EDP ANALYZER

FEBRUARY, 1978
VOL. 16, NO. 2

PROGRESS IN SOFTWARE ENGINEERING: PART 1

Over the years, programmers have tended to treat programming as an "art," involving creative, innovative talents that are not properly the subject of a discipline. At the same time, many people who are familiar with computer hardware engineering claim that computer programming inherently involves no more logical complexity than hardware engineering—which is subject to a discipline. In fact, there is some good evidence to support this view. Gradually the concept of software engineering has evolved, to the point where today it claims a substantial and growing body of knowledge. As the benefits become recognized, software engineering will permeate the whole development and modification process. Here is the first of two reports on where the field stands today.

Software engineering seeks to impose a methodology and a discipline on (1) the development and modification of software, and (2) the *management* of software development and modification.

Several points should be noted in this definition. Most authors on the subject limit the subject to a methodology for the development of software. We are suggesting here that the continued modification of that software also should be a part of the subject area. Similarly, many authors limit the subject to the activities of designing and constructing software. We are suggesting that the management of those activities is also a part of the subject area.

Two questions of course stand out:

- Can software in fact be engineered?
- Should software be engineered?

In addressing these questions, let us consider the viewpoints of two leading figures in the field—Donald Knuth and Edsger Dijkstra. Both have been the recipients of the prestigious ACM Turing Award.

Donald Knuth (Reference 1) argues for the *art* of computer programming. He seeks beauty and elegance in programs. He urges trying to achieve beautiful solutions to problems.

What is "beauty" in a computer program? We believe that Knuth means gaining an insight into a complex problem, grasping all of its essential elements and their relationships, and then coming up with a clean, crisp, "beautiful" solution.

A well-known example might help to illustrate what is meant. It concerns the case of the mother with two small boys and one piece of cake to be divided between them. How was she able to divide that piece of cake, she was asked, so that the boys would not argue over who would get the bigger piece. It was easy, she replied. She had one of the boys divide the cake and the other boy get the first choice. This is indeed a beautiful solution, in our opinion.

But Knuth also seeks "goodness" of programs, in addition to beauty. He includes "works correctly" and "not hard to change" among the characteristics of goodness. He also asks that programs interact gracefully with users, give meaningful error messages, and use flexible, non-error-prone input formats.

It seems to us that Knuth is saying programming is an "art" to the extent that sufficiently good solutions are not yet a part of common practices. But these "goodness" characteristics might very well become a part of common practices and a part of a programming discipline. Of course, one might argue that even better solutions might be sought and that programming will thus always remain an art. Programming managers faced with deadlines may be reluctant to support such searches *ad infinitum*.

Edsger Dijkstra (Reference 2) advocates a discipline for mastering complexity in large computer programs. He proposes an approach for handling complexity via the concurrent development of the program and its proof of correctness. His approach includes structured programming by stepwise refinement, as well as proving the correctness of the program as it is developed. We will have more to say about Dijkstra's views later in this report. Suffice it to say here that he advocates a discipline for programming and one that is in harmony with software engineering.

It seems to us that there is, and there always will be, an "art" element to programming. This element seeks beautiful solutions to problems. But once these solutions have been developed, they might well become a part of good programming practices. The discipline of "good programming practices" is necessary for handling problem complexity and assuring program correctness.

It would appear, then, that a discipline of good programming practices (in other words, software engineering) is needed and is accomplishable. Software engineering seeks the basic principles that are the fundamentals of good programming practice.

The scope of software engineering

We participated in a planning session on a software engineering handbook several years ago. The planning session was sponsored by the U.S. National Bureau of Standards, the National Science Foundation, and the Association for Computing Machinery. The results are reported in Reference 3. NBS has since initiated a project to develop such a handbook, in part based upon the recommendations of the planning session.

The planning session included such leading figures in programming methodology as Edsger Dijkstra, Barbara Liskov, and Robert Floyd, and practitioners and managers such as Joel Aron, Anthony D'Anna, Dennis Fife, Aaron Finerman, John Gosden, Harry Larson, Charles Lecht, and Daniel Teichroew. The group thus included representatives of both research and production environments, and both technical and managerial functions.

This planning session defined the scope of software engineering to include both the technical and the managerial aspects. We feel that the definition of the subject area made by this group has stood up well during the interim. Here are the highlights of that definition.

Software engineering techniques. The scope of software engineering technology should apply to systems, programs, and data. The techniques should apply to the *architecture* of these elements, to their *engineering*, to their *construction*, and to their *acceptability testing*. Special considerations should be given to large systems.

Software engineering management techniques. Every bit as important as the software engineering techniques are the management techniques under which development takes place. For instance, in its assignment of a tight time schedule to a project, management may be in fact making the basic architectural decisions for the system. The planning session identified some candidate first principles for managing software projects, as well as the need for a standard project discipline.

We see some possible modifications to the scope of software engineering, as proposed by this planning session. The software engineering techniques should include methodologies for *problem definition* and *requirements* analysis. Also, the methodologies for construction should be enlarged to include techniques for the *modification* of systems, programs, and data definitions. Further, techniques for the *evaluation* of systems, programs, and data might be included.

In the management of system development and modification, managers should have an understanding of several important behavior patterns. These patterns include staff behavior, project behavior, and program evolution. We will discuss these behavior patterns next month.

We have attended conferences, conference sessions, and workshops on the subject of software

engineering. In addition, we have reviewed a substantial amount of the technical literature. It is apparent to us that there is a tremendous amount of work going on in this subject area. Further, it is evident that the methodologies are gradually becoming a part of the "good practices" of the computer field. We discussed some of these methodologies in our November and December 1977 and January 1978 reports. But we also began to see the need for one or two "overview" reports that would show the breadth of what is emerging. Hence this report and the one next month.

Let us look first at what is emerging in the area of technical methodologies.

SOFTWARE ENGINEERING METHODS

There is really too much work going on in the area of software engineering techniques for us to be able to review it in one or two issues. But quite a bit of that work is in the early stages of research or development. While much of that work is interesting, it is perhaps too early for us to discuss it; we prefer to wait until it is ready for practical application. So we have selected for discussion some of the work that most appeals to us and that seems to have immediate application. However, the references cited at the end of these two reports point to literature that covers much of the whole subject area.

Problem definition area

The subject area of problem definition and requirements engineering is really just beginning to emerge. We gave an overview of this subject in our July 1977 report, which we will summarize shortly.

Ross and Schoman (in Reference 8e) point out that the total system development process consists of a series of steps leading to the solution of a problem. In these steps, only once is the problem itself stated and the solution justified—and that is in the requirements definition phase. Requirements definition deals with *why* the system is needed, *what* features are needed, and *how* the system is to be constructed, they say.

They advocate the method of successive refinement and decomposition. This method considers everything relevant at a given point and nothing more. The not-yet-relevant decisions are postponed until they are relevant.

The authors have been parties to the development of SADT, which stands for Structured Analysis and Design Technique. SADT uses a relatively simple graphic technique with boxes and arrows. The boxes represent the system components and the arrows are the interfaces. The top level diagram shows the main components of the system. Successive diagrams decompose these components. While each diagram is relatively simple to draw, the technique of problem analysis requires a significant effort to learn. It generally takes from one to two weeks of classroom training, plus on-the-job training, before an analyst can begin to do useful work with the methodology, we gather. Further, proficiency comes only with experience. It is not a case of just reading the manual and then beginning to use the methodology. For more information on SADT, see Reference 5.

The ISDOS project at the University of Michigan, which we discussed in our November 1971 report, has developed a problem statement language (PSL) and a problem statement analyzer (PSA) program, for tackling the area of problem definition. PSL is in use by a good number of large organizations. For more information, see Reference 6.

In our July 1977 report, we discussed a suggested no-frills program that an organization can use to get the requirements right for new application systems, based on work that had been done in this subject area. Here are the main points of that program.

Recognize the types of errors. Begin to develop a list of the errors of omission and commission found in requirements statements. These errors come to light during the subsequent phases of all projects. Classify the errors by type and then start making management aware of the list.

Get user involvement. Hold two-day "requirements sessions" to set the requirements for each new application system. Make sure that key managers participate. The error list that has been developed not only will help to get this participation but also will act as a checklist for discussion. Users should also participate in the appropriate inspection sessions.

Select an approach for handling complexity. Functional decomposition (also called successive refinement, top-down expansion, or levels of abstraction approach) currently is the favored ap-

proach for handling complexity. Information flow analysis is another approach; it charts the flow of information from origin to ultimate use. There are also some software tools that help to make changes more easily, as complexity causes changes in design.

Use an inspection process. A key element for getting the requirements right is the inspection process. It involves a two to three hour review session at each inspection point in a project. Inspections should be performed on requirements, specifications, system designs, program designs, coding, and testing plans. The informal structured walk-through, discussed in our November 1977 report, is currently popular. Fagan (Reference 7a) advocates a more formal approach to inspections.

Define expected performance. Errors of omission and commission will come to light if performance validation tests are developed for all requirements statements. Attempting to develop such tests will bring the requirements into clearer focus.

The upshot of the work being done in this area of problem definition is that it does involve a discipline and further that this discipline is imposed throughout the development process. This work recognizes that errors of omission and commission permeate the requirements statements for almost all computer-based systems. It takes a continuing effort to flush them out.

Architecture, engineering, design

Mills (in Reference 8b) makes the point that entirely too much of the programming effort today is spent on corrective and adaptive maintenance. Corrective maintenance seeks to remove the errors that should not have been built into the systems, and adaptive maintenance seeks to enhance the systems to perform additional functions. While not all of the adaptive maintenance can be avoided, much of it can, Mills implies, if the job were done properly from the outset. The key to getting programs right, says Mills, is clean, compelling, rigorous design. He then goes on to propose a general approach based on functional decomposition.

If design plays this key a role in getting systems and programs right, then one would expect that a good amount of software engineering literature would address this question. Right? Well, unfortunately, this is not so. The whole design area, including overall design (architecture) and the more detailed design aimed at goodness of operation (engineering), has received very spotty treatment.

Architecture

As just mentioned, the "architecture" of a system deals with its overall design and general structure. Ideally, the system architects seek "beautiful" solutions. In practice, computer-based systems usually involve so much complexity that the architects simply seek solutions that will work and that have some goodness characteristics.

Dijkstra (Reference 2) discusses mental tools that the system architect can use for handling complexity and still come up with a good overall design. We discussed Dijkstra's "levels of abstraction" approach in our June 1974 report. His newer work builds on this earlier work and adds the proof of correctness concepts. With the levels of abstraction approach (or functional decomposition), one starts with the overall system and then tries to identify all of the major components of that system. No attempt is made to analyze each of the components until all of the top-level parts have been identified. Then the architect steps down one level and goes through the same sort of analysis for each of these top-level parts. By handling complexity one level at a time, the architect not only has a better chance of handling it "right" but also has more of a chance of developing a "beautiful" solution

Most of the discussion of this type of approach has dealt with the design of programs rather than with the design of complete applications or software systems. While the principles would seem to be appropriate, we think that much more investigation is needed to see how these principles can be applied over a wide range of application and software systems.

Engineering

Engineering involves the "good practices" part of solutions. When the engineering is based on a theoretical foundation, such as a mathematical foundation, it becomes inter-mixed with the architectural process. In general, software engineering does not yet have a widely accepted theoretical foundation. However, next month we will discuss the concepts developed by Kenneth

Kolence of what he calls "software physics." He sees this work providing a theoretical foundation for software engineering.

The American Federation of Information Processing Societies (AFIPS) has sponsored some work dealing with goodness characteristics of information systems. Their Best Practices Manual on *Security* (Reference 9) provides design guidelines plus an extensive checklist of some 900 questions relating to security that the system designer should consider. A second Best Practices Manual, on designing for system integrity, is under development.

Design

The term "design" is a general term used by many (including ourselves) to cover both the architectural and the engineering functions. Every system, program, and data structure has a "design," even if that design has not been explicitly planned. The design may be rambling, with a horribly involuted control structure, if the builders just start building before thinking through the design.

Even in this more catch-all subject area of design, the literature and the conference sessions have been sparse. Very little has been said or written on system and sub-system design methodology.

Freeman and Wasserman (Reference 22) provide reprints of 18 key papers on software design plus some 100 pages of original material that explains software design concepts. This paperback book gives an overview of the system and program design process. The design techniques and tools that are described apply mainly at the program design level. These include structured programming, decomposing systems into modules, and the use of program design languages. More books of this nature clearly are needed.

Peters and Tripp (in Reference 23) compare a number of the leading software design methodologies. These include structured design (as proposed by Constantine, Yourdon, and Myers), the Michael Jackson method, Warnier's Logical Construction of Programs, the META stepwise refinement method, and the high order software method.

Ben Schneiderman (Reference 10c) catalogs a number of approaches to program and data structure design and gives a 38-item bibliography for further details. The program design methods include: single module programs, linear structures, tree structures, level structures, and network structures for programs. The data structures that he catalogs include: single node, linear, tree, and network structures.

L. C. Carpenter and L. L. Tripp (Reference 11) describe a computer program (DECA) that is used in conjunction with a top-down dominated design methodology. This program organizes, validates, and produces documentation depicting the design of a software system. It significantly enhances the quality of the software design, say the authors.

We are not saying that these are the only papers on design that we have come across. But they were the papers that most appealed to us. Further, the selection was *very* limited.

Construction and modification

The situation in the construction subject area is far different from what we encountered in the design area. In construction, much work has been done in the development of tools and techniques. Some attention has been paid to construction methods that make subsequent modification easier. But not much has been reported on modification techniques themselves.

Construction techniques

In our November 1977 report, we discussed user experiences with a number of programming aids marketed by IBM under the name of Improved Programming Technologies (IPTS). These methods include top-down development, structured programming, chief programmer teams, HIPO, pseudo code, development support library, structured walk-throughs, and an interactive debugging facility. All of these seek to impose a discipline on the program construction process. As we said in that report, "People we talked with were genuinely impressed (if not somewhat surprised) with the gains they had made in their software development process through the use of certain IPTS." So here is an example of some software engineering methods that are already having an impact on the software development process.

F. T. Baker (in Reference 11) discusses how some of these IPTS were introduced at IBM's Federal Systems Division. While work remains to be done on the methodologies, he feels that FSD's ex-

periences have been very positive. Further, he feels that the plan that FSD used to introduce these techniques was successful and could serve as a model for other organizations.

D. J. Reifer and Stephen Trattner (Reference 12) provide an extensive glossary of software tools and techniques. Some 70 types of tools and techniques are identified and classified in terms of where they can be used in the life cycle of a software system. A 61-item bibliography then gives sources of more detailed information on these tools and techniques. This paper in itself gives a good idea of the large amount of work that has been going into better methods to support the construction of computer programs.

Leon Stucki (in Reference 13), of Boeing Computer Services, gives a brief position paper on the acquisition of software development tools, based on experience at his company. A case example is presented, based on a lengthy investigation Stucki performed, wherein some of the more sophisticated programming support tools were analyzed. But, says Stucki, "Much work remains to be done in this area. We are still, for the most part, bench marking tools to find their costs and guessing as to their potential benefits."

In our review of construction tools and techniques discussed in the past year or two, most of the attention has been given to some aspect of structured programming.

Structured programming methodology. There have been numerous important books on modular and structured programming. These include the books by Dijkstra (Reference 2), J-D. Warnier (Reference 14), Michael Jackson (Reference 15), Edward Yourdon (Reference 16), and Glen Myers (Reference 17). We will make no attempt to review these works here, but we have discussed in previous reports Dijkstra's work (June 1974) and Warnier's work (December 1974).

Peter Neely (Reference 10a) describes a programming discipline that draws heavily on the work of Warnier but also acknowledges the work of others such as Dijkstra and Jackson. Neely advocates the top-down expansion of programs. Every task has a beginning, a middle, and an end, where the beginning and the end are executed at most once. The middle is typically a loop, and consists of a number of tasks. Each of these subtasks is next decomposed into its beginning, middle, and end. Neely gives some examples of how the discipline can be used in practice. And then he makes an interesting statement: "The only bug that I have encountered in my own programming in the past three years was in one (module) in which I did not use structured programming." How many programmers can come anywhere near making a statement like that?

Support of structured programming. "Structured programming sounds fine," some people say, "but it involves a lot of changes in method on the part of programmers. We have some pretty large application systems under development. How do we introduce structured programming in such situations without wrecking our schedules?"

V. R. Basili and A. J. Turner (Reference 8a) provide one answer to this question that they say has worked well for them. It is "iterative enhancement" of a software system. Trying to use a well modularized, top-down approach to building a system requires that the problem and its solution be well understood, they say. But even if the problem is a familiar one, it is often hard to achieve a good design on the first try. Design flaws may not show up until construction is underway, at which time corrections can involve a major effort.

Their approach starts construction with a simple skeletal subset of the problem. This skeletal solution should include a sampling of the key aspects of the problem, simple enough to understand and to build, and which will deliver a usable and useful product to the user. Further, this skeletal solution represents only an initial guess at the structure of the final solution. In addition to choosing the skeletal subset of the problem, the builder should develop a project control list of all of the things that still must be done.

Build this skeletal solution, they say, and give the outputs to the user. Find out if the design must be changed; if so, change it. Since only part of the overall problem is being tackled, complexity is much less and changes are not too much of a problem.

After the skeletal solution is working satisfactorily, take the next item(s) on the project control list and add it (them) to the solution. Repeat the process of checking outputs with the users. As the system expands, analyze it for structure, modularity, usability, reliability, and efficiency, they say. These analyses may require that new tasks be

added to the project control task list. Further, any difficulty in design, coding, or debugging a modification should signal the need for redesign or recoding of existing components. While this "do over" work may seem discouraging, most of it tends to occur during the early stages. As the iterations converge to the full solution, fewer and fewer modifications will need be made.

This approach of Basili and Turner is in contrast to other iterative approaches. The most commonly advocated iterative approach we are aware of might be termed "iterative refinement," where the *entire* system is initially built and then iteratively refined. It would appear that changes would be much easier and less expensive to make with Basili and Turner's method.

(Parenthetically, it was pointed out to us that this approach of Basili and Turner is an effective way to get user involvement, particularly when the problem and its solution are not well understood. Users usually can deal better with concrete system outputs than with abstractions or with a myriad of system details.)

R. J. Cunningham and C. G. Pugh (Reference 10c) discuss a software system they have developed (GLIDE) to support structured programming. It encourages development by successive refinement by allowing the builder to delay the definition of a section of code which need not be defined at the present stage of the program's development. The package provides a text editor for making changes to the program, and it automatically checks for changes that are then needed in interfaces to other sections of code. An incomplete program may be compiled and executed. And if a refinement proves unsuitable, a return can be made to the earlier version since both the original and the revised versions are retained.

Other types of tools. Much work is going on in the development of new programming languages for improving the programming function. But the use of new programming languages is something that is usually carefully controlled in many data processing shops. The existing "installation standard" programming languages, such as COBOL and PL/1, may not be the best languages possible for structured programming, but they can be used. Future languages, or future generations of these two languages, may be much more appropriate for structured programming, due to the research that is going on.

Similar statements can be made of operating systems. There is a lot of research and development work going on in operating system features and characteristics. But again, most programmers have no option on the operating systems they will use.

Other types of techniques. Many of the types of construction techniques that are a part of software engineering ere listed in Reference 3. These include sorting techniques, data compression, searching, validation, segmentation, hashing, and so on.

Gilb and Weinberg, in Reference 18, discuss the subject of humanized input systems that, in Knuth's words, interact gracefully with the human users. Special attention is paid by the authors to the needs of the data entry specialists (the "key punch girls"), in order to make their jobe more humanized. Attention is also paid to the newer systems where on-line data entry is performed throughout the organization, as a part of regular operations.

Modification techniques

The modification of information systems is often considered to be a "bad" thing. When modification is needed to fix errors of omission or commission in the problem definition or the building process, then it is bad. But when modification is used to enhance a system, we do not see it as necessarily bad.

Mills (Reference 8b) argues for better development procedures in order to reduce the need for maintenance. "In only 25 years, some 75 percent of data processing personnel are already taken up with maintenance, not development," he says. One reason for the magnitude of this maintenance effort is that the systems are maintained for an indefinite period of time after they have been developed. So a fraction of each development staff must be converted to maintenance. If 20 percent of each staff must be converted to maintenance every two years, says Mills, then at the end of 12 years almost 75% of the total staff time will be spent on maintenance—and that is just about what the actual situation is, he adds.

The other major reason for the high maintenance factor is that it has turned out to be more

difficult to develop "good" systems than was commonly supposed. By "good," Mills refers to both correctness and capability. So a large amount of staff time is needed to fix software that could have been built correctly at the outset.

(But the following reasonable question has been posed, in connection with Mills' point. Considering the newness of the field and the number of "self-taught" programmers and analysts, is it reasonable to expect that most software should have been built correctly?)

As mentioned earlier in this report, Mills argues for cleaner, more rigorous design as the main means for reducing this large maintenance factor. Good design would eliminate much of the corrective maintenance. And by better identifying what the systems must perform, at least some portion of the adaptive maintenance may be eliminated.

One cannot argue with Mills on the need for better design and construction methodologies, for reducing the need for maintenance. At the same time, *some* adaptive maintenance—more commonly called enhancement—will continue to be needed. We think that methods for more efficient maintenance are now needed and will continue to be needed.

In our search of the literature, however, we did not come across any papers that addressed this subject.

In our June 1972 issue, we described the development methods that Copley Computer Services, Inc., of San Diego, California, began using some years back. This company switched completely to on-line program development, after they installed a DECsystem-10. With this plus some of the other procedures described in our report, they soon turned the whole maintenance picture around. Where before, with batch development methods, they were spending some 80 percent of staff time on maintenance and 20 percent on development, with the new procedures these percentages were reversed.

In our report last month, on data dictionaries, we pointed out that some dictionaries provide both production and test status. Maintenance can be performed on data definitions in the test status. Only when the changes have been satisfactorily checked out need the new data definitions be moved into production status.

As we say, we think much more attention should be paid to tools and techniques for supporting system, program, and data definition maintenance.

Quality assurance

There are many aspects of quality assurance for computer programs, but the prime aspect is, of course, program correctness. As with physical products, it is recognized that quality cannot be "tested into" computer programs. The programs have to be built right to begin with. Testing or any other form of checking or inspection only provides a measure of how well the building job has been done.

We will discuss three methods of checking systems, programs, and data definitions, to measure their quality. These methods are: inspections, testing, and audits. Following this, we will give a short discussion of the proof of correctness concept. From the standpoint of building quality into computer programs, probably *the* area of major interest today is the "proof of correctness" methodology. This discipline seeks to develop the proof that a program is correct concurrently with the development of the program itself. The method is still largely in the research and development stage but we feel that data processing management should be aware of it.

It should be mentioned here that quality assurance by checking is very closely related to the subject of *evaluation*, which we will discuss next month.

Inspections

There are a number of ways in current use by which computer programs are inspected for correctness during the development process. As far as we can tell, though, most of the attention is being given to two types of *informal* inspections and one type of more *formal* inspection.

Informal inspection methods. Probably the most widely used of the informal inspection methods is the *structured walk-through*, marketed by IBM (Reference 19). We discussed some user experiences with this method in our November 1977 report; in general, the users we have talked to have been very happy with the method.

In a structured walk-through, a programmer, say, describes his program design and his code to a small, selected group of other programmers. He provides copies of the documentation to the

group members and then "walks through" the logic of the program. The group raises questions, points out inconsistencies, and so on, in the course of the meeting. The programmer is then expected to consider all of the points raised and to refine the design and/or code as required.

Edward Yourdon (in Reference 13) has observed that the structured walk-through is perhaps the best way to introduce other programming productvity techniques. In Yourdon's words, "If a project manager establishes an environment of exposing everyone's code to public discussion, then he will ensure that a relatively uniform version of top-down implementation, structured design, and structured programming can be implemented later on."

Another type of informal inspection is that advocated by Gerald Weinberg in his popular book, *The Psychology of Computer Programming* (Reference 20). Weinberg urges the use of "egoless programming" and "self-adaptive" teams; we discussed his concepts in our May 1974 report. Programmers are organized in loosely structured teams; the team leader of the moment is the person with the most capability in the function that the team is currently working on. Further, each programmer submits blocks of code to the other team members, for them to read. The team members recognize that everyone makes mistakes—and that, on any given day, a person can make a horrible number of mistakes. Also, the team members look for better ways of doing things. The resulting programs are thus rightfully team efforts. A team member does not view a set of programs that he has worked on as "my" programs but rather as "our" programs.

In both of these types of informal inspections, the conduct of the inspections is left pretty much in the hands of the participants. Formal inspections, on the other hand, make use of a well-defined methodology.

Formal inspections. M. E. Fagan (in Reference 7) has described a formal inspection method that he has helped develop and which he claims is more effective than structured walk-throughs. We discussed his approach in our July 1977 report. A walk-through is just an educational process, he says; further, the participants tend to get sidetracked into discussing design alternatives, and the process is not self-improving. He seeks to correct these shortcomings.

Fagan's formal inspection method has five parts. (1) First is a short overview of the work to be inspected, presented by the analyst or designer who did the work. This is the educational, familiarization step. (2) Next, each of the inspection team participants is given copies of the documentation and is expected to do "homework" on it, to gain understanding. While some errors may be caught in this step, this is not the aim of the step. (3) The third step is the inspection meeting itself. The goal of the meeting is to *find errors*. The moderator of the meeting must not let the discussion get sidetracked into, say, considering alternative designs. Just find the errors, don't try to solve them, says Fagan. Following the meeting, the moderator is expected to write up an inspection report, listing all of the errors. (4) The next step is to rework the work, to get rid of the errors. (5) Finally, at least the moderator (and perhaps the whole team) must perform a follow-up to see that all fixes have been made and made properly.

Over a period of time, a checklist of error types can be developed from the inspection reports. The list should indicate their relative frequency and their severity. Also, procedures on how to look for the errors should evolve, with particular attention paid to the most frequently occurring and the most severe error types. The checklist and these procedures can be used in the inspection meetings themselves, as well as by the individual analysts and programmers to help them do their jobs better.

The inspection team should be a small one, says Fagan, with perhaps four people. The moderator is the key person. He or she should be competent in the particular application being reviewed. The moderator must keep the inspection process moving along and not let it get sidetracked.

These then are the inspection methods that seem, in our opinion, to be attracting the most attention today. They seek to remove errors and improve design during the construction process. But they do this by *checking* the work of the individual analyst or programmer.

Testing

Quality assurance in "conventional" programming seems to be based on the concept: "program as best you can and then test the dickens out of it." Testing has thus been a basic part of the programming function since the beginning

days of computers. One would suppose that testing would be fairly well understood by now. But this is just not so, we are advised. It is still pretty much a hit-or-miss affair. Even worse, it has not been studied deeply and not much technical literature has been written on it.

In our research for this report, we did not come across any literature which was cited as the "bible" on testing. Apparently that book or paper is still to be written.

But still, there is some literature on the subject. R. D. Hartwick (in Reference 13) discusses the subject of test planning. He covers test objectives, test planning and organization, error sources and detection methods, selection of test methodology, and test standards. He presents two tables on error categories, indicating the number of errors found in 11 projects, the severity of those errors, and the detection methods that were used for locating those errors. While this probably would be classified as an overview paper, it does provide guidance on how to approach the testing activity.

The *Transactions on Software Engineering*, published by the Computer Society of the Institute of Electrical and Electronic Engineers (Reference 8), frequently publishes papers on testing. The September 1976 issue, for instance, includes five papers on current research areas in software testing. In his editorial, the guest editor, Leon Stucki, says "Despite all of the software engineering rhetoric, it is really quite remarkable that we know so little about software testing . . . it is (my) hope that this set of papers might generate a greater motivation for research into many of the still unanswered questions relating to software testing."

The upshot is, then, that in this overview of the subject of software engineering, we are unable to direct you to definitive information on the subject of software testing. Like problem definition and design, it is a subject area that has received all too little attention.

Auditing of software

In March 1977, the U. S. National Bureau of Standards and the General Accounting Office jointly sponsored a working conference on the audit and evaluation of computer security. Reference 3b reports the results of that conference. While the conference topic concerned computer security, in point of fact much of the discussion dealt with the broader subject of audit and evaluation of computer software. The results of the conference will be used by a federal information processing system task group on computer system security to develop guidelines for federal agencies.

The conference had ten working panels, each addressing an aspect of the subject. Of the ten, we will single out two for discussion here.

The audit of program integrity. Program integrity was defined to be the characteristic that a program does what its specifications say it should do, and should do nothing else. But the discussion pointed out that this definition could be challenged; instead of "specifications," perhaps "requirements" would be more appropriate—or perhaps better yet, "mission." In other words, what defines what the program should do? It might be easier or more practical to audit the program against its specifications, but more meaningful to audit it against its requirements or mission.

Perhaps the major conclusion of this working panel was that program integrity has to be built into the program from the outset; it cannot be added as an after-thought. All that an audit can do is to see that program integrity considerations are a part of the regular program development process. So, as we see it, program integrity considerations should be a part of the body of knowledge of software engineering.

The working panel identified six types of threats that can challenge the integrity of a computer-based system. From high to low severity, these are: irrational attack, conspiracy by a team, a browser through the files, a user who stumbles upon something important, human errors, and natural failures. A program with a high degree of integrity should do what it is supposed to do, and nothing else, when subjected to such threats.

How can program integrity be achieved or enhanced? The panel identified three general categories of methods. *Program correctness* can be measured by static evaluations, such as reviews, inspections, and proof of correctness, and/or by dynamic evaluations such as by running the program to see how it works or by checking compiler results. *Program robustness* can be achieved by on-going testing after installation, by on-going monitoring and control, and by the use of planned

redundancy. Finally, *trustworthiness* in the program can be enhanced by the use of skilled people in the development process, by the use of good development practices, and by the use of good tools to aid in development.

In short, this working panel identified a number of tools, techniques, and methods that can be used to promote program integrity. But the panel pointed out that integrity must be *engineered* into the software.

Auditing under different system environments. This working panel concluded that the environment under which a system will operate must be identified and then steps must be taken to control that environment. Further, this control of the environment must be *engineered* into the system from the beginning. Auditing should verify that this control of the environment is being considered during the development process.

The panel identified a number of key factors that characterize the environment. These include the degree of resource sharing, the type of service (batch, interactive), centralized versus distributed, local or remote user access, the sensitivity of the information in the system, the threats that the system is likely to face, and so on.

There are many tools and techniques available for controlling the environment and providing protection, said the panel. A number of these were listed, including site perimeter controls, backup systems, audit trails, change control procedures, and so on.

Furthermore, said the panel, audit checklists can and should be developed to help the auditors determine how effectively environment controls have been engineered into the overall system implementations.

As we see it, auditing is one more method, along with inspections and testing, for checking to see whether good software engineering practices are being followed.

Proving correctness

In his review of Dijkstra's book, *A Discipline of Programming*, W. C. McGee (Reference 21) has this to say: "Anyone who has ever been associated with the development of a large or complex program is familiar with the tendency for such programs eventually to become so large and so complex that no one individual really understands them or can reliably predict how they will behave in some previously unencountered situation . . . Such programs are examples of what E. W. Dijkstra calls *unmastered complexity* . . . When (this) occurs in human artifacts such as computer programs, it is unconscionable."

To attack the problem of mastering complexity in large programs, Dijkstra seeks to develop the program and its proof of correctness concurrently, says McGee. In fact, the act of proving the correctness often suggests the form the program should take, derived in an almost mechanical manner.

What is the "proof of correctness" concept? David Gries, in Reference 8b, has presented an interesting overview of it. We will give our understanding of Gries' thoughts.

To illustrate the concepts involved, Gries uses an example of writing a program to justify lines of type—that is, to even out the right hand margins. Extra blanks are to be inserted between pairs of words so that the last character of the last word in a line appears in the last column of the line.

But, as always, there are complexities. So that the spacing appears even, the number of blanks between different pairs of words on one line should differ by no more than one. Also, to make the extra blanks less evident to the reader, blanks will tend to be inserted toward the left of even numbered lines, say, and toward the right on odd numbered lines.

The problem is, then, given the column numbers where the words begin on the unjustified line, to find the column numbers where the words will begin on the justified line.

In conventional programming, says Gries, the programmer starts out by naming the procedure ("justify") and the variables, and then proceeds to develop an algorithm to do the job. But this is not sufficiently precise to insure a correct program, he now has come to see.

With the proof of correctness method, the programmer must define the pre- and post-conditions for the program. The pre-condition is the assertions about the input. Gries identifies two such assertions. One, the extra spaces that are to be inserted must be equal to or greater than zero; no negative spaces can be inserted. Two, the column numbers of the beginning of the words on the line are equal to the initial column numbers for all of the words on the line.

If these assertions seem rather trivial and obvious, Gries points out that it took him several iterations to develop the assertions for the program. And Harlan Mills, at the Software Engineering Conference held in San Francisco in October 1976, says that a proof of correctness consists of the programmer making a number of assertions, for each of which he could justifiably say to another programmer, "Now that is obvious, isn't it?"

Next Gries makes assertions about the post-condition, the output. These assertions say, in general, that the extra blanks will be inserted between the first and last words of the line. Again, there can be no negative blanks. And the blank spaces between words to the left of word "t" (a word falling between the first and last words) are defined differently from the blank spaces to the right of word "t."

These assertions provide understandable, precise specifications for the algorithm, but not in so much detail that chaos results, says Gries. With the understanding of the pre- and post-conditions, he then makes his first attempt to write the algorithm—a function that defines the blanks for the word pairs to the left of word "t" and for the word pairs to the right of word "t."

While our description probably has not done justice to the concepts presented, we hope it does give an idea of the mental processes involved. Gries' proof of correctness really is just getting under way at this point, and the interested reader is referred to his paper for the details.

As "obvious" as the above assertions might seem to be, they surely are obvious only in hindsight (to the programmer who developed them) or when someone else has already developed them. The hardest part of writing this algorithm was not the programming itself, says Gries, but rather it was developing the specifications, the assertions.

Hopefully, this brief discussion has given some "feel" of the proof of correctness methodology that is now in the research and development stage. As we said earlier, proof of correctness is perhaps the area of major interest among researchers working on the problem of quality assurance for computer programs.

Next month

Next month we will continue our overview of the state of software engineering. We will discuss evaluation methods, which are a part of the software engineering methodology. We will then get into a discussion of software engineering *management* methods. As pointed out earlier in this report, some people we have talked to feel that while the software engineering methods may be necessary for the development of good software, they certainly are not sufficient for achieving that end. The management practices under which the development is carried on also play a major role.

REFERENCES

1. Knuth, Donald, "Computer programming as an art," *Communications of the ACM*, ACM (1133 Avenue of the Americas, New York, N.Y. 10036), December 1974; p. 667-673; price $5.00 prepaid.
2. Dijkstra, E. W., *A discipline of programming*, Prentice-Hall, Inc. (Englewood Cliffs, N.J. 07632), 1976; price $14.95.
3. Reports by U. S. National Bureau of Standards; order from Superintendent of Documents, U. S. Government Printing Office, Washington, D. C. 20402:
   a) Report on planning session on software engineering handbook, Tech Note 832, Nov. 1974; SD Cat. No. C13.46:832; price 70 cents.
   b) NBS Special Report 500-19, Audit and Evaluation of Computer Security, stock number 003003-01848-1; price $4.
4. *Proceedings of Second International Conference on Software Engineering*, IEEE Computer Society (5855 Naples Plaza, Suite 301, Long Beach, Calif. 90803), 1976; price $20. For papers presented at the conference but not in the proceedings, see Reference 8 below, the December 1976 and January 1977 issues.
5. For more information on SADT, write SofTech, Inc., 460 Totten Pond Road, Waltham, Mass. 02154.
6. For more information on PSL and PSA, write ISDOS Project, 231 West Engineering Building, University of Michigan, Ann Arbor, Mich. 48104.
7. Fagan, M. E., "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, IBM Corporation (Armonk, N.Y. 10504); Vol. 15, No. 3, 1976; p. 182-211.
8. *Transactions on Software Engineering*, IEEE Computer Society (address above), price $10 per copy, although Society members may subscribe for $6 per year; the following issues have been cited in this report: (A) December 1975; (b) December 1976; (c) May 1977; (d) September 1976; (e) January 1977.
9. *Best Practices Manual on Security*, AFIPS Press (210 Summit Avenue, Montvale, N.J. 07645. The first edition of this manual is now out of print and the revised edition is expected to be published by June.
10. *Software Practice & Experience*, John Wiley Sons, Ltd. (Baffins Lane, Chichester, Sussex, U.K.); price £17.50 per year; in U.S., $75 per year via "air speeded" delivery; the following issues have been cited in this report: (a) January-March 1976; (b) April-June 1976; (c) October-December 1976.
11. *Proceedings of International Conference on Reliable Software, April 1975*; SIGPLAN Notices, Vol. 10, No. 6, June 1975; order from ACM (address above), price $25 prepaid.
12. Reifer, D. J. and S. Trattner, "A glossary of software tools and techniques," *Computer*, IEEE Computer Society (address above), July 1977, p. 52-60; price $10.
13. *Proceedings of 1977 National Computer Conference*, AFIPS Press (address above), price $60 paper, $15 microfiche.
14. Warnier, J-D., *Logical Construction of Programs (L.C.P.)*, Van Nostrand Reinhold Co. (450 West 33rd Street, New York, N.Y. 10001), 1974; price $14.95.
15. Jackson, M. A., *Principles of Program Design*, Academic Press (111 Fifth Avenue, New York, N.Y. 10001), 1975.
16. Yourdon, E., *Techniques of Program Structure and Design*, Prentice-Hall, Inc. (address above), 1975.
17. *Myers, G. J., Software Reliability; Principles and Practices*, Wiley-Interscience (605 Third Avenue, New York, N.Y. 10016), 1976.
18. Gilb, T. and G. M. Weinberg, *Humanized Input*, Winthrop Publishers, Inc. (17 Dunster Street, Cambridge, Mass. 02128), 1977.
19. "Code reading, structured walk-throughs and inspections," IBM Manual GE 19-5200; order through your local IBM office.
20. Weinberg, Gerald M., *The Psychology of Computer Programming*, Van Nostrand Reinhold Co. (address above), 1971.
21. McGee, W. C., "Review of *A discipline of programming* by Edsgar W. Dijkstra," *Popular Computing* (Box 272, Calabasas, Calif. 91302), August 1977, p. 14-20; price $2.50.
22. Freeman, P. and A. I. Wasserman, *Tutorial on Software Design Techniques*, IEEE Computer Society (address above), 1976; 277 pages; price $12.
23. *Datamation* (1801 S. La Cienega Blvd., Los Angeles, Calif. 90035), November 1977. This issue features four papers on analysis and design.

SUBJECTS COVERED BY EDP ANALYZER IN PRIOR YEARS

1975 (Volume 13)

Number

1. Progress Toward International Data Networks
2. Soon: Public Packet Switched Networks
3. The Internal Auditor and the Computer
4. Improvements in Man/Machine Interfacing
5. "Are We Doing the Right Things?"
6. "Are We Doing Things Right?"
7. "Do We Have the Right Resources?"
8. The Benefits of Standard Practices
9. Progress Toward Easier Programming
10. The New Interactive Search Systems
11. The Debate on Information Privacy: Part 1
12. The Debate on Information Privacy: Part 2

1976 (Volume 14)

Number

1. Planning for Multi-national Data Processing
2. Staff Training on the Multi-national Scene
3. Professionalism: Coming or Not?
4. Integrity and Security of Personal Data
5. APL and Decision Support Systems
6. Distributed Data Systems
7. Network Structures for Distributed Systems
8. Bringing Women into Computing Management
9. Project Management Systems
10. Distributed Systems and the End User
11. Recovery in Data Base Systems
12. Toward the Better Management of Data

1977 (Volume 15)

Number

1. The Arrival of Common Systems
2. Word Processing: Part 1
3. Word Processing: Part 2
4. Computer Message Systems
5. Computer Services for Small Sites
6. The Importance of EDP Audit and Control
7. Getting the Requirements Right
8. Managing Staff Retention and Turnover
9. Making Use of Remote Computing Services
10. The Impact of Corporate EFT
11. Using Some New Programming Techniques
12. Progress in Project Management

1978 (Volume 16)

Number

1. Installing a Data Dictionary
2. Progress in Software Engineering: Part 1

(List of subjects prior to 1975 sent upon request)

PRICE SCHEDULE

The annual subscription price for EDP ANALYZER is $48. The two year price is $88 and the three year price is $120; postpaid surface delivery to the U.S., Canada, and Mexico. (Optional air mail delivery to Canada and Mexico available at extra cost.)

Subscriptions to other countries are: One year $60, two years, $112, and three years $156. These prices include AIR MAIL postage. All prices in U.S. dollars.

Attractive binders for holding 12 issues of EDP ANALYZER are available at $6.25. Californians please add 38¢ sales tax.

Because of the continuing demand for back issues, all previous reports are available. Price: $6 each (for U.S., Canada, and Mexico), and $7 elsewhere; includes air mail postage.

Reduced rates are in effect for multiple subscriptions and for multiple copies of back issues. Please write for rates.

Subscription agency orders limited to single copy, one-, two-, and three-year subscriptions only.

Send your order and check to:
EDP ANALYZER
Subscription Office
925 Anza Avenue
Vista, California 92083
Phone: (714) 724-3233

Send editorial correspondence to:
EDP ANALYZER
Editorial Office
925 Anza Avenue
Vista, California 92083
Phone: (714) 724-5900

Name_____

Company _____

Address _____

City, State, ZIP Code_____