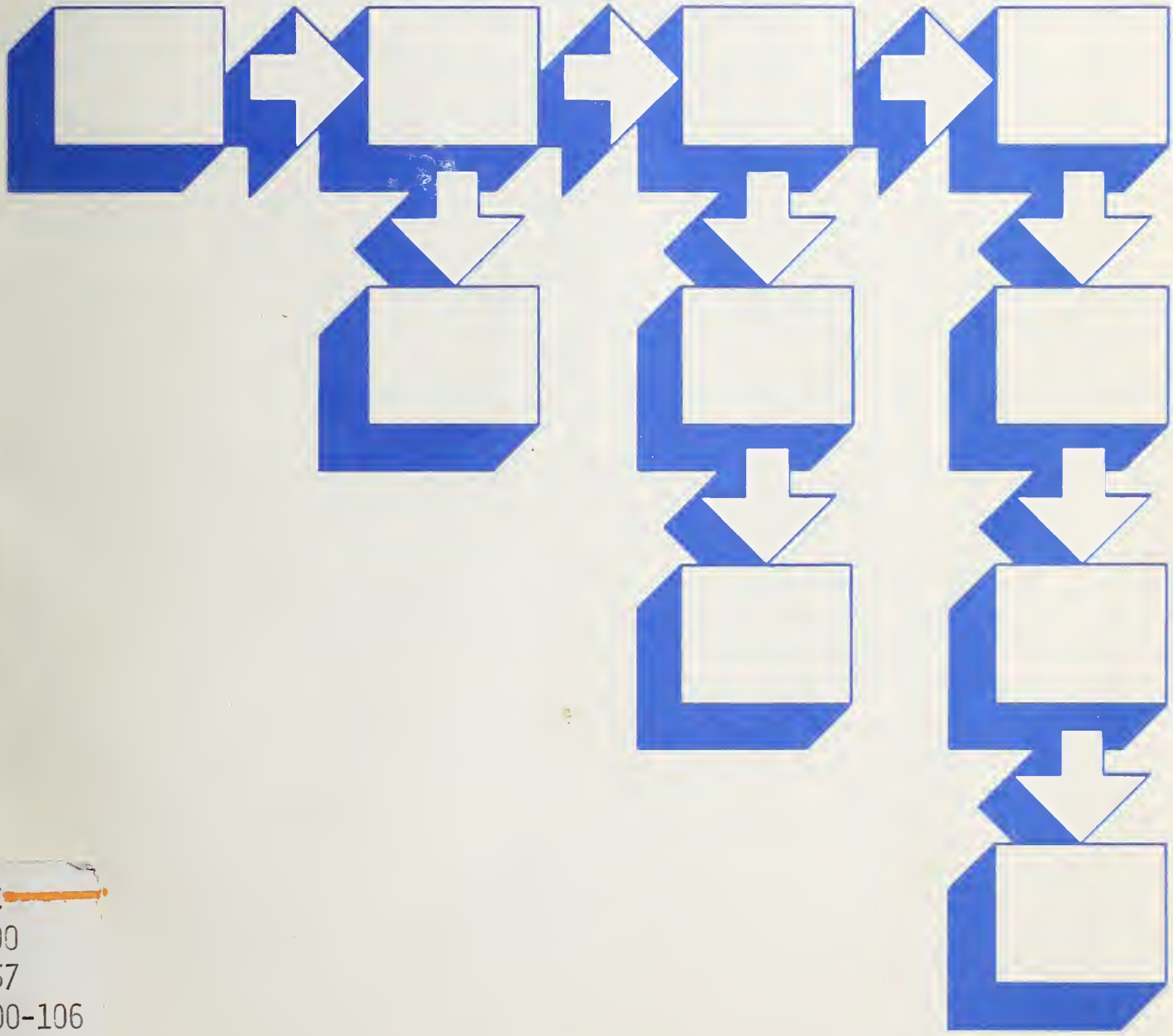
U.S. Department
of Commerce

National Bureau
of Standards

Computer Science and Technology

NBS Special Publication 500-106

Guidance on Software Maintenance



NATIONAL BUREAU OF STANDARDS

The National Bureau of Standards[1] was established by an act of Congress on March 3, 1901. The Bureau's overall goal is to strengthen and advance the Nation's science and technology and facilitate their effective application for public benefit. To this end, the Bureau conducts research and provides: (1) a basis for the Nation's physical measurement system, (2) scientific and technological services for industry and government, (3) a technical basis for equity in trade, and (4) technical services to promote public safety. The Bureau's technical work is performed by the National Measurement Laboratory, the National Engineering Laboratory, and the Institute for Computer Sciences and Technology.

THE NATIONAL MEASUREMENT LABORATORY provides the national system of physical and chemical and materials measurement; coordinates the system with measurement systems of other nations and furnishes essential services leading to accurate and uniform physical and chemical measurement throughout the Nation's scientific community, industry, and commerce; conducts materials research leading to improved methods of measurement, standards, and data on the properties of materials needed by industry, commerce, educational institutions, and Government; provides advisory and research services to other Government agencies; develops, produces, and distributes Standard Reference Materials; and provides calibration services. The Laboratory consists of the following centers:

Absolute Physical Quantities[2] — Radiation Research — Chemical Physics — Analytical Chemistry — Materials Science

THE NATIONAL ENGINEERING LABORATORY provides technology and technical services to the public and private sectors to address national needs and to solve national problems; conducts research in engineering and applied science in support of these efforts; builds and maintains competence in the necessary disciplines required to carry out this research and technical service; develops engineering data and measurement capabilities; provides engineering measurement traceability services; develops test methods and proposes engineering standards and code changes; develops and proposes new engineering practices; and develops and improves mechanisms to transfer results of its research to the ultimate user. The Laboratory consists of the following centers:

Applied Mathematics — Electronics and Electrical Engineering[2] — Manufacturing Engineering — Building Technology — Fire Research — Chemical Engineering[2]

THE INSTITUTE FOR COMPUTER SCIENCES AND TECHNOLOGY conducts research and provides scientific and technical services to aid Federal agencies in the selection, acquisition, application, and use of computer technology to improve effectiveness and economy in Government operations in accordance with Public Law 89-306 (40 U.S.C. 759), relevant Executive Orders, and other directives; carries out this mission by managing the Federal Information Processing Standards Program, developing Federal ADP standards guidelines, and managing Federal participation in ADP voluntary standardization activities; provides scientific and technological advisory services and assistance to Federal agencies; and provides the technical foundation for computer-related policies of the Federal Government. The Institute consists of the following centers:

Programming Science and Technology — Computer Systems Engineering.

[1]Headquarters and Laboratories at Gaithersburg, MD, unless otherwise noted; mailing address Washington, DC 20234.
[2]Some divisions within the center are located at Boulder, CO 80303.

Computer Science and Technology

NBS Special Publication 500-106

Guidance on Software Maintenance

Roger J. Martin and Wilma M. Osborne

Center for Programming Science and Technology
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, DC 20234

U.S. DEPARTMENT OF COMMERCE
Malcolm Baldrige, Secretary

National Bureau of Standards
Ernest Ambler, Director

Issued December 1983

Reports on Computer Science and Technology

The National Bureau of Standards has a special responsibility within the Federal Government for computer science and technology activities. The programs of the NBS Institute for Computer Sciences and Technology are designed to provide ADP standards, guidelines, and technical advisory services to improve the effectiveness of computer utilization in the Federal sector, and to perform appropriate research and development efforts as foundation for such activities and programs. This publication series will report these NBS efforts to the Federal computer community as well as to interested specialists in the academic and private sectors. Those wishing to receive notices of publications in this series should complete and return the form at the end of this publication.

National Bureau of Standards Special Publication 500-106
Natl. Bur. Stand. (U.S.), Spec. Publ. 500-106, 74 pages (Dec. 1983)
CODEN: XNBSAV

Library of Congress Catalog Card Number: 83-600611

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1983

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, DC 20402
Price
(Add 25 percent for other than U.S. mailing)

TABLE OF CONTENTS

LIST OF TABLES

Guidance On Software Maintenance

Roger J. Martin and Wilma M. Osborne

This report addresses issues and problems of software maintenance and suggests actions and procedures which can help software maintenance organizations meet the growing demands of maintaining existing systems. The report establishes a working definition for software maintenance and presents an overview of current problems and issues in that area. Tools and techniques that may be used to improve the control of software maintenance activities and the productivity of a software maintenance organization are discussed. Emphasis is placed on the need for strong, effective technical management control of the software maintenance process.

Key words: adaptive maintenance; corrective maintenance; management; perfective maintenance; software engineering; software maintenance; software maintenance management; software maintenance tools.

1.0 BACKGROUND

The Institute for Computer Sciences and Technology (ICST), within the National Bureau of Standards (NBS), has a responsibility under Public Law 89-306 (Brooks Act) to promote cost effective selection, acquisition, and utilization of automatic data processing resources within the Federal Government. ICST efforts include research in computer science and technology, direct technical assistance, and the development of standards and guidelines for data processing equipment, practices, and software. As part of this responsibility and the growing need to improve software maintenance methods and management, the ICST is developing software maintenance guidance designed to assist Federal agencies in the ongoing support of existing computer systems. While software systems vary in function, type, and size, many of the functions performed under software maintenance are universal in scope and the activities required to keep them operational are generally the same. This is the first in a series of reports which address both the management and technical practices, procedures, and methods for software maintenance.

This report provides general guidance for managing software maintenance efforts. It presents an overview of the various aspects and problems of software maintenance, and identifies those techniques and procedures designed to assist management in controlling and performing software maintenance. It addresses the need for a maintenance policy with enforceable controls for use throughout the software life cycle. The underlying theme is that improvements in the area of software maintenance will come primarily as a result of the software maintenance policies, standards, procedures, and techniques instituted and enforced by management.

1.1 Introduction

There is a growing interest in software maintenance as evidenced by the number of articles, reports, and textbooks on the subject (see Bibliography). This interest has been spurred by estimates that more resources are required to maintain existing systems than to develop new ones. Federal managers responsible for software application systems estimate that 60% to 70% of the total application software resources are spent on software maintenance [GA081a] [GA080].

Two of the major causes of this software maintenance burden are the growth of the inventory of software which must be maintained and the failure to adopt and utilize improved technical and management methods and tools. The issue which

must be addressed is not one of reducing the absolute cost of software maintenance, but rather improving the quality and effectiveness of software maintenance and thus, reducing the relative or incremental costs.

In order to improve the quality and effectiveness, it is necessary to not only improve software maintenance techniques, methodologies, and tools, but to also improve the management of software maintenance. This Guide discusses the problems associated with managing software maintenance and software maintainers, and examines management methods which can reduce those problems.

Informal discussions were held with selected Federal agencies and private sector organizations to gain a better understanding of the current state of software maintenance. These discussions provided background information on current practices, procedures, and policies relating to software maintenance. This information, along with additional research, is the basis for this report.

The major topic areas addressed in these discussions were:

1. Definition of software maintenance.
2. Methods and techniques in coordinating and performing software maintenance.
3. Major maintenance problems.
4. Types of applications being maintained.
5. Developmental history of existing software.
6. Maintenance staff profiles.
7. Management of maintenance activities.
8. Utilization of maintenance tools.

It was expected that there would be some commonality in the information provided by these discussions. In fact, while each organization has problems peculiar to its environment, there was an extremely high degree of consistency in the comments made and the problems cited.

The primary difficulties and deficiencies encountered in software maintenance fall into several categories: software quality, environment, management, users, and personnel. Specific problems which were consistently mentioned are listed in Table 1.

Table 1 - Software Maintenance Problems

Software	- program quality
	  - software design
	  - software coding
	  - software documentation
	  - programming languages used
	- lack of common data definitions
	- increasing inventory
	- excessive resource requirements

Environment	- growth - evolving/change - new hardware
Management	- maintenance controls - maintenance techniques and procedures - maintenance tool usage - standards enforcement
Users	- demanding more capabilities
Personnel	- lack of experience - image/morale problems - view of maintenance: unchallenging,unrewarding

As can be seen from the table, there are both technical and management problems. It appears, however, that many of the technical problems are often the result of inadequate management control over the software maintenance process. These problems arise for at least two different reasons. First of all, there is a great deal of code which was not developed with maintenance in mind. Indeed, the emphasis has often been to get the program up and running without being "hindered" by guidelines, methodologies, or other controls. The second reason is, that over the life cycle of a software system, the code and logic which may have been well-designed and implemented often deteriorate due to an endless succession of "quick fixes" and patches which are neither well-designed nor well-documented. Thus, in today's vast inventory of application systems, there are many programs which at the time of their development were considered "state-of-the-art," but today are, in fact, virtually unmaintainable.

The need to maintain old, outdated, poorly documented systems was consistently cited as a primary problem in software maintenance. There appears to have been some improvement in the quality of software over the last four to five years. These improvements, however, have come mainly on an individual basis where a programmer, analyst, or line manager has introduced one or more modern programming practices (e.g. structured code, top-down design and development, peer review). There usually has not been a systematic adoption of these practices at a higher level within an agency. Nor has there been extensive institutional introduction of standards and guidelines for software development and maintenance.

Sections 1.0 through 6.0 address the definitions and problems of software maintenance. These sections present an overview of the software maintenance process and discuss the primary technical and management software maintenance issues. Sections 7.0 through 10.0 address how to control and improve software maintenance through the adoption or use of various policies, techniques, and tools.

2.0 DEFINITION OF SOFTWARE MAINTENANCE

Software maintenance is a commonly "understood" term for which there is no single definition. This lack of a standard definition often results in confusion for those attempting to address the problems of software maintenance. Some examples of software maintenance definitions are included in Appendix I. The following definition of software maintenance is used throughout this report.

> Software maintenance is the performance of those activities required to keep a software system operational and responsive after it is accepted and placed into production.

Software maintenance then, is the set of activities which result in changes to the originally accepted (baseline) product set. These changes consist of modifications created by correcting, inserting, deleting, extending, and enhancing the baseline system. Generally, these changes are made in order to keep the system functioning in an evolving, expanding user and operational environment.

2.1 Functional Definition

Functionally, software maintenance activities can be divided into three categories which were originally proposed by Swanson[SWAN76]: perfective, adaptive, and corrective.

Many software managers consider requirements specification changes and the addition of new capabilities to be software maintenance. Although these areas were not addressed by Swanson, the definition of perfective maintenance has been expanded to include them. The three maintenance categories are defined in the following manner:

Perfective maintenance includes all changes, insertions, deletions, modifications, extensions, and enhancements which are made to a system to meet the evolving and/or expanding needs of the user.

Adaptive maintenance consists of any effort which is initiated as a result of changes in the environment in which a software system must operate.

Corrective maintenance refers to changes necessitated by actual errors (induced or residual "bugs") in a system.

Table 2 - Functional Definition of Software Maintenance

Perfective	changes, insertions, deletions, modifications, extensions, and enhancements
Adaptive	adapting the system to changes in the environment
Corrective	fixing errors

2.1.1 Perfective maintenance

Perfective maintenance refers to enhancements made to improve software performance, maintainability, or understandability. It is generally performed as a result of new or changing requirements, or in an attempt to augment or fine tune the software. Activities designed to make the code easier to understand and to work with, such as restructuring or documentation updates (often referred to as "preventive" maintenance) are considered to be perfective. Optimization of code to make it run faster or use storage more efficiently is also included in the perfective category. Estimates indicate that perfective maintenance comprises approximately 60%-70% of all software maintenance efforts.

Perfective maintenance is required as a result of both the failures and successes of the original system. If the system works well, the user will want additional features and capabilities. If the system works poorly, it must be fixed. As requirements change and the user becomes more sophisticated, there will be changes requested to make functions easier and/or clearer to use. Perfective maintenance is the method usually employed to keep the system "up-to-date", responsive and germane to the mission of the organization.

There is some disagreement whether the addition of new capabilities should be considered maintenance or additional development. Since it is an expansion of the existing system after it has been placed into operation, and is usually performed by the same staff responsible for other forms of

maintenance, it is appropriately classified as maintenance.

Fine tuning existing systems to eliminate shortcomings and inefficiencies and to optimize the process is often referred to as "preventive maintenance". It can have dramatic effects on old, poorly written systems both in terms of reducing resource requirements, and in making the system more maintainable and thus, easier to change or enhance. Preventive maintenance may also include the study and examination of a system prior to occurrence of errors or problems. Fine tuning is an excellent vehicle for introducing the programmer to the code, while at the same time reducing the likelihood of serious errors in the future.

2.1.2 Adaptive maintenance

Adaptive maintenance refers to modifications made to a system to satisfy or accomodate changes in the processing environment. These environmental changes are normally beyond the control of the software maintainer and consist primarily of changes to the:

- rules, laws, and regulations that affect the system
- hardware configurations, e.g., new terminals, local printers
- data formats, file structures
- system software, e.g., operating systems, compilers, utilities.

Changes to rules, laws and regulations often require the performance of adaptive maintenance on a system. These changes must often be completed in a very short time frame in order to meet the dates established by the laws and regulations. If rules and their actions are implemented modularly, the changes are relatively easy to install. Otherwise, they can be a nightmare.

Changes to the computer hardware (new terminals, local printers, etc.) which support the application system are usually performed to take advantage of new and/or improved features which will benefit the user. They are normally performed on a scheduled basis. The usual goal of this maintenance is to improve the operation and response of the application system.

Changes to data formats and file structures may require extensive maintenance on a system if it was not properly designed and implemented. If reading or writing of data is isolated in specific modules, changes may have less impact. If it is embedded throughout the code, the effort can become very lengthy and costly.

Changes to operating system software (compilers, utilities, etc.) can have varying effects on the existing application systems. These effects can range from requiring little or no reprogramming, to simply recompiling all of the source code, to rewriting code which contains non-supported features of a language that are no longer available under the new software.

Maintenance resulting from changes in the requirements specifications by the user, however, is considered to be perfective, not adaptive, maintenance.

2.1.3 Corrective maintenance

Corrective maintenance consists of activities normally considered to be error correction required to keep the system operational. By its nature, corrective maintenance is usually a reactive process where an error must be fixed immediately. Not all corrective maintenance is performed in this immediate response mode; but all corrective maintenance is related to the system not performing as originally intended.

There are three main causes which require systems to undergo corrective maintenance:

1. Design errors
2. Logic errors
3. Coding errors

Design errors are generally the result of incomplete or faulty design. When a user gives incorrect, incomplete, or unclear descriptions of the system being requested, or when the analyst/designer does not fully understand what the user is requesting, the resulting system will often contain design errors.

Logic errors are the result of invalid tests and conclusions, faulty logic flow, incorrect implementation of the design specifications, etc. Logic errors are usually attributable to the designer or previous maintainer. Often, the logic error occurs when unique or unusual combinations of data, which were not tested during the development or previous maintenance phases, are encountered.

Coding errors are the result of either incorrect implementation of the detailed logic design, or the incorrect use of the source code. These errors are caused by the programmer. They are usually errors of negligence or carelessness and are the most inexcusable, but usually the easiest to fix.

3.0 THE SOFTWARE MAINTENANCE PROCESS

The life cycle of computer software covers its existence from its conception until the time it is no longer available for use. There are a number of definitions of the software life cycle which differ primarily in the categorization of activities or phases. One traditional definition is: requirements, design, implementation, testing, and operation and maintenance.

The requirements phase encompasses problem definition and analysis, statement of project objectives, preliminary system analysis, functional specification, and design constraints. The design phase includes the generation of software component definition, data definition, and interfaces which are then verified against the requirements. The implementation phase entails program code generation, unit tests, and documentation. During the test phase, system integration of software components and system acceptance tests are performed against the requirements. The operations and maintenance phase covers the use and maintenance of the system. The beginning of the maintenance phase of the life cycle is usually at the delivery and user acceptance of the software product set.

Table 3 - Software Maintenance Process

---

1. Determination of need for change
2. Submission of change request
3. Requirements analysis
4. Approval/rejection of change request
5. Scheduling of task
6. Design analysis
7. Design review
8. Code changes and debugging
9. Review of proposed code changes
10. Testing
11. Update documentation
12. Standards audit
13. User acceptance
14. Post installation review of changes and their impact on the system
15. Completion of task

The process of implementing a change to a production system is complex and involves many people in addition to the maintainer. Table 3 outlines the software maintenance process. This process begins when the need for a change arises and ends after the user has accepted the modified system and all documentation has been satisfactorily updated.

Although the process is presented in a linear fashion, there are a number of steps where iterative loops often occur. The change request may be returned to the user for additional clarification; the results of the design review may necessitate additional design analysis or even modification of the change request; testing may result in additional design changes or recoding; the standards audit may require changes to the design documents, code, and/or documentation; and the failure of the users to accept the system may result in return to a previous step or the cancellation of the task.

One way of describing the activities of software maintenance is to identify them as successive iterations of the first four phases of the software life cycle, i.e. requirements, design, implementation, and testing. Software maintenance involves many of the same activities associated with software development with unique characteristics of its own, some of which are discussed in the following paragraphs.

Maintenance activities are performed within the context of an existing framework or system. The maintainer must make changes within the existing design and code structure constraints. This is often the most challenging problem for maintenance personnel. The older the system, the more challenging and time-consuming the maintenance effort becomes.

A software maintenance effort is typically performed within a much shorter time frame than a development effort. A software development effort may span one, two, or more years while corrective maintenance may be required within hours and perfective maintenance in cycles of one to six months.

Development efforts must create all of the test data from scratch. Maintenance efforts typically take advantage of existing test data and perform regression tests. The major challenge for the maintainer is to create new data to adequately test the changes to the system and their impact on the rest of the system.

4.0 SOFTWARE MAINTENANCE PROBLEMS

The responses to the ICST survey of selected Federal and private sector ADP organizations consistently cited a common set of software maintenance problems. Generally, these problems can be categorized as technical and management. Most of these problems, however, can be traced to inadequate management control of the software maintenance process. This section presents an overview of the technical aspects of the maintenance problems identified in the survey. Management control issues are addressed in subsequent sections of this report.

4.1 Software Quality

Modern programming practices, which utilize a well-defined, well-structured methodology in the design and implementation of a software system, address at least one major software maintenance problem - poor program quality. The importance of these methodologies, whether they are formal or informal is to give structure and discipline to the process of developing and maintaining software systems. While this may alleviate some of the software maintenance problems for systems developed using these methodologies, it does not solve the problem of existing systems which were designed, developed, and maintained without utilizing a disciplined structure.

A lack of attention to software quality during the design and development phases generally leads to excessive software maintenance costs. It should be clearly understood during the design and development phases that the maintainability of the system is directly affected by the quality of the software.

4.1.1 Poor software design

The design specifications of a software system are vital to its correct development and implementation. Poor software design can be attributed to:

- a lack of understanding by the designer of what the user requested.
- poor interpretation of the design specifications by the developers.
- the use of convoluted and complex logic to meet a requirement.
- disjointed segments which do not fit together into a nicely integrated whole.
- a lack of discipline in design which results in inconsistent logic.
- large, unmodular systems (or worse yet one large system

with no component segments) which are bulky, unwieldy, and very difficult to understand.

4.1.2 Poorly coded software

A great deal of existing software contains poorly written code. As computer programming evolved, much of the code development was performed in an undisciplined, unstructured manner. This resulted in a great deal of software which does not effectively utilize the programming language in which it is coded. Poor programming practices exhibited by this lack of discipline include:

- unmeaningful variable and procedure names
- few or no comments
- no formatting of the source code
- overuse of logical transfers to other parts of the program
- use of non-standard language features of the compiler
- very large, poorly structured programs.

The task of understanding poorly written code becomes even more arduous for the maintainer when the program has been modified by different individuals and there is a multiplicity of programming styles. Often, such code simply does not do what it was intended to do. Even if this code produces expected results, it is sometimes harder to use than anticipated; is not suited for the skill level available to use it; or is slow and unresponsive. Attempting to change such code without the aid of up-to-date specifications or other documentation is often a time-consuming effort.

4.1.3 Software designed for outdated hardware

There are many problems associated with maintaining software which was designed to run on previous generation, outdated hardware. Oftentimes, the investment in the software is such that it cannot be discarded or rewritten and must be kept functioning as efficiently as possible. The first difficulty is in finding maintainers who are ready, able and willing to maintain these systems. Few 'good' programmers will be willing to work on hardware which is unique and for which the acquired skills are not relevant to other potential work. The career advancement opportunities from working on such a system are minimal to non-existent. Additionally, most systems of this type are very difficult to maintain.

4.1.4 Lack of common data definitions

An application system (whether it is large or small) should have common data definitions (variable names, data types, data structures, etc.) for all segments of the system. These common definitions entail the establishment of global variable names which are used to refer to the same data values throughout the system. In addition, the structure of any data array or record should be defined and used for all programs in the system. Problems invariably arise when two or more programmers independently create data names and structures which conflict or do not logically associate with one another.

4.1.5 More than one programming language used

The use of more than one programming language in an application system (for example, assembly language subroutines to perform specific processes in a Cobol program) is often the cause of many software maintenance problems. If the maintainer is not proficient in the use of each of the specific languages, the quality and consistency of the maintenance can be affected. Changes to any of the languages, or corresponding compilers, may also necessitate changes to the application system.

4.1.6 Increasing inventory

Rapidly changing technology and its impact on the practices, procedures, and requirements in many organizations has resulted in a substantial growth in the number of new application systems. In addition, the average life expectancy of a software system has increased from about three years, a decade ago, to seven-to-eight years today [GREE81].

4.1.7 Excessive resource requirements

While some types of maintenance (especially enhancements) may legitimately result in increased resource requirements, other maintenance often results in needless increases. This occurs primarily because of the maintainer's inability to correctly and quickly determine the optimum solution for the required change. The changes are accomplished by making a "patch" to the source code (or worse, to the object code) which does not fit well and is not carefully integrated into the system. Subsequent maintenance efforts may compound this problem until the resource requirements become excessive.

4.2 Documentation

One of the major problems in software maintenance can be summarized in the single phrase - " a failure to communicate." The maintainer who receives the assignment to perform maintenance on the system must first understand what the program is doing, how it is doing it, and why. This job is greatly simplified if the original requester, the designer, the developer, and the previous maintainers have communicated all the pertinent information about the system. This communication should include design specifications, code comments, programmer notebooks, and other documentation.

Too often, the maintainer receives little, no, conflicting, or incorrect communication from those who have previously handled the system. There is often inadequate documentation; no detailed record of the original request and subsequent updates; no explanation of existing code and changes which have been made to the code; a weak understanding of new user requests; and no explanation concerning why seemingly complex or convoluted logic and coding structures were selected over a more simple implementation.

Thus, the problems of software maintenance begin simply with a breakdown in communication between those involved with ensuring that the system does what it is supposed to do. This communication is hampered by the inability of those involved to speak the same language (jargon), the inability to understand the basic requirements (users not understanding computing; programmers not understanding user requirements), and most importantly the time frame in which the actions occur. There may be months or years between the original development of a system and each subsequent maintenance activity. When a problem occurs, none of the individuals involved with the original design, implementation, and previous maintenance may be available. The only source of information available may be the documentation and the code. Thus, good documentation is the only means for good communication. The more complete, clear, and concise this communication is, the greater the chance that maintenance can be performed in a timely, efficient, and accurate manner.

4.3 Users

Users are often unable to concisely specify what they want from an application system. The initial requirements definition and design often lack the detailed specificity which would enable the developer to create a system which accurately performs all of the functions the user needs. Thus, an incomplete system is placed into production. The maintainer must enhance the system using the initial, inadequate specifications and the new, sometimes vague, sometimes conflicting, often incomplete, change requests from the user.

If a system is well-specified, well-designed, well-implemented, and does what the user needs, the user will often think of things to add. The old adage that "nothing succeeds like success" holds true for software development and maintenance. The more successful a system is, the more additional features the user will think of. If the system works well, the user will be constantly demanding more features. If it does not work well, there will be a constant demand for remedial action to make it function properly. Therefore, it is essential that management establish and enforce controls to ensure that the change requests are both justified and do not interfere with the maintenance workload.

User requests for changes and enhancements which are excessive, conflicting, or vague have a major impact on the maintenance of an application system. Much of the difficulty in this area stems from the fact that the user is often unaware of the impact that one change can have on both the system and the maintenance workload. The number of user requests for a specific system is usually directly proportional to the success of the original system and the previous maintenance efforts. A careful and thorough management review of user change requests is essential for controlling the level of software maintenance and ensuring adequate feedback to the user on the cost and consequences of each request.

4.4 Personnel

A common and widespread complaint by maintenance personnel is that software maintenance is considered to be unimportant, unchallenging, unrewarding, uncreative work which is not appreciated by the user or by the rest of the ADP organization. Software maintenance requires the efforts of experienced, well-qualified, dedicated professionals. It should not be solely the responsibility of the new or junior staff. With the development of more multi-purpose, complex software systems, there is a greater need for software

maintainers who can readily understand the entire system.

Traditionally, management has not rewarded personnel who performed software maintenance as generously as those who performed software development. It was generally thought that systems analysts, designers, and developers were responsible for the most difficult, challenging tasks, and therefore, must be more capable.

While this attitude is still common, there is an increasing awareness by management of the importance of software maintenance to the successful, smooth operation of an organization. Many technical personnel, however, still view software maintenance as an assignment to be avoided at all costs. There is too often a general lack of recognition that a good maintainer must be a highly skilled, competent programmer and analyst concerned both with making the actual changes and with assessing the impact of those changes on the system and its environment.

5.0 THE IDEAL MAINTAINER

Software maintenance is the lifeblood of an ADP organization. Persons assigned to perform maintenance must effectively meet the challenge of maintaining a software system while keeping the user satisfied, costs down, and the system operating efficiently.

The characteristic qualities of this ideal maintainer include:

Flexibility - The ability to adapt to different or changing styles of coding, user requests, and priorities.

Self-motivation - the ability to independently initiate and complete work after receiving an assignment.

Responsibility - reliability; performance of assigned tasks in a dependable, timely manner.

Creativity - the ability to apply innovative and novel ideas which result in practical solutions.

Discipline - the ability to be consistent in the performance of duties and not prone to trying haphazard approaches.

Analytic - the ability to apply well thoughtout analysis to a problem.

Thorough - to address even the smallest detail to ensure that all aspects of the problem are understood and nothing is left untested.

Experience - to have been exposed to a variety of applications and programming environments.

The ideal maintainer should be a senior, experienced professional who can perform all of the functional activities which occur during the software life cycle. Equally important from a maintenance standpoint, the maintainer should be extremely knowledgeable about the existing system before attempting to change it.

The maintainer must be able to analyze the problem and the impact on the program, determine the requirements and design changes necessary for the solution, test the solution until the desired results are obtained, and then release the revised software to operations or the user. The maintainer's task is both intellectually and technically difficult. Maintenance is an activity where everything that can go wrong eventually does. The problems will continue to surface and

enhancements will be requested as long as the system is used. It is a function which must be anticipated and planned for. It is also a function for which there may be an unending succession of emergencies to which staff must be assigned from other "more important" work.

The maintainer is also an intermediary between the application systems support staff and the users. Maintenance, unlike development, cannot start with a clean slate and not be affected by previous decisions and work. It often takes a great deal of time and patience to analyze both the users needs and the existing system, and then to carefully and adequately implement the existing changes.

In the final analysis, the most important function of an application system software support activity is software maintenance. It is the maintenance, and the response to the user problems which arise, which are always in the spotlight. Unfortunately, there is usually far less attention paid to maintenance when it is done well and the users are pleased. Maintenance is an ongoing, almost always intense, effort which should be spotlighted for its successes, as well as its failures.

6.0 SYSTEM MAINTENANCE VS SYSTEM REDESIGN

Although maintenance is an ongoing process, there comes a time when serious consideration should be given to redesigning a software system. A major concern of managers and software engineers is how to determine whether a system is hopelessly flawed or whether it can be successfully maintained. Admittedly, the thought of software redesign may not be a comfortable one. Nevertheless, the costs and benefits of the continued maintenance of software which have become error-prone, ineffective, and costly must be weighed against that of redesigning the system.

While there are no absolute rules on when to rebuild rather than maintain the existing system, some of the factors to consider in weighing a decision to redesign or maintain are discussed in this section. These characteristics are meant to be general "rules of thumb" which can assist a manager in understanding the problems in maintaining an existing system and in deciding whether or not it has outlived its usefulness to the organization.

Table 4 - Characteristics of Systems Which Are Candidates for Redesign

---

1. Frequent system failures
2. Code over seven-to-ten years old
3. Overly complex program structure and logic
4. Code written for outdated hardware
5. Running in emulation mode
6. Very large modules or unit subroutines
7. Excessive resource requirements
8. Hard-coded parameters which are subject to change
9. Difficulty in keeping maintainers
10. Seriously deficient documentation
11. Missing or incomplete design specifications

When a decision has been reached to redesign or to stop supporting a system, the decision can be implemented in a number of ways. Support can simply be removed and the system can die through neglect; the minimum support needed to keep it functioning may be provided while a new system is built; or the system may be rejuvenated section by section and given

an extended life. How the redesign is affected depends on the individual circumstances of the system, its operating environment, and the needs of the organization it supports.

The potential for redesign as opposed to continued maintenance is directly proportional to the number of characteristics listed in Table 4. The greater the number of characteristics present, the greater the potential for redesign.

6.1 Frequent System Failures

A system which is in virtually constant need of corrective maintenance is a prime candidate for redesign. As systems age and additional maintenance is performed on them, many become increasing fragile and susceptible to changes. The older the code, the more likely frequent modifications, new requirements, and enhancements will cause the system to break down.

An analysis of errors should be made to determine whether the entire system is responsible for the failures, or if a few modules or sections of code are at fault. If the latter is found to be the case, then redesigning those parts of the system may suffice.

6.2 Code Over Seven Years Old

The estimated life cycle of a major application system is seven-to-ten years. Software tends to deteriorate with age as a result of numerous fixes and patches. If a system is more than seven years old, there is a high probability that it is outdated and expensive to run. A great deal of the code in use today falls into this category. After seven-to-ten years of maintenance, many systems have evolved to where additional enhancements or fixes are very time-consuming to make. A substantial portion of this code is probably neither structured, nor well-written. While this code was adequate and correct for the original environment, changes in technology and applications may have rendered it inefficient, difficult to revise, and in some cases obsolete.

However, if the system was designed and developed in a systematic, maintainable manner, and if maintenance was carefully performed and documented using established standards and guidelines, it may be possible to run it efficiently and effectively for many more years.

6.3 Overly Complex Program Structure And Logic Flow

"Keep it simple" should be the golden rule of all programming standards and guidelines. Too often, programmers engage in efforts to write a section of code in the least number of statements or utilizing the smallest amount of memory possible. This approach to coding has resulted in complex code which is virtually incomprehensible. Poor program structure contributes to complexity. If the system being maintained contains a great deal of this type of code and the documentation is also severely deficient, it is a candidate for redesign.

Complexity also refers to the level of decision making present in the code. The greater the number of decision paths, the more complex the software is likely to be. Additionally, the greater the number of linearly independent control paths in a program, the greater the program complexity. Programs characterized by some or all of the following attributes are usually very difficult to maintain and are candidates for redesign:

- excessive use of DO loops
- excessive use of IF statements
- unnecessary GOTO statements
- embedded constants and literals
- unnecessary use of global variables
- self-modifying code
- multiple entry or exit modules
- excessive interaction between modules
- modules which perform same or similar functions.

6.4 Code Written For Previous Generation Hardware

Few industries have experienced as rapid a growth as the computer industry, particularly in the area of hardware. Not only have there been significant technological advances, but, the cost of hardware has decreased ten-fold during the last decade. This phenomenon has generated a variety of powerful hardware systems. Software written for earlier generations of hardware is often inefficient on newer systems. Attempts to superficially modify the code to take advantage of the newer hardware is generally ineffective, time-consuming and expensive.

6.5 Running In Emulation Mode

One of the techniques used to keep a system running on newer hardware is to emulate the original hardware and operating system. Emulation refers to the capacity of one system to execute a language written for another machine. In effect, it extends the architecture (hardware and software) of the host machine to include the range of the machine being emulated. This is normally done when resources are not available to convert a system, or the costs would be prohibitive. These systems run a very fine line between functional usefulness and total obsolescence. One of the major difficulties in maintaining this type of system is finding maintainers who are familiar with the original hardware and who are willing to maintain it. Since the hardware is outdated, the specific skills developed in maintaining the system have little applicability elsewhere. Thus, the career development potential of supporting such a system is not very promising.

6.6 Very Large Modules Or Unit Subroutines

"Mega-systems" which were written as one or several very large programs or sub-programs (thousands or tens-of-thousands of lines of code per program) can be extremely difficult to maintain. The size of a module is usually directly proportional to the level of effort necessary to maintain it. If the large modules can be restructured and divided into smaller, functionally related sections, the maintainability of the system will be improved.

6.7 Excessive Resource Requirements

An application system which requires a great deal of CPU time, memory, storage, or other system resources can place a very serious burden on all ADP users. These "resource hog" systems which prevent other jobs from running, may not only require the addition of an extra shift, but may degrade the service to all users. Questions which should be answered when deciding what to do about such a system include:

- Is it cheaper to add more computer power or to redesign and reimplement the system?

- Will a redesign reduce the resource requirements? If it won't, then there is no use in redesigning.

6.8 Hard Coded Parameters Which Are Subject To Change

Many older systems were designed with the values of parameters used in performing specific calculations "hard coded" into the source code rather than stored in a table or read in from a data file. When changes in these values are necessary, (withholding rates, for example) each program in the system must be examined, modified and recompiled as necessary. This is a time-consuming, error prone process which is costly both in terms of the resources necessary to make the changes and the delay in getting the changes installed.

If possible, the programs should be modified to handle the input of parameters in a single module or to read the parameters from a central table of values. If this can't be done, serious consideration should be given to redesigning the system.

6.9 Difficulty In Keeping Maintainers

Programs written in low level languages, particularly assembler, require an excessive amount of time and effort to maintain. Generally, such languages are not widely taught or known. Therefore, it will be increasingly difficult to find maintainers who already know the language. Even if such maintainers are found, their experience with low-level languages is probably dated.

6.10 Seriously Deficient Documentation

One of the most common software maintenance problems is the lack of adequate documentation. In most organizations, the documentation ranges from nonexistent to out-of-date. Even if the documentation is good when delivered, it will often steadily and rapidly deteriorate as the software is modified. In some cases, the documentation is up-to-date, but still not useful. This can result when the documentation is produced by someone who does not understand the software or what is needed.

Perhaps the worst documentation is that which is well-structured and formatted but which is incorrect or outdated. If there is no documentation, the maintainer will be forced to analyze the code in order to try to understand the system. If the documentation is physically deteriorated, the maintainer will be skeptical of it and verify its accuracy. If it looks good on the surface, but is technically incorrect, the maintainer may mistakenly believe it to be correct and accept what it contains. This will

result in serious problems over and above those which originally necessitated the initial maintenance.

6.11 Missing Or Incomplete Design Specifications

Knowing "how and why" a system works is essential to good maintenance. If the requirements and design specifications are missing or incomplete, the task of the maintainer will be more difficult. It is very important for the maintainer to not only understand what a system is doing, but how it is implemented, and why it was designed. Missing or incomplete design specifications often result in end products which do not perform as intended. The user must then request new changes and enhancements.

7.0 CONTROLLING SOFTWARE CHANGES

The key to controlling software maintenance is to organize it as a visible, discrete function and, to the extent possible, plan for it. It is not enough for the software manager to manage the budget, people, and schedules. It is equally important that the software changes be managed and controlled.

Table 5 - Suggested Policies for Controlling Software Changes

1. Require formal (written) requests for all changes.
2. Review all change requests and limit changes to those approved.
3. Analyze and evaluate the type and frequency of change requests.
4. Consider the degree to which a change is needed and its anticipated use.
5. Evaluate changes to ensure that they are not incompatible with the original system design and intent. No change should be implemented without careful consideration of it ramifications.
6. Emphasize the need to determine whether a proposed change will enhance or degrade the system.
7. Approve changes only if the benefits outweigh the costs.
8. Schedule all maintenance.
9. Enforce documentation and coding standards.
10. Require that all changes be implemented using modern, programming practices.
11. Plan for preventive maintenance.

7.1 Controlling Perfective Maintenance

Perfective maintenance comprises an estimated 60% of the total maintenance effort. It deals primarily with expanding, extending, and enhancing a system to give it greater power, more flexibility, additional capabilities, or greater reliability. Requests for perfective maintenance are initiated by three different groups: the user, upper management, and the maintenance staff.

The user is almost never completely satisfied with a system. Either it does not perform up to expectations, or, as the user gains confidence in the system, additional features become obvious and the maintenance staff is asked to add those features. This is a normal evolution in all software systems and must be planned for when developing budget requests and resource allocation schedules.

Upper management drives the perfective maintenance process by requesting new and enhanced features which must be incorporated into existing application systems. Once again, this is a normal part of the functioning of any organization and must be planned for in the maintenance budget.

Finally, the maintenance staff drives the perfective maintenance process. As a maintainer works with a system, inefficiencies and potential problems are often found. These problems, while not requiring immediate attention, are such that at some point in time they could have a significant impact on either the functioning of the system or on the ability to maintain it. Thus, the "cleaning up" of code (often referred to as "preventive maintenance") is an important perfective maintenance process which should be planned for and included in the resource allocation schedule. The proverbial "stitch in time" of preventive maintenance can often prevent minor problems in a systems from becoming major problems at some later date. This undoubtedly will make future maintenance easier as a result of the "cleaning up" of the code.

The management of perfective maintenance deals primarily with maintaining an orderly process in which all requests are formally submitted, reviewed, assigned a priority, and scheduled. This does not mean that unnecessary delays should be built into the process, or that in small organizations these steps are not consolidated. Rather, it defines a philosophical approach which can help the maintenance manager bring order to the maintenance environment.

There should be a centralized approval point for all maintenance projects. This may be the maintenance project manager or, for larger systems or organizations, a review

board. Changes should not just happen to a system. When the need for a change or enhancement arises, a formal written request should be submitted. Each request should be evaluated on the basis of resource requirements, time to complete the work, impact on the existing system and other maintenance efforts, and justification of need. The centralized approval process will enable one person or group of persons to have knowledge of all the requested and actual work being performed on the system. If this is not done, there is the likelihood that two or more independent changes to the system will be in conflict with one another and as a result, the system will not function properly. Additionally, different users will often request the same enhancements to a system but will have small differences in the details. By coordinating these requests, details can be combined and the total amount of resources required can be reduced.

If the system requires maintenance as a result of changes in policy or procedures in the organization, an evaluation of the cost and effects of the changes should be prepared for upper management. Ideally, this should be prepared prior to the decision to institute the changes, but even if it is not, management and the users must be aware of the costs. Users often request enhancements to a system because it "would be nice to have" or another system has a similar feature. These requested enhancements should be evaluated and the estimated costs reported to the user. Regardless of whether or not the users are responsible for funding the work, it is important to keep them aware of the actual costs of their requests. Doing so will help to minimize the amount of unneeded or marginally needed enhancements which must be installed on the system. In addition, this type of interchange with the user will help the maintenance manager in evaluating and assigning priorities to the work requests.

In many organizations there is a significant backlog of maintenance work requests. Users need to understand the level of effort required to meet their requests and the relative priority of the work in relationship to other user requests. This can only be accomplished by involving all parties in the discussions and keeping everyone informed of the schedules and actual progress.

7.2 Controlling Adaptive Maintenance

Adaptive maintenance comprises approximately 20% of the maintenance burden. It consists of any effort required to keep a system functioning as a result of changes in the environment in which it must operate, and is, to a great degree, beyond the control of the software maintenance manager. Changes to the operating system, system utilities,

terminal devices, and the rules, laws, and regulations which the software must incorporate, are the primary causes of adaptive maintenance. The maintenance efforts required are usually non-productive in terms of improving the application system.

There is little that the software maintainer can do to control changes to rules and legislation. These changes, to the extent possible, should be anticipated and the code structured in a manner which facilitates making the needed changes. This type of adaptive maintenance usually must be performed whenever it is required. Management should always be given feedback regarding the impact that changes in policies and regulations have on the maintenance of a system, especially the cost. This feedback will improve the future decision making process and may reduce the level of adaptive maintenance.

In many organizations, the application support organization functions independently of the computer facility organization. As a result, there is inadequate communication and understanding by each group regarding the impact of decisions and work on the other function. Thus, changes may be made to the environment and announced to the user community without giving the application support function an opportunity to analyze the impact of the changes and the effect on the application system. Similarly, changes or additions to an application system which increase the computer resource requirements may cause serious problems with the functioning of all applications using the computer.

Therefore, it is very important that the facilities organization and the applications support organization work closely to minimize the impact of one organization's work on the other organization. There are times when a choice simply does not exist, but usually, through adequate planning and evaluation, both organizations can accomplish their objectives with a resulting net improvement for each.

The application support manager has the responsibility to know what changes to the environment are being planned and considered, and to keep management informed of their potential impact (both negative and positive). In doing this, the total costs and the implications of the changes can be reviewed by management. Decisions can then be made regarding which organization should bear the costs of the resulting required adaptive maintenance of the application systems.

7.3 Controlling Corrective Maintenance

Corrective maintenance is primarily the identification and removal of errors, bugs, and other code defects that either reduce the effectiveness of the software or render the product useless. This category of maintenance is concerned with returning the code to an operational state. Controls are needed to ensure that the occurrence of errors or bugs are the exception rather than the rule.

Most of the cost of software maintenance is often assumed to be the result of poor workmanship during development and prior maintenance phases of the system. While this is a contributing cause, it is very rare for even a "perfect" system to not require significant maintenance during its lifetime. While software does not "break" in the sense that a piece of hardware can fail, it can become non-functional, or faulty due to changes in the environment in which it must operate, the size or sophistication of the user community, the amount of data it must process, or damage to code which is the result of other maintenance efforts on other parts of the system. Corrective maintenance is necessitated by discovery of a flaw which has always existed in the system or was introduced during prior maintenance.

Difficulties encountered during corrective maintenance can be reduced significantly by the adoption and enforcement of appropriate standards and procedures during the development and maintenance of the software. While it is probably not possible to eliminate corrective maintenance, the consistent and disciplined adherence to effective design and programming standards can, and will, significantly reduce the corrective maintenance burden.

8.0 IMPROVING SOFTWARE MAINTENANCE

Maintainability is the ease with which software can be changed to satisfy user requirements or can be corrected when deficiencies are detected. The maintainability of a system must be taken into consideration throughout the life cycle of that system. Many techniques and aids exist to assist the system developer, but there has been little emphasis on aids for the maintainer. However, since the processes which occur in the maintenance phase are similar to those of the development phase, there is considerable overlap in the applicability of the development aids in the maintenance environment.

The philosophies, procedures, and techniques discussed in this section should be utilized throughout the life cycle of a system in order to provide maintainable software. Software systems which were not developed using these techniques can also benefit from their application during major maintenance activities. In other words, if a system must be maintained, the maintainability of the system can be improved by applying the ideas discussed in this section to the parts of the system which are modified during the maintenance process. While the effect will not be as pronounced as when programs are "developed with maintenance in mind", future maintenance efforts can be made easier by utilizing the techniques described in this section to "maintain systems with future maintenance in mind".

Table 6 - Factors Which Affect Source Code Maintainability

---

1. Use of a single high order language
2. Coding conventions for variable names, structures, format, grouping, etc.
3. Structure and modularity
4. Standard data definitions
5. Meaningful comments in the code
6. Avoidance of compiler extensions

8.1 Source Code Guidelines

Source code guidelines and standards aid maintainability by providing a structure and framework within which systems can be developed and maintained in a common, more easily understood, manner.

8.1.1 Use a single high-order language

The use of more than one programming language or the use of machine, assembler or outdated languages, when it is not absolutely necessary to do so, can seriously impact the maintainability of a system. When more than one language is employed, the potential for communication problems between modules is increased. Systems written in low-order or outdated languages are difficult to maintain because they generally require more source code to perform the same amount of work. Wherever possible, a single high order language (HOL) should be used. Advantages of using a HOL include:

- HOLs resemble English and are easy to learn, read and understand.
- There are standards for the commonly used HOLs (COBOL and FORTRAN).
- There are a substantial number of programmers who understand and can use HOLs effectively.
- Many of the older machine languages are no longer supported by the manufacturer.
- Fewer programmers understand machine languages, and fewer still can use them effectively.
- HOLs are self-documenting to a large degree.
- It is easier to move from one environment to another with an HOL.

8.1.2 Coding conventions

The first obstacle a maintainer must conquer is the code itself. Unfortunately, a great deal of the source code written by developers and maintainers is not written with the future maintainer in mind. Thus, the readability of source code is often very poor.

> Source code should be self-documenting and be written in a structured format

Regardless of the programming language(s) used, simple rules regarding the use of the language(s) and the physical formatting of the source code should be established. Code standards do not have to be lengthy or complex in order to be effective. In fact, like the code itself, the best standards are simple and short. The following techniques can improve program readability and should be used as the basis for a code standard.

- Keep it simple. Complicated, fancy, exotic, tricky, confusing, or "cute" constructions should be avoided whenever a simpler method is available. Use common sense and write code as if you had to pick it up and maintain it without ever having seen it before.

- Indentation, when properly utilized between sections of code, serves to block the listing into segments. Indentation and spacing are both ways to show subordination. It is very difficult to follow code which continues line after line without a break or change in form.

- Extensively comment the code with meaningful comments. Do not comment for comment's sake. Rather, comment in order to communicate to subsequent maintainers not only what was done and how it was done, but why it was done in this manner.

- Use of meaningful variable names is one of the most important coding principles to follow when developing and maintaining programs. The name of a variable should convey both what it is and why it is used.

- Similar variable names should be avoided. Each variable name should be unique in order to prevent confusion.

- When numerics are used, they should be placed at the end of the variable. Some of the more common errors are caused by mistaking variable names which begin with the numerics 0,1,2,5 for O,I,Z,S, respectively. Numbers used as program tags or labels should be sequential.

- Logically related functions should be grouped together in the same module or set of modules. It is extremely difficult to analyze the program flow when execution jumps in and out of different portions of code. To the extent possible, the logic flow should be from top to bottom of the program.

- Avoid non-standard features of the version of the language being used unless absolutely necessary. Failure to do so will exacerbate problems of conversion or movement of the program to another machine or system.

8.1.3 Structured, modular software

While there has been considerable debate regarding structured programming, the consensus is that generally, such code is easier to read. A structured program is constructed with a basic set of control structures which each have one exit and one entry point. Structured programming techniques are well-defined methods which incorporate top-down design and implementation and strict use of structured programming constructs. Whether the strict definition, or a more general approach (which is intended to organize the code and reduce its complexity) is used, structured programming has proven to be useful in improving the maintainability of a system.

Modularity refers to the structure of a program. A program comprised of small, hierarchical units or sets of routines, where each performs a particular, unique function, is said to be modular. It is not, as is often thought, mere program segmentation. A module is said to have two basic determinants: cohesiveness and coupling.

Cohesion refers to the degree to which the functions or processing elements within a module are related or bound together. It is the intra-module relativeness. The greater the cohesion, the less impact changes will have on the software.

Coupling refers to the degree that modules are dependent upon each other. The less dependency or interaction there is between modules, the better, from both a functional and a maintenance standpoint. A high degree of cohesion almost always assures a lower degree of coupling. Controlling cohesion and coupling are very effective techniques in the design and maintenance of structured, modular software.

One of the most obvious advantages of designing and coding structured modules is that if it is determined that a function is no longer needed, only that module is affected. The size of a module is dependent upon its function. It should, however, be kept as small as possible. Modules should be constructed using the following basic design principles:

- Modules should perform only one principal function.
- Interaction between modules should be minimal.
- Modules should have only one entry and one exit point.

8.1.4 Standard data definitions

It is very important that individual modules of a system not only be able to communicate with one another, but that the maintainer understand what is being communicated. A typical problem in a large multi-module system is that one person will use a set of names for data items which do not match the names used by another person on the team. Even more serious is the use of the same names to represent two different data items. Thus, it is imperative that a standard set of data definitions be developed for a system. These data definitions will define the name, physical attributes, purpose, and content of each data element utilized in the system. These names should be as descriptive and meaningful as possible. If this is consistently and correctly done, the task of reading and understanding each module and ensuring correct communication between each module is greatly simplified.

8.1.5 Well-commented code

Good commentary increases the intelligibility of source code. In addition to making programs more readable, comments serve two other vital purposes. They provide information on the purpose and history of the program, its origin (the author, creation and change dates), the name and number of subroutines, and input/output requirements and formats. They also provide operation control information, instructions, and recommendations to help the maintainer understand aspects of the code that are not clear.

Maintainers (and managers) often mistakenly confuse quantity for quality when writing comments. The purpose of comments is to convey information needed to understand the process and the reasons for implementing it in that specific manner, not how it is being done. Comments should be thought of as the primary form of documentation. They should include the following:

- what the code is doing,
- why a process is being performed,
- why it is implemented in the specific manner,
- how this section of code affects and interacts with other sections of code,
- any known or potential problems,
- when the changes were made,
- who made the changes,
- what specific code was modified,
- any other information which might help a future maintainer in understanding and modifying the code.

8.1.6 Avoid compiler extensions

The use of non-standard features of a compiler can have serious effects on the maintainability of a system. If a compiler is changed, or the application system must be transported to a new machine, there is a very great risk that the extensions of the previous compiler will not be compatible with the new compiler. Thus, it is best to refrain from language extensions and to stay in conformance with the basic features of the language. If it is necessary to use a compiler extension, its use should be well-documented.

8.2 Documentation Guidelines

The documentation of a system should start with the original requirements and design specifications and continue throughout the life cycle of the system. Good software documentation is essential to good maintenance.

Table 7 - Documentation Guidance

---

1. Keep it simple and concise.

2. The maintainer's first source of documentation is the source code.

3. The manager's first source of documentation is the design specifications and implementation reports.

4. The user's first source of documentation is the Users Guide and the maintainer.

5. Do not under document. Do not over document.

6. Documentation cannot be "almost correct". Either it is up-to-date, or it is useless.

7. Documentation maintenance is a vital part of system maintenance.

8. Documentation should be available to the maintainer at all times.

The documentation must be planned so a maintainer can quickly find the needed information. A number of methodologies and guidelines exist which stress differing formats and styles. While preference may differ on which methodology to use, the important element is to adopt a documentation standard and to then consistently enforce adherence to it for all software projects.

The success of a software maintenance effort is dependent on how well information about the system is communicated to the maintainer. Documentation should support the useable transfer of pertinent information. Documentation guidelines should include instructions on what information must be provided, how it should be structured, and where the information should be kept. In establishing these guidelines and standards, keep in mind that the purpose is to communicate necessary, critical information, not to communicate all information.

Basically, the documentation standards should require the inclusion of all pertinent material in a documentation folder or notebook. This material should cover all phases of the software life cycle and must be kept fully updated. Management must enforce documentation standards and NOT permit short cuts. There should be a requirement to complete and/or update documentation before new work assignments are begun.

The key to successful documentation is that not only must the necessary information be recorded, it must be easily and quickly retrievable by the maintainer. On-line documentation which has controlled access and update capabilities is the best form of documentation for the maintainer. If the documentation cannot be kept on-line, a mechanism must exist to permit access to the hard-copy documentation by the maintainer at any time.

If documentation guidelines, or any other software guidelines or standards, are to be effective, they must be supported by a level of management high enough within the organization to ensure enforcement by all who use the software or are involved with software maintenance. Such guidelines, when supported by management, will help direct attention toward the need for greater discipline in the software maintenance process.

For further information on documentation guidelines and standards, see [FIPS38], [FIPS64], and [NBS87].

8.3 Coding And Review Techniques

The techniques listed in this section have been found to be very effective in the generation of maintainable systems. Not all techniques are generally applicable to all organizations, but it is recommended that they be considered.

Table 8 - Coding and Review Techniques

---

1. Top down / Bottom up design and implementation
2. Peer reviews
3. Walkthroughs
4. Chief programmer team

8.3.1 Top down/bottom up approach

A top-down design approach (development or enhancements) involves starting at the macro or overview level and successfully breaking each program component or large, complex problem into smaller less complicated segments. These segments are then decomposed into even smaller segments until the lowest level module of the original problem is defined for each branch in the logic flow tree.

In general, top-down implies that major functions are considered first. Once it is clear how they fit together, the next, lower level functions are designed. During the first phase, the lower level functions are often created as empty black boxes or modules that simply return control to the major level or calling functions.

The bottom-up design approach begins with the lowest level of elements. These are combined into larger components which are then combined into divisions, and finally, the divisions are combined into a program. A bottom-up approach emphasizes designing the fundamental or "atomic" level modules first and then using these modules as building blocks for the entire system.

Both of these approaches are valid and superior to a random "seat-of-the-pants" approach. In most situations, a combination of top-down and bottom-up can be utilized to

develop a clear, concise, maintainable system. The adoption and adherence to either approach provides a structure or methodology which enables persons working on a system to communicate with one another in a manner which is consistent and understandable.

8.3.2 Peer reviews

Peer review is a quality assurance method in which two or more programmers review and critique each other's work for accuracy and consistency with other parts of the system. This type of review is normally done by giving a section of code developed by one programmer to one or more other peer programmers who are charged with identifying what they consider to be errors and potential problems. It is important to establish and to keep clearly in the participants' minds that the process is not an evaluation of a programmers capabilities or performance. Rather it is an analysis and evaluation of the code. As stated in the name, such reviews are performed on a peer basis (programmer to programmer) and should never be used as a basis for employee evaluation. Indeed, project managers should not, if possible, be involved in the peer reviews.

8.3.3 Walkthroughs

Walkthroughs of a proposed solution or implementation of a maintenance task can range from informal to formal, unstructured to structured, and simple to full-scale. The principle involved in walkthroughs is simply that "two heads are better than one." In its simplest form, a walkthrough can be two maintainers sitting down and discussing a task which one of then is working on. In its more complex forms, there may be a structured agenda, report forms, and a recording secretary. Managers may or may not participate in walkthroughs. However, this is an excellent way for a manager to keep informed about the work being performed by the team.

The basic format of a walkthrough is for the person whose work is being reviewed to describe in detail the proposed solution or the draft of the code. The reviewer(s) ask(s) questions to clarify areas where questions arise and point out any errors or potential problems which are spotted. The goal, as in peer reviews, is to minimize the number of design, logic, and/or coding flaws which remain in the system. Walkthroughs are similar to peer reviews, but differ in that the manager may be present; the reviewers meet as a group to discuss the work under consideration; and there are often formal recording and reporting mechanisms.

Two important points should be stressed regarding the manager's role in a walkthrough:

1. Walkthroughs should never be used as part of an employee evaluation. The goal is an open, frank dialogue which results in the refinement of good ideas and the changing or elimination of bad ones.

2. The manager's role should only be as active as his or her technical expertise regarding the subject matter permits. The manager must recognize that the other members of the walkthrough team probably have greater technical knowledge about the specific subject being discussed. Participating in a passive manner can be an excellent means to attain an understanding of the maintenance effort and to improve the manager's technical understanding of the system.

8.3.4 Chief programmer team

The chief programmer team is based on the premise that an experienced programmer, supported by a team of programmers, can produce computer programs with greater speed and efficiency than a group of programmers working under the traditional line and staff organization. The size of the team can range from 3-10, with the chief programmer being responsible for overall design, development, review, and evaluation of the work performed by the members of the team. This can include the establishment and enforcement of rules regarding programming style, control, and the integrity of the programs.

The chief programmer functions as the focal point of the maintenance team and is required to be aware and familiar with all work performed by the team. There is an enormous amount of administrative and technical responsibility placed on the chief programmer. This person must have impeccable leadership abilities, a strong technical capability, and the ability and willingness to delegate work and responsibility.

8.4 Change Control

Change control is necessary to ensure that all maintenance requests are handled accurately, completely, and in a timely manner. It helps assure adherence to the established standards and performance criteria for the system and facilitates communication between the maintenance team members and the maintenance manager.

Table 9 - Controlling Changes

---

1. Change request
2. Code audit
3. Review and Approval

8.4.1 Change request

All changes considered for a system should be formally requested in writing. These requests may be initiated by the user or maintainer in response to discovered errors, new requirements, or changing needs. Procedures may vary regarding the format of a change request, but it is imperative that each request be fully documented in writing so that it can be formally reviewed. The review may be performed by the project manager or a change review board. The key, however, is that there must be a formal, well-defined mechanism for initiating a request for changes or enhancements to a system. Change requests should be carefully evaluated and decisions to proceed should be based on all the pertinent areas of consideration (probable effects on the system, actual need, resource requirements vs resource availability, budgetary considerations, priority, etc.). The decision and reasons for the decision should be recorded and included in the permanent documentation of the system.

The change request should be submitted on forms which contain the following information:

- name of requester
- date of request
- purpose for request (error reported, enhancement, etc)
- name of program(s) affected
- section of code/line numbers affected
- name of document(s) affected
- name of data file(s) affected
- date request satisfactorily completed
- date new version operational
- name of maintainer
- date of review
- name of reviewer
- review decision

8.4.2 Code audit

The code review or audit is a procedure used to determine how well the code adheres to the coding standards and practices and to the design specifications. The primary objective of code audits is to guarantee a high degree of uniformity across the software. This becomes a critical factor when someone other than the original developer must understand and maintain the software. Audits are also concerned with such program elements as commentary, labeling, paragraphing, initialization of common areas, and naming conventions. The audit should be performed by someone other than the original author. Questions addressed during an audit should include:

- Are comments well constructed?
- Do the comments provide meaningful information?
- Are the comments consistent throughout the code?
- Are the constants centrally defined and locally initialized?
- Are the statement labels descriptive and sequential?
- Is the code formatted in a readable manner?
- Is indentation and paging used to make the code easier to read and understand?

8.4.3 Review and approval

Review and approval is the final phase of the software change control process. Prior to installation, each change (correction, update, or enhancement) to a system should be formally reviewed. In practice, this process ranges from the review and sign-off by the project manager or user, to the convening of a change review board to formally approve or reject the changes. The purpose of this process is to ensure that all of the requirements of the change request have been met; that the system performs according to specifications; that the changes will not adversely impact the rest of the

system or other users; that all procedures have been followed and rules and guidelines adhered to; and that the change is indeed ready for installation in the production system. All review actions and findings should be added to the system documentation folder.

8.5 Testing Standards And Procedures

Testing, like documentation, is an area of software maintenance which is often not done well. Whenever possible, the test procedures and test data should be developed by someone other than the person who performed the actual maintenance on the system. The testing standards and procedures should define the degree and depth of testing to be performed and the disposition of test materials upon successful completion of the testing.

Testing is a critical component of software maintenance. As such the test procedures must be consistent and based on sound principles. Whether the testing is performed on the entire system or on a single module within the system, the same principles are required. They include the following:

- The test plan should define the expected output.
- Whenever possible, the test data should be prepared by someone other than the tester.
- Both the valid, invalid, expected, and unexpected cases should be tested.
- The test should examine whether or not the program is doing what it is supposed to.
- Testing is done to find errors, not to prove that errors do not exist.

For further information on testing, see [FIPS101], [NBS75], [NBS93], [NBS98].

9.0 SOFTWARE MAINTENANCE TOOLS

Software tools are computer programs which can be used in the development, analysis, testing, maintenance, and management of other computer programs and their documentation. This section discusses some tools which can be useful in maintaining a software system. Generally, these tools can be divided into two categories: technical and management. The technical tools can be further subdivided into those which process, analyze, and test the system, and those which help the maintainer manipulate and change the source code and the documentation. The management tools assist the maintenance manager in controlling and tracking all of the maintenance tasks. Table 10 lists some of the tools available to the maintainer and the maintenance manager. A glossary of software tools and techniques can be found in [REIF77].

Table 10 - Software Maintenance Tools

Technical Tools
- Processing Tools
  - Compilers
  - Cross referencer
  - Comparator
  - Traces/Dumps
  - Test data generator
  - Test coverage analyzer
  - Preprocessor
  - Verification/Validation
- Clerical Tools
  - On-line Editor
  - Documentation Library
  - Archival Capabilities
  - Reformatter
  - Data Dictionary

Management Tools
- Problem Reporting
- Status Reporting
- Scheduling
- Configuration Management

9.1 Cross Referencer

One of the single most useful aids to the maintainer is the cross reference list which accompanies the compiler source listing. It usually provides a concise, ordered analysis of the data variables, including the location and number of times the variables are used, as well as other pertinent information about the program.

In large systems, it is often difficult to determine which modules are called or used by other programs, and where within the system a specific module or parameter is used. What is often needed too is the capacity to produce and develop a cross reference listing on an interprogram rather than on an intraprogram basis. This information can be obtained from some of the available cross reference generators. To the maintainer, such information is useful when attempting to backtrack to determine where an error occurred.

9.2 Comparators

Comparators are software tools which accept two (or more) sets of input and generate a report which lists the discrepancies between the input data sets. This tool can be used for finding changes in the source code, input data, program output, etc. It is extremely useful to the maintainer who must ascertain if a change made to the system caused it to fail or work differently. It can also be used to ensure that one set of test results is identical to a previous set, or identify where the results have changed. Most comparators are developed for a specific system. They may be general in nature or work on specific parts of the system and perform specific functions. They are relatively simple to build and are very valuable tools in the maintainer's tool box.

9.3 Diagnostic Routines

Diagnostic routines assist the maintainer by reducing the amount of time and effort required for problem resolution. Some of the more commonly used routines include:

- trace which generates an audit trail of actual operations during execution
- breakpoint which interrupts program execution to initiate debug activities

- save/restart which salvages program execution status at any point to permit evaluation and re-initiation

- dumps which give listings (usually unformatted or partially formatted) of all or selected portions of the program memory at a specific point in time.

Compilers often provide diagnostic capabilities that can be optionally selected to assist the programmer in analyzing the execution flow, and capture a myriad of data at predetermined points in the process. In the hands of a skilled maintainer, these diagnostics can help identify the sections of code which cause the error, as well as what is taking place there. While these aids are extremely useful, they are usually "after the fact" tools used to help determine what has gone wrong with an operational system. Far more useful are diagnostic capabilities which are designed and implemented within the source code as it is developed. This latter type of diagnostic is normally disabled, but can be turned on through the use of one or more control parameters.

9.4 Application Utility Libraries

Most operating systems provide support and utility libraries which contain standard functional routines (square roots, sine, cosine, absolute values, etc.). In addition, HOL compilers have many built-in functions which can be utilized by the programmer to perform standard functions. Just as these libraries provide standardized routines to perform processes which are common to many applications systems, large application systems should have a procedure library which contains routines which are common to various segments of the application system. These functions and utility routines should be available to all persons working on the system from the developer to the maintainer. Application support utility libraries assist by:

- saving time (the programmer does not have to reinvent the wheel).

- simplifying the changing of common code (changes all programs which utilize a module). This usually requires relinking or recompiling each affected program, but it eliminates the need to change lines of code in each of the programs.

- enabling wider use of utility procedures, developed by one person or group, by all persons working on the system.

- facilitating maintenance of the system by keeping the code in a central library or set of libraries.

In addition to the stored library routines, all the source code for the applications system should be stored in a centralized, on-line library. Access to this library should be controlled by a librarian who has the duty of maintaining the integrity of the library and the code.

9.5 On-line Documentation Libraries

System documentation normally consists of one or more folders or files in hardcopy form which are stored at a central location. The need for the maintainers to have access to the information in these documentation folders and the need to keep the documentation up-to-date and secure are sometimes at cross-purposes with one another. Thus, it is recommended that as much documentation as practical also be kept on-line in documentation libraries which the maintainer can access at any time. Updating of this library should be controlled by a librarian.

9.6 On-line/Interactive Change And Debug Facilities

Interactive debugging provides significant advantages over the batch method because of the convenience and speed of modification. With interactive processing, the maintainer can analyze the problem area, make changes to a test version of the system, and test and debug the system immediately. The alternative, to submit a batch job to perform the testing, requires much more time to complete. While in some instances this may be necessary because of system size or resource requirements, most maintenance activities (including perfective maintenance) are highly critical problems which must be addressed and solved as quickly as possible. Interactive processing provides a continuity which enables greater concentration on the problem and quicker response to the tests. Although the estimates of the increase in productivity vary widely, it is clear that there is a substantial improvement when the maintainer has on-line interactive processing capabilities.

9.7 Generation And Retention Of Test Data

Standardized procedures (often developed in-house) for generating and retaining test data are recommended. One of the perennial problems in software maintenance is the lack of test data. While in most instances, test data are generated by the maintainer, studies have found that more errors and inconsistencies are uncovered when test data are prepared by the user, and testing is more effective if samples of the actual data are included in the test data.

Once a test data set has been generated and the system successfully run against it, the data should be retained for use in future maintenance regression testing. Regression testing is the selective retesting of the system to detect any faults which may have been introduced and to verify that the maintenance modifications have preserved the functionality of the system. The system testing verifies that the system produces the same results and continues to meet the requirements specifications. In addition, the results of the testing should be saved in machine readable form so that the results of future maintenance testing can be compared with the previous test results through the use of a comparator.

Although some test data set generators are commercially available, most are developed either as part of the original development effort of a large system or as part of the maintenance effort. A test data generator is usually built for a specific system and designed to test the system to a selected level of detail. Guidance on testing is available in several NBS/ICST publications [FIPS101], [NBS75], and [NBS93].

10.0 MANAGING SOFTWARE MAINTENANCE

The effective use of good management techniques and methodologies in dealing with scheduling maintenance, negotiating with users, coordinating the maintenance staff, and instituting the use of the proper tools and disciplines is essential to a successful software maintenance effort. Software maintenance managers are responsible for making decisions regarding the performance of software maintenance; assigning priorities to the requested work; estimating the level of effort for a task; tracking the progress of work; and assuring adherence to system standards in all phases of the maintenance effort. A software maintenance manager must not only be a good technician, but also a good manager. While this may seem to be an obvious point, it is, in actual practice, far too often ignored.

There appears to be a common failure to recognize the importance of the word "management" in the phrase "software maintenance management". In many instances, technical persons are promoted to positions of management within an organization with the assumption that technical expertise is all that is required to manage effectively a software maintenance operation. On the contrary, a software maintenance function has the same organizational needs and managerial problems as any other function.

The primary duties of a software maintenance manager include:

1. Evaluate, assign, prioritize, and schedule maintenance work requests.
2. Assign personnel to scheduled tasks.
3. Track progress of all maintenance tasks and ensure that they are on or ahead of schedule.
4. Adjust schedules when necessary.
5. Communicate progress and problems to the user.
6. Communicate progress and problems to upper management.
7. Establish and maintain maintenance standards and guidelines.
8. Enforce standards and make sure that the software maintenance is of high quality.
9. Deal with problems and crises as they arise.
10. Keep the morale of the maintenance staff high.

This list is not complete, but is sufficient to illustrate the point that if the words "software maintenance" were deleted, it would simply be a list of management duties for any other organizational function. Thus, it is imperative that a software maintenance manager be qualified both technically and managerially to hold such a position. If the person is not, the ability to be an effective maintenance

manager will be severely diminished.

Just as the importance of management skills has not been recognized in the selection of many software maintenance managers, in other instances the need for technical maintenance expertise has not been addressed. While many of the required skills involve dealing with and coordinating people, the software maintenance manager also has the responsibility to control the technical aspects of the process. Without a strong technical background and actual experience in performing software maintenance, the manager may not be able to deal with the conflicting needs and requirements of many maintenance tasks.

The software maintenance manager should be aware of, and familiar with, all of the work being performed by the software maintenance staff. While this is not always practical or possible in large organizations, each specific application system must have a central authority who is responsible for controlling and coordinating the maintenance of that system. Too often, a form of anarchy exists in software maintenance organizations. The maintainers are not adequately coordinated and are permitted to address problems as they arise without adhering to established standards and procedures. In the short term this may be the most effective manner of addressing immediate problems. The long term consequences, however, are usually a decreased level of maintainability for the system, and an increased need for maintenance. This section discusses standards, guidelines, procedures, and policies which will facilitate the management of the software maintenance function and will improve the capability to maintain application systems.

10.1 Goals Of Software Maintenance Management

The goal of software maintenance management is to keep all systems functioning and to respond to all user requests in a timely and satisfactory manner. Unfortunately, given the realities of staffing limitations, computer resource limitations, and the unlimited needs and desires of most users, this goal is very difficult to achieve. The realistic goal, then, is to keep the software maintenance process orderly and under control. The specific responsibility of the software maintenance manager is to keep all application systems running and to facilitate communication between the three groups involved with software maintenance.

The user must be kept satisfied that everything possible is being done to keep each system running as efficiently and productively as possible.

Table 11 - Goals of Software Maintenance

1. Keep the maintenance process orderly and under control.
2. Keep the application systems running.
3. Keep the users satisfied.
4. Keep the maintainers happy.
5. Keep maintenance viewed as a positive aspect of ADP - one which contributes to the meeting of the goals of the organization; not something that has to be done because the ADP staff just can't do it right the first time.

Upper management must be kept informed of the overall success of the software maintenance effort and how software maintenance supports and enhances the organization's ability to meet its objectives. In dealing with upper management, one of the primary responsibilities of the software maintenance manager is to keep maintenance viewed in a positive perspective. Software maintenance is an important effort which supports and contributes to the ability of the organization to meet its goals. Too many of the problems encountered in software maintenance are the result of the negative attitude that it is a function which exists because the software support staff can "never do it right". Rather, the emphasis should be on the concept that software maintenance enables an organization to improve and expand its capabilities using existing systems.

Finally, the software maintenance manager has the responsibility for keeping the maintenance staff happy and satisfied. Software maintenance must be thought of as the challenging, dynamic, interesting work it can be.

10.2 Establish a Software Maintenance Policy

A software maintenance policy should employ standards which describe in broad terms the responsibilities, authorities, functions, and operations of the software maintenance organization. It should be comprehensive enough to address

any type of change to the software system and its environment, including changes to the hardware, software and firmware. To be effective, the policy should be consistently applied and must be supported and promulgated by upper management to the extent that it establishes an organizational commitment to software maintenance. When supported by management, the standards and guidelines help to direct attention toward the need for greater discipline in software design, development, and maintenance.

The software maintenance policy must specifically address the need and justification for changes, the responsibility for making the changes, the change controls and procedures, and use of modern programming practices, techniques and tools. It should describe management's role and duties in regard to software maintenance and define the process and procedures for controlling changes to the software after the baseline has been established. (Baseline refers to a well-defined base or configuration to which all modifications are applied.) Implementation of the policy has the effect of enforcing adherence to rules regarding the operating software and documentation from initiation through completion of the requested change. Once this is accomplished, it is possible to establish the milestones necessary to measure software maintenance progress. Plans, however, are of little use if they are not followed. Reviews and audits are required to ensure that the plans are carried out.

The primary purpose of change control is to assure the continued smooth functioning of the application system and the orderly evolution of that system. The key to controlling changes to a system is the centralization of change approval and the formal requesting of changes. The software maintenance surveys found that each successful organization had a formal trouble report/change request process with a single person or a change review board approving all changes/enhancement requests prior to the scheduling of work. When this is not done, the confusion which results from independent maintenance efforts is usually disastrous.

Everything done to software affects its quality. Thus, measures should be established to aid in determining which category of changes are likely to degrade software quality. Care must also be taken to ensure that changes are not incompatible with the original system design and intent. The degree to which a change is needed and its anticipated use should be a major consideration. Consideration should also be given the cost/benefit of the change: "would a new system be less expensive and provide better capabilities?". The policies establishing change control should be clear, concise, well publicized, and strictly enforced.

Table 12 - Establishing a Software Maintenance Policy

1. Review and evaluate all requests for changes.
   - The change must be fully justified.
   - The impact on other work and users should be taken into consideration.

2. Plan for and schedule maintenance.
   - Each change request should be assigned a priority.
   - Work should be scheduled according to priority.
   - The scheduled should be enforced and adhered to.

3. Restrict code changes to the approved/scheduled work.

4. Enforce documentation and coding standards through reviews and audits.

10.2.1 Review and evaluate all requests for changes

All user and staff requests for changes to an application system (whether enhancements, preventive maintenance, or errors) should be requested in writing and submitted to the software maintenance manager. Each change request should include not only the description of the requested change, but a full justification of why that change should be made. These change requests should be carefully reviewed and evaluated before any actual work is performed on the system. The evaluation should take into consideration, among other things, the staff resources available versus the estimated workload of the request; the estimated additional computing resources which will be required for the design, test, debug and operation of the modified system; and the time and cost of updating the documentation. Of course, some flexibility must be built into the process with some delegation of authority to initiate critical tasks. However, each request should be reviewed and judged by either the software maintenance manager or a change review board. Doing so will reduce the amount of unnecessary and/or unjustified work which is often performed on a system.

10.2.2 Plan for, and schedule maintenance

The result of the review of all change requests should be the assignment of a priority to each request and the updating of a schedule for meeting those requests. In many ADP organizations, there is simply more work requests than staff resources to meet those requests. Therefore, all work should be scheduled and every effort made to adhere to the schedule rather than constantly changing course in response to the most visible crisis.

10.2.3 Restrict code changes to the approved work

In many cases, especially when the code was poorly designed and/or written, there is a strong temptation to change other sections of the code as long as the program has been "opened up". The software maintenance manager must monitor the work of the software maintenance staff, and ensure that only the authorized work is performed. In order to monitor maintenance effectively, all activities must be documented. This includes everything from the change request form to the final revised source program listing.

Permitting software maintenance staff to make changes other than those authorized can cause schedules to slip and may prevent other, higher priority work from being completed on time. It is very difficult to limit the work which is done on a specific program, but it is imperative to the overall success of the maintenance function to do so.

10.2.4 Enforce documentation and coding standards

Some programmers do not like to document, some are not good at it, but primarily, documentation suffers because of too much pressure and too little time in the schedule to do it. Proper and complete communication of necessary information between all persons who have, are currently, and who will work on the system is essential. The most important media for this communication is the documentation and the source code.

It is not enough to simply establish standards for coding and documentation. Those standards must be continually enforced via technical review and examination of all work performed by the software maintenance staff. In scheduling maintenance, sufficient time should be provided to fully update the documentation and to satisfy established standards and guidelines before a new assignment is begun.

10.3 Staffing And Management Of Maintenance Personnel

Selecting the proper staff for a software maintenance project is as important as the techniques and approaches employed. There is some debate on whether or not an organization should have separate staffs for maintenance and development. Many managers have indicated that separate staffs can improve the effectiveness of both. However, the realities of size, organization, budget, and staff ceilings often preclude the establishment of separate maintenance and development staffs.

Management must apply the same criteria to the maintainers that are applied to software and systems designers or other highly sought after professional positions. If an individual is productive, consistently performs well, has a good attitude, and displays initiative, it should not matter whether the project is development or maintenance. Recent studies on the motivation of programmers and analysts [COUG82] indicate that there are three major psychological factors that can impact the attitude, morale, and general performance of an individual.

- the work must be considered worthwhile by a set of values accepted by the individual, as well as by the standards employed by the organization.

- the individual must feel a responsibility for his or her performance. There is a need to feel personally accountable for the outcome of an effort.

- the individual must be able to determine on a regular basis whether or not the outcome of his or her efforts is satisfactory.

When these factors are high, the individual is likely to have a good attitude and be motivated.

Some organizations have attempted to improve morale and the image of maintenance by simply renaming the maintenance function. This is a superficial approach. It does nothing to change what is in fact being done, or the way it is perceived by the maintainer and supported by management. A more positive approach is to acknowledge the importance and value of good maintenance to the organization through career opportunities, recognition, and compensation.

Often, a maintainer is responsible for large amounts of code, much of which was developed and previously maintained by someone else. This code is generally old, unstructured, has received numerous patches, and is inadequately documented. The potential for errors, delays, and unhappy users is considerable. Praise, thanks and recognition are often as

important as salary and challenging assignments in keeping good analysts and programmers.

It is essential that work assignments offer growth potential. Continuing education is required at all levels to ensure that not only the maintainers, but the users, managers, and operators have a thorough understanding of software maintenance. Training should include: programming languages, standards and guidelines, operating systems, and utilities.

There is a common misperception that maintenance has to be dull, tedious, non-creative work which offers little chance for reward or advancement. This view can only be changed through management initiatives. The maintainer is a critical part of the process -- the key to delivery of the product both promised by management and desired by the users. Indeed, the maintainer is one of the most important members of the application software staff. The importance of maintenance must be acknowledged in terms of both position value and function.

Some points to keep in mind when managing a software maintenance function are outlined in Table 13.

Table 13 - Managing the Software Maintenance Function

1. Maintenance is as important as development and just as difficult and challenging.

2. Maintainers should be highly qualified, competent, dedicated professionals. The staff should include both senior and junior personnel. Do not short change maintenance. Don't isolate the maintenance staff.

3. Maintenance should NOT be used as a training ground where junior staff are left to "sink-or-swim".

4. Staff members should be rotated so they are assigned to both maintenance and development. It takes a good developer to be a good maintainer, and conversely, it takes a good maintainer to be a good developer.

5. Good maintenance performance and good development performance should be equally rewarded.

6. There should be an emphasis on keeping the staff well trained. This will keep performance at an optimum level and help to minimize morale problems.

7. Rotate assignments. Do not permit a system or a major part of a system to become someone's private domain.

11.0 SUMMARY

While the ICST survey identified software maintenance problems which were both managerial and technical in nature, management is clearly the most important factor in improving the software maintenance process. Most of the problems cited in the survey were the result of inadequate management control and review of software maintenance activities. Management must take a closer look at how the software is maintained, exercise better control over the process, and ensure that effective software maintenance techniques and tools are employed.

Recommendations have been made in sections 7.0 through 10.0 of this report to help a manager gain better control, and to help the maintainer improve the quality of the maintenance performed. In order to maintain control over the software maintenance process and to ensure that the maintainability of the system does not deteriorate, it is important that software maintenance be anticipated and planned for.

The quality and maintainability of a software system often decrease as the system grows older. This is the result of many factors which, taken one at a time, may not seem significant but become cumulative and often result in a system which is very difficult to maintain. Quality programming capabilities and techniques are readily available. However, until a firm discipline is placed on how software maintenance is performed, and that discipline is enforced, many systems will be permitted to deteriorate to the point where they are impossible to maintain.

Software maintenance must be performed in a structured, controlled manner. It is simply not enough to get a system "up and running" after it breaks. Proper management control must be exercised over the entire process. In addition to controlling the budget, schedule, and staff, it is essential that the software maintenance manager control the system and the changes to it. The now frequently cited maxim that a system "must be developed with maintenance in mind" is insufficient; a system also must be maintained with future maintenance in mind. If this is done, the quality and maintainability of the code actually can improve. Otherwise, today's maintainable systems are destined to become tomorrow's unmaintainable systems.

BIBLIOGRAPHY

[ARTH83] L.J.Arthur, Programming Productivity, John Wiley and Sons, New York, 1983.

[BASI82] V.R.Basili and H.D.Mills, "Understanding and Documenting Programs," IEEE Transactions on Software Engineering, Vol SE-8, No 3, May 1982, pp 270-283.

[BERS79] E.H.Bersoff, V.D.Henderson, and S.G.Liegel, "Software Configuration Management: A Tutorial," Computer, January 1979, pp 6-14.

[BOEH78] B.W.Boehm, J.R.Brown, H.Kasper, M.Lipow, G.J.MacLeod, and M.J.Merritt, Characteristics of Software Quality, North-Holland, Amsterdam-New York-Oxford, 1978.

[BOEH81] B.W.Boehm, "An Experiment in Small-Scale Application Software Engineering," IEEE Transactions on Software Engineering, Vol SE-7, No 5, September 1981, pp 482-493.

[BOEH82] B.W.Boehm, Software Engineering Economics, Prentice-Hall, Englewood Cliffs, 1982.

[BRIC83] L.Brice and J.Connell, "A Methodology for Minimizing Maintenance Costs," AFIPS 1983 National Computer Conference Proceedings, AFIPS Press, Arlington, Virginia, May 1983, pp 113-121.

[BROO75] F.P.Brooks, The Mythical Man Month, Addison-Wesley, Reading, Massachusetts, 1975.

[BUCK77] J.K.Buckle, Managing Software Projects, MacDonald and Jane's, London and American Elsevier Inc, New York, 1977.

[CENT82] J.W.Center, "A Quality Assurance Program For Software Maintenance," AFIPS 1982 National Computer Conference Proceedings, AFIPS Press, Arlington, Virginia, May 1982, pp 399-407.

[CHAP83] N.Chapin, "Software Maintenance Objectives," AFIPS 1983 National Computer Conference Proceedings, AFIPS Press, Arlington, Virginia, May 1983, pp 779-784.

[COOP79] J.D.Cooper and M.J.Fisher, editors, Software Quality Management, Petrocelli Books Inc., 1979.

[COUG82] D.J.Couger and M.A.Colter, "Effect of Task Assignments on Motivation of Programmers and Analysts," research report, University of Colorado, 1982.

[CURT79] B.Curtis, S.Sheppard, P.Milliman, M.A.Vorst, T.Love, "Measuring The Psychological Complexity of Software Maintenance Tasks With the Halstead and McCabe Metrics," IEEE Transactions on Software Engineering, Vol SE-5, No 2, March 1979, pp 96-103.

[DITR71] A.E.Ditri, J.C.Shaw, and W.Atkins, Managing the EDP Function, McGraw Hill, New York, 1971.

[DONA80] J.D.Donahoo and D.Swearinger, "A Review of Software Maintenance Technology," Rome Air Development Center, RADC-TR-80-13, February 1980.

[EBER80] R.Ebert, J.Lugger, and R.Goeke, editors, Practice in Software Adaption and Maintenance, North-Holland, New York, 1980.

[ELSH82] J.L.Elshoff and M.Marcotty, "Improving Program Reliability to Aid Modification," CACM, Vol 25, No 8, August 1982, pp 512-521.

[FIPS38] "Guidelines for Documentation of Computer Programs and Automated Data Systems," NBS Federal Information Processing Standards Publication 38, February 1976.

[FIPS64] "Guidelines for Documentation of Computer Programs and Automated Data Systems for the Initiation Phase," NBS Federal Information Processing Standards Publication 64, August 1979.

[FIPS101] "Guideline for Lifecycle Validation, Verification, and Testing of Computer Software," NBS Federal Information Processing Standards Publication 101, June 1983.

[FRAN82] W.L.Frank, Critical Issues In Software, John Wiley and Sons, New York, 1982.

[FREE80] H.Freeman and P.M.Lewis, editors, Software Engineering, Academic Press, New York, 1980

[GAO80] "Wider Use Of Better Computer Software Technology Can Improve Management Control And Reduce Costs," Comptroller General Report to Congress of the United States, FGMSD-80-38, April 29 1980.

[GAO81a] "Goverment-Wide Guidelines And Management Assistance Center Needed To Improve ADP Systems Development," Report by the U.S. General Accounting Office, AFMD-81-20, February 20, 1981.

[GAO81b] "Federal Agencies' Maintenance Of Computer Programs: Expensive And Undermanaged," Comptroller General Report to Congress of the United States, AFMD-81-25, February 26, 1981.

[GLAS79] R.L.Glass, Software Reliability Guidebook, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

[GLAS81a] R.L.Glass and R.A.Noiseux, Software Maintenance Guidebook, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[GLAS81b] R.L.Glass, "Persistent Software Errors," IEEE Transactions on Software Engineering Vol SE-7, No 2, March 1981.

[GLAS82] R.L.Glass, Modern Programming Practices: A Report From Industry, Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[GREE81] J.F.Green, et al, "Dynamic Planning and Software Maintenance - A Fiscal Approach," Naval Post Graduate School, Dept. of Commerce, NTIS, 1981.

[HALS77] M.H.Halstead, Elements of Software Science, Elsevier Science Publishing Company, New York, 1977.

[HAML79] W.T.Hamlen, "Application Program Maintenance Study - Report to Guide," Proceedings of Guide 48, May 1979, pp 1751-1758.

[HURL82] R.B.Hurley, Decision Tables in Software Engineering, Van Nostrand Reinhold, New York, 1982.

[JENS79] R.W.Jensen and C.C.Tonies, Software Engineering, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

[JONE78a] R.A.Jones, "Maintenance Considered Harmful," ACM Forum, CACM, Vol 21, No 10, October 1978, p 882.

[LEHM77] M.M.Lehman, "Evolution Dynamics - A Phenomenology of Software Maintenance," Proceedings of Software, Life Cycle Management Workshop, August 1977, pp 313-323.

[LIEN78] B.P.Lientz, E.B.Swanson, and G.E.Tompkins, "Characteristics of Application Software Maintenance," CACM, Vol 21, No 6, June 1978, pp 466-471.

[LIEN79] B.P.Lientz and E.B.Swanson, "Software Maintenance - A User/Management Tug-of-War," Data Management, April 1979, pp 26-30.

[LIEN80] B.P.Lientz and E.B.Swanson, Software Maintenance Management, Addison-Wesley, Reading, Massachusetts, 1980.

[LIEN81] B.P.Lientz and E.B.Swanson, "Problems in Application Software Maintenance," CACM, Vol 24, No 11, November 1981, pp 763-769.

[LYON81] M.L.Lyons, "Salvaging Your Software Asset (Tools Based Maintenance)", AFIPS 1981 National Computer Conference Proceedings, AFIPS Press, Arlington, Virginia, May 1981, pp 337-342.

[MARS83] N.L.Marselos, "Human Investment Techniques for Effective Software Maintenance," AFIPS 1983 National Computer Conference Proceedings, AFIPS Press, Arlington, Virginia, May 1983, pp 131-136.

[MARSH83] R.E.Marsh, "Application Maintenance: One Shop's Experience and Organization," AFIPS 1983 National Computer Conference Proceedings, AFIPS Press, Arlington, Virginia, May 1983, pp 145-153.

[MART83] J.Martin, C.McClure, Software Maintenance - The Problem and Its Solutions, Prentice Hall, Englewood Cliffs, New Jersey, 1983.

[MART82] J.Martin, Application Development Without Programmers, Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[MCCL81] C.L.McClure, Managing Software Development and Maintenance, Van Nostrand Reinhold, New York, 1981.

[MILL79] E.Miller, Tutorial : Automated Tools for Software Engineering, IEEE Computer Society Press, Silver Spring, Maryland, 1979.

[MILL83] H.D.Mills, Software Productivity, Little Brown and Co, 1983.

[MUNS81] J.B.Munson, "Software Maintainability: A Practical Concern for Life-Cycle Costs," Computer, Vol 14, Nov 1981, pp 103-109.

[MYER76] G.J.Myers, Software Reliability: Priciples and Practices, John Wiley and Sons, New York, 1976.

[MYER79] G.J.Myers, The Art of Software Testing, John Wiley and Sons, New York, 1979.

[NAVE79] "Computer Software Life Cycle Management Guide," Naval Electronics Systems Command, NAVELEXINST 5200.23, March 1979.

[NBS75] W.R.Adrion, M.A.Branstad, and J.C.Cherniavsky, "Validation, Verification and Testing of Computer Software," NBS Special Publication 500-75, February 1981.

[NBS87] A.J.Neumann, "Management Guide For Software Documentation," NBS Special Publication 500-87, January 1982.

[NBS93] P.B.Powell, editor, "Software Validation, Verification and Testing Technique and Tool Reference Guide," NBS Special Publication 500-93, September 1982.

[NBS98] P.B.Powell, editor, "Planning For Software Validation, Verification and Testing," NBS Special Publication 500-98, November 1982.

[PARI83] G.Parikh, N.Zvegintzov, Tutorial on Software Maintenance, IEEE Computer Society Press, Silver Spring, Maryland, 1983.

[PARI80] G.Parikh, editor, Techniques of Program and System Maintenance, Ethnotech, Lincoln, Nebraska, 1980.

[PEER81] D.E.Peercy, "A Software Maintainability Evaluaition Methodology," IEEE Transactions On Software Engineering, Vol SE-7, No 4, July 1981, pp 343-351.

[PENN80] R.H.Pennington, "Software Development and Maintenance - Where Are WE?," Proceedings COMPSAC80, IEEE Computer Society's Fourth International Computer Software and Application Conference, 1980, pp 419-422.

[PERR81] W.E.Perry, Managing System Maintenance, Q.E.D. Information Sciences, Inc., Wellesley, Massachusetts, 1981.

[PRES82] R.Pressman, Software Engineering: A Practioner's Approach, McGraw Hill, New York, 1982.

[RAYN83] R.J.Raynor and L.D.Speckmann, "Maintaining User Participation Throughout the Systems Development Cycle," AFIPS 1983 National Computer Conference Proceedings, AFIPS Press, Arlington, Virginia, May 1983, pp 173-180.

[REIF77] D.J.Reifer and S.Trattner, "A Glossary of Software Tools and Techniques," Computer, Vol 10, No 7, July 1977, pp 52-60.

[RICH83] G.L.Richardson and C.W.Butler, "Organizational Issues of Effective Maintenance Management," AFIPS 1983 National Computer Conference Proceedings, AFIPS Press, Arlington, Virginia, May 1983, pp 155-161.

[SCHN79] N.F.Schneidewind, H.M.Hoffman, "An Experiment In Software Error Data Collection And Analysis," IEEE Transactions on Software Engineering, Vol SE-5, No 3, May 1979, pp 276-286.

[SCHN83] G.R.Schneider, "Structured Software Maintenance," AFIPS 1983 National Computer Conference Proceedings, AFIPS Press, Arlington, Virginia, May 1983, pp 137-144.

[SHNE80] B.Shneiderman, Software Psychology, Winthrop Publishers, 1980.

[SWAN76] E.B.Swanson, "The Dimensions of Software Maintenance", IEEE Computer Society, Proceedings of the 2nd International Conference on Software Engineering, October 1976, pp 492-497.

[TAUT83] B.J.Taute, "Quality Assurance and Maintenance Application Systems," AFIPS 1983 National Computer Conference Proceedings, AFIPS Press, Arlington, Virginia, May 1983, pp 123-129.

[THAY81] R.H.Thayer, A.B.Pyster, and R.C.Wood, "Major Issues in Software Engineering Project Management," IEEE Transactions on Software Engineering, Vol SE-7, No 4, July 1981, pp 333-342.

[TINN83] P.C.Tinnirello, "Improving Software Maintenance Attitudes," AFIPS 1983 National Computer Conference Proceedings, AFIPS Press, Arlington, Virginia, May 1983, pp 107-112.

[WALK81] M.G.Walker, Managing Software Reliability - The Paradigmatic Approach, North Holland, New York, 1981

[WEIN72] G.M.Weinberg, The Psychology of Computer Programming, Van Nostrand Reinhold, New York, 1972.

[YAU78] S.S.Yau, J.S.Collofello, and T.MacGregor, "Ripple Effect Analysis of Software Maintenance," IEEE Proceedings of COMPSAC 78, 1978, pp 60-65.

[ZAK83] J.R.Zak, "When a Data Processing Department Inherits Software," AFIPS 1983 National Computer Conference Proceedings, AFIPS Press, Arlington, Virginia, May 1983, pp 163-172.

[ZELK78] M.V.Zelkowitz, "Perspectives on Software Engineering," Computing Surveys, Vol 10, No 2, June 1978, pp 197-216.

[ZELL83] L.Zells, "Data Processing Project Management: A Practical Approach for Publishing a Project Expectations Document," AFIPS 1983 National Computer Conference Proceedings, AFIPS Press, Arlington, Virginia, May 1983, pp 181-187.

[ZVEG83] N.Zvegintzov, "Nanotrends," Datamation, August 1983, pp 106-116.

APPENDIX I

Software Maintenance Definitions

"Software maintenance in its broadest sense, includes error corrections, changes(also called modifications or amendments), enhancements, and improvements to the existing software. It includes maintenance of all software, including structured (software developed using structured technologies) and unstructured software (software developed without...)."

Girish Parikh,"World of Software Maintenance"
Techniques of Program and System Maintenance,
1981

Maintenance is "the process of modifying existing operational software while leaving its primary functions intact."

Barry Boehm, "Software Maintenance",
IEEE Transactions on Software Engineering,
December, 1976.

"Maintenance is the continuing process of keeping the program running, or improving its charateristics"

J.L. Odgen, "Designing Reliable Software,"
reprinted in [PARI81]

"Most generally, it is the process of adaption, i.e., updating existing systems functions to reflect new constraints or additional features."

Chester Liu, "A Look At Software Maintenance",
reprinted in [PARI81]

"Traditionally, program maintenance has been viewed as a second class activity, with an admixture of on-the-job training for beginners and of low-status assignments for the outcasts and the fallen.

Richard Gunderman, "A Glimpse into Program
Maintenance", reprinted in [PARI81]

"Maintenance is the process of being responsive to user needs - fixing errors, making user-specified modifications, honing the program to be more useful."

"Software maintenance....is the act of taking a software product that has already been delivered to a customer and is in use by him, and keeping it functioning in a satisfactory way."

R.L.Glass and R.A.Noiseux, Software Maintenance Guidebook, 1981.

"Systems maintenance includes any activity needed to ensure that application programs remain in satisfactory working condition."

W.E.Perry, Managing Systems Maintenance, 1981.

"...changes that have to be made to computer programs after they have been delivered to the customer or user."

James Martin and Carma McClure, Software Maintenance - The Problem and Its Solution, 1983.

"The maintenance of software includes two major activities - the removal of defects and the enhancement of operations."

Werner S. Frank, Critical Issues in Software - A Guide to Software Economics, Strategy, and Profitability, 1983.

NBS-114A (REV. 2-8C)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET (See instructions)	1. PUBLICATION OR REPORT NO. NBS SP 500-106	2. Performing Organ. Report No.	3. Publication Date December 1983

4. TITLE AND SUBTITLE Computer Science and Technology:

Guidance on Software Maintenance

5. AUTHOR(S)

Roger J. Martin and Wilma M. Osborne

6. PERFORMING ORGANIZATION (If joint or other than NBS, see instructions)

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covered

Final

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (Street, City, State, ZIP)

National Bureau of Standards
Department of Commerce
Washington, DC 20234

10. SUPPLEMENTARY NOTES

Library of Congress Catalog Card Number: 83-600611

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)

This report addresses issues and problems of software maintenance and suggests actions and procedures which can help software maintenance organizations meet the growing demands of maintaining existing systems. The report establishes a working definition for software maintenance and presents an overview of current problems and issues in that area. Tools and techniques that may be used to improve the control of software maintenance activities and the productivity of a software maintenance organization are discussed. Emphasis is placed on the need for strong, effective technical management control of the software maintenance process.

12. KEY WORDS (Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)

adaptive maintenance; corrective maintenance; management; perfective maintenance; software engineering; software maintenance; software maintenance management; software maintenance tools.

13. AVAILABILITY

☒ Unlimited

☐ For Official Distribution. Do Not Release to NTIS

☒ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

☐ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

74

15. Price

ANNOUNCEMENT OF NEW PUBLICATIONS ON COMPUTER SCIENCE & TECHNOLOGY

Superintendent of Documents,
Government Printing Office,
Washington, DC 20402

Dear Sir:

Please add my name to the announcement list of new publications to be issued in the series: National Bureau of Standards Special Publication 500-.

Name _____

Company _____

Address _____

City _____ State _____ Zip Code _____

(Notification key N-503)

NBS TECHNICAL PUBLICATIONS

PERIODICALS

JOURNAL OF RESEARCH—The Journal of Research of the National Bureau of Standards reports NBS research and development in those disciplines of the physical and engineering sciences in which the Bureau is active. These include physics, chemistry, engineering, mathematics, and computer sciences. Papers cover a broad range of subjects, with major emphasis on measurement methodology and the basic technology underlying standardization. Also included from time to time are survey articles on topics closely related to the Bureau's technical and scientific programs. As a special service to subscribers each issue contains complete citations to all recent Bureau publications in both NBS and non-NBS media. Issued six times a year. Annual subscription: domestic $18; foreign $22.50. Single copy, $5.50 domestic; $6.90 foreign.

NONPERIODICALS

Monographs—Major contributions to the technical literature on various subjects related to the Bureau's scientific and technical activities.

Handbooks—Recommended codes of engineering and industrial practice (including safety codes) developed in cooperation with interested industries, professional organizations, and regulatory bodies.

Special Publications—Include proceedings of conferences sponsored by NBS, NBS annual reports, and other special publications appropriate to this grouping such as wall charts, pocket cards, and bibliographies.

Applied Mathematics Series—Mathematical tables, manuals, and studies of special interest to physicists, engineers, chemists, biologists, mathematicians, computer programmers, and others engaged in scientific and technical work.

National Standard Reference Data Series—Provides quantitative data on the physical and chemical properties of materials, compiled from the world's literature and critically evaluated. Developed under a worldwide program coordinated by NBS under the authority of the National Standard Data Act (Public Law 90-396).

NOTE: The principal publication outlet for the foregoing data is the Journal of Physical and Chemical Reference Data (JPCRD) published quarterly for NBS by the American Chemical Society (ACS) and the American Institute of Physics (AIP). Subscriptions, reprints, and supplements available from ACS, 1155 Sixteenth St., NW, Washington, DC 20056.

Building Science Series—Disseminates technical information developed at the Bureau on building materials, components, systems, and whole structures. The series presents research results, test methods, and performance criteria related to the structural and environmental functions and the durability and safety characteristics of building elements and systems.

Technical Notes—Studies or reports which are complete in themselves but restrictive in their treatment of a subject. Analogous to monographs but not so comprehensive in scope or definitive in treatment of the subject area. Often serve as a vehicle for final reports of work performed at NBS under the sponsorship of other government agencies.

Voluntary Product Standards—Developed under procedures published by the Department of Commerce in Part 10, Title 15, of the Code of Federal Regulations. The standards establish nationally recognized requirements for products, and provide all concerned interests with a basis for common understanding of the characteristics of the products. NBS administers this program as a supplement to the activities of the private sector standardizing organizations.

Consumer Information Series—Practical information, based on NBS research and experience, covering areas of interest to the consumer. Easily understandable language and illustrations provide useful background knowledge for shopping in today's technological marketplace.

Order the **above** *NBS publications from: Superintendent of Documents, Government Printing Office, Washington, DC 20402.*

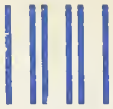
Order the **following** *NBS publications—FIPS and NBSIR's—from the National Technical Information Service, Springfield, VA 22161.*

Federal Information Processing Standards Publications (FIPS PUB)—Publications in this series collectively constitute the Federal Information Processing Standards Register. The Register serves as the official source of information in the Federal Government regarding standards issued by NBS pursuant to the Federal Property and Administrative Services Act of 1949 as amended, Public Law 89-306 (79 Stat. 1127), and as implemented by Executive Order 11717 (38 FR 12315, dated May 11, 1973) and Part 6 of Title 15 CFR (Code of Federal Regulations).

NBS Interagency Reports (NBSIR)—A special series of interim or final reports on work performed by NBS for outside sponsors (both government and non-government). In general, initial distribution is handled by the sponsor; public distribution is by the National Technical Information Service, Springfield, VA 22161, in paper copy or microfiche form.

U.S. Department of Commerce
National Bureau of Standards

Washington, D.C 20234
Official Business
Penalty for Private Use $300



POSTAGE AND FEES PAID
U.S. DEPARTMENT OF COMMERCE
COM-215

SPECIAL FOURTH-CLASS RATE
BOOK