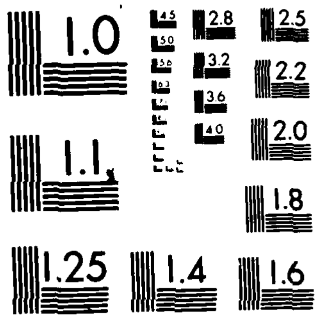
AD-A138 663 DOCUMENTATION FOR SOFTWARE MAINTENANCE(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA J F HALL DEC 83 1/1

UNCLASSIFIED F/G 9/2 NL

END
DATE
FILMED
4-84
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DOCUMENTATION FOR SOFTWARE MAINTENANCE

by

John F. Hall, II

December 1983

Thesis Advisor: Gordon Bradley

Approved for public release; distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A138663	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Documentation for Software Maintenance		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis December, 1983
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John F. Hall, II		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 December 1983
		13. NUMBER OF PAGES 62
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 Documentation, software maintenance, documentation hierarchy, minimal documentation, documentation categories

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

Documentation as an effective method of transferring information between individuals in order to reduce software maintenance costs is examined. Various categories of documentation are identified and evaluated as to their effectiveness toward easing the maintenance effort. The concept of minimal documentation is introduced as the solution to the problem of determining the correct amount of information required for a specific maintenance task. The idea of utilizing an explicit documentation hierarchy as the ideal method for storing explicit documentation is proposed. With (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued)

the proper implementation of the documentation hierarchy, the minimal documentation concept can be realized, and the maintenance effort reduced.

Documentation for Software Maintenance

by

John F. Hall, II
Lieutenant, United States Navy
B.S.E.E., University of Washington, 1978

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1983

Author: John F. Hall II

Approved by: [illegible]
Thesis Advisor

Paul W Callahan
Second Reader

[illegible]
Chairman, Department of Computer Science

[illegible]
Dean of Information and Policy Sciences

ABSTRACT

Documentation as an effective method of transferring information between individuals in order to reduce software maintenance costs is examined. Various categories of documentation are identified and evaluated as to their effectiveness toward easing the maintenance effort. The concept of minimal documentation is introduced as the solution to the problem of determining the correct amount of information required for a specific maintenance task. The idea of utilizing an explicit documentation hierarchy as the ideal method for storing explicit documentation is proposed. With the proper implementation of the documentation hierarchy, the minimal documentation concept can be realized, and the maintenance effort reduced.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. THE PROBLEM

There is much discussion in the software engineering literature concerning the overwhelming cost of software maintenance. It has been indicated that in some systems up to eighty percent of the cost of a software system is consumed in the maintenance phase of the software life cycle [Ref. 1]. In order to properly maintain the software, it must be properly documented. Often the same person does not perform tasks in all phases of the life cycle, thus without documentation, continuity between the phases can be lost. Sometimes the only interface between each phase is a piece of documentation. This points out the criticality of documentation in the software life cycle; if the documentation between phases is not done well, much of the work on the project must be recreated for subsequent phases.

B. PURPOSE AND APPROACH

There is a lack of cohesive discussion in current literature concerning proper documentation for efficient software maintenance. Because of the tremendous cost involved with software maintenance, an attempt to ease the maintenance effort needs to be made through the use of adequate documentation.

The purpose of this thesis is to address documentation as a method of information transfer throughout the life cycle of a software project in the support of software maintenance. Various types of documentation are discussed and evaluated as to their effectiveness toward easing the maintenance effort. An attempt is made to determine the proper type and amount of information needed to effectively maintain the software project. An effort is made to categorize and quantify different aspects of documentation based on task and user needs. The concept of a documentation hierarchy is put forth as a method for organizing these aspects. The idea is to give the receiver of that information precisely the amount of information required to complete the maintenance task. Too much information can bog one down with unnecessary details while too little information can cause one to waste many hours in trying to understand the program. Thus a solution to the problem of accessing the exact quantity of documentation is offered.

Chapter I gives an overview of the documentation problem as it relates to software maintenance. A description of the approach for the thesis is given along with some general definitions of terms used in the software maintenance environment. Also, the idea of minimal documentation is introduced in this chapter.

Chapter II discusses software maintenance in detail with a look at the software life cycle. A software project scenario is described to set basic guidelines for the thesis. The different types of maintenance as they relate to the software modification task are described along with the identification of some of the causes and solutions for the software maintenance problem.

Chapter III introduces the idea of the transfer of knowledge between individuals as being the goal of effective documentation. This knowledge transfer is accomplished by utilizing various methods for recording information. Documentation is then catagorized according to the type of information that is conveyed and also according to dependencies based on a person's skill and position on the maintenance team. Finally, the role of documentation for a project is discussed.

Chapter IV introduces the concept of a documentation hierarchy in support of minimal documentation. The levels of the hierarchy are based on the level of detail contained in the documentation. The various users of the documentation need only to access the proper level in the documentation hierarchy in order to have the minimal documentation that is required for the completion of the task at hand.

Chapter V discusses the effectiveness of several specific forms of documentation in relation to the performance of the maintenance task. The eval· tion is made that there is not a "best" form of documenta on for all maintenance tasks. The most effective docum .cation form varies with the maintenance task and the · e of programming processing (sequential or concurrent) be - used.

Chapter VI consists of the thesis conclusions and recommendations.

C. DEFINITIONS

Certain basic definitions are needed in order to properly address the issue of software documentation as it relates to software maintenance. For the purposes of this thesis, software maintenance will be considered to be the process of updating and correcting a software system once the project is delivered and made operational.

Software documentation is the recorded information that can be used to transfer information and ideas from one person to another. Unlike software maintenance, which has been defined to begin after the project is delivered, software documentation is produced throughout the entire software life cycle from the conceptual phase to the support phase. Since documentation follows the evolution of a software system, adequate and reliable documentation is invaluable when it comes to maintaining the system.

Minimal documentation is defined as the exact amount of documentation on a project that is required by the receiver to accomplish the receiver's task. When the minimal documentation concept is used, only the precise amount of documentation that is needed is accessed, and the receiver is not forced to wade through unnecessary information. Just enough information is recorded so that the receiver is able to proceed with the job at hand. This, then, is an idea of documentation efficiency with no more and no less information being exposed to the receiver than is actually needed.

Maintainability is a term that must be clarified. Martin and McClure [Ref. 2] define maintainability as the "ease with which a software system can be corrected when errors or deficiencies occur, and can be expanded or contracted to satisfy new requirements." Maintainability of a software system can be enhanced with the availability of adequate minimal documentation.

Understandability is considered to be one of the most important concepts in the realm of maintainability. Martin and McClure define understandability as "the ease with which we can understand the program purpose and how the program achieves its purpose". Since documentation transfers information concerning software system evolution, the minimal documentation concept aids in the achievement of program understandability.

II. THE SOFTWARE MAINTENANCE PROBLEM

A. THE SOFTWARE LIFE CYCLE

The development of a software project goes through several phases from conception to actual system operation. This development process is called the software life cycle. There are several models available to represent the software life cycle. The one used by the Department of Defense as indicated in Department of Defense Instruction 5000.1 is presented in Figure 2.1. It gives a reasonable representation of most simple models. An advantage of this model is that each major phase is broken into its subsequent subphases. This model is general enough to be applied to most software systems, with the details being left to the specific project.

Documentation must be carried throughout the life cycle in order to promote understandability in subsequent phases. Ultimately, the ideal documentation contains enough information such that when the program is completed and is operational, it can be maintained effectively.

The major problem with the Department of Defense model is the implication that is given concerning the flow of the software life cycle. One is left with the idea that as one phase abruptly halts, the next phase begins. In practice, the phase boundaries are somewhat obscure. Quite often work on one part of a phase begins before all work in a previous phase is completed. Also one gets the impression that there are no interdependencies between the phases. In reality, decisions made in one phase often directly affect the work of a subsequent phase. This makes each phase somewhat dependent upon decisions made in a previous phase. Also

DEFENSE SYSTEM LIFE CYCLE MAJOR PHASE	SOFTWARE LIFE CYCLE SUBPHASE
Conceptual	Requirements Definition
	Requirements Validation
Validation	Validation
Full-Scale Development	Full-Scale Development
Production	Production
Deployment	Debugging
	Fine Tuning
Support	Maintenance
	Modification

Figure 2.1 Department of Defense Life Cycle Model.

there are times when a decision made in one phase is determined to be unrealistic by restrictions or actions taken in a following phase. Therefore, a feedback mechanism is needed to carry information between phases in order to keep the software project development moving.

Documentation is the method of recording information that is to be transferred forward and backward to aid in software modification. Figure 2.2 gives a more realistic view of the software cycles indicating some of the phase interrelationships [Ref. 3].

Figure 2.2 shows validation and verification subphases in the requirements and design phases of the cycle. This is important because each phase should be verified as being possible and feasible as early in the cycle as possible in order to avoid unnecessary work. For example, it would be wasteful to work through to the implementation phase only to find out that the project was never feasible in the first place.

Studies indicate that the most economical time to catch and correct a problem is as early in the development cycle as possible. The cost of detecting and correcting an error more than doubles for each phase through which it passes undetected. This rate of cost increase holds true for each subsequent phase through which the problem passes without detection. [Ref. 3] and [Ref. 4].

While the simplistic view presented in Figure 2.1 is relatively easy to comprehend, it is extremely important to remember the interrelationships between the various phases as indicated by Figure 2.2. With those interrelationships being kept in mind, the simplified life cycle model shown in Figure 2.1 will be adequate for use in this thesis.

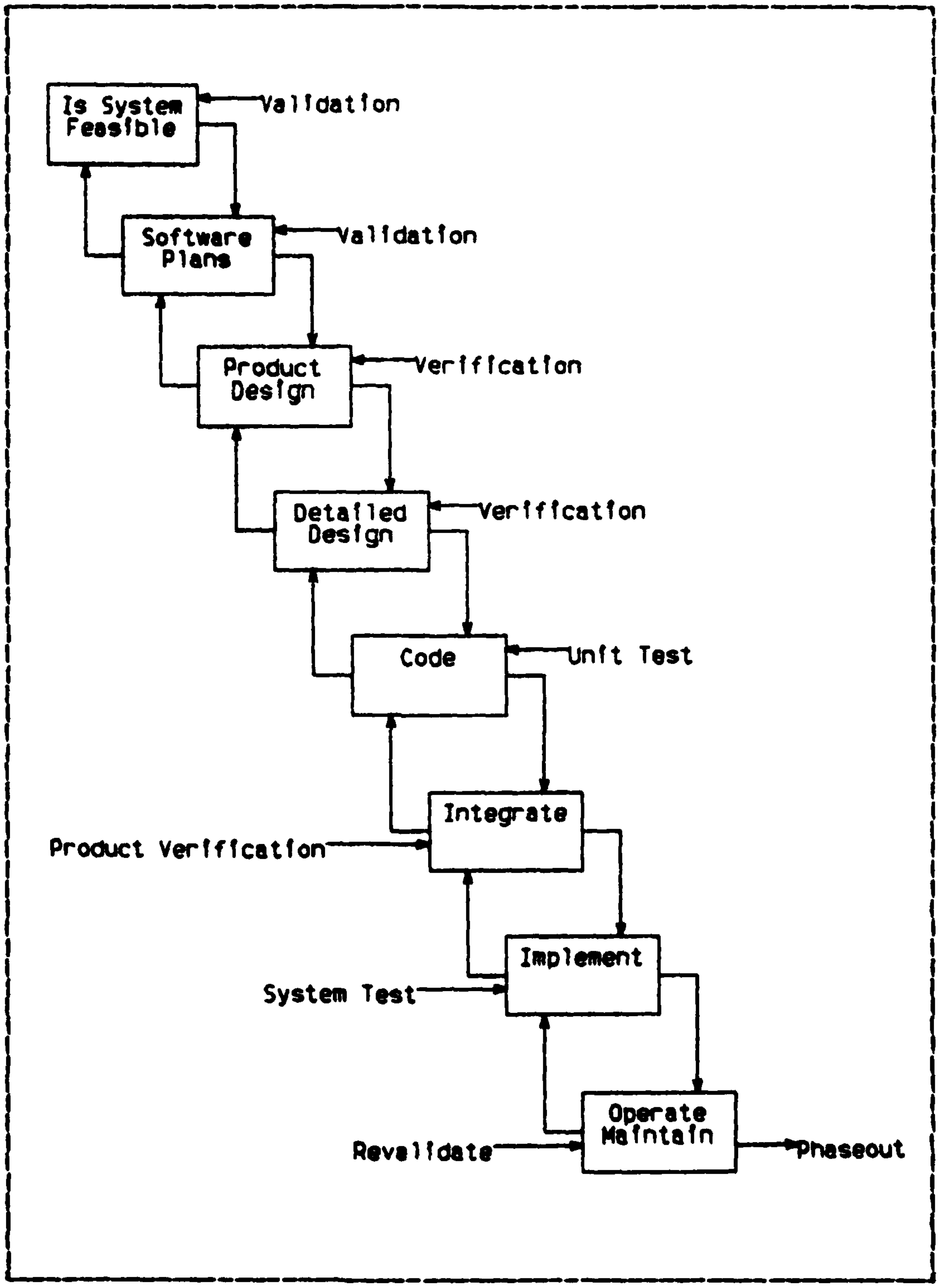


Figure 2.2 The Waterfall Model of the Software Life-cycle.

B. THE SOFTWARE PROBLEM

Bohem [Ref. 3] gives us some insights into the magnitude of the economics involved with the software problem. The annual cost of software for the United States in 1980 was about 2 percent of the Gross National Product. The cost is expected to grow faster than the general rate of the economy thus representing an even larger proportion of the Gross National Product as time goes on. The portion of the effort spent on software maintenance has increased faster than the effort spent on software development. With the growth of software maintenance taking such a large portion of the total cost of a system, it would be wise to find ways to enhance the efficiency of the maintenance effort.

Along with the economic issues of software maintenance, we must look at the social aspects of computers as they relate to software. Things such as computerized billing and banking have made a permanent impact upon the lives of most Americans. An increasing number of workers in the United States will be relying on computers to perform tasks involved with their daily work. By 1985 it is predicted that approximately 40 percent of the working population will fall into that category. With this kind of computer and software proliferation, there will be continued growth in the amount of software that is needed. This growth of software translates into a significant amount of necessary software maintenance as both the software system and the state of technology change.

As the need for software maintenance increases, it becomes imperative that maintenance efficiency be improved. The idea of using minimal documentation in order to improve understandability, which in turn aids maintainability, is seen as a way to increase the efficiency of the software maintenance effort.

1. Scenario

There are many types of programs that are developed ranging from very small to very large, and the size of a program can determine the software documentation issues related to that particular program. In order to address specific documentation issues we need to focus on a particular Scenario.

The program with which we will be concerned is one of medium length involving thirty to forty thousand lines of code. It is a software program that is to be maintained, so it is neccessary that software documentation be generated. (If a program were never to be modified, then documentation would not be necessary.) For the purpose of this thesis, the program code is not considered to be a form of documentation. The program is one that was developed by a software development team (as opposed to being developed by a solo programmer) with documentation maintained throughout the development. The development followed the basic guidelines as indicated in the Department of Defense life cycle model (Figure 2.1). The program developers are not the end users of the system. The program is embedded in an environment that is subject to change.

When a modification to the system is required, a change request protocol is followed in which a requested change is considered and a determination is made as to whether the change should be incorporated into the existing system. If the change is to be made, it is acted upon by a designated maintenance team. The personnel assigned to the maintenance team may or may not have other collateral duties in the organization, and they may or may not have had any connection with the original development of the system. Emergency changes are implemented as quickly as possible while routine modifications are implemented in an annual system update.

The system is considered to have a life expectancy of approximately twenty years, and it is further assumed that the system has been in operation for several years with maintenance being accomplished and documentation being updated accordingly.

2. Understandability

Understandability is considered to be one of the most important concepts in the realm of maintainability. If a piece of software that is to be maintained proves to be both efficient and successful, yet is not understandable to the maintainers, it can be difficult and expensive (if not impossible) to modify to meet changing needs.

In a good system, there is information available as to the purpose of the system, the proper use of the system, and the proper maintenance of the system [Ref. 2]. All phases of the life cycle will have accompanying documentation concerning the development at each stage of the system, and that documentation will carry the required information that aids in the understandability of the program.

Familiarity is a factor that helps determine the effectiveness and understandability of a program. A person who is very familiar with the code and the functioning of the system would probably not have great difficulty in understanding the system, even if the documentation were somewhat lacking in quality and the system itself were very complex. On the other hand, the inexperienced or unfamiliar maintainer would probably have difficulty in understanding the program. We will see later how the factor of familiarity determines for an individual the level of detail needed in the documentation.

Martin and McClure state that understandable programs generally have several common characteristics: structuredness; consistency; completeness; conciseness; and documentation. Each of these characteristics will be discussed in more detail.

a. Structuredness

The effective structuring of a program increases understanding by standardizing the program format. The standardization will set restrictions and guidelines on the logical flow of the program. Program modules will be set up in a hierarchical manner with the order of execution determined by the guidelines. The use of these guidelines for program construction will provide a consistent logic that will aid the understandability of the overall system.

b. Consistency

A program should be written in a consistent style in accordance with established programming standards. The structuredness mentioned above can be considered to be a method of developing a consistent style. It is difficult to understand a program in which the style of writing does not follow a common method of construction. This is sometimes difficult to accomplish when several members work together as a team on a project unless close communication control is maintained. Consistent types of comments must be maintained. When a module is described, it should handle the description of the piece of code the same way every other module is handled in terms of the amount of detail discussed and the order in which the information is provided. Variable names should be selected with the same sort of reasoning throughout the program, and the program should be consistent with the design instructions. In-line comments should be used to clarify coding statements, and this practice should be consistent throughout the program. There should be no visible evidence that the program was not written by one person with a consistent train of thought throughout the development process.

c. Completeness

A complete program has all of its components available to use and to be perused by the maintainer. The maintenance person must be able to access all parts of the program that are related to the maintenance function if understandability is to be accomplished. Any variables or modules should be included in a cross-reference scheme so that the maintainer can trace a program component throughout the system. Every unusual feature in the program should be clearly explained, and error messages should be made understandable.

d. Conciseness

A concise program is one that uses only the coding necessary to achieve the design requirements with no extra (perhaps unused) pieces of code. Every piece of code must be reachable by some action of the program. Unused variables and duplicate functions should be nonexistent and comments should not be excessively verbose or cryptic in meaning. Understandability decreases when complexity overtakes simplicity. The system, though it in itself may be complex, can be simplified through the proper use of conciseness principles.

e. Documentation

Perhaps the concept that pulls all of the above understandability methods together is the development (and use) of good documentation techniques. The use of all of the above ideas that promote understandability in a system may not in themselves be successful.

The structuredness of a program must be documented in such a way that the structurization methods are understood by the maintainers. The consistent program must have the modules described and documented in a consistent way. Comments should be arranged near each module to describe the module in some detail. The module comments should include the purpose of the module, the variables used or modified in the module, and a description of the output of the module. The module description should indicate the relative postion of the module with respect to other modules in the program to give some idea as to how the module was reached and how it fits into the program hierarchy.

Documentation also aids understandability by containing information as to the completeness of the program. Information is recorded as to how a module of the program is reached and how each module is related in the overall system scheme. Proper documentation should also be concise with only the necessary information being provided to the maintainer so as to enhance understandability without confusion.

3. Maintenance Effort

Since the maintenance effort represents such a large portion of the overall cost of a software system, a look at the software maintenance effort and ways in which to make this effort efficient by means of minimal documentation is in order.

A survey of data processing maintenance activities by Lientz and Swanson [Ref. 5] shows that only half of the people who are assigned to maintain programs actually worked on their development. This fact is significant in that a lack of continuity of the original thinking occurs. To make matters worse, the number of development personnel assigned to the maintenance effort diminishes even further as the life of a software system is extended. Good documentation passes information about the program between those personnel developing the program and those maintaining the program, thus preserving the continuity of thought.

In order to gauge the magnitude of the maintenance effort, Lientz and Swanson describe the effort as the result of the combination of four variables: system age; system size; relative amount of routine debugging; and the relative development experience of the maintenance personnel. As the system age extends, the system size tends to increase leading to a greater maintenance effort. With an increase in system size, the system tends to need more routine debugging, again increasing the maintenance effort. When the system age increases, there is also an increasing amount of personnel turnover which leads to the declining relative development experience of the maintainers, thus again causing an increase in the maintenance effort. This maintenance effort increase, of course, results in a rise of overall maintenance costs.

Minimal documentation can be used as a means to promote program understandability which will make the maintenance effort become more efficient, and in turn cause a decrease in software maintenance costs.

4. Types of Maintenance

Generally software maintenance can be divided into three categories: corrective; adaptive; and perfective maintenance [Ref. 6]. Corrective maintenance is considered to be purely the correction of software errors. Though corrective maintenance is traditionally seen as the most obvious type of maintenance task, it is interesting to note that the time spent solely on the correction of errors amounts to only seventeen to twenty-five percent of the maintenance person's time [Ref. 5] and [Ref. 7].

Adaptive maintenance is considered to be the process of adapting or changing the software to meet the environmental constraints of the system. An adaptive change would be considered to take place if a new operating system is to be installed on the computer that is used by the program being maintained. This area of maintenance takes up about eighteen to twenty-five percent of the time spent on maintenance [Ref. 7].

Perfective maintenance concerns the "perfecting" of the system by making user-requested changes to the software to make the system perform "better". It is interesting to note that while the perfective maintenance activity (sometimes called providing enhancements) does not involve the act of correcting errors, it takes between fifty and sixty percent of the maintenance person's time--by far the largest single time chunk in the software maintenance effort [Ref. 7]. This is in contrast to what is generally considered to be "real" maintenance, that of the corrective type.

5. Causes of Maintenance Problems

It is beneficial to look at some of the things that create the need for software maintenance and ways that the use of proper documentation can ease the maintenance task.

Schneidewind [Ref. 8] indicates that several items bring about maintenance problems. One is the fact that maintenance is often viewed by both designers and users as a task that is not very glamorous. This leads to a tendency for personnel to want only to design systems while letting the maintenance aspect of the system take a low priority. This leads to many of the maintenance problems being ignored during the development phase, including the proper documentation of the project as it progresses. Maintenance is often not even considered during the software development process.

In reality, documentation and maintenance ideas must parallel system development through all of the software life cycle phases, and actually become an integral part of the design criteria. This is necessary if a program is to be easily understood and efficiently operated and maintained.

Glass and Noiseux [Ref. 9] show that a traditional approach to the software problem has led to the practice of simply tacking on maintenance aids as an afterthought. The concept of software maintenance, particularly in something other than that of a corrective nature, seems to be something that receives very little attention.

Now let's look at the software maintenance problem as viewed by managers who plan the maintenance effort based on how they perceive the maintenance problem. Lientz and Swanson [Ref. 5] report in a study that managers perceive the lack of user knowledge as by far the most dominant problem in the maintenance effort. Following the user knowledge problem in order of highest to lowest significance were: programmer effectiveness; product quality; programmer availabiltiy; machine requirements; and finally system reliability. When the system is properly documented, user knowledge about the program development and maintenance techniques can be increased. Increased knowledge can cause more efficient use of the maintainer's time thus making the maintainer available more often to perform other tasks. Documentation can also be used to record machine requirements in such a way that they are made available for consideration in maintenance decisions. When documentation is used properly, it can reduce these problem areas as mentioned and result in an overall increase in system reliability.

The study further revealed a misconception commonly held by managers. Problems were perceived by managers to be greater if a lot of corrective maintenance time was spent on the system. As mentioned earlier, corrective maintenance is not the big time consumer whereas perfective maintenance does in fact consume the bulk of the maintainer's time. It is interesting to note that there were no significant findings to indicate that there was any time at all allotted to the maintenance personnel for the act of performing purely perfective maintenance. When it is understood that most of the actual maintenance time is spent in making program enhancements (that is, the perfective maintenance), it becomes clear that managers need to re-evaluate the way they allot time for maintenance.

Another reason for maintenance problems can be considered to be both a cause and a result of the above mentioned problems. This reason for maintenance problems is the lack of good documentation. Schneidewind points out that one of the problems involved with software maintenance is that no tracebility is built into the software. This problem could be resolved with the appropriate documentation technique. In order to convey ideas from old maintainers to new maintainers, good documentaion is needed. For example, formal specifications of the problem for which the system is designed must be presented and documented so that a later change to the program can be evaluated against the documented reason for a particular design system. This thesis will explore the documentation problem in more detail in later chapters.

6. Solutions

As implied from the above discussion of the causes of the software maintenance problem, the use of proper documentation is the key to many maintenance problem solutions.

The solutions require the cooperation of managers, users, and maintainers. Good maintenance techniques are also necessary and must begin at the design and requirements phases of the life cycle. Several ideas are presented by Schneidewind to ensure good maintenance practices and good documentation.

To start, good maintenance techniques must be planned from the beginning of a software project. This means that the project must be designed with maintenance integral to the entire life cycle, not added later when the project is completed. Since enhancements take up most of the maintenance effort, it is necessary that the design ideas incorporate the understanding that enhancements will take place and that managers plan maintenance time accordingly.

One way to enhance maintenance personnel understanding is to use modularity techniques in the design of the software project. This means that common ideas or concepts should be kept together in a logical sense that would be easy for the maintainer to follow utilizing proper documentation. The documentation should be able to convey the module concepts used in the design of the software to the person who needs to make software modifications.

Along with the modularity techniques, the idea of independence among code, data, and data base is in order. This independence allows certain amounts of code, functions, subroutines, etc. to be changed without devastating effects taking place in other portions of the program. Information hiding techniques are valuable when designing a program with modular independence for ease of maintenance.

Schneidewind also gives several ideas concerning the proper development of documentation along with the project in order to ensure the ease of maintenance. One idea is to design the documentation first. Many a programmer knows how bothersome this task can be. Just think of how many times the flow charts or refinement procedures were written after the program itself was actually written. The problem with writing the documentation after the program is completed is that the documentation that is supposed to aid in the development and decision making processes cannot possibly be used. Also the documentation becomes static in nature. That is, all the dynamic creativity cannot be included in the process of documentation and all we see is the final resulting document. This is very much like a college professor receiving a math or physics exam with only the answers and no actual work shown.

Another idea for good documentation is that the specifications and standards should require aids that promote the understandability of the program. This includes the concept of providing comments in the program listing, references in the source listings for certain software specifications, and any other references necessary to trace through other related documents.

System specifications should be designated as to the kind of information that is to be conveyed to the maintainer via documentation. This idea implies that certain types of documentation convey certain kinds of information, and that only certain bits of information are required for certain maintenance functions. The idea of different kinds of documentation delivering various kinds of information, and the effectiveness of the forms of documentation will be discussed below.

III. SOFTWARE DOCUMENTATION

This chapter covers many aspects of software documentation. Some background is given on documentation in general, including a discussion about the purpose of documentation. The various categories of documentation are explained based on the characteristics of the documentation and the needs of the user. Control and development documentation is discussed, as is static and dynamic documentation. Also the categories of implicit and explicit documentation as they pertain to the maintenance role are explained. Documentation dependencies with regard to skill levels and position level of personnel on the documentation team are discussed.

A. DOCUMENTATION BACKGROUND

The primary purpose of documentation is to impart knowledge about the system to pertinent personnel by transferring recorded information. Other than the code itself, the only source of written information about the program is the documentation. Misunderstandings can occur when our informal, abstract ideas are translated into formal, concrete pieces of information. This information transfer is accomplished by recording on various forms of information storing media. This includes such methods as manually recording facts on paper, or electronically placing them in a computer memory. These various information recordings become forms of documentation. The documentation challenge, then, is to find a way to ensure the unmolested receipt of intended concepts by the documentation users. When this challenge is met effectively, documentation is considered to have accomplished its goal of successfully conveying relevant information from one person to another. Chapter V discusses in more detail the particular documentation formats and their relative effectiveness.

One cannot hope to effectively transfer knowledge about the system by simply flooding the maintainer with all the information that can possibly be assembled. It is necessary to distinguish between types of documentation, and discern the type of information that each conveys. Not all types of documentation are adequate for all types of information transfer, and consequently, not good for all types of maintenance chores.

B. DOCUMENTATION CATEGORIES

While the idea of transferring information is clear, the idea of what kind and how much information to transfer is not so clear. The amount of information to transfer is both task dependent and programmer dependent. Both the maintenance task to be accomplished and the level of expertise of the maintenance person should be considered when deciding upon the category of documentation to be used. For example, if the documentation is to be used by someone who is very familiar with the system and the type of changes to be applied to the program, the documentation needed would be of a level that is far less detailed than that of someone who had never worked on the system before. It would be helpful to find ways to categorize documentation in order to gain an understanding of the values of each.

1. Internal and External Documentation

One way of categorizing documentation is based on how the documentation itself is transported (internally or externally). Internal documentation is the documentation that is carried along with the code (perhaps in a separate file. It is usually embedded in the form of comments or cross reference listings, and is not executable. It is an integral part of the program, so it is always available. It is easy to maintain because it is as easy to update as the code itself. When a software modification is made, it is a simple matter to modify the internal documentation. Because of the ease of update, internal documentation is considered by programmers to be very reliable, and the programmers have a high level of confidence in the currency and accuracy of the internal documentation.

External documentation is the documentation that exists outside the source code of the program. This includes such things as data flow diagrams, flow charts, and any other mode of recording program information that is not an integral part of the program. This type of documentation is more difficult to maintain than the internal documentation because it usually exists in hard copy only, thus pen and ink changes are required for making documentation modifications. Because of the difficulty encountered in the updating of external documentation, often it is not updated when the system is modified. This leads to a low level of trust among programmers with regard to the reliablility of external documentation. This low level of confidence in the currency of external documentation is often perpetuated by the feeling that there is no reason to update it since it is not current anyway, and even if it is updated, it won't be trusted.

2. Dynamic and Static Documentation

Documentation can also be considered to be either static or dynamic in nature. Dynamic documentation involves the conveyence of ideas about the actual developmental thought processes. This would include the recording of ideas that begin with the first thoughts in the conceptual phase of the life cycle. It also includes the mistakes made and the ideas considered and rejected for any reason. Prior mistakes and the reasons as to why they were mistakes can provide vital insights to the maintainers when new changes are being considered.

The dynamic nature of the documentation comes from the fact that the entire decision-making process can be actively recorded and transmitted to the receiver of the documentation. The significance of this type of documentation, then, is the fact that later enhancements to the program (remember, it is the enhancements that make up the biggest part of the maintenance workload) can be considered in light of original design decisions. Much redundancy in the consideration of enhancements stands to be saved when proper dynamic documentation is used.

Static documentation can be considered to be the "final product" of the documentation process. It is a recording of the current static state of the program at some point in time, and it does not provide any indication of the dynamics involved in the evolution of the program reaching that state. This type of documentation includes things such as a system or program flow chart or a resource diagram. It is this type of documentation that conveys the ideas of the program or system itself, and how it functions.

It must be realized that both of these types of documentation are necessary for the proper transfer of knowledge from the designer to the maintenance person. Without both types, either the original thought "flavor" of the designer's intent is lost with the passage of time, or the understanding of the processing of the program is lost.

3. Implicit and Explicit Documentation

Another categorization of documentation is the notion of implicit and explicit documentation. Explicit documentation can be thought of as the documentation that is physically available, in whatever form (dynamic or static), at varying levels of detail. Explicit documentation could, therefore, include documentation such as comments, manuals, and flow diagrams.

Implicit documentation is a more subtle and abstract type of documentation. This type of documentation consists of the "essence" of a program that is made available by consolidating information from one or more forms of either the dynamic or static documentation. This concept of implicit documentation, then, involves a synergistic effect that provides a high level understanding of the system without large amounts of explicit physical information necessarily being accessed. Implicit documentation provides the "Big Picture" for the receiver of the information when various amounts of physical documentation are assimilated. Thus implicit documentation captures the concept of transferring between individuals knowledge that would be difficult to impart through language or explicit documentation.

It is sometimes very difficult to convey abstract ideas through the use of explicit informational documentation, yet enough documentation (but not so much as to overwhelm) must be explicitly available to successfully generate the implicit documentation notion. This supports the concept of minimal documentation.

C. DOCUMENTATION DEPENDENCIES

Yet another way to categorize documentation is to consider audience-dependent and life cycle phase-dependent documentation divisions. It is necessary to understand the needs of the audience for which the documentation is intended and also the life-cycle phase to which the documentation is related. These determinations are necessary so that the documentation user who is knowledgeable about the system is not completely bogged down by the effort of trying to sort through a myriad of details that have nothing to do with that particular maintenance task, or are superfluous in the sense that the user already knows the necessary details. By the same token, it is inefficient for a person who is not well versed in certain aspects of the project to spend many hours searching through lots of documentation just to find out something specific about the program on which maintenance is to be conducted. There must indeed be a balance between very detailed and high level documentation. The idea is that unnecessary work that adds to the overhead of the maintenance task should not be given to the maintenance person. (More is discussed in Chapter V about how to access the proper level of detail of documentation.)

When considering audience-dependent documentation, several factors must be taken into account. These factors include the reader's skill level (or familiarity with the project), the reader's position relative to the maintenance job, and the particular type of maintenance to be accomplished. A skilled person can be defined as one who understands basic organization programming policies or techniques. The skilled person often possesses the quality of familiarity discussed earlier.

Different kinds of documentation are appropriate for the different factors mentioned above, and in order to run the maintenance job effectively, these kinds of documentation are critical.

1. Skill Level

In considering the idea of skill level for the documentation user, the level of documentation detail should be of concern. The documentation should be of sufficient level so as to give the user the precise amount of detail necessary to carry out the required maintenance function. This means that, if the user is skilled, the provided document should not contain minutely detailed explanations of the program if the ideas are commonly understood. On the other hand, the documentation must be an adequate level of detail so as to provide the unskilled person with the needed amount of system specifics.

The ideas of explicit and implicit documentation come into play here. For the highly skilled user, the amount of physically explicit documentation can be small and condensed in nature. The amount of implicit information would be large because the skilled user can accept high level concepts that do not have to be explicitly described. That is, a highly skilled user can make use of implied notions, such as the notion that the data in a certain program goes through a "sort and eliminate" routine. Here the explicit documentation would consist of the information that the data is sent to the routine, while the implicit information would be made up of commonly understood details of the routine itself.

As for the unskilled user of the documentation, the nature of the explicit and the implicit idea conveyence would be quite different. The unskilled user would need much more explicit information to absorb the same amount of knowledge of the portion of the program to be maintained. The necessary explicit information would likely include many of the specific details of the "sort" and "eliminate" routines separately. If the criteria for elimination of certain data were understood by the unskilled user at this level of detail, then those details would be considered to be implicit information. If those criteria were unknown to the unskilled user, then the explicit documentation concept must move down another level of detail to incorporate these details explicitly. The levels of detail are translated continually from the implicit to the explicit realm as the need for information detail (determined by the skill level) moves down to lower levels.

Conversely, as the skill level of the user increases, information and details required by the user move from the explicit to the implicit realm, thus allowing broader concepts to be absorbed by the user. As more implicit information is required by the user, a corresponding lesser amount of explicit information (or documentation) is required.

The consequences of having less explicit information being required means that less documentation (and consequently less overhead) needs to be sorted through in order to complete the maintenance task. The end result is that as skill level increases, less time is required for the specific maintenance task, and a corresponding maintenance efficiency results.

2. Position

A look at the idea of a person's position with regard to the maintenance effort is useful. In order to better understand how the relative position of the person utilizing the documentation affects documentation needs, first we must know whether the person is considered to be a maintainer, a manager, or a user.

a. Maintainer

The maintainer is defined to be the person who is actively involved in the actual act of maintaining the program. The maintainer requires the lowest level of abstraction of documentation, or the most detailed level of information because of the actual physical maintenance that is accomplished. The type of documentation required by the maintainer to successfully complete the maintenance task then will be on a level that is fairly detailed. The depth of detail required will of course be dependent upon the maintainer's skill level or familiarity as discussed earlier. The documentation type will be of the kind that will promote the detail necessary to complete the job. The amount of explicit documentation will also be determined by skill level and familiarity.

As far as dynamic documentation is concerned, the maintainer relies less heavily on this level of documentation than on the static documentation. The maintainer is very concerned about the present state of the program because the present state is what is to be modified. The decision as to whether or not to make the modification is usually made at a higher level, and thus the dynamic documentation will be better used at that higher level (probably the manager level).

b. Manager

The manager category can be defined so as to include anyone who is directly connected with the maintenance of the system, but not actively involved with the actual physical maintenance of the project. This could include the maintenance team leader in the supervisory role, the department head, or higher level decision makers. The type of documentation that the manager needs should be of a much higher level of abstraction requiring less detail than that required by the maintainer. This means that much more documentation of an implicit nature is acceptable in order to meet the needs of the manager level personnel. The greater the amount of implicit documentation needed, the less the amount of explicit documentation necessary. Likewise, the less the detail level of the explicit documentation, the less the amount of details through which the manager must sort. The smaller amount of detail required leads to the saving of time and money.

The manager could very well be the biggest user of dynamic documentation. The manager at the maintenance group supervisor level is likely to be the one who must look at the way prior decisions were made in order to verify the practicality of requested maintenance enhancements. The manager might want to avoid re-deciding something that is already a given and has been recorded in the dynamic documentation.

The manager could also be a very heavy user of static documentation in the sense that it might be necessary to reference the present status of the maintenance effort in order to properly set up the maintenance team. Thus the manager must rely heavily on the static documentation to understand the program status after design and also to modify the existing documentation as the program is modified. This also implies that dynamic documentation by the maintenance team is occurring along with program modifications, and the manager is responsible for updating or creating the appropriate dynamic documentation.

c. User

The user is anyone who actually uses the system (the pilot using an avionics system, a fire control technician on board ship, etc.) The user might be someone who is considering purchasing services or products from the company and is interested in the stability of the company as a whole. The system is a vital part of the company, and consequently the user might be interested in the program or system from a very abstract point of view. Other than specific user's manuals and system operational guides, the user needs little or no detailed information and can tolerate a large amount of implicit documentation. The amount of explicit documentation required for the user will be very small indeed, perhaps even a simple listing of the systems or programs available to the company. The user simply wants to know what capabilities are present and if they meet the user's needs. Any further detail is superfluous.

The user is probably not interested in any dynamic documentation and has very little interest in static documentation. He does not really care about the design decisions that occurred during the development of the system, but merely about the fact that there exists a system sufficient for his needs.

D. ROLE OF DOCUMENTATION

The idea of implicit and explicit information can be utilized here. Returning for a moment to the example in the last chapter concerning the likening of the receipt of one final piece of documentation to that of a college professor receiving only the answers on a physics or math test without explanation as to how the answers came about, more discussion is in order. The dynamic nature of showing the work, whether the work was on an exam or on a program, helps both the programmer and the maintenance person (or the student and the professor) follow the design and development of ideas. Since documentation, as described earlier, involves the transfer of ideas, it is crucial that these ideas be transferred dynamically in the form of progressive documentation. It is difficult to understand the thought and development process that goes into a problem when all that is seen by the receiver of the information is the final solution.

We need not worry that the final documentation product might contain some recording of our initial erroneous efforts. In fact it might be helpful to the maintainer if the initial trial and error efforts were made available. It could save the maintenance person the redundant effort of trying to rethink the designer's ideas in order to possibly change a previous logical decision.

The complete documentation could also keep the maintainer from overlooking some critical piece of information that would make the newly proposed enhancement an obviously bad move. Another advantage of documenting the creative process is that ideas that were not feasible (technologically or environmentally) at the time of design, and consequently rejected, could be used during the maintenance phase as a result of system requirement changes or technological advances.

Ideally, then, the maintainer will not receive as documentation something as simple as the statement that a system will use red ribbons for printed output. This gives no indication as to how much thought, if any, went into the decision. When the suggestion for an enhancement to allow a different color for printed output, the maintenance person must try to second guess the designer's decision as to why red was selected, and if another color is possible. If this knowledge were made readily available, the maintainer could possibly head off an expensive analysis of colored printed outputs that would discover that the designer already knew that red printouts were the only ones that could be read under the special lighting needed for a security project.

IV. DOCUMENTATION HIERARCHY

The idea of a documentation hierarchy is introduced along with an explanation of the proper use of the hierarchical organization and how it promotes the concept of minimal documentation. System and program documentation are discussed in detail as they relate to both the system hierarchy and the maintenance task.

A. SYSTEM DOCUMENTATION HIERARCHY

One problem that faces the manager is how to wade through all available documentation in order to glean out the pertinent information for the present task without getting bogged down with a massive volume of material. The same problem is faced by the maintainer who may not need to know all the system design information when the task at hand (as determined already by the managerial decision-making process) is simply to modify a small section of code. It is clearly wasteful in this case to force the maintainer to sort through huge amounts of irrelevant material concerning high level system information just to locate information pertaining to the immediate code modification task.

The user requires information about the system on a high conceptual level, but the deluge of unorganized documentation with all levels of detail would require time consuming searching, and most of the detailed information would be utterly useless.

A solution to the problem of unorganized levels of detail in documentation is to construct an organized hierarchical structure for the various forms of documentation based on the level of detail. This documentation hierarchy can provide precisely the appropriate level of detail in the available documentation for the receiver, regardless of whether the receiver is the user, manager, or maintainer. Figure 4.1 provides an example of a system documentation hierarchy organization that indicates the various levels of detail involved in the documentation of a system.

All of the documentation is available to the proper receiver in a concise format that gives the receiver the least amount of detail necessary (thus promoting the concept of minimal documentation as mentioned earlier). As more detail is needed, more explicit documentation is accessible. This promotes understandability and contributes to an efficient maintenance effort.

The system documentation hierarchy of Figure 4.1 shows arrows that indicate a downward and upward flow of documentation access. As progress is made down to lower levels on Figure 4.1, more detail is attained in the documentation. Conversely, as movement is made up the levels, less detailed descriptions and larger concepts are accessed.

B. SYSTEM DOCUMENTATION

To understand more about the system documentation hierarchy, it must be understood what is meant by system documentation. A system can be defined as one or more programs that work in conjunction to perform a particular function. The system can be very simplistic, such as a simple vote counting system, or it can be very complicated as in the case of a sophisticated weapons system. Since a system has been defined as the combination of one or more programs that perform a function, system documentation is defined as the documentation of the overall system life cycle from the project conception phase through the support phase.

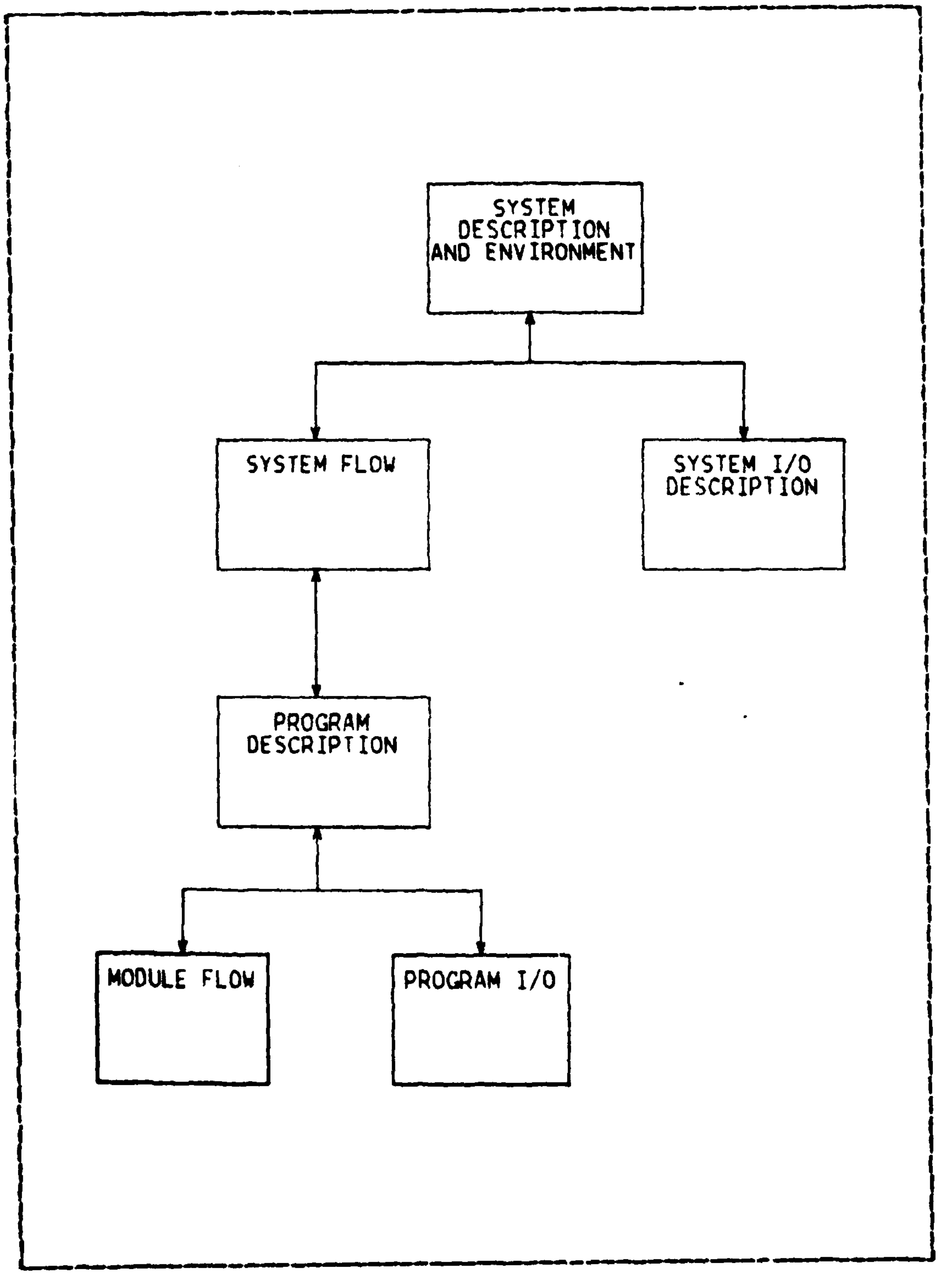


Figure 4.1 System Documentation Hierarchy.

System documentation contains recorded information pertaining to the complete description of the evolution of a system throughout its life cycle. It includes a record of the development process and the maintenance history in either implicit or explicit form. System documentation contains both dynamic and static information and can be used by maintainer, manager, and user personnel. The specific form of documentation is dependent upon the requirements of the task to be accomplished and the needs of the receiver.

Since a system is made up of one or more programs, program documentation is considered to be included as a part of the system documentation. In Figure 4.1 program documentation consists of levels 3 and 4. Both system and program documentation are discussed in greater detail later in this chapter.

1. <u>Level 1</u>

The overall system documentation follows a hierarchical structure with varying levels of detail as shown in Figure 4.1. The highest level of abstraction for the system, level 1, includes a narrative description of the system itself and a description of the system environment along with any assumptions about the system. The environmental description would include information pertaining to the system hardware and software environments. This includes system restrictions and limitations that might result from certain hardware or software constraints under which the system must operate. Military applications such as an avionics system or a submarine weapons system would dictate specific environmental restrictions because of the very nature of the system activities.

It is in this level that the documentation contains the most abstract information about the project.

This level of detail would probably be most often used by the user personnel, but this non-detailed narrative level could also be of use to the manager and maintenance personnel who require an overall understanding of the system.

2. Level 2

Level 2 is the next level of abstraction in the hierarchical structure and contains slightly more detail than level 1. This level includes any system flow information, such as perhaps system flow diagrams, and system input and output descriptions. The inter-program module descriptions are included in this level; this level gives information about how individual programs are inter-related in the system. This type of information can be conveyed with the use of narrative remarks.

Since level 2 system documentation includes information such as system input and output specifications and requirements, maintainers and managers find this input/output information to be valuable because they must ensure that the maintenance of the program is accomplished in such a way that the output requirements are correctly attained when the appropriate system inputs are given.

Managers need to know the system input and output specifications that fit user needs in order to ensure maintainers have the proper information as translated from the user (who very likely is not as technically oriented as the manager or the maintainer) requirements. Thus the managers can take the user input and output requirements as requested by the users and translate them into an understanding between the maintainers and the users in order that effective maintenance can be accomplished.

Users obviously play an important role in the generation of the general input and output specifications, and these specifications must make sense to the managers before the maintainers can be expected to understand and perform maintenance tasks. An understanding between users and managers must therefore be reached as to what the users want (or think they want). The prudent user will heed management advice when considering reasonable input and output formats.

System logic information is also a part of level 2 of the system documentation hierarchy, and it conveys the logical flow of the project. This logical information could include a narrative section that describes the purpose of the system and how it is logically constructed. The hierarchy of programs, functions, and modules can be described in the narrative. The system flow documentation concerning the relationship of the individual modules can take the form of system flow charts or flow diagrams with accompanying comments.

The inter-program module descriptions provide information about the relationship of the programs to the system and to each other. Any restrictive characteristics or program environmental considerations are included in the inter-module narrative.

The rest of the lower levels of abstraction make up the program documentation portion of the documentation hierarchy. It is in these levels that the degree of detail is such that the system is no longer the focal point of the documentation, and the program specifics are brought into view.

C. PROGRAM DOCUMENTATION

Program documentation, as indicated above, consists of the recorded information about the program itself. It is of a more detailed nature than the system documentation and is most useful for the manager and maintainer. It contains information pertaining to program module construction and logic flow. Data structure, data flow, and control flow specifics are recorded so that the documentation receiver has relevant program information available. Programming methodology techniques and maintenance history become part of the program documentation as well. Dynamic documentation describing inter-module concepts and structures is included, but static documentation makes up the bulk of the program documentation. Specific explicit forms of program documentation include flow charts, English narrative statements, resource diagrams, and Petri nets.

While all of the levels in Figure 4.1 represent system documentation, levels 3 and 4 can be combined to make up the program documentation portion of the system documentation hierarchy.

1. Level 3

Level 3 of Figure 4.1 is the first level of the program documentation and conveys a particular program description. Particular program constraints that deal with specific programs are described in Level 3 along with any high level narrative about the program itself. Level 3 does not contain any inter-program relationships with other programs. It is the level that deals with strictly a single program.

This level of abstraction is more detailed than levels 1 and 2, and is very useful to both the manager and the maintainer. The manager needs to keep the high level program concept so that the maintainers can be properly managed without forcing the maintainers to be concerned with any unneccessary abstract information. The manager must keep the program concept in mind and relate it to the rest of the system (level 2 and higher).

The maintainer can, however, use this level of detail to aid in the understanding of how a particular maintenance task is constrained. The manager is responsible for the overseeing of the inter-program module relationships, but a knowledgeable maintenance person can be of immeasurable aid to the wise manager in this area.

2. <u>Level 4</u>

The next lower level of abstraction, level 4, provides the greatest level of detail. This level is used very heavily by maintenance personnel, and often by managerial personnel. This level consists of very detailed descriptions such as program flow information and input/output formats. Much of this documentation is very explicit in nature. It can be static or dynamic. Flowcharts, inter-code comments, logic, and data flow diagrams are included in this level of documentation. It is this level of detail that describes program modules in enough detail so as to promote understandability among maintainers.

While maintainers are the heaviest users of program documentation, and users are the primary users of the higher level system documentation, managers must bridge the gap between the two levels of documentation. Managers are involved with high level decisions that require an overall system understanding, yet they must also be involved with some of the lower levels of program documentation in order to properly manage the maintenance functions.

D. DOCUMENTATION HIERARCHY UTILIZATION

The documentation hierarchy is set up so that anyone can access the hierarchy at any of the indicated levels, and thus be exposed to the level of detail characteristic of that particular level. If more detail is needed for a given task, then a simple move down to the next level for greater detail is permitted. By the same token if it is determined that the level accessed is too detailed for the particular needs of the person using the documentation, then the arrow is simply followed up to a higher level of abstraction that meets the desired needs.

Each form of documentation, then, is catalogued as to its detail level, and a menu format (either paper or electronic) can be utilized to directly access the level needed. With the capability of moving either direction in the hierarchy structure, great flexability is built into the system, and only the exact amount of documentation needed is accessed. This promotes the minimal documentation concept and, therefore, keeps the documentation overhead down to a minimum. The amount of useless information that must be waded through in order to find the proper documentation is kept low as a result of proper utilization of the documentation hierarchy concept.

V. EVALUATION OF FORMS OF DOCUMENTATION

Chapter III discusses the various types of documentation and how they relate to the maintenance effort. It is important to carry the documentation discussion further and talk about not only the types of documentation that are useful, but also some specifics as far as physical arrangements are concerned. The discussion will focus on explicit types of program documentation, and how some of the physical characteristics of the documentation affect the efficiency of maintenance performance.

A. EVALUATION EXPERIMENTS

When dealing specifically with programming documentation (levels 3 and 4 of Figure 4.1) which is most often used by maintainers, it would be helpful to understand which different forms of documentation are most effective. Chapter III discusses the different types of information utilized by users, managers, and maintainers, depending on the maintenance task and the documentation receiver. This chapter discusses some specific forms of documentation and how effective they can be in promoting understandability for efficient program maintenance.

It has been determined by General Electric studies [Ref. 10] that the best form of documentation to be used for maintenance is heavily dependent on the type of program processing that takes place, in particular whether it is sequential or concurrent processing.

1. Documentation for Sequential Processing

In determining the most effective type of documentation format for sequential processing, a primary concern must involve the type of symbology used to present the information. It would be beneficial to ascertain the best form of symbology as seen by the maintenance personnel in terms of maintenance efficiency.

The three symbology types used in the General Electric Studies consist of narrative Englist text, an abbreviated program-like language called Program Design Language (PDL), and ideograms. The narrative text is frequently embedded in the source code as either global or in-line comments. The PDL is succinct and uses strictly defined keywords to describe arguments or predicates. Ideograms are often found in flow charts and HIPO charts. Sets of ideograms represent processes in a program [Ref. 11].

Another primary concern which must be dealt with when weighing effective documentation is the issue of spatial arrangement. Spatial arrangements can aid maintainers in understanding the flow of control in a sequential program, and it would be helpful if the best spatial format could be determined. The spatial arrangements provide different ways of representing control flow and nesting levels. The spatial arrangements used in the experiments are sequential, branching, and hierarchical representations.

The sequential arrangement represents both the control flow and the levels of nesting in a vertical manner. The branching arrangement presents the flow of control in a vertical manner while the nesting levels are presented horizontally. Finally, in the hierarchical arrangement, the control flow is represented horizontally and the nesting levels are presented vertically.

The sequential processing experiments were designed to run the gamut of many of the maintenance tasks performed by programmers. The tasks included answering questions about program coding, program debugging, program modification, and program operation. The maintenance tasks were to be completed using the various forms of documentation being tested. The studies were conducted with professional programmers who were asked to answer questions about programs. The programmers were allowed to reference only the various forms of documentation having the spatial and symbology characteristics mentioned above to get information about the programs.

Nine specification formats were presented to the programmers for their use in the experiments. Each of the three types of symbology was presented in each of the three spatial arrangements.

The participants were also asked to choose which format of documentation they found to be the easiest to use. This choice was then weighed against the type of documentation that produced the best results in terms of maintenance effectiveness.

In the first experiment, the programmers were asked to answer backward and forward-tracing questions and input/output questions about the program using the test documentation provided.

The results showed that the sequential PDL, the branching PDL, and the branching ideogram versions of documentation were the most effective for answering the tracing questions.

For the input/output questions, no significant differences were found between the forms of documentation. The most preferred combinations of documentation formats were the PDL symbology and the branching spatial arrangements [Ref. 11].

In another experiment programmers were asked to complete the coding of portions of programs referencing only the documentation under test. In this experiment the English narrative format took significantly longer to produce code than did the PDL format. The English version also produced the largest number of errors, while the PDL produced the smallest.

The spatial arrangement effects were not significant, but the formats of the sequential PDL and the branching PDL arrangements produced the best experimental results. The sequential English version produced the poorest performance.

The programmers also chose the PDL branching arrangements as the preferred format combination.

In yet another experiment the programmers had to correct error-seeded programs, again utilizing only the documentation under test as a source of program information.

The best results in performance occurred with the PDL and ideogram symbologies for this experiment. The spatial effects were again not significant. The sequential and branching PDL formats proved to be high performers, as did the branching and hierarchical ideograms.

The programmers had no preference for the type of symbology in this experiment, but they did prefer the branching spatial arrangement [Ref. 13].

Though slightly different results were produced in this experiment depending on the maintenance task, overall the indication is that performance is improved when the symbology is of a succinct nature, such as in the PDL format. The English narrative proved to be too wordy and awkward to provide efficiency when attempting software maintenance.

As for the spatial arrangement issue, the best overall performance resulted from the use of a branching arrangement in providing the clearest display of control flow. The PDL branching format, then, seemed to promote understandability for the maintainer, and the PDL branching format was selected by the programmers as the easiest overall format to use.

2. <u>Documentation for Concurrent Processing</u>

Since much of today's program processing is concurrent, it is wise to investigate documentation effectiveness for the concurrent realm of processing. Concurrent processing of programs entails two or more portions of the program executing simultaneously. Because of the complexity involved with concurrent processes, programs that contain concurrent processing must be carefully documented. It is important to convey information about the control flow of the program and the sharing of resources.

The formats of documentation used for the General Electric studies of concurrent processing documentation [Ref. 14] consist of three types: PDL; resource diagrams; and Petri nets. The first form of documentation is the same PDL as used in the sequential processing tests. The PDL emphasizes the control-flow characteristics of the program.

The second form of documentation, the resource diagram, places emphasis on the concept of providing resource sharing information to the programmer. The resource diagram uses communication circles containing abbreviated English statements to convey information about the relationships between processes. Natural English statements provide narrative information contained in process boxes to describe the process itself. Resource diagrams are arranged spatially in a branching format similar to the branching organization used in the sequential experiments.

The third form of documentation is that of a Petri net. Petri nets have nodes that contain information that indicates resource usage for required tasks, while control-flow information is conveyed with a constrained language description. The Petri net format of documentation places equal emphasis on control-flow and resource sharing information. The spatial arrangement of the Petri net is also similar to that of a branching organization.

In the concurrent processing experiment programmers were asked to make either data-structure or control-flow modifications to each of three programs. For both types of modifications, the resource diagrams proved to be the best performers. The Petri net gave the poorest performance.

Since the resource diagrams emphasize information about the resource-sharing aspect of the processing of the program, it is interesting to note that the control-flow information that was so important for the sequential processing of a program is not as vital for the maintenance of concurrent processes.

When asked to select the documentation format that was easiest to use, the PDL format was selected. It turned out, however, that the most efficient form of documentation for the concurrent processing was the resource diagram.

B. DISCUSSION

The results of the experiments yield some ideas that can be incorporated into explicit documentation types for program documentation. With proper incorporation of the ideas, understandability can be enhanced for maintainers resulting in a positive influence on maintenance efficiency.

When determining the type of documentation to be accessed in the documentation hierarchy of Figure 4.1 in Chapter III, it is important to realize that there is not one "best" form of documentation for all maintenance tasks. The type of processing (sequential or concurrent) must be taken into account when identifying the best documentation format to include in the hierarchy. This processing information is provided in a narrative sense in level 3. Level 4 will provide the actual flow information, be it resource-flow or control-flow information.

The General Electric studies show support for the concept of minimal documentation introduced in chapter II. The English narratives were found to be too long and awkward for the best performance of maintenance. When the method of transferring information took on the more abbreviated form of the PDL, maintainers showed a preference for this format of symbology presentation. This preference held true for both the sequential and the concurrent programming techniques. The implication is that, even though the ideas conveyed in both the formal English narrative and the PDL were the same, the programmers chose the succinct method of symbology as being easier to glean the necessary information for the maintenance task. A significant point is that the programmers chose not to wade through all the superfluous language provided by the English narrative, thus indicating a preference for minimal documentation. As far as sequential processing is concerned, the PDL proved to be not only the programmer's choice for symbology representation, it also proved to be the most efficient. In the case of the concurrent processing, the PDL was the preferred method of symbology representation, but the resource diagram proved to be more efficient.

The concept of minimal documentation is not contradicted by the fact that the PDL form of symbology was preferred by programmers, but resource diagrams proved to be the most efficient for maintenance purposes in concurrent programming. The fact is that the information required for concurrent processing maintenance is simply different than the information that is provided by the PDL. Concurrent processing requires information with emphasis on the resource-sharing aspect of the program, while the PDL provides information primarily concerning the aspect of control-flow (which is of primary concern in the sequential processing program). In this experiment it turned out that the actual minimal documentation was the resource diagram, and not the PDL the maintainers preferred.

When determining which format of documentation to access for the performance of maintenance, the format which best suits the task at hand should be considered in the selection process with emphasis on maintenance efficiency. When the proper level (or levels) of documentation are selected from the documentation hierarchy, along with the best physical representation of the documentation, then minimal documentation is accessed and effective understandability is achieved. The end result is an effective and efficient performance of the maintenance task.

VI. <u>CONCLUSIONS AND RECOMMENDATIONS</u>

Since maintenance costs make up the largest part of most software projects, it is vital to find effective ways to reduce or make more efficient the software maintenance effort. When good documentation techniques are incorporated into the project evolution, then development ideas and other relevart information about the system can be successfully recorded and transferred to other individuals.

Since it is critical that good documentation techniques be emphasized, accurately determining the precise type and amount of documentation for software maintenance is vital. Minimal documentation is the result of that determination and should, therefore, be incorporated into software projects where appropriate. (Some programs are simply not maintained and therefore do not need maintenance oriented documentation.)

Managers of the maintenance team often have misconceptions about how the time spent on software maintenance should be allocated. Because of these misconceptions, a closer look at how maintenance time is spent is in order. Perhaps an analysis of the maintenance effort on each project should be conducted so as to determine how the maintenance time is actually spent. The manager can then have an effective tool with which to schedule the maintenance effort without having to resort exclusively to the use of intuition.

Programmers should be trained not only to document the system as it develops, but to do so keeping the maintenance aspect in mind. Maintenance enhancing documentation should be developed simultaneously with the project as an integral part of the system.

Programmers must become aware of the fact that there is not one "best" format of documentation for all types of maintenance. More research like the General Electric studies should be conducted in order to determine the best documentation format for the particular maintenance task being performed. A particular format, then, should not be taught as the only proper way to document a program.

Well trained programmers will also raise the skill level of the maintenance team, and as skill level increases, the need for detailed explicit documentation decreases. The skilled programmer can then accept larger conceptual ideas about the program, thus avoiding the need to search through a large volume of information in order to perform the task at hand. Maintenance and cost efficiencies are therefore enhanced.

Since programmers have more confidence in internal documentation, it is recommended that, to the extent feasible, information be carried internally along with the source code. As "hard" copies are needed, they could simply be printed out for a specific use. Perhaps a physical copy of the documentation should be filed for back up purposes, but the amount of external copies should be kept to a minimum in order to avoid the reluctance to keep the hard copies updated. In all cases, however, all forms of documentation should be updated as modifications to the software are made in order to ensure that the documentation is an accurate reflection of the project.

In support of achieving minimal documentation, the internally stored documentation should be organized in the format of a documentation hierarchy. There should be one hierarchy structure that will contain all types of explicit documentation, and each physical format will be classified and filed according to the level of detail contained in the document. This "level of detail" type of categorization will necessarily cause the documentation to become a part of either the system or the program documentation.

Users, managers, and maintainers should be able to access the appropriate piece of documentation based on the amount of detail needed for the particular task at hand. The system should be set up in such a way that each level is easily accessed, and a method of moving up or down the hierarchical organization should be made available.

It is recommended that further research be conducted into the implementation of the hierarchical scheme in a menu driven window format that can display the indicated piece of documentation on a display screen for perusal. A pointer device can point to a place on the menu to request a particular level in the hierarchy. The capability to transcend to different levels will be built into the menu operation of the windows. This documentation hierarchy implementation will provide a powerful documentation tool that promotes the minimal documentation concept, and should result in an efficient maintenance effort.

LIST OF REFERENCES

1. Boehm, B.W. Software Engineering, IEEE Transactions, Computers, December 1976, pp 1266-1241

2. Martin, J., and McClure, C. Software Maintenance, the Problem and its Solutions, Prentice-Hall, Inc., 1983, pp 43-73

3. Boehm, B.W. Software Engineering Economics, Prentice-Hall, Inc., 1981

4. Fleckenstein, W.O. Challenges in Software Development Bell Laboratories, Computer, March, 1983, pp 60-64

5. Lientz, B.P., and Swanson, E.B. Software Maintenance Management, Addison-Wesley Publishing Company, 1980

6. Swanson, E.B. The Dimensions of Maintenance, 2nd International Conference on Software Engineering, Proceedings, October 13-15, 1976, pp 496-497

7. Lientz, B.P., Swanson, E.B., and Tompkins, G. Characteristics of Applicative Software Maintenance, Communications of the ACM, Vol. 21, June, 1978, pp 466-471

8. Schneidewind, N.F. Software Maintenance: Improvement Through Better Development Standards and Documentation, Naval Postgraduate School, February, 1982

9. Glass, R.L., and Noiseux, R.A. Software Maintenance Guidebook, Prentice-Hall, Inc., 1981, pp 1-9

10. Bohem-Davis, D.A. Representation of Information in Software Documentation General Electric Company, GEC/DIS/TR-83-388200-8, July, 1983

11. Sheppard, S.B., Kruesi, E., and Curtis, B. The Effects of Symbology and Spatial Arrangement on the Comprehension of Software Specifications, General Electric Company, GEC/DIS/TR-80-388200-2, October, 1980

12. Sheppard, S.B., and Kruesi, E. The Effects of the Symbology and Spatial Arrangement of Software Specifications in a Coding Task, General Electric Company, GEC/DIS/TR-81-388200-3, February, 1981

13. Sheppard, S.B., Bailey, J.W., and Kruesi, E. The Effects of the Symbology and Spatial Arrangement of Software Specifications in a Debugging Task. General Electric Company, GEC/DIS/TR-81-388200-4, August, 1981

14. Bohem-Davis, D.A., Fregly, A.M. Documentation of Concurrent Programs, General Electric Company. GEC/DIS/TR-83-388200-7, July, 1983

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.	Department Chairman, Code 52 Department of Computer Science Monterey, California 93943	1
4.	LT John F. Hall, II 401 South High Street Mount Orab, Ohio 45154	2
5.	Dr. Gordon Bradley Code 52BZ Department of Computer Science Naval Postgraduate School Monterey, California 93943	2
6.	LCDR Paul W. Callahan Code 52CS Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
7.	Patricia E. Roesner 231 Via Gayuba Monterey, Californea 93940	1

LMED

-8