.

The Infotech State of the Art report on "Computing Manpower" (Reference 1) gives a good summary of recent thinking on programming management. Most of the discussion applies to the conventional hierarchical organization of the staff, and some of the new thoughts that people have had for improving the performance of such staffs. For instance, Joel Aron tells of methods that have been successfully used at the IBM Federal Systems Division for managing very large projects. Aron makes the point that he is not prescribing these methods for others; he is just telling what has worked for IBM.

At the IBM Federal Systems Division, they do contract work for the U.S. federal government, as well as for other organizations and agencies. The projects generally require from 2 to 6 years elapsed time, and involve from 3 to over 800 people each. The procedures that IBM has developed for managing such projects were the basis for a course taught to programming managers in IBM. From the guidelines used in that course, Metzger has put together a useful book, *Managing a Programming Project* (Reference 2).

Hierarchical projects use a strong chain of command, says Aron. The first level managers manage the employees—for the purposes of our discussion, the programmers. These first level managers must be technically qualified, and must be able to teach, demonstrate, monitor, and evaluate the work of the programmers under them. If a first level manager performs well, he is promoted to a second level manager. (Aron says that about one-half of the programmers are women, and that the number of women managers is rising rapidly.) The second level managers manage the first level managers. These second level managers must learn the technology of project management. They are mainly concerned with setting objectives, monitoring progress, and deciding what to do next.

In a large project, there are additional levels of managers, culminating in the project manager. The higher and higher levels gradually get further and further away from the programming details and techniques. Since they are called upon to make both administrative and technical decisions, technical advisers are provided for these higher level managers. A technical adviser is at roughly the same job level as the manager he advises.

Aron says that they have identified four main types of programmers. One type is the people who, by skill and experience, have shown that they are capable designers. They handle the major system and program design functions. Another type is the test and integration specialists. A third type is the support programmers. The bulk of the staff consists of the regular programmers, the fourth type.

Reference 2 gives a good discussion of the methods that IBM has used for managing large projects, in the type of environment just described.

Fixed structure teams

The best known example of fixed structure teams is the IBM Chief Programmer Team approach, described by Mills (References 1 and 16) and Baker (References 3 and 4). Interestingly, it too was developed at the IBM Federal Systems Division.

Mills makes the point that the chief programmer team in concept is like a surgical team. It consists of a nucleus of people, with the chief programmer having a role comparable to the surgeon; the chief programmer is supported by a backup programmer and a programmer/librarian, in much the same way that the surgeon is supported by nurses, anesthesiologist, and so on. Then, as required, specialists are brought in to help perform the work; in the case of programming team, this would be additional programmers and system analysts. The chief programmer, chosen for his skill and experience, is the architect and key coder of the program(s) being developed. He codes the control code for the top level mod-

ules and defines the interfaces. He then assigns the functional programming tasks to the other programmers on the team. The backup programmer aids the chief programmer. The programmer/librarian relieves the chief programmer of most of the clerical work, and is responsible for maintaining the program library and listings.

Baker discusses the experience of a chief programmer team in developing a large information retrieval system for the *New York Times* newspaper. The project required almost two years elapsed time, and involved 11 man-years of team member time. A total of ten people were on the team; four programmers, for instance, were team members for between three and five months each. A project manager, not actually a member of the team, handled contract details and relieved the chief programmer of many administrative matters.

This project was significant in that it produced usable, relatively bug-free code at a very high production rate. Some 83,000 lines of source code were produced; counting just the programmer time for unit design, programming, debugging, and testing, the rate was 65 lines of source code per programmer day. In all of this code, only 21 errors were found during exhaustive acceptance testing. In the next year and one-half of operation, only 25 more errors were found. The bulk of these errors were missing functions or misunderstandings.

All in all, this instance of the chief programmer team approach was very impressive in its accomplishments.

Adaptive teams

Weinberg (Reference 5) is the chief spokesman for the adaptive team; Ogdin (Reference 7) lends his support to the concepts.

Weinberg gives an eloquent discussion of the problems of programming and how his ideas on team programming help to solve those problems. He advocates a close-knit team, of perhaps six to ten people in size. The team has no assigned leader; rather, leadership shifts between the team members based upon the needs of the moment and the strengths of the individual team members (hence the term "adaptive"). For instance, when a good amount of debugging is occurring, if one of the team members is particularly good at debugging, he might assume the role of team leader during that period.

Perhaps the main feature of Weinberg's concepts, if one feature can be singled out, is that of "egoless programming." In a conventional programming organization, says Weinberg, a program is assigned to a programmer. The programmer is responsible for that program; he designs it, codes it, debugs it, and hopefully documents it. The program is a part of himself; if bugs are found in it after he has said it was completed, he treats these as an attack on him. In short, in such a situation, a programmer's programs are a part of his ego. Instead of this, says Weinberg, a team should practice egoless programming. Each team member should read the code of the other team members—to understand what is being done, and to catch flaws. A piece of completed code is then not the product of an individual team member but rather of the team as a whole.

There are other important features in what Weinberg proposes. One is democratic team management—since there is no assigned leader—for the handing out of assignments and for taking corrective actions. Work is assigned based on the strengths of the team members. In fact, the program(s) may be designed based on the skills of the people on the team. The team stays together; new people are not normally assigned to it for short periods of time. In a sense, the team is like a close-knit family, with a spirit of friendship tempering the professional criticism of each other's code that occurs.

Multi-level hierarchy of teams

Both of the team structures described above are based on teams of approximately eight members. Eight is not a hard and fast number; six to ten might be a good range. If the team gets much larger, it tends to split into cliques.

So what should be done for projects where a team of eight or so people clearly is inadequate to accomplish the job in the required time? Baker (Reference 3) proposes a hierarchy of teams. The top level team would perform the overall system architecture. It would then separate, with the individual team members becoming chief programmers of the second level teams. This process might be continued for additional levels, if the project is large enough.

While Baker believes that this approach will work, he is just now conducting his first test of it.

So any discussions of this approach would be conjectural.

We will limit our discussion, then, to the three types of structure that have been tested—the hierarchy of individuals, the fixed structure team, and the adaptive team. We will compare these three structures in terms of:

- The role of the individual programmer
- Improving programmer performance

Much of the information for this discussion comes from References 1 through 6.

The role of the individual programmer

With the *hierarchical* structure, management tends to look upon the individual programmers as interchangeable units. This statement is extreme, of course, since individual differences are recognized. But there is a tendency to look at the programming function as so many slots on the organization chart, to seek to hire "experienced programmers," and to shift programmers between projects as the workload requires.

Management in a hierarchical structure tends to be autocratic, in that management makes the decisions (but often on the advice of subordinates). Managers assign tasks to their subordinates, monitor progress, and make decisions on any corrective action that may be needed.

The individual programmers, in turn, are held responsible for their programs. Each is expected to design, code, debug, and document the programs assigned to him.

Management in such an organization probably would welcome the certification of programmers, as an aid in the hiring and in assigning jobs.

The career paths open to a programmer in such an environment are the well-known ones. The programmer may rise up the technical ladder to systems designer. Or he may rise up the management ladder, as a first level, then second level, etc. manager. Or he may work toward a technical advisory position offered in some organizations.

One of the problems faced by the management of such organizations is the "old at 35" programmer. He may be receiving a much higher salary than the newly hired 23 year old programmer. But he may not be as well versed in the new technology as the new programmer, and his experience may not equip him to produce much more than the new programmer. In such situations, subtle (?) actions might be taken to encourage such a programmer to leave. Since technological obsolescence can be avoided through planned professional development, the "old at 35" syndrome is really an indictment of the managers, says Aron.

In the *fixed structure team*, some of the elements of the hierarchy exist. The chief programmer is the assigned leader of the team. The backup programmer, as the name implies, supports the chief programmer and stands ready to take his place should the need arise. Also, it is expected that other programmers will be added to the team for a period of time, as the workload requires. It is usually desired, according to Baker, that these programmers be experienced, but positions might be provided for one or two novices.

The management of the fixed structure team would be mainly autocratic. The chief programmer assigns the tasks, monitors progress, decides on corrective action.

At the same time, the team can display some of the characteristics that Weinberg advocates. Baker, in Reference 4, claims that egoless programming was used on the *New York Times* project. Each programmer on the team read the code of the other programmers. Generally this was accomplished, he says, by providing each team member with the up-to-date listings of the code that has been produced; the programmer/librarian saw that this was accomplished. In addition, Baker feels that the chief programmer and backup programmer should read all of the code produced, to catch errors, check for compliance with standards, and so on.

As far as career paths are concerned, Mills (Reference 1) makes an interesting point. The members of a surgical team, excepting the surgeon, do not plan to become surgeons. They plan to stay in their own lines of work and develop their proficiency. So it should be with the members of the programming team, says Mills. True, the backup programmer should be a potential chief programmer, and some of the other programmers might aspire to become chief programmers in time. But the goal of most team members should be to develop their talents, rather than climb a career ladder, he says.

Baker acknowledges that the chief programmer team concept has the usual hard time using novice programmers. It depends on experienced programmers for the chief and backup positions;

other team members need a reasonable range of experience. He suggests that novices might be assigned to maintenance programming until they have enough experience to be assigned to a team.

In the *adaptive team*, the situation is almost opposite from the hierarchical organization. As mentioned earlier, the goal is to develop almost a "family" atmosphere (where a woman team member might in fact play the moderating role of "mother"). Such a team does *not* look at a team member as an interchangeable unit. Instead, the team seeks to determine the individual strengths and weaknesses of each member. Work is then assigned to exploit the strengths and to avoid the weaknesses. It is not expected that every team member be good at design, coding, debugging, and documentation, as is generally expected of a "professional" programmer. Further, the leadership is democratic, not autocratic. The team decides on the assignment of tasks. The team monitors progress, by reading each other's code. The team decides what corrective action is needed and when.

The goal is not to change the membership of the team, or at most to change it very slowly. If a member leaves, says Weinberg, the other team members absorb his load. If a new member is added, it can take awhile before he is accepted by the others. The team itself has to decide which tasks to take from each of the others, to assign to the new member.

The certification of programmers would not be necessary, in order to qualify for team membership. Certification will probably attest that the programmer meets certain standards of design, coding, debugging, and documenting. In the changeable structure team, the person is assigned work that he has demonstrated he can accomplish.

From the standpoint of career paths for the programmers, there appears to be none in this type of organization. What an adaptive team offers to its members is the opportunity to work in a friendly yet challenging environment, where the members can develop their talents. A good programmer does not get further and further away from programming, as occurs in a hierarchical structure when he moves up the management ladder. Instead, he stays with programming and gravitates toward what he does best.

Of course, the team as a whole might have a career path, in the sense that it might be given more and more technically challenging assignments.

One of the risky aspects of this type of team is that if one member feels compelled to leave the company, the whole team might move with him. On the other hand, by its very nature, this type of team eliminates some of the things that cause programmers to leave.

Improving programmer performance

The above discussion compared the three types of structure in terms of the working environment that they produce. Now we consider how those environments might translate into productivity. The comparisons must of necessity be fairly subjective. A controlled experiment, to produce objective measures, is probably impractical. Also, as Baker points out, it is very hard to measure the productivity of programmers.

In the *hierarchical* structure, if it is desired to improve programmer productivity, the actions taken are similar to those that have been used to improve productivity in other parts of the organization. The computer is used to perform more of the programming tasks, such as through the use of higher level programming languages. Useful design principles are advocated, such as the use of modular design. The on-line development of programs might be used, as we discussed in our June 1972 issue.

Motivation techniques might be used so as to stimulate programmer performance. A flexible reward system might be instituted, so that each person can pick the rewards that are most meaningful to him.

Management training can be given to new second level managers, so that they are better able to do their jobs and are less likely to "turn off" the people under them.

Quality assurance of the programs that are produced is normally handled by stressing modular design, the testing of modules, and maintaining a library of test cases.

Since programmer turnover is to be expected, documentation of work is very necessary. The buddy system might be used, to assure that someone else knows a programmer's programs, in case he leaves. Since the loss of key managers might be very damaging to a project, more efforts are made to pacify and retain such persons than might be used for the average programmers.

In short, the types of steps taken to improve programmer performance in a hierarchical structure are the ones that most organizations have been using in the past.

In the *fixed structure team*, the picture changes dramatically from what takes place in the hierarchical structure. First of all, the team is headed by a very capable programmer, one of the best that the organization has. He is (hopefully) a highly productive programmer. Moreover, he is relieved of most of the clerical and administrative tasks that tend to reduce the productivity of such people.

Further, the chief programmer (assisted by the backup programmer) designs the system that is being produced and codes the more complex parts of it. In the hierarchical structure, it is typical for one group of people to perform the overall design functions and another group to do the detailed design, coding, and debugging. In the fixed structure team, the team leader does the design and some of the coding. He may use specialists to help in the design, but he is deeply involved in it.

Baker, in References 3 and 4, stresses that one of the reasons that productivity increased with the chief programmer team approach was because of the use of top-down programming. Usually, he says, top-down design is followed by bottom-up programming—that is, starting to code the lowest level modules first. Instead, he recommends, start with the operating system statements that will be required—the job control language and link edit statements. Then write the control code for the top level module, and define the interfaces for the second level modules. While these are being tested (using dummy second level modules), start coding the second level modules. Then test the second level modules (using dummy modules for the third level), while starting to code the third level modules. In general, the interfaces between the modules are defined and coded before any use is made of those interfaces. In general, too, the chief programmer and the backup programmer handle the control coding and the interfaces. The functional (applications) code is assigned to the other programmers.

So the increased productivity of the fixed structure team may result from the higher quality of code and the consequent reduced need to rework it. One group of people does both the design and the construction. The group is headed by a very competent programmer, who is backed up by another competent programmer. These two people write most of the control code and the interfaces, where many of the debugging problems originate. The whole group practices egoless programming in the sense that they read each other's code. (It probably is not as egoless as the adaptive team, because the chief programmer is, in fact, the boss.) The work is performed by the most capable people. The team is small so that communications are good. There is a reduced chance of misunderstandings.

One cause of loss of performance in a hierarchical structure is the waste that can occur. With top-down design but bottom-up construction, individual programming tasks may not be seen in context. Any consequent misunderstandings can cause erroneous lines of code to be written, which later have to be changed. Conventional programming practices tend to leave bugs in the code, which must be caught during debugging—which in turn means more debugging time. Administrative and clerical activities take up a good percentage of the time of the skilled programmers, meaning that most of the code is written by less skilled people. These are wastes that the fixed structure team tries to eliminate.

Many of the same things that contribute to performance also contribute to quality assurance. We discussed earlier in this report the very low error rate that occurred in the *New York Times* information retrieval system.

The team is also structured well for adapting to crises. If the chief programmer should leave or become sick, his position is filled by the backup programmer. No big problem occurs when another team member leaves or must be replaced, since the change of team membership is a normal event. If system specifications are changed, the chief programmer is in a position to decide on the best course of corrective action.

The *adaptive team* also aims at increasing programmer productivity by avoiding the waste that normally occurs in hierarchical structures. In many ways, it performs like the fixed structure team. But there are some important points of difference.

For one thing, *the team stays together.* The people learn how to work together. The team builds productivity by concentrating on the strengths of the individual members. Weinberg

restates something that has been well recognized all along—namely, that the individual differences between programmers can be very large. Ogdin (Reference 7) quotes one study that showed the following worst-to-best ratios, for twelve experienced programmers that worked on the same problem: debug time used, 26:1; computer time used, 11:1; coding time used, 25:1; code size, 5:1; and running time, 13:1. One person might be very good at design but very poor at debugging, says Weinberg. Another might be superior at locating bugs, but poor at actually correcting them. With these large individual differences that do exist, the team will soon learn just where a person's strengths lie. The team will then structure the project and assign tasks so as to exploit those strengths.

Egoless programming also greatly reduces debugging time, says Weinberg. Since the team members critically read each other's code, most of the bugs are caught before the code even reaches the machine.

In Reference 6, Weinberg carries his point even further. He discusses the results of a test that was run with five programming teams. He notes that variations in performance between teams can be almost as great as the variations between individuals. One of the reasons for these variations, he believes, is the possible misunderstanding of objectives. The five teams worked on the "same" problem. But each team was given a different objective. One was told to minimize the use of core, another to minimize running time, another to minimize the number of source statements, another to maximize clarity of the source program, and the last one to maximize the clarity of the output. As might be expected, the objectives were often in conflict. Maximum clarity conflicted with minimum core, for instance. Each team did well on its primary objective, but was outranked on all other objectives. That is, each team led the others only in terms of its primary objective. So if a programming team misunderstands the desired objectives, the program may in fact turn out to be poor according to the desired objectives.

Weinberg discusses one other test that is pertinent, where two programming groups worked on the "same" program. One group was asked to write the program as quickly as possible, while the other one was asked to produce as efficient code as possible. The "prompt" group required less than one-half the number of debugging runs that the "efficient" group needed, but their execution time was six times that of the "efficient" group. However, it was found that the "prompt" program was more flexible and tunable than the "efficient" program, and could in fact be changed so as to pick up most of the difference in execution time. So, says Weinberg, the best course of action may be to ask programmers to write their programs promptly, and then allow time for tuning them so as to improve performance.

It is up to the team, then, to make sure that it clearly understands the objectives that are desired. The job is divided and assigned to the team members, according to their individual strengths, as much as possible. Strive for getting the program written promptly, says Weinberg; the prompt programming tends to produce "sloppy" code. Then when the program is working, modify it to meet the desired objectives—core usage, run time, or whatever.

Quality assurance is achieved by using the strengths of the individual team members, and also by egoless programming. And the *adaptive team* adjusts well to crises and frustrations, says Weinberg, because the team itself decides what the best course of action is.

It is apparent from the above discussions, we think, that there are sizable differences of opinion on how programmer performance can best be improved. But the differences of opinion do not end there. We next will consider the question of "software engineering."

The engineering of software

Two questions stand out. *Can* software be engineered? (Is programming an art, or can it be accomplished under an engineering type of discipline?) *Should* software be engineered? (Even if an engineering type of discipline is feasible, is it desirable?)

We were privileged to attend a workshop on the subject of software engineering held in the Spring of 1973. While it was a small group, it included leaders in the software field from the U.S. and Europe. The group represented the spectrum from advanced research in programming methodology (such as proving the correctness of programs) to the management of a large programming staff in a major corporation.

In the discussions, there really was no debate on

the question "can software be engineered?" The question was raised and the consensus was that it can be engineered. There was some debate on the use of the term "engineering," but there seemed to be no real question that an engineering type of discipline could be used.

Nor was there any debate about whether such a discipline should be used. The consensus of the group was that the time is right for encouraging a discipline for the development of software. There was much discussion, of course, on what aspects deserved primary attention and what the scope of the subject area should be.

The consensus of a small group of leaders in the software field does not prove that software can or should be engineered. But as far as we are concerned, it does represent professional opinion on these two questions. We have neither read nor heard strong counter arguments that would seem to outweigh the consensus of this group.

The members of the workshop singled out two major aspects of the subject:

- The technology of software engineering
- The management of software projects

The technology of software engineering

The philosophy of top-down design and construction was recommended by workshop members for the development of software. This recommendation applied whether system software or application software was being developed.

The top-down approach consists of three major stages: architecture, detailed design, and construction. In each stage, consideration must include the overall system, the programs, and the data. Also, in each stage consideration must be given to quality assurance, which can include acceptability testing, proof of program correctness, or some combination of the two. (We will have more to say about proving correctness of programs shortly.)

In the *architecture stage*, the general structuring of the system, the programs, and the data are performed. This is when the overall plan of attack is determined. This is when the decision is made, for instance, on whether the program designs will be modular, structured, or monolithic.

In the next two reports, we will discuss modular versus structured programming, so we will mention only some highlights here. In a sense, structured programming is a particular type of modular programming. With structured programming, the modules usually are small—say, in the order of fifty lines of source code. The flow of control is rigorous. Expressed in terms of family relationships, control can flow from a grandfather module to father module to son module, and then back up this same path. Control does not pass directly between cousin modules; rather, it must go up the hierarchy until the common parent (the grandfather module) is reached and then down the other branch; see Reference 16. When the term "modular programming" is used, the modules can be almost any size, and the flow of control is not as rigorously defined.

Also, during the architectural stage, the criteria by which the final system will be judged and evaluated should be determined. Some, if not all, of these criteria will have been set when system requirements and system specifications were developed, just prior to the architectural stage. But it is important in the architectural stage to insure that there is agreement between the users and the designers as to what is wanted.

The members of the workshop identified the following types of criteria, for judging the final system:

CRITERIA FOR JUDGING SYSTEMS

1. Basic design criteria—general, simple, modular, using proven techniques.
2. Development and operating criteria—economic use of resources, effective, useful (meets user needs).
3. Criteria for change—able to understand system and programs; able to maintain, modify, extend, tune, and transfer to other equipment.
4. Schedule criteria—will system be available in time?

In addition, as we discussed earlier, Weinberg (References 5 and 6) has stressed the importance of telling the developers just what is wanted. For instance, the programmers should be told at the outset what resource usage is to be minimized, for the economic use of resources:

ECONOMIC USE OF RESOURCES

What is to be minimized?

1) Computer resources used (minimize core, running time, or other)

2) Development programmer time (develop as promptly as possible)
3) Maintenance programmer time (provide clarity of source programs)
4) User time (provide clarity of output)

If the user's and/or management's desires are not clearly spelled out on these items, says Weinberg, the programmers may pick the "wrong" goals.

Also during the architectural stage, the decisions must be made on how quality assurance will be provided. There are two main approaches to quality assurance, one widely used and the other under development. The widely used approach, of course, is program and system testing. The approach being developed is "proving the correctness" of programs.

With all of the program testing that is being done, it would seem that it should be a well developed discipline by now. Not so, we are told. Much development work is going on, both in research environments and in commercial enterprises, for developing better methods. We will discuss some recent work on program testing next month. Interested readers might read Reference 8, particularly the paper by Krause and Smith on techniques for predicting the optimum set of test cases.

But, say some researchers, testing will never really solve the problem. These people point out that testing can indicate the presence of bugs, by detecting them. But testing cannot indicate the absence of bugs—simply because it is not possible to develop the test cases for testing all possible conditions. Instead, what is needed is a means of proving the correctness of a program, in much the same way that a mathematical theorem is proved.

Linden (Reference 4) gives a good summary of the status of program correctness proving techniques, as of the end of 1972. In his presentation, he mentioned that correctness proving is still limited to smaller programs, in the order of 200 lines. Elspas et al (Reference 9) present an assessment of the techniques for proving program correctness.

Proof of correctness techniques cannot yet really be considered to be a part of software engineering, except in special cases. The program sizes to which they can be applied are too small for general use. But it might be possible to prove the correctness of the small "kernel" of a security system, for instance, and then use testing methods for the remainder of the system.

Moving on to the *detailed design* stage, we know of no definitive description of the process. Baker (Reference 3) describes how the chief programmer team approached the *New York Times* information retrieval system. The chief programmer started with the instructions to the operating system, after which he tackled the control code of the highest level (control) module. At the same time, he was defining the appropriate interfaces. It would seem that the chief programmer was doing program and data *design and construction* simultaneously, but at the highest module level first. Then he worked down through the other module levels, concentrating on the flow of control and data interfaces. The design and coding of the functional processing was assigned to other programmers. But in all cases, a top-to-bottom sequence was followed.

Weinberg (Reference 5) makes the point very strongly that the development process is iterative, and that it is impossible to really separate it into well defined stages. And what he portrays is certainly a common occurrence. It is not unusual, as far as we can tell, to start with a top-down design, and then begin coding with the lowest level modules—on the basis that they will influence the design.

But IBM's experience with the chief programmer team does raise the question of whether bottom-up coding is necessary or desirable. We will have more to say on this month after next.

The "engineering discipline" of the detailed design and construction stages thus has not been well established. It is still an area of debate.

There are a number of tools and techniques available to the programmer that can be used in the construction stage. The tools include programming languages, compilers, operating systems, data base management systems, testing aids, and such. The techniques include quantitative methods (modelling, simulation, etc.), logic definition (flowcharts, decision tables), data handling techniques (sorting, searching, etc.), and so on. Standards and conventions also are a part of what the programmer should use during program construction.

The members of the workshop on software engineering felt that the technology will now sup-

port a disciplined approach to software development. The above discussion is an overview of how they saw the process structured. But it seems to us that the concept of top-down design *and construction* needs more attention. It appears to be a very important element of the discipline, and yet the methodology seems fuzzy.

The management of software projects

While there was reasonable consensus among the software engineering workshop members as to the status of the technology, there were some substantial differences of opinion on the management of software projects. The research-oriented participants seemed to look at the management related matters as necessary evils, and wanted to get back to a discussion of the technology. The management-oriented participants felt just the opposite. One participant, in fact, from a management position in an academic and research environment, felt that the only real stumbling block for software engineering was the lack of an engineering management discipline in our field. He felt that this discipline must be defined, taught, and enforced—and only then does software engineering have a chance of success.

What is involved in applying engineering management principles to software development? We see the following elements. *Clear statement of task to be done.* The project needs clearly specified goals. *Clearly defined project phases.* The project must go through distinct phases, with clearly identified milestones. The overall system must be divided into parts, and the parts in turn divided into sub-parts. System specifications must be developed, along with quantitative criteria for evaluating the system. *Use of standards and conventions.* The use of standards and conventions will reduce the development effort, make the system easier to change, and protect the organization against programmer turnover. *Control of changes.* As soon as a design has been "completed" so that it is acceptable to the user, then that design should be frozen. Should changes be desired or needed subsequently, they have to go through a formal change control process before being implemented. *Project milestones and checkpoints.* There should not be just an end target date, but a good number of intermediate target dates. *Creeping commitment.* Tied in with the checkpoints are a number of management reviews, especially in the early phases of the project. If project performance is satisfactory, permission is granted to proceed to the next checkpoint, but no work beyond that checkpoint is authorized. If performance is not satisfactory, if some of the work is still not complete, or if there have been any significant changes in estimated costs or benefits, then management must decide what corrective action to take. *No-nonsense management.* Larson (Reference 10) states this case well. In some organizations, he observes, there is altogether too much "fun and games" with the computer—using it for games, puzzles, analyzing the stock market, and so on. These activities have to be drastically curtailed. Also, Larson emphasizes the importance of a thorough statement of the problem at the outset, the establishment of measures of performance, and cost and schedule goals.

This is the engineering management approach to projects with which we are familiar; back in our engineering days, this is the type of environment we worked in. As far as we can tell, it is still the environment. It has been and is being used for advanced technology hardware projects. It has been used successfully, also, for managing a large software project in the U.S. APOLLO space program. We have discussed various aspects of it in our June 1968, October 1970, and May 1973 issues, for instance.

But not everyone agrees with this type of management for software projects, by any means. Some say that the development of software is "different" from that of hardware—more complex, perhaps, or more unknowns, or the fact that a program is intangible. And whether software is different or not, others say that they will do a much better job if they can work in the humanistic environment proposed by Weinberg. Give a project to a team, says Weinberg, with an end target date. Then leave the team alone. The team will do the design, construction, and testing of the system, using egoless programming. There is no standard project structure, no distinct project phases, no (or perhaps a very few) project milestones, and no lockstep discipline on project progress.

The question might then be raised: can engineered software be created by an adaptive team (Weinberg's approach)? It seems to us that the answer must be: Yes, it *can* be. Perhaps a more ap-

propriate question would be: what is the probability that an adaptive team *will* produce engineered software? The answer hinges on the degree of both knowledge and self-discipline that the team has. Assuming that at least some of the team members are experienced programmers who know the necessary tools and techniques mentioned earlier, it will then depend upon the team's discipline. To what extent will the team follow the installation's standards and conventions? How much importance will the team attach to each of the types of criteria listed earlier? How likely is it that the team members will get sidetracked on complex design points that have only minor influence on the total system operation? How important does the team regard the overall schedule and the milestones?

Such questions cannot be answered in general, of course. We have heard proponents of the strict discipline approach argue that, if left to themselves, programmers are great wasters of time. We imagine that the proponents of the humanistic approach would say that if the project goals are reasonable and meaningful, an adaptive team will have an effective self-discipline. We suspect that it is not just a matter of the teams themselves but perhaps more importantly, the enlightened management environment in which the teams work. We would like to see more experience in this area.

At the same time, it appears to us that the fixed-structure team *can* provide the needed discipline for software engineering. It seems to combine the benefits of a hierarchical leadership with the stimulating working environment of a close-knit team.

In fact, from all reports, one would have to say that the *New York Times* information retrieval system is an impressive instance of engineered software.

The certification of programmers

The basic idea of certifying programmers is that there is some defined body of knowledge that a programmer should have, and that a person cannot be considered a "qualified" programmer unless he has that knowledge. The most effective way to find out if he has that knowledge is to test him on it.

We might digress briefly to differentiate between certification and licensing. As discussed in Reference 11, certification is the affirmation that an individual has met certain qualifications. Uncertified persons can perform the same tasks if they can find patrons. On the other hand, licensing is the administrative lifting of a legislative prohibition. An unlicensed person cannot legally perform the tasks for which a license is issued.

If certification of programmers is to occur (and we will discuss *why* it is being proposed shortly), the following steps are needed. First, the job of the programmer must be defined—a universal job description for a programmer is needed. Then the minimum standards of job knowledge and skills must be defined, for that job description. Then a testing program must be developed, perhaps building upon any testing experience already available in the field. Finally, a public information program must be instituted, to make the affected publics aware of what is available and what its benefits are.

There are two major programs underway in the U.S. leading toward the certification of programmers; we have no recent information on the status of similar programs in other countries. These two programs are:

- AFIPS Professionalism Project
- Institute for Certification of Computer Professionals

AFIPS Professionalism project

The professionalism project of the American Federation of Information Processing Societies Inc. began with a roundtable meeting in January 1970, chaired by The Honorable Willard Wirtz, former U.S. Secretary of Labor. The roundtable considered accreditation, certification, and ethics. A report of that meeting has been published by AFIPS, Reference 11.

As a first step toward professionalism, it was decided to develop universal job descriptions for programmers. It was recognized that the certification of other job categories might also be desired in the future.

The programmer job description sub-project was initiated with a one-day meeting in October 1970, chaired by the committee chairman, Donn Parker. Based on the recommendations of that meeting, a small panel of programming experts was selected. The Delphi method was used to get a convergence of opinion in the group. The subject of study was: what are the tasks and skills that

make up the job of programmer? A total of 75 tasks were identified for business programmers, 84 for scientific/engineering programmers, and 79 for system programmers. A total of 163 skills and techniques was identified. The results of the study were refined at a one-day meeting of project members at the 1972 SJCC.

Following that meeting, a national survey of 684 programmers was conducted by the project's consultant, Dr. Raymond Berger. A random stratified sample of organizations was approached, from which 248 business programmers agreed to participate, as well as 250 scientific/engineering programmers, and 186 system programmers.

Both the Delphi panel and the 684 programmers rated the tasks, skills, and techniques as to importance on their jobs. Or, more precisely, the Delphi panel rated these in terms of what the programming job ought to be, and the 684 programmers rated them in terms of their own jobs. Scatter diagrams are presented in the report, Reference 12, of these two sets of ratings. It is interesting to note that while there was general agreement between the two groups, there were still substantial differences of opinion.

The AFIPS Professionalism project now feels that it has a statistically valid set of job definitions for business programmers, scientific/engineering programmers, and system programmers. They are requesting permission to repeat the project, to define system analyst job descriptions. Then they would like to make the results of their work (on both programmers and analysts) available for use in general and for the ICCP in particular. The ICCP, which we discuss next, is concerned with the development of meaningful certification programs.

The ICCP

The Institute for Certification of Computer Professionals (originally called the Computer Foundation) was formed in 1973 as a not-for-profit organization. There were eight charter member societies, including the Data Processing Management Association, the Association for Computing Machinery, and the IEEE Computer Society. See Reference 13 for more background details.

One question that has come up is: Why was the ICCP formed? Why could not this have been done under AFIPS? The reason is that one of the prime movers, DPMA, was not then a member of AFIPS. In fact, of the eight chartering societies of ICCP, only ACM and IEEE CS were AFIPS members. The time may come when ICCP is a part of AFIPS; the two organizations are on good terms.

DPMA is turning over to the ICCP the ownership of its two examinations—the Certificate in Data Processing, and the Registered Business Programmers examination. The ICCP is reviewing both of these examinations in arriving at a decision on its future direction. Present plans call for the ICCP continuing to offer these two examinations while improving them in an evolutionary manner. In addition, it is recognized that certification might be needed in a number of other computer-related areas.

The debates on certification

Why should programmers be certified? Proponents argue that it is now too easy for people to wander in and out of programming—and that the quality of programming is suffering thereby. More and more computer-based systems are directly affecting the public, or the economic health of organizations, or individual privacy. We can best upgrade such systems by certifying the programmers who work on them, say the proponents of certification.

(A counter proposal is to certify the *systems*, rather than the people who develop them. We have mentioned in past issues the continuing AFIPS' efforts on system improvement and system certification.)

Weinberg, in Reference 5, argues against the need for certification. The job of programming has so many facets which are specific to the project that the job cannot be universally defined, he says. So it will not be possible to test for the knowledge and skills that will be needed on a specific project. Also, personality factors might well be more important than native intelligence factors, within reasonable limits. By using egoless programming, each individual's strengths are used.

Another argument against certification is that what the examination tests for is not relevant to the job environment. The examination tests for what an individual can do by himself in a short period of time. But his normal work environment involves team work over an extended period of time.

Paul Armer, in Reference 14, does not argue against certification—but he does argue against the ICCP and DPMA's CDP. Instead of creating another organization, there should be a consolidation. More importantly, he feels that the CDP is not a meaningful test. The ICCP should suspend testing until it develops a meaningful test, Armer feels, but he recognizes that the ICCP is counting on the income from this examination for the next several years.

We have discussed the CDP in our July 1965 and December 1968 issues, and gave examples of the types of questions used. The CDP *does* discriminate; less than one-half of the people who have taken the examination have passed it. But the CDP is not tied to the knowledge and skills required for any particular job. This is the main argument, we believe, against the meaningfulness of the examination; see Reinstedt and Berger in Reference 15.

Issues in programming management

Here it is 1974, and the controversies about how best to manage the programming function are as intense as were the debates on this subject ten and fifteen years ago.

We believe that a good number of data processing managers have recently been facing up to the question of team programming, versus the more conventional hierarchy of individuals. A number of organizations that we have interviewed recently on other subjects have mentioned that they have converted to a form of team programming.

But *which* form of team programming should be considered? In actual practice, the fixed structure team and the adaptive team concepts will probably turn out to be not too different from each other. A team programming concept somewhere between these two concepts may well develop. We suspect that the originators of these two types of team programming are more interested in practical results than in users absolutely adhering to the dogma of the concepts.

Baker made an interesting point, in a letter to us. Trying to install the engineered software approach can be a good way to move to team programming, he said. If an organization sets out to do a good job of top-down program development, it will find that it comes close to a team organization—although not all of the team structure and working relationships may be there. The organization might try out a few projects this way, find out who the best leaders are in that environment, and then set up one or two teams. As we indicated earlier, the chief programmer team concept that IBM used on the *New York Times* information retrieval system produced an outstanding example of engineered software.

The state of the art today will support both engineered software and team programming. The certification of programmers is still some years away, because of the need to validate and gain acceptance of a certificate program.

It seems to us that all three of these issues represent interesting developments in the management of programming. They are issues that deserve management's consideration.

REFERENCES

1. *Computing Manpower*, Infotech State of the Art Report, Infotech Information Ltd. (Nicholson House, High Street, Maidenhead, Berkshire, England), 1973, $95 (£40).
2. Metzger, P. W., *Managing a Programming Project*, Prentice-Hall, Inc. (Englewood Cliffs, New Jersey 07632), 1973.
3. Baker, F. T., "Chief programmer team management of production programming," *IBM Systems Journal*, IBM Corporation (Armonk, New York 10504), Vol. 11, No. 1, 1972; price $1.50.
4. *Proceedings of 1972 Fall Joint Computer Conference*, AFIPS Press (210 Summit Avenue, Montvale, New Jersey 07645), 1972; price $40, microfiche $10. See particularly the sessions on software engineering, p. 173-212 and 311-344.
5. Weinberg, Gerald M., *The Psychology of Computer Programming*, Van Nostrand Reinhold Co. (450 West 33rd Street, New York, N.Y. 10001), 1971, price $9.50.
6. Weinberg, Gerald M., "The psychology of improved programming performance," *Datamation* (1801 S. La Cienega Blvd., Los Angeles, Calif. 90035), November 1972, p. 82-85; microfilm copies available from University Microfilms, 300 N. Zeeb Road, Ann Arbor, Mich. 48106.
7. Ogdin, J. L., "The mongolian hordes versus superprogrammer," *Infosystems* (Hitchcock Building, Wheaton, Illinois 60187), December 1972, p. 20-23; price $2.
8. *Record of 1973 IEEE Symposium on Computer Software Reliability*, IEEE Computer Society (9017 Reseda Blvd., Northridge, Calif. 91324), 1973, price $12.
9. Elspas, B., K. N. Levitt, R. J. Waldinger, and A. Waksman, "An assessment of techniques for proving program correctness," *Computing Surveys*, (ACM, 1133 Avenue of Americas, New York, N.Y. 10036), June 1972, p. 97-147; price $8.
10. Larson, H. T., *The Larbridge Letter*, a monthly service for top executives, (18521 Embury Drive, Santa Ana, Calif. 92705), price $300 per year.
11. *Professionalism in the Computer Field*, AFIPS Press (address above), 1970, price $3.
12. Berger, R. M. and D. B. Parker, *Computer Programmer Job Analysis*, report of the AFIPS Professionalism Committee; for information, write Executive Director, AFIPS (address above).
13. Harris, F. H. and J. K. Swearingen, "Report on the status of the Institute for Certification of Computer Professionals," *Data Management* (505 Busse Highway, Park Ridge, Illinois 60068), October 1973, p. 18-21, 32-34; price $1.50.
14. Armer, Paul, "Suspense won't kill us," *Datamation* (address above), June 1973, p. 53.
15. *Datamation* (address above), November 1973:
    a) Reinstedt, R. N. and R. M. Berger, "Certification: A suggested approach to acceptance"
    b) Guerrieri, J. A., Jr., "Certification—Evolution, not revolution"
16. *Datamation* (address above), December 1973; five articles on structured programming, under the general heading of "revolution in programming." One of the papers, by Baker and Mills, discusses IBM's chief programmer team concept.

Next month's report, the second in a series of three on issues in programming, will discuss modular programming. In the United States at least, modular programming has received much more support in theory than in practice. But in the United Kingdom, the approach has been reasonably well accepted, and a number of software packages to support modular programming have been marketed successfully. So we decided to visit several organizations in England to find out how they are using modular programming and what results they have been obtaining. Next month we will tell about these experiences. And in the June issue, we will conclude this series with a discussion of user experiences with structured programming.