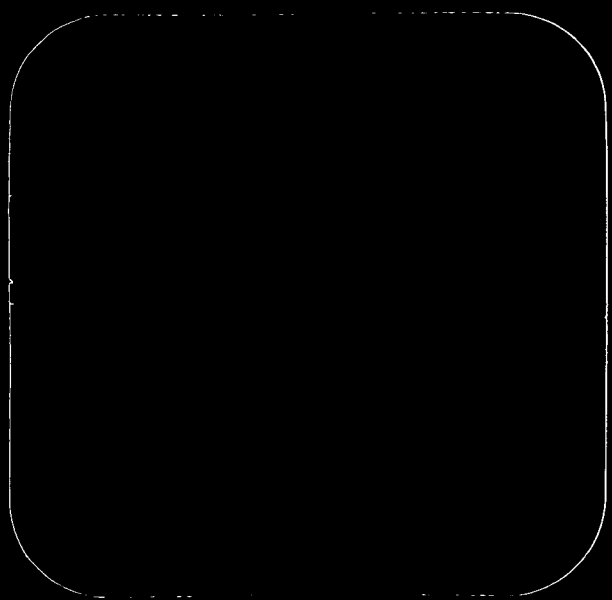
Information and Management Sciences



AUERBACH

110 pr

THE SYSTEMATIC EVOLUTION OF A NASA SOFTWARE TECHNOLOGY

APPENDIX C TO

TECHNICAL REPORT
1958-100-TR-004

By

M. P. DEREGT
JOHN E. DULFER

Submitted to:

NASA Headquarters

Under

Contract No. NASW-2285

August 24, 1972

AUERBACH Associates, Inc.
1501 Wilson Boulevard
Arlington, Virginia
22209

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

TABLE OF CONTENTS (CONTINUED)

SECTION 1. SUMMARY

1.1 PURPOSE

This document describes a long-range program whose ultimate purpose is to make possible the production of software in NASA within predictable schedule and budget constraints and with major characteristics - such as size, run-time, and correctness - predictable within reasonable tolerances. As part of the program a pilot NASA computer center will be chosen to apply software development and management techniques systematically and determine a set which is effective. The techniques will be developed by a Technology Group, which will guide the pilot project and be responsible for its success. The application of the technology will involve a sequence of NASA programming tasks graduated from simpler ones at first to complex systems in late phases of the project. The evaluation of the technology will be made by monitoring the operation of the software at the users' installations. In this way a coherent discipline for software design, production maintenance and management will be evolved.

1.2 BENEFITS

The potential benefits include the ability of management to understand and control the plans, activities, budgets and products of its software development group, major increases in the quality and quantity of software delivered at a given cost, and in the control of software quality, expressed in terms both of responsiveness to the users' expressed needs and in the correctness and other characteristics of the software itself. These benefits, obtained and tested at the participating center's software group, may be propagated throughout NASA by exporting the resulting technology to other NASA computer installations and adapting it to the operations peculiar to each one.

Scarcely a programming activity in NASA exists which will not be a beneficiary of this program. For example, NASA general purpose and mission oriented computer installations which must provide high reliability service to their users (which is especially important for real-time systems) will benefit from a systematic application of techniques designed to enhance system reliability. They will also benefit from the critical review, evaluation and development of techniques for the design of complex systems. A standardized set of design and design verification techniques which provide for enhanced system effectiveness will eventually drastically reduce the cost of system development and maintenance.

1.3 SCOPE

The Program will be of an evolutionary nature, as indicated by its title. It is seen as a series of cycles, each beginning with the selection of a programming job. The first cycle will involve a relatively straightforward application program, so that the working together of the three groups - technology, software, and user - can be developed. Each cycle after the first will involve a program appreciably larger and more complex than the previous. In each case, the program to be produced will be one actually required by a user. In this way, problems of great size and complexity will be approached gradually, the magnitude of the increments being limited as decided by the Technology and Software Groups.

Each cycle consists of three phases. The first, a Research and Development Phase, is for the Technology Group to analyze the results of previous cycles, incorporate indicated changes and improvements in the software technology and in the documents, manuals, papers, procedures, and standards, and to train the Software Group personnel in the changes and improvements.

The second phase will be a Software Design and Fabrication Phase, in which the Software Group, using the improved technology, produces the software selected for the current cycle. In the third and last phase, the User Group incorporates the new software into its operation. Communication is maintained between all three groups during all phases.

Selection of programs for new cycles can continue until the NASA software community is satisfied that a usable software technology exists. At some point, the scope of the program could be expanded to include micro-programming, and at a still later point, hardware, so that an overall Computer System Technology can evolve.

Typical of the facilities which will be developed and/or applied are the following: languages to express user requirements, languages and graphics to express computer program design, documentation aids and standards, structured programming and topdown programming, computer based aids such as debugging tools and standard application modules.

The first cycle takes place over a period of 18 months. The Technology Group is seen as a four man team initially. The funding for this group should be provided by NASA Headquarters. In this way the long-range plans and policies of NASA will be able to influence directly the development of the software technology responsive to NASA needs. The size of the software and user groups, which are funded by their respective centers, is at present undetermined, being subject to later determination based on the nature of the particular application selected.

1.4 SUMMARY OF REMAINING SECTIONS

1.4.1 Section 2. Background

The basic problems of programming are introduced: lack of theory, lack of languages to describe user problems and the structure of programs, lack of an engineering discipline. These problems are assumed to exist in NASA, for the purpose of the program description in following sections. The remedy is a project modeled after a mature engineering technology - computer hardware production, for example - which supplies the coherent engineering approach lacking in current software production activities, and transforms them into a NASA Software Technology.

1.4.2 Section 3. Software Technology

Recently developed techniques needed in the design and development of software, such as structured programming, chief programmer teams, and proofs of correctness are reviewed.

1.4.3 Section 4. Statement of the Problem

The problem tree is set forth in three parts: basic or root causes, practical difficulties (the stem of the tree), and characteristic programming deficiencies (the branches). The latter are those evident to the external observer of a programming activity which is in difficulty, namely, schedule and cost problems, poorly performing computer programs, inflexible (unmaintainable, inextensible) programs and errors.

1.4.4 Section 5. Description of the Program

The problems of, and the lack of an engineering approach to software development have been noted at international symposia on software engineering. No sustained effort dedicated to the solution of the problems or the development of a software engineering approach, however, has been undertaken. The program recommended for NASA is such a sustained, dedicated effort. Its objective will be to develop a software technology, consisting of languages, procedures, models, organizations, documentation, specifications, job descriptions, plans, and standards. For example, a satisfactory

means to represent the structure of programs prevents the exercise of spatial perception, an innate human ability, in the detection of programs not responsive to users' needs or impossible to produce, and in the detection of structural anomalies. Such anomalies are not revealed by flow charts, which describe processes rather than structures.

The program is designed to correct such deficiencies in language. The resulting ability to represent quite complex software structures rather than merely processes will permit greatly improved communications between the successive steps of software development. Consequently, division of development responsibilities can be made and well-defined professional and technician specialties can emerge, permitting improved control over cost and quality by meaningful inspection of stages of production. One such division of responsibilities is that between design and fabrication, common in other technical industries but absent in the software industry.

These differences between development of software and traditional engineering having been pointed out, terms are defined and software engineers' job positions are described. The goals and objectives of the program are stated, and finally the organization and operation of the Technology, Software, and User Groups are described.

1.4.5 Section 6. A Specimen Technology Group

The Software Group and its associated User Groups already exist; the Technology Group must be created. Section 6 provides guidance on how this might be done. The purpose and mission of the Technology Group is examined in more detail, relative to the NASA management groups sponsoring the program. The organization and operation of the group is discussed, also in some detail, including its size and composition, the qualifications of the staff, and job descriptions.

The products of the group are described in some detail. Forms of communication are the physical products. Included will be primarily internal technical memoranda, papers for professional journals,

articles for trade journals, perhaps books. Also included will be oral presentations such as seminars, symposia, and training programs.

However, the principal topic of this section concerns the subject matter of these communications. Specific examples of products discussed are: industrial standards, representation and languages, software production techniques, and performance standards.

Implicit throughout the report is the close interaction between the Technology Group and all segments of the computer community and especially with the Software and User Groups. Such interaction is the foundation of effective communication, and that, of course, is the most important product of the Technology Group.

SECTION 2. BACKGROUND

2.1 THE ORIGIN OF COMPUTER PROGRAMS

The first electronic computer, the ENIAC, was built during World War II at the Moore School of Engineering at the University of Pennsylvania to produce mathematical tables required for the firing of projectiles. These tables required numerical integration of differential equations -- hence the name ENIAC (Electronic Numerical Integrator and Calculator). The machine was completed in 1946 after three years of work and was a success. Whereas a single trajectory had taken about twenty hours to compute using a desk calculator, ENIAC could complete the computation in thirty seconds.

To prepare ENIAC for a calculation, it was necessary to set up all the connections manually beforehand for the transmission of data among units. Each unit had to be wired to recognize when it was to start and which operation it was to perform. The sequence of processing was determined by the input and output signals between units. All of the units operated synchronously with each other. Hence, it was possible

for several of them to operate simultaneously. Under this system it was a day's work to set up the equipment. Checking the wiring for accuracy, an arduous task, accounted for much of this time.

John Von Neumann, a child prodigy born in 1903 and one of the greatest mathematicians of the twentieth century, contributed the major theoretical advance in computer design, the stored program. Based on a study of the logical design of computing machines, he saw the need for numerous loop sequences in scientific and engineering computation and saw that a new technique was required to carry out these computations by machine. This led to the suggestion of storing instructions in a machine in the same manner as data, which would enable machine commands to be manipulated by arithmetic and logical operations. In this way a machine would have the ability to change or modify its instructions. In 1947 Von Neumann suggested a method for converting the ENIAC into a stored program machine.

As proposed by Von Neumann, the operation carried out by each ENIAC unit was fixed by permanent wiring and each unit placed under a central control. Codes were stored in memory for the actuation of the necessary operations. The sequence of commands necessary for solving a problem was given to the machine in the form of a list of coded instructions stored in the memory. The central control system was designed to obtain and execute commands in the same order in which they were stored. Special instructions enabled the machine to perform branching and looping by modifying the sequence in which instructions were obtained from memory. These methods have since become basic to all data processing machines. Thus, ENIAC was a prototype of all later equipment not only in being the first electronic computer but also the first stored program machine in operation.

All of ENIAC's 18,000 vacuum tubes had to be operating for the machine to function. The task of tracing down a malfunctioning vacuum tube, however, is straightforward compared to that of determining the source of a malfunction in the stored program. Tubes operate independently of each other, and if necessary each can be tested. Programs,

however, are subject to subtle interdependencies among their parts; and, in addition, no objective test of their "correctness" is possible. A measure of the complexity of programs is afforded by the degree to which their various parts are cross-coupled, or by their interconnectivity. Since the days of the ENIAC, programs have become ever more complex. Let us trace the history of this growth.

2.2 THE EVOLUTION OF COMPUTER OPERATING SOFTWARE

In early computing installations, the method of operation was, simply, one task at a time in sequence. Typically, a single program was loaded into the computer and run to completion. Only then was a second program physically loaded and run -- a sequential, discrete program execution concept.

2.2.1 Discrete Program Execution

A primary disadvantage to the sequential execution philosophy was inefficient use of equipment. Generally, programs which made great demands on the internal data processing capacity of the system required input/output devices only infrequently during processing. Conversely, those programs which required the input/output system much of the time frequently permitted the data processing elements of the system to idle at a fraction of capacity.

However, inefficient equipment usage was not the only drawback to discrete program execution. Each time a person wanted to run a program, he had to perform the same tedious steps. He prepared his program in a form suitable for the computer, described his run requirements and expected behavior, waited in turn for machine time (usually programs were processed in the order submitted), allowed time for solution, and finally, awaited the return of his results.

The total elapsed time to complete all these steps exploded to many hours and often days. One source of delay was the operator, who had

to read the operating instructions associated with each job and determine that the requirements of the program were met. Another source was the occurrence of errors. An error might be a machine failure, a mistake by the programmer, or a blunder by the operator himself. In any case, the operator usually did not have adequate information about the problem to correct the error, or he needed time to search for a solution. It became obvious that the computer should be given some of the responsibility for managing and controlling its own facilities.

2.2.2 Job Stream Concept

In order to increase the efficiency of computer systems, computing centers began to require that the programmers write their programs and operating instructions in a uniform format. At the computing site there was also standardization. Programs to be executed were all submitted in the same way and were batched for processing, the data for these batched programs was entered from a standard input device, and the results of the programs were directed to a standard output device. The computing center could now begin one problem program (job) and go right on to a second and a third without physical interruption. A system program was devised to monitor the transition from one job to the next in the batch. Standardization and the job-to-job transition monitor (the job stream monitor) reduced dependence on the operator and introduced the concept of batch processing procedures through job stream processing.

Still further efforts to increase equipment utilization led to greater use of the computer itself to perform clerical and error-recovery tasks that had been required of the system operator. This program, devoted to internal regulation and control over the total operation of the computing system, became known as the operating system. Increasingly, the operating system supplanted certain operator functions and improved equipment utilization. The function of the system operator changed from essential intermediary to assistant to the computer.

2.2.3 Operating System Concept

An operating system, then, is a software unit which makes the decisions that previously were left to an operator. In addition to providing standard responses to error conditions (both in programs and in the equipment itself), the operating system must also allocate and control the resources of the computer: input/output devices, memory, time, and all other components comprising a modern computing installation. Further, the operating system must keep accounting records for each resource allocated to each program.

2.2.4 Multiprogramming Concept

Despite the improvements that operating systems make in efficient equipment management, a given program still did not occupy all the equipment at all times; usually one program loaded the input/output system fully while another was limited by the internal data processing unit. A way of loading all the equipment to its maximum was still needed.

In any large computing center there are usually different types of programs requiring varied system resources. Under the control of an expanded operating system, several programs can be loaded at once. One program is started; then the operating system surveys the other problem programs resident in the computer to see if any can make use of the remaining available equipment resources. The operating system continually examines each available program in an attempt to keep all parts of the system totally employed. Such control and manipulation of several different programs in the system is called multiprogramming. The multiprogramming operating system was designed to make the most effective use of all equipment under any given situation. Although turnaround time was reduced slightly, there was still no direct communication between the computer system and the programmer -- or the nonprogrammer, such as the scientist or engineer.

2.2.5 Communications Capability Added

As multiprogramming operating systems were being developed, the capabilities of computing systems were being extended by adding a communications facility. There are three principal applications which communications devices support.

- Rapid and varied data gathering. Many different device types can be attached to a computer so that it can receive or transmit data. Typically, these might be voice-grade units, graphic displays, telemetry equipment, remote card readers, and even remote processors.
- Message switching. One station can initiate a message and send it to any other station(s) in the network.
- Inquiry/response of data retrieval. A collection of data files (data base) is searched for a datum; or conversely, the data base is altered by new information coming from some remote point.

All three applications share the common feature of situating the processor at a central site and, through communications channels, sending information to or receiving information from remote locations. Herein, the computer is used less for calculation and more for data handling during the communication process, for if used solely for calculation it would need few external communications.

2.2.6 Interactive Operating Systems

The obvious affinity of multiprogramming systems and communications equipment led logically to the next step of operating system development: interactive systems.

The interactive operating system differs functionally from the multiprogramming operating system in the control of multiple user tasks within the same relative time frame. Whereas multiprogramming allows two or more users access to a computer but discriminates between their core resident time through a priority scheme, an interactive system provides each individual terminal user equal opportunity to contend for use of the system.

The inquiry/response system is a rudimentary interactive system; a query can be entered at a remote terminal and transmitted to the computer to elicit a response. However, the inquiry/response system does not permit the inquirer to do more than request information from a single collection of data or to send new information for inclusion into such a data base.

Under an interactive operating system, the terminal user is no longer restricted to a single data base, but can create new data bases, write programs, or alter existing programs. In fact, the user of an interactive system has available at his terminal all of the capabilities of the complete computing system and facility.

There is an added difference between interactive and conventional systems. Unlike earlier input/output devices, the interactive terminal can be programmed to respond. If the user makes a mistake, he will be so advised immediately, and may be prompted to provide corrections. He may ask for explanations of the various features that he wants.

Most important, the user can get information, answers, or reports in minutes instead of hours. With present-day equipment, operating speeds within the computer are so fast that each user at his own terminal does not realize that any other terminal is in use. The ordinary reaction time of man is so much longer than the computer's processing time that the system seems to respond immediately.

There are obvious advantages to the interactive philosophy. One former deficiency inherent in previous systems was that of inadequate turnaround time. With an interactive system, however, turnaround time is principally a function of the processing time to compute the answer. If the user request can be calculated quickly, it will be turned around quickly; if the request begins a long computation, the result will turn around when the computation is completed.

Another benefit of the interactive system is that the user is directing and communicating to the computer conveniently. The user operates his own terminal device, does his own programming, and receives his own results as they are computed.

However, an exclusively interactive operating system had certain limitations in computing power and programming language flexibility, while the computer still possessed the untapped capability to run background programs concurrently with interactive tasks. Background programs do not interact with any terminal device although they may be initiated from a terminal. They are the conventional batch programs that had been run under the multiprogramming or job stream operating systems, or formerly even without operating systems.

This observation led to the development of symbiont operating systems, which support both interactive conversational jobs and background batch jobs simultaneously.

2.2.7 Symbiont Systems

A symbiont system combines with its interactive capabilities the complete and concurrent facility to support a broad range of production/batch tasks. Whatever resources are not occupied by interactive requests are given over to processing typical production jobs that are now performed under an exclusively batch or multiprogramming operating system. Conventional production tasks are those that can be submitted to the computing center on a given day, but whose solutions are not required immediately. Often these tasks require a relatively long time for solution but need no interactive operation. Such programs are payroll, inventory, billing, and sales analysis runs.

Under symbiont operation, interactive terminal users are generally treated equally as they contend for system resources. The emphasis

when executing background programs along with terminal processing can be determined and balanced by the individual computing installation. Hence, a particular installation can control the allocation of resources between terminals and background programs dynamically and according to immediate needs; it is even possible to exclude one or the other.

2.3 COMPUTER BASED AIDS

Programs for the earliest machines had to be written in the machine's own language. The vocabulary consisted of a set of commands, each invoking a particular machine function and represented by a string of digits of some definite length. This language, of course, was very awkward for programmers to use, because they are accustomed to dealing with words and diagrams. While computer operating software was evolving, a number of computer-based aids were being developed to ease communications between the programmer and the central processor, relieving the former of the tedium of translating his thoughts into machine language. The means for doing this are known as assemblers, at the lowest level, and compilers, for high-level languages. Assemblers are language translation programs that convert symbolic source language into numeric machine language, usually with a one-to-one correspondence. The source language translated by an assembler is called the assembly language and is highly dependent on the computer's instruction set. A compiler also translates source code into machine language, but each written statement in the compiler language is translated into several machine instructions. Generally, the term "programming language" is employed to specify the source language translated by compilers.

Language translators, together with other computer-based aids and the operating system constitute what is known as system software, or the collection of programs the programmer may reasonably expect to find resident on his computer for use in developing applications programs. Other aids include mathematical routines (e.g., for evaluating a sign function); generalized input/output routines; programs to manipulate data, such as a sort routine; programs to assist in the diagnosis of

machine malfunctions and program errors; and even systems of programs to manage data, called data management systems. As the magnetic storage immediately accessible to the computer's processing unit is too small to contain it all, the system software generally is located on secondary storage. When commanded to do so, the operating system transfers the required system programs to central storage for use. This service the programmer invokes by inserting the prescribed instructions in his program at the point where the systems program is required.

2.4 APPLICATIONS PROGRAMS

These are the programs required actually to perform a job for some user. Unlike the systems programs, which can be shared among all users, applications programs are tailored to their specific purpose. This may be a business function, such as accounting; a library function, such as the storage and retrieval of textual information; scientific calculations, which are characterized by numerical evaluation of mathematical formulae; or communications and other non-numeric data processing, which require the manipulation of individual bits of information. Applications programs also are stored on auxiliary storage and brought into core as needed.

2.5 SOFTWARE SYSTEMS

By a system we mean that mechanism capable of all the functions implied by its external interface. A software system is a collection of programs just sufficient to discharge such system functions and would include programs of all the foregoing types; that is, computer operating programs, computer based aids, and applications programs. For example, the Apollo software system includes all of the applications programs necessary to process communications, drive displays, compute trajectories, etc., as well as the IBM operating system supplied with the Apollo computers, and the various utility programs required for maintenance and improvement.

Two points should be made about software systems. The first is that they can get very large and very complex. The basic IBM operating

system, DOS/360, is a commonly cited example of a large and complex program. It required as much investment to develop as the system/360 hardware itself, amounting to many thousand man-years and resulting in a program running to several million instructions in length. The Apollo software system is of similar magnitude, probably costing somewhat more because of the need for absolute dependability.

The second point is that the determination of the users' requirements is an essential first step in designing a data processing system. Following this, the system, comprising hardware and software components, is postulated so as to satisfy these requirements. If the system is both feasible and in some sense efficient, detailed design work proceeds. The term software technology as used in this appendix is meant to include both the requirements definition and design phases of information system building. One of the processes which will come in for careful scrutiny and discussion in later sections of this work is defining and expressing users' requirements. While this may stretch the notion embodied in the term software, its inclusion is deliberate and indeed necessary in order to focus on deficiencies in the software building process.

2.6 THE PROBLEM

Computer programs from the very first have been subject to errors -- missteps in coding, perpetrated by the programmer and not found until after the results of the program's operation are examined and seen to be in error. Errors may be obvious or elusive, but in either case they have to be diagnosed after the fact, for the computer proceeds at such a pace as to make concurrent diagnosis out of the question. The human tendency of programmers to err is with us in undiminished form today as it was twenty-five years ago.

Programmers seem to be unable to estimate the size or the difficulty of writing a program which they have never attempted before. This becomes highly undesirable in large programming projects, requiring dozens or even hundreds of programmers, which therefore have a tendency

to miss their scheduled target dates and costs by wide margins. Unfortunately the miss is usually in the direction of an overrun, a fact attributed to inefficiencies due to the large organizations required and a source of discomfiture to project managers.

In the words of one observer:

> These problems are symptomatic of the lack of an adequate basis in the methodology, technology, and theory of information systems and/or a lack of disciplined application of the methodology and technology we do possess. We are cursed with the problem of the large, complex system -- problems of dimensionality and scale -- for which there is neither an adequate science nor an adequate engineering discipline.
>
> The problem has been compounded by laying too much stress on what poses for efficiency as a design criterion; namely, speed of computation, in terms of the number of machine instructions executed. This is well understood, we can measure it, and it has become the accepted measure of computational efficiency.
>
> Unfortunately, in information systems this type of efficiency is often bought at too high a price: namely, lack of structural modularity. More stress should be placed on structural modularity and the utilization of predeveloped, pretested, multipurpose elements, as well as tolerance to abnormal traffic and error conditions as measures of system effectiveness.
>
> Too often trial and error is the practiced methodology to match an information processing system to the need. The heuristic approach is still the rule rather than the exception in a computer systems design.
>
> In defining the information system requirements, frequently the real problem is not clearly known or, even worse, is incorrectly defined. As a result good solutions are formulated to wrong problems. System specifications may propagate incorrect problem definitions that are biased by the designer's experience so that they will reflect the limitations and errors of other systems. Empirical solutions are frequently "force-fits" and inefficient solutions to the problem. 1

The existence of problems has been recognized outside the United States. In 1968 and 1969, conferences were convened in Europe by the NATO Science Committee to define the problems better and try to find solutions, for which a special term, <u>software engineering</u>, was coined.

In the meantime, although software development deficiencies are getting a lion's share of the attention, a great majority of programs have been running satisfactorily on comtemporary computers. Numerous examples of very complex operating systems are functioning satisfactorily, and there is a growing inventory of checked out, debugged, and functional programs for almost any purpose one can name. Spectacular troubles with software still can arise when a novel program or a programming system of very large size is attempted. A criterion for the external success of a software development project, as measured by its remaining on-time and within budget, seems to be: obtaining the services of program design personnel who have successfully completed a similar system. Naturally, this criterion cannot be met by programming projects to extend the state of the art. Currently, these projects seem to be chiefly large-scale computer control systems such as an airline reservation system, an anti-ballistic missile defense system, or possibly the system required to support the NASA Space Shuttle.

SECTION 3. SOFTWARE TECHNOLOGY

3.1 The term Technology usually means the systematic application in an industrial setting of a discipline, usually based on a science. In this section we shall be writing about the means for producing software outside of the research laboratory, for short term or protracted use. Such a broad subject has to be narrowed, in a document of this extent, and therefore we will limit ourselves to a discussion of design and programming methods, and organization and management. Even in this narrowed context, space and time force further refinement of the topic. Where appropriate, the discussion will be confined to aspects of these topics which are undergoing or have potential for improvement.

3.2 DESIGN AND IMPLEMENTATION

In most industrial processes design and production are separate. Different skills and different plant facilities are required. In the development of computer programs, design and production, or coding, cannot be separated. As in an art such as sculpture or prose composition, the medium of design thought and the substance of which the final product is made are identical. In designing a computer program the medium is a

programming language. The fine points of the design can't really be thought out until an attempt is made to implement them in computer code of one level or another. Coding the program in turn affects the design; for example, it sometimes invalidates design assumptions of what can be implemented. Thus, the design process is tantamount to coding the program.

The industrial practice of programming must recognize different skill levels. Master designers can't write every line of code for a large system, nor can junior programmers execute major design decisions. So, industrial programming is more like medieval guild practice, in which small groups - six or seven programmers - are needed to insure sufficient communication between the junior and senior members. Large programs are produced by many such teams. Relatively poor results have been experienced in very large programming groups where this communication was not possible.

The word design is used both as a verb, to design, and as a noun referring to the structure or other properties of the program. A few techniques for both types of design will be discussed: for example, modular design (a design structure) and structured programming (a design method). The procedure termed top-down coding is an implementation technique necessary, because of the intermixing of implementation with design responsibilities, to avoid design difficulties.

3.2.1 Properties of Good Program Design

Having distinguished between software design methods and design structures, we will simply write design at times notwithstanding the ambiguity, thus: a good design should be efficient both in the use of machine resources and of designer effort.

A good design has the following properties. It should be correct. It should be segmentable, or modular, so that parts of the program can be replaced, in restricted central storage, by other parts automatically. It should be robust, to survive unexpected data or processing loads or

unforeseen error conditions. A good program should also be extendable if it is to have a long service life. While efficiency and correctness are probably the most important properties of a good program, recent developments in program design concentrate on the idea of correctness.

3.2.2 Conceptual Design

There are two phases of design. In the initial, or conceptual, phase an overall design of the program system is developed in flow chart or block diagram form. Usually, the conceptual phase includes an elaboration of the design level by level in increasingly greater detail until the entire block structure of the program has been defined. Then there follows a coding phase in which the details of the blocks are filled in in machine executable instructions.

A traditional approach to conceptual design has been to break the program system into modules. Each module discharges a single function or a group of related tasks and has the property that relatively little data crosses its boundary. The essence of this approach is subdividing the program into entities of some kind, requiring a minimum of input and output data. The need for modules has two historical causes. First, as programs got larger, it was necessary to divide the work among a number of programmers. Second, the impossibility of debugging a large, monolithic section of code suggested splitting it into more or less self-contained modules which permitted debugging. System software available in the past may also have contributed to the requirement for the modules' "clean" interfaces. Operating systems imposed low limits on the number of parameters which can be passed to a module, thus giving rise to the tradition for a clean interface, meaning a relative paucity of data flowing across this boundary.

An even older tradition, and one that goes beyond the computer programming field, is for top-down conceptual design. This is a procedure for transforming a computer program specification into code by systematic

steps. The first step is the system block diagram; it represents, at a high level of abstraction, the computer program which will ultimately result. The system at the top level is next fully defined to comprise a set of modules, which are assumed to exist. The sequencing conditions and data structures for these modules are also fully defined. Each module is then defined in terms of its submodules and the procedure repeated level by level until ultimate primitives are defined. Only at this point are the primitives and modules actually implemented in executable code. A disadvantage in this approach is that the program for any given component may be difficult to implement because of its interfacing data structures or because its submodules may be awkward to use.

Surprisingly little has been written on the subject of designing computer programs, aside from works on the above mentioned classical methods. The conclusions of one author,[1] summarized below from an article intended to illustrate the principles which should be taught in a program design course, are:

- Program construction consists of a sequence of refinement steps. Refinement of the description of program and data structures should proceed in parallel.
- A notation which is natural to the problem at hand should be used as long as possible before switching to a programming language. The latter should then exhibit basic features and structuring principles natural to the machine (author cites FORTRAN as outstanding for this purpose).
- Students must be taught to be conscious of the decisions involved in program design, to reject solutions, to weigh the various aspects of design alternatives, and to revoke earlier decisions if necessary.
- Careful programming is not a trivial subject.

From this and other writings it is impossible to descry a body of design theory and practice to guide the software designer.

The most outstanding author on the subject of computer program design is Professor E. W. Dijkstra of Technological University Eindhoven,

The Netherlands. In connection with his development of a multi-programming system in the mid 60's, Professor Dijkstra developed a concern for the matter of correctness in real time operating systems. This concern apparently lead him to exercise great care in the construction of THE, the Multi-programming system whose design he discusses at some length in reference 2. The essence of the difficulty apparently solved by Professor Dijkstra was the probabilistic and intermittent conflict of two sources of interrupts - the real time clock and the storage drum. His solution involved a partitioning of the design into modules exclusively occupied by an interrup source and between which the possibility of conflict was reduced to a limited set of possibilities. These possibilities were exhaustively exercised by the test procedure used by Professor Dijkstra. The design was evidently successful, but apart from recounting his experiences in developing the design, Professor Dijkstra has only a single conclusion to leave for the benefit of other designers: "It seems to be the designer's responsibility to construct his mechanism in such a way . . . that at each stage of the testing procedure the number of relevant test cases will be so small that he can try them all and that what is being tested will be so perspicuous that he will not have overlooked any situation."

Again, Professor Dijkstra in another paper[3] has described the use of semaphores in solving the real time interaction of two modules. While the semaphore is a useful invention (we don't know whether it should be attributed to Dijkstra or a predecessor) for designers of real time systems, it is his sole contribution to the art of design, in the reference. As Dijkstra himself recalls, the final version of the design came "straight out of the magician's hat," that is, directions necessary to retrace the path traversed by Dijkstra in arriving at his ultimate design are, in spite of his rather elegant exposition, apparently incommunicable.

Professor Dijkstra describes a design principle which seems to be of fundamental significance to program extendability:

> Any large program will exist during its life-time in a multitude of different versions, so that in composing a large program we are not so much concerned with a single program but with a whole family of related programs, containing alternative programs for the same job and/or similar programs for similar jobs. A program, therefore, should be conceived and understood as a member of a family; it should be so structured out of components that various members of this family, sharing components, do not only share the correctness demonstration of the shared components but also of the shared substructure.[4]

Insofar as these excerpts are indicative of the existence of a systematic design practice, one may conclude that it is largely an uncharted, somewhat personal and inchoate process.

3.2.3 Design - Implementation Phase

Once the conceptual design is complete, the process of implementation - of writing the program in some computer language - can begin. Programming has been viewed from the first (as it is today) as the formulation of appropriate instructions to "get everything done," the major problem being not to forget anything. Hence, the earliest implementation procedure was to code the first instruction, then the next, and so on until the program was complete. It must have soon become apparent that a large program written in this way lacks form, and having no structural features to support the development of a conceptual model in the mind of the hapless reader, it thus eludes his comprehension.

In the early days of programming, little thought was given to the effect of a program's structure on its communicability. Certainly computers don't "care" what the structure of their program is and can execute programs of arbitrary structure, if they are free of logical errors. However, computer programs cannot be certified as error-free either by testing or by algorithmic proof, and the burden of discovering their errors is the programmer's alone.

Perhaps an early approach to program correctness would have been the following. The correctness of a single statement can be judged by inspection; similarly the correctness of the following statement can be so judged. Hence by mathematical induction the entire program - which consists of a sequence of statements - can be inspected for correctness, one statement at a time. The existence of errors from the very earliest programs should have exposed the fallacy of this reasoning, which is that the correctness of a given statement depends in many cases on the context of statements which preceded it. The very length of programs permits a context far in excess of human ability to recall, and therefore proving the correctness of an individual statement requires feats of association and recall beyond the ability of any programmer.

The question then arises, "How can this context be supplied in a form concise enough to be within the retentive and perceptive powers of an individual, so that a program's correctness can be evaluated by inspection?" The first record of an answer to this question (although he claims the answer had been known for years) is in a 1968 communication[5] from Professor Dijkstra. In this he pointed out the GO TO statement (or arbitrary jump instruction) as a prime source of complexity which makes the written version of computer programs difficult to follow and understand. His explanation:

> . . . our intellectual powers are rather geared to master static relations . . . Our powers to visualize processes evolving in time are relatively poorly developed . . . We should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Dijkstra asserts that human intellect has no difficulty following a sequence, starting at the beginning and advancing one step

at a time until finished. He argues that if the sequence is a computer program limited to executable statements (either conditioned or unconditioned) and repetitive clauses (viz. "DO loops"), then a pointer to the step under consideration uniquely indentifies the progress through the program. The pointer corresponds to the human notion of "where we are" in the program; amenability to its use is the necessary condition for any program to be humanly comprehensible. If GO TO statements are allowed, no simple index will show how much of the program has been executed. Noting that the GO TO statement has been shown by Bohm and Jacopini[6] to be logically superfluous, Dijkstra concludes it should be avoided and that whatever statements are used, the resulting program should be indexable by simple, "programmer-independent" coordinates to be comprehensible.

Dr. H. D. Mills of the IBM Federal Systems Division further develops[7] Dijkstra's contribution to the comprehensibility of written programs. Continuing the injunction against arbitrary control jumps occasioned by GO TO statements, he permits a limited set of basic control structures, such as IF-THEN-ELSE statements, DO loops, CASE statements, DECISION tables, etc. The result is a program which can be read sequentially, although practical programs would be much too long to do this comfortably.

Dr. Mills deals with the problem of length by setting bounds on the maximum length of a program segment - normally one page. What constitutes a segment Dr. Mills leaves to the programmer's sense of proportion with the following words of advice:

> Imagine a hundred page PL/1 program written in GO TO - free code . . . begin a process, which we can repeat over and over until we get the whole program defined. This process is to formulate a one-page skeleton program which represents that hundred page program. We do this by selecting some of the most important lines of code in

> the original program and then filling in what lies between those lines by names. Each new name will refer to a new segment to be stored in a library and called by a macro facility. In this way, we produce a program segment with something under 50 lines, so that it will fit on one page. This program segment will be a mixture of control statements and macro calls with possibly a few initializing, file, or assignment statements as well.
>
> The programmer must use a sense of proportion and importance in identifying what is the forest and what are the trees out of this hundred page program.[8]

The matter of control paths linking the segments (pages) is handled carefully. First, it is clear from the preceding quotation that the segments form a hierarchy: each except the topmost "belongs to" some other segment one level more senior and, together with a group of peer segments, comprises that common parent. Then, control is allowed to enter a segment at the top only, and to exit from the bottom only. These two conditions guarantee that a segment can be read from top to bottom, and that any internal exits will be to an immediately subordinate segment, which will return control without detour. The reader can merely note the internal exit/return for later analysis; such a checkpoint makes it possible for him (Mills claims) to comprehend the entire segment.

Mills assumes traditional approaches such as modularity, the use of clean interfaces and top-down design to the conceptual design of programs. For implementation, he insists on a departure from tradition - to something called top-down programming. This practice, which avoids difficulties with simultaneous interfaces (Mills likens them to a theoretically uncomputable class of functions), is described as follows:

> . . . programs can be coded in such a way that every interface is defined initially and uniquely in the coding process itself, and referred to thereafter only in its previously coded form.

> In practical application, this . . . leads to "top-down" programming where code is generated in an execution precedence form. In this case, programmers write job control code first, then linkage editor code, then source code. The opposite (and typical implementation procedure) is "bottom up" programming, where source modules are written and unit tested to begin with, and later integrated into subsystems and, finally, systems. This latter integration process, in fact, tests the proposed solutions of simultaneous interface problems generated by lower level programming; and the problems of system integration and debugging arise from imperfections of these proposed solutions.[9]

Advantages claimed for the structured/top-down programming technique advanced by Dr. Mills are error-free code (or nearly so), and a program which can be well documented and effectively maintained. There has been but a single known application of the techniques. This was in the successful completion of a system for the morgue of the New York Times in spite of a severe curtailment of the schedule. The effect of Mills' programming conventions is obscured in that application however, by a novel organization of the work force, originated by Mills, which helped to accelerate the implementation. The results of this application will be discussed presently.

3.2.4 Formal Proofs of Correctness

Most of the foregoing discussion has been devoted to techniques concerned with computer program correctness. This bias accurately reflects reality, where program faults enjoy a major share of attention as the most disfiguring blemish on the programming profession's record of accomplishment. Structured programming, the principal new technique, aims at making the program comprehensible to a human reader, for verification. Human comprehension must be employed, because no computer-based technique operates well enough to do the job. However, there is active research on developing suitable proofs which can be mechanized, and some progress is being made.

Current research centers around proving the correctness of programmer-supplied annotations ("assertions") about key checkpoints throughout the program, and in particular at the end. The historic derivation of the notion of assertions is interesting and worth recounting. In the earliest approach to proofs of correctness, a computation was viewed as transforming the memory contents from some initial state through a succession of intermediate states to the final (and presumably the desired) one. Efforts to prove that each state implies its successor, however, were unsuccessful.

Then Naur[10] suggested preserving a record in symbolic form of the operations on each memory word, or variable. At any point in the program each variable could be evaluated from its then-current history of operations and compared with its value just arrived at by the program. This approach was restricted to key variables and applied only at strategic locations within the program, to make it less cumbersome. The resulting annotations scattered throughout the program were called assertions and had to be provided by the programmer, in the absence of algorithms to derive them.

After Floyd[11] showed how existing computer-generated proofs of statements in predicate calculus could be used to prove the correctness of a sequence of assertions, London began applying these techniques to programs of some size. Although the computer-generated proofs are crude, they were successfully used to prove the correctness of some programs which were operational, and not written simply to demonstrate the technique. The largest of these contains 433 Algol instructions[12] and performs interval arithmetic on the Burroughs B5500.

Although a few computer-based aids for using the techniques described above have been developed, the assertions must still be supplied by the programmer, constituting a great burden, and the techniques fail on programs of greater complexity. Because of these

limitations, automatic proof methods in their present form can contribute little to the production of programs for operational use. Liskov and Towster suggest that applying automatic proof methods to structured programs may accelerate the development of improved methods, because of the simpler structure of such programs. Therefore it is still a good idea to keep abreast of research in automatic program proving, as results of practical significance to programming of an industrial character may be produced at any time.

3.3 ORGANIZATION AND MANAGEMENT

Much has been written about organizing for the production of computer software. Large projects especially have been the subject of intensive effort to find the best organization for pursuing these projects. Definitive contributions in this area have been made under contract to the Department of Defense, whose procedures for the administration of large software projects currently embody these original contributions.[14] There is a likelihood, however, in large organizations that most of the experienced, senior programmers will be assigned to management jobs. As a result, the programs which are written are the work of the less experienced, while the more experienced programmers cannot apply their software skills to the job at hand. A large organization also requires much in the way of interpersonnel communication among its programmers. Exchange of information in this process is hindered by the lack of standards for program descriptions, making the communication channel a noisy one at best.

These two difficulties are addressed by an organization termed chief programmer teams, originated principally by Dr. H. D. Mills of IBM. The chief programmer teams concept assembles a team of specialists around a highly qualified senior programmer, comprising a group not unlike a surgical team. The senior programmer himself devises the nucleus of the system that is being programmed and assigns the remainder of the programming to his assistants. In this way his skills can be brought to bear directly on the design and coding of the programming system

being developed. The team concept also relieves programmers of clerical duties, such as key-punching and report writing, which are transferred to a librarian. The librarian, who maintains all system documentation in a current state, alone is permitted to communicate with the computer. This she does for the programmers. To make this possible, only a few standard computer functions are used, which a secretary can be taught in a couple of weeks to invoke as required. The library also serves as a medium of communication for programmers, who consult it for definitions of interfaces, program operating details, etc. It also constitutes ready-made documentation for the system. For writing code, the team employs the structured and top down programming techniques developed by Dr. Mills.

The chief programmer team concept has been put into practice by IBM in the development of a system for the New York Times.[15] During the course of approximately a year, this rather complex system was completed by a staff of eight programmers producing at above average levels. Few errors were found in the resulting system, and it was concluded that the combination of the team concept with structured and top down programming techniques was responsible for approximately doubling the productivity of the programmers.

These promising results are qualified by the author, who notes that application of the team concept to large scale systems, though feasible, has yet to be tried. Further, chief programmers of the caliber required are scarce, because they must have the qualities of a manager and at the same time be able to make significant technical contributions. In fact, all the personnel on the New York Times project were unusually high qualified. This appears to be a requirement for successful application of the chief programmer team concept to software developments of moderate or greater difficulty.

SECTION 4. STATEMENT OF THE PROBLEM

4.1 BASIC CAUSES

The existence of problems in the construction of programs has been alluded to in previous sections. In this section we will discuss characteristic programming deficiencies in terms of the root causes. First let us see what these causes are.

4.1.1 Lack of Theory

Unlike the physical sciences, computation is a practice largely unaided by formal, mathematically expressed relations. This lack hinders the design of computing processes, by depriving the designer of the means to trade off one parameter for another, while predicting the behavior of the computation process. The lack extends to an expression for the efficiency of the computation, with the consequence that it is impossible to distinguish except in a rough way between processes of lesser and greater efficiency. More important, there is no algorithm which when followed assures that the efficiency of a process will be increased. A result is the observed 20 to 1 variability in the efficiency.

of code produced by different programmers: lack of the algorithm prevents their convergence on equivalent processes of comparable size and speed, or even in comparable periods of time.

4.1.2 Lack of System Description Languages

Apart from informal languages, which arise more or less on an ad hoc basis, there is no reasonably concise and unambiguous language in general use to convey the meaning of computation processes among humans. Natural language, flow charts, and higher level programming languages are frequently used, but their use involves the possibility of misunderstanding.

Natural language is the medium generally used for communication with the user about his requirements. In such language, the danger of misunderstanding is great. Numerous examples of information systems which failed to answer the needs for which they were designed can be cited as proof of the need for a less ambiguous communications medium.

Language also has an effect on the thought processes of those who use it, and the particular design language employed by a computer systems engineer will influence the character of the design he produces. Dijkstra has roundly denounced FORTRAN as being inimical to constructive design thought.[1] It has been shown that a design team must first agree on a common language suited to its project before it can progress with the design.[2] Designers may use any agreed-upon subset of language, perhaps augmented by flow charts, for communication among themselves, but with the possibility of lacunae.

If a programming language is used to communicate design instructions to the coders, it may be sufficiently rigorous to prevent misunderstanding of the program which is to be produced.

4.1.3 Test of Correctness Impossible

A computing process can be viewed as a succession of machine states dictated by the input data. It has been shown that the number of possible input sequences, and hence the number of possible states, is so great that it would take tens or even hundreds of human life spans to demonstrate them all on a computer of practicable speed. While it is possible to test the logic flow of a program in finite time, demonstrating the correspondence of the output to that required is what would take impossibly long. This obstacle constitutes a gulf separating the design of computation processes from that of physical entities: no formal check can be made of the correctness of a design. Designers must use informal methods, at least until algorithms for formal proof are perfected. Currently, human intelligence is the only means available to check the correctness of programs. Programs must be concisely expressed (limited to one page or so as we have seen in Mills' structured programming) to remain within the limits of human understanding.

4.1.4 Programmer Psychology

This is not so much a lack as a characteristic of the programming process, which influences design structure. The psychology of programming teams has been little studied: we only know that it appears to be less amenable to conventional methods of organization and management than other developmental activities.

Programming and more especially systems design, is acknowledged to be a creative activity and attracts creative people. However, the lesser aspects of programming (the production of "dull" sections of code) do not appeal to the creative sense. In consequence, design functions tend to be distributed among all the programmers on the project as compensation for purely production coding, with resulting

lack of control over the design process. The lack of restraints on designers' inventiveness, such as a deadline or firm system goals, has been known to cause programming project failure.

Second, programmers identify with the code they produce, to the extent that errors in code tend to be glossed over by its inventor. Once the programmer's ego is divorced from his code, the errors become highly visible, and in fact "egoless programming" is a term which describes the practice of critical review of code by the programmer's peers.

4.1.5 Changing Understanding of the Problem

When a detailed information processing solution is applied to an incompletely understood problem, the resulting system is in for a lengthy period of development. This consists of solution by successive approximation in a series of iterations of the initial system design. In each, the designer learns more about the problem, asymptotically approaching the state of being adequately informed; several cycles of system redesign later, the solution is satisfactory. This process has been noted in management information systems where the problems are seldom understood by the prospective system owners; it was observed in the construction of early compilers, and probably has been with us from the beginning.

4.2 PRACTICAL DIFFICULTIES IN SYSTEM DEVELOPMENT

SAGE is regarded as the first large scale complex programming system; a thousand people were involved in its development. Based on a prototype system developed at MIT, the full size system should have required a reasonable number of people and time, but more effort - orders of magnitude more in fact - were needed. Various specialists were

required at all levels of the program. All of these specialists required managers, themselves at a variety of levels. The managers required help, both admistrative and technical in nature. As schedules tended to slip or difficulties be recognized, more people were hired which required more management (and more communication). This cycle continued for several years until many hundreds of people were involved in the programming effort. The program, considered by most people to be a landmark as well as one of the few successes in large scale system programming, nevertheless was delivered later than originally planned and with somewhat less capability. When asked what he would do differently if he had to do a system like SAGE again, the manager of SAGE development said, after some reflection, that he would hire twelve good people to do the whole job. Outside of that he couldn't think of much else that he would have done differently.[4]

4.2.1 Partitioning the Design Structure

The fact that SAGE was one of the largest programming efforts of all time doesn't mean that smaller programs are immune from similar difficulties. The use of conventional methods of organizing groups of people engaged in a development and production effort militates against a successful design, even of medium scale systems. As has already been mentioned, the design should flow through one head; that is, it should be comprehended by one designer in order that its correctness might be perceived. Hence the problem of how to partition the design such that many independent design groups can work on it simultaneously, arises in the development of any program which exceeds the ability of a single man to design.

4.2.2 Postponing or Evading Design Effort

Design also has been characterized as a series of decisions. Many thousands of decisions are taken in the design of an average system. These have to be based on knowledge: of the details of the rest of the design, of the problem which the system is being built to

solve, etc. Such knowledge takes time to acquire, and sufficient time is not always allowed in the large system development, because of schedule pressures.

4.2.3 Precipitate Coding

The pressure of a schedule and awareness that a great deal of coding has to be done has caused managers to commence work on coding just to get started on a job which is obviously huge. When combined with an organizational philosophy which puts coders at the bottom of the management structure, this hasty commencement of coding throughout the system leads to design difficulties. We recall that even at the coder level some design latitude is allowed as a compensation for the dullness of mere coding. Hence the process of design is commenced throughout the system at the very bottom level by the coders before the design has been properly thought out. A classical bottom-up design emerges, leading to difficulty in integrating the resulting components into a system, but its most serious drawback is that the resulting system design itself may be influenced by the existence of modules already coded.

4.2.4 The Organization Imprints the Design

It has been found that the organizational structure can also influence the structure of the emerging design in a large scale development. One of the men who participated in the development of OS 360 recalls that at the beginning of design work, a group was formed within the development organization for each of the functions considered to be important in the original design. When after a few months it was found that there were additional important functions, there was apparently no way that additional groups could be fit into the design organization.[5] The conclusion of course is that the design which emerged didn't give proper emphasis to the added functions. Conway[6] shows that this effect

is general, in that a software design structure will always pretty much conform to the structure of the organization which designed it. If this effect is unavoidable, the conclusion to be drawn is that serious thought should be given to establishing a final design structure before organizing to build it. Our impression, however, is that this is not generally done.

4.2.5 The Communications Burdon

Communication in a large organization is also difficult. As Conway remarks, the number of communications paths are approximately half the square of the number of persons in the organization. If the design is fragmented by an early commencement of coding and by the diffusion of design responsibilities all the way down to the ultimate coders, then the maze of paths becomes an insuperable obstacle to the flow of necessary design information. As a result, the designers tend to make their modules self-sufficient, which tendency implies overdesign and a consequent surfeit of code in the system.

4.2.6 Programmer Training and Selection

Software design principles are largely untaught in courses for programmers, or elsewhere. The burden of what is taught is how to use a programming language, with the implication that design ability is conferred with mastery of the language and consists simply of employing it correctly. It is generally acknowledged that programmer aptitude tests distinguish not between poor and good prospective programmers, but more nearly how these programmers will do in training or how easily they will learn programming. Although college degrees have been required for

programmer recruits, no correlation has been shown between the quality of programs produced and the amount of such education received, except in scientific programming which requires a knowledge of advanced mathematics. It has been acknowledged that the identifying chatacteristics of potentially good programmers have not yet been isolated.

4.2.7 Industrial Versus University Software Development

As was indicated, the prototype SAGE system was well designed, and there was every indication that its development could be repeated on a transcontinental scale with little loss in efficiency. The prototype was developed by a small, presumably highly intelligent group in a laboratory of one of the country's best engineering schools. Similarly, elegantly designed systems have been put together in academic computer centers in this country and abroad. These systems are probably little documented and lapse into disuse when their originators depart, but this doesn't alter the fact that well designed complex systems are frequently produced in academic circles. The difficulty seems to be in equaling these achievements in the industrial practice of programming, where large size introduces severe management and organization problems. A dearth of effective communication between academic and engineering information systems designers has been observed; bridging this gap may solve some of the problems noted. The striking difference between these two groups is in the level of education and the continuity of the constituent personnel; both characteristics are greater in academic development.

4.3 CHARACTERISTIC PROGRAMMING DEFICIENCIES

To the question, "what effects of the foregoing causes do software systems characteristically exhibit?" the answer is, Slipped schedules and/or cost overruns, performance short falls, errors or bugs, and difficulty in revising and maintaining programs. This is not to say that software systems invariably exhibit these deficiencies. Some of course do not, but if we were to summarize the typical errors that have been reported over the past ten years or so it appears that they would fall without exception into one of these four categories.

4.3.1 Schedule and Cost Difficulties

There are numerous stories of system development cost estimates that missed their marks by factors of two or three, and a factor of ten is not unknown. There are also examples of competent professional programmers' estimating the same job and differing by over 100%. Not all estimates are bad. Small well-defined assembly line programming jobs can be estimated closely by a competent and experienced estimator. On the other hand, estimates of mammoth computer program system developments - projects requiring hundreds of man years, thousands of computer hours, and millions of dollars - take on more of the character of a wild guess than of something resembling scientific prediction.

Most bad guesses fall short of actuality rather than overestimating the effort required. It is not uncommon for an estimator to go through his calculation and then add a contingency factor of 25, 50 or even a hundred percent. He is in effect saying that lots more is going to happen that he doesn't know about, so he estimates the rest as a percentage of what he does know something about. This points to a fundamental difficulty in the estimation of system development, which is that we don't understand what has to be estimated well enough to make accurate estimates.

It is relatively easy to estimate the amount of effort required to produce a set of programs - for example, program specifications, flow charting, coding, unit testing and so on. Even on these straight forward programs, cost and/schedule overruns are not unknown. These are due to the unknown character of the program design process. There are no work standards, by means of which the amount of work to be done can be transformed into the amount of programming effort required. Again, there is the problem of variability in the productivity of programmers which tends to throw estimates off.

Ancillary activities required in the production of a large system tend to be overlooked in estimating. These are system tests, data conversion, support functions such as documentation, test development, planning and controlling the development, hardware tests and so on. The larger the system the relatively greater importance these functions take on. In a large system development each task is dependent upon and influenced by many other tasks. What to a given programmer is a relatively simple process becomes in the large system development one of intimidating complexity.

The cost of large software developments is considered by some to be too high. For example the cost of support software for the Appolo project is estimated at a billion and a half dollars. Another system, OS 360, cost as much to develop as the System 360 hardware itself, or about ½ billion dollars. However, the absence of any standard against which to measure software development costs prevents an objective assessment of them. We can only point to inefficiencies in a particular development and speculate on what the costs would have been without them.

4.3.2 Performance

A program can fail to meet specified performance in terms of

speed of computation, amount of central storage required, specified mappings between input and output data, and operations desired by the user. Obstacles to meeting the last two performance specifications have been discussed previously; they account for the major part of the effort in demonstrating that a program meets its functional requirements. However, occasionally it also happens that a program exceeds the amount of core storage available in the target machine or performs much too slowly to be of use. The usual remedy for excessive storage requirements is an expansion of core resources of the machine or a dispersal of parts of the program onto auxiliary storage. In this way the failure of the program to meet storage specifications, bought at the price of additional investment in storage resources, is overcome. A slow program, however, usually cannot be fixed by increasing machine power, in which case it has to be redesigned. This process can range all the way from tuning to redesign-from-scratch; the relevant trade off is between operating speed and development time or cost.

4.3.3 Inflexibility

This term refers to a property of programs which makes them difficult to maintain or to modify. When it occurs, it is caused either by a lack of documentation or by the structure of the program. Writing documentation is, like producing mere code, considered a dull job by programmers. They prefer the more creative activity of design. As a result, documentation is frequently ignored or slighted. Then when it comes time to revise the program because of a change of input conditions or of machine, most often the persons who developed the program are employed elsewhere. The written description of the program is the only means of conveying an understanding of its structure to the programmers charged with maintaining it. When this documentation is deficient, the maintenance job is made correspondingly more difficult.

Even though a program be fully described by its documentation, it may resist change. For one thing, if program modules are highly interconnected, changes can ramify through them, and the task of understanding and controlling all the effects of even a superficial change can become horrendous.

Highly inventive or individualistic code can defy or discourage precise documentation. This type of program has an erratic logic flow which is hard to follow, reuses existing sections of code which "just happen" to do something that is required, contains many patches, etc. Another faulty programming practice is the building of dependencies which are not required by the problem specifications. Thus a given module may be so constructed that it relies on register values remaining unchanged, or on a specific sequence of modules being executed, or on a specific sequence of fields in a table. A given module may be more restrictive than the problem specifications in its assumptions regarding the range of values in a field, the upper limit on the size of a table or the volume of input. Such practices produce a fragile program which collapses under the stress of more varied input data or attempted modifications. These and similar practices elude such documentation as accompanies the program. The program will be employed while it works, but attempts at major repairs are foredoomed to failure because of its fragility.

4.3.4 Errors.

Program errors, whose effects are probably the most widely denounced of the computer's artifacts, stem from a mismatch between the programmer's circumspection and the computer's. When doing anything for the first time, a human being, lacking the ability to forsee many consequences of his actions, makes an initial attempt and waits for results on which to base a new attempt. When this method of working is applied to computer programming, the computer can be depended upon to

reveal a few mistakes to the programmer. The remainder are left as an exercise for the programmer or the unwary user to identify and cure. The symptoms may be a program loop, intermittent garbling of the output, or a complete collapse of the program; more rarely the computer will simply halt. While debugging tools such as traces and core dumps are available to assist in finding the cause of errors, diagnosis at present requires the application of human intelligence and cannot be performed by the computer.

The debugging process is one of tracing backwards in time from the observed difficulty to its origin by tracking the occurrences of unreasonable outputs in the record of a computation process. Since millions of computations, the results of which may be overlaid in the same section of core, can be completed in the time it takes a human observer to recognize difficulty, the cause may be obliterated. If not, the masses of data produced between the cause and the first observance of difficulty have to be reviewed by the programmer. This takes time, and hence debugging is a slow process.

Not all bugs can be eliminated in a large system. For example OS 360 had at last count a standing collection of 2,000 bugs.[8] Existing bugs in a program are simply avoided by programming around them until they can be corrected, an expedient that works well once the bugs have been discovered. OS 360's constant quota of errors indicates that they are being found as quickly as they are being eliminated, implying a rather large reserve of unlocated and unsuspected bugs. Commenting on the essentially infinite period of time required to locate all bugs, Turski[9] has proposed adoption of less than 100% as an error-free criterion for programs. Unfortunately, this will remain an unquantifiable notion while the total number of bugs is unknown.

We adopt a taxonomy of bugs which classifies them according to the point at which they are found in the life cycle of a program.

The main divisions of the resulting entomology are among intra-module, inter-module, data-caused, and in-service bugs.

- Intra-module. Bugs which occur during system development can be of a low level type which are discovered during the testing of modules. Among these are the array overwrite, the "off-by-one" group (off-by-one in indexing, off-by-one in shifting), and the operation irregularity bugs, introduced by computer arithmetic of only finite precision. Use of a higher level language for coding modules may prevent some of these bugs, but such languages invariably deprive the programmer of certain machine operations and their use should be foregone in novel or difficult programs.

- Inter-module. The inter-module bug shows up during initial tests of the joint operation of a group of modules which have been designed, coded, and tested independently. The key difficulty is in achieving an interface among several of the modules. Each module places constraints on the variables which must cross the interface. As the modules have been designed independently, the constraints may conflict, leading to redesign work in one or many of the modules until the interface is achieved.

- Data-caused Errors. We have assumed that, up to this point, modules are tested individually and in combinations in the absence of any input data. That is, the logical paths in the program have been tested, but the algorithms themselves have not. Data driven bugs of particular interest are those in a real-time system which are caused by particular combinations of timing and data and hence are intermittent. Such bugs are difficult to eradicate because of their unpredictable occurrence. In the sense that the variable, time, is a data input, data driven bugs include time-simultaneous (and hence, conflicting) claims for computer resources by various processes. Dijkstra has revealed some of the difficulties of the intellectual process necessary to prevent conflicts of this sort in a note on the design of THE, a multi-programming system.[10]

- In-service Errors. After the designers of a system have reasonably assured themselves that the system is error free, it invariably happens that further bugs are

found once the system has been turned over to the users. In effect, the many diverse and perhaps unanticipated uses of the system continue to disclose the existence of bugs. The larger the system, the longer this process goes on.

The manner of using a system can actually contribute to errors or reveal unsuspected difficulties inherent in the design. This fact is sometimes perceived by users as program "fatigue". The problem of program fatigue may be explored from four points of view: residual data (noise), saturation, entropy (disorder), and equipment strain. We shall assume that our tired program is not suffering from errors of program design or of implementation. This exclusion is done to better isolate the problem of fatigue from other categories causative of program performance deterioration.

- Residual data (noise). At program startup, data sets are initialized to values which are highly relevant to current system operation. As time goes on, some of these data sets may lose some of their relevancy. Entries may be retained in program directories which index data sets no longer in use; queues for devices, communication channels, etc., may contain entries whose timeliness is past. These conditions may reflect themselves in sluggish system performance. An example of residual data may occur in an information system or inventory system where reports cease for one reason or another, but the inventory item is retained in the active file anyway.

- Saturation. Programs, if driven long enough and hard enough, will saturate. Variable length tables achieve their limits, queues fill up, and counters overflow. It is common to design a program to run near its saturation point, in the interest of achieving maximum use of the computer. On the other hand, such design carries with it the requirement to recognize saturation when it occurs and to handle it, usually by regulating the load. Measures normally taken are to throttle input and eliminate low priority tasks, etc. during the stress period. When the delay or even delete measures taken are inadequate, the result may be

a breakdown in performance. Often the breakdown is temporary, with the result that the need to correct the problem is overlooked.

- Entropy. A parallel to a law of thermodynamics suggests that in time a system runs down, loses information, and increases its noise. Certainly weakly designed systems do exhibit increasing entropy effects. The following are common causes of increasing entropy in computer systems:

  (1) Error Propagation. Programs are subject to propagated error - error which is passed from its originator program unit to and through a sequence of other program units. The problem is particularly keen in programs operating within closed (feedback) loops. Unless the propagation errors are sufficiently damped, they may grow out of control with time.

  (2) Disorder. Programs often depend for good performance upon a favorable arrangement of data in storage. Quite often, the program design does nothing overt to guarantee that favorable arrangement, set up at program initiation, will more or less be retained throughout the course of program execution. When over the course of time the data evolves into an unfavorable arrangement, the program performance may suffer. A familiar example occurs when data is stored on a moving arm disk over a large number of bands. Initially, the data is located on the first few bands, and data access delays due to arm movement are small. Later in time, the data spreads randomly over many bands, with the result that average disk data access time becomes long.

  (3) Priorities. Programs that depend upon a priority structure may slow down or degrade when priority differentiation breaks down. Priority breakdown may occur when program activities of the same priority cluster in time. In dynamic priority systems, normally low priority program activities may be upgraded for one reason or another, causing the troublesome clusters.

(4) Iatrogenic errors. These are introduced unknowingly by the maintenance programmer while making needed corrections. An increase in the incidence of program errors and of activity to repair them results. When the iatrogenic errors respond to comparatively minor repair, the effect tends to be self-dampening.

SECTION 5. DESCRIPTION OF THE PROGRAM

Those attending the international software engineering conferences raised many problems, aired many opinions, and presented many excellent ideas for solutions to some of the problems. However, these opinions and ideas, many of which had actually been successfully applied, had been generated as means to specific and generally limited ends. The primary goal for virtually all of these attendees is that of producing software to perform, in a reliable manner, the functions intended by them or their clients. The problems attendant on such production and their solution are therefore, by definition, of secondary importance. This implies a lack of comprehensive, systematic and sustained efforts to solve the overall problems that beset the software-producing industry; and upon closer examination of the industry, such effort is indeed missing.

The proposed program is specifically advanced to fill this gap; its sole objective is to provide graphic and verbal languages, procedures, constructs, models, organizations, documentation, specifications, job descriptions, test plans and various kinds of standards. That such an effort is needed is attested to by both the title and existence of the

Software Engineering Symposia. They comprise a recognition by the leaders in the industry that an engineering-like approach is essential to the vigorous growth of the software industry. At the same time, the difficulty of achieving such an approach, because of the nature of the end-product, is also recognized. Thus, the proposed program.

A program of relatively complex structure, sustained duration, and evolutionary nature, in which the exact nature of the final products cannot be determined, can best be kept on-track by laying out its goals and objectives. The program's status at any point, and the thrust of its current activities, can be compared with the stated goals and objectives, to assure that the program is still on the track. A suggested set of such goals and objectives is provided in a later section. In another section, we shall define some of the words and phrases used. In still later sections, the requirements of the program will be outlined. The organization and its composition will be presented. The long-range program plan will be explained, and the functions, operations, and interactions of the several participants will be developed.

5.1 INTRODUCTION

The ultimate purpose of the proposed program is to make possible the production of software in NASA within predictable schedule and budget constraints, with major characteristics of the product - size, run time, correctness - predictable within reasonable tolerances. At the moment, such predictability is confined to relatively small programs; the delivery time and budgets of large, complex programs can be off by an order of magnitude or more, while the performance of the product can be deficient in important ways. The situation has been called a crisis by many who should know; it is getting worse rather than better.

This situation is in sharp contrast to that on the computer hardware side. By definition, industries for stored-program computers --software and hardware--were conceived and born simultaneously

and have evolved concurrently. However, in computer hardware, costs have dropped by as much as several orders of magnitude, speeds have increased, size, capicity, and complexity have increased substantially, and reliability has increased in spite of the foregoing facts. Predictability of hardware costs, schedules, and of hardware characteristics has kept pace with the maturing of the hardware industry. Many extremely useful standards have been firmly established at many system levels, and have been widely adopted.

Part of the reason for this contrast can be attributed to the nature of the product. This part of the reason is non-trivial, but so is the other part: the basic approach used by the producers in each case. In the case of hardware products, the producers from the beginning included the engineers for design on the one hand and the electronics and components industry for fabrication on the other. The end product is a tangible artifact, generally (but not necessarily) produced in large quantities. Separate designs are required for prototypes and for quantity production.

In the case of software, the essence of the end product is not tangible. The product is a stored program, which consists of the status of a multitude of (generally bi-stable state) storage devices. This product, the software, in combination with the corresponding hardware product, a computer (program, memory, I/O), provide the user with an operational combination capable of doing (presumably) useful things when the "start" button is depressed. The producers of this product were not engineers, but in the beginning were mathematicians, and somewhat later, accountants, and still later, specially trained personnel at increasingly lower levels of general intelligence, aptitude and competence. Thus, the producers of software have come, since the beginning, from many diverse backgrounds, with a wide diversity of approaches to the task of producing software. This diversity still exists.

The difference in the nature of the product then, is that in the case of hardware, the steps beyond the development of an explicit verbal and graphical description (preliminary design) by engineers include:

1. Fabrication of a prototype (or model)

2. Development of a production design

3. Fabrication of a multiplicity of identical copies of the tangible product.

In the case of software, the steps byond the development of an explicit verbal and graphical description (source language program) include:

1. Converting the program to machine-readable form (human operation)

2. Translating the human language (source program) to computer language (object program)

3. Loading the program into computer memory (at which time it becomes the final end-product, the stored program, ready to do its job in combination with the hardware and any executive software).

In both instances, tests are performed to establish that structure and performance of the interim and end products are in accordance with sponsor requirements. The difference in the nature of the tests is a reflection of the difference in the nature of the product. In the case of hardware, the product can be - and is - tested in two inherently different ways: (1) the structure and physical characteristics of the end product are tested against the engineer's description of the product to assure that the fabrication is in complete agreement with the design and that the engineer's intentions are realized; (2) the operation and functioning of the end product is tested within the context of the user's total operation to demonstrate that the design as realized in the fabrication is completely responsive to his expectations and that it "fits" within his operation.

In the case of software, the end product has no physical characteristics of its own - only those of its container - a storage medium. Roughly, this storage medium has the same relationship to a program that a computer room or shelter has to a central processing unit.

For all practical purposes, this end product, the stored program, is generally considered to be identical to the source-language program prepared by the programmer. Thus, the equivalent of the design/fabrication comparison, a structural test, cannot be performed. Substituted is a functional test or series of tests to demonstrate that the program will perform as intended within an environment, hardware and software, that approaches increasingly closely the user's total operation.

This lack of the equivalent of a structural test is a very important deficiency, as far as the production of a properly-functioning end-product is concerned. It is important because the deficiency denies the use of intuition, experience and physical perception to detect structural anomalies that are responsible for product malfunctions. The significance of this deficiency can be grasped by considering the difference between a road map and verbal directions as a guide to get from "here to there." In the absence of a map, the only sure way to test the correctness of the directions is to follow them exactly. If the directions are not correct, there is no obvious way of correcting an error. If it is correct, there is still no obvious way of establishing other and perhaps better routes. On the other hand, with a map all possible ways are evident; topological anomalies are instantly detected, and it can be checked, point by point, for physical inaccuracies. When it is structurally accurate, any set of correct directions can be drawn from the map, with positive assurance that all of them will get the traveler "from here to there," and an optimum way can be determined. If some roads are known to be blocked, alternative routes can be used.

The importance of this deficiency in testing software is based on the difference between the enormous amount of information in a graphic representation, which can be perceived visually and effectively processed in parallel at very high speed, and the relatively small amount of information in a verbal representation. The written material must be read; that is, acquired serially, stored in human memory in small groups, and interpreted and integrated with previously read and stored information, all at relatively low speed.. This difference is epitomized by the graphic

representation of "impossible objects" of Figure 1: how long would it take to establish beyond reasonable doubt that the verbal equivalents of these very real two-dimensional "structures" represented three-dimensional non-structures? And what levels of education and intelligence would it take?

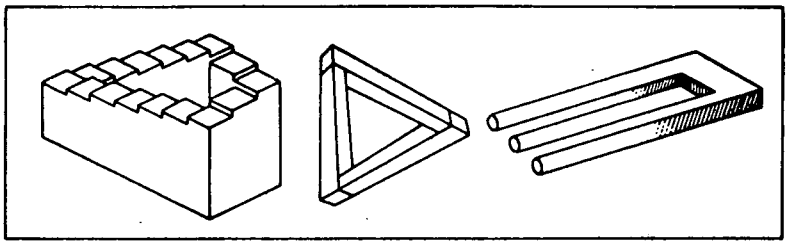


FIGURE 1. IMPOSSIBLE OBJECTS*

The reason that a traditional engineering approach was not taken to software fabrication is thus quite clear: a means of representing the structure of a complex computer program was, and still is, an elusive matter. However, the importance of such a structural representation to achieve program correctness also became quite clear as computers and memories and peripherals and programs became increasingly larger and more complex. Thus, the need for an engineering approach to software construction became increasingly evident to the leaders in the field, stimulating the international Software Engineering Conferences. [1, 2]

These conferences were primarily concerned with the problem of software production, with the primary objective being the exchange of opinion and individual approaches and attempts at solution of those problems. The conferences did not have as an objective any systematic attempt at solution of the problems. The attendees--some fifty or sixty persons of international reputation in the software field--were engaged primarily in the production of software, and not in the solving of software production problems. These were addressed only as necessary to achieve the specific ends of each participant.

*Due to Richard Gregory, Cambridge University, and reprinted from The Observer, London.

Thus, while the Software Engineering Conferences did much to define the basic problems and to point the way to their overall solution, they did little or nothing to set in motion any systematic or dedicated effort toward such solution.

The present proposed effort, in effect, picks up where these conferences left off: the formation of a group dedicated to solve the problems of software production over an extended period of time. The planning, launching and execution of such an effort requires both the resources and the promise of large payoff that apply only to an organization of the size and scope of NASA. The autonomous nature of the organizational entities of NASA will assure that the success of such an effort will depend on its merits rather than on the authority of its sponsor, NASA Headquarters.

It should be noted at this point that whereas the conferences were concerned with "Software Engineering," the subject dealt with here is "Software Technology." Engineering is a design and planning activity; technology includes the total body of information on which such activity depends: standards, languages, techniques, measurements. Also, the conferences referred to the final stage of software development as "production: which carries a distinct connotation of quantity production. The term "fabrication" has been used here to retain the concept of physical realization based upon an abstract design, while avoiding the notion of replication or quantity. The idea of hand-crafting one of a kind, so much an essential aspect of programming, is conveyed by "fabrication" whereas it is lost in "production." The point here is that "design" and "fabrication" and "test" are three distinct stages in producing an end-product which must work -- which must operate to perform some explicit, planned function. The implication of these three distinct steps is lost in the term "programming," and the term "engineering" needs to be qualified to correspond: Design Engineering, Production Engineering and Test Engineering.

Thus, "Software Design Engineering" can be conceived to cover interpretation of requirements and creation of a design, but not (in the tangible world) the actual fabrication of the product. If a structural representation can ever be successfully developed, then "Software Design Engineering" will be a meaningful term applied to the activity of a corps of design specialists, and the realization of this structural/representation in a stored program ready to work can be considered as the fabrication, executed by a corresponding corps of fabrication specialists. In such a technology, the software design engineers will complete the design to perform the functions the user requires; production engineers will plan the flow of work and assembly; and test engineers will specify the test and test set-ups required to assure that fabrication corresponds to design, and maintenance engineers will be responsible for maintenance and service. The design engineers will be responsible for the design being responsive to user operating needs. The fabrication (ideally) will be accomplished by software fabricators, who will be responsible (ideally) only to make the stored program or system of programs responsive to the design.

In this concept, the design engineer, as any engineer, must make the compromise between the limited resources available and the highest possible "factor of safety," or reliability, consistent with the needs of the user. The production engineer need not decide; his only responsibility is to make sure that the software can be assembled into the system and that it will pass the tests designed by the test engineer. If it does that, then whether or not it satisfies the customers requirement is not his problem, but the design engineers'.

When that point is reached, we shall have a true software technology, It is the purpose of this proposed effort to try to establish such a software technology. To this end, a dedicated, sustained development effort is proposed: dedicated in the sense that the technology group will not be responsible for other missions, such as programming; sustained in the sense that the technology will be tested by a software group on successively more complex programs, and that the using groups will be monitored in the operation of the completed programs. Feedback from the software and user groups will be used to modify the technology.

5.2 DEFINITION OF TERMS AND PHRASES

The general approach taken in setting up the proposed program is that of the most recent authoritative group--the International Software Engineering Symposia at Garmisch, Germany in October of 1968 and its follow-up at Rome, Italy in October of 1969. This approach is implied by the title, which is that producing and maintaining software is an industry rather than one of the humanities. It would be in order to use as a model the computer hardware industry, especially since the trade-off that has always existed between hardware and software implies that the industries of producing hardware and software ought to cooperate much more closely than they have been. The terms that we will define here are thus drawn from both industry in general and programming in particular. We provide them to preclude misinterpretation and misunderstanding. In most cases, we have avoided the use of jargon, preferring to use ordinary words with explicit and specialized meanings that are consistent with generic meanings.

In the previous section, we defined Software Technology in terms of its major component parts: Software Design, Software Fabrication, Software Test and Software Maintenance; each of these parts included an engineering specialty.

In this section, the relationships of Software Technology with Computer Technology and Computer Science will be briefly developed. These latter two terms are only a few of the many used rather loosely to describe various aspects of the practice and theory of computer and information systems.

Computer Technology, in this document, will be an extension of Software Technology to the areas of hardware and microprogramming read-only memories (ROMs), or firmware. Thus, specialties corresponding to design, fabrication, test and maintenance stages may be defined for firmware; they are already well-defined for hardware. As the use of microprogramming increases, a system design effort will more and more be

concerned with trade-offs between hardware, firmware and software, and the need for a definitive structural representation of the whole system comprising these three will become increasingly evident. This system design effort, and the specialties needed for it and overall system assembly, System test and system maintenance, will be accomplished by Computer System Engineers. Software Design Engineers will pick up where the Computer System Engineers leave off -- by designing the software modules defined in system-level specifications to lower levels of detail (still in structural terms) as routines, subroutines and programs.

The relationship between Science and Technology is the same as that between theory and practice. Thus, we look for Computer System and Software Design Engineers to make maximum use, for practical ends, of what the theorists in Académe develop in their research. Where theory and explicit knowledge are inadequate, then, like engineers in any field, they will have to substitute empirical knowledge.

The chances are good that the technology group will be able to identify some areas where theory and research are needed. Such requirements can be defined and explained in detail; in this way, applied research in Computer Science may be engendered. The theory produced as a result of these effects will be available for experimental applications to practical problems; where such experimentation succeeds, the theory will become practice and a part of Software Technology.

On the practical side, the technology group will be able to identify areas where engineering-type standards might profitably be established; preliminary ideas on such standards may be described and lead to a definitive study and formation of an industry committee. When refined and approved by an authoritive industry group, such standards can be issued formally, and become part of Software Technology. The foregoing relationships are shown in Figure 2. The relationship between the three groups and NASA management, illustrated in Figure 2, is discussed in Section 6.1.

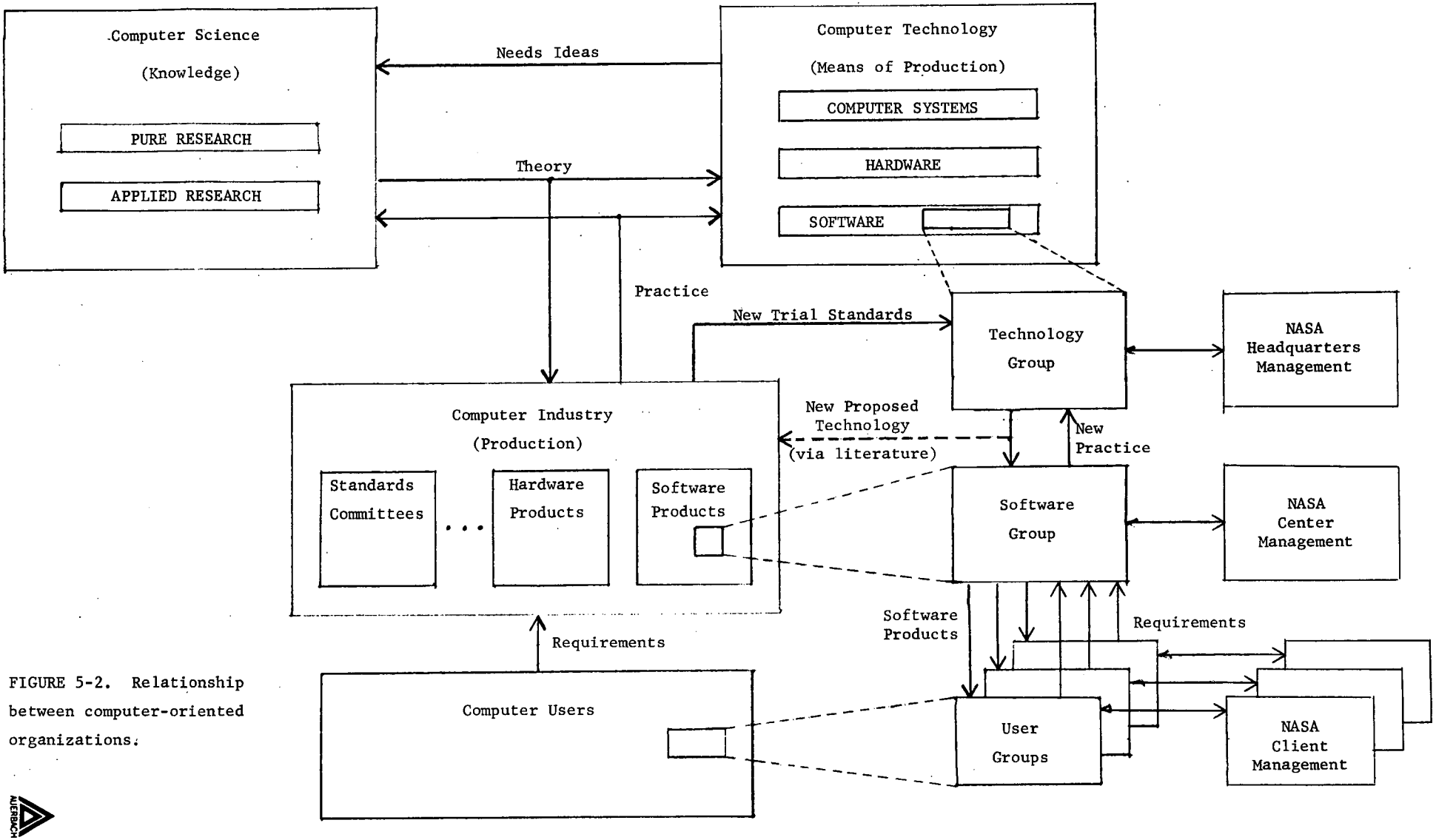


FIGURE 5-2. Relationship between computer-oriented organizations.

The definitions that follow have the primary purpose of clarifying subsequent discussion. It should be recognized that formal definitions and job descriptions will be an important part of the results of the proposed program. In that light, the following may be viewed as representative of the kinds of things to be expected. Definitions of jobs, and position descriptions are also presented in Section 6.

1. Computer Science. A department of systematized knowledge having stored-program computers, computer programs, storage and peripheral devices, computer languages and their relationships and operations as objects of study. The object of a computer scientist is to increase his own and human knowledge of computer structure and operation and to improve thereon.

2. Computer Technology. The totality of the means employed for producing stored-program computers, computer programs, storage and peripheral devices and combinations of the same necessary for application to human needs. Practitioners of Computer Technology will keep up with the work of Computer Scientists, and apply the results to produce computer products, including hardware and software.

   Computer Systems Technology comprises Hardware Technology and Software Technology.

3. Software Technology. The means for producing computer software and firmware, exclusive of the storage media used to store the programs and microprograms.

   Software Technology includes the means employed for producing Software Designs, performing Software Fabrications, executing Software Tests, performing Software Maintenance. Each of these activities has an aspect of Engineering associated with it. These are: Software Design Engineering, Production Engineering, Test Engineering and Maintenance Engineering.

4. Software Design. The act of converting a user's requirements for an automated operation to a design for the structure of his input data and a design for the mechanism (program, computer system software, storage and peripherals) to transform the input data into the required output data.

5. Software Fabrication. This is the art of transforming the structural design of the program, including the structure of the data base and input data, and the program components and their hierarchical inter-relationships, into detail source language code, and assembling these routines with standard and previously tested built-to-order modules into increasingly higher levels of assemblies in accordance with the structural design.

6. Software Test. The act of testing software. Software tests are of two kinds: (1) hierarchical tests of modules, to assure proper relationships and functioning of the internal structure and interfaces of the module or assembly, including entry to and return from subroutines; and (2) sequential tests of modules to assure proper interaction and sequencing between modules at the same structural level and their interfaces, linkages, and parameter and data passing. Tests may involve use of simulation or emulation of certain hardware or software system components, preparation of data base, and data inputs.

7. Software Maintenance. The action required to find and correct software errors found after installation of software at the user's site. It may be established that "Software Maintenance" is rather inseparable from Computer System Maintenance for

8. Preliminary Software Design. The collection of documents, graphical and verbal, used by the software design engineer to represent the user's requirements, the structure of his (existing or planned) data base, data input forms or mechanisms, the operation that he wants to automate (flowchart showing data sets) including actions of operating personnel. Will include a preliminary graphical and verbal description of the structure of the data base, the hardware, system software, and software envisioned to do the user's operation, and a flow chart of the proposed user's operations showing actions and steps of operating personnel using proposed system and the data base. (Conceptual Design or Base Line Configuration.) Flow charts describe user operations, not computer operations.

5.3 GOALS AND OBJECTIVES

A program of the sort proposed is a natural for NASA, because of its size, mission and organization. It takes a large organization to have a sufficiently great vested interest in such general and long-range goals as those proposed for this program. A small organization simply cannot afford to take a global or long-range view; satisfying immediate needs is all it can afford, and generally this is sufficient. Such a situation applies to most of the parent organizations of the attendees of the Software Engineering Symposia.

The mission of NASA involves a truly incredible array of computing power, from the smallest computer to the very largest complex, from the slowest processor and memory to the fastest, and from the most accessible to the most remote. It also involves an unprecedented array of applications. Altogether, there is little in hardware, software, or application that is not represented in a significant way in NASA centers or by NASA users.

The organization of NASA is uniquely appropriate in that the source and mechanism for the special funding and subsidies that may be required exist, and at the same time the autonomy of individual centers and users is such that the program can proceed with a maximum of freedom and virtually no bias or explicit technical direction from the top. There is enough variation in software development practice that objective criticism can be expected. The fact that it is a government organization, at the same time a user of enormous size and influence, and one with clearly no vested interest in specific hardware or software producers is also significant.

The goals and objectives of such a large and diverse organization will be sufficiently global and general that no effort need be made to keep it from having a uniquely NASA flavor. Conversely, there will be no difficulty in interpreting goals and objectives stated in general terms to specific NASA or center interests. Thus, the goals and objectives that follow are general.

5.3.1 Goals

The long range goals of the proposed program, that is, of the Software Technology it is designed to establish, are as follows:

1. To fulfill the user's requirements and expectations in the end product with respect to usability, usefulness, cost and time;

2. To produce an end product satisfying goal 1 and having predictable characteristics such as modularity, size, run-time, response time, numerical resolution, and correctness;

3. To produce an end product which makes measurably efficient use of available resources both in the process of its production and in its structure and operation; and

4. To establish quantifiable parameters for describing the properties of computer software and firmware, develop means for measuring the value of these parameters in specific instances, and develop procedures for applying these techniques in assuring goals 1, 2, and 3.

These qualitative and general statements of intent can be broken down into more detailed objectives. These are defined in the next section.

5.3.2 Objectives

The objectives stated below start with the perspective of an entire computer system, and then consider hardware and software individually. Actually, the program in the very long run includes a gradual expansion of scope to include firmware and hardware. It could, of course, include data transmission and communications at some point and to some extent and depth best determined by those involved in the program.

Additional objectives could be defined. More detailed objectives touching on explicit design, fabrication, and test procedures could, for example, be added. Those listed below will be sufficient for the present purpose.

1. The user will be able to describe his functional, procedural and data problems to a computer systems engineer who will express them explicitly and rigorously in documentation comprehensible to the user or his agent.

2. The computer systems engineer will be able to translate the functional, procedural and data aspects of the user's problem into structural terms using standard verbal and graphical languages and appropriate measurements.

3. An arbitrarily selected computer systems engineer of established reputation and competence will be able to review the planned structure of the proposed computer system, hardware, software and firmware, and certify its structural integrity; and examine the functional, procedural, and data descriptions, and certify that the planned structure and data sources will be adequate to accomplish the required functions and procedures. (Verify preliminary design.)

4. Computer Design Engineers of various specialties (hardware, software, firmware) at successively lower levels will be able to generate designs and/or specifications to correspondingly lower levels of detail, using standard "parts" wherever possible.

5. Computer Engineers and Technicians of various categories and levels will be able to schedule, fabricate and test individual modules, and assemble and test them in successively higher levels of structural and functional assemblies. (This applies separately and collectively to hardware, software, and firmware.)

6. It will be possible to include in the designs and specifications at all levels any values of various numerical parameters: for each component part, the manhours and elapsed time to design, fabricate and test; and for each testable component, performance measurements that can be traced back through the structural hierarchy to the user's requirement: execute time, response time, propagate time, throughput for various defined initial load conditions.

7. The product at any stage of completion (including designs and specifications) can be measured and meaningfully compared quantitatively with the requirements and design parameters of higher levels.

8. It will be possible to establish procedures for checking and approving components and assemblies at all stages of design and fabrication; the object will be to permit establishing responsibility and accountability for deficiencies or errors.

9. The establishment of positions of defined responsibility and defined procedures and standards will make it possible to establish well-structured general and special organizations capable of exerting effective management control upon projects and upon their funding and scheduling.

The way in which the program is organized to achieve these objectives will be discussed in the succeeding sections.

5.4 THE BASIC REQUIREMENTS FOR THE ORGANIZATION AND OPERATION OF THE PROGRAM

In the preceeding section we examined the characteristics of NASA that made it particularly suited to undertake a program of the sort proposed. One of these characteristics was that of its lack of vested interest in hardware or software or methods; that is, its objectivity.

It is important that this objectivity be preserved in the program itself. For that reason, the group charged with the basic mission of developing the technology will be dedicated to that end; it will be a completely separate group, not responsible itself for turning out software designs. Now the program cannot really be objective if it fails to take into consideration the effect of its work. Thus, it is necessary to do much more than simply develop a technology comprising the planning of an organization to design and fabricate software, together with functional descriptions of its staff members, and descriptions of procedures, languages, graphics, documents, etc. The plan must be tested: a software development group must be formed and set into action to build some real software in response to a real problem; the initial technology will be modified as a result of this experience.

The user, too, must be taken into consideration at all stages of system development. In addition, the software group that will be responsible for applying the technology to produce real software whould not be depended upon to judge the quality of its product, the software.

This function will be performed by those uniquely equipped for it--the users. Thus, the responsibilities for developing the technology, for applying it, and for judging its results are separated. In this way, maximum objectivity will be assured in those areas where it counts.

Finally, provision is made for the program to evolve the technology by imposing software problems of increasing difficulty, complexity and diversity. Each of these will comprise an iteration or cycle consisting of three basic phases. The output of each cycle will serve as input to the next cycle. This output will comprise all of the technology documents: organization; job descriptions; procedures; languages for requirements, preliminary design, specifications, test plans; standards for interfaces (engineering standards), "parts" or subroutines (product and component standards), graphics (language standards), production (quality standards), group and individual performance (process standards); estimating methods, documentation, etc. These are discussed more fully in Section 6.2 and 6.3.

The underlying philosophy is that the Technology Group will employ both scientific and engineering methods to "design" and "build," over an extended period of time and in an evolutionary fashion, an organization--a Software Group--whose function it is to design, fabricate, and test and maintain computer software and firmware.

The approach will be to use industrial and engineering practice of a mature industry as a model; for example, the intimately related and historically contemporaneous computer hardware industry.

The specific model for the Technology Group might well be an organization that designs, fabricates, tests and maintains prototype hardware. Software is intrinsically different from hardware: there is no software industrial activity analogous to quantity hardware production. (Replication of tapes and cards is trivial and not analogous.) Consequently, there is no software engineering activity analogous to the design of a production unit based upon prototype and suited to quantity

production. It is for this reason that we suggest design and fabrication of a hardware prototype as a model for the program rather than design and fabrication of a mass produced item. It is also the reason that we use the words "fabricate" and "fabrication" rather than the words "produce" and "production."

This approach -- that software development is analogous to the design and fabrication of a hardware prototype -- can be used to examine the psychology and approach to their work of system analysts, programmers, computer software scientists and engineers -- whatever they may call themselves. We recommend it, at least as a take-off point for the program.

In the next section we shall examine more closely the organization and operations of the proposed program.

5.5 THE ORGANIZATION AND OPERATION OF THE PROPOSED PROGRAM

The basic requirements for the program are (1) that it provide not only for developing the technology, but for applying, testing, and evaluating the results as well; (2) that the responsibilities for development, application and evaluation be assigned to separate groups; and (3) that the development be evolutionary, that is, that the technology be applied to successively larger and more complex problems, and modified and improved after each application. There is further the longer-range desirability of merging the software technology with that of computer hardware, computer systems, communications systems, and information systems.

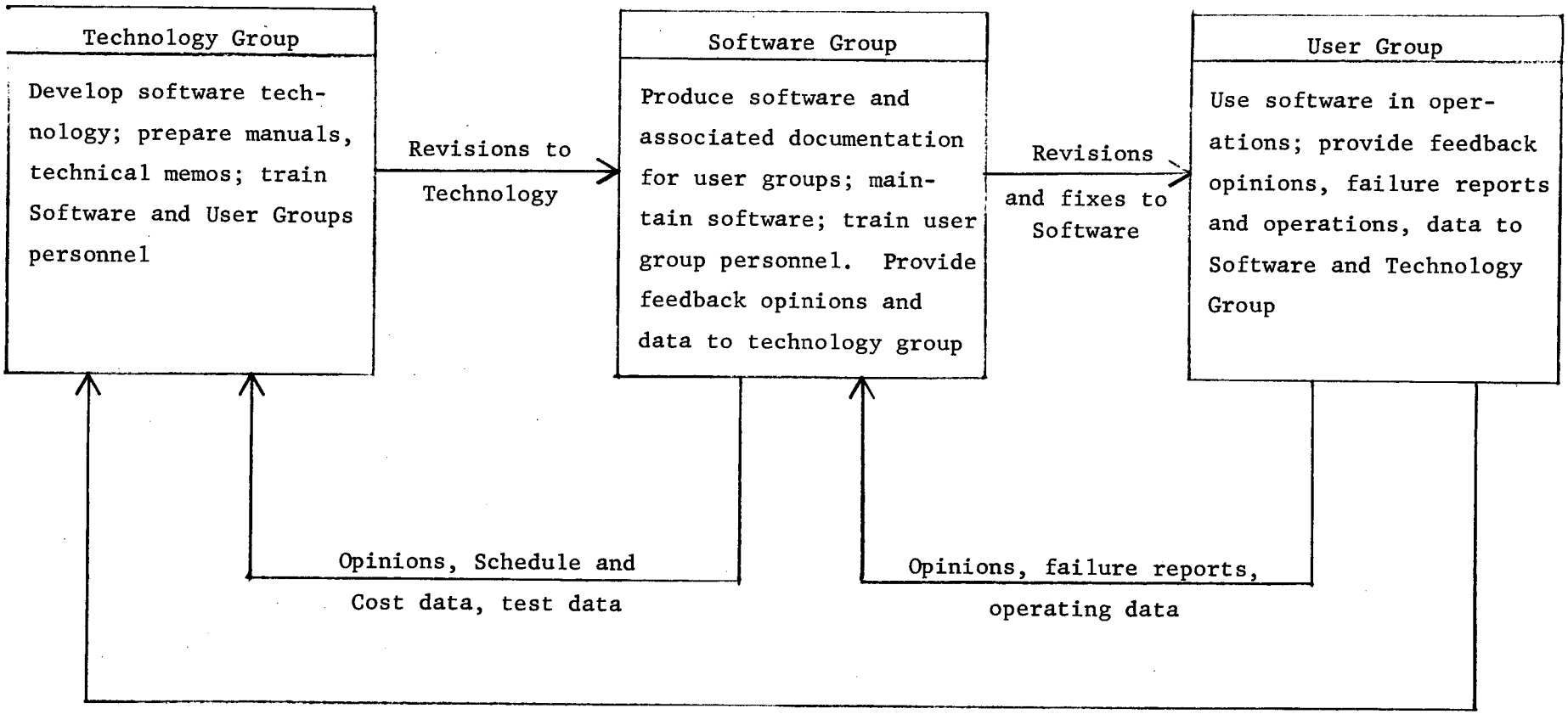
The responsibilities for development and application of the technology and of evaluation of the results will be assigned to three separate groups which we shall call, respectively, the Technology Group, the Software Group, and the User Group. These will be discussed in more detail in the next section.

Each of the succession of applications of the technology will be called a cycle. A new software development problem will be undertaken in each cycle, the nature of which will be determined at the conclusion of the preceding cycle. Each cycle will be divided into three major phases: Phase A, in which the Technology Group will amend or modify the technology as a result of the previous cycle, and select a specific software development project from among those coming up to test the new version of the technology; Phase B, in which the Software Group applies the new version of the technology and develops the software package selected by the Technology Group; and Phase C, in which the User Group will operate with the new software which will have been developed for them. These phases are discussed in more detail in the next section.

5.5.1 The Participating Groups

Although the three participating groups will be separate and will probably even lie within separate higher-level organizations, it is essential that they cooperate fully and coordinate on pretty much a continuous basis. This is shown in Figure 5-3. Aside from the fact that both relative independence and a community of interest between the three groups can exist at the same time (because of the overall NASA sponsorship), little can be said at this time about formal organization. It is possible, however, to say something about the general nature of each group.

The Technology Group. We suggest that this group be sponsored and funded directly by NASA Headquarters. In this way, the long range plans and policies of NASA will be able to influence directly the development of a software technology responsive to NASA needs. The initial



Technology Group: Sponsored & funded by NASA Headquarters

Software Group: Sponsored & funded by NASA Center

User Group: Any user group

Figure 5-3. Participants in the Software Technology Program.

group should be limited to four or five very senior people. In later stages of the program, the group might be augmented with people drawn from the Software Group and the User Groups. Such additions would bring directly to bear the personal experiences of those on the receiving end of the new software technology. Also, in later stages, as the group turns increasing attention to microprogramming and hardware, people experienced in the application of these areas should be considered.

The Software Group. The Software Group will be selected from one of possibly several operating within the NASA center which will have indicated its interest in participating in the program. The Technology Group, formed first, will make it one of its first tasks to determine the desirable characteristics of such a group. Center personnel can then be interviewed for their interest and opinions. There is little doubt that the project will attract unusual attention, and that there will be great interest on the part of existing software organizations to participate in the project and play a creative role. The Software Group will be sponsored and funded in the normal way by its Center; however, extra funding from NASA headquarters to support certain experimental or risk aspects of the program might be in order. It is possible that initially a separate Experimental Software Engineering section should be established to work on the program, rather than try to reorganize an entire software group. As the emerging technology proves itself, the size of the experimental group can be increased at the expense of the established group until the desirability of complete cutover to the new approach is apparent.

The User Groups. If the initial Software Group is an integral part of an existing software shop within a Center, user groups will comprise the normal clientele of that Center and that shop. The desirable characteristics of the participating Center might well include the nature of its mission, clientele, problems, and organization. Thus, the user groups would use their normal funds to secure their normal software services. Consideration for some extra funding, made available from NASA Headquarters through the Center by means of extra services or manpower

of its software shop, might be given to expand or modify software problems slightly to make them more appropriate vehicles for program objectives. It will be desirable that the mechanisms for such supporting funding be already established.

There is, of course, no reason that the experimental group should not undertake more than one problem at a time, provided that one and only one problem at a time be undertaken of an untried size and complexity. It would be in order for the Software Group to undertake programs of a size and nature that it has demonstrated it--and the new technology--can handle. However, the same careful follow-up and feedback through the software design, fabrication, test, operation and maintenance stages should be observed by all groups. It is to support those activities that extra funding might be arranged for the software and cooperating user groups.

5.5.2 The Cycles

There is little that can be said at this time on the nature of the software projects that will comprise the succession of application cycles of the evolving technology. Certainly, the first project should be a fairly straightforward application, either scientific or business, well within the capability of the present technology. Later cycles should undertake the kind of development project whose size and complexity have been well beyond the ability of the present technology to cope with in a straightforward fashion. It is up to the Technology Group to set its own pace; the responses of the Software and User Groups can be used to determine how well they are doing and how fast they should go.

It is also clear that the sequence should involve, in order: applications for which computer hardware, operating system software and proven source languages and translation software already exists; higher level software, such as file management and report generating software; still higher levels, such as data and data base management systems and multi-programmed and multi-processor operating systems. Within each such

category of complexity, there should be a natural progression of size.

To an extent best determined as a result of experience and success in the program, the scope can be expanded to cover microprogramming and hardware--that is, merging software and hardware technologies. Still more complex undertakings could then include building translators for higher level languages, including not only software but microprograms and perhaps some hardware specifications. At some period the Technology Group would start to look more like a Computer Applied Research Group investigating the relationships between hardware, firmware and software structures. At that point, its contributions to the technology would in fact, be that ascribed to Computer Science, as defined earlier--that is, research for the sake of knowledge, rather than for the sake of direct application. The burden of developing the technology in these final cycles would then shift from the Technology Group back to all of the software groups that, hopefully, will be applying various versions and interpretations -- that is, to the software industry.

5.5.3 The Phases and the Actions and Interactions of the Three Groups

Although the projects for the cycles will be different, the phases within each cycle, and the activities of each group in each such phase, will remain very much the same. The phases are as follows:

- Phase A. Technology Research and Development
- Phase B. Software Design and Fabrication
- Phase C. Software Operation and Maintenance.

The relationship between the groups and the phases are shown in Figure 5-4. The nature of each phase and the activities of the three groups in each Phase are given below.

Phase A. Research and Development (R&D Phase). In this phase, the activities of the Technology Group will dominate. This group will examine the chronic problems of software development and will conduct research

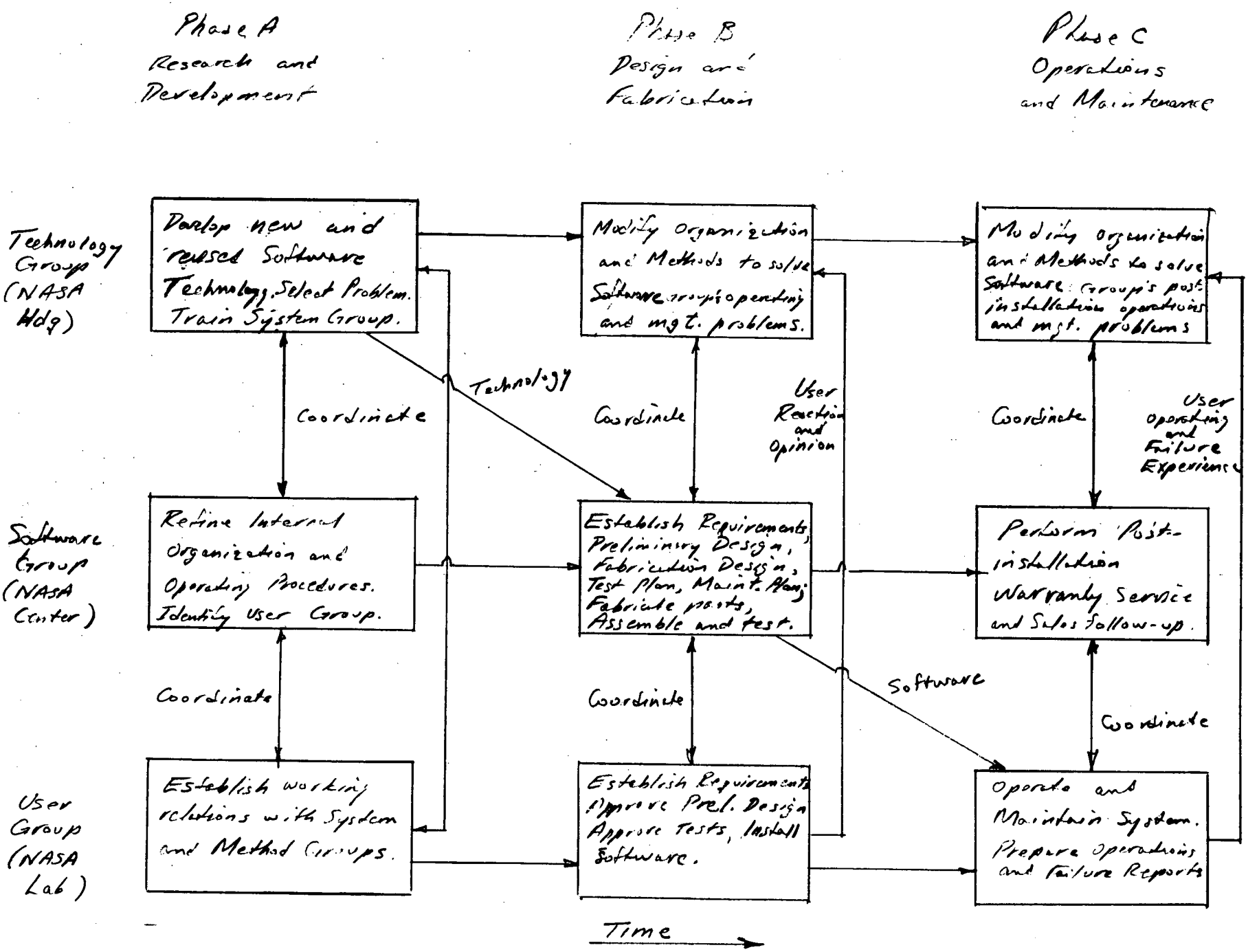


Fig. 54. The three-phase cycle for Software Technology development.

on software development methods, techniques, approaches, organizations, etc. that have been advanced to solve the problems. It will then develop an overall approach to the solution of these problems, comprising descriptions of an organization for designing, fabricating and testing software (and firmware), descriptions of the staff positions in the organization, procedures, techniques languages and graphics, standards and estimating methods. It will then assist in the organization and staffing of the Software Group, and cooperate with this group in selecting a user and his problem as its first or next effort. It will also establish tentative communications with the User Group so that it can obtain independent and objective information from both Software and User Groups on their respective problems and experience.

The Software Group will refine its internal organization during this phase and make preparations for using the revisions to the technology being developed by the Technology Group. It will interact with the Technology Group all during this phase by giving its reactions and opinions to the additions and modifications to the technology being planned by the Technology Group. It will also cooperate closely with the Technology Group in selecting a specific software development task from among those being presented to it by its users.

When an application has been selected, the sponsoring User Group will establish working relations with the Technology Group, and become familiar with the kind of information desired as feedback. Examples are the effectiveness of its communications with the Software Group, the latter's responsiveness to and comprehension of its needs, its ability to read and interpret preliminary designs and specifications, its reaction to various functional performance and acceptance tests, and finally, its reaction to the effectiveness of the software product itself and associated documentation, training, maintenance, and so on.

Phase B. Software Design and Fabrication (D&F Phase). The Computer Group dominates in this phase, in which it will work with the User Group in explicitly and unambiguously defining the problem,

establishing user constraints (time, cost, environment, operating and using personnel and specifications), developing a preliminary design for approval by the user, and the subsequent detail design, fabrication and test of the system.

The User Group will work with the Software Group to develop the requirements and preliminary design, and again in monitoring performance and acceptance tests on the major assemblies and completed system.

The Technology Group will observe and coordinate with the Software Group for deficiencies or weaknesses in the organizational structure, job functions, languages, etc. as the development work proceeds. It will not in general be concerned with assessing the quality of the product; rather, it will be concerned with such matters as lack of communication or understanding, schedule delays and slippages, missed estimates on man-hours, interface or system integration (assembly of parts) problems and the like. It will also coordinate with the User Group to obtain its reactions and opinions on the responsiveness of the Systems Group to its needs and its opinion of tests on major assemblies and subsystems.

Phase C. Software Operation and Maintenance Phase (The O&M Phase). In this phase, the activities of the User Group will dominate. The software developed by the Software Group will have been delivered in this phase, and its operation and maintenance will have begun.

It is assumed that the user will make his own arrangements for operation and maintenance, and that adequate documentation for this purpose will be prepared by the Software Group, accompanied by training of operating and maintenance personnel. However, it is also assumed that the Software Group, as part of its contract, will be responsible for some post-installation warranty service.

There will therefore be some communications between the User and Software Group during this phase. As a matter of fact, there probably

ought to be arrangements for failure reporting for an extended period after installation--enough time for failures to settle down to a "steady state." There will also be communication, for about the same "extended period" mentioned above, between the User and the Technology Groups. Such continuous cooperation and communication during this phase is most important. The Software Group will be interested in user feedback to modify its design to minimize warranty costs and customer maintenance; the Technology Group will be interested in user feedback to see how well the user anticipated his own needs and wants, how well he expressed them to the Software Group, how well the design engineers translated these to structure, and how well the fabricators within the Software Group were able to follow the design, how usable the software was by the user's operating personnel, and how useful the system was for the user's top management. This feedback will be used by the Technology Group to recommend changes to the organization, procedures, etc. of the Software Group for another cycle and another problem.

SECTION 6. A SPECIMEN TECHNOLOGY GROUP

Of the three groups that are proposed as participants in the program, only the Technology Group would have to be started "from scratch." For that reason, this section is presented as a more detailed description of what such a group might look like, do, and produce. It is written to serve as an example for management at NASA Headquarters, at NASA Centers, and at NASA client laboratories. It is addressed, in particular, to the person or persons designated to organize and operate such a group.

In thus addressing such management personnel, we do not presume to prescribe formally the organization and operation of a proposed new group; rather, we offer as a straw-man, or point of departure, the description of an organization that will be suitable for achieving the goals and objectives stated in Section 5.3, with the cooperation of already-organized and operating software and user groups. To the extent that a sponsoring organization modifies those goals and objectives, and to the extent made necessary by the structure and operating procedures of a specific Programming Group and User Group, this description should be changed.

In the succeeding paragraphs we shall address the benefits that should accrue to management as a result of the activities of the proposed group, its organization and operation, and its composition and products.

6.1 PURPOSE AND MISSION OF THE TECHNOLOGY GROUP

A detailed statement of the goals and objectives of the overall program, in which the Technology Group is slated to play the leading role, is given in Section 5.3. Because the roles of the Programming Group and the User Groups are essentially unaffected by the program, it is basically the purpose and mission of the Technology Group, and of the sponsoring parties, to achieve those goals and objectives. These goals and objectives, modified as deemed necessary by the sponsors, are therefore of fundamental importance. The plans made by the Technology Group's leadership, its actions taken over a period of time, and the products it will produce are all supposed to contribute to meeting those objectives. The effectiveness of the group at any given point in time can be realistically assessed only relative to those objectives; they comprise fundamental guidance and the basis for effectiveness evaluation.

Although the specific and detailed goals and objectives can be modified, there are a few so basic to software development that they must be included in the purpose and mission of the Technology Group. These can be phrased to apply specifically to the NASA groups that would be intimately involved in sponsoring the proposed program. There are three such sponsoring groups, each corresponding to a participating group. Without being specific, the three sponsoring groups and their corresponding program groups will be identified as follows:

NASA Headquarters	-	Technology Group
NASA Center	-	Software Group
NASA Clients	-	User Groups

The purpose and mission of the Technology Group with respect to each of these sponsoring groups will be briefly stated. They will be consistent with the goals and objectives of Section 5.3, but will be oriented specifically to the Technology Group rather than the entire program, and to the needs of several kinds of NASA management rather than NASA as a whole.

6.1.1 NASA Headquarters Management

The case for NASA Headquarters' retaining sponsorship of the Technology Group is directly tied to the Group's mission with respect to NASA Headquarters. That mission is essentially to promote the development, adoption and use of standards in such a way that greatly enhanced efficiency results in hardware and software procurement, maintenance and utilization without impairing the flexibility and autonomy of the various NASA centers and clients. Such a mission is best accomplished by a single Technology Group, even if the Group may eventually be split into several physical locations. A single sponsorship and administration will inhibit the formation of independent "Technology Groups," which would be more likely to develop and push competing standards than to promote the development of a single set of uniform standards.

In executing this mission, the Technology Group would provide, in the long run, a clearing house for the practices and products of all the Centers and their Software Groups. It would also act as a communication central, or interface, between the many and diverse groups. Thus, by proper organization, funding and sponsorship, the Technology Group will be in a position to promote increased cooperation in the computer activities of the various centers, without impairing or even suggesting an impairment of their automony. The net, long range result will be making each NASA dollar spent on computing matters do much more work. This is essentially the Computer Group's mission with respect to NASA Headquarters.

6.1.2 NASA Center Management

The matter of making a dollar do more work is of course a fairly universal mission, and it applies as well to NASA Centers as to NASA

Headquarters. It was brought out explicitly, however, to emphasize the fact that the proposed program is intended to provide NASA-wide benefits, and not just a high-status and high-visibility project and increased support for the particular Center sponsoring the Programming and User Groups.

The purpose and mission of the Technology Group with respect to NASA Center Management areas follows:

- to provide general management with effective and mutually comprehensible means of communication with software development management;
- to provide general management with effective and usable means of controlling a software development group's charter, budget, and products;
- to provide general management with effective and professionally acceptable means of measuring and comparing the performance of its software development groups with respect to the quality and quantity of product delivered.

These expressions of purpose and mission address a general and chronic problem of general management with respect to its computer organization. The essence of this aspect of the Technology Group's purpose and mission is therefore to remove software production completely from the atmosphere of pure research, and bring it firmly into the world of industry, production schedules, and cost accounting.

6.1.3 NASA Client Management

The primary goals and objectives stated in Section 5.3 apply explicitly to the users, with respect to whose needs the software industry has faltered, causing the present crisis. Saving money is always worthwhile, and exercising effective management over a production process is desirable, but neither is of critical importance if the end customer is happy. With respect to software, this is far from the case. The ultimate mission of the Technology Group, then, is to develop a technology that will be at least as satisfactory to software purchasers as, say, the computer hardware

technology is to hardware purchasers. By a satisfactory technology we mean one with the ability to express needs and constraints so that they can be compared meaningfully and credibly with the suppliers' expression of capabilities and costs.

Stated explicitly, the purpose and mission of the Technology Group with respect to NASA Client groups are:

- to provide the user with the means for describing his functional, procedural and data problems to a computer systems engineer
- to provide the computer system engineer with the means for expressing these functional, procedural and data needs explicitly and rigorously in documentation comprehensible to the user or his agent
- to make such documentation legally and logically meaningful as a performance specification and preliminary design. To provide the user with the means for testing software and comparing the results of the test meaningfully with the contractual descriptive documentation
- to assure that the languages and standards developed to express requirements, legal descriptions, and tests are compatible and will permit impartial third parties to determine contractual deficiencies.

One of the more important characteristics of software is its correctness. This property, not yet defined in a standard way, is one of the most difficult to predict in undeveloped software, and it is almost as difficult to measure in completed, "de-bugged" and delivered software. This shortcoming of the software industry, traced backwards through the software development cycle, is responsible for (or is the result of) most of the other faults that might justifiably by attributed to the industry.

It is our belief, expressed implicitly throughout this report, and explicitly in several places, that the root cause of this situation is that software development has been considered almost synonomous with "programming." That this is not the case is now quite evident; the remedy,

however, is not so evident. This program is predicated on the assumption that there is no simple remedy; that the remedy will have to evolve rather than be prescribed; that such an evolution must be (or at least can be) directed; and that such direction must come from a group not itself engaged in developing or using software.

Thus, the Technology Group is responsible most of all to the NASA Client Management; yet its dealings with that management are minimal. In fact, they will consist largely of judging the Technology Group's effectiveness with respect to its function and mission as viewed by NASA Client Management.

Achievement of the Technology Group's purpose as far as NASA Client Management is concerned, therefore, is indicated by affirmative answers to the questions, "are you satisfied that software suppliers understand and can respond to your requirements?" and "are you satisfied with their products, costs, and services?" To the extent that the answers at the end of successive program cycles indicate increasing satisfaction, the Technology Group is fulfilling its purpose and mission. The manner and degree of dissatisfaction will provide feedback and guidance to the Technology Group for the succeeding cycle.

6.1.4 The Technology Group's Management

In the preceding three sections we have discussed the purpose and mission of the Technology Group as viewed by each of the three sponsoring groups. These views will of course do much to drive the Technology Group's sponsorship, charter, organization, activities and products. The management of the Technology Group will in a sense be responsible to each of those three groups, since each has a stake in its operation.

In addition to these responsibilities, the group has a larger purpose with respect to the Computer Scientific community, and to the Computer Industry at large. This purpose is quite clear: it is to communicate objectively to these communities, both the positive and

negative results of its efforts. Beyond such communication, it may assume the larger mission of posing questions, suggestions, research projects, and the like, for these communities to pick up and respond to. In this way, its investigative powers may be multiplied many times, at no extra cost to the government and to industry. Thus, confirmation of tentative results may be obtained, trial standards may be tested, and research projects may be initiated. With the expenditure of relatively little manpower, enormous leverage will therefore be exerted to coordinate and give direction to the many efforts that are even now being expended to develop a software technology.

The essence of the function and mission of the Technology Group as viewed by its own management is therefore technological communication, oral and written. Technical competence is always implied, and innovative ability is clearly needed. The ability to communicate effectively and succinctly, however, is essential, in both expository and tutorial forms of presentations.

6.2 THE ORGANIZATION AND OPERATION OF THE TECHNOLOGY GROUP

Given the general goals and objectives relative to the technology to be developed as stated in Section 5.3, and given the purposes and missions relative to the NASA sponsors of the program, some reasonable conclusions can be drawn about the organization and operation of the Technology Group. These conclusions will be described and discussed in this section, with expansion of several relatively important topics in later sections.

6.2.1 Size and Composition

The group initially should be quite small - perhaps four or five persons. The formal organizational structure at this stage will therefore be inconsequential - a leader, three or four innovative, top-level people having fairly broad systems backgrounds with a concentration in computers and non trivial programming experience, and one or two support personnel.

The group's initial roles and missions will have been determined in discussions between the sponsors and the designated leader, using this report as the take-off point. These will establish the qualifications of the first staff members, as determined by the group's leader. During these organizational discussions, the projected growth of the group for a one-year period will also be established, with the corresponding expansion, if any, of the group's roles and missions. The organization should remain fairly informal during this growth period, so that ample opportunity is provided for a natural evolution of working arrangements, reporting and documentation procedures, and liaison and communications arrangements with the designated Software and User Groups.

The major point of the lack of formality during the initial stages of operation is the diplomatic aspects alluded to in the previous section. About the most important objective is to establish good working relationships with the other two participating groups. Formality implies explicit responsibilities; explicit responsibilities imply established authority and its exercise. It is important that the Software Group acquire no impression that the Technology Group has any authority over it. Practices or standards developed by the Technology Group and tried out by the Software Group should be the result of definition of problems and selection of which ones to attach - by discussion and negotiation.

Similarly, it is important that the User Group does not develop an impression that the Technology Group is a complaint bureau.

The organization and composition of the Technology Group at the end of, say, a year should be determined by the interactions, relationships and procedures that develop in a natural manner between the people involved. The formal aspects of organization and procedure, which will most certainly be required in later stages of the Group's operation, will therefore be tailored to the characteristics of primarily the Software Group, which is as it ought to be. The Software Group has a production responsibility which the program is charged to improve over time; its capability ought not to be impaired by its having to adapt in a short time to a new situation.

Virtually all of the adaptation should be undertaken by the new element - the Technology Group - which has no vested interest in an existing organizational structure and operating procedure.

6.2.2 Staff Qualifications and Job Descriptions

The most outstanding general qualifications required of each group member in the initial stages are:

- analysis ("20 questions")
- discrimination (what's important)
- reporting (natural, jargon-free language)
- open-mindedness
- diplomacy

To these must be added technical competence in the areas of system engineering and computer technology, including hardware and software.

The initial group should include top-level System and Computer Engineers as described below. These people should be selected to have a mix of backgrounds equivalent to the lower level Production and Test Engineer's, which implies a strong hardware content. This requirement is based on the need for a transfer of knowledge and approach to production from the highly successful computer hardware industry, as advanced in other parts of this report. Some details of such an approach will be given in a later section.

Following are some job titles and descriptions which are suitable for staffing the group.

Senior Systems Engineer

Performs systems engineering projects and studies involving the wide application of engineering principles, theories and concepts to the

designs, development, testing and evaluation of systems, sub-systems and components for complex user requirements and applications. Applies current state-of-the-art knowledge to areas of specialization. Serves as a technical staff planning member and consultant on projects; may serve as a project leader on systems engineering assignments. Should have a strong background in systems having a digital computer as a major subsystem.

Computer Systems Engineer

Performs systems engineering assignments involving the application of substantive computer engineering knowledge to the solution of a variety of operational problems associated with the design, operation and maintenance of complex systems. Analyzes, evaluates and recommends designs involving hardware and software components and subsystems based on operational analyses and other studies. Develops preliminary designs, specifications, standards and tests to be used in developing and testing systems. Makes estimates of cost, size, labor and other resources necessary to complete design, fabrication, interpretation and test of computer systems, including hardware and software components.

Software Systems Engineer

Responsible for the design, development, and maintenance of advanced computer programs and/or program systems to solve complex problems to meet customer needs. Applies expert knowledge of programming techniques, languages, translators and of hardware configurations to determine software requirements. Establishes working parameters and formats, identifies potential problem areas, insures system flexibility to accommodate future changes to system or requirements, and identifies and solves hardware/software interface problems. Performs preliminary design, detailed system design, program design, software production engineering, test and maintenance. Conducts evaluations of programs or systems including assessing existing software for potential application, investigating the appropriateness of design change suggestions and verifying that required modifications have been tested, integrated and documented. Makes estimates of

cost, manhours and size of software components, and schedules design, fabrication, assembly and test.

Software Design Engineer

Consults with System Engineers and with the user, and develops preliminary or detailed designs for software structure to perform the user's operations within his constraints (time, money, computer, system hardware, data base, etc.) Has knowledge of computer system hardware structures, operation, and logic design as well as of source ans object languages, machine code, peripheral operations, program libraries, proprietary software. Has significant programming experience in assembly and higher level languages. Responsible for producing the specifications for the structure and content of the data base and data input, and for the structure and components, in functional terms, of the mechanism, including software and firmware for transforming the data. Lower levels of software design engineers are responsible for transforming the software functional specifications to detailed designs.

Production Engineer

Responsible for scheduling the fabrication process and assigning these tasks to various specialties (shops) for fabrication (coding) or assembly of subroutines and program components into larger modules or assemblies. Establishes source languages to be used, and designates the assemblers, compilers, or cross assemblers and compilers, and machines upon which test assembly and compiling will be accomplished.

Test Engineer

Designs and develops test plans, test specifications and procedures to determine the functions, performance, reliability and life cycle of various components, subsystems and systems. Conducts tests, analyzes the results and makes recommendations to improve the characteristics and performance of components and system. Evaluates test procedures

to determine that tests conform to plans. Conducts special engineering studies relating to findings of test and analysis function.

Works with the user and design engineers to develop the specifications or descriptions of the test data, including data base, and data inputs, test conditions and environment, including simulations and emulations, determines correct output from the test, and the test set-up, procedures and reporting requirements. Also determines machine and machine configuration upon which tests will be run. Consults and coordinates with hardware test engineers as required in developing software for new hardware.

Software Maintenance Engineer

Customer Engineer; the individual that the customer depends on to get his system running after it has failed or after a malfunction has been detected. Implies close coordination with all other engineers involved in building the software and hardware.

Senior Systems Analyst

Develops user requirements relating to the generation, processing and retrieval of information. Analyzes data processing procedures including those involving hardware and software and recommends improved methods, systems, and procedures for increasing operating effectiveness based on weaknesses, anomalies, redundancies and other undesirable characteristics. Develops and refines graphical and verbal descriptions of procedures, processes, data sources, data structures and human functions, operations and tasks. Specifies in detail the logical, mathematical or physical operations to be performed by various machines, programs and personnel. Prepares technical reports, memoranda and instruction manuals relating to system operations and procedures.

Operations Research Specialist

Performs as an operations research consultant by planning and conducting studies and programs involving advanced operational analyses,

techniques, principles and concepts applied to the design, development and operation of complex systems. Conceives and conducts studies directed toward achieving optimum operation of all elements in complex man-machine systems. Provides authoritative direction based on scientific analyses of the interaction of various system elements such as process hardware and software, communications, storage, display, and terminal equipment. Initiates studies utilizing and extending knowledge in areas such as mathematical statistics, queueing theory, system simulation, logistics, linear programming and operational gaming.

The foregoing descriptions apply, of course, to just those high-level positions required for staffing the Technology Group. Personnel in lower levels of system and computer practice, such as programmers, coders, operators, system analysts, and system engineers do not at the moment appear even to be required.

The descriptions taken together can be interpreted as applying to the background, experience and abilities required. Such an interpretation would permit transferring certain capabilities from one job description to another, merging job descriptions, or breaking them apart to some extent. Also, the job descriptions can be seen to be ideal. Certain of the sought-for capabilities - for example, estimating time and cost - represent objectives of the program. Nevertheless, in this example, people having estimating experience can be found.

These job descriptions, it must be emphasized, are points of departure. They will have to be modified and tailored to suit the actual program.

6.2.3 Initial Role of the Technology Group

Considering the discussion of program goals and objectives in Section 6.3, and the Technology Group function and mission presented in Section 6.1, this section will be very short. It is included primarily to emphasize the need in the initial stages for a small, top-level group,

each member of which is accustomed to dealing with both technical and management people in consulting and administrative capacities. The stature, knowledge, reputation and bearing of each member should be such as to inspire respect in the Software and User Groups, obviating explicit authority.

The role of this initial group will be that advisors and consultants to both the Software and User Groups. In fact, the problems of these two groups, as perceived by them, in developing and using software (respectively), comprise perhaps a better set of initial tasks than an ideally-defined set derived from a study of the industry at large. Careful analysis of such problems, developed through informal interview, interaction and discussion, will reveal areas within the goals and objectives of the program. Quick solution or assistance in small, irritating but perhaps superficial problem areas will establish credibility, trust and a good rapport much sooner and easier than deeper and more subtle problems.

The initial role, then, will not be that of super-duper computer hot-shots out to revolutionize the computer industry. It will be that of competent, high-level computer and system consultants dedicated to improving the tools and procedures of a software development group, at its option, and, in the process, generalizing the improvements and publishing the results (probably jointly with senior and junior Software Group personnel).

6.2.4 Products

The products of the Technology Group will comprise technical reports, manuals, text-books and presentations, both expository and tutorial. All legitimate media will be employed, including institutional (NASA) reports, proceedings and papers in the professional journals, informal articles in the trade journals, books, and presentations, classes and seminars. As stated earlier, the essence of the purpose of the group, as viewed by the group itself, is communication: The promulgation throughout the computing community of the results of its own and other people's work (with credit).

The subjects of these communications will lie generally within one of the following topics:

- Industrial standards
- Representation and languages
- Software Production Techniques
- Production Performance Measurement

Each of these will be discussed in more detail in the next section.

6.3 THE PRODUCTS OF THE TECHNOLOGY GROUP

The attempt was not made in this study to establish a taxomony of topics of investigation for the Technology Group. The four topics mentioned at the end of the preceding section represent some of the more obvious deficiencies in the technology presently used by the software industry, particularly when compared with its fraternal twin industry, the computer hardware industry. Consequently, it can be expected that the principal products of the Group will be in these areas. They will be discussed in more detail in the following paragraphs.

6.3.1 Industrial Standards

An industrial standard is a criterion of measurement, quality, performance or practice, and may be established in a number of ways. One way, well known in the computer industry, is simply the adoption, by small concerns, of certain of the technical specifications of a line of products of an industrial giant or leader. Other ways include the action of standards committees established by the industry in question, custom, consent or governmental authority. An industrial standard may be technical, in which case it usually specifies what and how. It may be an operative standard, which usually involves human elements, and specifies who, when and why. An industrial standard may also involve both types. Specific examples of standards are:

- Product standards
- Engineering design standards
- Quality standards
- Procedural standards

In considering the adoption of a standard, the maturity of an industry, a product or a production technique in that industry is a factor. Premature adoption of a standard has distinct disadvantages, and failure to adopt at an appropriate time will also have non-trivial drawbacks.

There are a few basic principles with respect to the establishment of standards which the Technology Group will find it well to adhere to. One is that standards are not imposed; they are adopted. The role of the Technology Group with respect to standards should therefore not be arbitrary action, but arbitration. The general principles are:

- Standards that are adopted at too early a stage of maturity of an industry, product or procedure are subject to frequent and possibly continual revision in order to keep pace with progress in the corresponding technology. This will defeat the purpose for advancing the standard.
- Standardization tends to inhibit technological progress and development, and to stabilize conditions at the level of development at which it occurs. The implication is double-edged: premature standardization conflicts with orderly development; delayed standardization impairs stability and orderliness once reasonable maturity is achieved.
- The need for flexibility and adaptability to change coupled with the need for, but undesirability of, changing standards implies that standards should be adopted, but that they should be as few in number as is consistent with technological stability.

With these principles in mind, we can explore, breifly, some of the areas in which standards might be considered in the software industry. We shall use the breakdown mentioned earlier.

6.3.1.1 Product Standards. These establish the form, size, quality and performance of a product or series of products that can be considered as major components or small subsystems. Examples in the field of computer hardware are medium scale integrated (MSI) circuits, keyboards, batteries and amplifiers. Standards for some of these particular products have long been established. Analogous examples in the software industry might include subroutines for mathematical functions and tables of mathematical functions, random numbers, chemical and physical properties. Since such tables must be stored on physical media, product standards could also include the characteristics of such tables, including table structure, word size, increments of arguments, argument word size, along with the physical characteristics of the storage medium: tape, number of channels, blocking factors, access methods. Finally, (in the same example), product standards might include interpolation routines. In all cases, the point would be that a program or data set fabricated in accordance with the standard could be specified for use by a prospective system software engineer in the same manner that the designer of a miniature electronic calculator can specify a battery, keyboard, display, MSI circuit and so on, by standard nomenclature.

Product standards obviously benefit the design engineer, since the existence of standard products allows him almost immediately to make certain assumptions or estimates concerning structure, measurement, form and cost. However, in the long run, they benefit the user because they provide a product that is essentially modular and includes modules that are interchangeable, uniform in quality and performance, and probably lower in cost than hand-tailored equivalent products. In a certain sense, the proprietary software industry has already started building standard software products. At the moment, however, the programs are primarily end products that perform specific end-user functions. True product standards are components that perform low-level functions and are used by "original equipment manufacturers" (proprietary and custom software houses) as component parts of their end products. This topic has been discussed in some detail at the Software Engineering Symposia.

6.3.1.2 Engineering Design Standards. These are like product standards in that they represent software components, but at a distinctly lower level.

Examples in the twin field of computer hardware include transistors, resistors, sockets, connectors, capacitors. Examples in general industry include screws, nuts, bolts, sockets, wire and fittings of all types; they tend to involve physical and electrical interfaces. Analogous engineering design standards for the software industry might include standards for parameter passing and linkage between programs. As in all standards situations, the existence of standards does not really restrict; non-standard design is always a freedom that can be exercised, and there are almost always an adequate number of alternative standards to choose from.

A specific example of a linkage engineering design standard is one in which a push-down stack is employed to pass three parameters:

Top word:	Identification of calling routine
Second word:	Return address
Third word:	Location of data pointer list

A second linkage standard might involve passing only one pointer to a complete parameter list. Additional engineering design standards, both industry-wide and corporation-local, might apply to program identification codes and the associated security procedures, system structures and hierarchies, and data pointer list formats and content. The point, of course, is that the two standards represented by the above example would not be the only two linkage and parameter passing standards. The present difficulty is that each corporation, each software shop, and even each programmer within a software shop - all use different linkage "standards."

6.3.1.3 Quality Standards. This is an area where much of the present software crisis is concentrated. Quality of software products is low, compared with computer hardware products; quality is unpredictable and cannot be designed for, and quality control is virtually non-existent.

Errors and "bugs" are an accepted way of life; meaningful verification or checks of programs by third parties are difficult, and few if any software shops include such second party audit procedures.

Quality is not an absolute property but must be assessed relative to these considerations (which apply to hardware as well as software products):

- The end-use of the product is relevant.
- Quality must be expressed in definable and measurable characteristics of the product.
- Quality is related to the cost of production and the sales price.
- Maintenance cost is affected by quality.

It is desirable to be able to express degrees and kinds of quality, and to control quality, not merely to achieve perfect programs, but to permit some level of imperfection at correspondingly lower cost wherever such imperfection can be tolerated.

For example, the words "correctness," "robustness," and "reliability" have all been used frequently as descriptors of quality. They have yet to be authoritatively defined and quantified. In fact, it is unlikely that rigorous definitions have been advanced; the words are used generically. Such definitions would be at least starting points.

6.3.1.4 Process Standards. Process standards include standards that apply to operating methods. Some of these are discussed in the literature and have to do with documentation standards, utilization policy of computers and system software, use of source languages and compilers, and verification and check-out procedures. These standards include the development of production standards used in performance measurement: lines of debugged code per day per coder, storage requirements per line of high-level source language, and compilation time and check-out time data on a per-line basis. Such standards not only enhance production and communication between personnel, but provide the basis for informed estimating and scheduling.

6.3.2 Representation and Languages

Graphic, verbal and machine languages and conventions are also "standards," but are of a special enough nature and purpose to merit a special category. Of course, software is essentially expressed in terms of various languages; it is not intended that the Technology Group expend any effort in developing new source languages. Examples of the particular kinds of representation and languages to be addressed do include:

- definitions of terms and phrases
- a "requirements" language
- a software structure language
- more formal and useful operational flowcharting conventions
- production scheduling, routing, and assembly forms
- functional and structural specification standards
- representation (symbolic) of hardware/software and software/software interfaces.

Typical examples of words that need explicit definition and universal adoption are correctness, robustness, reliability. A more careful and authoritative analysis should be made of the kinds of errors or bugs that occur in software, so that they can be named and their incidence reported and recorded. Such data will help in developing procedures to reduce errors. The use of jargon may in this way be reduced, and communication between computing personnel in various specialties, installations and parts of the country will be enhanced.

A requirements language is needed to provide for improved communication between user or user representative and software engineers and analysts. The object is to assure that statements of requirements can be set down in explicit and written form by systems or computer analysts and engineers as the result of an operational process analysis and interviews with user personnel. The language should permit representation of procedural steps, data sets, automatic equipment and operator

action and yet be simple enough so that user personnel relatively new to computer and systems work will have little difficulty verifying the written expression of their requirements. Some special form of flow charting, making minimum use of flow charting symbols and virtually no use of highly specialized notation would seem to be appropriate, accompanied by normal text to supplement the flowchart boxes and the describe data sets and sources.

Perhaps the greatest need is for a means of representing a software structure, complete with interconnections (interfaces). The art of representing processes and procedures is highly developed, although further development is required even here. In fact, the representations for structure and process should be complementary; the common element should be descriptions of data sets and structures. This is the greatest deficiency in flowcharting; the emphasis is all on process and sequence. Even here, the data input and output at each step is generally inadequately described, which is responsible for faulty interface design or planning. Intrinsic to the nature of the structural language, in fact, is the ability to represent the connectivity between programs - linkage, parameter passing, data access and identification (for security purposes). This relates, of course, to linkage standards. Once such standards exist, they can easily be graphically represented. Once graphical standards have been adopted, the nesting of computer programs and components to successively lower levels of detail can be meaningfully represented. At that point, experience, intuition, visual perception and the native sense of structural propriety that human beings possess can be fully exercised in developing sound software designs. As in the case of hardware, means of representing structures at all levels will be needed; at the highest level, to provide a preliminary design to accompany general specifications as the basis for user negotiation and contractual arrangement; and at the lower levels, to provide "blue prints" for fabricating software; coding and assembling software components into successively higher levels of subassemblies and assemblies.

Production scheduling, routing, and planning, highly developed production techniques in the hardware world, are at best still in their

infancy in the software world. In this respect, the software industry is still in the age of guilds, in which each component of an end product is hand-crafted with the help of a few apprentices. In today's world of interchangeability, complexity and sheer size, the job must be broken down into layers of buildable and testable pieces, each in turn being an assembly of smaller pieces. Clearly, this requires that the design be appropriate to the end-product's function and operation, but that it also be appropriate for building, assembling and testing. In other words, the procedures used in building and assembling have almost as profound an effect on design at the lower levels as functions do at the higher levels. This statement applies, of course, to both structure and the interconnections of structures at a given level.

There are indications that the nature and function of specifications is not clearly understood by some software specialists. A specification is a design. This is, of course, not the case. A design can exist without a specification, and a specification can, in general, be not fully representative of a design. In fact, a specification is a legal document, an adjunct to a contract, that sets forth a verbal desciption of the item to be purchased. Other adjuncts to the contract include plans, drawings and diagrams to which the specifications may refer. Still other adjuncts have to do with schedules, testing and performance criteria. Thus, work is sorely needed to develop standards for specifications that will complement the representations and languages for requirements, preliminary design, and structure and also consider the nature of the basis of agreement and reciprocal responsibilities between software purchaser and software supplier.

6.3.3 Software Production Techniques

These apply to the processes of design and fabrication, rather than to the techniques or tricks used in programming and coding. Examples are:

- structured programming
- chief programmer teams

- operational assemblies or "builds"
- production engineering

These are relatively new techniques that have been advanced and successfully tested within recent years. They appear to provide good techniques to start with, because they have been successful enough to offer much promise, but yet not so well developed that they can be considered fool-proof.

One of them, structured programming, treats a computer program, system of programs, and program components as structures. This is an approach that merits much more attention and development than it is receiving. The approach highlights the lack of a structural language - that is, a method of representing software structures that is as well developed as the methods of representing hardware structures. Examples of the latter are logic diagrams, wiring diagrams, block diagrams, exploded views, isometric diagrams and so on. These are graphical, but are complemented by the meets and bounds. Therefore, additional initial techniques should include the search for or development of sound structural representations of software and interfaces with hardware and other software.

Another of the foregoing techniques, chief programmer teams, is organizational in approach rather than technological. However, it is a consequence of the structured approach and is therefore closely related to it. If the organizational aspects are stripped from the technique, what is left is an emphasis on the identification and resolution of interface problems as a part of the design process rather than as a part of the debugging process, a much greater emphasis on the importance of the design process as distinct from coding, and the adoption of a certain structural philosophy or software architecture. This architecture is a specific example of the structured approach, and involves the establishment of such structural standards as single program entry points, single program exit points, entries only from and exits to only the next higher program level (no GO TO's), uniform parameter-passing and linkage conventions, etc.

This also points to the need for an effective, unambiguous and standard means of communication on software structure and form between design echelons of the software organization, and between the design and the fabrication groups.

In general, various techniques will apply to various stages of software development. The foregoing two techniques apply primarily to the later stages of design and earlier stages of coding. Techniques applying to statements of requirements, preliminary design and general specifications, and to the system assembly, test, operation and maintenance stages of development and employment should also be sought and modified or developed. These are discussed in various other paragraphs of this section.

Other techniques that may be suitable for the initial stages may be found in the literature (see Section 3). The important thing is that they be compatible with the Software Group's general organization and method of operation. The use of techniques that may have substantial immediate impact on the Software Group should be avoided. It is probable that some "home-grown" techniques can be picked up, modified and improved. This should be done wherever possible.

6.3.4 Performance Measurement

The ability to assess the performance of a software producer, whether an individual or a group, is basic to both the ability to estimate costs and time, and to exercise corresponding control over the production process. At the same time, the ability to measure performance will not of itself provide good production techniques. It will permit responsible management to measure the difference between alternative techniques and procedures. This, of course, is the primary motivation for mentioning performance measurement as one of the more important product categories of the Technology Group.

Performance measurement applies to the management of computer software production rather than to the operation of computer hardware or software. However, the quality of product delivered is certainly an important aspect of performance, and so the measurement of hardware and software performance must be included. However, in this context, performance measurement is the measurement of the capacity of a Software Group to produce working software, together with qualifying measurements of cost, time, and quality of product. This area is closely related to the ability of software management to be able to estimate costs, manhours, manpower skill and level requirements, schedules of software and to exercise some measure of control over its quality.

Basic to the establishment of such measurements is the establishment of some significant portion of the standards, languages, and software production techniques discussed earlier. In fact, the topic of performance measurement is discussed elsewhere in this chapter from other points of view. The point of view here is that science, technology, industry and commerce are based on the ability to measure things or phenomena, describe the results in numerical terms, make comparisons, and make decisions based on these comparisons. For the software industry to emerge as a stable member of the industrial family, its products must be describable and measurable in standardized ways. Predictability of function, performance, cost, size and other qualities then becomes a characteristic of the industry. This is what technology is all about, and assisting in the establishment of sound measurements of performance is perhaps the most important and ultimate job of the Technology Group.

REFERENCES

Section 2

1. I. L. Auerbach, "Need for an Information Systems Theory," an address before the International Federation for Information Processing, Amsterdam, Netherlands; AUERBACH Associates, Inc.

Section 3

1. Niklaus Wirth, "Program Development by Stepwise Refinement," Communications of the ACM, Volume 14, No. 4, April 1972.

2. Edsger W. Dijkstra, "The Structure of the "THE" Multiprogramming System," Communications of the ACM, Vol. II, No. 5, May 1968.

3. E. W. Dijkstra, "A Constructive Approach to the Problem of Program Correctness," BIT 8 (1968), 174-186.

4. E. W. Dijkstra, "Structured Programming," Software Engineering Techniques, Report on a Conference sponsored by the NATO Science Committee, ed. J. N. Buxton and B. Randall, October, 1969.

5. E. W. Dijkstra, "Go to Statement Considered Harmful," (Letters to the Editor) Communications of the ACM, Volume 11, Number 3, March 1968.

6. Corrado Bohm and Guiseppe Jacopini, Flow diagrams, Turing machines and languages with only two formation rules. Communications of the ACM, 9, May 1966, 366-371.

7. H. D. Mills, "Structured Programming," October 1970.

8. Ibid, page 12.

9. Ibid, page 5.

10. P. Naur, "Proof of Algorithms by General Snapshots," BIT 6, 4, 1966, 310-316.

11. R. W. Floyd, "Assigning Meanings to Programs," Proceedings of Symposia in Applied Mathematics, Vol. XIX, Mathematical Aspects of Computer Science, American Mathematical Society, Providence, Rhode Island, 1967, 19-32.

12. D. I. Good and R. L. London, "Interval Arithmetic for the Burroughs B5500: Four Algol Procedures and Proofs of Their Correctness," Computer Sciences Technical Report No. 26, University of Wisconsin, June 1968.

REFERENCES

Section 3 - continued

13. B. H. Liskov and E. Towster, "The Proof of Correctness Approach to Reliable Systems," The Mitre Corporation, July 1971, p. 10.

14. N. E. Willmorth, "System Programming Management," TM-(L)-2222, SDC, Santa Monica, California, 1965.

15. F. T. Baker, "Chief Programmer Team Management of Production Programming."

Section 4

1. E. W. Dijkstra, "Concern for Correctness as a Guiding Principle for Program Composition," Fourth Generation International Computer State of the Art Report, 1971, 360.

2. C. A. R. Hoare, "The Use of High Level Languages in Large Program Construction," Efficient Production of Large Programs, edited by Barbara Osuchowska, Proceedings of International Workship Jablonna, August 10-14, 1970, 82.

3. D. B. Mayer and A. W. Stalnaker, "On the Management of Computer Programming," Auerbach Publishers, Inc., 1970.

4. J. I. Schwartz, "Analyzing Large Scale System Development," Software Engineering Techniques, edited by J. N. Buxton and B. Randell, Report on a Conference Sponsored by the NATO Science Committee, October 1969, 128.

5. B. Randell, "Efficient Production of Large Programs," edited by B. Osuchowska, Proceedings of International Workshop Jablonna, August 10-14, 1970, 116.

6. M. E. Conway, "How do Committees Invent?" DATAMATION, April 1968.

7. J. N. Buxton and B. Randell, eds., Software Engineering Techniques, Report on a Conference Sponsored by the NATO Science Committee, October 1969, 7.

8. G. K. Manacher, "Efficient Production of Large Programs," edited by B. Osuchowska, Proceedings of International Workshop Jablonna, August 10-14, 1970, 16.

Section 4 - continued

9. W. M. Turski, "Defining Large Programs," (introduction to the working session on the same subject) in Efficient Production of Large Programs, edited by B. Osuchowska, Proceedings of International Workshop, Jablonna, August 10-14, 1970, 4.

10. E. W. Dijkstra, "The Structure of the 'THE' Multiprogramming System," Communications of the ACM, Vol. II, No. 5, May 1968, p. 343.

Section 5

1. J. N. Buxton and B. Randell, eds., Software Engineering Techniques, Report on a Conference Sponsored by the NATO Science Committee, October 1969.

2. P. Naur and B. Randell, eds., "Software Engineering," Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, October 1968.