MAIDS STUDY, TMDE TASK 05 - INFORMATION DYNAMICS COMPUTER SOFTWARE: PROBLEMS AND POSSIBLE SOLUTIONS

FRANKFORD ARSENAL

APRIL 1973

AD

MEMORANDUM REPORT M73-13-1

MAIDS STUDY
TMDE TASK 05 - INFORMATION DYNAMICS

Computer Software: Problems and Possible Solutions

by

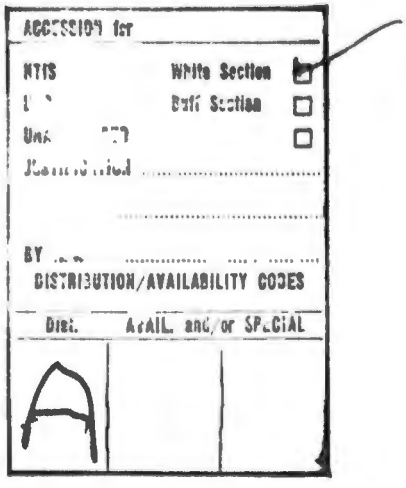
Dr. EDWARD LIEBLEIN
Chief, Computer Software Team
Communications/ADP Laboratory
U.S. Army Electronics Command
Fort Monmouth, NJ
(Customer's Order M1-2-50028-M1F6)

April 1973

DEPARTMENT OF THE ARMY
FRANKFORD ARSENAL
Philadelphia, Pa. 19137

DISPOSITION INSTRUCTIONS

Destroy this report when it is no longer required. Do not return it to the originator.



The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*

Frankford Arsenal,
Philadelphia, PA 19137

2a. REPORT SECURITY CLASSIFICATION

Unclassified

2b. GROUP

N/A

3. REPORT TITLE

MAIDS STUDY, TMDE TASK 05 - INFORMATION DYNAMICS.
Computer Software: Problems and Possible Solutions

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Technical Research report

5. AUTHOR(S) *(First name, middle initial, last name)*

Dr. Edward Lieblein (Communications/ADP Laboratory, U.S.Army Electronics Command)

6. REPORT DATE

April 1973

7a. TOTAL NO. OF PAGES

~~28~~ 33

7b. NO. OF REFS

28

8a. CONTRACT OR GRANT NO

AMCMS Code 562H.12.J2900.02

b. PROJECT NO.

DA Project 1J662601AJ29

c.

Customer's Order M1-2-50028-M1F6

d.

9a. ORIGINATOR'S REPORT NUMBER(S)

Frankford Arsenal Memorandum Report
M73-13-1

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY

Frankford Arsenal SMUFA-N7200

13. ABSTRACT

Software systems fail when delivered, during acceptance testing, and throughout the life of the systems. The reasons for their failure are explored and means to prevent or overcome these problems are suggested. The latter include development and use of good documentation standards; use of improved software design techniques (described as "defensive programming") and of good high level languages; design of adequate tests concurrent with the software design process; and adequate preparation for hardware/software integration.

DD FORM 1 NOV 68 1473 REPLACES DD FORM 1473, 1 JAN 64, WHICH IS OBSOLETE FOR ARMY USE.

UNCLASSIFIED

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Computer Software Software Systems Software Design Techniques Programming Software Testing and Validation						

MEMORANDUM REPORT M73-13-1

MAIDS STUDY

TMDE TASK 05 - INFORMATION DYNAMICS

Computer Software: Problems and Possible Solutions

by

Dr. EDWARD LIEBLEIN
Chief, Computer Software Team
Communications/ADP Laboratory
U.S. Army Electronics Command
Fort Monmouth, NJ
(Customer's Order M1-2-50028-M1F6

AMCMS Code 562H.12.J2900.02
Project 1J662601A J29

Approved for public release; distribution unlimited.

Fire Control Development & Engineering Directorate
FRANKFORD ARSENAL
Philadelphia, PA 19137

April 1973

ABSTRACT

Software systems fail when delivered, during acceptance testing, and throughout the life of the systems. The reasons for their failure are explored and means to prevent or overcome these problems are suggested. The latter include development and use of good documentation standards; use of improved software design techniques (described as "defensive programming") and of good high level languages; design of adequate tests concurrent with the software design process; and adequate preparation for hardware/software integration.

FOREWORD

Information Dynamics is Task 05 of Multipurpose Automatic Inspection and Diagnostic Equipment and Techniques. The name "Information Dynamics" was assigned to the program to more properly identify the nature and goals of the overall task. While much of the activity in this area is referred to as software, this name has many meanings to different people and does not adequately describe the nature of work to be performed.

The terms "software" has been literally applied to computer programs. However, computer programs have often been described as a plan for the solution of a problem implemented on a digital computer. Programming consists of planning and coding, and includes numerical analysis and other functions necessary for the integration of a computer into a system. Since software also implies the documentation describing the computer program and the languages associated with programming, it can be seen that this term is more all encompassing than the literal interpretation as simply programs for digital computers. The word "software" does not appear in the unabridged Webster's Dictionary; however, if we examine its component words and their definitions, we find a meaning which would be acceptable to buyers and users of software, but not to developers of software. For example:

a. Soft: lacking firmness, strength of character; lacking robust strength, stamina, or endurance.

b. Ware: manufactured articles products of art or craft; goods, commodities, manufactures, or produce of a specific class or kind.

Thus, software can be literally interpreted as meaning products of an art or craft, lacking in strength of character, firmness, and endurance. On the other hand, Information Dynamics can be interpreted as follows:

a. Information: the communication or reception of knowledge or intelligence; facts or figures ready for communication or use as distinguished from those incorporated in a formally organized branch of knowledge.

b. Dynamics: characterized by continuous change, having or relating to nonphysical force or energy, producing an effect of energetic movement or progression.

Thus, Information Dynamics is the communication or reception of knowledge or intelligence, characterized by continuous change, having or relating to nonphysical force or energy, and producing an effect of energetic movement or progression, as applied to computer control, communications, knowledge, or intelligence.

Literally, Information Dynamics is concerned with the technology and mechanisms required for the dynamic interaction of man and machine in any computer-controlled environment. Man generates the intelligence to be

acted upon; the machine interprets the intelligence received, operates on it, and returns its analysis and/or synthesis to man. Thus, Information Dynamics includes all facets of the development of languages understood by both man and machine, their translation to a form that can be most efficiently operated upon by a machine, the control of the sequence of operations the machine must perform, the analysis and synthesis of the information generated by the execution of control programs, and the output to man of the information developed. The computer-generated intelligence may be in many forms, such as numeric, alpha-numeric, graphic, or verbal.

Many papers have been written on the nature of software and software activities covering the gamut encompassed by Information Dynamics. However, most of these papers are written in a language and level of complexity suitable for review and analysis only by a small group of specialists. The intangible nature of computer programs or software and its meaning in a world almost totally oriented toward hardware, has resulted in lack of understanding, fear, disbelief, and indifference. Those who do understand the nature of programming have done nothing to educate the others, thereby contributing to the so-called "mystique" of software.

Dr. Edward Lieblein, Chief of the Computer Software Team, COM/ADP Laboratory, U.S. Army Electronics Command, was requested to prepare a paper covering computer software. This paper was to be written in "layman's terms." The paper is reproduced in its entirety, without modification.

The reader should read the paper, then reread the paper, and try to draw analogies between hardware development and computer software. Many of the problems contributing to the software dilemma would likewise contribute to hardware dilemmas if they were handled in the same way. For example, systems analysis in hardware is considered routine. However, systems analysis from the software viewpoint is a function of the individual assigned the software problem. However, the Army has a very small cadre of people having any experience in the computer software area.

The same factors affecting the successful completion of hardware programs, which require that experienced personnel manage the programs, also apply to the software activity. Yet, inadvertantly or through a lack of understanding of the impact of software, the Army has almost no capability to manage large software programs. Through the Information Dynamics task activity, not only will the technology in the various areas be expanded, but the task will also provide the Army's cadre with expanded software capabilities and the experience necessary for managing software activities.

TABLE OF CONTENTS

List of Illustrations

1. INTRODUCTION

Why does software for major systems usually turn out to be unsatisfactory with respect to cost, delivery, performance, reliability, documentation, maintenance, future modification, and transferability? This paper will explore the problems associated with each of the areas above and suggest a set of possible solutions. Unfortunately, some of the proposed solutions are more easily stated than implemented. Several are technically feasible today but would be difficult to implement as they would be impeded by current practices and traditions in software design and management. Others require the development of new techniques, methods, and standards.

2. PROBLEM AREAS AND POSSIBLE SOLUTIONS

2.1 Cost, Delivery and Performance

In a recent Air Force study of future command and control requirements it was estimated that current Air Force annual software expenditures are in the range of $1 billion to $1.5 billion compared to between $300 million and $400 million for computer hardware (1). The study states that the recent AF WWMCCS procurement involved approximately $50-$100 million for computer hardware and $772 million for software. However, only 30% of the Air Force R&D budget for information processing is dedicated to software.

It is difficult to draw a conclusion on the average ratio of software to hardware costs based on this data alone since the formation of a replacement system may involve the use of some hardware from the system it replaces. In any case, for medium to large-scale tactical software systems based on militarized hardware one should expect that the total software development costs will, at least, equal the hardware costs and may, for very large systems, exceed hardware costs by as much as 700%. To make matters worse, these rather high software costs are quite difficult to estimate. According to David (2, p69), estimates for such software systems are likely to be low by factors of 2.5 to 4.

If one defines a successful project as one which was delivered on time within the allotted funding and satisfied the customer then, according to Aron (3, p52), an IBM study of a variety of large-scale software projects concluded that about one project in 10 or 12 was successful. (It may be safe to assume that this ratio would be worse for less experienced organizations.) The referenced Air Force study indicates that 6-12 month slippages are typical for such systems, and delivery, when finally made, is usually based on serious compromises in software performance.

The trouble is that software design, which will be discussed more fully in the sequel, is a recently developed and much oversold art. As such, its implementation capability usually falls far short of management's aspiration level. Furthermore, there is an inherent complexity in tactical software development that makes its outcome difficult to predict, especially in view of the relative newness of the tasks to be performed. Consequently systems are often slower and require more memory than anticipated. The same degree of difficulty does not usually attend the development of routine business software systems such as cost accounting and payroll systems. Not only have the latter systems been built over and over again but they lack much of the mathematical sophistication and real-time control fundamental to most tactical systems. In fact, today one finds a variety of cost and time estimation techniques (based on familiar task lists) which work quite well for routine business systems (4) especially when combined with PERT/CPM task scheduling systems. For an alternate presentation of a quantitative manpower estimation scheme for large software systems see (5).

2.2 Reliability

Software systems fail. Not only do they fail when delivered, they fail during acceptance testing and continue to fail during the life of the systems. For example, about 1000 errors were found in each new release of IBM's Operating System, OS/360, which was a 5000-man year effort. How is this number related to the actual number of errors in the system? See Hopkins (3,p20) for some possible (albeit unsatisfying) answers. The trouble is that "testing shows the presence, not the absence of bugs." (Dijkstra (3, p21)) Thus, the so called "debugged" program is actually one for which the conditions that make it fail have not yet been found.

Not only do software systems fail, it can be shown that most large-scale monolithic software systems degrade as well. Such degradation is based on the fact that almost every actively used system will be changed frequently during its life. (Many are in a constant state of change.) Due to time and cost constraints, changes most often take the form of "patches" which usually are made with utter disregard of the impact of such patches on the system as a whole (including all previous patches). What happens then is that after a while the interrelationships between the different parts of the program become opaque. For example, it may be almost impossible to determine whether a given section of code is ever executed, or worse, which portions will be affected by a change in any given portion.

The ideal failure rate curve is shown in Figure 1. Figure 2 shows the impact of successive changes on the chance of first run success following each change (1). The numbers are based on 84 scientific batch programs. It is believed that there would be far fewer comparable successes for large-scale real-time tactical systems. This curve implies a degradation rate which can be interpreted as shown in Figure 3 (6). Each spike represents the sharp increase in failures due to a program revision. In time, each successive patch tends to make the system more opaque. Finally the interactions become so complex that further changes (error-corrections or modifications) cause more trouble than they correct. This is the point where degradation starts. A smoothing of the changes turns the ideal curve of Figure 1 into the unfortunate situation shown in Figure 4.

The degradation of software may not be as severe a problem as the appearance of a single bug at the wrong time:

> "Recently, a software error aboard a French meteorological satellite caused it to 'emergency destruct' half of its force of weather balloons instead of interrogating them. Current Air Force software reliability problems indicate that similar software errors could cause the Air Force to lose critical command post or satellite capabilities in a strategic crisis. --- The Air Force implicityly provides a guarantee to the nation that there are no errors in its command and control software that might escalate a crisis situation or seriously degrade performance during a crisis. Current software technology does not provide the highest possible confidence to back up that guarantee." (1)

Also, in this connection David and Fraser report:

> "There is a widening gap between ambitions and achievements in software engineering. This gap appears in several dimensions: between promises to users and performance by software, between what seems to be ultimately possible and what is achievable now and between estimates of software costs and expenditures. This gap is arising at a time when the consequences of software failures in all its aspects are becoming increasingly serious. Particularly alarming is the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death, not only for individuals, but also for vehicles carrying hundreds of people, and ultimately for nations as well." (2, p120)

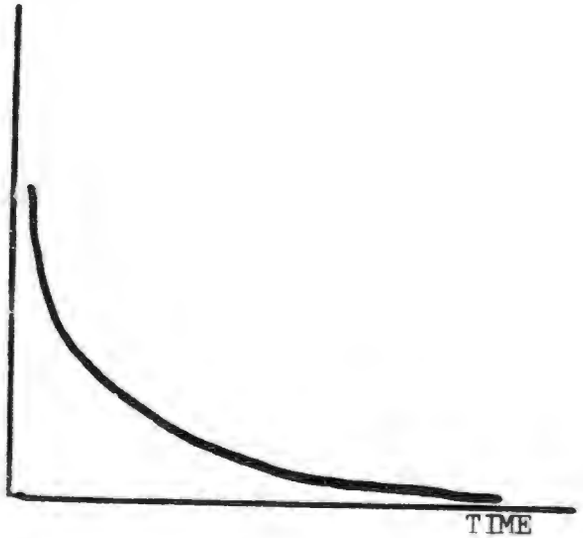


Figure 1. Ideal Failure Rate

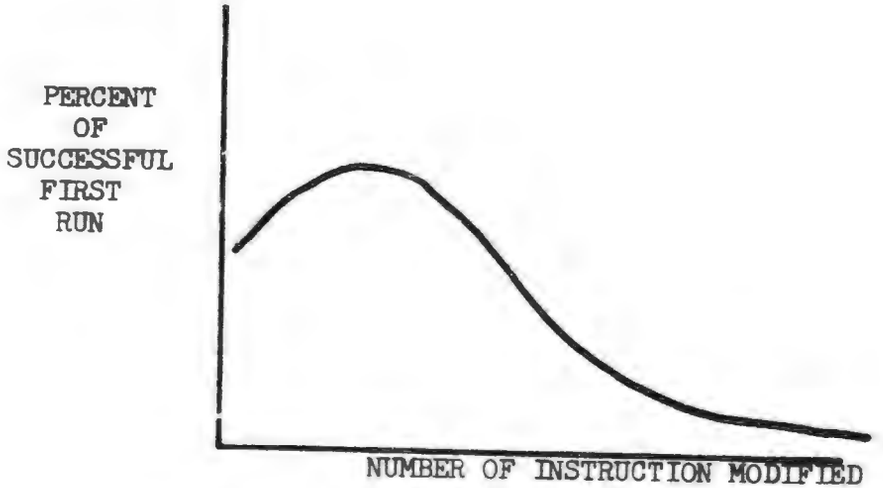


Figure 2. Effect of Excessive Instruction Modifications

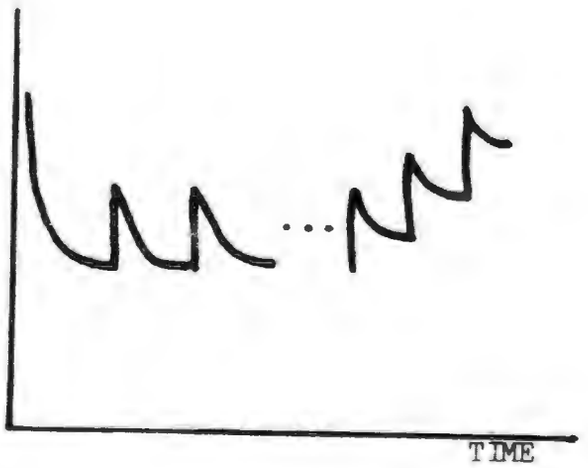


Figure 3. Degradation of Performance

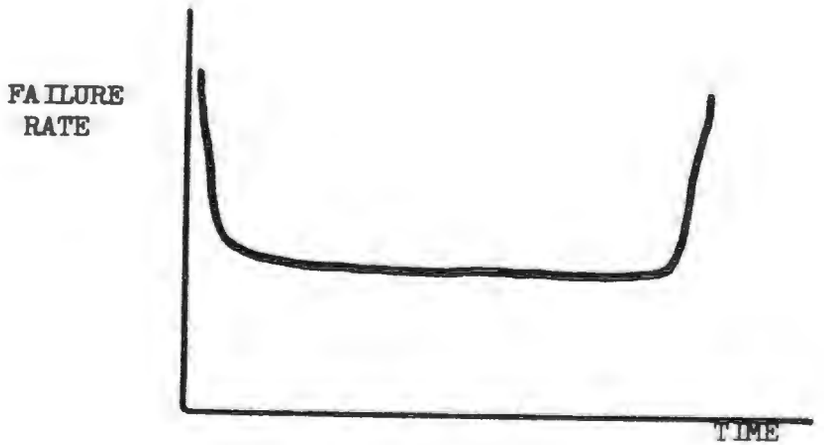


Figure 4. Smoothed Actual Failure Rate

Harkins, speaking from the Navy side of the house finds:

> "Software performance is never critically analyzed, and is never integrated into total system performance evaluation. Indeed, most analyses of weapon system effectiveness we have seen implicitly and often explicitly assume perfect performance of the software." (7)

There is virtually no way to improve the reliability of a poorly operating program, short of total redesign. In fact, the initial technical and managerial approaches to design, testing, documentation, personnel, and project organization are precisely the factors that have the strongest influence on ultimate reliability. Suggested approaches to these areas are given in the sequel.

2.3 Documentation

Software documentation is usually inadequate principally due to failure of the Government to specify precisely the nature and extent of such documentation. The current paucity of documentation standards is to blame. The generation of good documentation is a rather time consuming process which is considered by most programmers to be a dull chore at best. Documentation is one of the first areas to be relaxed when funds and time get tight because the "important" thing is to deliver something that works (at least when delivered). In most cases, inadequate documentation places the Government at the mercy of the developing contractor. Good documentation is more difficult to achieve in assembly language programs than in higher level language programs, however the quality of documentation of the latter, more often than not, is quite disappointing. It is widely held that adequate documentation requires 30%-40% of the total development effort and should be created continuously throughout all phases of the project. The following comment by Nash is appropriate:

> "I would like to report on documentation of the F-level PL/I compiler, where we had a team of about two dozen people on the actual development. We did not have any private memos, or notes, although we had a considerable amount of verbal communication. What we did establish was a book which described in complete detail every part of the compiler. All members of the team were obliged to describe their parts of the compiler by means of flow-diagrams and English prose in complete detail at design time. The book grew very large--eventually to about 4000 pages. It was a lot of work to maintain it--perhaps about 10-20% of total effort--but it was worth it." (2, p93)

(Note that the 10-20% figure does not include source-code documentation.)

The key to successful documentation is an adequately commented source listing. It cannot be emphasized strongly enough that this commenting is rarely successful when done as an afterthought. It should be considered as an active part of program generation, as important as the source statements themselves. Documentation should not be considered as something generated now for use later. On the contrary, it is an essential management tool for successful project control. Some have advocated the use of an on-line documentation system where each member of the project team has instant access to the latest version of his own and his colleagues' work (3, pp 53-60).

It is unfortunate that a current measure of software productivity is the number of machine instructions generated per man-day. These instructions are produced during the programming phase of the software development cycle which is often less than 15% of the total project effort. The number of instructions generated during this short period divided by the total project time produces the number of instructions per man-day. For a large-scale system, this figure would be low even if all programming were done in zero time. However, great efforts are made to increase this number, which is used as the "measure of goodness" of a programmer. Of course he can increase his instruction rate by decreasing his documentation efforts. In his haste, he may generate an inefficient program. If he is using a higher level source language he might use constructs which would increase the number of source statements or the number of instructions or both (8).

2.4 Post-Design (Maintenance and Future Modification)

After a system has been delivered, four types of software design are usually required:

Type 1: Correction of heretofore latent bugs.

Type 2: Enhancement of the existing program (perhaps to make time/space available for future changes, to increase accuracies, etc.).

Type 3: Expansion of the program to accommodate minor changes in the mission often involving changes in function, procedures or equipment.

Type 4: Major redesign of the system due to (1) an extensive equipment replacement, (2) a major shift in the mission, or (3) severe degradation of performance as a result of extensive patching.

Each of the first three types have been called "maintenance," "enhancement," "redesign," "modification," etc., however, all involve a continuation of the design process. In this respect use of the term "maintenance" is unfortunate since it implies a process analogous to hardware maintenance, which it isn't. In hardware maintenance, faulty components are replaced but the design remains unchanged. In software maintenance, the detection of a bug is the detection of a design error. For correction, redesign is required. The term "post-design" is suggested here as being better suited.

It was once held that software could be delivered fault-free and that expansion, where necessary at all, would involve very low costs in relation to the cost of the original development. The sad fact today is that, for large systems, life cycle post-design costs usually meet or exceed development costs (7). Life cycle costing of software is essential, yet is usually ignored in budgets. Software is expensive to change due to poor practices in design, testing and documentation. An unstructured, poorly-documented program is going to be difficult to understand and, consequently, difficult to change. The testing of each change must be comprehensive. It must take into account every relationship between the change and the parts with which the change is logically associated. With poor documentation, associations are usually very difficult to detect. With regard to type 3 changes d'Agapeyeff concludes:

> "In large on-line systems, the cost of testing a change is almost 100 times as much as the cost of producing the change. We cannot afford to go through this process too often." (2, p72)

Most experience now dictates that type 1 post-design is inevitable. On the other hand, the mere suggestion of the possibility of type 4 post-design to programmers or programming managers is usually met with disbelief. Yet, for example, we have:

> "95% of the 465L software delivered to SAC had to be reqritten to meet SAC's operational needs. --- 67% of the Seek Data II software delivered in Viet Nam had to be rewritten." (1)

The software design approach should be based on the assumption that each portion of the system will be involved in post-design of each of the four types (correction, enhancement, expansion and major redesign). The discussions concerning design practices in the sequel recommend a variety of approaches and techniques applicable to the above philosophy.

2.5 Transferability

Transferability (often called "portability" and "transportability") involves the moving of software from one computer (i.e., "environment") to another. This should not be confused with "adaptability" of software, which is concerned with the ease of expansion or change of software with respect to the current system of equipments. "Adaptability" involves type 3 while "transferability" involves type 4 post-design. Every system that will be operational for a long period of time will eventually be faced with the transferability question since there will be compelling reasons to decide whether to take advantage of the state-of-the-art hardware advances or to live with obsolescent equipment which, by then, will be increasingly difficult to maintain. The software costs associated with this type of transfer may be extremely high. For example, conversion of SAC's software to the new WWMCCS computer required 600 man-years of effort (1).

The transferability problem is usually severe where the software has been written in assembly language. The situation is eased somewhat if (1) the software was written in common higher level language, (2) documentation is of a high quality, (3) techniques of modularity and structured programming are employed, and (4) hard-to-understand short cuts are avoided. In regard to the last item, the following comments by Strachey on one of programming's major ills are pertinent:

> "I want to come back to this business of how can we help people to make systems, write systems, in such a way that they can be transportable. It's not entirely a question of not putting in features that we haven't though of -- not putting in the right number of registers or something like that. I think one of the chief difficulties is that the general standard of programming is extremely low. Now this is a thing which I know Dijkstra's been talking about and this is actually what Doug Ross was saying. I think I would like to suggest again, that the general standards of programming and the way in which people are taught to program is abominable. They are over and over again taught to make puns: to do shifts instead of multiplying when they mean multiplying; to multiply when they mean shifts; to confuse bit patterns and numbers and generally to say one thing when they actually mean something quite different. Now this is the standard way of writing a program and they seem to take great pleasure in doing so -- "isn't it wonderful? It saves a quarter of a microsecond somewhere every other month." Now I think we will not get a proper standard of programming, we will not make it possible to have a subject

> of software engineering until we can have some proper professional standards about how to write programs: and this has to be done by teaching people right at the beginning how to write programs properly. I don't know that we know how to do this yet but I think we should make a very serious effort to do so ..." (3, p52)

2.6 Software Engineering

The field of electronic engineering including computer design has been with us for some time now, yet the design of large-scale software system is a relatively new technology. In the former, designers have from four to eight years of formal professional traning while in the latter it is frequently believed that a few years of trade school training is sufficient. As a consequence, for all systems except perhaps business-type applications, the software design field is barely limping by. This is believed to be one of the more serious reasons for the ills associated with the development of large-scale tactical systems. Recently, the phrase "software engineering" has been advocated since it implies "the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering." (2, p13) The generation of software for large-scale systems is a complex engineering job that, from the viewpoint of the logical functions implemented and the levels of control and interaction, is much more complex than the hardware on which the system will run; e.g., consider the amount of computer logic required to hard-wire implement a large-scale operating system. What we have then is the more difficult part of the system being designed and built by less capable people, many of whom work at the clerical level. It has been shown that projects that used large numbers of programmers could have been completed with far fewer personnel of higher caliber, where such personnel used a variety of software engineering techniques including structured programming. In one case it was shown that 20% of the original work force would have been adequate (3, p50). Also in this regard:

> "One of the managers of the SAGE program development effort was once asked to recount the experience with the development of that system. He described the history that led to this being one of the largest programming efforts of all time. The early effort was based on experience with a prototype system at MIT. Based on this effort, the SAGE program should have required a reasonable number of people

and time. But it soon became apparent that the actual system required substantially - that is, orders of magnitude - more effort than the prototype. Documenters, testers, operational programmers, utility programmers, designers, table experts and interface specialists were needed. All of these required managers at a variety of levels. The managers required help of both administrative and technical nature. As schedules tended to slip, or difficulties were recognized, more people were hired. This, of course, required more management and communication. This cycle continued for several years until many hundreds of people were involved in the programming effort. The program, although considered by most people to be a landmark as well as one of the relatively few successes in large-scale system programming, was delivered later than originally planned and with somewhat less capability than that originally desired. When asked what he would do differently if he had to do a system like this again, the manager, after some period of reflection, said he would hire twelve good people to do the whole job." (9)

In the development of hardware we recognize the need for a series of engineering steps and possibly some research prior to production of the final product. We build prototype models, large portions of which are often redesigned following thorough engineering tests. The software design process, on the other hand, does not benefit from such a cycle. Software is developed in one step and when it fails or performs poorly we wonder what went wrong. Why do we think we can successfully develop such a complex structure in one sitting? Obviously what happens is that the missing steps are completed after delivery, to the unhappiness of all concerned. Is this one of the reasons that post-design costs exceed those of initial development?

Now we shall turn to the design process itself. For the most part the process is ad hoc with little aid from formal theory or proven methodology. (Some recent developments in this area are associated with program-correctness-proofs and structured programming.) Even though we lack a complete understanding of the software design process, we do have some ideas about what constitutes a good software design. Software should be designed based on a concept of simplicity, e.g., the program should be easily understood. In this connection, a good measure of the quality of a program that is still in the design phase is the ease with which it can be transferred to another designer.

There have been recent discussions concerning the effect of the "go to" on the quality of software design (10) (11) (12). Dijkstra (13) claims that the quality of a programmer is inversely proportional to the density of "go to's" in his program. The problem is that free use of the unconditional transfer does a remarkable job of obscuring the logical structure of a program. (The terms "rat's nest" and "bowl of spaghetti" have been used to describe the control flow of such programs.)

Some of the approaches discussed above and in previous sections may be best described as "defensive programming," which is meant to imply that each program should be written as if it were to be attacked (before and after design completion) by another programmer. Defensive programming forces the designer to (1) generate adequate documentation along with the design, (2) write high quality, simple, clear, concise and efficient statements, (3) insert adequate tests, (4) avoid unnecessary "go to's," (5) consider ease of transferability and (6) plan for high reliability and post-design.

2.7 High Level Languages

The design process is off on the wrong foot if a good high level language is not used. A "good" high level language is essentially one that allows easy expression of the desired data structures and associated operations and has adequate constructs for (1) grouping, (2) nesting, (3) control structures for iteration, (4) if-then-else statements and (5) procedure calls and returns. Another measure is its usefulness in structured programming, which will be discussed in section 2.9. There is a broad spectrum of programming languages from machine languages to low level assembly languages to assembly languages with extensive macro-capabilities to a rather wide range of high level languages.

The use of a good high level language will (1) shorten development time, (2) reduce development costs, (3) improve readability, (4) increase flexibility, (5) permit self-documentation, (6) increase reliability and error detection, (7) facilitate all phases of post-design, (8) permit transfer of the software to new machines (this is virtually impossible without total reprogramming where assembly language is used), and (9) require fewer programmers. Clearly, a 4,000 statement program written in pseudo-English is much easier to manage than its equivalent of about 100,000 assembly language instructions.

Graham's experience with a high level language on the MULTICS project is summarized in the following:

> "MULTICS is written essentially in a subset of PL/1 except for a very few basic programs. Whether

> we would do it again: yes. The advantages show up particularly in big projects, with complex tasks. We get increased productivity of programmers. The programs are more easily understood, hence we can move people around easier, or replace them easier. One cannot predict the best techniques in advance, hence there is a need to rewrite parts of the system on the fly, which is easier with a high-level language. The machine code produced is not as good as that of good bit twiddlers, but probably as good as that of the average programmer." (2, p57)

Another argument against the use of assembly language is the fact that machine instructions in newer computers are becoming more numerous and more complicated to use. On the other hand, many arguments have been put forth in favor of assembly languages. The use of a high level language may involve the cost of designing a compiler. Assembly language programs usually run faster and require less memory. However, it is generally the case that about 95% of the program time is consumed by about 5% of the code. Thus it should be possible to "home in" on and optimize this 5% of the high level language program. Optimizations should first be made in the high level language. Only if this isn't satisfactory should assembly language "tuning" of sections of the 5% part be pursued. It may be necessary to lower to assembly language for larger percentages of programs that are more machine-oriented such as operating systems and input/output procedures. As to compiler efficiency, studies conducted on the Space Programming Language (SPL) have shown that it is possible to obtain compiler-generated code that is within 10% of the theoretical optimum (14). Of course, there may be real-time systems with very tight time constraints that for the chosen machine, require efficiencies that could only be satisfied by sophisticated assembly language techniques. (Perhaps a faster machine should have been chosen.)

Earlier, it was mentioned that post-design costs usually equal or exceed original design costs, however these costs are based on experience with assembly language programmed systems. In such systems, the transfer to a new machine usually requires a completely new design cycle. By comparison, the cost of a faster machine or an extra memory unit that may be required to accommodate the compiler-generated code is insignificant. The days of machines with expensive 2K - 4K word memories have long since passed. (Today, the combination of large-core machines with secondary mass memories usually is sufficient to relieve the program size problem.) Why than is assembly language used at all? One plausible explanation is given by Randell:

> "In debates like this most people agree on the benefits of high-level languages, but back in the

> field very few large projects use such languages. I believe the reason is that project managers, if left to make the decision, will decide on the basis of their own small world. They will seek to optimise the easily measurable figures on which they expect to be judged, and will aim to minimise core store used and maximise speed. They will ignore advantages such as portability, ease of re-coding, etc., even though in the long term these factors may well be of paramount importance. If such decisions were made at a higher level of the organization, the outcome would be very different." (2, p57)

Compiler efficiency is often a major problem. Some compilers are inefficient due to a poor design while others are inherently inefficient due to the nature of the languages being translated. Generally, the more powerful languages (e.g., PL/1) have less efficient compilers due to the frequent conversions required between data types, and the extensive use of built-in functions and external procedure calls. In this respect the Army's TACPOL language is superior since it retains the structural sophistication of PL/1 while eliminating costly types of data conversion and other expensive frills.

2.8 Testing

Alternately called "debugging," "validation," "verification," and "certification," the testing of software currently requires from 1/3 to 1/2 of the total design and development time. In (1) it is concluded that the distribution of software design and development efforts is 38% for analysis and design, 15% for actual program writing, and 47% for testing. It now should be clear that only through changes in the design approach (as previously discussed) can we hope to simplify and shorten the testing process.

The ideal testing philosophy involves a considerable amount of work to continuously design and exercise a complete set of tests of both individual modules and the system as a whole. If the design itself is based on the philosophy of structured programming then the testing problem will automatically be eased. Clearly, testing is inevitable and, like documentation, will always be less than successful when done as an afterthought.

Each of the discussions of the previous sections bear directly on the testing of software and will not be repeated here. For a discussion of specific testing techniques useful for system evaluation such as hardware/software monitors and simulation/analytical models, see (15) - (20).

2.9 Structured Programming

Many are familiar with the notion of "modular programming" which involves the division of a large program into small, self-contained modules each of which is separately designed and tested. The design and test of individual modules is followed by sequences of tests of the interoperation of integrated sets of modules until the entire program has been tested. However, modularity, while important, is not enough.

Developed by Dijkstra between 1965 and 1971, "structured programming" is a philosophy for total system design which may offer significant improvements in software reliability as well as in initial and post-design costs (10) - (13), (21) - (28). Not only does it concern the separation of a task into modules, but it involves the relationship between language structures and the internal design of the module itself. Structured programming demands that a high level language be used but it does not consider every such language to be satisfactory. To be acceptable, a language must contain and be restricted to certain control structures.

The basic approach is one of top-down design. The design proceeds through levels of abstractions where each level is a program for a machine that "exists" at the next lower level. Thus, each higher level is not simply a "sketch" of the work to be done with blanks to be filled in but a very definite set of commands for a machine that will eventually understand them. In going from level i to the next lower level, i + 1, we transform the level i primitives into new, finer primitives (by refinement). Dijkstra states that due to our "inability to do much" we cannot master the complexity of intrinsically large programs unless we follow the adage "divide and rule." The idea is that we cannot master the complexity of a large program if the mental effort required to understand it is more than directly proportional to the program length.

For example, suppose a programmer is asked to determine the complete behavior of an undocumented program that has been written in assembly language. He is not advised that this program is actually the assembly language equivalent a series of N "if--then--else--" statements. To thoroughly understand this program he must reason through all possible branches and determine the actions indicated, i.e., the behavior with respect to N binary decision points. This will require $N \times 2^N$ steps of reasoning.

Suppose we write the equivalent program in a high level language, where we make the N if-statements explicit, and give it to another programmer

to decipher. Since this program has more structure, less reasoning will be required as follows: Each if-statement will require two steps of reasoning, one for the "then" branch and one for the "else" branch. One additional step of reasoning is required for each if-statement to understand its relationship to the sequence of if-statements. For this case, the second programmer need only apply a total of 3N steps of reasoning.

Now suppose that each if-statement is replaced by a pseudo english sentence that would be clear to a non-programmer. In this case only N steps of reasoning would be required, one step for each sentence in the sequence.

For N = 20, the growth of complexity is N = 20, 3N = 60, N x 2^N = 20,971,520. It is believed that this is the underlying reason for the difficult development and poor performance of most large-scale software systems.

Dijkstra requires the software engineer to be knowledgeable in the field of program correctness proofs. Even though today's technology does not facilitate simple proofs of program correctness, familiarity with the techniques will aid in the design of programs that can be "proved" correct through less formal logical arguments. While such "proof" are not guaranteed, they are far superior to no proofs at all.

To illustrate the use of structured programming we present part of an example of Dijkstra (21). The problem is to instruct a computer to print a table of the first thousand prime numbers, with 2 as the first prime. In the following, each quoted instruction represents a primitive (for the given level) which is to be refined at the next lower level.

The structures are pseudo-ALGOL.

Level 0:

```
begin

"print first thousand prime numbers"

end
```

Level 2:

```
begin

"variable table p";
```

```
    "fill table p with first thousand prime numbers";
    "print table p";
    end
```

Level 2:

```
    begin
    2.1 integer array P (1:1000);
    2.2 "for k from 1 thru 1000, make P (k) equal to the kth prime
        number";
    2.3 "print P (k) for k from 1 thru 1000";
    end
```

In level 3, one possible refinement of step 2.2 of level 2 is:

```
    begin
      integer K, J;
      P (1) = 2;
      K = 1;
      J = 1;
    while  K < 1000
     do
      "increase odd J until next odd prime number";
      K = K + 1;
      P (K) = J;
     end
    end
```

In successive levels, the primitive "increase odd J until next odd prime number" will be further refined until all primitives are reduced to instructions acceptable to the source language.

This approach permits the formulation of a one page "program" that may represent a hundred page program where the former gradually expands to the latter via successive refinements. The design need not (and usually will not) flow smoothly from top to bottom. There will probably be frequent returns to earlier levels to change the choices made there followed by changes in subsequent levels. This is a natural part of the design process and will cause no difficulties if each module at each level is made to account for itself as a complete program.

If we start out with the overall program P as a single block (Figure 5) then the structured programming approach requires the partitioning of P into parts P_1, P_2, ---, P_n (Figure 6) with the flow of control from P_1 to P_n as shown (no branching) where each P_i is a simple assignment, a procedure call or one of the forms shown in Figures 7 - 11. Each of the allowed forms of P_i has the property of a single entry point and a single exit point. Adherence to this sequencing discipline, which rules out arbitrary jumps between standard structures, will confine the mental effort to understand P to a linear function of program length. It can also be shown that the restriction to the above structures and sequencing discipline will permit use of a very efficient dynamic monitoring scheme (Dijkstra, private communication).

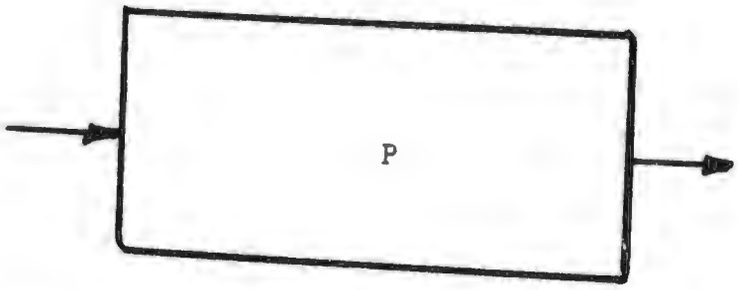


Figure 5. Program P as a Black Box

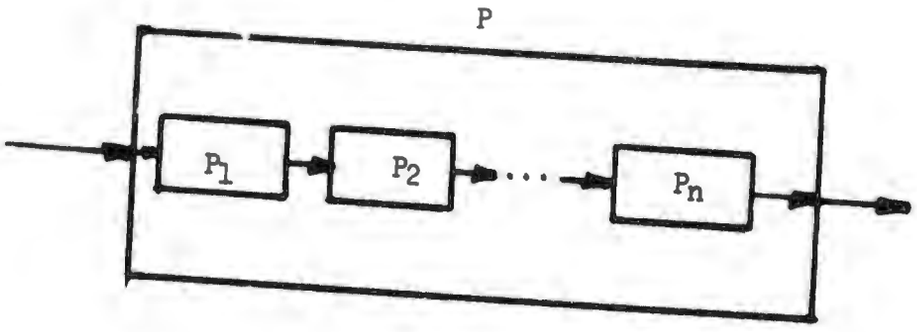


Figure 6. Program P as a Sequence of Simply Linked Programs

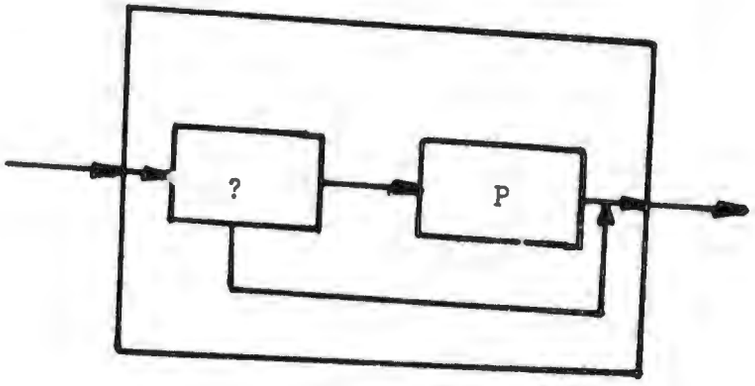


Figure 7. IF condition THEN P

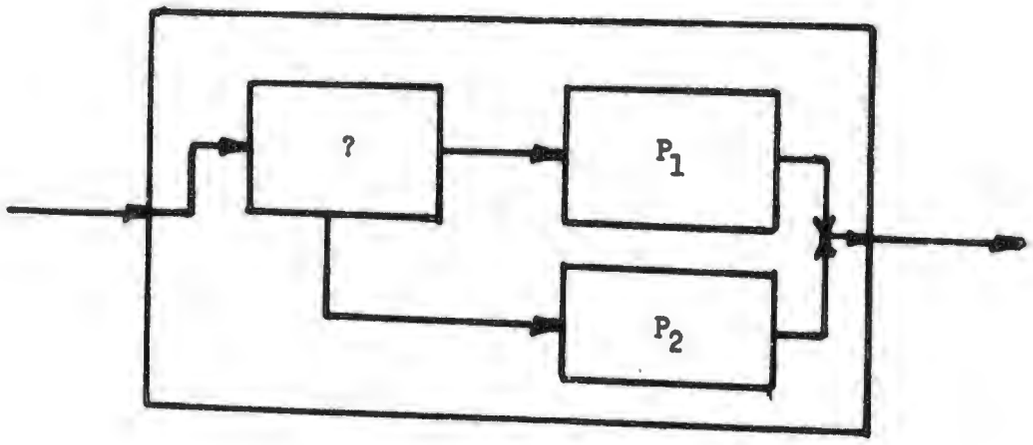


Figure 8. IF condition THEN P_1 ELSE P_2

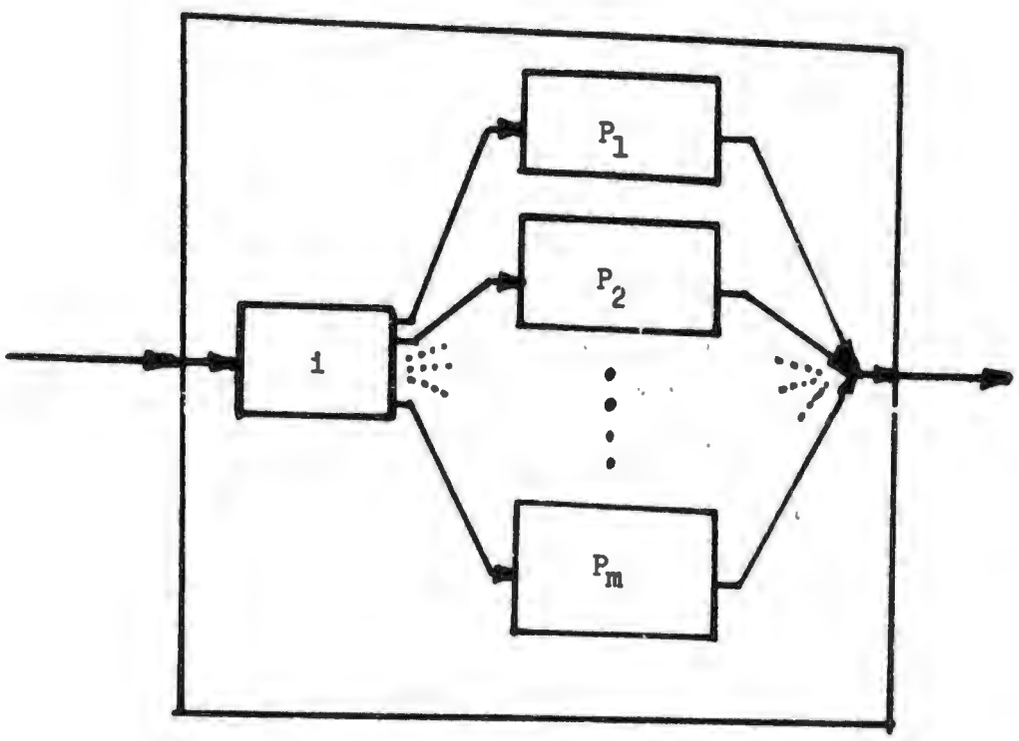


Figure 9. CASE i OF (P_1; P_2; ...; P_m)

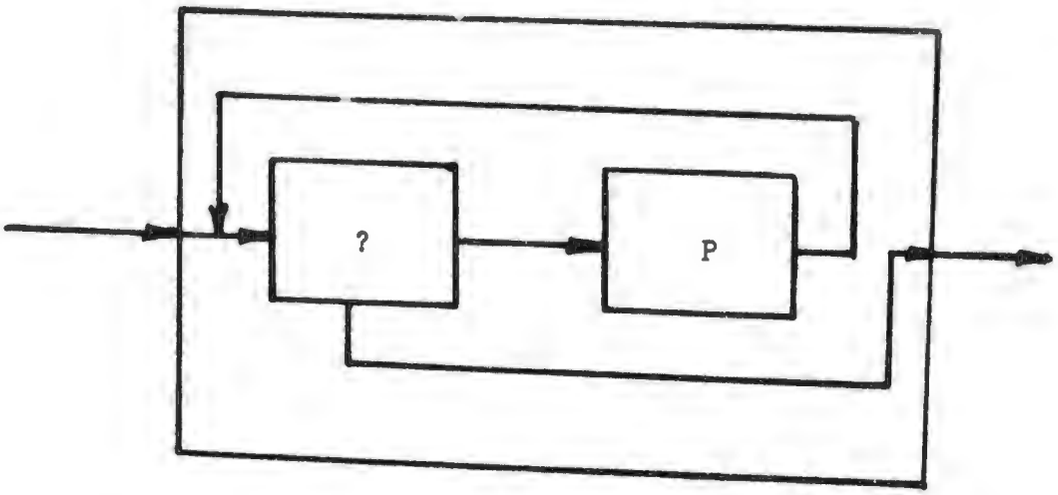


Figure 10. WHILE condition DO P

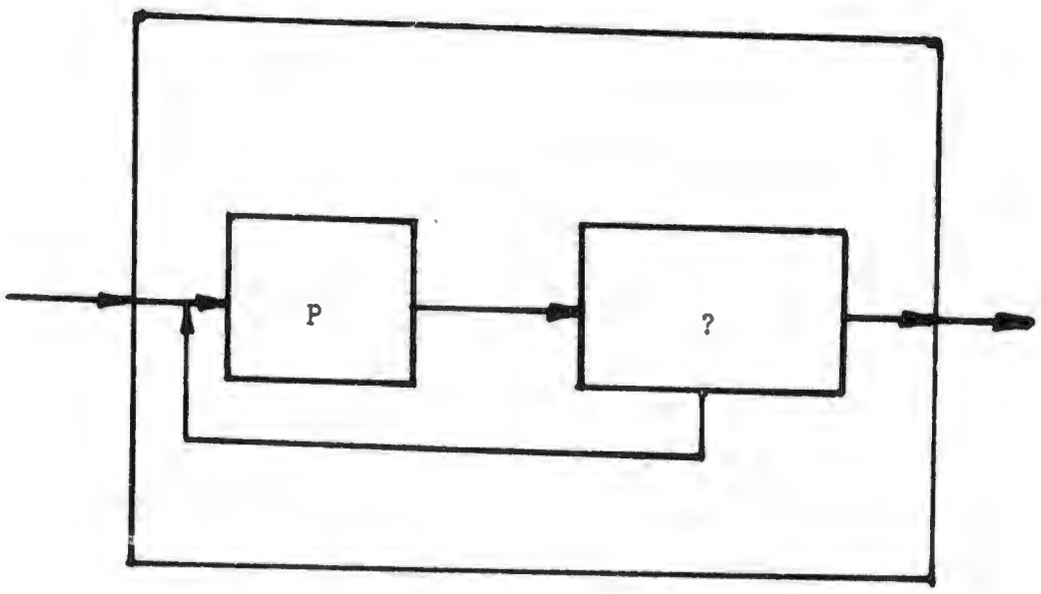


Figure 11. REPEAT P UNTIL condition

3. SUMMARY OF POSSIBLE SOLUTIONS

Following is a listing of feasible and near feasible approaches to software design and development.

a. Don't over-promise or over-expect with regard to the performance of tactical software systems.

b. Allow sufficient time for system formulation, analysis, and preliminary design.

c. Allow sufficient time for testing.

d. Don't modify systems using patches. Go through an extensive redesign.

e. Integrate software performance into total system performance evaluation and effectiveness, i.e., do not assume perfect performance of software.

f. Develop good documentation standards and use them faithfully.

g. Generate adequate documentation during, not after the design and development process.

h. Prepare adequately commented source listings.

i. Base all decisions on the total life cycle costing of software.

j. During design, anticipate and plan for post deployment redesign.

k. Plan for transferability by avoiding excessive machine dependence.

l. Use fewer but more highly qualified software personnel capable of performing software engineering.

m. Do not develop software in one step. Build a prototype system. Go through advanced development and engineering development phases for software.

n. In design avoid over-sophistication and trickery. Design software that is easy to understand.

o. Use the techniques and philosophy of structured programming, i.e., avoid the free use of "go to's", design using a good quality high-level language.

p. Design adequate tests concurrent with the software design process.

q. Give serious consideration to total system life cycle cost trade-offs between the use of faster, larger memory machines programmed in a high-level language versus slower, smaller machines programmed in assembly language.

r. Don't be parochial. Give adequate consideration to hardware/software trade-offs. They always exist.

s. Concentrate on the enhancement of compilers with respect to their compilation speeds as well as their object code efficiencies.

t. Make extensive use of quiescent and dynamic monitoring and evaluation techniques to discover poorly operating software.

u. Identify common design areas and technical problems among different systems. Modularize wherever possible by abstracting functional components and their structures.

v. Develop a comprehensive model of the software acquisition process in order to determine optimal paths through the life of a system.

w. Prepare adequately for hardware/software integration.

REFERENCES

1. BOEHM, B.W., and HAILE, A.C., et al. (1972) Information Processing/ Data Automation Implications of Air Force Command and Control Requirements in the 1980s, Executive Summary. Report SAMSO/ XRS.71.1. U.S. Air Force

2. NAUR, P., and RANDELL, B. (ed.) (1969) Software Engineering (Report on Conference). NATO Science Committee, Brussels 39, Belgium.

3. BUXTON, J.N. and RANDELL, B. (ed.) (1970) Software Engineering Techniques (Report on Conference). NATO Science Committee, Brussels 39, Belgium.

4. SHOEMAKE, J. (1960) EDP Systems Management. The Chesapeake and Potomac Telephone Co., Silver Springs, Maryland.

5. ARON, J.D. (1970) Estimating resources for large programming systems. Software Engineering Techniques (Buxton, J.N., and Randell, B., eds.), NATO Science Committee, Brussels, Belgium.

6. OGDIN, J.L. (1972) Designing Reliable Software, Datamation.

7. HARKINS, J.A. (1972) Computer Software: A Major Weapon System Component. Professional Paper No. 87, Center For Naval Analysis, Arlington, Virginia. (DDC AD 736357)

8. SAMMET, J.E. (1970) Perspectives on Methods of Improving Software Development. Proc. 3rd Symposium on Computer and Information Science. (COINS III) Academic Press, New York.

9. SCHWARTZ, J.I. (1970) Analyzing large-scale system development. Software Engineering Techniques (Buxton, J.N., and Randell, B., eds.), NATO Science Committee, Brussels, Belgium.

10. LEAVENWORTH, B.M. (1972) Programming with (out) the GOTO. Proc. ACM Annual Conf.

11. WULF, W.A. (1972) A case against the GOTO. Proc. ACM Annual Conf.

12. HOPKINS, M.E. (1972) A case for the GOTO. Proc. ACM Annual Conf.

13. DIJKSTRA, E.W. (1965) Programming considered as a human activity. Proc. IFIP Congress, Spartan Books, Inc., Washington, D.C.

14. NIMENSKY, R.E. (1970) Space Programming Language: Flight software comes of age. High Level Aerospace Computer Programming Language Conference, Naval Air Systems Command, Washington, D.C.

15. GOTTLIEB, C.C. and MAC EWEN (1970) System evaluation tools. Software Engineering Techniques, NATO Science Committee.

16. PINKERTON, T.B. (1969) Performance monitoring and systems evaluation, Software Engineering, NATO Science Committee.

17. LUCAS, JR., H.C. (1971) Performance evaluation and monitoring. ACM Computing Surveys, Vol. 3, No.3, Sept. 1971.

18. ESTRIN, G., et al. (1972) Modeling, measurement and computer power. Proc. IFIPS SJCC.

19. LLEWELYN, A.I. and WICKENS, R.F. (1969) The testing of Computer Software. Software Engineering, NATO Science Committee.

20. DARDEN, S.C. and HELLER, S.B. (1970) Streamline your software development, Computer Decisions.

21. DIJKSTRA, E.W. (1969) Notes On Structured Programming, EWD 249, Technical University, Eindhoven, Netherlands.

22. DIJKSTRA, E.W. (1970) Structured programming, Software Engineering Techniques, NATO Science Committee.

23. DIJKSTRA, E.W. (1968) Go to statement considered harmful (Letter to the Editor) CACM Vol. 11.

24. DIJKSTRA, E.W. (1968) A constructive approach to the problem of program correctness, BIT Vol. 8.

25. WIRTH, N. (1971) Program development by stepwise refinement. CACM Vol. 14.

26. MILLS, H. (1971) Top down programming in large systems, Debugging Techniques in Large Systems (Rustin, Randall, eds.) Prentice-Hall.

27. KNUTH, D.E. and FLOYD, R.W. (1971) Notes on avoiding "goto" statements, Information Processing Letters 1, North Holland, Amsterdam.

28. PARNAS, D. (1971) On the Criteria to be Used in Decomposing Systems into Modules, Computer Science Dept. Rept., Carnegie-Mellon University.