SOFTWARE ENGINEERING:
TOOLS OF THE PROFESSION

Arrena Sue Williams

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SOFTWARE ENGINEERING:
TOOLS OF THE PROFESSION

by

Arrena Sue Williams

September 1976

Thesis Advisor: G. L. Barksdale, Jr.

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Software Engineering: Tools of the Profession	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis; September 1976
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Arrena Sue Williams	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93940	12. REPORT DATE: September 1976
	13. NUMBER OF PAGES: 92
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Naval Postgraduate School, Monterey, California	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

software engineering
modular programming
structured programming
top-down programming
decision tables
chief and programmer teams
military software
tactical software

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Software engineering is presented as a new branch of the engineering disciplines. The tools and techniques of the profession are examined in an attempt to resolve definitional ambiguities and describe the concepts or attitudes generally associated with three specific programming methodologies. Language properties supportive of the methodologies are investigated. The professional tools and language characteristics are evaluated in terms of their effect on DOD software.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |
(Page 1)

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

SOFTWARE ENGINEERING: TOOLS OF THE PROFESSION

by

Arrena Sue Williams
Lieutenant, United States Navy
B.A., Southwest Texas State University

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

September 1976

ABSTRACT

Software engineering is presented as a new branch of the engineering disciplines. The tools and techniques of the profession are examined in an attempt to resolve definitional ambiguities and describe the concepts or attitudes generally associated with three specific programming methodologies. Language properties supportive of the methodologies are investigated. The professional tools and language characteristics are evaluated in terms of their effect on DOD software.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

Initially used less than eight years ago [31], the term software engineering is used today to describe such dissimilar activities as programming tools and standards, software development, and programming methodology. Several widely known techniques in the design of computer software today are not clearly defined. People who speak of these techniques have their own, usually unique impression of what they mean by such terms as modular or structured programming. There are those who even disagree over the meaning of the word "software" and further object to its design being referred to as an engineering discipline. Yet, numerous languages, programming methodologies and other aids are championed by authors offering solutions to the "software crisis". Software development is a highly individual activity where the programmer's skills, biases and motivations often govern the process. It is not surprising that such diversity of opinion has developed regarding the terminology, and the conceptual interpretation and relative value of the proposed tools.

A major goal of this thesis, therefore, is to investigate the tools and techniques by examining the problem of varying terminology in the literature and describing the concepts generally associated with top-down, structured and modular programming. In addition, an attempt is made to resolve the controversy regarding the scope of the software engineering profession, as it clearly offers solutions to the plaguing problems confronting the military today in the development, acquisition, deployment and support of major defense systems. This information is

consolidated into a unified presentation of current theory by providing a topical, annotated bibliography of software engineering literature and a dictionary of terms in Appendices A and B.

When writing short programs of a hundred statements or less, almost any programming language will satisfy the requirement; however, when producing large software systems, the programming language used will be of crucial importance to the success of the project. Language properties supportive of structured programming are therefore investigated, including the Navy High Order Language requirements. Finally, the effects of program structure on the design of military software is evaluated.

II. SOFTWARE CRISIS

Before any attempt is made to present the proposed solutions for designing reliable computer programs, it is first necessary to arrive at a working definition for the term "software" and the underlying reasons for the present software crisis. It was the recognition of this crisis that implied the need for the design of software to be based on the theoretical foundations and practical disciplines that are traditional in the established branches of engineering.

A. DEFINITION

The word software orginated about 1959 or 1960 in the United States [18]. There are those who still use the term to encompass only the area of what is commonly known as systems software; that is, assemblers, compilers and operating systems. To do so, however, is to imply the problems associated with developing reliable systems programs are unique from those associated with application programs and therefore require disparate solutions. In fact, the types of problems that are encountered in constructing large systems programs are much the same as those encountered in large applications programs. Yet to use the term computer software interchangeably with computer program is to overlook the necessity of considering compatability between systems and applications programs during the design process. When used in this thesis, computer programs will refer to either systems or applications programs, whereas the term software will mean a combination of associated

programs. Formal definitions of these terms are included in Appendix A and are consistent with those adopted as DOD standards within the weapon system arena [23].

B. CRISIS

During recent years, there has been increasing talk of a software crisis. Yet at the Garmisch NATO Conference in 1968 there was no unanimity on the existence of such a crisis [31]. Some felt that almost all the proponents of the software crisis were the "university types" who could not understand how to handle large projects [18]. Some may question the choice of the word "crisis", but the fact remains that, as a percentage of the total data processing costs of an installation, the cost of systems programming increased from 5% in 1950 to 50% in 1965. At the proceedings of a conference sponsored by SOFTWARE WORLD at the University of Sheffield in 1970, it was predicted that by 1976 this figure was expected to rise to about 80% [14]. There has been a radical shift in the balance of hardware and software costs; and computer technology has advanced to an era where hardware development costs are declining and the cost of computing is now dominated by the cost of software.

The demands in volume and complexity of software have outpaced the technology. Computer programs suffer phenomenal overruns in cost and delivery time, and the quality of the final product is often deficient in the area of correctness, adaptability and portability [7]. Often the maintenance costs have run beyond the original development price because of poor design and production.

Since the complexity of a program increases non-linearly

with size, the tendency for increasingly larger pieces of software to be developed is a prime factor in what has become known as the software crisis. The great complexity of large programs creates major problems both in terms of the number of errors that are written into programs and in terms of the difficulty of testing for these errors [14]. Many articles, phamphlets, and government documents have been written about "The Software Problem" and what can be done about it. The fact that there has been an accumulating body of literature on the topic is, in itself, indicative that some form of problem exists and that the future of the computer industry is dependent upon the manner in which software engineers direct their energies to solve it.

C. SOLUTION

In 1967 the NATO Science Committee, comprised of scientists representing the various member nations, proposed that an international conference be held that would focus attention on the problem of software. The phrase "software engineering" was deliberately chosen as being provocative in that it implied the need for software production to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering [31].

1. Engineering Foundation

Recently, an effort has been made to draw a correlation between the established principles of the engineering disciplines and the design of computer software. All of the various engineering disciplines have at least two things in common. First, they are based on and draw their

power from the use of known natural laws. Secondly, they use a design methodology that permits them to describe their designs in terms of a hierarchic structure; that is, in a form capable of detailing the design through successive levels of structure such as blue-prints and schematic drawings. Likewise, if the design of software is to be fundamentally related to the other engineering disciplines, then "software engineering" will require the formulation of a set of laws concerning the properties of software. It will furthermore require a hierarachical design approach that is discernibly structural in form and which permits the development of software descriptions equivalent to blue-prints and schematics. Perhaps software engineering cannot yet be viewed as the practical application of natural laws, but many of the same techniques that are useful to the engineer in designing bridges are useful for designing computer programs.

2. Software Profession

A computer system consists of hardware, firmware and software. Hardware deals with the design of the machinery and includes circuits, chips and peripheral devices. The design of firmware is concerned with aspects of computer architecture and organization such as word size, instruction format, register types, addressing schemes, memory hierarchies, storage requirements and input/output interfaces. Software, as noted above, encompasses the design of both systems and applications programs. When writing in low level languages, such as machine or assembly code, the programmer must have intimate knowledge of machine architecture, regardless of whether the firmware is hardwired or microprogrammed. As illustrated in Fig. 1, the software engineering profession includes the design of computer architecture and organization as well as systems

and applications programs [41].

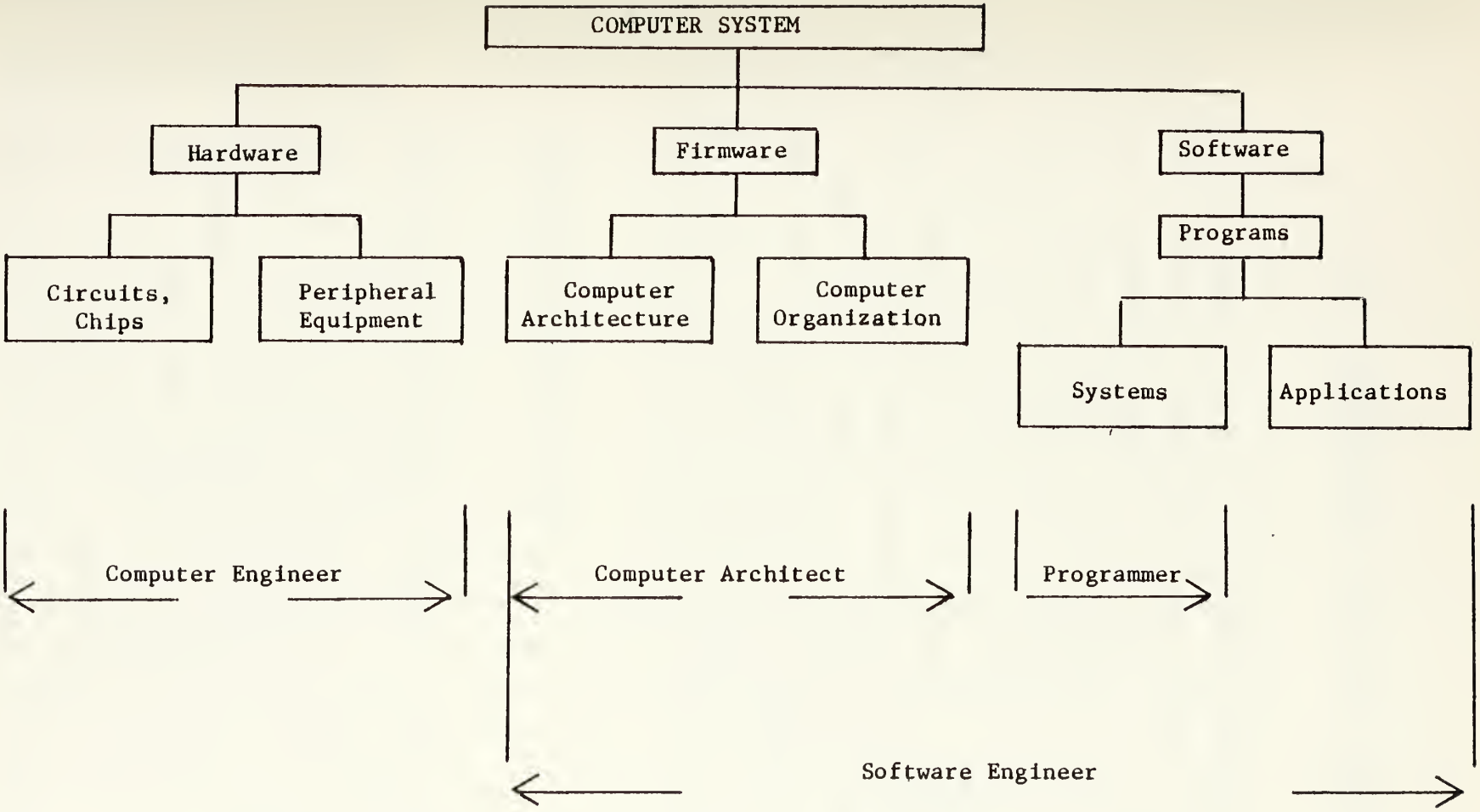


FIGURE 1 - PROFESSIONAL SCOPE

III. PROFESSIONAL TOOLS AND TECHNIQUES

As with other forms of engineering, software engineering also has its sets of techniques for solving problems and its tools for producing results. Whereas the rule of "higher quality implies higher price" applies to many products, it is generally acknowledged that programs will be cheaper if the errors are avoided in the beginning. In quest of this goal, the trend in DOD is to now place efficiency considerations subordinate to clear, logical structuring [7]. As a result, several design approaches and techniques have been found useful in trying to achieve quality programs. Known as programming methodologies, each has its following of steadfast disciples. Software engineers are somewhat constrained in their choice of methodology by available computers, languages and support systems.

A. PROGRAMMING METHODOLOGIES

Programming methodology refers to the method used to build a computer program. DOD has shown considerable interest of late in three specific methodologies: modular, structured and top-down structured programming. Each of the techniques is in some sense hierarchic in nature, although the approaches used differ considerably both in the degree to which they are explicitly hierarchic and in the implications they draw from hierarchic structuring. Although they can be implemented separately, the methodologies logically interconnect and build upon one another, as suggested by one author who describes a "top-down, modular

structured program" [27].

1. Top-down Programming

Top-down programming is an approach most programmers have used to some extent, but it is one that was not explicitly brought out until a few years ago. It was the topic of considerable discussion at the first NATO Conference. Perhaps the most succinct remark was made by Brian Randell on page 47 of the report: "There are two distinct approaches to the problem of deciding in what order to make design decisions. The top-down approach involves starting at the outside limits of the proposed system..." (by that he means the overall statement of what the program is to accomplish) "...and gradually working down, at each stage attempting to define what a given component should do, before getting involved in decisions as to how the component should provide this function. Conversely, the bottom-up approach proceeds by a gradually increasing complexity of combinations of building blocks" [31].

Normally, the bottom-up approach is used when coding a simple programming task with a given programming language. Individual routines are written first, then strung together to provide a complete program. With the top-down approach, the programmer starts with the problem and decides what are the main components that must be considered in order to solve the problem. It is at this point that the designer begins to puzzle over the best method to express them in terms of more primitive concepts.

Designing a program using the top-down approach is analogous to the formulation of a scientific hypothesis, which leads to and is confirmed by an experiment. The subsequent implementation similarly confirms the design, and

will include the making of minor adjustments and improvements to the design by debugging.

The term top-down programming is frequently used synonymously in the literature with "stepwise refinement", "levels of abstraction", "stepwise decompostion", "hierarchical design" or "top-down expansion".

2. Structured Programming

The term "structured programming" has gained wide currency in recent years in several different contexts. About all that the uses of the term have in common is that they refer to a programming methodology. The term was originally used by Dijkstra as the title of a paper presented at the NATO Conference in Rome, 1969 [5]. Later expanded, the paper was published as part of a book by the same name [10]. In 1967 Dijkstra reported on a moderate-sized multiprogramming system that he and his colleagues had built using his notions of stepwise decompostion and sequencing. He reported that no significant errors were found during the design and testing of the program [11].

Programmers immediately tended to see their own difficulties as the core of the subject and, as a result, widely divergent opinions on the theory have emerged. The controversy will continue as long as structured programming is approached definitively. Dijkstra uses the term to refer to the process he and his colleagues used. It is an approach, a way of thinking; it is not an algorithm. Yet popular usage of the term in many cases defines structured programming to be certain language structures or programming procedures to be followed without fail. In fact, approached as a collection of good programming practices, the

methodology will show encouraging results. If treated as a collection of inflexible rules replacing good judgment, it will doubtless lead to inefficiency [12].

a. Control Structures

Much of a program's complexity arises because the program contains multiple branches, making it difficult to follow the logic of the program and difficult to be sure at any given point in the program what the existing conditions are (such as variable values, and which paths of the program have already been executed). Futhermore, as the program undergoes modification during the debugging process, the complexity of the program grows accordingly. In maintaining the program, new code is often added if the programmer cannot find existing code that performs the desired function or cannot ascertain how the existing code actually performs the required function. The result is a program that is nearly unintelligible. Reducing program complexity is therefore the process of removing obscure structures, complicated control paths, and redundant and obsolete code from the program.

Improved program clarity can be attained through the use of self-explanatory variable and procedure names, succinct and informative comments, code identation which reflects the control flow to the reader, and simplification and limitation of detailed program function to that which can be readily expressed in less than a few dozen lines of code (a page). These simple concepts form the basis for most of the methodologies which have been advanced for improving program clarity.

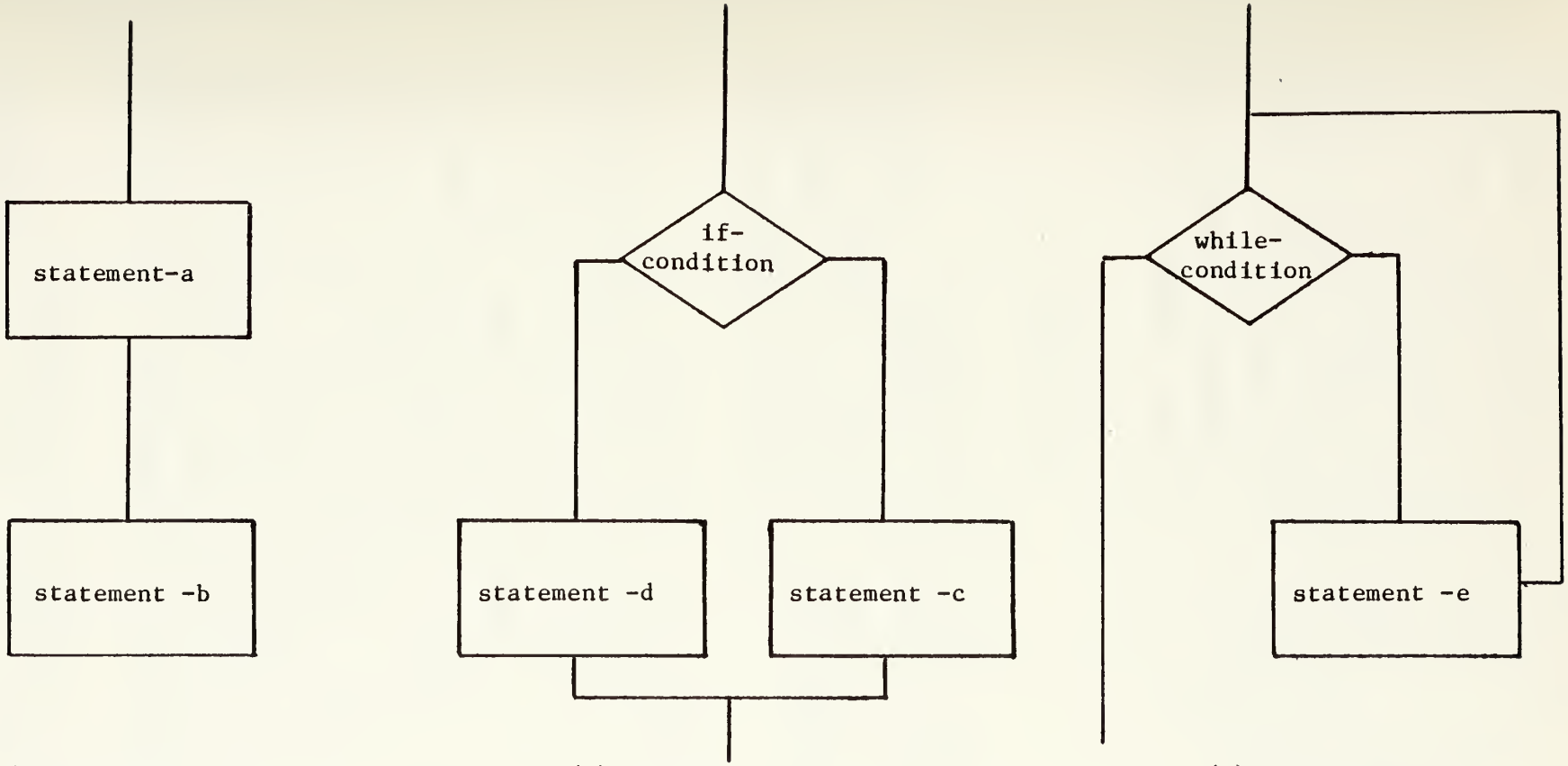
Bohm and Jacopini first proposed that inherently complicated program control structures were unnecessary and

that statement sequencing, conditional branching and conditional iteration would suffice as a set of control structures for expressing any flow-chartable program logic [3]. In a sense, Dijkstra advocated that the theoretically possible should become actual programming practice.

Fig. 2 summarizes the three structures identified by Bohm and Jacopini. The first diagram depicts sequencing from statement to statement. The two statements are of undefined internal complexity. Each statement could conceivably be a single instruction or an entire sub-structure. The center diagram represents the selective execution of alternative program segments. Again, there is no implication concerning the internal complexity of the two statements on the two legs of the condition; in fact, one of the statements may be empty. In this case only a single line is drawn and the written form is IF-THEN (without the ELSE-clause). Iteration is depicted in the third chart. In the example chosen here, the conditional test is made before the statement to be iterated is invoked. This is called a WHILE-loop, because the statement is performed as long as the condition stated is true. An alternative function can be introduced with the test performed after the statement is performed. Relevant theory makes the choice arbitrary, but one or the other should be used consistently in a program.

b. Sequencing Discipline

Each of the flowcharts share the property that they have a single entry (at the top) and a single exit (at the bottom). The three structures are alternately referred to in the literature as "concatenation", "selection" and "repetition" respectively. Strict adherence to these three structures is what Dijkstra refers to as a "sequencing discipline". Flowcharts of programs using only these



(a) Sequence
statement - a
statement - b

(b) Condition
if if-condition
then statement - c
else statement - d

(c) Iteration
while while-
condition
do statement - e

FIGURE 2 - CONTROL STRUCTURES

decompositions show a straight-line program (restricted topology) as compared with flowcharts of programs allowing multiple entry and exit points and lines drawn from any block leading into any other. Such a restricted flow pattern makes the program intellectually easier to manage as a programmer can readily envision a one-to-one mapping of the textual flow of the program to the computational flow. More simply, the structure of the computations is reflected in the printed structure of the program and the utility of a flowchart becomes marginal.

c. Abstraction

In addition to a coding technique and sequencing dicipline, structured programming embraces the idea of a design method. Dijkstra refers to it as stepwise refinement, but intrinsic to the understanding of the method is the concept of abstraction.

A "variable", for example, is an abstraction from its current value if it becomes necessary to change its value in the process of repetitively executing a sequence of code. There is also an abstraction involved in using some tool or device to accomplish a goal while totally disregarding the reason it functions as it does (e.g., using a mathematical theorm without considering how it was originally proved). A simulation might be regarded as a program model of an abstract machine. In regards to the complexity of a program, the concept of abstraction forces one to recognize the limitations of the mind and to use those limitations to advantage by writing programs which are intellectually manageable at each stage of the development.

Abstraction is therefore a tool for coping with complexity. A complex program should not be regarded

immediately in terms of computer instructions, but rather in terms of functional specifications formulated in some suitable notation, such as a natural language. In this way, an abstract program begins to emerge, with each functional level (or level of abstraction) subjected to the next lower level of abstraction. The stepwise refinement continues until a level is reached that can be processed by a computer. The reason for the restrictive control structures is easily seen here as they permit the designer to successively give greater structure to the functions without introducing new connections.

In essence, a top-down structured program may be viewed as consisting of a number of layers, the top layer being the overall definition, the bottom layer being the individual coded instructions using the basic control structures, and the various intermediate layers being definitions of functions in the whole system in terms of the more primitive concepts.

3. Modular Programming

Another methodology that has received increased attention is modular programming. The idea of breaking a large system into a number of smaller parts, or modules, is much older than the idea of structured programming. As defined by Liskov, a complex system is "one in which there are so many systems states that it is difficult to understand how to organize the program logic so that all states will be handled correctly" [22]. Even in the early days of computer programming, the technique of "divide and rule" was recognized as convenient method of approaching the problem of complexity. The term "modularization" has since been applied to the technique. The word "modular" means "constructed with standardized units or dimensions for

flexibility and variety in use". Applied to a programming methodology, modularity refers to the building of a program by putting together parts called program modules. In a modular program the "stardardized units or dimensions" are standards such that the modules meeting the maxims are conveniently fitted together to realize a large system. So the idea envisioned when the term modular programming is used is dependent upon the standards applied to a module description.

In its broadest description, the module is viewed as a group of program statements that are lexically together on the listing. The statements are bounded by identifiable boundaries (such as BEGIN and END statements) and are collectively referenced by a name (the module name). The statements can conceivably be called from any other part of the program. Although a great deal of flexibility is implied in the concept of a module, it is the primitive notion underlying the modular programming techniques espoused in the literature. As in the controversies surrounding structured programming, programmers in the past tended to refine the technique of "divide and rule" into methods that best met the demands of their programming environments. Individual authors will therefore vary in their descriptions of modular programming, depending on their insistance on the significance of such qualities as module independence, limited functional scope or program adaptability.

In essence, there are two classes of problems addressed in the literature: how should modules be interfaced to each other and how should they be defined. Interface design is concerned with passing of control and data back and forth between modules and the restrictions placed on the internal structure of the data [26]. A module's knowledge of the outside world is, therefore, an

important factor in the initial design process. The degree and method of interfacing is contingent on the functional scope (or definition) applied to a module.

a. Multiple Function Modules

One concept is to view the module as a procedure (subroutine or subprogram, depending on the language). The flow of control in a pattern described by a tree is characteristic of modular programs constructed as combinations of procedures. The top of the tree is the first code module; it depicts the program's overall control logic and functional capabilities. Intermediate modules in the program tree, in effect, summarize what is done by the modules below. The bottom modules are short programs which call no other modules. Thus the program itself becomes a principle tool of documentation. Using this concept makes it difficult to limit the number of functions performed by a module. Normally, modules viewed as procedures will take on multiple function characteristics, although most authors insist that an effort be made to keep functional scope to a minimum.

An example of this view of a module is suggested by Liskov [22]. At any point during the progress of a computation, one module (procedure) may initiate the activation of another procedure by specifying the input data. The new procedure activation is carried on, possibly making use of additional procedures, until it terminates, leaving a set of output data for use by the procedure from which it was activated. Calling sequences can only be downward in Liskov's method and return paths are the exact reverse of the calling paths. Her method further embodies the notion of stepwise refinement in the actual coding process as well as the use of Dijkstra's control structures.

Thus, higher modules are written first and tested at each level using stubs to represent the as yet uncoded lower modules. The performance of the lower level modules tends to be more or less dependent on the upper levels.

b. Single Function Modules

Other authors tend to be more restricitve in their definition of functional scope of a module, yet more restrictive in terms of accessibility to and from other modules. Maynard proposes that each module perform "a single logical function (e.g, READ INPUT FILE) or a number of small related logical functions (e.g., GROSS TO NET COMPUTATION)" [26]. Once the logical functions have been isolated, each module is coded and tested on its own. Only when all the modules are written and tested are they all linked together for final testing. Adaptability and individual module testing are considered critical factors by authors claiming the value of limited functional scope of the module.

D. L. Parnas has considered the problem of defining modules and proposed a particular strategy to follow [32]. His arguments are based on the observation that programmers get into trouble by implicitly assuming certain conditions to be true. His method is to divide a system into modules and make explicit statements about the complete context of each. That is, each module is described by a function to be performed, a set of inputs and a set of outputs. The programmer has free rein to build a module as desired, provided that only the information explicitly given for it is used. The division of the system is best made, according to Parnas, by encapsulating a single design decision in each module. Instead of making each module correspond to one step in a process, design decisions are made (for example, the representation of a data structure or

an algorithm for searching a table) and then as much as possible of the information about each decision is hidden in a module. In this way, it is ensured that any change in a design decision will cause minimal change in the system. Obviously, Parnas's goal was the design of a highly adaptable program.

c. NTDS Module

The Navy Tactical Data System (NTDS) represents a form of modular programming that was developed in response to a need for a large number of operational programs that were similar in many instances, but not identical. As the number of ships employing NTDS increased, it became evident that developing a unique program for each ship would involve considerable design and programming effort. Various ships were furnished with different equipment configurations and different operational requirements; yet many of the required functions were the same [30]. The Navy's goal, therefore, was to develop tactical programs that were highly responsive to changing mission requirements (adaptable) and equipment (portable).

The methodology used by the Fleet Combat Direction Systems Support Activities is referred to as "functional modular programming" [30]. Each module is viewed as an independent program which may perform one or more related tasks and is capable of being programmed and tested on its own. Once the mission, readiness condition and available equipment is defined, then the required modules are selected and compiled to form the NTDS package (program) for that specific ship. An executive module is installed in each system computer to provide for control of module execution. The executive modules vary in each computer only in their arrangement of the flag tables to provide for

either periodic or "upon demand" module execution. In addition, the executive module can delay low priority tasks during peak periods of loading.

Communication between modules is accomplished through the intermodule/intercomputer (IMIC) module resident in each computer in the system. A module with information for another module will pack a message in its output buffer area and reference IMIC. IMIC will then transfer the message to the receiving module and arrange for its execution by setting the appropriate executive flag. Obtaining data from another module is similarly accomplished, by request, through IMIC. Each module is responsible for its own local data store. Data that is common to several modules (e.g., velocity or track position) is stored in each computer and, insofar as possible, a single module is given the responsibility of maintaining a certain type of data within the common store area.

The method of packing messages for subsequent processing is a major contributing factor to the portability and adaptability of NTDS programs, as each module can be compiled into any of the systems computers. The executive and IMIC modules together with the common data store provide the elements necessary for overall program control. As is true with many software products, compromises to the original design principles are often forced by time or budget constraints. In such cases, it is not unusual to find globally-known information accessed directly rather than through IMIC.

B. DESIGN TOOLS

When constructing a small program, inconveniences and

inefficiences forced on the software engineer by poor development tools may not be noticed. When constructing a large system, however, even small losses of time on a repetitive operation can translate into months of wasted effort. A fundamental concern, therefore, is to provide the software engineer with good, reliable tools that fit the project at hand. The tools should be as general and powerful as needed for the operations expected on the project. Several of these tools and their impact on the design process will be investigated. Even though the "Chief Programmer Team" approach to a project is normally presented in the literature as a managerial vice a technical tool, the software engineer is responsible for the overall design and development of a project and this, of necessity, includes the management as well as the coding aspects. Accordingly, the team approach is presented here as a software engineering tool. Recognizing the particularly vital importance of computer languages as tools, a discussion of their design characteristics is contained in a separate section of this thesis.

1. Decision Tables

Ease of development corresponds closely to the systematic way the program was planned. Using tables to indicate decision logic is often an indication of a systematic approach. A decision table is a tabular form for displaying decision logic [20]. The literature on the subject refers to two types of tables: limited entry and extended entry. Limited entry tables frame the terms in the condition stub so as to constitute a Boolean with only two possible states (TRUE and FALSE or YES and NO).

An algorithm for the rotation of men between sea and shore billets is given as an example of constructing a

decision table. The following four conditions are considered:

C1: If a man has been at sea eight or more years, he must go to a shore billet, whether or not he requests shore duty.

C2: If a man has been at sea four or more years, he will go to a shore billet, unless he requests to stay at sea.

C3: If a man has been at sea less than four years, he will stay at sea.

C4: If a man is in a shore billet, he will go to a sea billet.

In a limited entry table, the number of possible rules is 2**n, where "n" is the number of conditions. In this case there are sixteen entries. The option of requesting sea or shore has been added to the original conditions to demonstrate the ease with which a decision table can be modified without completely redesigning the table from scratch. Note that the request will only affect the decision when an individual has been at sea between four and eight years. Therefore, it is not necessary to add a fifth condition. The rules are systematically drawn as follows:

```
C1:  Y Y Y Y Y Y Y Y N N N N N N N N
C2:  Y Y Y Y N N N N Y Y Y Y N N N N
C3:  Y Y N N Y Y N N Y Y N N Y Y N N
C4:  Y N Y N Y N Y N Y N Y N Y N Y N
```

Several of the rules can be immediately eliminated as being impossible situations. For example, a man cannot simultaneously be at sea for greater than eight years and

less than four. Fig. 3a represents the simplified decision logic remaining. The blanks indicate "don't care" situations. Note that the table can be streamlined even further by combining Rules 1 and 2 (the result is the same regardless of duty preference) and also rules 5 and 6 in addition to rules 7 and 8 (for the same reason). However, by listing the entire table, all possibilities are shown. Figure 3b is the resulting program translated directly from the decision table using the IF-THEN-ELSE control structure.

There are programmers who still persist in using flowcharts to describe computer logic. Although these charts are satisfactory in many instances, they have several inherent disadvantages, including the fact that they are not suitable for translating directly to code. In addition, depending on the complexity of the program logic, they can be particularly difficult to read. Decision tables overcome the disadvantages of flowcharting as the logic in the tables is stated precisely and compactly. Furthermore, complex situations are more easily understood as the decision table

ROTATION-TABLE	1	2	3	4	5	6	7	8
At sea ≧ 8 years	Y	Y						
At sea ≧ 4 years			Y	Y				
At sea < 4 years					Y	Y		
On shore							Y	Y
Request sea	Y	N	Y	N	Y	N	Y	N
Go to sea			X		X	X	X	X
Go to shore	X	X		X				

(a)

```
IF (at sea ≧ 8 years)
   THEN (assign to shore);
ELSE IF (at sea < 4 years)
   THEN (assign to sea);
ELSE IF (on shore)
   THEN (assign to sea);
ELSE IF (request sea)
   THEN (assign to sea);
ELSE (assign to shore);
```

(b)

FIGURE 3 - ROTATION TABLE

layout enables the programmer to systematically examine each possible combination of conditions to make sure that no possibility has been overlooked.

2. Chief Programmer Teams

A programming management technique called the "chief programmer concept" was developed by Harlan Mills and his associates at IBM [1]. In a paper describing an experiment using the concept [8], two major motivations for trying this approach are cited. One is the realization that, because of the newness and rapidly expanding nature of computing, many projects are staffed primarily by inexperienced people; at the same time, those with technical expertise are pushed into higher management where their contributions to the technical aspects of a project are limited. The second motivation is the observation that not much functional specialization is used on a project. A single person is typically responsible for designing, programming, coding, and testing a single module. The main feature of the chief programmer concept, as developed by IBM, is a functional organization centered around a competent, experienced person (software engineer) who has total responsibility for the technical development of the system. The chief programmer personally develops the overall system and programs the most difficult parts of it.

Other members of the team are chosen and assigned tasks primarily on the basis of whether or not they can extend the capabilities of the chief. Thus, other competent professionals may be placed on the team to help detail the overall design formulated by (or under the direct leadership of) the chief. Routine jobs, such as coding programs once detailed designs are available, removing syntax errors, and running simple tests, are carried out by junior members of

the team who have less experience than the chief and main assistants. Clerical duties such as key-punching, maintaining listings, and actually running jobs on the computer are given to a secretary or clerical worker.

The size of the group was not large. In the experiment reported by Baker [8], a team consisting of no more than eleven people produced a system of more than 80,000 lines of source code. For larger projects, division of the total task into separable parts permits utilization of the functional-specialization technique in each of the resultant subtask areas.

As described by Baker, there are three additional components of the chief-programmer concept: programming support libraries, top-down programming, and structured programming. The use of a programming support library is intended to isolate clerical functions from the technical aspects of system production. Such a library system consists of four main parts. There is an "internal library" of source code, load modules, and test bases in machine-processable form. An "external library" contains listings of the internal library and records of superseded versions of the system. A set of "machine procedures" for updating libraries, retrieving modules, link editing, testing and so on, is established. Finally, "office procedures" are followed by the clerical help in maintaining and adding to both the internal and external libraries.

The top-down design of the program and the use of Dijkstra's control structures is identical in description to that of Liskov's technique [22]. The overall control flow is implemented and executed first using stubs for lower level routines that have not yet been implemented. The choice of terms becomes a matter of semantics. Liskov prefers to view the technique as a form of modular

programming; Mills sees the technique as a modification of structured programming. As pointed out by Baker, the chief programmer approach basically contains nothing new. Its contribution, however, is that it has integrated for the first time in a production environment four existing techniques--functional specialization, programming support libraries, top-down design and structured programming. Additional detail and description of the "experiment" noted above can be found in Ref. [8].

3. Text Editors

Regardless of the programming methodology employed, a text editor offers considerable advantages to the software engineer. Text editors for entering and modifying online textual information range widely in power and usability. Simple program editors permit changes to be made only to entire records; more advanced editors operate on a character-by-character basis, thus eliminating a good bit of manual retyping. One of the underlying factors in the usefulness of editors to the software engineer is the unit of information with which they work. For some operations, working with an entire record is sufficient; in many, the ability to work with characters is a necessity. As a minimum, a useful text editor should provide for

* the insertion and deletion of source records;

* the global or singular substitution of character strings within records;

* the location of items both by context and record number;

* the preservation and retrieval of intermediate edit sessions in case of system failure;

* the ability to display the altered line (with optional graphic interpretation of non-printing characters) [36].

Desirable facilities for a text editor might include

* command "macros" for complex processing;

* extended pattern matching within locative strings;

* verbosity control;

* abbreviated notation for the contents of the locative string in the replacement string;

* reading input (source) from other files;

* moving or duplicating (multiple) records within a file;

* selective write of (multiple) records to a target file.

IV. LANGUAGE DESIGN

The primary purpose of a programming language is to serve as a tool for the software engineer in the most difficult aspect of the profession, namely program design. A programming language provides both a conceptual framework for thinking about algorithms and a means of expressing those algorithms for machine execution [35]. Thus, the language chosen not only determines how to express a problem; it also determines the scheme chosen for the problem solution.

A compendium of computer languages is beyond the scope of this thesis. The interested reader is referred to Appendix B where several authors are listed who have surveyed the primary features of common programming languages. Rather, an attempt is made here to identify the broad categories of available languages and the distinguishing characteristics of each in order to provide a basis for further study. The primary language design features supportive of structured programming are investigated along with requirements of DOD in regards to languages used in the tactical environment.

A. LANGUAGE CATEGORIES

All computer languages may be roughly categorized as either Machine-Oriented (MOL) or High Order (HOL). The MOL category can be divided into machine, assembly, and macro-languages.

1. Machine-Oriented Languages (MOL)

Machine language strings together numeric statements representing the value of the operating instructions to specify the sequence of operations a particular computer should perform. In an assembly language, alphanumeric statements, which generally relate on a one-to-one basis with the machine language statements for a particular computer, also give semantic information as to the nature of the instruction by ordering the alphanumeric symbols. These statements are also strung together in a sequence for the computer. Assembly language, as a rule, also contains some statements which relate to a sequence of several machine instructions. Macro-language is highly oriented to a structure which has a distinct sequence of machine instructions related to each alphanumeric statement.

The main characteristic common to these machine-oriented types is that they are a vehicle for communication between the programmer and the computer at the lowest logic level possible. This characteristic allows an optimization in both operating time and core space of the program and facilitates real-time input/output control. However, for a programmer who is not expert and for a reasonably challenging problem, the resulting program will often fall significantly short of the optimum.

2. High Order Languages (HOL)

HOL's allow the programmer to communicate with the computer in a language close in syntactic and semantic structure to the language in which a human thinks. This contrasts with MOL's which are close in structure to the

language with which the computer operates. It has been found that, in practice, programmers think in a class of languages too broad to be adaptable to a programming language [42]. High Order Languages are therefore designed to operate with a particular interest area in mind, and the syntax and semantics of the language are most compatible with the language of this area (e.g., FORTRAN with mathematics or COBOL with business). Thus, the programmer is freed from the job of guiding the computer through the task the program is to perform and can concentrate on how the problem should be solved and how the program should be organized to optimize the use of available resources.

a. Procedure-Oriented

Procedure-Oriented languages are used to describe algorithms. The coded algorithm consists of a group of ordered source statements. These source statements must be translated into a machine-executable form such that the execution order corresponds to that described in the algorithm. Thus the source statements control the order in which the machine-executable statements are performed. In addition, bookkeeping functions are involved which are not really part of the algorithm (e.g., assigning data storage).

b. Problem-Oriented

This language type is designed for a narrow class of problems where the programmer writes the program in terms of the problem formulation. An example is CSMP (Continuous System Modeling Program) which is a language for the scientific user who wishes to simulate a continuous dynamic system modeled by systems of differential equations.

Languages of this type differ from Procedure-Oriented languages in that the procedure for solving the problem is imbedded in the compiling program. Procedural languages require the procedure for solving the problem to be specified as part of the source program.

c. Nonprocedure-Oriented

This group of languages is sometimes called a "Very High Order Language" category in order to convey the notion of languages which in some sense are "higher" than COBOL, FORTRAN or PL/I. They are more commonly referred to in the literature as nonprocedure-oriented (or nonprocedural) because these languages are more concerned with the prcgrammer's goals than specific solution methods; that is, they seek to ask the question "what" rather than "how" [21]. For example, the programmer would be able to write "FIND INTEGERS A AND N SUCH THAT A**N<1000." Here the programmer has stated the problem, but is not concerned with how the complier actually solves it.

B. METHODOLCGY SUPPORT

In theory, there is no computer language for which the basic concepts of structured programming cannot at least be simulated. However, the complexity of a program diminishes and clarity increases to a marked degree if algorithms are described in a language in which appropriate control structures are primitive or easily expressed. As aptly stated by one author: "Programmers should never be satisfied with languages which permit them to program everything, but to program nothing of interest easily" [16].

1. Control Structures

Since the concept of restricting control structures followed rather than preceded the development of the majority of HOL's, it is not surprising that many current compilers do not directly support the methodology. Thus the control logic must either be simulated, using the native characteristics of the language [17], or a revised language syntax (language extension) must be introduced which is consistent with structured programming requirements. The latter may be accomplished in several ways. McGowan demonstrates a technique using the OS/360 Assembler F and a set of macros to realize the structures. The macros were based on a similar design developed and used by IBM for several years [27].

Another approach is to design a preprocessor to convert the source language into a format which the language compiler can readily translate. There are several types of preprocessors discussed in the literature (with particular concentration on FORTRAN), but there are basic similarities among them. A preprocessor is normally executed immediately preceding a program compilation (hence, the synonym "precompiler" or occasionally, "front end"). Its input is a set of programming statements, of which all or part are unacceptable to the compiler, and its output is a program in a syntax acceptable to the compiler. For example, a more or less FORTRAN-like structured language is designed along with a preprocessor to convert the programs written in this structured language into statements that the existing compiler will accept.

One of the major advantages of a preprocessor is that it does not require modifications to the existing

compiler. Conversely, its major disadvantage is that the programmer must contend with two variations of the program when tracing errors: the one coded using the language extensions and the output of the preprocessor which the compiler receives as input.

The use of preprocessors to achieve structuring in a language should be regarded as an interim solution. The long term objectives should be to modify current language capabilities. However, at this time, it is a valid mechanism. The primary function of a preprocessor is to improve the productivity of an installation by expanding the capabilities inherent in a compiler and its language. In accomplishing this, preprocessors also provide a test environment for various prospective control structures, thus giving the software engineer the opportunity to observe the effectiveness of each without actually changing the compiler. In addition, gradual changes in an existing language are less likely to cause discontent within the programmer community where there is a tendency to be protective toward a "favorite" language.

2. Block Structure

Block structure is another language characteristic that is, at least implicitly, associated with the structured programming methodology. In a block-structured language each program or subroutine is organized as a set of nested blocks, usally delimited as in ALGOL by the symbols BEGIN and END. Each block begins with a set of declarations which serve two purposes. First, each declaration sets up associations for one or more identifiers. These form the local referencing environment for the block. Secondly, some of the declarations may also define data structures or simple variables to be created upon entry to the block. The

psychological advantage of the block structure is to make the levels of nested logic (abstraction) immediately visible to the programmer. In addition, the block may be viewed as a parameterless subroutine (or module) that is coded in line at the point of call. Recognizing the advantages of such a nested feature as opposed to a strictly left-justified code sequence, several indentation aids have been developed. Meissner proposes a mechanism that is fully compatible with existing FORTRAN language features [28]. However, as with the simulation of control structures in an existing language, such aids should be regarded as only interim measures. Eventually, the step will have to be taken to change exisiting compilers.

C. NAVY TACTICAL LANGUAGE

It is generally agreed that MOL's facilitate program optimization, input/output handling, real-time control and, perhaps, debugging. Although each of these characteristics is vital within the Navy tactical community, the disadvantages of an MOL far outweigh the advantages. Machine-Oriented languages challenge the programmer with the details of coding, require difficult organization of the programming, and extend programming time and cost. Their key disadvantages, however, are that they do not offer the adaptive and portable characteristics necessary for the modular programs required by the tactical community. As a result, the Navy developed an HOL which contained the facilities for performing certain functions not common in most commercial High Order Languages.

In an article published in the 1973 AFIPS Conference proceedings [37], Raymond Rubey attempted to enumerate the peculiar characteristics of tactical military languages and

compilers. It should be noted, however, that of all the facilities listed by Rubey, there is not one which cannot already be found in several general purpose commercial compilers, albeit to varying degrees. For example, Rubey emphasizes the need for a military HOL to allow for the definition and manipulation of logical, Boolean, textual, and character data. Also, since the debugging and validation of a tactical military program is the most expensive part of the development, Rubey feels that the language itself must have a minimum number of error-prone features or syntactic constructions. However, time and expense alloted for the testing phase of software development is not peculiar to the military environment; therefore, neither is the user's desire for an error free compiler limited to the military community. There are even HOL's that provide for easy regression to MOL's, thus satisfying the military's need for input/output operations, real-time control and interrupt processing.

The difficulty, therefore, was not defining new, unique characteristics, but rather finding one language that encorporated all of the desired facilities identified by the tactical community. Rather than modify an existing extensible language, the Navy chose to define their own. The HOL CS-1 was the first such standard developed in 1960 [40]. It has since been superceded by the CMS-2 family, although NTDS still supports CS-1.

Reference [40] enumerates the High Order Programming Language requirements of the Navy's Tactical Digital Systems User community. It is interesting to note, requirements specifically state that the language must be supportive of structured programming. Although the usual control structures and block structure are included in the description of the methodology, it has been expanded to include recursion, modularity and the use of GOTO's (limited

to the containing block). Machine-dependent code may not appear in the source language program, thus insuring there are no conflicts between compilers of different host machines. The programmer must be able to control I/O operations in the HOL.

The compiler must be unforgiving; that is, defaults should be limited to implementation requirements vice user conveniences. For example, the type and initial values of each variable must be explicitly specified in the source program. In essence, the compiler should never attempt to read the programmer's mind. Most critical, however, the compiler should be amenable to future changes that may result from technological advances in software and hardware.

V. MILITARY SOFTWARE

The software "crisis" reported earlier is not limited to the commercial world. Total annual expenditures for system analysis, design and programming of software in DOD are estimated at $3-3.5 billion, divided among the services as follows: Army 23 percent, Navy 36 percent, Air Force 36 percent and other DOD agencies 5 percent [15]. Within the Air Force, for example, it is estimated that this expenditure represented four to five percent of the total service budget. By 1985, it is predicted that software will possibily represent up to 90 percent of the total hardware/software budget for DOD [7].

The rapidly decreasing costs of computation resulting from new technological advances has caused an expansion in the variety of applications within the tactical community. The result is not only more computer usage, but also the need for more software. In a speech presented to the National Aerospace and Electronics Conference in May 1976 [24], the Assistant Secretary of the Navy for Research and Development suggested that the software costs within the Navy tactical ccmmunity could partly be attributed to the problem of "proliferation" (i.e., each system provides its own computer, compilers and support software) which in turn affects the portability of tactical programs. He further suggests that the Navy will be seeking to capitalize on the law of "supply and demand" by looking more closely at the investments and advancements in the commercial sector in order to seek more commonality with commercial systems.

The commercial sector realizes more profit with a

reduction in software costs; DOD simply realizes a reduced budget. Regardless, their goals are the same. As in the case of programming languages, authors have attempted to define the unique characteristics of military tactical programs [6]. It will not be debated that perhaps efficiency is important in a tactical computer system where real-time response is critical; however, an unreliable program is worthless no matter how efficient. Therefore, whether the computer program to be produced is tactical or commercial in nature, the designers have one goal in mind--quality; and the quality of the final product cannot be divorced from its structure and design. It is not surprising, therefore, that the interest in structured programming has brought with it a wave of optimism within DOD where the expenditure of funds is more subject to public scrutiny.

Within the tactical community, a quality program is usually viewed in such terms as reliability, portability, adaptability and ease of verification and validation. Each of these characteristics is a primary factor in the overall development cost. The ways in which program structure can enhance these characteristics is discussed below.

A. RELIABILITY

The reliability of a program is defined in terms of its behaviorial pattern over time; that is, whether or not it will satisfy the stated operational requirements over a certain time interval. It is usually measured in terms of the degree to which a program is "free of errors" (validation) and whether or not the program does what it purports to do (verification). Dijkstra has stated that testing can be used to show the presence of bugs but not

their absence [10]. In short, the only way to guarantee a program is free of errors, is to write it correctly to begin with. However, as suggested by R. M. Graham at the Garmish NATO Conference, the all too frequent approach is to "build systems like the Wright brothers built airplanes--build the whole thing, push it off the cliff, let it crash, and start over again" [31]. Therein lies Catch-22. How does one guarantee that a program has been designed and written correctly from the start? D. L. Parnas argues that reliability and correctness are not synononymous. Program reliability can be improved by use of several techniques, while the production of truly correct software remains beyond reach [33]. It is fairly hopeless to establish the correctness of a program beyond even the mildest doubt, without taking structure into account.

1. Verification and Validation

The basic purpose of verifying and validating is to ensure that a program will perform its intended function at the time those functions are needed by the user. Large programs are never completely verified and validated for it would require the execution of an astronomical number of tests designed to exercise combinations of data through every data path in the program. Such a degree of testing is neither feasible nor practical. Structured programming (combined with some traditional coding practices, such as good annotation, descriptive lables, and judicious spacing in the source code) greatly clarifies source coding. The increased clarity and the reduced complexity of structured programs can contribute considerably to the testing process. Since the flow of control is less complicated in a structured program, the development and execution of test cases to adequately debug the program is simpler. Also since the program is more understandable, its correctness

can occasionally be proved by desk checking.

Program verifiers make use of formal specifications of the the program's intent that are written in some formal assertion language (such as predicate calculus). An input assertion defines the input domain of the program, and an output assertion defines the computation the program is intended to perform. Starting with the input domain, the verifier "walks through" the program and mathematically proves that the output assertions are satisfied whenever the input data meets the conditions specified by the input assertions [19]. Although such a mathematical approach to verification may never be practical, a corollary approach may prove of some practical significance. If a programming language especially designed to make verification easier is used, and if programs are structured to make it easier to state relevant assertions, then verification may become practical.

In the testing approach to program validation, the program is tested over a finite set of data by executing the program on that data. It is considerably easier to test a program in this manner than to prove its correctness over all cases. Several projects have been carried out to standardize and automate the program testing process. Some of the work has been directed toward the construction of program preprocessors which automatically insert "instrumentation statements" into a program [39]. The instrumentation statements keep track of how many times each statement or branch in a program has been executed during a run. The resultant statistics can be analyzed by the programmer to determine which parts of the program have been checked out.

2. Current Navy Research

Program structure analysis is concerned with the analysis of control paths of a program and the generation of test cases which systematically exercise the different paths. Research in this area is presently being conducted at the Naval Postgraduate School, supported by the Naval Air Development Center [4,38]. Working under the hypothesis that the ease of debugging and testing is related to structural complexity, researchers have developed simulation and analytical models to measure the relationship between error detection capabilities and program structure.

In the model, a program flow is represented in the form of a directed graph consisting of various nodes and arcs. The nodes represent merge and/or branch points within the program and the arcs represent sets of sequentially executed instructions between the nodes. Each test input defines a unique path from the start node to the exit node. The input proceeds along its predetermined path until an error is detected. It is then corrected and restarted along the same path, with the detecting-correcting-restarting process continuing until the end node is reached by the input. For each error encountered, measurements of test and correction time are made. In addition, statistics are accumulated on the number of errors detected in a fixed time, number of errors detected with a fixed number of inputs, mean time between errors, percent arcs traversed by one or more inputs, and percent errors remaining. Complexity is measured in terms of the number of nodes, paths, arcs and source statements; path length; and correctivity and reachability. In the simulation model, various probability functions are used to generate statistics such as the number of instructions per arc, the

number of instructions between errors, the arc traversed by an input at each branch node, and test and correction times.

The authors suggest several uses for the model. Comparing the error detection characteristics of several design alternatives can enable the software engineer to select the design that will be the least costly in the testing phase of development. Also, the model can be used to indicate the additional testing which may be caused by the increased program complexity. The relationship between complexity and error detection will assist the test manager in allocating resources to the programs to be tested.

B. PORTABILITY AND ADAPTABILITY

Portability and adaptability are particularly critical program characteristics within the tactical community where different equipment configurations and operational requirements are the rule rather than the exception. If the programmer is cognizant of the fact that the architecture or user needs may change, the program can be structured to accomodate the change using a high degree of modularity. Those program functions which will need the most attention upon transfer often can be isolated and functionally identified as distinct modules. The modules, if organized and documented properly, can be worked on with little reference to the rest of the program.

Adaptability and portability can also be enhanced by avoiding or isolating code that is difficult to transfer. For example, CMS-2 permits the programmer to intermix assembly code with high-level statements. One approach is to require that all assembly code exist in unique procedures. The most effective approach is to modify the

CMS-2 compiler to permit low-level macros, thus only the macros would require modification [25].

A program may be dependent upon aspects of system configuration in ways that make it infeasible to transfer the program to a system in which those aspects differ significantly. For example, a program written for a machine with a large amount of storage may be impossible to move to a machine with less storage unless initially written with this in mind.

VI. CONCLUSIONS

Although there is consensus in the literature on what constitutes quality software, there are almost as many approaches to building quality software as there are software designers. Most of these approaches do, however, have at least one common facet in that they are all in some sense hierarchic. Hierarchic design approaches have proved so far the most convenient way of simplifying the connection patterns in complex software.

The engineering analogy has had two effects on software development. First, the equivalent of production in traditional engineering fields is straightforward replication in software. What is referred to as software production is really analogous to building a prototype. It has important implications for management in that management of a software project is more research and development management than production management. Second, developments in the last decade have led to the production of software becoming an activity of an increasingly industrialized nature. With talk of software engineers' workshops and software factories, software production techniques have evolved rapidly and now the programmer has a wealth of design methodologies, tools and aids from which to choose.

Small software projects are frequently reported in the literature by authors and researchers who claim their success can be directly attributed to their use of sound and advanced programming techniques. Glowing reports on the success of large scale projects are noticeable scarce. Perhaps the success of the smaller projects is due more to

their size than to the techniques and tools of implementation. Regardless, it is difficut at times to distinguish between new, sound, software engineering principles on the one hand, and good sounding new ideas that would never work in practice, on the other hand. Complex systems require a very high caliber of staff who specialize in both the development and maintenance of software.

Preprocessors were developed to provide appropriate control structures in deficient languages. Yet, even in languages where the structures advocated by Dijkstra are primitive, programmers persist in their old coding methods. In January 1974, an analysis was made of a representative sample of General Motors' production PL/I programs. It was concluded that "programs were quite large, more difficult than necessary to read, and almost impossible to comprehend" [13]. These characteristics were attributed to several factors. For example, modularization was essentially avoided in order to conserve storage and call/return overhead. Variable names were not indicative of their functions, declarations were inconsistently indented and descriptive comments were sparse. Programmers appeared unfamiliar with the language they were using. The IF-THEN-ELSE form was used in only 17 percent of the IF-statements. The DO-WHILE structure appeared only 11 times out of 7385 total DO statements.

A study was conducted to describe the "average coder" involved in producing Navy tactical software [9]. The goal was to determine the level of user at which the complexity of a language should be targeted. It was found that, on the average, coders have two years of college, know two languages and have two years of experience. Yet, their personalities are "basically that of introverts. They are naive, lack aggressiveness, are non-gregarious and are

reluctant to venture into the unfamiliar". Such observations suggest a relationship between the personality traits of programmers and the coding practices observed in the General Motors' study.

At the Garmish NATO Conference it was suggested that those who felt a crisis existed were the "university types". The software crisis is, in fact, a crisis in education. The benefits to be reaped from modular, structured or top-down programmming are no longer open to debate, only to refinement of the techniques. However, as hinted above, the "good word" has not filtered down to the average coder. The language study further concluded that average coders are "under managed..." and under educated, "...which results in poor working habits and a lack of a sense of obligation to communicate technically" [9]. Well-trained and disciplined software engineers are needed so that there will be a professional standard by which to judge performance and to make comparisons. Only an experienced, highly trained professional should be given the responsibility of deciding which combination of tools and methodologies best meets the needs and constraints of the system under development. The major problem affecting DOD software is an institutional one. DOD should provide better incentives, education and career paths within the services for good software engineers.

APPENDIX A

DEFINITION OF TERMS

The following definitions are provided in an attempt to cope with the problem of ineffective communication in the field of software engineering. Although frequently amplified, if a particular author's definition adequately expressed, either in whole or in part, the generally acknowledged connotation, then appropriate references are cited. Frequently such definitions were not available as writers on a particular subject often tended to apply a narrow definition in the process of developing the thesis of their material. In such cases, an attempt was made to render a sufficiently clear and concise interpretation of the term as it was encountered in the major portion of the literature.

ADAPTABILITY: A measure of the ease with which a program can be altered to fit changing user requirements [34]. Less frequently used terms are "modifiability", and "changeability".

APPLICATIONS PROGRAM: A computer program such as payroll, inventory control, operational flight, satellite navigation, automatic testing, crew simulation and engineering analysis.

ASSEMBLER: A program which inputs a program written in assembly or macro-language and translates this into a program written in machine language.

ASSEMBLY LANGUAGE: A programming language in which there is a one-to-one correspondence between each assembly language statement and a machine language statement. Assembly language statements use alphanumeric symbology which suggests the statement function. The assembly language is in direct correspondence with a machine language and therefore relates to only one computer.

CALLED MODULE: A module that receives control from another module at an entry point and expects to return control to that module via a return point [26].

CALLING MODULE: A module that passes control to the entry point of a called module and expects control to be returned (via a return point in the called module) to the statement following the call [26].

CERTIFICATION: The process of endorsing a program as being of a certain quality [19].

CLARITY: The ease with which a person unfamiliar with a program reads code to determine its function and implementation.

COMPILER: A program which inputs a program written in a High Order Language and translates this to a program written in an assembly or, more usually, a machine language.

COMPUTER: Electronic machinery which, by means of stored instructions and data, performs rapid, often complex calculations or compiles, correlates and selects data. Examples: analog and digital processors, data processors, information and real-time control processors, electronic calculators, hybrid computers and communications processors [23].

COMPUTER DATA: A representation of facts, concepts or instructions in a structured form suitable for acceptance, interpretation or processing by communication between computer equipment. Such data can be external in computer-readable form or resident within the computer equipment and can be in the form of anolog or digital signals [23].

COMPUTER EQUIPMENT/COMPUTER HARDWARE: Devices capable of accepting and storing computer data, executing a systematic sequence of operations on computer data or producing computer outputs. Such devices can perform substantial interpretation, computation communication, control and other logical functions. Examples: central processing units, terminals, printers, analog/digital converters, tape drives, disks and drums [23].

COMPUTER PROGRAM: A series of instructions or statements, in a form acceptable to computer equipment, designed to cause the computer equipment to execute an operation or operations. Computer programs include systems and applications programs and may be either machine-independent or -dependent and general purpose in nature or designed to satisfy the requirements of a specialized process or particular user [23].

COMPUTER SOFTWARE: A combination of associated computer programs and data required to command the computer equipment to perform computational or control functions [23].

COMPUTER SYSTEMS: An interacting assembly consisting of computer equipment, computer programs and computer data [23].

CORRECTNESS: A computer program is "correct" if it actually

does what it purports to do and is free of all errors.

DATA BASE MANAGEMENT SYSTEM: A term which refers to a group of computer programs and files schema which, together, will asscciate the files and the data in various ways such that predefined and unanticipated information needs are satisfied. In the literature, a DBMS is also called a "data management system", or an "information retrieval system."

DEBUGGING: The process of locating and correcting an error that has been discovered as a result of testing.

DUMMY MODULE: An artificial module inserted in a object deck to satisfy a CALL from a module under test. A dummy module consists only of an entry point and a return point. Sometimes referred to as a stub.

EFFICIENCY: That quality of a program which relates to storage space and execution time.

EMBEDDED COMPUTER SYSTEM: A computer system that is integral to an electro-mechanical system such as a combat weapon system; tactical system; aircraft, ship, missile, spacecraft, certain command and control system; and civilian systems such as automated rapid transit systems. Its key attributes are:

* it is physically incorporated into a larger system whose primary function is not data processing.

* it is integral to such a larger system from a design, procurement and operations viewpoint.

* its outputs generally include information, control signals and computer data [23].

FIRMWARE: A program which is so basic it would be impossible to operate the computer without it and which

can therefore be thought of as being part of the machine. Often referred to as "hardwired-software".

FUNCTION: A description of what a program does. When speaking of a module function, then it is the transformation (input to output) that occurs when the module is called. In other words, a module's function is "what happens when the module is called" [29].

GENERALITY: A measure of the scope of functions that a program performs [29].

LANGUAGE: A set of symbols, with rules for the grouping of the symbols, that provides a means of communication between man and the computer.

LINKAGE EDITOR: A computer program which combines the outputs of language translators (assemblers and compilers) into executable phases. The linkage editor will attempt to resolve all external references in the routines being edited [26].

MACHINE INDEPENDENCE: Those qualities of a program making it independent of the details of the computer structure such as word length and types of registers.

MACHINE LANGUAGE: A programming language that can be interpreted directly by the computer for which it is intended; the internal operating language of the computer.

MACRO-LANGUAGE: An assembly language with the additional capability of providing a set of multiple-instruction blocks which are commonly used in programs [26].

MAINTAINABILITY: A measure of the effort and time required

to fix bugs in the program.

MODULAR PROGRAMMING: A programming methodology which defines a program as a set of interrelated individual units (called Modules) which can later be linked together to form a complete program [26].

OBJECT PROGRAM: The program in terms of the computer--presented in assembly or, more usually, machine language. This program is the "object" of the assembler's or compiler's efforts.

PERFORMANCE: A description of how well a program performs its function. It is measured in such terms as execution speed, storage size, resource usage, and mean-time-to failure.

PORTABILITY: A measure of the ease with which a program can be transferred from one machine environment to another. A highly portable program is one in which the effort required to move it is much less than that required to implement it initially [34]. Sometimes referred to as "transferability", particularly in the United Kingdom.

PROCEDURE-ORIENTED LANGUAGE (POL): A language used to describe an algorithm by using code consisting of a group of ordered source statements.

PROGRAM MAINTENANCE: Correcting, improving, adapting and extending of computer porgrams to further use.

PROGRAMMING METHODOLOGY: The "building" method used to produce a computer program from nothing.

PROPER PROGRAM: A program with one entry and one exit; that is, control is received with the 1st instruction and

returned with the last [3].

RELIABILITY: That quality of a program which can only be defined in terms of its behaviorial pattern over time; that is, whether or not it will satisfy the stated operational requirements for a specified time interval.

RELOCATABILITY: A measure of the ability to write a section of code without being aware of the core storage address which the code will eventually occupy.

SEGMENT: A term that is often used synonymously with "module", although many authors prefer to differentiate between the two. In such cases, a segment is a group of statements that are lexically together, bounded, and may or may not have a collective name. Modules would then be made up cf one or more segments and be referred to by name.

SEMANTICS: The relationship between meaning or concept, and expressions (symbol groupings) in a language. For example, the symbol "+" can have two meanings. It can denote an operation (summing) or it can denote a state (positive). The semantics of the language consists of the meanings assigned by the compiler to expressions.

SOURCE PROGRAM: The program written in assembly language or HOL by the programmer. This program is a "source" to the assembler or compiler.

STRUCTURED PROGRAMMING: A programming methodology which embraces the concepts of a design method, a sequencing discipline and a coding technique.

SYSTEMS PROGRAM: A computer program which forms a part of

the operating environment utilized by an applications program; examples include operating systems, assemblers, compilers, interpreters, data management systems, utility programs, sort-merge programs and maintenance/diagnostic programs.

SYNTAX: The rules of a language governing how the symbols of the language may be grouped to have meaning. Syntax does not relate to the meaning of symbols or groupings, but rather may be considered synonymous with the "structure" of the language.

TESTING: The process of supplying inputs and observing outputs to a program. The tester frequently has no knowledge of the program structure; normally he needs only to understand the function [38].

TRANSLATOR: A computer program which accepts as input a communication in a language interpreting the meaning of the communication. An assembler, a compiler and an interpreter are special cases of a translator [2].

VALIDATION: The process of determining whether executing the program in a user environment causes any operational dificulties [19].

VERIFICATION: The process of determining whether the results of executing the program in a test environment agree with the specifications [19].

APPENDIX B

ANNOTATED BIBLIOGRAPY OF SOFTWARE ENGINEERING LITERATURE

Seven topical categories were chosen as a means of indexing the principle software engineering reference material available at the Naval Postgraduate School. The individual listings are available through the Dudley Knox Library, the W. R. Church Computer Center Library, or the extensive collection of published and unpublished material maintained by Professor G. L. Barksdale, Department of Computer Science. Only material dated since 1970 was included. The exceptions were publications and articles which have become classics in the field. The list is by no means exhaustive. Rather, an effort was made to annotate material which provided sufficient general information to cover the topic or which contained a bibliography for further investigation. Cross-references to other categories are also provided. The seven categories are as follows:

A. General Concepts
B. Methodologies
C. Design Tools
D. Languages
E. Quality Characteristics
F. DOD Software
G. Management

****

A. General Concepts

****

ACM Computing Surveys, Special Issue: Programming, v. 6, No. 4, December 1974.

This issue covers a range of viewpoints about good programming by several well-known authors in the field of software engineering. The first two papers in the issue focus on the environment in which programmers work. Top-down and structured programming are addressed by Niklaus Wirth's article while Donald Knuth gives a concise, balanced view of the GOTO controversy. Kernighan and Plauger also contribute to the issue with a capsule presentation of several points made in their book, The Elements of Programming Style.

(B,G)

AFIPS Conference Proceedings, v. 44, p. 263-377, 1975.

This issue contains a special section consisting of position papers which concentrate on a number of fundamental issues related to software. Included are papers under the following categories: Software--Portability and Reliability; Programming--Art, Science or Engineering; Issues in Programming Language Design; COBOL 74--Its Impact on Software Engineering; Software Engineering; Operating System Theory; and Program Verification in 1980.

(E,D)

Bauer, F. L., Advanced Course In Software Engineering, Springer-Verlag, 1973.

This book represents the consolidated effort of a group of experts, prepared in a two-week seminar in Garmish and later presented at a course in February-March 1972.

The goal of the course was to take the first step toward identifying and making available teaching material on software engineering. Practically every aspect of the profession (including its tools and techniques) are covered in a series of lectures that are concise, yet easily understood. It serves as excellent introductory material for the student who eventually intends to conduct further investigation into a particular area of the profession.
(B,C,D,E,G)

Buxton, J. N. and Randall, Brian, Software Engineering Techniques, Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27 to 31 October 1969.
This conference is a direct sequel to the NATO conference on software engineering held at Garmish, Germany the previous year. The report summarizes the discussions held at the conference and includes a selection of working papers prepared by the participants. The major difference between the two conferences is that this conference was devoted to a more detailed study of technical problems rather than including the managerial problems which figured so largely at Garmish.
(B,C,D,E,G)

Cheatham, Thomas E., The High Cost of Software, proceedings of a symposium held in Monterey, California, NTIS, AD 777121, September 1973.
The objective of the symposium was to consider what research was needed to achieve a major reduction in software costs. Five workshops considered the areas of understanding the software problem, semantics of languages and systems, programming methodologies, software-related advances in computer hardware and problems in large systems. (B,C)

Computer, "Hardware vs Software: The Two Faces of Computers", v. 6, No. 11, November 1973.
The recent rapid developments in semiconductor technology have resulted in hardware being designed with insufficient regard for the special requirements of supervisory software. This issue addresses the problem of how hardware technology can most effectively reduce total computer system costs by attacking the predominant software portion; that is, moving significant portions of operating systems to firmware/Hardware.

Computer, "Software Engineering: The Age of the Software Factory Is Here", May 1975.
This special edition contains three articles on software engineering which respectively examine the basic principles and goals, the software factory, and reliability modeling.
(C,E)

Graham, Robert M., Principles of Systems Programming, John Wiley and Sons, Inc., 1975.
Systems programming is a broad field that encompasses many specialities. The author addresses such topics as operating systems, assemblers, compilers, loaders, memory managers, I/O and security at an introductory level of systems programming. A basic knowledge of assembly language for some computer is assumed.

Joslin, Edward O., Software For Computer Systems, College Readings Inc., 1970.
The first half of the book consists of a series of essays on various aspects of software engineering and is geared toward the manager rather than the technician. The second section consists of primers on COBOL and FORTRAN. The compendium is concerned with teaching the

basic concepts of the two languages without regard to any particular hardware configuration.
(D,G)

Kernighan, Brian W. and Plauger, P. J., The Elements of Programming Style, McGraw-Hill Book Company, 1974.
This book consists of a large number of programs in FORTRAN and PL/I which provides one or more lessons in style. The authors discuss the shortcomings of each program, rewrite it, then draw a general rule from the specific case. Each chapter ends with a summary of the rules presented. The book has become a classic in the literature and is frequently referenced by noted authors in the field. It is excellent reference material for the beginner as well as the experienced programmer.
(B,C)

Naur, Peter and Randall, Brian, Software Engineering, report of a conference sponsored by the NATO Science Committee, Garmish, Germany, 7 to 11 October 1968.
The Garmish conference is notable for the range of interests and experience represented amongst its participants. The goal was to identify, classify and discuss the problems, both technical and managerial which faced the various different classes of software projects. Thus, sections of the report are written for those who have no special interest in computers but who are concerned with its impact on society. Other sections are specifically directed toward managers, university officials or researchers in fields other than computer science. The major outcome of this conference was the realization of the full magnitude of the software crisis.
(B,C,D,E,G)

Proceedings of the 1st National Conference on Software Engineering, sponsored by the National Bureau of Standards and the IEEE Computer Society, Washington, D. C., 11 to 12 September 1975.

This publication consists of a collection of papers submitted to the various conference committees. Although some are technical in nature, the majority are broad overviews of methodologies, techniques and tools of interest to the manager.

(B,C,D,E,G)

Software World, Software 72, proceedings of a conference held at the University of Kent at Canterbury, 24 to 26 July 1972.

The Software 72 Conference, together with its predecessors Software 70 and Software 71, were a trio of conferences sponsored by Software World on the state of the art in the United Kingdom. They provide a basis for comparison with the problems faced in the United States. Certain terminology will be unfamiliar to the U. S. reader at first but can usually be interpreted from its contextual use. "Middleware", for example, is similar to firmware. Software refers strictly to systems programs.

(B,C,D,E,G)

Stewart, S. L., Concepts in Quality Software Design, National Bureau of Standards Technical Note 842, August 1974.

An edited summary is given of five seminars on quality software held at the National Bureau of Standards in 1972. The first three seminars provide a motivation for studies in quality software and a review of top-down and structured programming. The fourth provides a table of programming proverbs of use to the novice. The final

seminar is an introduction to a review of proof-of-correctness techniques.
(B,E)

Tou, Julius T., Software Engineering, Volume I and II, Academic Press, 1971.
These two volumes consist of papers presented for discussion at the Third Symposium on Computer and Information Sciences held in Miami Beach in December 1969. The first volume contains papers concerning computer organization, systems programming and programming languages. The second volume is devoted to information retrieval, pattern processing and computer networks.
(B,C,D,E,G)

Van Tassel, Dennie, Program Style, Design, Efficiency, Debugging and Testing, Prentice-Hall, Inc., 1974.
The book provides excellent introductory material for the beginning programmer on the style or readability of programs, program design, efficiency or optimization of programs, debugging, and testing. The five topics are augmented by a large number and wide variety of programming problems.
(E,G)

****

B. Methodologies

****

Armstrong, Russell M., Modular Programming In COBOL, John Wiley and Sons, 1973.
The book provides a well-defined framework and detailed guidelines for the implementation of modular programs in COBOL. The chapters are organized in blocks of progressively more technical material. Information of

primary interest to system managers, analysts and designers is placed early in the book, while material directed toward system project leaders and programmers appears in later chapters.
(D,G)

Computer, "Structured Programming: Highlights of the 1974 Lake Arrowhead Workshop", June 1975.
The goal of this workshop was to determine the industry-wide applicability of structured programming. It was therefore slanted toward applications and objective descriptions rather than technical material. Several chairmen summarized their sessions, while others submitted position papers and speeches.
(G)

Dijkstra, E. W., "Notes on Structured Programming", Structured Programming, Academic Press, 1972.
This article is a classic in the field of software engineering. It is tutorial on the methods of structured programming and the rationale for Dijkstra's techniques.

Griszl, L. R., Computer Program Modularization, TG 1223, Johns Hopkins University, September 1973.
The paper presents a five-step procedure for writing a complex computer program in such a way that the product is modular to the user as well as to the designer. The example used is that of a computer-reliant war game. The publication will be of interest to the programmer who is already familiar with the fundamentals of modular programming.

Maynard, Jeff, Modular Programming, Auerbach Publishers, 1972.

The author gives a concise yet thorough presentation of modular programming. It is primarily written to enable programming managers and programmers to comprehend and then implement the technique for their own use. Managers with some computer experience will get an appreciation of the potential benefits of modular programming from the first two chapters as the design of programs is discussed in some detail before explaining the actual workings of the method.

(G)

McGowan, Clement L. and Kelly, John R., Top-Down Structured Programming Techniques, Petrocelli/Charter, 1975.

The authors present a very detailed and extensive interpretation of structured programming. First proposing a preliminary answer to "what is structured programming", the authors then give an account of its major aspects including correctness considerations, structured coding, top-down design and integration, the chief programmer team approach to project organization and an extended example in PL/I. Excellent reading references are also provided. Highly recommended as initial reading material on the subject prior to any further investigation in the literature.

(G)

Parnas, D. L., A Review of "Structured Programming", NTIS, PB 223572, June 1973.

The report contains a detailed review of topics treated in Structured Programming in the form of three informal "open letters" to the three authors (Dahl, Dijkstra, and Hoare).

Parnas, D. L., "Some Conclusions From an Experiment in Software Engineering", AFIPS Conference Proceedings, v. 41, Part 1, p. 325-329, 1972.

This paper describes the outcome of an experiment to test the validity of some proposed software engineering techniques. The experiment showed that it was possible to combine the work of many programmers to produce systems which could exist in many versions. The results support the validity of the techniques being tested and conclusions about project management.

(G)

****

C. Design Tools

****

Gales, Laurence E., "Structured Fortran With No Preprocessor", SIGPLAN, v. 10, No. 10, October 1975.

Numerous articles are available in the literature on the design cf preprocessors. This paper offers an interesting contrast by proposing a method of designing structured FORTRAN programs using the native characteristics of the language.

(D)

Humby, Edward, Programs From Decision Tables, Macdonald and Co., 1973.

This book is concerned primarily with the translation of decision tables to computer programs and is not intended as a primer on drafting decision tables. The author demonstrates that for any given decision table there are several flowchart equivalents, some better than others, depending on the criteria set (i. e., storage requirements or average run time). Several methods of guaranteeing the best solution are demonstrated. An

extensive bibliography is provided for further reference material on the subject.

Kernighan, Brian W. and Plauger, P. J., Software Tools, Addison-Wesley Publishing Company, 1976.
The authors concentrate on two subjects. The first is how programmers can view substantial parts of what they do as tool building and tool using. By studying specific examples of general purpose tools, the authors show how programs can be packaged as tools, so other programmers will use them in preference to building their own. The second concern is how to write good programs. Rather than devoting specific chapters to ideas like structured programming and top-down design, the authors continually demonstrate their use throughout the numerous programming examples. All the programs are written in RATFOR (RATional FORtran) which is easy to read, write and understand by anyone having even a cursory knowledge of FORTRAN.
(B)

Ramamoorthy, C. V. and Ho, S. F., "Testing Large Software with Automated Software Evaluation Systems", IEEE Transactions on Software Engineering, v. SE-1, No. 1, p. 46-58, March 1975.
The authors contend that software tools are valuable in improving software reliability and attacking the high cost cf software. This paper describes in detail the many features of automated software tools and some software evaluation systems that are currently available.
(E)

****

D. Languages

****

De Remer, F. and Kran, H., "Programming in-the-large Versus Programming in-the-small", SIGPLAN, proceedings of the International Conference on Reliable Software, p. 114-121, June 1975.

The authors argue that two different types of languages are needed for programming in-the-small; i. e., one for writing the modules, and a "module-linkage language" for knitting the modules together. The software reliability aspects of such a module-linkage language are explored. (B)

Elshoff, J. L., "An Analysis of Some Commercial PL/I Programs", IEEE Transactions on Software Engineering, v. SE-2, No. 2, p. 113-120, June 1976.

The author scanned the source code for 120 production programs from several General Motors' computing installations, both manually and automatically, to consider five attributes: size, readability, complexity, programmer discipline and use of the language. Although the programs were written in PL/I, the author indicates that the observations and conclusions are typical of many installations.
(E)

Gannan, J. D. and Horning, J. J., "Language Design for Programming Reliability", IEEE Transactions on Software Engineering, v. SE-1, No. 2, p. 179-191, June 1975.

This paper identifies language features that enhance the reliability of programs and presents impirical evidence concerning the effects of some specific features. An

excellent bibliography is provided for further investigation.
(E)

Groams, David W., Programming Language Design, (A Bibliography with Abstracts), NTIS/PS-75/588, August 1975.
The bibliography contains 127 abstracts of research papers on the design, development and implementation of programming languages. The research includes specifications and applications for the programming languages in systems development and their use in specific cases such as interactive graphic systems, UNIVAC computers and others. The report also includes research on language compilers, syntax, semantics and logic modules. It covers the period 1970 to July 1975.

Hoare, C. A. R., Hints on Programming Language Design, NTIS, AD 773391, December 1973.
This paper presents the view that a programming language is a tool. It discusses the objective criteria for evaluating a language design, and illustrates the criteria by application to language features of both high level languages and machine code programming. An annotated reading list is also provided.

Meissner, Loren P., "On Extending FORTRAN Control Structures to Facilitate Structured Programming", SIGPLAN, v. 10, No. 9, September 1975.
The author attempts to identify some of the common features that can be perceived from the numerous preprocessor designs that have recently been espoused in the literature. He primarily is concerned with those language extensions designed for control structure augmentation.
(C)

Pratt, Terance W., Programming Languages: Design and Implementation, Prentice-Hall, Inc., 1975.
Computer programming language design and implementation are the two central concerns of this book. The software engineer, faced with the task of choosing a language appropriate to a given problem solution, needs to be able to evaluate the strengths and weaknesses of a language. The author organizes the study of language around the central areas of data operations, sequence control, data control, storage management, operating environment and syntax. Example analyses of seven languages are given.

SIGPLAN, Proceedings of a Symposium on Very High Level Languages, v. 9, No. 4, April 1974.
A "Very High Level Language" has been described as one which is used to specify "what" is to be done, rather than "how" it is to be done. The purpose of the symposium was to more adequately identify and define the characteristics of this class of languages. The papers are grouped according to the topics: Introduction, Set Oriented Languages, Data and Program Structures, Simulation and Modeling, and Specific Languages.

****
E. Quality Characteristics
****

Brooks, F. P., "The Mythical Man-Month", The Mythical Man-Month, Addison-Wesley Publishing Company, 1975.
This essay presents and interprets statistics on prediction versus actual time spent coding and debugging the development of various large software systems. The

book itself contains other essays relative to the management problems inherent in large programming projects.
(G)

Edwards, N. P., "The Effect of Certain Modular Design Principles on Testability", SIGPLAN, proceedings of an International Conference on Reliable Software, p. 401-410, June 1975.
This paper is a nonprogrammers view of design principles which are considered essential to testability of complex structures. The principles are are related to the programming problem.
(B)

Elspas, B. and others, "An Assessment of Techniques for Proving Program Correctness", ACM Computing Surveys, v. 4, No. 2, p. 97-147, June 1972.
While techniques of Proof of Correctness to verify software are not yet ready for practical application, many approaches offer promise for improving the correctness of software systems of the future. This survey indicates the current state of the art.

Fleiss, Joel E. and others, Programming for Transferability, NTIS, AD 750897, September 1972.
This document presents the results of an investigation of design and documentation techniques used in programming in order to develop recommendations and guidelines for program portability. The first part of the study presents guidelines that are language independent. The second section includes specific suggestions for improvements of FORTRAN, JOVIAL, and COBOL program design.
(B,D,E,G)

Linden, T. A., "A Summary of Progress Toward Proving Program Correctness", AFIPS Conference Proceedings, v. 41, Part I, p. 201-211, 1972.
This paper provides a summary of progress in developing techniques for proving that programs satisfy formally defined specifications. An extensive bibliography is provided for further research.

Liskov, B. H., Guidelines For The Design and Implementation of Reliable Software Systems, NTIS, AD 757905, February 1973.
This document describes experimental guidelines governing the production of reliable software systems. Both programming and management guidelines are proposed. Mostly the material covers information on structured and modular programming found elsewhere in the literature. However, the section on how to effectively select levels of abstractions provides several good suggestions.
(B)

Richards, F. Russell, Computer Software: Testing, Reliability Models, and Quality Assurance, NPS55RL, 740 71A, Naval Postgraduate School, July 1974.
The problems of measuring and assuring the quality of computer software are addressed. Mathematical models for estimating a quantitative measure of software quality is presented. Also included is a discussion of the customer's role in software quality assurance.

Schneidewind, Norman F., Analysis of Error Processes in Computer Software, NPS-55ss74071, Naval Postgraduate School, July 1974.
This paper describes a mathematical model for statistically analyzing software error detection and correction processes during software functional testing.

Using error detection histories as inputs, the model outputs forecasts of the future behavior of error detection and correction processes. Also, definitional ambiguities are resolved for key software error terms.

Schneidewind, Norman F. and others, Structure and Error Detection in Computer Software, NPS55Ss75021, Naval Postgraduate School, February 1975.

This paper reports on a FORTRAN simulation error detection model developed to investigate the relationship between program structure and error characteristics. A directed graph representing the program flow is input to the model as a node arc incidence matrix. The authors felt that it was not possible to draw firm conclusions from running three successive inputs into only 20 programs of increasing complexity. However, since the publication of this paper, several NTDS program modules have been placed in the form of directed graphs and used as inputs to the model.

Schneidewind, Norman F. and others, System Test Methodology, Volume I and II, NPS55ss75072A, Naval Postgraduate School, July 1975.

These volumes report the results of a research project covering the period 30 June 1974 to 30 June 1975 under the sponsorship of the Naval Air Development Center. The project addressed the areas of prototype testing, maintenance testing, software error detection analysis and issues in systems testing. Noting the impossibility of completely testing a complex system, the authors attempt to answer the question "How can a subset of the test inputs best be selected to thoroughly test the system?"

****

F. DOD Software

****

AFIPS Conference Proceedings, v.42, p. 787-816, 1974.

In a series of four articles, several authors attempt to identify the unique characteristics of military computer systems. Included in the discussion are tactical executive systems, hardware, languages and compilers, and operational programs.

(D)

Cooper, CDR John D. and Perkins, John D., Informal Report: A Description of the Average Coder Involved in Producing Navy Tactical Software, prepared for submission to the ODDR and E Committee on High Order Languages (HOL), May 1975.

This report presents a composite of characteristics which describe the individual at which the complexity of a new language should be aimed; that is, the "average coder".

(D)

Defense Management Journal, "Hardware/Software", October 1975.

This special issue addresses the problem of the increased use of, and dependency on, software in weapons systems and the management and production methods necessary to control its direct and indirect costs.

(G)

Department of Defense Directive Number 5000.29, Management of Computer Resources in Major Defense Systems.

This is an instruction which establishes policy for the management and control of computer resources during the development, acquisition, deployment and support of

major Defense systems. It directs that particular emphasis be placed on the review, analysis and validation of software. DOD recognizes that a primary means of accomplishing this objective is "to establish and/or maintain appropriate education, training, and experience career paths" for computer professionals.
(G)

Fisher, P. and others, Steps Toward Reliable Software: Proceedings of a Workshop Held at Falls Church, Virginia on November 19-20, 1974, NTIS, AD A010396, January 1975.
This report describes the proceedings of a workshop sponsored by the U. S. Army Computer Systems Command Research and Development Program. The objective was to identify potentially beneficial research approaches for improvement of software reliability in the military's software production environment. The group focused on abstract program development and refinement, modular top-down design, and program verification. A bibliography of references related to reliable software is provided.
(B,E)

Manley, LT Col John H. and Lipow, Myron, Findings and Reccommendations of the Joint Logistics Commanders Software Reliability Work Group, Volume II, November 1975.
This report documents over a year's work by 30 software professionals from DOD, civilian industry and the academic community. Volume II states the major problem and proposes solutions concerning the question of how to improve reliability of computer software embedded in military electronic systems. A bibliography of literature on software reliability is inculded.
(E)

Pryor, C. Nicholas, A Comparative Description of Several High Level Computer Languages, NTIS, AD A015335, 9 July 1975.

Several high level computer languages in use or considered for military applications are described, including FORTRAN, BASIC, ALGOL, PL/I, CMS-2, JOVIAL, CS-4 and SPL-1. The author investigates the basic statement types that are common to all the languages and compares them on a side-by-side basis.

(D)

SECNAV NOTICE 5230, Department of the Navy Short-Range Plan for ADP (FY 76-77), 14 January 1976.

This plan presents general guidance concerning the objectives, major strategies and significant actions to be pursued in the ADP Program of the Department of the Navy during the period 1976-1977. The general objectives of the program are listed as well as constraints faced by Navy management in the form of federal regulations and DOD policy. Of particular interest is the plan to develop an improved ADP career management program for both military and civilians.

(G)

SHAPM Management Strategy (Software): A Handbook, developed under the aegis of the Submarine Subcommittee, ASW Advisory Committee, and National Security Industrial Association, November 1975.

This paper is the draft of a handbook prepared for Ship Acquisition Project Managers (SHAPM) on how to plan and manage a submarine acquisition project so as to assure scheduled delivery of software. Three major events, called "gates", are outlined for management attention, as well as several inter-gate activities, which serve as a checklist on the effective utilization of project time. The stragety is summarized in a foldout chart at

the back of the handbook.
(G)

Syms, G. H., Notes on Modular Operating System Design: Specialization and Simulation of Basic Modules, Naval Electronics Laboratory Center, San Diego, California, January 1974.
The problem addressed is that of specifying and simulating basic operating system modules for the All Application Digital Computer System. The programs are useful for instructional purposes as well as research in modular OS specification and design. The report presents the results of basic modeling that is considered preliminary to the development of modular operating systems.
(B)

Tactical Digital Systems Office, U. S. Navy Tactical Digital Systems User's Requirements of a High Order Programming Language (HOL), Naval Material Command, MAT-09Y, 12 November 1975.
The document contains the High Order Programming Language (HOL) requirements of the Navy's Tactical Digital Systems User community. It is expected that the present standard, CMS-2, will eventuallly be superseded. The purpcse of the document, therefore, is to answer the question "What are the user's requirements of an HOL?"
(D)

****

G. Management

****

Baker, F. T., "Chief Programmer Team Management of Production Programming ", IBM Systems Journal, v. 2, No. 1, 1972.

This paper is representative of the works of both Mills and Baker in their efforts to improve reliability of software through new programming approaches. See SIGPLAN, International Conference on Reliable Software for other articles on chief programmer teams by these authors.

(B,C)

Ridge, Warren J. and Johnson, Leann E., Effective Management of Computer Software, Dow Jones-Irwin, Inc., 1973.

A new approach to cost problems in computer software is presented--the value engineering approach. This approach was originally designed to be used with hardware. Value engineering identifies and isolates the basic "function" of the study object, suggests alternatives, then evaluates each alternative in such terms as cost, practicability and potential roadblocks. The book is not written as a technical treatise for programmers. It is addressed to members of general management that are subjected to the impact of computers.

SIGPLAN, International Conference on Reliable Software, v. 10, No. 6, June 1975.

The purpose of this conference was to examine the meaning of software reliability and the problems involved from the standpoint of the customer, producer and user. The impact of reliable software on the public

at large is discussed as well as the importance of safeguarding the individual's right to privacy. The government's contribution to improving software quality is presented. Articles of particular interest from the conference are listed individually in this Appendix.
(E)

Weinberg, Gerald M., The Psychology of Computer Programming, Van Nostrand Reinhold Company, 1971.
The book is basically an expose of entirely under-estimated human factors in programming. Even factors like the size of a chair or distance to the nearest candy machine should be considered in the day-to-day programming environment. Weinberg's book contains numerous well-documented examples of the influence of these factors on the success or failure of a programming project. The concept of "egoless programming" is also introduced.

Weiss, David M., The MUDD Report: A Case Study of Navy Software Development Practices, Naval Research Laboratory Report 7909, 21 May 1975.
The MUDD report chronicles the development of a fictional system with requirements typical of Navy tactical systems. Material for the study was obtained from interviews with individuals responsible for the development of comparable Navy systems. A history of the decisions made during the development of the system is first given, followed by an analysis of the impact of each on the development and life-cycle of the software. The author makes recommendations on how the mistakes can be avoided in the future.
(G)

LIST OF REFERENCES

1. Baker, F. T., "Chief Programmer Team Management of Production Programming", IBM System Journal, No. 1, p. 56-73, 1972.

2. Bennett, Richard K., A Base for the Definition of Computer Languages, AD 664 086, Clearinghouse For Federal Scientific and Technical Information, October 1967.

3. Bohm, C. and Jacopini, G., "Flow Diagrams, Turing Machines, and Languages With Only Two Formulation Rules", Comm ACM, v. 9, No. 5, p. 396-371, May 1966.

4. Bradley, Gordon H. and others, Structure and Error Detection in Computer Software, NPS55Ss75021, Naval Postgraduate School, February 1975.

5. Buxton, J. N. and Randall, B., Software Engineering Techniques, Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27 to 31 October, 1969.

6. Chapin, George G., "What's Different About Tactical Military Operational Programs", AFIPS Conference Proceedings, v. 42, p. 787-795, 1974.

7. Cheatham, Thomas E. and others, The High Costs of Software, AD 777121, National Technical Information Service, September 1973.

8. "Chief Programmer Teams: Principles and Procedures", Report No. FSC 71-5108, IBM, Federal Systems Division, Gaithersburg, Maryland, June 1971.

9. Cooper, CDR John D. and Perkins, John D., Informal Report: A Description of the Average Coder Involved in Producing Navy Tactical Software, prepared for submission to the ODDR and E Committee on High Order Languages (HOL), May 1975.

10. Dijkstra, E. W., "Notes on Structured Programming", Structured Programming, Academic Press, 1972.

11. Dijkstra, E. W., "The Structure of the 'THE' Multiprogramming System", Comm ACM, v. 11, No. 5, p. 341-346, 1968.

12. Donaldson, James R., "Structured Programming", Datamation, p. 52-54, December 1973.

13. Elshoff, James L., "An Analysis of Some Commercial PL/I Programs", IEEE Transactions on Software Engineering, v. se-2, No. 2, p. 113-120, June 1976.

14. Evans, D. J., "The Current Software Situation", SOFTWARE 70, Proceedings of a Conference Sponsored by SOFTWARE WORLD, University of Sheffield, p. 19-29, April 1970.

15. Fisher, David A., "Programming Language Commonality in the Department of Defense", Defense Management Journal, v. 11, No. 4, October 1975.

16. Fisher, David A., Control Structures For Programming Languages, Ph. D. Thesis, Carnegie-Mellon University, Pittsburg, Pennsylvania, May 1970.

17. Gales, L. E., "Structured Fortran With No Preprocessor", SIGPLAN, v. 10, No. 10, October 1975.

18. Gill, S., "The Origins and Meanings of Software Engineering", Software Engineering, International State of the Art Report, p. 217-242, 1972.

19. IBM, Federal Systems Division, Structured Programming Series. Validation and Verification Study, Volume XV, AD-A016 668, prepared for Rome Air Development Center Army Computer Systems Command, 22 May 1975.

20. Kirk, H. W., "Use of Decision Tables in Computer Programming", Comm ACM, v. 8, No. 1, p. 41-43, January 1965.

21. Leavenworth, Burt M., "An Overview of Nonprocedural Languages", SIGPLAN Proceedings of a Symposium on Very High Level Languages, v. 9, No. 4, p. 1-12, April 1974.

22. Liskov, B. H., "A Design Methodology For Reliable Software Systems", AFIPS Conference Proceedings, v. 41, Part I, p. 191-199, 1972.

23. Manley, John H., "Embedded Computer Systems", Findings and Recommendations of the Joint Logistics Commanders Software Reliability Work Group, v. 2, p. 33-38, 1975.

24. Marcy, H. Tyler, To Master Evolution in Tactical Systems, speech presented at the National Aerospace and Electronics Conference (NAECON) 76, Dayton, Ohio, 18 May 1976.

25. Mathis, N. S., CMS-2 Software Transferability Study, AN/UYK-7 to AADC, Naval Electronics Laboratory Center, AD 755133, San Diego, California, 13 November 1972.

26. Maynard, Jeff, Modular Programming, Auerbach Publishers Inc., 1972.

27. McGowan, Clement L. and Kelly, John R., Top-Down Structured Programming, Petrocelli, 1975.

28. Meissner, Loren P., "A Compatible 'Structured' Extension to Fortran", SIGPLAN, v. 9, No. 10, p. 29-36, October 1974.

29. Meyers, Glenford J., Reliable Software Through

Composite Design, Petrocelli, 1975.

30. Naval Tactical Data System Programmers Guide, Volume I, prepared by Fleet Computer Programming Center, Pacific, San Diego, California, Manual M-5002, 1 December 1969.

31. Naur, Peter and Randall, Brian, Software Engineering, report on a conference sponsored by the NATO Science Committee, Garmish, Germany, 7 to 11 October 1968.

32. Parnas, D. L., "On the Criteria To Be Used In Decomposing Systems Into Modules", Comm ACM, v. 15. No. 12, p. 1053-1056, December 1972.

33. Parnas, D. L., "The Influence of Software Structure on Reliability", SIGPLAN International Conference on Reliable Software, p. 358-362, 21 to 23 April 1975.

34. Poole, P. C. and Waite, W. M., "Portability and Adaptability", Advanced Course In Software Engineering, Springer-Vertag, 1973.

35. Pratt, Terrence W., Programming Languages: Design and Implementation, Prentice-Hall, Inc., 1975.

36. Procedure For Ranking The Software Bases of Candidate Architectures For the Military Computer Family, prepared by The Software Evaluation Methodology Subcommittee of the Computer Family Architecture (CFA) Selection Committee, 5 April 1976.

37. Rubey, Raymond J., "What's Different About Tactical Military Language and Compilers?", AFIPS Conference Proceedings, v. 42, p. 807-809, 1974.

38. Schneidewind, Norman F. and others, System Test Methodology, Volume I, NPS55ss75072A, Naval Postgraduate School, July 1975.

39. Stucki, L. G., "A Prototype Automatic Program Testing Tool",AFIPS Conference Proceedings, v. 41, p. 829-836,

1972.

40. Tactical Digital Systems Office, U. S. Navy Tactical Digital Systems User's Requirements of a High Order Programming Language (HOL), Naval Material Command, MAT-09Y, 12 November 1975.

41. Tou, Julius T., Software Engineering, Volume I, Academic Press, 1970.

42. Weinberg, Gerald M., The Psycology of Computer Programming, Van Nostrand Reinhold Company, 1971.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
2.	Library, Code 0412 Naval Postgraduate School Monterey, California 93940	2
3.	Professor G. L. Barksdale, Code 52Ba Naval Postgraduate School Monterey, California 93940	1
4.	Professor Norman F. Schneidewind, Code 55Ss Naval Postgraduate School Monterey, California 93940	1
5.	Defense Documentation Center Cameron Station Alexandria, Virginia 93940	2
6.	LT Arrena S. Williams, USN Bureau of Naval Personnel Pers-3C21 Washington D. C. 20370	1

Thesis 166504
W5955 Williams
c.1 Software engineering:
tools of the profession.