NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

PRINCIPLES OF SOFTWARE
ENGINEERING ENVIRONMENT DESIGN

by

John Richard Frost

June 1984

Thesis Advisor: B. J. MacLennan

Approved for public release; distribution unlimited

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

Principles of Software Engineering Environment Design

5. TYPE OF REPORT & PERIOD COVERED

Master's Thesis
June 1984

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

John Richard Frost

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Naval Postgraduate School
Monterey, California 93943

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS

Naval Postgraduate School
Monterey, California 93943

12. REPORT DATE

June 1984

13. NUMBER OF PAGES

107

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

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, programming environment, software design

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

The history of programming languages, operating systems and computer hardware is briefly reviewed. Then the general methodology of established engineering disciplines is examined. Software "engineering" is then examined in light of its history and by analogy with the general engineering methodology. Here, a critical difference between software engineering methods and those of other disciplines is revealed. Software design is not separated (Cont)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

ABSTRACT (Continued)

from its implementation nor is there an effective means to communicate a software design from a designer to an implementor. It is shown that without an analog to the engineering blueprint, software engineering is not, and cannot become, a true engineering discipline. In following the engineering analogy, twenty-one principles of software engineering environment design are put forth. These touch on technical, management and ergonomic issues. Finally, it is concluded that work on software engineering environments holds much more promise for improved productivity than the traditional approach of programming language design.

Approved For Public Release, Distribution Unlimited

Principles of Software Engineering Environment Design

by

John Richard Frost
Lieutenant Commander, United States Coast Guard
B.S., Florida State University, 1969

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1984

ABSTRACT

The history of programming languages, operating systems and computer hardware is briefly reviewed. Then the general methodology of established engineering disciplines is examined. Software "engineering" is then examined in light of its history and by analogy with the general engineering methodology. Here, a critical difference between software engineering methods and those of other disciplines is revealed. Software design is not separated from its implementation nor is there an effective means to communicate a software design from a designer to an implementor. It is shown that without an analog to the engineering blueprint, software engineering is not, and cannot become, a true engineering discipline. In following the engineering analogy, twenty-one principles of software engineering environment design are put forth. These touch on technical, management and ergonomic issues. Finally, it is concluded that work on software engineering environments holds much more promise for improved productivity than the traditional approach of programming language design.

TABLE OF CONTENTS

I. INTRODUCTION

A. THE "SOFTWARE CRISIS"

In the late sixties it was realized that the importance of software was rapidly exceeding that of the hardware on which it was implemented. This was manifested by sharply escalating software costs while the cost of hardware underwent rather dramatic decreases. The reduced cost of computers increased the demand for them and hence their numbers and the number and variety of applications in which they were used also increased. There was a growing demand for the ability to convert existing applications software to make it executable on the newer, more powerful and less expensive hardware. The complexity and size of new applications also increased significantly with corresponding increases in the complexity and size of the software needed to support them. This in turn led to a far greater demand for software than the existing software industry could supply. Furthermore, it became apparent that software was not a "consumable" product that was used once or a few times and then discarded. It was becoming more and more like a large capital investment in a physical plant that required maintenance, alteration and enhancement throughout its relatively long useful life.

Unfortunately, it was extremely difficult to make even trivial changes to the software of the day in a reliable, efficient, and effective manner. This was especially alarming since the cost of developing software seemed to grow exponentially with its size and complexity. This meant that even after making a substantial investment in software to support a complex application, the user faced an even greater cost in maintaining its continued usefulness. In fact, it was found that most of the cost of a software system often occurred after it became operational leaving fewer and fewer resources available for new software development. It was from such observations and concerns that the term "software crisis" was born.

Unhappily, things have not changed much and so a decade and a half later we still speak of being in the midst of a "software crisis" even though this immensely popular, but inaccurate, description may have done as much to obscure the real issues as it has to call attention to the legitimate concerns of the industry.

Often the exact word used to describe an imprecisely understood situation is not of paramount importance. However, "crisis" usually describes a brief situation which is largely beyond the control of those affected and in which immediate, short term actions may significantly affect their chances for survival. The so-called "software crisis" clearly has not been brief and is not beyond the control of

those affected since they are also its creators. "We has met the enemy, and it is us.", from the comic strip *Pogo* by Walt Kelly accurately describes the current situation. Further, this "crisis" has not threatened the industry's survival, and immediate, short term actions, while often helpful, have not significantly reduced or altered the scope of the problem. Unfortunately, the perception of the software dilemma as a crisis has led to many proposals of limited scope, usually dealing with technical issues alone, that taken singly have had very little impact. If we are to have a more pronounced effect, we must broaden our outlook considerably.

B. SOFTWARE ENGINEERING

Shortly after the existence of the "software crisis" was recognized came the first hint that it wasn't really a crisis but merely an undesirable situation in urgent need of amelioration. In the early seventies the term "software engineering" began to appear in the literature with increasing frequency. This is significant because it implies a recognition of the need for a disciplined, orderly and effective way to approach the problem of producing high quality software efficiently. Rather than merely responding to the more glaring aspects of a "crisis", we can and should develop engineering methodologies for the formulation and analysis of problems having software solutions and for the

specification, design, development, implementation, maintenance, and evolution of practical software systems that solve such problems. We also need, of course, to include in our methodologies a way of determining when a problem does not have a feasible software solution.

Although the term "software engineering" is now in rather common usage, finding a definition of the term is surprisingly difficult, even in texts devoted to the subject. One particularly extreme example is [Ref. 1] where the page given in the index as containing that author's definition is completely blank! Nor is a definition to be found elsewhere in the text. However, some definitions can be found and we will state and analyze two of them here.

In [Ref. 2], Boehm presents the following definition:

> *Software Engineering:* The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them.

and in [Ref. 3] Jensen and Tonies offer Bauer's definition taken from [Ref. 4]:

> The establishment and use of sound engineering principles (methods) in order to obtain economically software that is reliable and works on real machines.

The first definition has rather little to offer since the base of well established "scientific knowledge" in software design and construction is still quite small. However, this should not concern us too greatly. Humans were undertaking engineering projects of significant size and complexity, some of which have lasted more than a thousand years, long before there was an established base of scientific knowledge applicable to them. In other words, engineers have never relied on scientific knowledge alone or waited for scientific advances before attacking pressing problems. Experience, ingenuity, and just plain trial-and-error have long been hallmarks of the engineering profession.

The second definition may have more to offer. In the words of Jensen and Tonies [Ref. 3] it "...encompasses the keywords that are the heart of all engineering discipline definitions: sound engineering principles, economical, reliable, and functional (works on real machines)." The critical question is whether we can reason by analogy from the sound engineering principles of other (non-software) engineering disciplines in general to a set (not necessarily complete) of sound engineering principles for software engineering in particular. Another somewhat less critical question is whether principles that at first appear unique to software engineering can be effectively applied to other

disciplines. The former question will be of great concern throughout the remainder of this thesis.

C. THE SOFTWARE ENGINEER

We turn now to the problem of defining, or at least identifying, the software engineer. In other engineering disciplines, we find "engineers" in many different roles. These range from that of a technician with limited formal education to that of a researcher with a doctorate degree. They also include many managerial functions. There are "chief engineers" who manage the engineering resources of a company, "project engineers" who are concerned with only a specific project or product line, "design engineers" who create and document designs, "production engineers" who determine how designs are to be implemented (manufactured), "quality assurance engineers" who devise and carry out tests on subassemblies and the finished product, "maintenance engineers" who perform preventive maintenance, repair and field alteration or upgrade functions, etc. In short, when we use the term "engineer" with respect to a particular product, we are speaking of anyone employed by the manufacturer who is responsible for any of the technical aspects of that product or directly manages those who are. Even the most cursory survey of the literature will show that all of the above functions have already been identified as being important to software engineering.

D. SOFTWARE ENGINEERING ENVIRONMENTS

The problem with which we are faced can be stated as follows: "How can the productivity of competent software engineers at all levels be substantially improved?" Since competence is assumed, we must look at how these people are organized and used, and at the conditions under which they are required to work. In other words, we must examine the work environment.

Environment tends to be a rather all-inclusive term. It includes such obvious things as lighting, the architecture of the building, layout of office spaces, air quality, noise level, etc. It also includes the equipment and tools used for the production of products or as aids in carrying out other necessary functions (specification, design, maintenance, quality assurance, project management, etc.). Equipment and tools typically include all of the machinery and tools on the production floor, drafting equipment (design and design documentation), test equipment and accurate measuring devices (quality assurance), and computers.

In this thesis, we will focus on the equipment and tools portion of the total environment. We will define, for purposes of this discussion, a software engineering environment as the set of automated tools available for carrying out the various activities associated with (1) the formulation and analysis of the target problem, (2) the specification, design, development, implementation, quality

assurance, maintenance, alteration, and enhancement of the software to solve that problem, and (3) the management of the personnel and other resources used in all these activities. The purpose of this thesis is to present principles that should be used to guide the design of such environments.

E. OUTLINE OF THE THESIS

In the next two chapters we will build the basic knowledge base needed in order to pursue a meaningful discussion of software engineering environment design. Chapter II will provide a brief historical perspective on how our present views of computers and software evolved. The objective will be to identify as many prejudices and hidden assumptions as possible so we may relieve ourselves of their burden. The three main areas of discussion will be the programming language, operating systems, and hardware developments of the past 40 years.

Chapter III will present and discuss an outline of the general engineering methodology which seems to be common to all current fields of engineering. The objective here will be to provide a generalized model of the activities we call "engineering" with a view toward applying this model to software engineering in later chapters.

Chapter IV will be concerned with software engineering issues. Given the background material on the general

engineering methodology and the historical perspective of the computer industry, we will analyze why "software engineering" is not yet a mature engineering discipline. We will suggest how the maturation process might be accelerated through the software engineering environment concept.

Chapter V will be concerned with some software management issues. Although we will not have time to examine these issues very deeply, we will emphasize their importance to the overall software engineering process and we will point out some types of automated aids which a software engineering environment could provide.

In Chapter VI we will discuss ergonomic issues. That is, we will look at the man-machine interface and emphasize its importance to the successful use of interactive tools. We will also argue that an integrated environment is needed, whereas a mere "toolkit" of loosely related automated aids is not sufficient.

Finally, in Chapter VII we will draw some conclusions and address some philosophical issues. In addition, the principles which will be developed throughout the thesis will be gathered together in a compact form and placed in the appendix.

II. HISTORICAL PERSPECTIVE

A. PROGRAMMING LANGUAGES

1. Introduction

Even though the general history of programming language evolution is well known, we will review part of it here for the purpose of highlighting certain events and concepts that are key to our understanding of the present state of affairs regarding software engineering environments. We will accomplish this by following a line of descent that leads more or less directly from the first major high-level language, FORTRAN, to one of the most recent - Ada. In particular, we will wish to ponder the question of which software engineering problems are best addressed by programming language design and which are best addressed by other means.

In [Ref. 5], MacLennan develops a number of principles which can be applied to the design of programming languages. However, these principles are, by and large, applicable to most engineering design problems and are not specific to language design. For this reason, they are listed in Appendix A for ease of reference. In spite of their general nature and the fact that he uses some very early programming techniques (pseudo-code interpreters) and languages (FORTRAN and Algol) to illustrate them, only one

of the principles he cites seems to have been consciously followed in those early days. Significantly, it is the first one mentioned in [Ref. 5] and is quoted below:

The Automation Principle

Automate mechanical, tedious, or error-prone activities.

The key to programmer productivity seemed to lie in providing higher levels of abstraction for programming purposes which could then be reduced to machine level instructions by automatic means. Given the lack of previous experience and the hardware limitations of the period, the higher level languages developed prior to 1970 turned out remarkably well. However, there were some unspoken, perhaps even unconscious, assumptions about software that became increasingly false with the passage of time. These included such assumptions as

Programs apply to specific, well-defined problems.
Programs are at most a few thousand lines long.
Programs have a relatively short life expectancy.
Programs are rarely modified.

While the incorrectness of some of these became apparent to writers of "systems software" (e.g. operating systems and compilers) as early as 1960, their incorrectness with

respect to "applications software" (i.e. programs written by users for their own purposes) was not appreciated until much, much later. Because high-level languages were thought of as applying more to applications software than to systems software, this lack of foresight was the main contributor to the shortcomings of high-level languages and the paucity of other software development tools for many years.

2. FORTRAN

The first major high-level language was FORTRAN. It was developed by John Backus of IBM between 1954 and 1958. FORTRAN is an acronym for FORmula TRANslation and this is precisely what the language was designed to do. It was aimed at the numerical problems of the scientific community. It was heavily machine dependent, as the correspondence between its control structures and the branching instructions of the IBM 709 computer shows. From a linguistic point of view, it was "grown" more than designed. In fact, Backus is quoted in [Ref. 5] as saying (in 1978), "As far as we were aware, we simply made up the language as we went along. We did not regard language design as a difficult problem, merely as a simple prelude to the real problem: designing a compiler which could produce efficient programs." Although FORTRAN was to come under heavy fire on linguistic grounds in later years, it was enormously successful. It proved the viability of high-level languages when many doubted their feasibility and it was a huge

commercial success. Much of its initial machine-dependence was removed in later versions and it became available on the computers of almost every manufacturer.

3. Algol

After FORTRAN proved that programs written in high-level languages could be automatically translated to efficient and logically equivalent machine language programs, a number of computer scientists on both sides of the Atlantic decided that a new "universal", machine independent language suitable for communicating algorithms between humans as well as between humans and machines was needed. The end result was a language called Algol. The work on Algol produced a great many significant advances in programming language design, probably more than any other single piece of work to date. Nevertheless, its goals were essentially the same as those of FORTRAN and it too suffered from the tacit assumptions stated above.

Algol suffered from another problem as well. In their enthusiasm for increased expressive power, the designers of Algol included some very general control structure constructors. These made it possible to represent some very sophisticated algorithms very compactly, but at the same time made those representations almost unreadable and in some cases misleading to all but the most experienced. Despite this generality, Algol remained a relatively compact language until its 1968 (Algol-68) incarnation.

4. PL/I

Algol wasn't the only language to suffer from the drive to generalize. While FORTRAN and Algol were aimed at scientific computing, another language named COBOL (COmmon Business Oriented Language) was developed. Having never jumped on the Algol bandwagon, IBM wanted to develop a "universal" language of its own aimed at both the scientific and business communities. It therefore began an effort in 1964 to extend FORTRAN with ideas from COBOL and Algol. It soon became evident that instead of an extended FORTRAN, a completely new language would result. This language was called PL/I (Programming Language One). It was an extremely large and complex language, the mastery of which was next to impossible. It had so many features which could interact in so many different ways that it wasn't even possible for an individual programmer to learn only a subset of features for his own use while ignoring the remainder. The drive to incorporate in one language all the features that any programmer could possibly want or use very nearly resulted in a language that no programmer could understand.

5. Pascal

Other significant events in software development were taking place. In 1966 Bohm and Jacopini published [Ref. 6] proving that *any* flowchart can be converted to one containing only certain types of flowchart elements - the so-called "structured programming" forms. The significance

of this work was that it showed complex and undisciplined control structures to be unnecessary. In the years that followed, many computer scientists argued that use of complex control structures was also unwise. A new concern for such things as software readability and reliability was growing as some of the earlier assumptions about the nature of software began to crumble under the weight of experience. Unfortunately, those assumptions listed earlier still remained largely intact as shown by the following excerpt taken from Dijkstra's now famous 1968 letter [Ref. 7] to the editor of the *Communications of the ACM*: "My first remark is that, although the programmer's activity ends when he has constructed a correct program, ..."

In 1970 the language Pascal was introduced. Its control structure constructors were simple and direct implementations of those associated with the new concept of "structured programming". Pascal was also a strongly typed language and had a block structure similar to that of Algol. This new language represented a quantum change in the nature of programming languages in its attempt to encourage and/or enforce certain programming practices. The emphasis on programming style was consistent with its goal: to be a suitable language for teaching programming. Because of its small size, excellent (though certainly not perfect) design, rich and efficient set of data structure constructors, and its strong typing it has not only met its original goal; it

has been successfully extended and used in many "real world" applications. Like Algol before it, Pascal has influenced the design of almost all later languages, including one of the latest - Ada.

6. Ada

Finally during the seventies the early assumptions about programs having a short life, being rarely modified, being relatively small, and being applied to specific, well-defined problems were revealed and discarded - violently. Everywhere one turned in the computing world, the phrase "software crisis" seemed to be emblazoned in neon lights. The situation wasn't far short of a general panic. Everyone seemed to have an idea of what the "key" problem was and thought that solving that one problem would cure most, if not all, of the software industry's ills. Of course, for each such perceived problem there was at least one proposed solution. The only central point of agreement was that *something* had to be done.

Into these stormy seas was launched the Department of Defense project to develop a new standard language to meet the needs of software development for "embedded" computer applications, i.e. situations where a computer is contained in and an integral part of a larger system (e.g. weapons systems and command, control and communications systems). DOD's experience with such software had been an expensive one up to this time. A wide variety of languages

were employed and some of them were used nowhere else in the world. This made software maintenance and the integration of subsystems from different vendors extremely difficult. The outcome of the work to overcome these and other difficulties was the programming language Ada.

Like PL/I, Ada has drawn on several earlier languages and like PL/I it is a very large and complex language. However, the reasons for its large size and complexity are quite different from those of PL/I. The two main driving forces behind Ada's size are its generalization and improvement of Pascal features and its attempt to incorporate a number of software engineering/management functions. For example, software design reusability is addressed by *generic packages* which contain templates for generating Ada code. Like many other languages, Ada supports separate compilation of modules but unlike them it also provides a considerable amount of error checking across modules. It remains to be seen if these and the many other features introduced by this new language have been chosen and implemented with sufficient care to avoid repeating the PL/I experience.

Another aspect of Ada which is of particular interest here is the inclusion of an Ada program support environment (APSE) in DOD's overall software development strategy for embedded systems. The rationale for this is described in [Ref. 8]. Unfortunately, the implementation of

a standard APSE is still far in the future even though at least one Ada compiler has already been implemented and certified. Nevertheless, the ongoing DOD effort stands as the most comprehensive approach to a software engineering environment to date.

7. Summary

The first high-level language had only one goal: the automatic translation of mathematical notation to efficient machine language programs. Later, more expressive power and generality were incorporated with little thought given to the consequences or implications of such a move. These traits and the changing nature of the software being developed caused many to question the wisdom of large complex languages. Many programming practices were questioned and a number of techniques such as structured programming, modular programming, etc. were devised. Out of this work came languages that in addition to performing the translation function also encouraged these newer techniques. While this initially resulted in a smaller, simpler language, the latest offering has tried to address so many programming issues that its great size and complexity has led many to doubt its viability. To quote C. A. R. Hoare [Ref. 9] on Ada, "We relive the history of the design of the motor car. Gadgets and glitter prevail over fundamental concerns of safety and economy."

B. OPERATING SYSTEMS

In the very earliest days of electronic computers, those who designed and built them also operated and programmed them. The early machines were dedicated to single users. Before a program could be run, there was a certain amount of "setup" work to be done to get the machine ready. It wasn't long before the operators/programmers wrote software to automate some of the setup activity. This marks the beginnings of what we now call operating systems.

As long as computers were dedicated to single users, operating systems were only used to provide services to the human operator. Later, as computers grew into complex computing *systems* capable of processing many programs concurrently and utilizing a large number and variety of peripheral devices, software was developed to manage system resources. The objectives of this management included (1) *security* to control the interaction between applications programs and the hardware and to keep applications programs from interfering with each other, and (2) *maximum throughput* to ensure that the system's resources were utilized as efficiently as possible.

By this time computers were being widely used commercially and customers demanded, in addition to an operating system to manage hardware resources, that a growing number and variety of service or utility programs be supplied as well. Vendors supplied file management subsystems, high

level language compilers, accounting software for resource usage, etc. Since all of these were tightly coupled with the "bare" operating system and were supplied along with it in one package, the entire package came to be called "the operating system." As the hardware continued to grow in capability while decreasing in cost, interactive use replaced the previous batch mode of operation. This led to even more demands on the operating system and its associated utilities. Interactive communications with large numbers of terminals, on-line utilities for file creation and manipulation, and text editors were but a few of the many additional demands placed on "operating systems."

What all this means is that, in effect, software developers have come to view the operating system as the "software engineering environment." Unfortunately, so many demands are placed on an operating system that its developers can give only limited attention to software engineering utilities. So far, these are usually limited to assemblers, compilers, subroutine libraries, a limited object code librarian facility, linkers, loaders, general purpose text editors and file manipulation utilities, assembly level debuggers, and, with luck, source level debuggers that are consistent with the compilers. As we will see in Chapter IV, these tools address only a small fraction of the fundamental needs of a "software engineering environment."

C. HARDWARE

We will not recount the history of computer hardware here since it is so well documented elsewhere. However, it is important to examine the changing nature of human-machine interaction that has resulted from hardware advances.

We have already noted how the difficulty of machine language programming led to the invention of high-level languages and how the desire to automate operator functions led to the development of operating systems. Another important characteristic of the earlier computers was that they operated almost exclusively in *batch* mode. Programs were recorded on punched cards and submitted in their entirety for translation to machine code. Programs were altered by removing, replacing and moving punched cards within the "source deck." Since nothing could be done to automate this manual process, software programming aids were necessarily limited to the features of the programming language itself, the diagnostic abilities of the compiler, and the diagnostic abilities of the operating system (register status reports on program abortion, "core dumps", etc).

When the hardware became capable and cheap enough to support interactive users and vast amounts of online storage, the possibilities for new applications, including software engineering environments, exploded. Furthermore, the continued decrease in computing costs means the possibilities are continuing to expand at a dizzying pace. We can

now put the computing power, speed and data storage capability of what was a multimillion dollar mainframe computer a few years ago on the desk of every individual in an organization, if we so desire. Likewise, the cost of graphics display devices have fallen into the realm of affordability. If one picture really is worth a thousand words, there are few places where pictorial representations would be more welcome than in software development. Whether we can effectively apply such vast amounts of computing power to applications in general will depend in large part on how much of that power we can apply to software engineering environments in particular.

Another hardware topic currently under vigorous debate involves instruction sets. We have seen how hardware availability influenced the growth and nature of high-level programming languages. This was not a one-way street, however. After a time, hardware designers began to consider how high-level languages would be implemented on the hardware they were designing. Soon they began to include instructions specifically for certain high-level constucts. For example, special index registers and instructions were included for iterative loops and array indexing. The hardware stack, immensely useful for languages that permitted recursion, was another feature frequently added (see p. 39 of [Ref. 10]). Even more elaborate and "high-level" instructions have been included on some machines.

This trend has been called into question by supporters of a concept popularly called "RISC" - Reduced Instruction Set Computers [Ref. 11]. RISC adherents claim that a small instruction set made up of very simple and very fast operations is better than a very large one more (apparently) attuned to high-level languages. They claim increased efficiency based on research which indicates that a reasonably "intelligent" compiler could do more optimization and therefore generate more efficient object code than it could if forced to use the "higher-level" and more complex instructions. They also claim more reliability based on the old notion that simple machines are inherently more reliable than complex ones. There may be a lesson here for designers of software engineering environments. Which is better, a simple language supported extensively by the environment or a large, complex language that presumably requires less support?

D. CONCLUSIONS

The vast majority of research and development effort expended in the computer field since its beginnings in the 1940s has been directed toward three major areas: programming languages, operating systems, and hardware. The critical importance of software was recognized in the early seventies along with a rather long list of problems encountered in the design, development, and maintenance of

reliable, effective software. The primary vehicle for addressing these problems was language design. Pascal and Ada are two related examples of languages designed during this period. However, the persistence of the "software crisis" indicates that language design alone, while important, will not solve the pressing problems of the software industry. Advances in hardware and operating systems have made a great deal of computing power available to software developers but they have not yet harnessed more than a tiny fraction for their own purposes.

III. ENGINEERING METHODOLOGY

A. INTRODUCTION

The "engineering" of a product can be thought of as four basic types of activity. These are design, implementation, maintenance/evolution and the management of all these activities. We will discuss each of these briefly in the sections that follow.

Before we begin to look at the general nature of engineering as a human activity, we should dispense with one particular misconception that seems to haunt the software industry. In [Ref. 12], Peters notes that, "Many software developers who lack an engineering background think of engineering as an exact discipline that produces formulated, precise, closed-form solutions to problems. The inexactitude associated with software design seems intolerable to many designers, who feel that if there were a true engineering discipline for software, all estimating and scheduling problems would go away." He continues, "Actually, nothing could be further from the truth: Engineering depends as much on common practice and empirical knowledge as it does on scientific fact." Certainly this observation is borne out by the number of times "rules of thumb," "safety factors," and the like are used in all engineering disciplines.

As we look at the general engineering method for clues regarding the way we should conduct software engineering endeavors, we must not lose sight of our objective. We wish to improve the ability of software engineers to produce high quality software efficiently. Any such improvement that promises significantly more in benefits than it consumes in resources is worth exploring.

B. DESIGN

1. Introduction

According to Jensen and Tonies [Ref. 3], the engineering design process can be broken into six phases. These are illustrated in Figure 1. Note its neat, straight line structure. While it provides a good way organize the discussion which follows, it is not very representative of the way an engineering project actually proceeds. Recognizing this, Jensen and Tonies provide a second diagram in [Ref. 3] which is reproduced here as Figure 2. With its numerous feedback loops, it is a much more accurate rendition of actual engineering processes.

One other feature of these diagrams requires comment. The "Implementation" phase is included because of the importance of its feedback to all the other phases. In the discussion which follows, implementation will not be considered part of the design process but, for reasons which will become clear, as a separate entity.

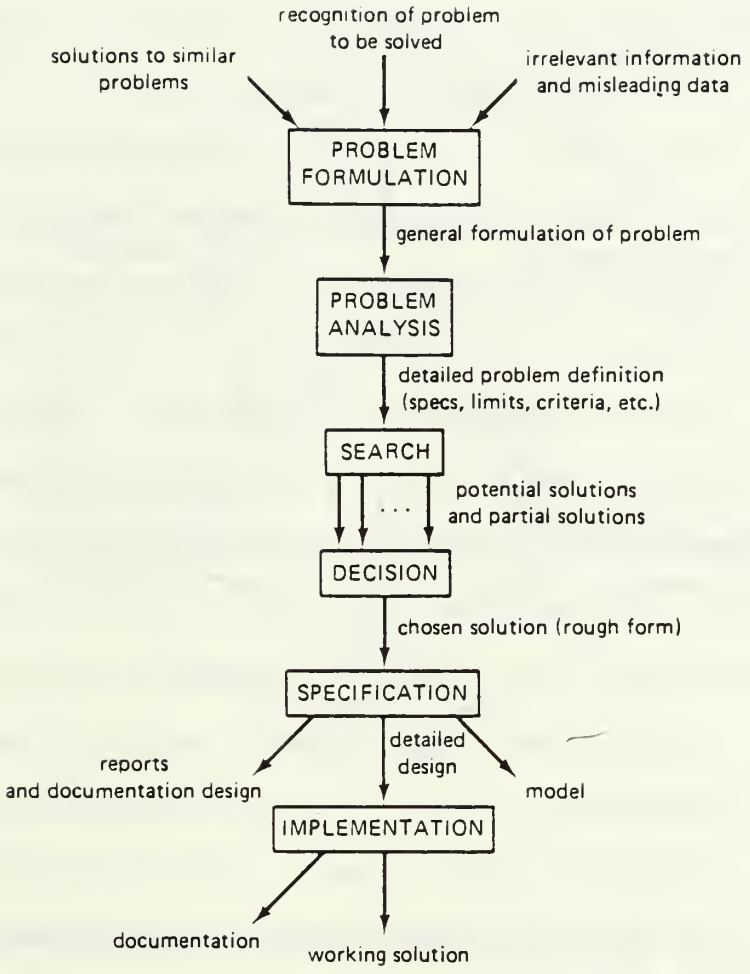

Fig. 1 Engineering Design Process

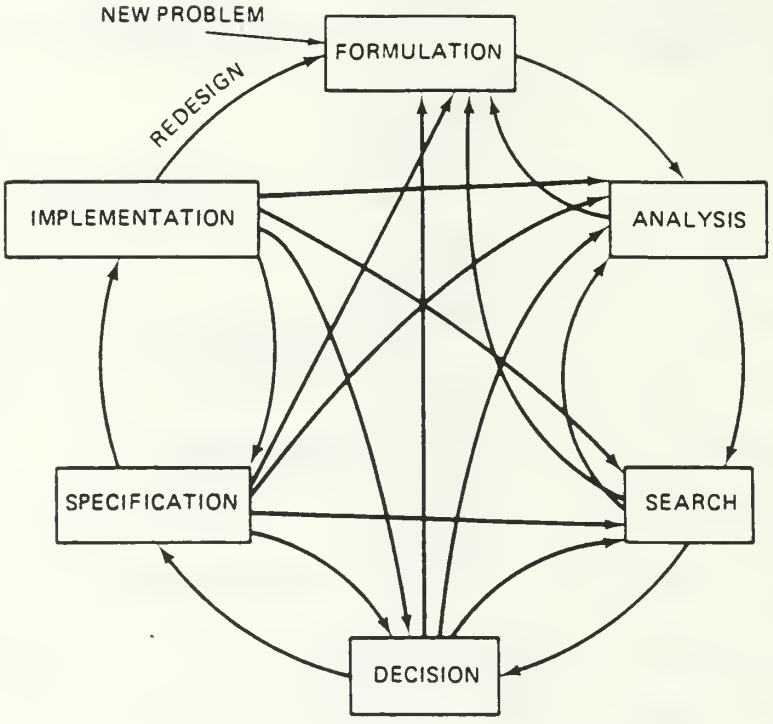

Fig. 2 Realistic Engineering Design Process

2. Problem Formulation

According to [Ref. 3], the inputs to the problem formulation process are the recognition of the problem to be solved, solutions to similar problems, and a fair amount of irrelevant and misleading information. The output is a general, but accurate, statement of the problem to be solved. The primary goal of the engineer in this phase is to gain a broad perspective on the problem and to remove prejudices caused by "misleading opinions, current solutions, and standard ways of viewing the problem." [Ref. 3] This usually involves a great deal of human-to-human communication as all parties try to bridge what William James characterized as, "The most immutable barrier in nature...", the one, "... between one man's thoughts and another's."

Another objective of the engineer at this time is to avoid thinking of possible solutions. A method frequently used is the so-called "black box" technique. Here, the problem is viewed as that of finding a process to transform inputs to outputs (e.g. steel into car bodies). In this phase, the objective is to identify the inputs and outputs but to keep the details of the transformation process hidden in the "black box" that forms the bridge between them.

3. Problem Analysis

In this phase, the engineer sets out to sharpen his understanding of the problem and to formulate a list of the constraints and requirements that will be placed upon any

solution. These constraints and requirements may be imposed by management (e.g. company standards), the customer (e.g. price and delivery date), or nature itself (e.g. strengths of materials). Another objective is to develop criteria for selecting the best of several possible solutions later in the project. Engineers, being pragmatists, normally will see several ways of solving any problem and so there must be some criteria for choosing among them.

One of the most useful ways of sharpening one's understanding of a large and complex problem is to decompose it into successively simpler problems. This technique is known as *stepwise* or *successive refinement*. Although there is no fixed algorithm for this process, engineers usually try to do *functional decomposition*. That is, they try break the overall function a device must perform into a set of smaller and simpler functions which have a composition equivalent to the original total function. Since more than one decomposition is possible, there may be several attractive alternatives.

During problem analysis it all but impossible to keep from thinking of potential solutions. It is natural for solutions or partial solutions to suggest themselves during the analysis. The disciplined engineer will note these and lay them aside for consideration during the search and decision phases. The undisciplined engineer may well allow his analysis to become biased by a potential solution.

4. Search

In this phase the objective is to locate and describe as many potential solutions as time and other resources permit. The broader the field of selection the more ideas the engineer will have to draw upon in selecting the approach he will finally use. It is important at this point that no attempt be made to eliminate potential solutions or partial solutions no matter how awkward they might seem on the surface. Selection is the objective of the next phase while collection is the objective of this phase.

Up to this point we have been using the term "solution" without having defined it in any special way. While a special definition is probably not necessary, it is important to realize its use in the current context implies an approach or route to be followed in obtaining a complete and detailed solution and not the complete solution itself.

5. Decision

This phase is where the various proposals are objectively compared to find a "best" approach to use for completing the solution design in sufficient detail to enable implementation. In order to carry out such an objective comparison, [Ref. 3] proposes the following four steps:

1. The selection criteria must be defined, and the relative weight of the individual elements of the criteria assigned;

2. The performance of the alternate solutions with respect to these criteria must be predicted as accurately as possible;
3. The performance of the alternate solutions must be compared on the basis of their predicted performance;
4. The selection of the preferred solution must be made.

The selection criteria may be based on a number diverse aspects of problem solution such as deadlines for product delivery, availability of the technology required to implement the solution, manufacturing costs, product performance, reliability, maintainability and ease of use, etc. The assignment of weights is necessarily a difficult and subjective task but the most difficult part of this process is the performance prediction of step two. Such predictions can only be based on rough estimates and the judgement of the engineer since the solutions themselves are still rather vague and ill-defined. Furthermore, it is almost certain that none of the proposed approaches will provide the "optimal" solution (even if one could be recognized as such) because the search phase did not exhaust the solution space but merely sampled from it. For this reason, it is a very poor engineer who can study the design of a very good engineer without finding some way to improve upon it.

6. Specification

This is the phase we most often associate with engineering design. It is here that the rough solution is refined and developed in considerable detail, and the bulk

of the design documentation produced. The design documentation is an absolutely necessary and integral part of any engineering methodology. Its most obvious purpose is to act as the output of the design process and provide all the information necessary for actual implementation of the design. However, design documentation sees considerable service prior to completion of the design process. It is used for design reviews which ensure the continued feasibility of the solution and provide others involved in the project an opportunity to suggest improvements while also keeping them up to date on project progress. Design documentation also provides an opportunity for error detection. Drawing checkers can detect and report such things as internal dimensional inconsistencies, incomplete bills of materials, etc. They can also check to make sure that mating parts, if within the specified tolerances, will in fact mate in all cases. Although this is a tedious, mundane task, it is very important to detect and correct such errors before the implementation phase begins. Correction of even minor design errors during implementation can be extremely expensive whereas their detection and correction prior to that time is relatively inexpensive.

Once the design has been specified, documented, and verified a very important event occurs. Up to now only the design staff under the supervision of the project engineer has been directly involved. Upon completion, the design is

"released" for implementation (production). All of the design documentation is then passed from the "engineering" (design) staff to the "production" staff. Although this does not end the design engineer's involvement in the project, it does reduce it rather sharply. With respect to this project, he becomes a consultant to the "production" department. He clarifies the design documentation if needed, makes design changes to accomodate unforseen manufacturing difficulties, etc., but is otherwise uninvolved with the actual implementation of his design.

This separation of design activities from implementation activities is one of the most important principles of modern engineering. In the "functional decomposition" of the engineering process that has occurred naturally over the years, design and implementation have become separate modules. The interface between them is the design documentation, which is made up largely of drawings - the well-known engineering blueprints.

C. IMPLEMENTATION

The last of the phases in the Jensen and Tonies [Ref. 3] description of the engineering design process is that of implementation. It should be clear from the preceeding discussion why it is considered as a separate entity from design in this thesis. It is in this phase that an actual artifact is produced according to the specifications of the

designer. For purposes of this discussion, we will postulate a "production staff" whose responsibility is to develop manufacturing and quality assurance methods, and to carry them out to produce and assemble parts into a complete working product. Often this staff is broken into three "departments" - methods, production, and quality assurance.

When a design is released for production, someone must decide how to manufacture and assemble the component parts in order to arrive at a finished product. Normally, the manufacture of a part requires many operations (e.g. machining operations) to be performed in a specific sequence on a piece of raw material before completion. This task is often given to a "methods department" which must develop these sequences of operations along with intermediate and final inspections for quality assurance. Once these have been specified, they are sent to the production and quality assurance departments along with a work request specifying how many parts of each type are to be made and blueprints of the design engineer's original drawings. In this way, the methods department quite literally "programs" the production floor (e.g. machinists and inspectors) with respect to the manufacture and inspection of each part. (Keeping the total production floor operating efficiently for maximum throughput (productivity) and minimum idle time is the job of the production department.) This "programming" function becomes even more obvious when numerically controlled (NC) machines

are used. Methods departments also "program" the assembly floor in a similar fashion by providing instructions for assembling the parts and testing the assemblies to ensure they work properly. They may also be charged with developing procedures for implementing field modifications to be carried out by the maintenance department.

Once the manufacturing and inspection procedures have been determined, production proceeds. We will not attempt to analyze this part of the process since it varies considerably depending on what type of product is being produced. For this discussion it is sufficient to assume simply that as components are produced they are inspected for "correctness" at several levels (e.g. parts, subassemblies, complete machine) using the inspection criteria contained in the design documentation. Rejected components are either discarded or reworked until they can pass inspection.

D. MAINTENANCE/EVOLUTION

All man-made devices require some sort of maintenance. Routine maintenance activities (e.g. periodic lubrication, preventive maintenance) are specified in the design documentation. Such things as "wear tolerances" are also specified so maintenance personnel can detect when parts need replacing. When a device begins to malfunction, repairs must be made. In such cases the maintenance person must do some "trouble-shooting" to locate the cause of the malfunction.

His primary aids in this endeavor are past experience and the original design documentation.

In addition to making repairs and performing preventive maintenance, maintenance personnel typically must report the results of their activities. Reviews of maintenance reports and customer complaints and comments often reveal design deficiencies or identify needed enhancements. Once identified, management must decide whether to pursue them. If changes are to be made, the design process described above is initiated and the engineering process continues from that point with one additional input - the original design documentation. That is, the design engineers go "back to the drawing board" with copies of the current design and make the necessary changes via the full design process from problem formulation to the release of the modified design for production.

E. MANAGEMENT

If left to his own devices, a good design engineer might never complete a design. This is because engineers are rarely satisfied with their own designs and can always see some way to make them better. However, economic realities restrict time and other resources to finite levels. Management must determine these levels and then husband the available resources to produce a profitable outcome.

In [Ref. 13], Reifer of *TRW Systems* states that management has "five constituent parts:"

(1) Planning
(2) Controlling
(3) Staffing
(4) Organizing
(5) Directing

He goes on to state eighteen "fundamental principles of management" which are reproduced in Appendix B. We will mention a few of these in the next chapter. For now, we will consider only the five management functions stated above.

According to [Ref. 13], "Planning is the primary function of management ...", and

> Plans are either strategic or tactical. Strategic plans identify major organizational objectives and govern the acquisition, use, and disposition of resources to achieve those objectives. Tactical plans deploy resources to achieve strategic objectives. Plans help managers decide in advance what will be done, how it will be done, and who will do it.

Plans provide a standard against which progress can be measured and communicate management's goals and expectations to the organization.

Organizational structures are created by managers so that people can work together effectively and efficiently to achieve the desired goal. Once created, the organizational structure must be staffed with personnel having the requisite skills and attitudes. When the staffing is completed,

the manager must direct and lead the staff to the desired goal.

Finally, there is the matter of control. No manager can be effective if he cannot control his project. Maintaining adequate control is just as essential as good planning. As [Ref. 13] puts it,

> Planning and control are inseparable activities. Unplanned actions cannot be controlled because control involves keeping events on course by correcting deviations to plans. Plans furnish the standards for control. Controls should be diagnostic, therapeutic, accurate, timely, understandable, and economical. They should call attention to significant deviations and should suggest alternative means of correcting the difficulty.

One of the traditional ways to exercise control in engineering is to require management approvals at several points in the product's life cycle. During the design process, design documents are carefully reviewed, the costs and benefits of various design alternatives are carefully estimated and compared, compromises are made and finally a course of action is approved. This happens many times and at many levels before the final design is released. Some decision are "internal" to the company while others require significant customer involvement. Similar things happen during the implementation and maintenance/evolution phases. The importance of the control function cannot be overemphasized.

F. DESIGN DOCUMENTATION

1. Introduction

We have seen that design documentation is used in all aspects of engineering. Because of its crucial importance to engineering methodologies, it deserves closer examination. We will now try to determine those features which contribute to its central role in the engineering process.

2. Levels of Abstraction

Engineering design documentation provides descriptions of the product at many levels of abstraction. There is a hierarchy of descriptions that vary in detail from the general outline of the finished product down to the most intimate details of the smallest component part. In mechanical engineering, for example, there are "detail drawings" of each unique part and various levels of "assembly drawings" for the assemblies and subassemblies that make up the final machine. As the assemblies become larger, their representations must necessarily show only major features while suppressing many details. However, sometimes a particular feature is shown in great detail by superimposing a "magnified view" on an otherwise high-level representation. This demonstrates the property of "fine granularity" present in engineering design documentation. Fine granularity may be defined as the ability to simultaneously display different levels of abstraction so that both the details of a feature and the context in which that feature occurs may be shown.

This ability to represent a complex device at various levels of abstraction is the key to engineer's ability to design complex devices that actually work. If the engineer could not decompose a complex design problem into a set of smaller and simpler design problems with corresponding smaller and simpler implementations, modern technology simply would not be possible.

3. Types of Abstraction

There is often more than one conceptual view of a product and its components. In electronics engineering, a logic circuit can be viewed as a set of connected transistors, capacitors, resistors, etc. It can also be viewed in terms of logic gates (AND, OR, NAND, XOR, etc.) which are represented by entirely different symbols from those used to depict the electrical components. Even in mechanical engineering we have "exploded views" of assemblies to illustrate how the assembled parts relate to one another even though the parts will never actually appear in the "exploded" configuration.

4. Completeness

Taken as a whole, the design documentation provides a ***complete specification*** from which the artifact may be implemented. It specifies both the "perfect" artifact and the types and degrees of imperfections that can be tolerated. Design documentation does not, however, specify how

the implementation is to be accomplished. Different implementors are free to employ different methods to accomplish the same end.

5. Universality

Design documents tend to be universally understood by engineers in the same field (mechanical, electrical, etc.) regardless of any differences among them including national origin. That is, a mechanical engineer in the United States can largely understand the design documentation generated by a mechanical engineer in Japan even though neither engineer has met the other or even understands the other's native tongue. This is because engineering design documentation depends heavily on graphical representations and the use of standard symbols and formats accepted worldwide, which together form a more-or-less "universal design language".

6. High Cost

The generation of design documentation is an extremely expensive process. Often years of design effort are expended before the first component of the end product is produced. The necessity of this large "design overhead" has long been recognized and accepted by the engineering profession. This is because the investment is repaid many times over in reduced communications costs throughout the product's life. Without design documentation, the designer would have to either perform all the engineering functions

(design, implementation, maintenance, etc.) himself or personally educate all those responsible for non-design activities.

7. A Valuable Asset

The three immediately preceeding characteristics make the design documentation of its products one of the most valuable and closely guarded assets of a company. Because of their completeness and universality, design documents could be used by competitors to produce the same or similar products. The expense of their development makes them attractive targets of industrial espionage since even their reproduction from an actual artifact is an extremely expensive proposition.

G. CONCLUSION

We have looked at the general engineering methodology used to design, implement, maintain and improve a product. In so doing, we found that a central and crucial role is played by the design documentation produced by the design engineers. In fact, the design documentation is essential to all "engineering" aspects of the product life cycle including the design process itself, the production of the product, the maintenance of the product, the enhancement or evolution of the product over time and the management of all these activities.

IV. SOFTWARE ENGINEERING ISSUES

A. INTRODUCTION

Before we can call ourselves "software engineers" we must subscribe to some software engineering methodology. We have already described the general engineering methodology in use today. However, this description lacks the detail necessary for actual practice.

In the last fifteen years, a number of software development techniques have come into being but a complete engineering methodology does not yet seem to exist. There are a number of companies whose sole business is software generation for various customers and many others which develop significant amounts of software "in-house" for their own purposes. We could analyze the software-related activities of these companies to determine their "software engineering methodologies." In so doing, we would probably find that most really have no well defined methodology. In other words, if we were tasked with developing a software engineering environment for any of these firms, we would face all the same problems and difficulties we face when designing software systems for other "unsophisticated" users. We would have to listen to the customer's stated needs and desires and then figure out how he really operates and what

he really needs and wants. We might even have to change his way of doing business to enable him to meet his goals. In short, we must know what we are trying to support before we can support it. This leads to the following principle:

Methodology Support Principle

A software engineering environment should be designed to support a specific engineering methodology.

The central position of the engineering methodology in the "ecology of software development environments" and its eight essential characteristics are given in [Ref. 14]. These eight characteristics basically include features already identified here (in rough form at least) but they are included as Appendix C for easy reference.

An interactive software engineering environment must address three basic classes of issues - (1) technical, (2) managerial and (3) ergonomic. These are not strictly orthogonal because real environments have two fundamental properties. First, "Everything is connected to everything else" and, second, "There is no such thing as a free lunch." Taken together, these characteristics imply that altering one feature of an environment will have *some* effect on the remainder of the environment. However, for discussion purposes, we will treat technical, managerial and ergonomic issues separately although we will see a good deal of overlapping among them.

B. TECHNICAL ISSUES

1. Software Design Support

When the word "engineering" is mentioned two things immediately come to mind - technology and blueprints. Engineers are people who design devices and structures which can be implemented using current or forseeable technology, and who document these designs with engineering drawings (blueprints) and related materials. The importance of design documentation to the engineering process and the essential properties of this documentation were described in the previous chapter.

Let us now consider the term "software engineering". Certainly we should associate "software engineering" with "technology" since it is part of the computer industry which is one of the most technologically advanced and rapidly evolving industries on the planet. But where are the blueprints? What does a "software blueprint" look like? Just how does one "design" a software system or even a part of one? Given a "design", how can it be communicated to an implementor? If there is a "key" to the software crisis, it most likely will be found in the answers to these questions.

While there is no software design documentation system that enjoys all the advantages possessed by those of the other, more mature, engineering disciplines, there are a a number of documentation techniques that have been proposed. One of the oldest of these is flowcharting, which

can be traced back to pre-FORTRAN days according to [Ref. 15]. Although it was widely used for a time, flowcharting fell into disfavor for a number of reasons. It was regarded as cumbersome because the time and effort required to construct flowcharts was significant. Brooks [Ref. 16] characterized flowcharting as a, "... space-hogging exercise in drafting." Furthermore, since the software implementations of the designs represented by flowcharts were more easily altered than the flowcharts themselves, programmers tended to modify the software without making corresponding changes in the flowcharts. As Yourdon [Ref. 17] observed, "... flowcharts are rarely maintained with any enthusiasm after the program has been finished." Hence, the implementation soon deviated from the design and so the design documents were discarded. (Yes, this *is* engineering heresy!)

There were other problems as well. Flowcharts represent only the flow of control through a program. This is like having blueprints with only one view. Ledgard and Chmura [Ref. 18] argue, "... program flowcharts can easily suppress much useful information in favor of highlighting sequential control flow." Hence, many important features cannot be seen clearly and some cannot be seen at all.

Another problem lay in the designs themselves. Before the advent of "structured programming", the control structures in most programs could only be described as "helter-skelter" - that is, no particular structure at all.

Confused "designs" resulted in equally confused flowcharts and program code.

Perhaps the greatest problem was one which remains to this day. There is no more significant indication of software engineering's immaturity than the lack of separation between design and implementation activities. In the preface to [Ref. 19], Chu states, "The application of engineering methods and practices to software development implies (1) the separation of software design from software implementation and (2) a software-blueprint interface between software design and software implementation." As long as design and implementation remain inseparable, engineering in the modern sense cannot take place. This was the most important reason flowcharts were largely abandoned. Because the designer was also the implementor (programmer in both cases), the making of flowcharts or other design documentation served no useful purpose in his view. As Yourdon [Ref. 17] also observed, "Indeed, most programmers will admit that they rarely bother writing the flowchart until the program has been finished (and then only because the manager insisted on it)..." In other words, software is often developed *without any design documentation* at all.

Other design documentation aids have been proposed in the years since flowcharting was introduced. These include Nassi-Shneiderman Charts [Ref. 20], HIPO (Hierarchy plus Input-Process-Output) [Ref. 21], PDL (Program Design

Language) [Ref. 22], Structure Charts [Ref. 23], and the "software blueprints" of [Ref. 19] as well as some others. This last technique at least strives toward providing all the essential characteristics of engineering design documentation although it still lacks the richness and variety of the more mature documentation systems. Chu's "software blueprint" provides three levels of abstraction, four processing data (as distinguished from control data) types, seven processing data structures, two control data types, two control data structures, a large number of high-level data operators, seven reference structures (e.g. procedure reference structure, data reference structure), and some other constructs (e.g. defined data types) as well. This technique does not make extensive use of graphical representations, although Chu allows that such representations could be usefully employed in conjunction with the "software blueprint."

All of the above aids seem to have been used with at least some success. Unfortunately, we do not have space here to explore them in detail. For our present discussion, the details are not that important. The important thing for designers of software engineering environments to realize is that some reasonably complete design documentation system must be chosen *before* a software engineering environment that supports software design may be developed. We state the following principle:

Design Documentation Principle

A software engineering environment should support a design documentation system that is (1) complete, (2) capable of representing appropriate levels and types of abstractions with fine granularity, (3) universal, and (4) economical.

Completeness and universality serve the same function in software engineering as in other engineering disciplines. They allow design to be largely divorced from other engineering activities while reducing the total communications burden over the product's life. Although universality may seem an impossible goal at this early stage of software engineering's development, we can begin by following the example set by the other engineering disciplines. That is, we need to develop graphical symbols and standard formats for representing the various important features of a software design. Flowcharts and hierarchy charts already can be considered universal and adaptations to increase their utility would be very valuable for this reason. Easily understood graphical representations of certain common data structures such as stacks, queues, and linked lists could be developed without much difficulty. In short, the key to universality lies in the consistent use of graphical representations, formats (both graphical and textual), and symbols that are easily recognized even though only limited standards for these currently exist.

Abstractions also serve the same function as in other disciplines but we will list a few examples here to show how they apply to software engineering. Programs can be viewed in many ways and so several types of abstraction are useful. The most obvious and familiar view is the textual (program listing) one which is often "pretty printed" to emphasize certain structures. (As design documents, listings can at best be regarded as analogous to the "detail" drawings of the mechanical engineer. It is probably best think of listings as representing an implementation rather than a design.) Flowcharts depict control flow, hierarchy charts show the procedure reference structure, data flow diagrams illustrate how and when data items are modified, etc. The reader can no doubt think of several other types of useful abstractions.

By having a design documentation system for representing the various aspects of a software design, we can analyze the design to see if it satisfies various design criteria. For example, if we can display all of the connections between modules, we can estimate levels of coupling. We can, like the drawing checkers mentioned in the previous chapter, inspect the design for technical flaws, inconsistencies, and adherence to design and documentation standards mandated by management. This suggests the following principle:

Enforcement/Aid Principle

A software engineering environment should enforce technical correctness and conformance with management policies while aiding the user in maintaining these standards.

It should be made clear that "technical correctness" in this context does not necessarily mean "proof of correctness" in the formal sense. Although engineers of all disciplines use mathematical calculations in creating and checking their designs, rarely if ever is the "correctness" of a design confirmed by a formal mathematical proof. We are more concerned here with such mundane things as ensuring that the interface specifications between modules are consistent or that the design does not call for the use of side effects. Of course, where proofs of correctness can be accomplished economically, they should be done.

There is an important corollary to the Enforcement/Aid Principle. Designs change frequently while they are being developed as well as less frequently after release. Steps must be taken to ensure that changes to one part of a design are accompanied by appropriate changes to the remainder of the design in order to maintain consistency. It is altogether too easy to make a seemingly innocuous design change that actually proves to have widespread ramifications. This leads to the

"Permanent" alteration of one view of a design should not be "accepted" by the environment until all related views are made to be consistent with the change.

Related views include all levels and types of abstraction that show either the altered module or modules that interface with it. "Permanent" and "accepted" are quoted to call attention to the need for allowing temporary inconsistencies so local evaluation of alternative design changes and environmental transition states may be accomodated.

2. Implementation Support

It is difficult to say where design ends and implementation begins in software development. Here, we will define *implementation* as that activity which transforms a software design into an executable program or system of programs. We will take *executable* to mean either directly executable on the target hardware (machine code) or translatable by automatic means to a form that is directly executable. Thus, *programming* languages are considered distinct from *design* languages. This distinction is far from being absolute, however. For example, Pascal could be used as part of a design language system for specifying assembly language programs to be created by hand if a compiler for the target machine either did not exist or generated intolerably inefficient machine code. Another thing making the

design/implementation boundary fuzzy is that no matter how complete the design, programmers are always allowed some leeway and so perform some detailed design functions. Therefore, placement of the design/implementation boundary will be a difficult chore for the developers of software engineering methodologies for the forseeable future.

a. Enforcement/Aid and Consistency

Implementors face many of the same sorts of problems as designers but in a different context. For example, the Enforcement/Aid and Consistency principles can be applied to such things as ensuring consistency between the implementation and the design, enforcing various programming standards and policies, enforcing syntactic and semantic rules of the programming language being used for the implementation while aiding the programmer in conforming to those rules, etc.

When applied to programming itself, the Enforcement/Aid principle can be extremely powerful. It is, in fact, the driving principle behind most modern interactive "programming environments" including the Cornell Program Synthesizer [Ref. 24], the DWIM (Do What I Mean) feature of Interlisp [Ref. 25], and any syntax-directed editor. These tools show that there is nothing sacred about having the compiler perform enforcement functions. In fact, given today's computing power and the prevalence of interactive use, one could effectively argue that compilers should *not*

be in the enforcement business at all but should handle only the translation function from a form that can be made human-readable to a form suitable for machine execution. Note the possibility here of storing a program in an intermediate "bidirectional" form that could be "unparsed" into a textual or other understandable form (one direction), or directly translated into executable machine code (other direction) by either an interpreter or a compiler.

The Enforcement/Aid principle can be used to define a completely new programming language or effectively alter an existing one. For example, in [Ref. 26], Kernighan and Plauger describe RATFOR, a preprocessor for FORTRAN that adds Pascal-like control structure constructors and character manipulation to FORTRAN by translating these constructs from RATFOR to FORTRAN which can then be translated into machine code by the FORTRAN compiler. A syntax-directed editor for RATFOR that restricted the use of the GO TO statement (which RATFOR does not) could aid in the enforcement of structured programming techniques on FORTRAN programmers, particularly if it was interactive (which RATFOR is not) and provided templates (like the Cornell Program Synthesizer) and other aids for program construction. It is easy to see how strong typing could, in essence, be added to FORTRAN as well.

The Enforcement/Aid principle can also be used to effectively create a subset of a language. In fact, the

Cornell Program Synthesizer does precisely this for PL/I. Of particular interest in this regard will be the relationship between Ada and various implementations of the APSE. While DOD has decreed that it will allow no subsets of Ada, it is difficult to see how this can be effectively prevented. Ada's large size and complexity practically beg for some sort of subsetting. As Hoare observes in [Ref. 9],

> It is not too late! I believe that by careful pruning of the ADA language, it is still possible to select a very powerful subset that would be reliable and efficient in implementation and safe and economic in use. The sponsors of the language have declared unequivocally, however, that there shall be no subsets. This is the strangest paradox of the whole strange project. If you want a language with no subsets, you must make it *small*.

The APSE provides the mechanism for putting subsets into effect from a programming standpoint even if no "incomplete" versions of the Ada compiler are allowed.

b. Structure Manipulation

Those who perform any amount of word processing are familiar with the concepts of deleting and inserting words, sentences, and paragraphs. Word processors are organized around these structures because they are such an intimate part of written prose in most western languages. It seems clear, then, that software engineering environments should provide programmers with the capability to manipulate structures common to software development. An example of this is the Cornell Program Synthesizer which provides for

direct manipulation of structural elements in the PL/I subset it defines. "While" loops, "If-Then_Else" blocks, expressions, etc., can be inserted or deleted as complete entities. Furthermore, the cursor movements are keyed to such structural elements rather than the textual elements of lines and characters. This illustrates the

Structure Manipulation Principle

> The user of a software engineering environment should be able to create, reference, locate, alter and delete structures defined within the environment. He should also be able to display meaningful representations of them within the context of any applicable type or level of abstraction.

Note that although our examples apply to actual "source code" generation, this principle applies to other activities as well. For example, the programmer might wish to interrupt his programming and view some part of the design documentation or consult the subroutine "librarian" to see if a needed function has already been implemented.

c. Analysis Support

During the construction of a program or module, the programmer will need to perform various analyses to check his work, locate bugs, and ensure conformance with standards and policies of his immediate supervisor. There are two types of analysis - static and dynamic.

In static analysis, the programmer checks the static structure of the program for various attributes. For example, he might check a FORTRAN subroutine to ensure that certain formal parameters did not appear on the left of an assignment operator if the design called for them to be used for input only (like in parameters in Ada) or check an entire program to ensure that no coercions are present. In Pascal, he might want to ensure that a certain global variable was referenced only in certain procedures. These examples lead us to the

Static Analysis Principle

A software engineering environment should (1) allow the user to make assertions about the static structure of a module, program, or system of programs and then report back to the user which assertions are not valid and why, and (2) allow the user to request certain information about the static structure and then report this information back to the user in a lucid format.

For example, if a user asserted that the actual parameters used in calling a particular procedure never contained named or literal constants, the environment should be able to check for conformance with this assertion and, if exceptions were found, provide the user with a list of where the offending procedure references were located. It should also

be able to display these in any applicable context, highlighting in some way the offending parameters themselves.

The Static Analysis principle is very similar to the Enforcement/Aid principle in many respects. The main difference is that the Enforcement/Aid principle is more concerned with general rules which all software generated by an organization must obey. The Static Analysis principle addresses features and characteristics peculiar to the specific program under development which is why the programmer himself needs the ability to make assertions and have them checked. Quality assurance persons can also use static analysis to provide defense in depth (see principle 3 in Appendix A).

The purpose of dynamic analysis is to study program behavior. In order to be complete, a set of design documentation must contain acceptance criteria. Some of these may be static, others dynamic. When we speak of "testing" and "correctness" regarding software, we are usually thinking of the program's behavior. We may need to know if a certain set of inputs produces the expected outputs, if procedures are called in the expected sequence, the frequency distribution of procedure calls for a typical input stream, actual execution times, or similar facts concerning the program's behavior. To support this need, we state the following principle:

Dynamic Analysis Principle

A software engineering environment should (1) allow the user to make assertions about the dynamic structure of a module, program, or system of programs and then report back to the user which assertions are not valid and why, and (2) allow the user to request certain information about the dynamic structure and then report this information back to the user in a lucid format.

These last three principles strongly suggest that a software engineering environment should provide many of the same types of support we normally associate with database and/or artificial intelligence applications. Certainly database and artificial intelligence techniques should be considered in the design of any software engineering environment.

C. CONCLUSIONS

We have seen that before we can design an environment to support a software engineering methodology, we must first define that methodology. We have also seen that design documentation is just as essential to software engineering as it is to the other engineering disciplines although no system of design documentation presently exists that has all the fundamental characteristics of those associated with the more mature disciplines. The idea that software developers seem to have been so busy building systems to increase the

productivity of others that they haven't yet performed that service for themselves has been observed. Finally, we saw indications that during design and implementation a significant "knowledge base" is built up and that the techniques of database management and artificial intelligence might be brought to bear in a productive manner.

V. MANAGERIAL ISSUES

If the design and implementation of reliable, effective software are not well understood, then the management of projects which have such responsibilities must be a total mystery. In [Ref. 3], Tonies observes,

> If management science is immature, then we can expect software management science to be especially immature, since the software industry is itself so new and is expanding so quickly. The probability, then, of finding effective software management would seem to be small. It is.

We cannot hope to treat this immensely important topic adequately here. However, we will point out some issues which could be addressed by a software engineering environment. In particular we will look briefly at the *planning* and *control* functions of management.

In [Ref. 27], Thayer, Pyster and Wood report the results of a survey of "... experienced and knowledgeable data processing managers as well as some of the leading computer scientists from industry, government, and universities within the US and abroad." In this survey, "...participants were asked to express their opinions concerning 20 hypothesized major issues or problems of SEPM." SEPM is an abbreviation for Software Engineering Project Management. The 20 hypothesized problems are reproduced in Appendix D.

Eight of the ten hypothetical planning issues, three of the six control issues, organizational accountability, and the staffing of the project manager position were all considered definite problems. None of the 20 issues could be eliminated as a unimportant.

A. PLANNING

Planning requires estimation and estimation presupposes the existence of some system of measurement. So far, researchers have had a very difficult time in their attempts to develop meaningful measurements of software. In [Ref. 28], a number of papers on software metrics have been collected. In their preface, Perlis, Sayward and Shaw note that, "No matter what aspect of software one studies, there is a noticeable lack of collected and categoried field data on which to build." They continue, "... past software projects have rarely integrated data collection into their production schedule ..." When this is combined with DeMarco's assertion in [Ref. 29] that software managers are such poor estimators because they don't collect data on project performance and compare it with the actual results in even the most trivial manner so they can learn from their mistakes, we see that the need for data collection is a universal one. Ironically, there are few situations where large-scale data collection would be easier than in a software development project, where most of the work is done on

one or a few computers. Even if we don't have a well established set of metrics, almost any data collected would be useful to the collecting organization in making future estimates as well as being more grist for the research mill. We don't have space here to discuss exactly what kinds of project data might be collected but we can state the following principle:

Data Collection Principle

A software engineering environment should provide for the unobtrusive collection of software management data.

In addition to the collection of general management data, one specific class of data should be collected for each software product developed. Enough data should be collected to establish a complete set of audit trails. It should be possible to follow, after the fact, a software development project from beginning to end through its residual documentation. For example, an "auditor" should be able to find the alternative designs considered and the rationale used for choosing the one that was actually used. He should be able to find out who made a major design decision, who implemented a particular module, who tested and accepted it, the test specifications and actual test results, the costs of these activities, and other such useful information. Such things are not only useful for general planning purposes, they are also valuable to those who may

be tasked with designing enhancements or alterations to a particular system and they are essential to any system of configuration management. We state the following principle:

Audit Trail Principle

A software engineering environment should maintain audit trails showing the relationships between specifications, designs, implementations, maintenance and enhancements, and the resources expended in these activities. Further, such audit trails should be navigable in both directions from any point.

Another service which the environment should provide is that of librarian. Software engineering efforts are notorious for "reinventing the wheel," that is, failing to effectively use the knowledge gained from or output of previous efforts. If the results of earlier work were organized and cataloged, managers could find at least some data to guide their estimates. Technical personnel could likewise locate and study designs or implementations which might be applicable to the problem at hand. It is often more efficient to use or combine existing products or designs than to create a totally new product. We therefore state the following:

Information Organization Principle

The information gleaned from the various documentation and data collection efforts should be organized for easy reference and maintenance.

B. CONTROL

Chapter 1 of [Ref. 29] begins with the statement, "*You can't control what you can't measure.*" We submit that it is even more difficult to control what you can't even *see*. Many software managers must feel like the fabled emperor who wanted some new clothes. When they ask for software project progress reports, the one thing they rarely see is the software or its design. The control issue is the main reason why design documentation is so important to management.

We saw earlier how the Enforcement/Aid principle applied to designers and implementors. Recall that this principle involved ensuring "conformance with management policies." In order to carry out this function, the following principle should be applied:

Management Control Principle

A software engineering environment must be controllable by the management of the organization it serves.

C. ORGANIZATION

Organizational structures are developed so that a variety of persons with expertise in a variety of needed disciplines or specialties can work together effectively and efficiently to achieve a common goal. The computer support for an organizational structure should reflect that structure. This leads to the

Organizational Structure Principle

> A software engineering environment should be partitionable along the structural lines of the organization so that individuals and groups may have the necessary levels of privacy and isolation from other individuals and groups, and so the communications among them may be reasonably controlled by forcing them through established interfaces. However, it should also be possible to allow information to flow across organizational boundaries, when needed, through specially designated and controlled interfaces.

This is really an application of the Information Hiding and Manifest Interface principles (numbers 4 and 7 respectively in Appendix A) and the Management Control principle (above) to human organizational structures.

D. CONCLUSIONS

In this brief chapter we have tried to emphasize the importance of management to software engineering, but

without delving into the various details and styles of management. The principles we have outlined are very broad and deal only with automated support of management functions. No doubt any practicing software manager could think of a host of additional features and tools he would like to see in a software engineering environment. The one concept that should be clear at this point is that designers of software engineering environments should not be satisfied with supporting only the technical personnel. The solving of management problems is just as important to the goal of increased software productivity as the solution of technical problems. However, software management problems are much more difficult to solve and are often not regarded as being within the realm of "computer science."

VI. ERGONOMIC ISSUES

A. INTRODUCTION

So far we have looked at some of the engineering and managerial issues involved in software engineering environment design. We will now look at some ergonomic issues. That is, we want to examine the interface between the human user and the automated environment in which he will (we hope) immerse himself.

Many of the principles stated up to this point have ergonomic overtones and some, like the Structure Manipulation principle, could even stand as ergonomic principles alone. We wish to continue with a list of principles which apply to the design of *any* interactive application. Before going on, however, we wish to take time to emphasize the need for good ergonomic features.

The first and foremost reason for good ergonomics is that tools which are difficult or awkward to use will be ignored. A similar fate awaits tools which are unpredictable or unresponsive. To employ a greatly overworked phrase, the tools must be "user friendly." The second reason is that the investment required to bring an "unfriendly" tool on-line is probably not much less than that required for a similar "friendly" tool. We shouldn't

waste money on tools that will be discarded. A third reason is productivity. Even if designers and programmers must work with the given tools because no others are available, they will be much more productive with "friendly" tools.

We have been speaking here of "tools" as if we were going to supply them in a "toolkit." We actually wish to avoid such a concept. Obviously a software engineering environment must put tools (capabilities) into the hands of its users. However, it will not do to have a toolkit of miscellaneous devices, however useful, that do not work together in harmony. This makes the environment as a whole "unfriendly." Nevertheless, this is how most of the present environments have come about. (Smalltalk is a notable exception.) To quote Spier et al. [Ref. 30], "Generally, tools sprang into spontaneous existence as desperate programmers resorted to improvisation; such improvisations are colloquially termed 'midnight projects'." This is certainly how the tools of UNIX originated although it is claimed by Kernighan and Mashey [Ref. 31] that the system was built up by evolutionary means (survival of the fittest tools) which achieved a reasonable degree of integration. Nevertheless, they also note in [Ref. 31] that things could be better in this regard.

In [Ref. 32], Hansen lists a number of principles for the design of interactive systems. We will examine these briefly in the sections that follow and add a few as well.

B. USER ENGINEERING PRINCIPLES

The user will need to communicate with the environment via some "language of interaction." It is clear that such a language should be designed in accordance with the principles of language design formulated in [Ref. 5] and reproduced here as Appendix A. This leads to our first principle:

Interface Language Design Principle

The language of interaction between a user and an interactive application should be designed according to the principles of good programming language design.

The first principle listed in [Ref. 32] is "Know the User". This principle is seen as having two parts. First, Hansen suggests building, "... a profile of the intended user: his education, experience, interest, how much time he has, his manual dexterity, the special requirements of his problem, his reaction to the behavior of the system, his patience." Second, and more important, is the need for the designer to appreciate two traits common among humans: "... they forget and they make mistakes."

While the second assertion is well beyond dispute, the need for a detailed user profile is highly questionable. In the first place, humans are highly variable beings and in the second place, most of the characteristics listed above

will change over time. Therefore, rather than having the designer tailor the system to a particular type of user, he should, at most, discriminate only among broad classes of users, e.g. professionals vs. amateurs. In any case, he should make the system flexible enough to allow users to "customize" the interface to their own liking. The old engineering adage, "If you can't make it right, make it adjustable" applies.

In the sections that follow, we will follow Hansen's outline and comment on his principles.

1. "Minimize Memorization"

The first principle listed in this category is that of "Selection Not Entry." Here, Hansen recommends selecting items from menus via keyboard codes. While this certainly has advantages for novice users, menus can quickly become frustrating for experts. The word processor on which this document was produced provided multiple levels of interaction. First, it allowed selection of a "help level" to determine how much command information was displayed continuously. Second, for multi-stroke commands, rapid typing of the keystrokes resulted in immediate execution of the commands while a delay following the first keystroke caused a menu of second-stroke commands to be displayed. In other words, it minimized the level of *required memorization* while allowing the user to take advantage of the increasing level of expertise he gained naturally through experience. In

[Ref. 30], Spier et al. recommend, "at least two modes of interface: 'novice' mode and 'expert' mode: preferably more." This leads to the

Multi-level Help Principle

For any interactive tool, there should be a hierarchy of "help" levels ranging from no prompts at all to on-line instruction. Furthermore, within an environment, the user should be able to set the "help level" on a tool-by-tool basis (fine granularity of help levels).

As an example, suppose a user wanted to use a PL/I "while" loop in a program. He could first ask for a template. The system might then want to know which form of the "while" loop was desired and display a list of short but descriptive names. If this wasn't enough, the user could ask to have the alternative templates themselves displayed and if he still couldn't make up his mind, he could ask for detailed instruction on the characteristics and typical uses of the different types of "while" loops. Such a system would provide for all levels of expertise while penalizing no one. (Note the incorporation of the Localized Cost principle of [Ref. 5] here.)

The second principle in this category is that of using "Names Not Numbers". This is almost identical to the Labeling principle of [Ref. 5]. Hansen does, however, provide the additional recommendation that a dictionary of

names (labels) be maintained by the environment on-line so they can be easily referenced to refresh the user's memory.

The third principle is that of "Predictable Behavior". Although Hansen provides no precise definition, his concerns appear to fall mainly within the realm of the Regularity, Simplicity and Syntactic Consistency principles of [Ref. 5]. Certainly, unpredictable behavior cannot be tolerated since a user would quickly become frustrated.

The last principle listed by Hansen in this category is that of "Access To System Information." Here, he seems to be discussing the need for a user to have at least partial control over his interface with the environment. This need is captured in the following principle:

Configurability Principle

> The user should be able to configure his interface to the environment (tool) within the constraints mandated by higher management. This capability should display a fine granularity.

For example, in order to support the organizational goals of a software development organization, the various levels of management must be able to reserve the alteration of environmental features according to organizational policies. Those features not reserved to management should be accessible to the end user so that he may tailor the remainder of the environment to his own liking. One example

of such "tailoring" was given earlier in connection with setting "help" levels.

2. "Optimize Operations"

The first of Hansen's principles in this category is, "Rapid Execution Of Common Operations." Efficient execution of frequently used commands is needed to reduce user frustration and make effective use of system resources. Less frequently used functions or ones that involve so much work they cannot be made to appear instantaneous will take longer but should, nevertheless, give the user some positive feedback while they are in progress. We capture these ideas in the following three related principles:

Meaningful Response Principle

The appearance of the display and the text of messages in response to user actions must be appropriate to those actions.

Rapid Response Principle

Simple and frequently used functions should have an immediate (in terms of human reflexes) effect upon the display.

Status Reporting Principle

Lengthy functions should periodically update the display to assure the user that progress is being made in carrying out the requested function.

The second principle given by Hansen in this category involves "Display Inertia". It may be stated as follows:

Display Inertia Principle

The display should change by the least amount possible in response to a user action. The display should not, however, violate the Meaningful Response principle.

The third of his principles involves what Hansen calls "Muscle Memory". He notes that repetitive operations like typing are relegated to the lower part of the brain and therefore different tools should use similar keystrokes to perform similar functions. For example, the "escape" key should not be used as an "emergency exit" to return one tool to some "base state" while another tool in the same environment uses the "escape" character as a special kind of delimiter. This author, like anyone else who has used a variety of similar interactive tools (e.g. text editors) on a variety of systems, has found the lack of standardization of commands and key assignments to be extremely frustrating when the same small number of basic functions are being performed.

Another important aspect mentioned by Hansen is "burst mode input." He notes that interactive users tend to type in short bursts sometimes exceeding 100 words per minute. While it is not essential that the system be able to respond to commands at this rate, it is essential that it

be able to reliably accept both commands and data at whatever rate they are being typed.

The last principle Hansen mentions in this section involves being prepared to "Reorganize Command Parameters". His main concern here is in being able to adjust the user interface to reflect the lessons learned through actual experience. The most powerful way to do this is to put the necessary adjustments in the hands of the user himself. We have already mentioned the Configurability principle in this regard. Since we can't possibly expect to anticipate all of a user's needs, we should also seek to make the environment *extendable* within certain constraints. That is, we should give the user the opportunity to create and use some of his own "private tools" [Ref. 30] and commands. For example, some operating systems have "command files" where commonly used command sequences can be placed by a user and invoked as a single command. We state the following principle:

Extensibility Principle

An environment should be extensible in the sense that it must be possible to add tools at any time and at any level which enjoy the same level of control and integration as the original tools.

The combined effects of configurability and extensibility effectively remove the need for the detailed user profiles mentioned earlier.

3. "Engineer for Errors"

As noted earlier, humans are error prone. Hansen's first principle in this section is the provision of "Good Error Messages." We consider this as being included in the Meaningful Response principle since the correct response to an erroneous input is a good error message.

Hansen's next principle is worthy of its own place in the sun, however. He states it as, "The System Must Provide Reversible Actions." We state it as follows:

Reversibility Principle

Any action taken by a user must be reversible for some period of time or number of subsequent actions.

Reversibility is one of the most important ergonomic principles. Because humans are so error-prone, they tend to take actions which they later need to reverse or "undo." No one would expect a user to be happy if, after spending 30 minutes composing a paragraph, he then struck the "delete paragraph" key when he meant to strike the "delete word" key and found he could not recover from his error except by retyping the entire paragraph.

The Cornell Program Synthesizer [Ref. 24] carries reversibility one step further - "reverse execution" of a program in support of debugging activities. As described in [Ref. 24], "... the forward execution interpreter maintains a history file of the flow control and the values destroyed

by assignments to variables. The reverse execution interpreter restores values and updates the screen to give the illusion of the program executing backwards."

4. "Perception Aids"

This category comes from [Ref. 30]. There, Spier et al. describe the "windows" concept which has come into recent popularity through the work on Smalltalk [Ref. 33] and some recent personal computer products (e.g. Apple Computer's Lisa and MacIntosh systems, and Microsoft's Windows) Windows basically provide a mechanism for easy context switching by the user without the irretrievable loss of previous contexts. As pointed out in [Ref. 30], "It should be trivial to interrupt an activity, embark upon another (which is similarly interruptable), and later resume the first activity." The ability to display multiple windows simultaneously (though some may be partially hidden) gives the user interface a fine granularity.

Spier et al. also advocate the use of high resolution color graphics to enhance user perception. Graphics have application throughout the software engineering process and are often a much more effective communications medium than even the most imaginatively formated text. The prominence of graphical representation in the design documentation systems of the other, older engineering disciplines is no accident.

C. CONCLUSIONS

In the preceeding paragraphs we have tried to stress the importance of having a well engineered user interface to any interactive application. We then listed a number of important principles for the design of user interfaces. The list was by no means exhaustive but it did point out a number of often overlooked but important issues. The most important conclusion to draw is that a tool which is difficult or awkward to use will not be used if an alternative exists, and will be used inefficiently at best even if there is no alternative.

VII. CONCLUSIONS AND RECOMMENDATIONS

A. SOFTWARE DEVELOPMENT AS "ENGINEERING"

When software is compared with other complex artifacts which are "engineered" rather than "crafted" we find many more similarities than differences. In spite of this, software development is still more of a craft than an engineering discipline. Although this situation may be understandable given software engineering's youth, it is nonetheless important to speed its maturation as much as possible.

The similarities software artifacts share with other products of an engineering development process include a high degree of complexity, a requirement for a reasonably long useful life of continuous, reliable operation, the need for alteration, enhancement and, in a sense, "repair" during its life, a requirement that it be serviceable by people not intimately involved with its original development, and a requirement that it be operable by persons who are ignorant of its internal structure and operation. While software has no moving parts to wear out, neither, in most cases, to solid state electronic circuits. However, like an electronic circuit, software may be released with flaws that are not immediately apparent and, when these manifest themselves, "repair" or replacement is in order.

Software is different from other artifacts in that its implementation is not a physical object. This is really the only significant difference and its main impact is to require that a little extra imagination be used to represent software designs.

As implied by the first paragraph, software *engineering* is quite different from the methods of the more mature engineering disciplines. In fact, it is questionable at this time whether a software engineering discipline can be said to exist. The most obvious difference seems to be the lack of a complete and universal software design documentation technology. This failing seems to be at once a symptom and a cause of a significant difference in the way engineering project resources are employed. In all the older engineering disciplines, design and implementation are separate activities usually carried out by separate groups of people who communicate mainly via the design documentation (e.g. blueprints). In software "engineering" design and implementation are still inseparable activities, and it is for this reason that software development remains a "craft."

Design documentation is also needed to communicate from the designer to maintenance personnel, operators and later designers a wealth of needed information. Without design documentation, the burden of communicating this information to all those with a "need to know" would be unmanageable.

It is a strange paradox that Brooks [Ref. 16] decries flowcharting as a "space hogging exercise in drafting" while at the same time observing that adding people to a late software project only serves to make it later because the communications burden of bringing the new people "up to speed" exceeds the amount of useful work they can do. This is not to say that flowcharting is an adequate means of documenting a software design. Rather, it serves to point out the reluctance of software developers to expend the considerable effort required to develop *any* reasonable amount of design documentation. This reluctance stands in stark contrast to the other engineering disciplines. A recent radio commercial announcement for a new model of automobile points this out admirably. In the announcement, it was claimed that the manufacturer had spent over five years and over a billion dollars in design effort on that particular model *before* it went into production. Thus, even in such a well established sub-field of engineering as automotive design, vast amounts of design effort are expended routinely, and much of this is spent developing the necessary design documentation. Software development has no analogy in this regard and until it does it cannot truly be called "engineering."

B. SOFTWARE ENGINEERING ENVIRONMENTS

We can increase the rate of maturation from software development as a craft to software development as an engineering discipline by designing (with whatever design tools we have at hand), developing and implementing software engineering environments that support an analog of the general engineering process as adapted to software development. The knowledge gained from this work will suggest many more improvements in the way we produce software, while at the same time improving software productivity in general by making many useful tools available. So far, the reluctance of the software industry to engage in this activity seems paradoxical.

While basic research on programming languages in general should continue, there is probably no need for further efforts with respect to imperative languages. This is because we can effectively control, through the environment, most of the language's characteristics which are apparent to the programmer. If increases in productivity and quality are truly our goals, then, at this point, the most naive efforts at software engineering environment design probably hold more promise, in the short term, than the most sophisticated research on programming language design.

C. FUTURE WORK

Clearly the most critical area for future efforts is in the area of software design documentation. Design and its associated documentation techniques are crucial to the operation of all engineering processes. Unfortunately, the needed research probably doesn't hold much glory for the academician. Certainly the design documentation systems of engineering in general were not the result of such research (although some may have been by-products of academic inquiries), but evolved more or less naturally over a long period of time. This will eventually occur with software as well. The problem is that many do not believe we can afford the luxury of such "natural evolution" in the face of what they term the "software crisis."

In particular, we need to find more ways of employing graphics in software design documentation. Pictures can often communicate more information in less space. They also tend to be more universally understood than textual or spoken languages. Furthermore, graphics display devices and a reasonable amount of general purpose software to support them has fallen into the realm of affordability. This means that much of the "drafting" burden can be automated. In fact, there is already a trend in the more established engineering disciplines toward "computer aided drafting."

Another area requiring further research is software management. Software management is in an even more immature

state than software engineering in general. So far we don't know how to estimate the cost, time, or complexity of developing a piece of software. Furthermore, we don't know how to measure the result of a project for a meaningful comparison with the original (probably meaningless) estimates and, worse than that, we usually don't even try. At least a well designed software engineering environment would allow for the collection of project data which, when analyzed, might yield some insight into the management problem.

D. CONCLUSIONS

Software engineering is still very immature as an engineering discipline. Before significant further maturation can take place, work must begin on establishing a design documentation system that shares the essential characteristics of other engineering design documentation systems. Software engineering environments with a well integrated set of useful tools for aiding design, implementation, quality assurance, maintenance and enhancement, and management of software hold much more promise for increased software productivity and quality than the traditional approach which has restricted itself to programming language design.

APPENDIX A

Principles of Language Design
(After MacLennan [Ref. 5])

1. *Abstraction*: Avoid requiring something to be stated more than once; factor out the recurring pattern.

2. *Automation*: Automate mechanical, tedious, or error-prone activities.

3. *Defense-in-Depth*: Have a series of defenses so that if an error isn't caught by one, it will probably be caught by another.

4. *Information Hiding*: The language should permit modules designed so that (1) the user has all of the information needed to use the module correctly, and nothing more; (2) the implementor has all of the information needed to implement the module correctly, and nothing more.

5. *Labeling*: Avoid arbitrary sequences more than a few items long; do not require the user to know the absolute position of an item in a list. Instead, associate a meaningful label with each item and allow the items to occur in any order.

6. *Localized Cost*: Users should only pay for what they use; avoid distributed costs.

7. *Manifest Interface*: All interfaces should be apparent (manifest) in the syntax.

8. *Orthogonality*: Independent functions should be controlled by independent mechanisms.

9. *Portability*: Avoid features or facilities that are dependent on a particular machine or a small class of machines.

10. *Preservation of Information*: The language should allow the representation of information that the user might know and that the compiler might need.

11. *Regularity*: Regular rules, without exceptions, are easier to learn, use, describe, and implement.

12. *Security*: No program that violates the definition of the language, or its own intended structure, should escape detection.

13. *Simplicity*: A language should be as simple as possible. There should be a minimum number of concepts with simple rules for their combination.

14. *Structure*: The static structure of the program should correspond in a simple way with the dynamic structure of the corresponding computations.

15. *Syntactic Consistency*: Similar things should look similar; different things different.

16. *Zero-One-Infinity*: The only reasonable numbers are zero, one, and infinity.

APPENDIX B

Principles of Software Management
(After Riefer [Ref. 13])

Principle 1: The Precedence Principle. Planning logically takes precedence over all other managerial functions.

Principle 2: The Effective Planning Principle. Plans will be effective if they are consistent with the organization's policy and strategy framework.

Principle 3: The Living Document Principle. Plans must be maintained as living documents or they quickly lose their value. Plans serve as the foundation for control. When they are not updated, control is severely impeded.

Principle 4: The Early Assignment Principle. Make *one* person responsible for software as early in the life of the project as possible. Ensure that he or she occupies a high enough position within the hierarchy to successfully compete for resources (dollars, people, etc.). Make this person accountable for the final results.

Principle 5: The Interface Principle. The efficiency of an organization is inversely proportionate to the number of interfaces it has to maintain during the performance of a job.

Principle 6: The Parity Principle. A software manager's responsibility for action should be no greater than that implied by the authority delegated to him.

Principle 7: The Quality Principle. Using a few experienced people for critical tasks (such as design) is often more effective than using larger, unskilled teams. An experienced software engineer is "worth his weight in gold."

Principle 8: The Personnel Development Principle. An open commitment to personnel development often pays dividends. Better trained technical and managerial personnel can effectively cope with tomorrow's problems instead of today's.

Principle 9: The Dual Ladder Principle. Promotion should be possible up either a technical or a managerial career path.

Principle 10: The Motivation Principle. Interesting work and the opportunity for growth and advancement will motivate people to achieve high productivity. McGregor's Theory Y holds - the individual will rise to the challenge of his capabilities.

Principle 11: The Leadership Principle. People will follow those who represent a means of satisfying their own personal goals. Success will come to those who ensure that personal goals are compatible with those of the organization.

Principle 12: The Communications Principle. Productivity is a function of the communications burden. As the burden increases, productivity decreases. In other words, the less communication required, the higher the productivity.

Principle 13: The Significance Principle. Controls should be implemented to alert managers promptly to significant deviations from plans.

Principle 14: The Measurement Principle. Effective control requires that we measure progress against objective, accurate, and meaningful standards.

Principle 15: The Exception Principle. The efficient manager will concentrate his control efforts on exceptions.

Principle 16: The Technology Risk Principle. Technology should be used only when the risk associated with it is acceptable.

Principle 17: The Improvement Principle. The manager who insists on doing things the way they have always been done will fail. New approaches must be used to meet new challenges. Your competition will not allow you to remain conservative in the extreme.

Principle 18: The Peter Principle of Software Management. Managers rise to their level of incompetence and are then transferred to head a software development project.

APPENDIX C

Characteristics of a Methodology
(After Wasserman [Ref. 14])

1. The methodology should cover the entire software development cycle.

2. The methodology should facilitate transitions between phases of the development cycle.

3. The methodology must support determination of system correctness throughout the development cycle.

4. The methodology must support the software development organization.

5. The methodology must be repeatable for a large class of software projects.

6. The methodology must be teachable.

7. The methodology must be supported by automated tools that improve the productivity of both the individual developer and the development team.

8. The methodology should support the eventual evolution of the system.

APPENDIX D

Twenty Hypothesized Problems in SEPM
(After Thayer, Pyster, and Wood [Ref. 27])

Planning

1. *Requirements:* Requirement specifications are frequently incomplete, ambiguous, inconsistent and/or unmeasurable.

2. *Success:* Success criteria for a software development are frequently inappropriate, which result in "poor-quality" delivered software; i.e., not maintainable, unreliable, difficult to use, relatively undocumented, etc.

3. *Project:* Planning for software engineering projects is generally poor.

4. *Cost:* The ability to estimate accurately the resources required to accomplish a software development is poor.

5. *Schedule:* The ability to estimate accurately the delivery time on a software development is poor.

6. *Design:* Decision rules for use in selecting the correct software design techniques, equipment, and aids to be used in designing software in a software engineering project are not available.

7. *Test:* Decision rules for use in selecting the correct procedures, strategies, and tools to be used in testing software developed in a software engineering project are not available.

8. *Maintainability:* Procedures, techniques, and strategies for designing maintainable software are not available.

9. *Warranty:* Methods to guarantee or warranty that the delivered software will "work" for the user are not available.

10. *Control:* Procedures, methods, and techniques for designing a project control system that will enable project managers to successfully control their project are not readily available.

Organizing

11. *Type:* Decision rules for selecting the proper organizational structure; e.g., project, matrix, function, are not available.

12. *Accountability:* The accountability structure in many software engineering projects is poor, leaving some question as to who is responsible for various project functions.

Staffing

13. *Project Manager:* Procedures and techniques for the selection of project managers are poor.

Directing

14. *Techniques:* Decision rules for use in selecting the correct management techniques for software engineering project management are not available.

Controlling

15. *Visibility:* Procedures, techniques, strategies, and aids that will provide visibility of progress (not just resources used) to the project manager are not available.

16. *Reliability:* Measurements or indexes of reliability that can be used as an element of software design are not available and there is no way to predict software failure; i.e., there is no practical way to show the delivered software meets a given reliability criteria.

17. *Maintainability:* Measurements or indexes of maintainability that can be used as an element of software design are not available; i.e., there is no practical way to show that a given program is more maintainable than another.

18. *Goodness:* Measurements or indexes of "goodness" of code that can be uses as an element of software design are not available; i.e., there is no practical way to show that one program is better than another.

19. *Programmers:* Standards and techniques for measuring the quality of performance and the quantity of production expected from programmers and data processing analysts are not available.

20. *Tracing:* Techniques and aids that provide an acceptable means of tracing a software development from requirements to completed code are not generally available.

APPENDIX E

Principles of Software Engineering Environment Design

1. *Methodology Support*: A software engineering environment should be designed to support a specific engineering methodology.

2. *Design Documentation*: A software engineering environment should support a design documentation system that is (1) complete, (2) capable of representing appropriate levels and types of abstractions with fine granularity, (3) universal, and (4) economical.

3. *Enforcement/Aid*: A software engineering environment should enforce technical correctness and conformance with management policies while aiding the user in maintaining these standards.

4. *Consistency Principle*: "Permanent" alteration of one view of a design should not be "accepted" by the environment until all related views are made to be consistent with the change.

5. *Structure Manipulation*: The user of a software engineering environment should be able to create, reference, locate, alter, and delete structures defined within the environment. He should also be able to display meaningful representations of them within the context of any applicable type or level of abstraction.

6. *Static Analysis*: A software engineering environment should (1) allow the user to make assertions about the static structure of a module, program, or system of programs and then report back to the user which assertions are not valid and why, and (2) allow the user to request certain information about the static structure and then report this information back to the user in a lucid format.

7. *Dynamic Analysis*: A software engineering environment should (1) allow the user to make assertions about the dynamic structure of a module, program, or system of programs and then report back to the user which assertions are not valid and why, and (2) allow the user to

request certain information about the dynamic structure and then report this information back to the user in a lucid format.

8. *Data Collection*: A software engineering environment should provide for the unobtrusive collection of software management data.

9. *Audit Trail*: A software engineering environment should maintain audit trails showing the relationships between specifications, designs, implementations, maintenance and enhancements, and the resources expended in these activities. Further, such audit trails should be navigable in both directions from any point.

10. *Information Organization*: The information gleaned from the various documentation and data collection efforts should be organized for easy reference and maintenance.

11. *Management Control*: A software engineering environment must be controllable by the management of the organization it serves.

12. *Organizational Structure*: A software engineering environment should be partitionable along the structural lines of the organization so that individuals and groups may have the necessary levels of privacy and isolation from other individuals and groups, and so the communications among them may be reasonably controlled by forcing them through established interfaces. However, it should also be possible to allow information to flow across organizational boundaries, when needed, through specially designated and controlled interfaces.

13. *Interface Language Design*: The language of interaction between a user and an interactive application should be designed according to the principles of good programming language design.

14. *Multi-level Help*: For any interactive tool, there should be a hierarchy of "help" levels ranging from no prompts at all to on-line instruction. Furthermore, within an environment, the user should be able to set the "help level" on a tool-by-tool basis (fine granularity of help levels).

15. *Configurability*: The user should be able to configure his interface to the environment (tool) within the constraints mandated by higher management. This capability should display a fine granularity.

16. *Meaningful Response*: The appearance of the display and the text of messages in response to user actions must be appropriate to those actions.

17. *Rapid Response*: Simple and frequently used functions should have an immediate (in terms of human reflexes) effect upon the display.

18. *Status Reporting*: Lengthy functions should periodically update the display to assure the user that progress is being made in carrying out the requested function.

19. *Display Inertia*: The display should change by the least amount possible in response to a user action. The display should not, however, violate the Meaningful Response principle.

20. *Extensibility*: An environment should be extensible in the sense that is must be possible to add tools at any time and at any level which enjoy the same level of control and integration as the original tools.

21. *Reversibility*: Any action taken by a user must be reversible for some period of time or number of subsequent actions.

LIST OF REFERENCES

1. Shooman, M. L., Software Engineering, McGraw-Hill, 1983.

2. Boehm, B. W., "Software Engineering", *IEEE Transactions on Computers*, Dec. 1976, Vol. C-25, No. 12, pp.1226-41.

3. Jensen, R. W. and Tonies, C. C., Software Engineering, Prentice-Hall, 1979.

4. Bauer, F. L., "Software Engineering", *Information Processing 71*, North Holland Publishing Co., 1972.

5. MacLennan, B. J., Principles of Programming Languages: Design, Evaluation, and Implementation, Holt, Rinehart and Winston, 1983.

6. Bohm, C. and Jacopini, G., "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", *Communications of the ACM*, Vol. 9, No. 5, pp. 366-71.

7. Dijkstra, E., "Go To Statement Considered Harmful", *Communications of the ACM*, Vol. 11, No. 3, pp. 147-48.

8. Buxton, J. N. and Druffel, L. E., "Rationale for STONEMAN", *Fourth International Computer Software and Applications Conference. October 1980. Chicago. Illinois*, pp. 66-72, IEEE, 1980.

9. Hoare, C. A. R., "The Emperor's Old Clothes", *Communications of the ACM*, Vol. 24, No. 2, pp. 75-83.

10. Hayes, J. P., Computer Architecture and Organization, McGraw-Hill, 1978

11. Patterson, D. A. and Sequin, C. H., "A VLSI RISC", *Computer*, Sept. 1982, pp. 8-21

12. Peters, L. J., Software Design: Methods and Techniques, Yourdon, Inc., 1981

13. Reifer, D. J., "The Nature of Software Management: A Primer", Tutorial: Software Management (First Ed.), IEEE, 1979

14. Wasserman, A. J., "The Ecology of Software Development Environments", Tutorial: Software Development Environments, IEEE, 1981

15. Chapin, N., Flowcharts, Petrocelli Books, 1971

16. Brooks, Jr., F. P., The Mythical Man-Month, Addison-Wesley, 1975

17. Yourdon, E., Techniques of Program Structure and Design, Prentice-Hall, 1975

18. Ledgard H. and Chmura, L., COBOL With Style, Hayden, 1976

19. Chu, Y., Software Blueprint and Examples, D. C. Heath, 1982

20. Yoder, C. M. and Schrag, M. L., "Nassi-Shneiderman Charts: An Alternative to Flowcharts for Design", *Proceedings. ACM SIGSOFT/SIGMETRICS Software Quality and Assurance Workshop*, Nov. 1978

21. Stay, J. F., "HIPO and Integrated Program Design", *IBM Systems Jorunal*, 1976

22. Caine, S. H. and Gordon, E. K., "PDL - A Tool for Software Design", *Proceedings. National Computer Conference*, 1975

23. Stevens, W. P., Myers, G. J., and Constantine, L. L., "Structured Design", *IBM Systems Journal*, 1974

24. Teitelbaum, T. and Reps, T., "The Cornell Program Synthesizer: A Syntax Directed Programming Environment", *Communications of the ACM*, Sept. 1981, pp. 563-73

25. Teitelman, W. and Masinter, L., "The Interlisp Programming Environment", *Computer*, April 1981, pp. 25-34

26. Kernighan, B.W. and Plauger, P. J., Software Tools, Addison-Wesley, 1976

27. Thayer, R. H., Pyster, A., and Wood, R. C., "The Challenge of Software Engineering Project Management", *Computer*, Aug. 1980, pp. 51-59

28. Perlis, A. J., Sayward, F. G., and Shaw, M. (ed.), Software Metrics, MIT Press, 1981

29. DeMarco, T., Controlling Software Projects, Yourdon Press, 1982

30. Spier, M. J., Gutz, S., and Wasserman, A. I., "The Ergonomics of Software Engineering - Description of the Problem Space", Software Engineering Environments, H. Hunke (ed.), North-Holland Publishing Co., 1981

31. Kernighan, B. W. and Mashey, J. R., "The UNIX Programming Environment", *Computer*, April 1981, pp. 25-34

32. Hansen, W. J., "User Engineering Principles for Interactive Systems", *Fall Joint Computer Conference Proceedings*, 1971, AFIPS Vol. 39, pp. 523-532

33. Goldberg, A., "The Influence of an Object-Oriented Language on the Programming Environment", Interactive Programming Environments, Barstow, D. R., Shrobe, H. E., and Sandewall, E. (ed.), 1984, pp. 141-174

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	Commandant (G-PTE) U. S. Coast Guard Washington, D. C. 22590	2
4.	Commander (As) Atlantic Area Coast Guard Room 3, Building 110 (AMVER) Governors Island, NY 10004	1
5.	Professor B. J. MacLennan, Code 52 Ml Naval Postgraduate School Monterey, California 93943	2
6.	Professor G. H. Bradley, Code 52 Bz Naval Postgraduate School Monterey, California 93943	2
7.	Mr. E. F. Frost, Jr. 1954 Bruce Street Lakeland, Florida 33801	1
8.	Captain L. L. Jackson 6542 Divine Street McLean, Virginia 22101	1
9.	Commander J. H. Hanna 9311 Locksley Road Fort Washington, Maryland 20744	1
10.	Lieutenant Commander J. R. Frost 6542 Divine Street McLean, Virginia 22101	2