EDP ANALYZER

©1978 by Canning Publications, Inc.

MARCH, 1978
VOL. 16, NO. 3

PROGRESS IN SOFTWARE ENGINEERING: PART 2

This is the second of two reports that survey what is happening in the area of "software engineering"—the design, construction, and maintenance of software systems under an engineering discipline. Last month we discussed some of the tools and techniques that have been developed for problem definition, design, construction, modification, and quality assurance when software systems are created—both system software and application software. In this report, we discuss some tools and techniques for managing the creation and maintenance of software. We think you will be hearing a lot more about software engineering from now on.

Engineering is concerned with how to design, construct, and operate human artifacts such as machines, bridges, buildings, and so on. The engineering disciplines that are the strongest are those that have mathematically based theoretical foundations. Examples include aeronautical engineering, electrical engineering, and chemical engineering. Some other disciplines that use the term "engineering" (and often deal mainly with human activities) are based not on mathematical foundations but rather on field experiences. Quality of results seems to be much more variable in these latter disciplines, we gather.

What about the engineering of software? Does it have a theoretical foundation upon which to base the engineering practices?

One of the people who gives a "yes" answer to this question is Kenneth Kolence, who has developed what he calls "software physics" as a foundation for software engineering. In our August 1975 report, we discussed some of Kolence's ideas and how they were being applied in a rational method for computer job costing and charging. We still feel it to be one of the most interesting methods of job costing and charging that we have come across.

In 1975, Kolence organized the Institute for Software Engineering, which has members and which acts as a technical trade association. The purpose of the Institute is to assist its members to put into practice quantitative software engineering techniques. The main focus of Institute activities to date has been on the applications of software physics to capacity management, including job costing and charging. As of this writing, the Institute has 120 member organizations, of whom about 10 have fully implemented the costing aspect of capacity management.

Kolence's concepts of software physics include software power, software work, and storage capacity usage. An approximate definition of software work is: a processor does one unit of software work on a storage device when it transfers one byte to that storage device. Software work thus is always associated with changing storage. For example, a tape drive does 1000 units of work on main storage when it reads a 1000 byte block. It also does 1000 units of work on the tape when it writes the same block. It is immaterial how many bits within the byte are changed, as long as the symbol state is changed.

The concept of software power is used in two ways, says Kolence. One is for capacity measurement—determining the capacity available to do work per unit of time. The other is for performance measurement—the actual rate at which work is performed. Further, work is invariant but power is not; the work associated with a given workload is essentially fixed but the power to perform the workload will vary with machine configuration.

Let us consider briefly an application of these concepts to capacity management. A frequently asked question in computer centers is: "How much capacity is still available with our present hardware configuration?" And a related question is: "How much capacity will we have left after we add that specified new application?" Kolence seeks to give objective, quantitative answers to such questions.

CPU power. The CPU power means the number of units of software work that the CPU can perform per second of execution time. It is determined by measuring all of the bytes transferred to or from main storage and the CPU in a second. Average power is the number of units of software work that can be performed over an extended period of time; peak power is a rate that can be sustained only for a second or less.

It should come as no surprise that the power of a specific model of CPU can be determined only by extensive measurements, using (primarily) hardware monitors. But even with monitors, there may be problems. The needed probe points are not always known—or even available. However, as more people attack these problems, answers seem to be found. At a seminar we attended, Kolence gave the following general, average power figures as illustrative of what is being found. An IBM 370/158 has the power to perform about 7 million units of software work per second. The power figure for the IBM 370/165 was about 12 mw/s and for the Model 168, 20 mw/s. Kolence cautioned against taking these figures too literally, for reasons we will mention shortly.

Tape drive power. The power of a tape drive to perform work should be a function of the block size and the number of blocks written per second. Power increases as the block size increases and approaches the maximum transfer rate. But power is constrained by contention for the tape control unit. Software work calculations point out how little the power of a configuration is increased when a tape drive is added but the control unit bottleneck continues.

Disk drive power. Disk drive power is calculated much the same as tape drive power—but the results can be even more dramatic. In one study that was presented at the seminar mentioned above, after the system had six 3330-type spindles, there was a trivial increase in available power with each spindle added. The constraints occurred in the control units and channels.

What we are saying is that Kolence has developed a theoretical approach that makes use of software work and software power concepts. His goal is to use these (and other related) concepts to provide quantitative, objective answers to questions about the ability of configurations to perform work.

The concepts are still evolving. For instance, for the smaller IBM 370 models, 135 and below, each instruction fetches one byte—and in the process of transferring that byte, does one unit of software work. But in the larger CPUs, an instruction might cause 100 or more bytes to be fetched when actually only one byte has been requested. This design has been used to increase machine efficiency. The machine is so fast that not much time is wasted in fetching the additional bytes—and further, the next instructions needed may well be in those bytes. But when the extra bytes are *not* used, "waste work" has been done. The CPU has not done 100 units of software work, say, but perhaps only 1 or 2, and the other 98 or 99 bytes that were transferred represented waste. In other instances, a large number of the bytes transferred might actually be used, so that the waste would be less. The amount of waste here can be appreciable, so efforts are continuing to find ways to measure it. In so doing, the ways of increasing the effective capacity of a CPU are being identified.

We find Kolence's ideas stimulating. And we think they represent one approach to how theoretical concepts can be used to form the foundations for software engineering. For more information, see Reference 1.

In these two reports, last month and this month, we are reviewing the status of software engineering as we see it, based on conferences and workshops that we have attended, plus a review of a good amount of literature. Last month, we discussed some important progress in the subject areas of (1) problem definition and requirements analysis, (2) architecture, engineering, and design of software systems, (3) construction and modification of these systems, and (4) quality assurance.

In this report, we will conclude our discussion of software engineering methodology *per se* with an overview of what is happening in methods for evaluating quality and performance of software. Then we will discuss some improved techniques for *managing* software engineering projects. As we mentioned last month, there are people in the field who believe that the management techniques are every bit as important for ultimate success as are the software engineering techniques themselves.

Evaluation of software

The methods to be discussed here are those used for evaluating the quality and performance of software systems. Since the context here is an engineering one, the methods should be quantitative in nature. Ideally, they should also have good theoretical foundations and not be based solely on experience. However, this ideal is as yet seldom realized.

Evaluation for design purposes

Gilb (Reference 2) has collected a substantial amount of empirical data for defining "good" performance of software. In the absence of theory for defining good performance, the field must rely on empirical data. In addition, empirical data can be useful for checking the validity of a new theory.

The contents of some of the chapters covered in Reference 2 will give an idea of how Gilb approaches this subject. In his first two chapters, he discusses methods of testing and of predicting the reliability of programs. One method described is what he calls the "bebugging" technique. In this technique, a number of known bugs are inserted into a program that is to be tested. The ratio of how many of these are found by testing, as compared with the number inserted, is then used to estimate how many bugs in total are in the program. That is, if testing found three-quarters of the planted bugs, and if, say, fifteen non-planted bugs were also found, this is a rough indication that there might be about five more non-planted bugs in the program. There is obviously no assurance that the ability to find planted bugs is the same as the ability to find non-planted bugs. But the technique does give the programmer (and management) some feeling about the number of bugs that may be remaining in the program.

For his discussion of inspection of software, Gilb draws heavily on the work of Michael Fagan of IBM, which we discussed last month.

Gilb then gets into several chapters dealing with human behavior aspects of producing software. He describes "motivational metrics" which include financial penalties for not meeting contract performance measures. Another technique uses weights and points for making multi-element component comparisons and analyses. And he suggests the use of "dual code"—having the same program coded twice as an indirect measure of the quality. This dual coding adds only about 5% to 10% to the total cost of the project, he says, and conceivably could actually save money.

We have touched on only a few of Gilb's chapters, but this discussion should give an idea of how he approaches the subject of software metrics. We have classified this material as applying to the design of software; it could equally well be classified as applying to the construction of software.

Evaluation to improve performance

This use of evaluation methods is somewhat like the subject of industrial engineering. Techniques are developed and applied for detecting wasteful activities and operations, so that they can be redesigned in order to reduce the amount of waste. And, in fact, much of the application of Kolence's concepts of software physics has been directed to this area.

Evaluation techniques for improving performance can be used for "tuning" the operation of existing systems. In addition, they can contribute to the "good practice" type of design principles, for improving the design of new software systems.

There is a wealth of material being published on computer performance evaluation methods. We would single out for mention two periodicals: (a) *Performance Evaluation Review,* Reference 3, published by the ACM Special Interest Group on

Measurement and Evaluation, and (b) *EDP Performance Review*, Reference 4, published by Applied Computer Research.

As an illustration of the type of material being published within this subject area, we will cite two references.

K. Sreenivasan (Reference 5) discusses how accounting data collected by an operating system (in this case, IBM's SMF facility) can be used for evaluating computer system performance. This is a theme that appears frequently in the literature. The technique should not be overlooked by people interested in evaluation methods.

J. L. Elshoff (Reference 6a) describes an analysis of 120 commercial PL/1 programs obtained from several data processing installations at General Motors Corporation. The programs were scanned both manually and automatically, to identify areas in which a better programming job could be done. Here are some of the findings. In the area of *program size*, some programs exceeded 3700 source statements in length, with an average of 853 statements. The programs were, in general, monolithic in structure and were not modularized. As far as *program readability* is concerned, specific attributes that aid readability include readable identifiers, indentation, pagination, and descriptive comments. The analysis showed that these were inconsistently used or not used at all, Elshoff reported. *Program complexity* is indicated by the number of data items used by a program and the flow of control through the program. The analysis found an average of 384 data items identified in a program; of these, 277 were used one or more times, requiring the programmer to know the names and definitions, while the other 107 items were defined but not used by the program. For flow of control, the average program had 152 IF statements, 100 GO TO statements, and 162 "spans" (number of statements between two references to the same identifier), with 13% of these spans exceeding 100 statements. For these and other reasons, discussed in the paper, Elshoff argues for a structured discipline for program development.

In a follow-on paper (Reference 6e), Elshoff compares these 120 non-structured programming (NSP) programs with 34 structured programming (SP) programs developed later by the same groups. The average SP program was about 30% smaller than the average NSP program, in number of source statements. Most programmers were able to get rid of most GO TO statements but this was offset by an increase in such other types as CALL and PROCEDURE statements. The SP programs had much cleaner structures, they read from top to bottom and from left to right (not true of NSP programs), had deeper nesting levels but more constrained flow of control, and were more readable and understandable. Structured programming certainly improved the end product of the programmers, said Elshoff.

So a wide variety of quantitative methods are being used to evaluate the quality and performance of the software development staff, the products that they develop, and the ability of users to effectively use that software. We feel that such methods are properly part of the emerging discipline of software engineering.

Let us now turn our attention to some aspects of managing the software development process, where methods are needed to support software engineering.

SOFTWARE ENGINEERING MANAGEMENT

As we reported last month, we participated in a planning meeting for a software engineering handbook some time back. The meeting was sponsored by the U.S. National Bureau of Standards, the National Science Foundation, and the Association for Computing Machinery, and is reported in Reference 7. Some of the participants in this meeting felt that the *management* techniques used in the software development process are every bit as important as the engineering techniques that are used. Since that meeting, we have heard this same theme repeated on numerous occasions. (In fact, Kolence contends that *no* technical methods are acceptable as software engineering techniques unless they provide effective management techniques based on them.)

The main argument used in support of this position is that if poor management techniques are used, the development staff can become frustrated by the seemingly arbitrary edicts and changes to plans that management imposes. This frustration, in turn, detracts from the effectiveness of the engineering techniques used.

The subject being discussed here, then, is the methodology that *managers* (at several levels) can use for the better management of software development and modification.

A better management methodology comes from a better understanding of the software development process. True, some of the characteristics of this process seem to be reasonably well understood. Certain problems have arisen again and again, and in some cases, fairly effective solutions have been developed. But, as we will point out, some important behavior patterns are just now coming into focus.

There are some characteristics of the software development and modification process that are not too evident nor well understood, we believe. And even though they are not well recognized, they still can be very significant. Software engineering management seeks to determine such characteristics and to determine the management problems they can cause. The next step, of course, is to develop practical solutions for alleviating such problems.

For our discussion here, we will single out the following three characteristics of the software development and modification process, expressed in terms of behavior patterns:

- Staff behavior patterns
- Project behavior patterns
- System behavior patterns

We will begin with the staff behavior patterns.

Staff behavior patterns

Many of the behavior patterns of the system development and maintenance staff are well known to data processing management, of course. One of the most common behavior patterns is the resistance of this staff to the acceptance of new methods, particularly by the older staff members.

Software engineering seeks to introduce a large number of new, disciplined methodologies. The *discipline* aspect of these methodologies will often be the most difficult for a development staff to accept. Let us see what software engineering management has to offer for this problem area.

Introducing change

Edward Yourdon (Reference 8a), in discussing the choice of new methodologies for software development, points out that production type projects have different trade offs from research and development projects. The latter can experiment more with new methods. But production projects have hard deadlines to meet and must thus be more careful in choosing what is to be tried out.

(It was pointed out to us that the use of the terms "production type projects" and "research and development projects" can lead to misunderstandings. Perhaps a better way to differentiate projects is on the extent to which their problems and solutions are defined at the outset. A true R & D project should experiment with methods only if that experimentation was planned in advance.)

Yourdon presents an overview of the main methodologies currently being introduced into software development. These include structured programming (perhaps more accurately called structured coding, involving the control constructs of sequence, selection, and iteration), structured design, structured analysis, HIPO and other documentation techniques, top down development, structured walk-throughs, and chief programmer teams. (We discussed user experiences with all of these in our November 1977 issue.)

The point to be made here is that this set of methodologies represents a substantial change in the way software is developed at most organizations. What are the main points about which data processing management should be concerned, for introducing these new methods?

From his experiences with introducing such new methods, Yourdon has developed a number of suggestions. First and foremost, he says, trying to introduce all of these new methods at one time will generally lead to disaster. The methods should be installed gradually. Next, he points out, techniques that involve organizational change—such as chief programmer teams—are often the most difficult to install. Further, it can be a waste of time to try to install structured coding without also installing structured design. These seem to be the main "do not do" suggestions made by Yourdon.

On the positive side, the most successful approach to installing the new methods has often been to start with walk-throughs, he says. These walk-throughs are informal meetings for reviewing the correctness and quality of the analysis, design, code, test data, and documentation work products. Project management might start with walk-through meetings for reviewing code, and create the environment where *everyone's* code is exposed to public discussion. These meetings will allow each person on the development staff to see how he is doing relative to a "standard practice." Further, the problems that arise from not following the standard practice will become apparent. So the walk-throughs will often set the stage for the gradual introduction of other methods.

The second positive suggestion made by Yourdon is that top-down design and programming might be the next methods to install. These methods are often a good way to introduce the new structured development methods to the staff, he says.

In brief, software engineering involves the introduction of many new disciplined methods into the software development and maintenance process. It is up to data processing management to determine how best to introduce these changes, in order to minimize "upheaval" and to promote the efficient use of the new methods. For a further discussion, see our November 1977 report.

Matching skills to job needs

This subject might be termed "human engineering" in which it is attempted to effectively interface the new methodologies with the humans that will be using them.

Freedman, Gause, and Weinberg (Reference 8b) recommend using an environment which encourages continual on-the-job education through formal and informal review, as opposed to formal training. What they advocate is the use of self-organizing teams, as opposed to the idea of chief programmer teams.

They report that a common cause of failure in trying to use some new organizational structure for a software development staff is mis-estimating both the staff talents and the task to be done. When the chief programmer team approach is tried, an inadequate person may be selected for the chief programmer, or the person selected to back up the chief programmer may be more qualified than the chief. Both of these mistakes can lead to failures of the approach. The chief programmer *must* be the best person on the team and in addition must be adequate for the job, including the ability to lead the team. The existing staff talents just may not support the idea of chief programmers.

It is very important, say the authors, to let the technical leaders do the *technical* leading of the project. For one thing, teams should be small, with no more than 5 to 6 members, so that supervision activities are minimized. Perhaps more important, recognize that the different team members have varying strengths and weaknesses; the "chief" role can thus float among the team members, as the project progresses through its several phases. For instance, whoever on the team is strongest in analysis and problem definition probably would be the "chief" during that phase of the project.

Instead of first organizing the project itself, say the authors, select the people who will be on the project, and then let them organize the project. Further, make sure that each team uses a continual review process, where all work products are reviewed by the several team members. Never let a problem become undefined, as new or changed user requirements come to light; make sure that all such changes are carefully reviewed and understood by the team.

The self-organizing team concept may be a more flexible approach to team programming than the chief programmer team idea. It is an approach that data processing management might well want to consider.

Limiting the objectives

Mills (Reference 6b) points out that human fallibility tends to lead from grand to grandiose systems which soon outstrip human intellectual capability for management and control. Further, such systems are often developed in an "open loop" manner, with neither periodic reviews nor any go/no-go checks by users. Here, then, is an all-too-frequent staff behavior pattern that causes serious problems in software development.

What Mills proposes is the "divide and conquer" approach. Divide large projects into small sub-projects that can be conducted by teams of from 3 to 6 members. Further, since for all practical purposes it is impossible to get both complete and accurate user requirements at the outset of a project, provide means for modifying the system easily when new or changed requirements come to light. Provide user reviews and approvals at the appropriate intermediate points during a project, he says.

As we see it, there are several implicit assumptions in Mills' approach that need to be considered. One assumption is that a large system can be effectively partitioned into sub-systems before the user requirements for the whole system have been analyzed in detail. As we discussed in our July 1977 issue, some people in the field feel that this study of the complete requirements may be necessary before partitioning can be done for very large, complex systems.

Another implicit assumption is that a complete sub-system be implemented and modified to meet user requirements by means of iterative refinement. For instance, if an interactive order-shipping-billing system is being considered, the Mills' approach might be to divide the project into three (or more) sub-projects, such as order processing, shipping processing, and billing processing. The order processing might be chosen for the first sub-project. It would be developed so as to meet the full range of user needs for handling orders, by being able to modify the system as new or changed needs come to light.

As we discussed last month, Basili and Turner (Reference 6d) propose another approach to the question of sub-system development that we feel has merit. They call their approach "iterative enhancement." Some useful sub-set of the project is chosen and is implemented first. After it is working, it is gradually enhanced by adding further functions until finally the whole project is completed. In our example of the order processing sub-project, perhaps the handling of regular, routine orders would be selected for the first implementation. Once it was working, then the other types of orders would be gradually added—rush orders, order cancellations, order adjustments, and so on.

These are just some of the more important staff behavior patterns that software engineering management is attempting to identify and to develop disciplines for. They include means for introducing the new methodologies as standard practices in an effective manner, and for controlling the all-too-frequent tendency to let system plans get grandiose.

Let us now consider some interesting "behavior" patterns of software development projects, and see what software engineering management has to offer.

Project behavior patterns

In a sense, this topic is misnamed. It is the people on a software project that exhibit behavior; "project behavior" is then a consequence of the people behavior. Still, there are characteristic patterns of behavior that occur from project to project and from organization to organization. It is these characteristic patterns that we are discussing.

R. Caudill (Reference 8c) points out two main problems with software development projects. For one thing, there is generally a lack of management visibility and control. It is not possible for management to "see" all of the parts, as would be possible, say, in a building construction project. So management cannot visually tell how a software project is progressing. Secondly, software projects are often subjected to pressures from higher authority. These pressures are generally in the form of demands to complete the projects sooner, or to do more within the original schedule. Project management is then faced with the question of what can in fact be cut out of the projects in order to meet these demands.

Caudill argues that the concepts and principles of "configuration management" be used for controlling software development projects, as a solution to these problems. The discipline of configuration management has been used successfully in other high technology fields and can be used with software development, he feels.

Caudill describes the five major principles of configuration management. One, a phased approach is taken for each project. That is, a standard set of phases is defined, through which each project must progress. Second, the work products ("deliverables") of each phase are carefully defined. Third, both technical and management reviews are used throughout the development cycle. These reviews might occur at the end of requirements determination, system design, detailed design, and system test design phases, for instance. Fourth, "baselines" are set after the work products of each phase have been completed and reviewed. These baselines represent a freezing of design; after a freeze has been instituted, all subsequent changes must go through a change control procedure. Finally, project scaling is used, for adjusting to the size of each project. For instance, the number of reviews used for a small project would be less than the number required for a large project.

We discussed progress in the use of project management systems in our December 1977 issue. Modern project management systems follow the principles of configuration management outlined by Caudill. The point to be made is that these principles have been used successfully for managing other types of high technology projects that have many points of similarity with software development projects. So data processing management should not feel that software development projects are "unique," from a management control standpoint.

In the planning meeting for a software engineering handbook (reported in Reference 7), a number of the participants felt that this whole concept of project management, as just described, was at least as important as any other aspect of software engineering.

The man/month trade-off

It is widely believed that the completion of a complex software project can be speeded up by adding more people to the project. In fact, this can be an erroneous assumption, as pointed out by Frederick Brooks, Jr.

Brooks, in his collection of essays on software engineering (*The Mythical Man-month,* Reference 9), states what he calls "Brooks' Law": *"Adding manpower to a late software project makes it later."* His essay on this subject challenges the assumed interchangeability of people and months on a large software project. Some tasks are essentially sequential in nature, he says, so that they are conducted at the speed at which one person can accomplish them. No matter how many people might be assigned to the project, it still takes nine months to create a baby, he points out. The other characteristic of large software projects is that they involve complex relationships and inter-communication. The more people on a project, the greater the complexity of relationships and the more inter-communication that is needed. Both of these factors consume time and tend to delay the projects.

Gordon and Lamb (Reference 10) feel that while Brooks' Law is for the most part accurate, it is still an over-simplification. To analyze where the law might and might not apply, they constructed a simple model of software worker productivity.

The model has three main elements. The first element is the learning curve, the well-known exponential rate at which a person learning a new job "comes up to speed." Based on learning curve experience (and many studies of this have been made by industrial engineers), they say that about one-third of a person's time is "lost" due to learning, during the learning period.

To illustrate their analysis, they consider a project that requires one productive man-year, or 2000 man-hours. If one person is assigned to do the project and it takes that person six months to learn the project (that is, in six months the person has reached the point where he spends only 5% of his time learning more about the project), then two man-months of time will be lost. It will take 14 months to complete the project, not 12. And if the person were to leave the project at the end of six months and another person were assigned to take over, the process would be repeated and the project would now require 16 months to complete.

The second element in their model is teaching. If one or more additional people are added to a project, the original project member(s) must spend time teaching the new people about the project. Gordon and Lamb assume that the time that is lost in teaching is the inverse of learning. So if a second person were added to the above project, when it was apparent that it was falling behind schedule, about two months of the original person's time might be lost in teaching.

The third element in their model is communicating and coordinating the efforts of multiple team members. They assume that the time loss here is a logarithmic function of team size.

Now consider a four man-year (8000 productive man-hour) project, they say. Assume that four people are assigned to the project, and that progress is to be checked every three months. Learning plus communication and coordination will cause the project to fall behind schedule, which fact is detected at the first check point. Seeing that the project is falling behind, management decides to add one more person. Teaching, learning, and communicating losses increase, and the project falls further behind schedule. If this same management decision (to add one more person) is made at each of the next two check points, what happens? At the end of 12 months, 11,000 man-hours will have been expended on the project, but only 7,500 *productive* man-hours will have been realized. The project will still not have been completed and it will already be almost 40% over budget.

What can be done? The solution, as Gordon and Lamb see it, is for management to add *more* people to the project than might be expected to be needed, and as early as possible. In other words, over-staff the project at the first sign of trouble. But this takes courage on the part of data processing management (to request it) and unusual understanding on the part of higher levels of management (to approve it).

We think Brooks plus Gordon and Lamb have enriched the field's understanding of a common project behavior problem area. Perhaps a better solution is not the one suggested by Gordon and Lamb but rather to allow for the loss factors more realistically in the original project time and cost estimates and then staff accordingly. But whatever solution is chosen, here is a characteristic of projects that data processing management *must* be aware of.

In short, it is not valid to assume that people and months can be interchanged on software projects of reasonable size and complexity.

System behavior patterns

This subject area perhaps is not quite so likely to be misnamed as in the case of project behavior patterns. True, it is through human activities that systems are created and operated. But somehow, systems seem to take on characters and personalities of their own. Behavior patterns for systems do not seem too incongruous.

Last month we quoted at some length a paper by Mills, presented at the Second International Conference on Software Engineering, held in October 1976, and published in Reference 6b. At the risk of duplication, we will quote from this paper again because it sets the stage for the discussion of system behavior patterns.

The maintenance of software systems, says Mills, is consuming altogether too many data processing resources. Some 75% of programming personnel are already concerned with maintenance; unless significant changes are made, this percentage will rise even higher and at the expense of development.

There are two main reasons for this dominance of maintenance activities, says Mills. For one thing, application systems are maintained for an indefinite period of time after they have been developed. So a fraction of the development staff must be shifted over to perform maintenance. If 20% of the development staff is shifted to maintenance every two years, then after 12 years maintenance will represent about 74% of the effort—which is about what the effort is, he feels. The only stable point is when 100% of the effort is on maintenance.

The other reason for the dominance of maintenance is that it has proved to be much more difficult to develop good systems than has been commonly supposed, where "good" means both correct and capable. A part of the work force is needed to perform corrective maintenance, to remove errors that were built into the systems by mistake. And another part of the work force is needed to perform adaptive maintenance, for enhancing the system. As adaptive maintenance continues, the organization may eventually lose sight of the original system.

In order to contain this growth of maintenance, Mills argues for better development methods. Better development methods (for which he describes his ideas and which we discussed last month) will greatly reduce the need for corrective maintenance. Fewer errors will be built into the systems. Also, better development methods will foresee some of the likely enhancements and will allow for them. So adaptive maintenance will be easier.

As we see it, Mills is right in arguing that good development methods will lead to greatly reduced corrective maintenance. And it is also possible that such methods will make adaptive maintenance *easier*, but they certainly will not eliminate the need for adaptive maintenance. There are many reasons why application systems go through a long series of enhancements. An organization can accept only so much change in its procedures at one time, otherwise the upheaval factor is too great. So it may be necessary to install a system that performs only the most essential functions and then gradually enhance it. Further, it is hard for users to conceive of what they may want a few years hence until they have grasped the power of the new system. Business conditions will change, requiring changes in the application systems. New technology will become available which can provide new, desirable opportunities; the new distributed systems might be a case in point here. Also, new hardware and operating system configurations will become available but will require the existing application systems to be modified in order to exploit their capabilities. For these and similar reasons, we believe that adaptive maintenance will continue to be needed.

The point we are making here is that software engineering methods should prove to be very helpful in greatly reducing corrective maintenance. And hopefully they will make adaptive maintenance easier to perform. But software systems will continue to go through almost continual enhancements, for reasons such as those just mentioned. This is one aspect of what we mean by system behavior patterns—the typical life cycle of software systems.

E. B. Daly (Reference 6c) reports on a study that compares hardware development and maintenance with that of software. The data for the study came from three large projects plus numerous smaller ones conducted at the GTE Automatic Electric Laboratories. Here are some of the findings that he reports.

Software development is slower. Hardware logic gates, as measures of complexity, are comparable to branching statements in programs, he says. Comparing hardware development projects to software development projects of about the same complexity, he found that it took about twice as many man-hours to develop one instruction as it did to develop one logic gate. Further, it took *four* times as many man-hours to do design maintenance (corrective maintenance) for software as it did for hardware.

The reasons for these differences, he feels, are the following. The management techniques and the development processes for hardware are more advanced than those used on software projects. The people involved (generally, the engineers) are more experienced than their counterparts in software development. More "building blocks" are available for hardware and they perform a wider variety of functions. Finally, the hardware gets tested twice—once on its own and once when the software is being tested on it.

One might conclude from these comments that hardware development and design maintenance represents a level of achievement which software development and maintenance should seek to equal. This equality might come from the use of a software engineering discipline that roughly matches the capability of the hardware engineering discipline. But the software discipline is currently far behind the hardware discipline.

Hard to compare software projects. Daly apparently found it harder to compare two software development projects than to compare two hardware development projects. At least, he felt that it was very hard to compare two software projects because of differences in about 15 factors (which he lists) that affect the software development rate.

For the development of "small" real-time systems (involving from 5,000 to 20,000 instructions), he found that production rates varied from 1.6 to about 5 instructions per man-hour. For large real-time systems (involving over 75,000 instructions), average production rates were 1.9 instructions per hour for data manipulation instructions and 0.24 instructions per hour for diagnostic instructions.

Two main points stand out from Daly's figures, in our mind. One is that comparable variations in production rates probably would not be tolerated in hardware development and maintenance, for systems of similar complexity. The second point is that Daly's figures confirm what many have been recognizing, which is that there are substantial differences in development rates between large and small systems. This is not a well understood phenomenon, as far as we know. What are the measures of complexity, and where does complexity begin to increase in a non-linear (exponential?) fashion? If data processing management had better answers to these questions, it would be possible to do a much better job of estimating development times and costs.

The development/maintenance time ratio. Daly presents some interesting figures on the man-hours required to both develop and maintain large real-time systems over a four year time period. By the end of the fourth year, some 54% of the total man-hours had been spent on development and the other 46% were spent on design (essentially corrective) maintenance. Daly says that "local support," which might be another form of corrective maintenance, was not included in these figures. The 54% development time was broken down as follows: developing specifications, 13%; building, 20%; testing, 13%; and documentation, 8%.

After the fourth year, design maintenance increases very slowly, says Daly. At the end of the tenth year, it might represent about 60% of the total man-hours expended. One wonders, of course, if the better development methods advocated by Mills would substantially reduce these maintenance figures.

It is interesting to compare Daly's figures with somewhat similar figures presented by Barry Boehm at the Second International Conference on Software Engineering, mentioned earlier. Boehm reported on studies made by TRW Systems on large software systems, mostly for military applications. Over a good portion of the life cycle of a software system, about 30% of the man-hours were spent on designing, building, and testing the systems, and the other 70% of the time was spent on maintenance. Boehm did not separate corrective and adaptive maintenance, and it seems likely that his figures lumped these two types of maintenance. Daly, of course, did not include adaptive maintenance and perhaps not all of the corrective maintenance.

What conclusions might one draw, then, based on what Mills, Daly, and Boehm have said? It seems to us that one can conclude that corrective maintenance will require about 50% to 60% of the total man-hours over a seven to ten year life of a complex application system (say, 75,000 or more instructions). For small interactive systems, probably the relative time spent on corrective maintenance will be less; these systems are less complex and hence fewer errors are likely to occur during construction.

One might also conclude that when adaptive maintenance is added to the corrective maintenance, it seems quite likely that at least 75% of the total man-hours are being spent on some form of maintenance. As Mills points out, this leaves precious little time for developing new systems.

But the problem does not end there. Software systems tend to lose structure and to grow to a point where "fission" occurs, over their life cycle. To see why this is so, let us consider program evolution.

Program evolution

Belady, Lehman, and Parr have studied the dynamics of program evolution and have reported their findings in References 11 and 12. The original study was concerned with what happened to IBM's os/360 operating system after its initial release. After finding what they believed might be patterns of evolution, they investigated two other operating systems in two quite different environments. The same general patterns were observed.

The authors feel that they may well have uncovered general behavior patterns for the combination of organizations, people, and the computer programs themselves. So they feel they can present some general conclusions on program evolution.

Continuing change. All three operating systems grew in size long after their initial development. This growth was due to correcting faults, improving performance, adding new functions, and supporting new hardware.

Breadth of change. The percent of each operating system that was changed increased almost linearly with time. Eventually, all modules were changed at least once.

Loss of structure. The authors found that repeatedly modifying a system without redesign tends to obscure its conceptual structure. So, over a period of time, the original structure of these three operating systems gradually was lost. Perhaps the "proof of correctness" to program development (which we discussed last month) might constrain this tendency.

Smooth growth in size. The growth in size seems to be statistically smooth, say the authors. Size increases rapidly at first, then gradually slows down—although the maintenance effort may remain quite constant.

Growth in release intervals. The time between two successive releases of each operating system tended to increase exponentially. The most likely explanation of this, said the authors, was that it became increasingly difficult to make changes. Even though maintenance effort remained about constant, this behavior resulted in a decelerated growth in size.

Decrease in work rate. Closely related to the above behavior was the tapering off in the number of changes made, due to the fact that changes became harder and harder to make. With a constant maintenance effort, this meant a decrease in the effective work rate.

Maximum change per release. There seems to be a practical maximum limit to the growth in program size per release. With OS/360, this limit was about 400 modules added to the system. If a release had more than this amount, it led to significant problems such as performance errors. Moreover, if this limit was even approached, the next releases turned out to be much smaller, perhaps to allow a catch up in correcting faults and completing the documentation.

Need for measure of complexity. Some measure of system complexity would be useful, say the authors, to indicate to management when a complete structural redesign becomes highly desirable. As one (only approximate) measure of complexity, the authors used the cumulative percentage of modules that had been handled (changed or added). They later revised this to be the cumulative *handlings*, where each of two changes to the same module would be counted. But they see the shortcomings of each measure. What is really desired, they say, is a measure of the degree of concentration of the changes. The more scattered the changes through the system, the more the conceptual structure will be obscured and the more error prone will be the changes. Eventually, the changes may well make the system so complex that a complete redesign will be needed. It may even be necessary to subdivide the system into two systems ("fission") in order to make it workable.

We think it is fair to say, then, that software systems exhibit a behavior. They tend to grow in size and complexity, as faults are corrected and new functions added.

Conclusion

As we have attempted to point out in these two reports, software engineering seeks to impose a discipline on the development and modification of software. A wide variety of tools and techniques have been developed, unfortunately with very spotty coverage of the life cycle of a software system.

Within the development phase, most of the work to date has been on developing tools and techniques for construction, quality assurance, and evaluation. Much less work has been done on determining requirements, performing design, and performing maintenance.

As the discussion just completed indicates, software engineering should address the *whole* life cycle of a software system—say, for instance, for ten years of life. Software systems tend to grow in size and complexity, as faults are corrected and as new functions are added. Structure is lost, maintenance becomes harder and more costly, and the risk of unreliable operation is increased. These are problem areas that simply *must* be addressed by software engineering.

In short, while much has been accomplished in software engineering, there is still much more that needs to be done.

REFERENCES

1. For more information on Kenneth Kolence's ideas on software physics and their application to software engineering, plus a free software physics primer, write to the Institute for Software Engineering, P.O. Box 637, Palo Alto, Calif. 94302.
2. Gilb, Tom, *Software Metrics,* Winthrop Publishers, Inc. (17 Dunster Street, Cambridge, Mass. 02138), 1977.
3. *Performance Evaluation Review,* ACM Special Interest Group on Measurement and Evaluation (ACM, 1133 Avenue of the Americas, New York, N.Y. 10036), membership $12 per year for non-ACM members; published quarterly.
4. *EDP Performance Review,* Applied Computer Research (8808 North Central Avenue, Phoenix, Ariz. 85020); price $48 per year; published monthly.
5. Sreenivasan, K., "Application of accounting data in evaluating computer system performance." *Software Practice and Experience* (John Wiley Sons, Ltd., Baffins Lane, Chichester, Sussex, U.K.); April-June 1976; p. 239-244; U.S. price $75 per year; published bi-monthly.
6. *IEEE Transactions on Software Engineering,* IEEE Computer Society (5855 Naples Plaza, Long Beach, Calif. 90803), individual copies $10:
   a) Elshoff, J. L., "An analysis of some commercial PL/1 programs," June 1976, p. 113-120.
   b) Mills, H. D., "Software development," December 1976, p. 265-273.
   c) Daly, E. B., "Management of software development," May 1977, p. 230-242.
   d) Basili, V. R. and A. J. Turner, "Iterative enhancement: A practical technique for software development," December 1975, p. 390-396.
   e) Elshoff, J. L., "The influence of structured programming on PL/1 program profiles," September 1977, p. 364-368.
7. U.S. national Bureau of Standards, *Report on planning session on software engineering handbook,* Tech Note 832, Nov. 1974; order from Superintendent of Documents, U.S. Printing Office, Washington, D.C. 20402; SD Cat. No. C13. 46:832; price 70 cents.
8. *Proceedings of 1977 National Computer Conference,* AFIPS Press (210 Summit Avenue, Montvale, N.j. 07645), price $60, microfiche $15:
   a) Yourdon, E., "The choice of new software development methodologies for software development projects," p. 261-266.
   b) Freedman, D., D. C. Gause, and G. M. Weinberg, "Organizing and training for a new software development project," p. 255-260.
   c) Caudill, R., "Understanding the developmental life cycle," p. 269-276.
9. Brooks, F. P. Jr., *The mythical man-month,* Addison-Wesley Publishing Co. (Jacob Way, Reading, Mass. 01867), 1975; price $5.95.
10. Gordon, R. L. and J. C. Lamb, "A close look at Brooks' law," *Datamation* (1801 S. La Cienega Blvd., Los Angeles, Calif. 90035) June 1977, p. 81-83, 86; price $3.
11. Belady, L. A. and M. M. Lehman, "A model of large program development," *IBM Systems Journal* (IBM, Armonk, N.Y. 10504), No. 3, 1976; p. 225-252; price 50 cents.
12. Lehman, M. M. and F. N. Parr, "Program evolution and its impact on software engineering," *Proceedings of 2nd International Conference on Software Engineering, October 1976;* order from IEEE Computer Society (address above), price $20.

You may have been seeing articles recently in the trade press about possible government regulation of "trans-border data flows." If you are concerned almost wholly with data processing within one country, it may appear that the possible regulations will have little effect on your operations. (Data processing executives in multi-national companies are very concerned about the possible regulations.) It is quite possible, though, that the regulations will have impacts on strictly domestic data processing. Next month, we discuss what is happening in this emerging subject area.