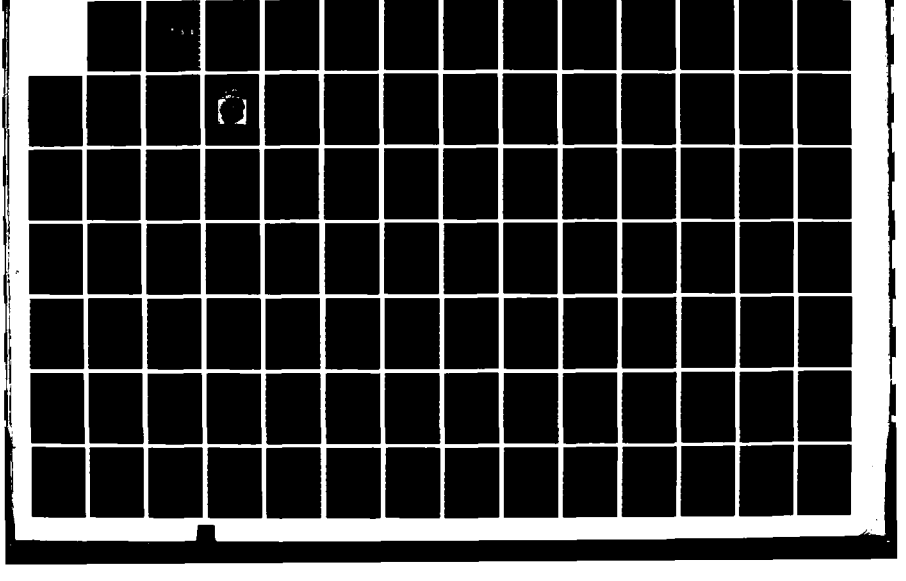
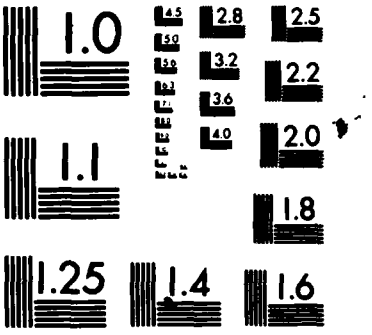
AD-A171 391 CONTROL AND MANAGEMENT OF THE SOFTWARE MAINTENANCE CHANGES PROCESS(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA N A AL-SUBAIEI JUN 86 1/2

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

CONTROL AND MANAGEMENT OF THE SOFTWARE MAINTENANCE CHANGES PROCESS

by

Nasser A. Al-Subaiei

June 1986

Thesis Advisor: Gordon H. Bradley

Approved for public release; distribution is unlimited

SECURITY CLASSIFICATION OF THIS PAGE

AD-A171 391

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (If applicable) 52	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
CONTROL AND MANAGEMENT OF THE SOFTWARE MAINTENANCE CHANGES PROCESS

12 PERSONAL AUTHOR(S)
Al-Subaiei, Nasser A.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM _____ TO _____	1986 June	101

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Software Maintenance, Control of Maintenance Change, Change Control Tools, Software Configuration Management

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The cost of software maintenance is very high and projected to climb higher in the future. Failure to adopt and utilize improved technical and management methods and tools contributes to the high cost and burden of maintenance. Software configuration management as an effective technique of controlling software development/maintenance is examined. Two change control models are identified and evaluated as to their effectiveness and completeness toward achieving efficient control and easing maintenance effort. A proposed change control model which addresses more aspects and promises better results through a set of guidelines for an "ideal" software maintenance change control is presented. Software maintenance change control tools are discussed by identifying two of the existing tools. With proper
(continued)

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED	
22a NAME OF RESPONSIBLE INDIVIDUAL Professor Gordon H. Bradley	22b TELEPHONE (Include Area Code) (408) 646-2359	22c OFFICE SYMBOL Code 52Bz

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

19. (Continued)

implementation of the proposed change control model and the use of an effective change control tool, better control of the maintenance process can be achieved, and the maintenance effort reduced.

Approved for public release; distribution is unlimited

Control and Management of the Software Maintenance Changes Process

by

Nasser A. Al-Subaiei
Captain, Royal Saudi Arabia Aire Defense Forces
B.S.E.E., Arizona State University, 1978

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1986

Author: _____
Nasser A. Al-Subaiei

Approved by: _____
Gordon H. Bradley, Thesis Advisor

_____
Bruce J. MacLennan, Second Reader

_____
V. Y. Lum, Chairman, Department of Computer Science

_____
Kneale T. Marshall, Dean of Information and Policy Sciences

ABSTRACT

The cost of software maintenance is very high and projected to climb higher in the future. Failure to adopt and utilize improved technical and management methods and tools contributes to the high cost and burden of maintenance. Software configuration management as an effective technique of controlling software development/ maintenance is examined. Two change control models are identified and evaluated as to their effectiveness and completeness toward achieving efficient control and easing maintenance effort. A proposed change control model which addresses more aspects and promises better results through a set of guidelines for an "ideal" software maintenance change control is presented. Software maintenance change control tools are discussed by identifying two of the existing tools. With proper implementation of the proposed change control model and the use of an effective change control tool, better control of the maintenance process can be achieved, and the maintenance effort reduced.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

The author would like to express his appreciation and gratitude to the Royal Saudi Air Defense Forces Command for giving him a scholarship and allowing him this opportunity to come to the United States to gain more knowledge.

The author owes very special thanks to his thesis advisor, Professor Gordon H. Bradley, for his continued inspiration, guidance, and warm friendship.

Thanks are also extended to my second reader, Professor Bruce J. MacLennan, for his valuable suggestions and reading of the final draft. Lastly, but certainly not the least, the author is grateful to his parents for their early support and especially grateful to his wife, Nuwair, and his children, Monerah, Ream, Abdullah and Hifa, for their understanding, encouragement, support, and patience throughout his studies.

I. INTRODUCTION

A. THE PROBLEM

Software maintenance is the most expensive phase of the life cycle of software systems. It has been indicated that in some systems up to eighty percent of the cost of software systems is consumed in the maintenance phase of the software life cycle [Ref. 1]. In order to properly maintain the software it has to be properly controlled. Control is considered by Swanson [Ref. 2] as a major problem in software maintenance. Software maintenance control requires understanding of the software system involved, the user of this system, and how the system interacts with the user's environment. Assuring that only complete, accurate, and authorized data is changed requires the implementation of methods and techniques which lead to achieving an effective control over the maintenance change process.

B. PURPOSE AND APPROACH OF THESIS

There is a lack of a cohesive discussion in the current literature concerning proper control for effective software maintenance. Because of the length of time of the maintenance phase (15 years in some military systems) and the tremendous cost, an attempt to ease the maintenance effort needs to be made through the use of effective

control. Software configuration management and change control provide a maintenance environment with high management visibility and control.

The purpose of this thesis is to address control as a method of managing the maintenance change process. Various types of change control models are discussed and evaluated as to their effectiveness in easing the maintenance effort. An attempt is made to determine the proper type of change control needed to effectively maintain software projects. The concept of a change control model is put forth as a method for organizing and controlling the maintenance process. The idea is to give the maintainer effective policies and procedures to follow when a request for a maintenance change is received. Not having these policies and procedures will result in an uncontrolled process which will affect the integrity and quality of the system.

Chapter I gives the overview of the control problem as it relates to software maintenance. A description of the approach taken by the thesis is given along with some general definitions of terms used in the software maintenance environment. Also, the idea of a controlled change process is introduced in this chapter.

Chapter II discusses software engineering with a look at the software life cycle. This chapter discusses software maintenance in detail, its life cycle, types and problems.

Chapter III introduces the idea of control, its objectives, and how it might influence people's behavior. Also the need to control software maintenance is discussed.

Chapter IV introduces the concept of configuration management and how it can be tailored to software. Also, the four elements of software configuration management are discussed in this chapter.

Chapter V discusses and evaluates two change control models in a software maintenance environment. Each model is considered for its effectiveness and completeness.

Chapter VI synthesizes a new change control model which is based on combining the good aspects of the two previous models plus some additional modifications to develop a better, more effective change control model which can be used for large software projects. A subset of this change control model can be used by smaller organizations who are dealing with small software projects.

Chapter VII introduces software change control tools which, when used, provide better management and control over the maintenance process which leads to a savings in time and money. Also some other useful software maintenance tools are presented. These tools, when utilized during the maintenance phase, will result in better control, improve productivity and lead to savings in time and money.

Chapter VIII consists of conclusions of the thesis and recommendations.

Definitions of the critical concepts of software maintenance and software life cycle are readily available in the literature. Martin and McClure [Ref. 3] contains a good definition of software maintenance, its problems and solutions. The software life cycle model was developed from Boehm [Ref. 4]. Bersoff and Buckel provides excellent guidance and definitions for software configuration management [Ref. 5], [Ref. 6]. The definitions of software change control tools were found in articles by Rochkind and Tichy [Ref. 7], [Ref. 8].

For the purposes of this thesis, software will be considered to be the programs and related documentations. Controlling the process of software maintenance is keeping things on track and heading toward an objective through several means including policy, procedures, and tools.

II. SOFTWARE MAINTENANCE

A. INTRODUCTION

The problem with software maintenance is that few people seem to understand it or how to deal with it. Software maintenance has received far less study or concern than other software engineering topics, such as development or design. Even college curricula contain little about maintenance or its techniques. It has been estimated that more resources are required to maintain existing systems than to develop new ones; the estimate is that 60% to 80% of the total application software resources are spent on software maintenance [Ref. 9]. Maintenance dominates the software life cycle in terms of effort and cost (Fig. 2.1) [Ref. 3].

B. SOFTWARE ENGINEERING AND ITS FUNCTIONS

Software engineering simply defined is a collection of methodologies, both technical and managerial, for development and maintenance of software. The field of software engineering includes technical as well as managerial functions for the equally important functions of software development and software maintenace [Ref. 10] as shown in Figure 2.2.

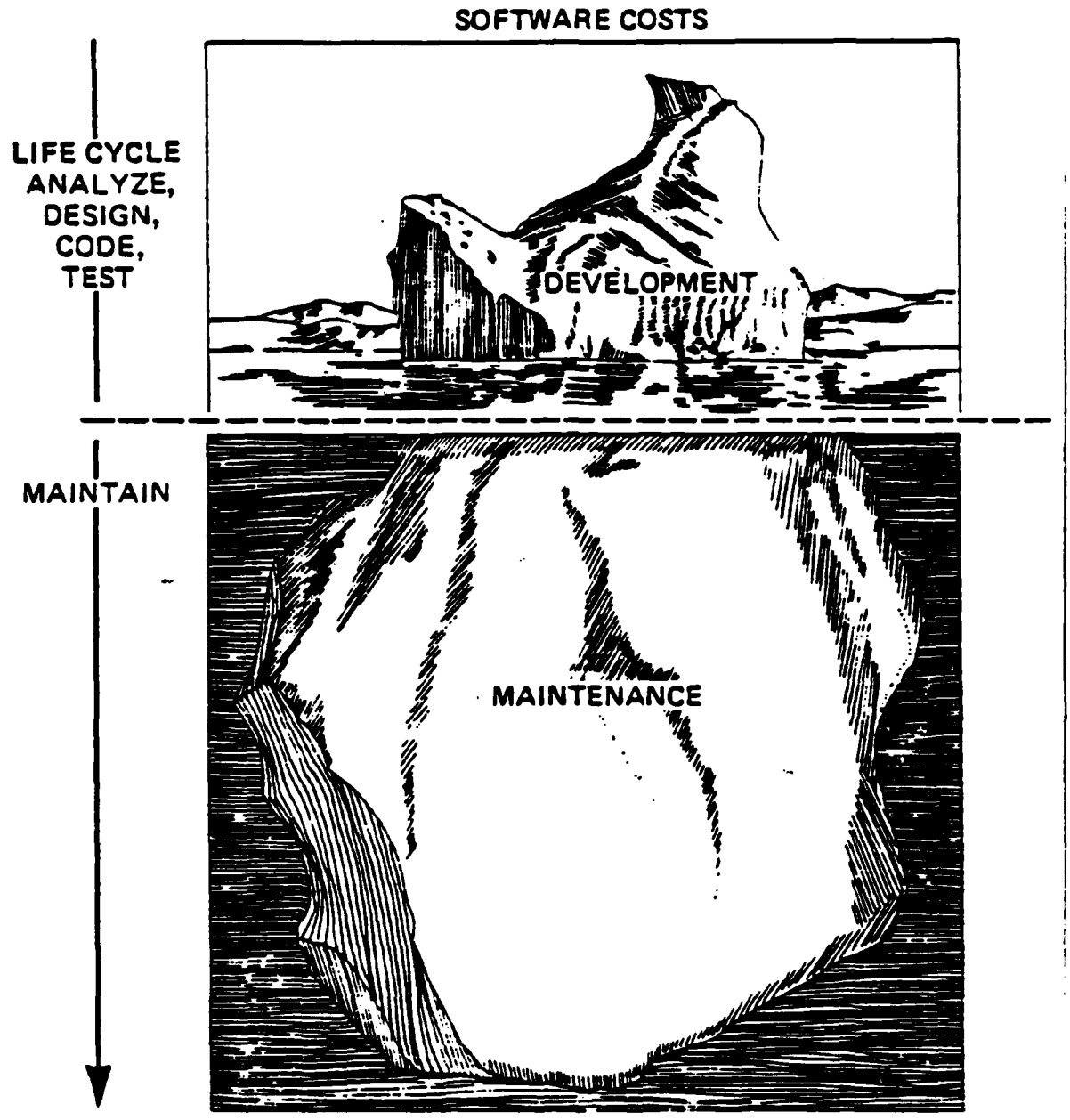


Figure 2.1 The Hidden Cost of Maintenance

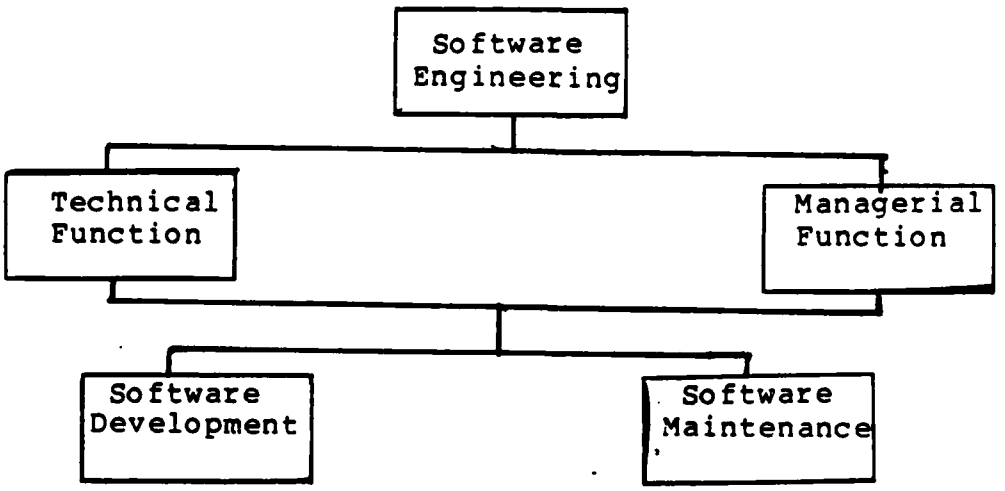


Figure 2.2 The Functions of Software Engineering

C. SOFTWARE LIFE CYCLE

The development of a software project goes through several phases. We define the software life cycle as a multiple process beginning with problem definition and continuing to software obsolescence.

Typically the life cycle is defined as:

1) Requirements analysis
2) Specification
3) Design
4) Code
5) Testing
6) Operation and Maintenance [Ref. 11].

The major problem with this model is the implication concerning the flow of the software life cycle. One is left with the idea that as one phase abruptly halts, the next

phase begins. In practice, the phase boundaries are somewhat obscure. Quite often work on one part of a phase begins before all work in a previous phase is completed. Also one gets the impression that there are no inter-dependencies between the phases. In reality, decisions made in one phase often directly affect the work of the subsequent phases. This makes each phase somewhat dependent upon decisions made in a previous phase. Also there are times when a decision made in one phase is determined to be unrealistic by restrictions or actions taken in a following phase. Therefore, a feedback mechanism is needed to carry back information in order to keep the software project development moving. Each phase should be verified as being a correct implementation of its requirements.

Studies indicate that the later an error is caught, the higher the cost to correct. The cost of detecting and correcting an error more than doubles for each phase through which it passes undetected. This rate of cost increase holds true for each subsequent phase through which the problem passes without detection [Ref. 4], [Ref. 12].

A better software life cycle model is the one seen by Boehm [Ref. 4] in Figure 1.3. This model represents the development of standard large scale application software system. It is based on an assumption which resolves the problem with the previous model.

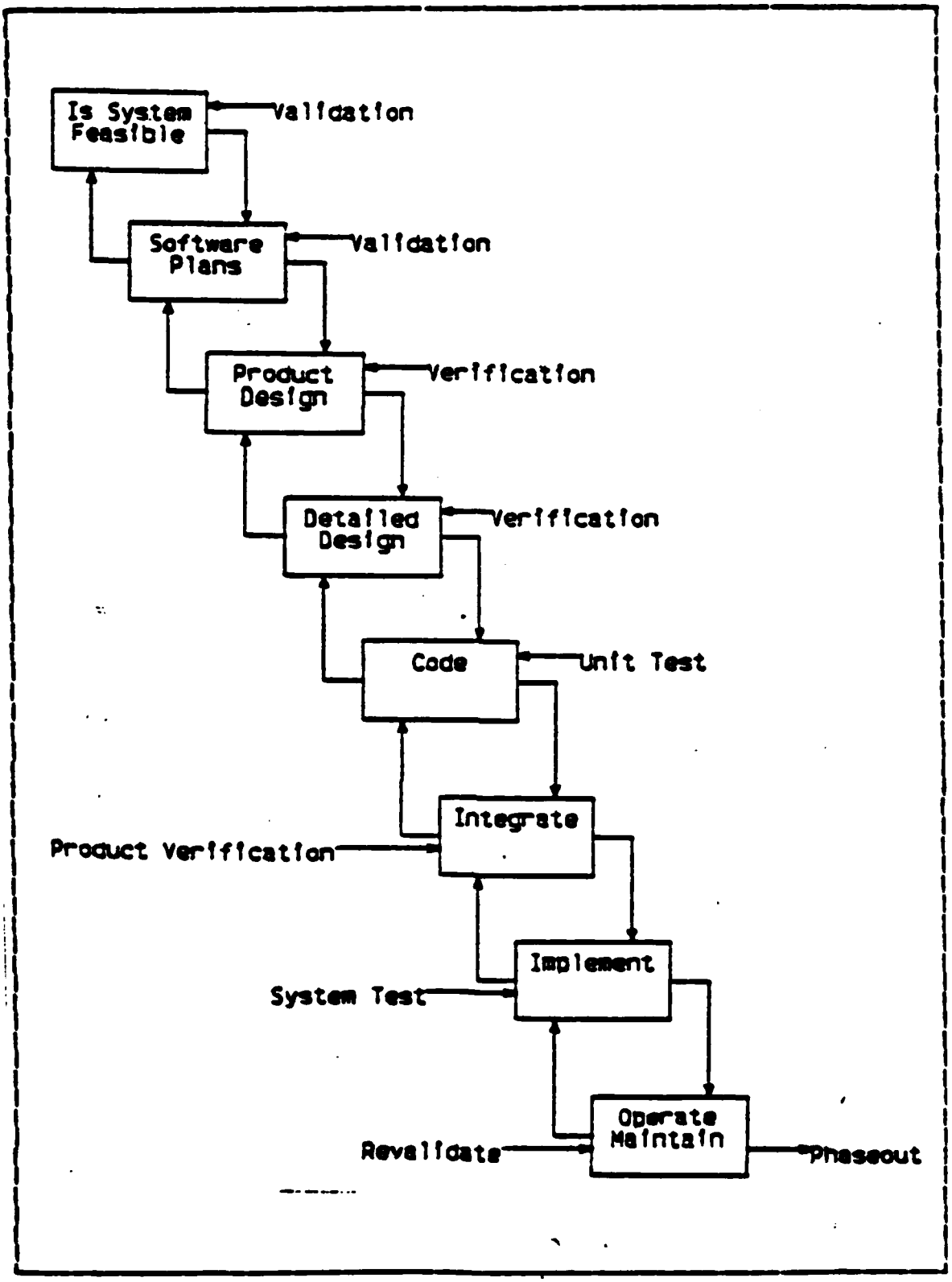


Figure 2.3 The Waterfall Model of the Software Life Cycle

Prior to moving from one phase to the next a verification phase will attempt to eliminate errors in the output of that phase.

D. WHAT IS MAINTENANCE

Maintenance is the function of keeping software in an operational mode. It refers to changes that have to be made to computer programs after they have been delivered to the customer or user. The maintenance function involves correcting error and design defects, improving the design, converting the programs to work in new environments and accommodating user requests for improvements.

Software maintenance is different from hardware maintenance. Hardware maintenance consists of replacing deteriorated components, putting in engineering changes that correct defects and making design enhancements; all these do not affect how the hardware is supposed to behave so the user sees no change. Software maintenance not only corrects defects and makes design enhancements, it also makes enhancements that change how the program behaves. Most maintenance work is caused by changing requirements rather than by reliability problems [Ref. 13].

E. MAINTENANCE CYCLE

Although maintenance does not follow the development life cycle, it nevertheless has phases of its own. A

significant difference is that a maintenance cycle if truncated prematurely because resources are exhausted, nevertheless results in a working change; an aborted development cycle has no useful result beyond sad experience.

The maintenance cycle can be described as consisting of the successive phases:

Understanding requirement

Specify the change

Developing the code

Regression testing

Documentation update.

Each is rather different than similarly named parts of development. Each requirement change is narrow and incremental. When a change must be made, no thought is given to its place in the entire system (and in fact it may so conflict with that system's intent or design that it is unwise to make it). Thus, requirement analysis consists of investigating the interaction of small parts with the whole. (This incremental character is also characteristic of succeeding phases.) If the maintenance cycle is terminated with requirement analysis, it is nevertheless a success; the change is shown to be unacceptable and rejected for cause. However, existing system deficiencies that generated the request for change still exist and another suggestion is

likely to be forthcoming. In this way, the cycle may loop back on itself, but for an entirely different reason than does the development cycle.

The coding phase is central to the maintenance cycle and it may include a distorted kind of design. What must be designed is a minimum-impact alteration of old code, not new code. Because of resource constraints it will probably be impossible to gain an understanding of the existing design; rather, syntactic features must be exploited to gain understanding. For example, to alter a module whose purpose cannot be grasped, it may be sufficient to note the contexts in which it is used, and see that the change is innocuous there. Of course, such a plan is very dangerous since updating this module for new future usage will not ensure correctness when such a dangerous change has been made. An entirely new kind of maintenance documentation must handle the problem.

If the maintenance cycle ends with coding, there is a useful product, the new code, which can be immediately tried by the user who requested the change and seen to be an improvement or not. But the cycle has been aborted before test and updating documentation, and if the last change introduced new problems, the cycle will begin again with new requirements to fix the new problems by redoing the code. The total cost now is higher for the original requirement

since the cycle started over to correct the newly introduced problem.

F. TYPES OF MAINTENANCE

Functionally, software maintenance activities can be divided into three categories which were originally proposed by Swanson [Ref. 14].

1. Perfective Maintenance

Perfective maintenance refers to enhancements made to improve software function by responding to customer and programmer-defined requests for changes. Perfective maintenance is required as a result of both the failure 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 improved. Perfective maintenance is the method usually employed to keep the system "up-to-date", responsive and germane to the mission of the organization. Perfective enhancement is the biggest maintenance consumer. According to [Ref. 13], 60% of the software maintainer's time is spent on these "make better" changes.

2. Adaptive Maintenance

Adaptive maintenance is the act of changing software to adapt to environmental changes. These changes consist primarily of the following:

- rules, laws and regulations that affect the system
- hardware configuration, e.g., new terminal, local printers
- data formats, file structure
- system software, e.g., operating system compilers, utilities

The software must be adapted to those changes. According to [Ref. 13], 18% of software maintenance is adaptive.

3. Corrective Maintenance

Corrective maintenance is the pure correction of software error required to keep the system operational. Corrective maintenance is usually a reactive process where an error must be fixed immediately.

There are three main causes for corrective maintenance:

1) Design errors
2) Logical errors
3) Coding errors

Corrective maintenance consumes only 17% of the maintainer's time, according to [Ref. 13]. (The remaining 5% is allocated to "other".)

G. SOFTWARE MAINTENANCE PROBLEMS

Software maintenance problems can generally be categorized as technical and managerial. Most of these problems, however, can be traced to inadequate management control of the software maintenance process. We will

present the technical aspects of maintenance problems here; they were taken mostly from Guidance in Software Maintenance [Ref. 15]. We will address the management control issues in later chapters.

1. Software Quality

Modern programming practices which utilize a well-defined well-structured methodology in the design and implementation of software systems address at least one major software maintenance problem--poor program quality. A lack of attention to software quality during the design and development phase generally leads to excessive software maintenance costs. The maintainability of the system is directly affected by the quality of the software produced during the design and development phases.

2. 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 specification by the developers
- The use of complex logic to meet the 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 which are bulky and very difficult to understand

3. Poorly Coded Software

Most of existing software contains poorly written code. As computer programming evolved, much of the code development was performed in an undisciplined, unstructured manner. 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 maintenance problem is worse when the program has been modified several times by different individuals with different programming styles. Often, such code does not do what it was intended to do, and it is sometimes harder to use than anticipated. Attempting to change such code without the aid of up-to-date specifications or other documentation is often a time-consuming effort.

4. Software Design for Outdated Hardware

Problems are associated with maintaining software which was designed to run on previous generation, outdated hardware. Often, the investment in 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 programmers are 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.

5. More Than One Programming Language Used

The use of more than one programming language in an application system 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.

6. Increasing Inventory

The impact of rapidly changing technology has resulted in a substantial growth in the number of new application systems. In addition, the average life expectancy of software systems has increased from about three years a decade ago, to ten-to-fifteen years for military programs today. Also, new programs are placed in service faster than old ones are retired leading to an increase in the amount of available code for evolution [Ref. 1].

7. Lack of Common Data Definitions

An application system should have common data definitions (variable names, data types, data structures, etc.) for all segments of 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 relate logically with one another.

8. Documentation

The programmer 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, conflicting or incorrect communication from those who have previously handled the system. There is often inadequate documentation, no detail records of the original requests and subsequent updates; no explanation of existing code and changes which have been made to code; and no explanation

concerning why complex logic and coding structures were selected over a more simple implementation. There may be months or years between the original development of the 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 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.

9. Chain Reaction (Ripple Effect)

A chain reaction occurs when changes to one part of a program unexpectedly affect other parts [Ref. 16]. This happens because of interdependence that can exist between program modules. Modules that share common functions or data are interdependent. Several modules may call a common module or may reference a global variable. Any change made to a common module may alter the internal processing of any module that accesses it. The more interdependent the program modules are, the more complicated it becomes to determine the ripple effect from even a simple program change. A major cause of software quality deterioration during the maintenance phase is not being able to determine completely the ripple effect arising from the program changes.

The possible ripple effect arising from program changes must be carefully examined. Usually this involves a manual search through the code beginning with the module(s) in which the change is made and continuing on to all modules sharing global variables or common routines with this module(s). Depending on how many modules and variables must be examined, the search can become very tedious and time consuming.

Module coupling is one measure of module inter-dependence [Ref. 17]. When modules are very interdependent (e.g., one module makes an unconditioned transfer of control to a label within the boundaries of another module), they are called tightly coupled. Great care should be taken when changing the code in such a module since the change is likely to affect the internal processing of any other modules that are tightly coupled with the changed module.

Cross-reference maps, storage maps, and traces provided by many compilers, automatic flow charters, and execution flow tracers can help identify the ripple effect from a change.

10. User Knowledge

The users are often unable to concisely specify what they want from an application system. The developer in order to develop the right system which accurately performs all of the functions the user wants needs a correct,

detailed specification based on clear concise requirements. The user usually doesn't provide this. The maintainer must enhance the system that was developed with inadequate specifications and the new, often incomplete and vague change request from the user.

On the other hand, if the user is knowledgeable and the system is successful, additional features will be requested, while if the system does not work, there will be a constant demand for corrective action to make it function properly. Therefore, it is essential that some sort of controls be established and enforced to ensure that the change requests are both justified and do not interfere with the maintenance workload.

11. Personnel

It is thought that software maintenance is often considered by maintenance personnel to be unimportant, unchallenging, unrewarding, uncreative work which is not appreciated by the user or by the rest of the ADP organization. Usually management does not reward personnel who performed software maintenance as generously as those who performed software development. New programmers usually start by working in maintenance and the more experienced professional are assigned to be analysts, designers, and developers. The importance of software maintenance to the successful, smooth operation of an organization is

increasing, which leads to selecting to do maintenance well-qualified dedicated professionals who can readily understand the system. The programmer who does maintenance must be a highly skilled, competent programmer and analyst not only to make the actual changes but to make sure of the impact of those changes on the rest of the system and its environment. A rotation of personnel between development and maintenance will give each group a different experience and help to reduce the morale problems of the maintenance personnel.

12. Understandability

Understandability is considered the most fundamental requirement for a maintenance program. To maintain a program you have to understand it first and if it is not understandable it is almost impossible to maintain. This is a major factor in the high cost of maintenance and the distaste for performing maintenance tasks.

Understandability is defined as the ease with which we can understand, by reading the program source code and its associated documentation, the function of a program and how it achieves this function. If the program is understandable, the reader can easily determine the programmer objective, assumptions, constraints, inputs, outputs, components, relationship to other programs and status. Programmer familiarity influences how easy or difficult it is to understand a program; the less familiar

the maintainer is the more difficult the program is to understand. Martin and McClure state that understandable programs generally have several common characteristics: structure, consistency, completeness, conciseness, documentation. Each of these characteristics will be discussed in more detail.

a. Structure

The effective structuring of a program increases understanding by standardizing the program form. The standardization will set restrictions on program control constructs, modularization, and documentation. Although helpful, good structure does not completely ensure all aspects of program understandability. Boehm suggests that in addition to being well-structured, an understandable program must also be concise, consistent, and complete [Ref. 18].

b. Consistency

The program is considered consistent if it follows a consistent design approach and is written in a consistent coding style. It is difficult to understand a program in which the style of writing does not follow a common method of construction. This is sometimes difficult to accomplish when several members work together as a team unless there is a standard coding style that each one should comply with. Consistency with design approach is what

Brooks called conceptual integrity [Ref. 19]. Conceptual integrity is preserved when one basic design approach is carried out through the entire program, simplifying the task of understanding the rational behind the program logic. Selecting consistent variable names throughout the program, and using inline comments to clarify the coding statements will ease the understandability of the program. The practice should be consistent throughout the program.

c. Completeness

A complete program has all of its components available for use and reuse by the maintainer. To accomplish understandability the maintainer should be able to access all parts of the program that are related to the maintenance function. Any variables or modules should be included in a cross-reference scheme so that the maintainer can trace a program component through the system. Error messages should be made understandable and every unusual feature in the program should be clearly explained.

d. Conciseness

A concise program is one that uses only the coding necessary to achieve the design requirements with no extra (perhaps unused) pieces of code. Every piece of code must be reachable by some action of the program. Comments should not be excessively verbose or cryptic in meaning. When complexity increases, understandability decreases. The program should be as simple as possible.

e. Documentation

The program must be well-documented in a consistent way. Comments should be arranged near each module to describe the module in some detail. The module comments should include the purpose of the module, the variables used or modified in the module and a description of the output of the module. The module description should also include which other modules invoke this module and which ones are invoked by this module. Information that is recorded as to how a module of the program is reached and how each module is related in the overall system scheme aids understandability. Proper documentation should also be concise with only the necessary information being provided to the maintainer so as to enhance understandability without confusion.

13. No Systematic Problem Solving Technique

Problem solving is a major part of programming, in fact, programmers normally are involved in two types of situations that could be considered problem solving, namely the design and writing of the program and its debugging and testing. To write a program, one must first understand the situation, devise a strategy to solve it, and convert that into lower level steps to implement it [Ref. 20]. This is a problem-solving sequence, although most programmers would not recognize it as such. (Note the similarity here between

designing and implementing a new program and doing the same for an existing operational one.) The study of problem solving can yield techniques and ideas that maintenance programmers can profitably use. Kepner and Tregoe state that "problem solving is a process that follows a logical sequence." Although their method as presented is geared to managers, their principles and examples also extend to solving technical problems. Their three steps are: finding the problem, analyzing its cause, and deciding on a course of action in solving technical problems such as those of a maintenance [Ref. 21]. The course of action is quite clear once the cause of the problem is found. The hard part is finding the bugs, which can be time consuming, frustrating and often times an undesirable task.

III. CONTROL TYPES AND MEANS

A. INTRODUCTION

The maintenance function is out of control in many organizations. According to Reutter "maintenance often has the outward appearance of being a helter-skelter, uncoordinated activity rather than a planned, methodical, controlled, necessary business function of any organization committed to computerized data processing" [Ref. 22]. Some of the problems in managing software maintenance are classical problems of control. These classes of problems exist in any organization. This chapter will cover some aspects of control. Some background is given for control in general, including some discussion about the objectives of control in data processing organizations. The characteristics of good control and how control influences behaviors is also discussed. The following chapter will discuss how general principles of control can be applied to software maintenance.

B. DEFINITION AND OBJECTIVES

Controls are the means by which we head toward an objective. In data processing organizations the main objectives are:

- to ensure that only complete, accurate and authorized data is processed.

- to prevent or detect accidental errors or fraudulent manipulation of data.
- to ensure the adequacy of management.
- to provide security against destruction of records and to ensure continuous operations [Ref. 23].

The objectives can be achieved through better control, which keeps us from veering off in undesirable directions and prevents unwanted things from happening. Essentially control means "keeping things on track" [Ref. 24]. Knowledge of the objectives of the organization is a necessary prerequisite for conscious control efforts, as without it, activity can only be described as aimless [Ref. 25]. Objectives do not necessarily have to be defined in specific, measurable terms, but it is critical to have a general understanding of what the organization is trying to accomplish.

C. CONTROL INFLUENCES BEHAVIORS

Control involves influencing human behavior, because it is people who make things happen in an organization. Control involves managers taking steps to help ensure that human beings do what is best for the organization. Control is necessary to assure that the people do what they should and prevents them from doing what the organization does not want them to do. The point is that if all personnel could always be relied upon to do what is best for the organization, there would be no need for a control system. But individuals sometimes either do not know the organization's

objectives or are unwilling to act in the organization's best interest; so management must take steps to guard against the occurrence, and in particular the persistence, of undesirable behaviors and to encourage desirable behaviors.

D. GOOD CONTROL AND ITS CHARACTERISTICS

Good control means that no major, unpleasant surprises will occur. The label "out of control" is used to describe a situation where there is a high probability of forthcoming poor performance, despite a reasonable operating strategy. Some important characteristics of good control are:

(1) Control is future oriented; the goal is to have no unpleasant surprises in the future. The past is not relevant except as a guide to the future.

(2) Control is multidimensional, and good control is not established over an activity or entity with multiple objectives unless performance on all significant dimensions has been considered.

(3) The assessment of whether good control has been achieved is difficult and subjective, not only because of human limitations and biases but also because adequacy must be measured against a future that can be very difficult to predict.

(4) Better control--meaning higher assurance of success--is not always economically desirable [Ref. 26]. Like any other economic goods, control tools are costly and should be implemented only if expected benefits exceed the cost. Some economists [Ref. 27] define the term control loss to be the cost of not having a perfect control system; that is, it is the difference between the performance that is theoretically possible, given the strategy selected, and the performance that can be reasonably expected with the control system that is in place. More or better controls should be implemented only if the amount by which they would reduce the

control loss is greater than their cost. Therefore, good control can also be said to have been achieved if the control losses are expected to be smaller than the cost of implementing more controls.

E. TIGHT CONTROL

The amount of control achieved or the degree of certainty provided by a control system can be described in terms of how tight or loose the system is. Assuming away the problems of costs and possibilities of harmful side effects that are often considered with tight control (e.g., negative attitude), tight control is good because it provides a high degree of certainty that people will act as the organization wishes. Tight control is only feasible where management has detailed and reasonably certain knowledge about how one or more of the control objects--results, actions, or personnel--are related to the overall organizational objectives. In other words, "our ability to control is a function of our knowledge" [Ref. 25].

F. NEED TO CONTROL SOFTWARE MAINTENANCE

Computer programs are designed to satisfy the needs of people. These programs are the tangible output of thought processes, the conversion of thought processes into products which should match the real needs of the people who will use the software product. This goal is product integrity, which is defined to be the intrinsic set of attributes that characterize a product:

- that fulfills user functional needs;
- that can be traced easily and completely throught its life cycle;
- that meets specified performance criteria;
- whose cost expectations are met;
- whose delivery expectations are met [Ref. 5].

Our view of software should not be restricted by improperly equating "software" and "computer program". Software is much more. A definition which can be used to focus the discussion in this chapter is that software is information that is

- structured with logical and functional properties;
- created and maintained in various forms and representations during the life cycle;
- tailored for machine processing in its fully developed state.

So software is not simply a set of computer programs but also includes the documentation required to define, develop, and maintain those programs.

Many development and maintenance failures or problems are the result of poorly defined requirements which are changed without control [Ref. 28].

The majority of the system and programming efforts in many organizations is spent on maintenance, but in most organizations the management of software maintenance has little information on what activities comprise the maintenance function. As mentioned before, most of the

maintenance effort is spent on perfective maintenance which is performed to improve and enhance the software to accommodate the user's new requests. Without adequate information, it is difficult for management to evaluate the legitimacy of this type of maintenance with respect to the software life cycle goals and overall organization goals, or to evaluate the necessity of devoting the majority of its software staff to maintenance. This is a serious problem: if management controls are not applied, the maintenance function may absorb all system and programming resources, leaving nothing for development of new software systems.

The problem of controlling the maintenance function arises because of the following:

(1) Most user requests are not based upon well-thought-out, legitimate requirements or are based upon personal preferences.

(2) Software changes could be performed more efficiently if user requests were better controlled and maintenance personnel better trained.

(3) Can we identify which portion of the maintenance problem can be attributed to poorly defined user requirements, poorly defined functional specifications, or poorly implemented and tested code?

Understanding what is being done and why, is the first step in controlling the maintenance function. We need to apply to software maintenance activities the formal controls used in software development projects. We need to:

- Categorize and record maintenance tasks.

- Organize the maintenance staff to better identify individual responsibilities.

- Control user requests with formalized change control procedures and an open communication channel between the maintenance group and user groups.

IV. CONTROLLING SOFTWARE THROUGH CONFIGURATION MANAGEMENT

Computer programs and applications continue to expand. This requires some sort of control for these programs and related documentation. If software is allowed to change in an uncontrolled manner, our ability to manage the applications will be lost and the cost will be very high. There is a need for assuring that the requirement definition is complete as well as controlling changes to requirements and design, and tracking the resulting impact on cost and schedule. One control technique which promises to be effective is Configuration Management (CM). CM is the discipline of identifying the configuration of the system at discrete points in time for the purpose of systematically controlling changes to the configuration and maintaining the integrity of and traceability of the configuration through the system life cycle [Ref. 29]. While CM was originally designed to control hardware production, its principles can be tailored and refined to relate to the development and maintenance of software. Careful control of changes improves product reliability and reduces the possibility of introducing faults into software products through the maintenance process. CM answers questions such as: when your next production item is delivered, or updated through

maintenance, will you know what the exact configuration is supposed to be? Will the new technical manuals support the developed system? Can you be sure that the contractor has met all performance requirements?

Excellent definitions and practices of configuration management for systems, equipment, munitions, and computer programs, can be found in MIL-STD-483A (USAF) dated 4 June 1985.

Review of the literature provides some basis for control and recommendations to follow through the use of software configuration management which contributes to the solution of some of the maintenance problems. Next, software configuration management and its four components will be discussed in some detail.

A. SOFTWARE CONFIGURATION MANAGEMENT

Software configuration management (SCM) is simply configuration management tailored to systems, or portions of systems, that are comprised predominantly of software. Thus, SCM does not differ substantially from the CM of hardware-oriented systems, which is generally well understood and effectively practiced. Of course, hardware engineering is different from software engineering, but broad similarities do exist and a term applied to one segment of engineering can easily be applied to another, even if specific meanings of those terms differ significantly in detail.

The primary objective of SCM is the effective management of the software system's life cycle and its evolving configuration. A concept fundamental to this management process is that of a "baseline". A baseline is a reference point or plateau in the development of the system; a baseline is formally defined at the end of each stage in system life cycle [Ref. 30].

A baseline has three connected functions:

- as a measurable progress point
- as a basis for subsequent development and control
- as a measurement point for assessing quality and fitness for purpose, before the final system goes into maintenance [Ref. 6].

The final stage of the system life cycle, the operational stage, is the most significant stage since the system life cycle terminates when the operational stage terminates, also this stage consumes the most time and costs the most when compared to other stages.

B. SCM PURPOSE AND BENEFITS

SCM then is a methodology and includes concepts, policies, and procedures. Its purpose is to aid in managing the functional and physical characteristics of an item and accompanying documentation. SCM is the vehicle controlling the development, maintenance, and documentation of the system. It provides several benefits, including:

a) A precise identification of the current configuration item, including traceability back to previous configurations.

b) The ability to reproduce defined environments (baselines) to permit diagnosis and correction of problems.

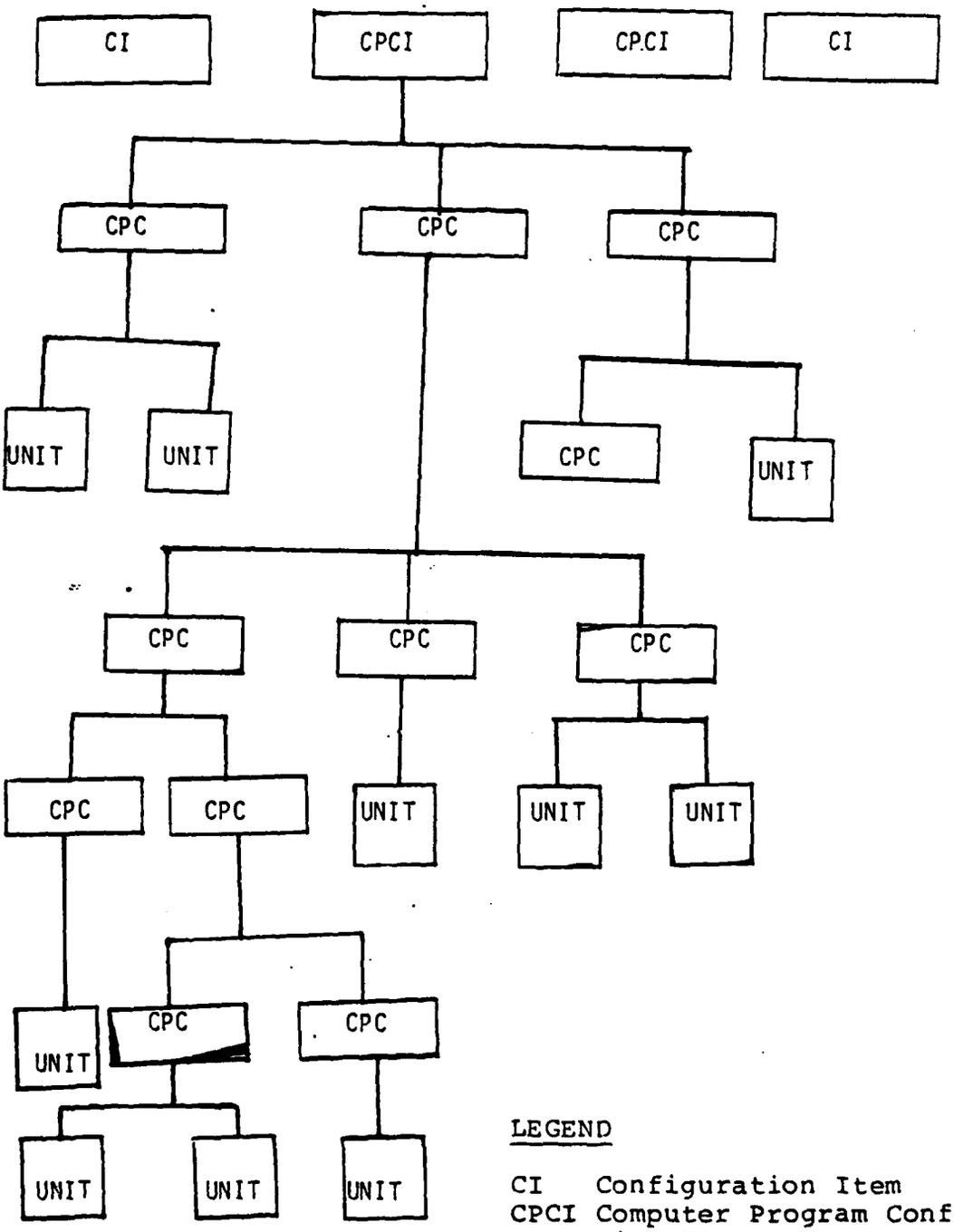
c) A formal structure for assessing impacts of proposed changes.

d) The procedures for notifying official baseline document holders of approved/released configuration and changes. [Ref. 28]

SCM works with configuration items (CI) which have specifically been designated in a given acquisition as being subject to configuration management. The "configurations" of the item or system refers to the totality of its functional and physical properties, which are defined and documented in the form of specifications. The configuration manager works with these specifications, not with the individual programs. Another key concept is a computer program configuration item (CPCI) which is a set of coded instructions on machine-readable media. CPCIs are algorithms, programs, groups of programs or an entire system which have been designated for configuration management [Ref. 31]. A hierarchical level of a system is shown in Fig. 4.1. The four components of SCM are identification, control, status accounting, and auditing.

1. Software Configuration Identification

Software configuration identification is the process documenting performance requirements, qualifications, and acceptance criteria. This identification is done in baselined documents which describe the functional and physical characteristics of a CPCI or system.



LEGEND

CI Configuration Item
CPCI Computer Program Configuration Item
CPC Computer Program Component
UNIT Non-Divisable Program Function

Figure 4.1 Hierarchical Levels of a System

Changes to baseline components need to be defined since these changes, together with the baselines, specify the system evolution. A system baseline is like a snapshot of the aggregate of the system components as they exist at a given point in time. The role of software configuration identification in the SCM process is to provide labels for the contents of these snapshots.

A baseline can be characterized by two labels. One label identifies the baseline itself. The second label identifies an update to a particular baseline. An update to a baseline represents a baseline plus a set of changes that have been incorporated into it [Ref. 30].

- Configuration Identification Elements

The following elements are necessary to establish configuration identification:

a) Mission Analysis. The initial element is the process of requirements analysis which is necessary to transform requirements into a Functional Description.

b) Functional Description (FD). This document is produced during the conceptual phase. It is based on studies and analysis done during the conceptual phase and is the first system life cycle document. The FD states the mission and functional requirements for a system and defines required external interfaces. The FD is baselined at the end of the conceptual phase.

c) System Specification (SS). This is a product of the definition phase and is baselined at the end of the phase. The SS is the result of allocating the requirements from the FD into components of the system. If the components are actually subsystems, then the design of the subsystem will be reviewed at the preliminary design review. The SS is the first of the CPCI configuration identification documents. This

document contains the information necessary to design a CPCI and test it against its performance criteria.

d) Program Specification. These are initially the "build to" specifications. They are reviewed prior to actual coding but are not baselined until the end of the development phase.

e) Other Types of Specifications. The additional documents that may be part of CPCI's configuration identification include the data requirement document (RD) and the data base specification (DS). Other documents, which may be required but are not a part of configuration identification, include the test plans and procedures, the computer operation manual (OM), and the program maintenance manual (MM). These documents are influenced by, but do not identify or control, a CPCI's configuration identification.

f) Configuration Identification Numbers and Markings. The configuration identification number (specification number) is an essential element of the identification and is used to reference a document during its life [Ref. 28]. CPCI selection is based on consideration of the following factors:

(1) Design Decisions. Processes with strong interactions should be the same CPCI.

(2) Separate Computer. Programs to be operated on different types or models of computers should be separate CPCIs.

(3) Separate Schedules. Programs scheduled for development, testing, or delivery on different schedules should be separate CPCIs.

(4) Different Operation Control. Programs which are largely identical during development may be identified as separate CPCIs; if they are to be run on different computer systems or to be evolved and controlled separately during the operational phase, they should be in separate CPCIs. Programs on the same computer model but at different locations and requiring adaptation data at each installation may be maintained as separate CPCIs or as a version with a single CPCI [Ref. 32]. Breaking a system down into a series of CPCI provides maintenance management with increased visibility and control into the maintenance

process. However, this is done at an increase in cost for documentation, reviews and audits, general paperwork, and manpower. Each CPCI requires its own specification, reviews and audits, acceptance and validation testing, configuration control logs, and configuration status accounting reports. Baseline management activities are normally more intensive and important during the development phase, since the designing, coding, and development testing will identify system errors which require modification and result in baseline change requests.

2. Software Configuration Control

The evaluation of a software system is, in the language of SCM, the development of baselines and the incorporation of changes into the baselines. That is, software configuration control is management of change. Software configuration control focuses on managing changes to SCIs in all of their representations. It involves three basic ingredients:

(1) Documentation (such as administrative forms and supporting technical and administrative material) for formally precipitating and defining a proposed change to software system.

(2) An organizational body for formally evaluating and approving or disapproving a proposed change to a software system.

(3) Procedures for controlling changes to software systems.

The objective of software configuration control is to properly identify and consider proposed changes within the scope of total project and program. The final product, the CPCI, must agree with the product specifications and must jointly agree with the requirements stated in the SS and FD.

The group that administers software change control is the *Configuration Control Board* (CCB). The purpose of CCB is to control the baseline. Ideally, membership includes senior managers of the impacted functional areas and of the interfacing/using commands. Developers should be present as consultants.

A Baseline Change Request (BCR), sometimes called Engineering Change Proposal (ECP), identifies the need for change to an approved configuration (specification). This change request document contains information such as:

- description of the problem and proposed change
- justification of the change
- the identification of the originator
- the objective of the change
- the benefits of the change
- identification of the affected baseline
- the impact on other programs

The next chapter will discuss what events cause a change, the classes of change, and the change process and how an ideal change process model can be used to control and solve the problem of maintenance change being out of control.

3. Software Configuration Status Accounting (SCSA)

SCSA is a management information system providing traceability of baselines and changes. A decision to make a

proposed change is generally followed by time delay before the change is actually made, and changes to baselines generally occur over some period of time before they are incorporated into baselines as updates. A mechanism is therefore needed for maintaining a record of how the system evolved and where the system is at any time relative to what appears in published baseline documentation and written agreements. SCSA provides this mechanism. SCSA involves the maintenance of records to support software configuration auditing. SCSA is thus the means by which the activity associated with the other three SCM functions is recorded; it therefore provides the means by which the history of the software system life cycle can be traced. The SCSA complexity increases as the product moves to the operational baseline because of the multiple software representations. This complexity generally results in large amounts of data to be recorded and reported and is generally supported in part by automated means.

4. Software Configuration Auditing (SCA)

Software configuration auditing provides the mechanism for determining the degree to which the current state of the software system reflects the software system pictured in the baseline and requirement documentation. It also provides the mechanism for formally establishing a baseline update. SCA serves two purposes, configuration

verification and configuration validation. Verification ensures that what is intended for each software configuration item as specified on one baseline or update is actually achieved in the succeeding baseline or update; validation ensures that the SCI configuration solves the right problem. SCA is applied to each baseline and corresponding update. Determination that an SCI structure exists and that its contents are based on all available information is an auditing process common to all baselines.

SCA is intended to increase software visibility and to establish traceability throughout the life cycle of the software product. This costs time and money, but it is justified by the avoidance of costly retrofits resulting from problems such as sudden appearance of new requirements and discovery of major design flaws. SCA makes visible to management the current status of the software in the life cycle product being audited. It also shows whether the project requirements are being satisfied and assures that no change other than those authorized have been made. Every requirement is traced successively from baseline to baseline as life cycle products are audited and baselines are established or updated. An excellent overview of the software audit process from which some of the above discussion has been extracted appears in [Ref. 33].

DoD Directive 5000.29 Management of Computer Resources in Major Defense Systems, states:

Defense system computer resources, including both computer hardware and computer software will be specified and treated as configuration items.

The primary objective of software configuration management is the effective management of a software system's life cycle and its evolving configuration. Configuration is a form of management in that it gives one the ability to manage change during both the development and maintenance processes.

V. THE NEED FOR MAINTENANCE CHANGE CONTROL

A. INTRODUCTION

Software maintenance change control is an important management problem. Once software is implemented and the maintenance phase started, its life cycle will extend from 10-15 years (sometimes even longer for military systems). During this period the software will experience significant changes and redevelopment to meet the user's changing needs. Even though the user prefers to have new system capabilities or better performance, when asking for a software change it is not desirable to implement the change without proper evaluation and control.

B. TYPES AND CLASSES OF SOFTWARE CHANGES

The request for a software change usually comes from the user, but the system analyst and the management could also request a change to the software. The requests for changes can be made in response to a variety of events.

1) Software deficiencies. The existing software baseline may be found to be inadequate or incorrect because of errors in the requirements, specification, design, implementation, or for other reasons.

2) Hardware changes. Problems with hardware components and the interfaces among hardware subsystems may yield to solution only through software change.

3) New operational requirement. The ground rules for the system's operation may be modified (i.e., the required performance may be increased or decreased).

4) Economic reason. Means for effecting cost savings may be determined, or lower development or operating cost may be decreed that requires software modifications [Ref. 30].

There are two classes of changes impacting software products and documentation. Each requires different processing and review procedures.

a) Class 1. These are changes that effect schedule, cost, or technical parameters of an established baseline. A class 1 change requires submission of a change request for CCB review and project manager approval prior to any work being done on it.

b) Class 2. These are basically changes to correct errors in documents or to add clarifying information. It does not require prior approval by the project manager or CCB, but when implemented would come under normal CM procedures [Ref. 34].

C. WHY IT IS IMPORTANT TO CONTROL CHANGES

From Chapter Two we have seen that about 80 percent of maintenance work is responding to user change requests. Technical, managerial, and economical factors have to be considered when dealing with a new change. To make a decision about a change one must address the following questions:

Why is the change needed?

What is the impact of this change on the rest of the system?

What is the cost involved to do the change?

What is the benefit of the change?

How important or complicated is this change?

There is always the risk of introducing new errors into the program every time a change is made. Also some user requests for change are not justifiable for cost, technical, or other reasons. One author [Ref. 35] stated that some users exaggerate their needs for particular enhancements, and once they are implemented they are seldom used.

One of the big problems in making changes is that changes are not independent. Performing one change may make others easier, harder, or sometimes even impossible. Some changes will cost less if implemented in a certain order making the decisions of assigning priorities a hard one. Some changes would be rejected mainly because when implemented these changes would either make others impossible or more expensive. When evaluating the change requests, besides considering the impact, the importance of the change, and the cost, one has to consider the useful life of the software when the change is requested. By this we mean programs have a certain life and it is not wise to implement a big change into a system which will be replaced by a new one in the near future.

It is obvious that approving and assigning priorities to change requests is not an easy task due to the nature of software. To assure that only the necessary, justifiable, and within budget changes are implemented with the proper

priorities (not necessarily in accordance with the user requested priorities), there is a need for controlling the software change process to avoid the high cost and to minimize the impact of a change on the rest of the system and on other changes as well.

Controlling software changes requires controlling the software programs and the associated documentation. It is useful to give some definitions:

- Program Control: a method of controlling the basic program system, user back-up copies and variants so their content is always known and only authorized change is implemented.
- Documentation Control: a method of approving, processing, assessing, and updating all related documentation in step with each other and corresponding programs.

D. SOLUTION

Controlling changes requires the estabishment of some kind of policies and procedures. These controls govern such things as the forms and procedures that people use and the methods for documenting and maintaining the software. There are two types of control: tight and loose; some organizations rigidly control change through CCB while others leave it to the programmer/analyst to make the decisions.

According to [Ref. 36] "Experience has shown that having good standards provide programmers with good practices and allow them to be creative in those areas where creativity

really improves productivity." Management has to enforce the policies and procedures once they have been established, otherwise complications will arise and control will be lost. The establishment of a good policy for change control is not straightforward; it depends on the type of organization and the size and type of the software involved. A policy might be considered good and effective for one organization, but is very costly or not effective for others.

Even though there is some discussion in the literature about the need for controlling the software changes, there is no detailed step-by-step procedure that covers all the actions that have to be taken if good and effective control is to be accomplished. Most of the existing policies agree in principle with the following steps:

- A request for a change has to be made by the user or other authorized personnel.
- The change has to be evaluated and if appropriate, approved.
- Once approved, the change request is sent to the responsible software organization for implementation.
- Related documentation has to be updated.

These principles only give a general idea about change control, but they don't specify the criteria for evaluating the request or who is responsible for making the decisions to approve or disaprove the change. These principles did not mention how to control multi versions of multi copy software.

In most organizations the authority to make a decision to change is given to the Configuration Control Board.

E. CONFIGURATION CONTROL BOARD (CCB)

Program change requests come from the maintenance group, user group, operational group, and the management group. Some of the change requests from different groups may be similar and on the other hand, different requests may suggest changes that are not compatible with each other. As mentioned before, some changes are not really needed. The CCB has the authority to review, approve or disapprove the change requests. Not all change requests require processing through the CCB. There are many instances when such a formal process is not desirable (for example, on a very small software project). The board is a permanent committee which is the final authority within the project on proposed major changes. The principle function of each board member as suggested by [Ref. 29] is to verify the following:

1) The change is necessary.

2) The method of implementation is feasible.

3) The schedule and cost requirements can be met.

The CCB group will have members of all concerned groups (i.e., users, engineers, management, programmers/analysts). This is necessary to have a global picture of the change request and its impact or importance to help the CCB make a good decision.

Some change requests will be approved and will be given some level of priority, others will be rejected or delayed for several reasons including:

- The change is not needed. The CCB might decide that this change is not really needed.
- Inter-system conflicts. There might be higher priority changes on some other application that require delay for this request even if it has been approved. This is true at organizations with a limited number of programmers to do the changes.
- Not a program change. Some errors or requests for change arise from using the wrong procedure or document. This type of problem can be solved with manual solutions to the right document and doesn't involve computer program change.
- Reordering the change request. Users request changes and assign their own priority to the changes, but since the CCB has more information about the system they might see that some changes might cost less if installed with other changes in a different order than the user requested.

The problem of controlling the change and tracking it requires a clear effective change control policy and procedures.

F. METHODOLOGY FOR CONTROLLING THE SOFTWARE CHANGE

Two methodologies (models) for change control will be discussed, compared and evaluated for their effectiveness and completeness in providing a useful model to be used for achieving the objectives of change control. Glass and Noiseux in their book Software Maintenance Guidebook [Ref. 37], suggested a method for controlling software maintenance error changes. They organized the maintenance into four

groups: error tracking people, configuration management group, software development group, and test group. Glass advised that the CCB should be placed very high in the software organizational structure and it should have representative membership from non-software parts of the organization. The CCB has the authority to disapprove or approve changes and assign priorities to each request. The responsibilities of CCB should state what the decision process is for those occasions where emergency change is necessary. Keeping track of the changes provides a good history of data to evaluate the system by knowing which parts of the system have had the most errors and the types of those errors. This could lead to a redesign of parts of the software that have a significant error history. Another useful reason for tracking and logging the errors is knowing what changes have been evaluated, rejected, or completed. Error tracking should start after the software product is in the maintenance phase since tracking errors during development is very difficult and not as useful. Following Glass' method, whenever a software problem is discovered after the tracking system is initiated, a software problem report (SPR) is generated. The SPR contains useful information such as:

- definition of the problem
- location of the problem

- priority
- proposed solution

The tracking group will give a number to each SPR and submit it to the responsible software organization which will examine the SPR to see its effects on other parts of the program and suggest its priority. The nature of the correction is submitted to the change boards which decide whether to approve it; if approved the change is implemented into the program and tested against the SPR to see if the problem has been corrected. Regression testing is needed to ensure that no problem has been created by the change.

The users and management need to know the status of specific problems. The user likes to know when to expect the completion of his request and which versions of a piece of software contain which fixes. The correction process as seen by Glass is shown in Fig. 5.1.

The SPR processing flow is mainly for correction requests which arise for several reasons including: software malfunction, documentation error, software inefficiency, and test case/procedure error.

The general scheme of this SPR flow can be applied with the same modification to a general change request flow, which includes user enhancement.

G. ANOTHER CHANGE CONTROL MODEL

Perry in his book Managing System Maintenance [Ref. 36] presented his model for change control which addressed most

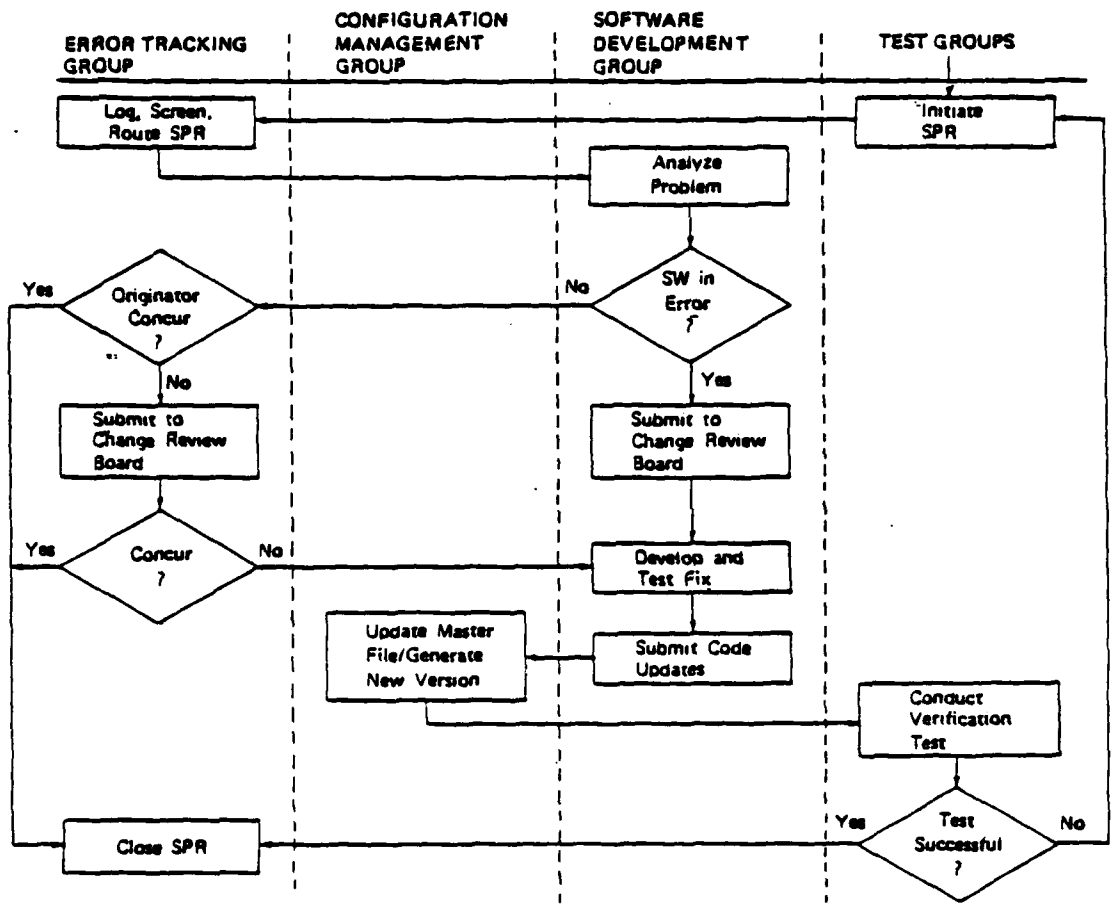


Figure 5.1 Glass' Change Control Model

of the missing aspects of change control in the Glass model. Perry defined four objectives for installing the change.

1) Install the Change as Specified. The needs have to be specified in such a way that they can be measured and only the agreed upon changes should be implemented.

2) Document Changes. The changes made to the application system have to be reflected into the system documentation. The documentation has to be current, reflecting the present system. The change is not considered complete unless the documentation is updated and complete. This documentation includes system documentation, operations documentation, user documentation and control documentation.

3) Keep Old System Operational. The old system must be maintained in an operational status during the time that the change is being implemented. Some special procedures might be required to ensure that the system can run and/or that new requirements are incorporated through manual or other means until the change is implemented.

4) Installing Change on Time and within Budget. Users requests have to be evaluated for time and cost, while it is not always possible to achieve either, it should be a high priority objective of the maintenance team to achieve the time and budget requirements.

There are several concerns when dealing with maintenance changes and there has to be adequate controls to reduce these concerns. Knowing these concerns and addressing them will significantly reduce their occurrence. Thus, the effort and time required to develop the methods, procedures, and controls will result in more effective, trouble free maintenance. These concerns include:

- Will it be known if the change achieves the change objective?
- Will the change be reflected in updated documentation?

- Will the system maintenance testing process be adequate?
- Will the change process be planned?
- Will serious problems occur before the change is installed as the time extends between the request for the change and implementing it?

To alleviate these concerns the methods and procedures for making the changes have to address these concerns and deal with them through using the necessary approval methods, forms, and feedback about the process.

Perry's change phase includes several steps which cover more aspects of the change control than what was mentioned in Glass' SPR process flow. This phase starts after the change has been specified and the priority for implementing it has been established.

There are seven steps that are normally executed in making a change:

Step 1 - Planning the change process

Step 2 - Operating the system until the change is made

Step 3 - Obtain needed access

Step 4 - Establish performance criteria

Step 5 - Implement the change

Step 6 - Test the change

Step 7 - Document the change

These seven steps are shown in Figure 5.2 and will be discussed below.

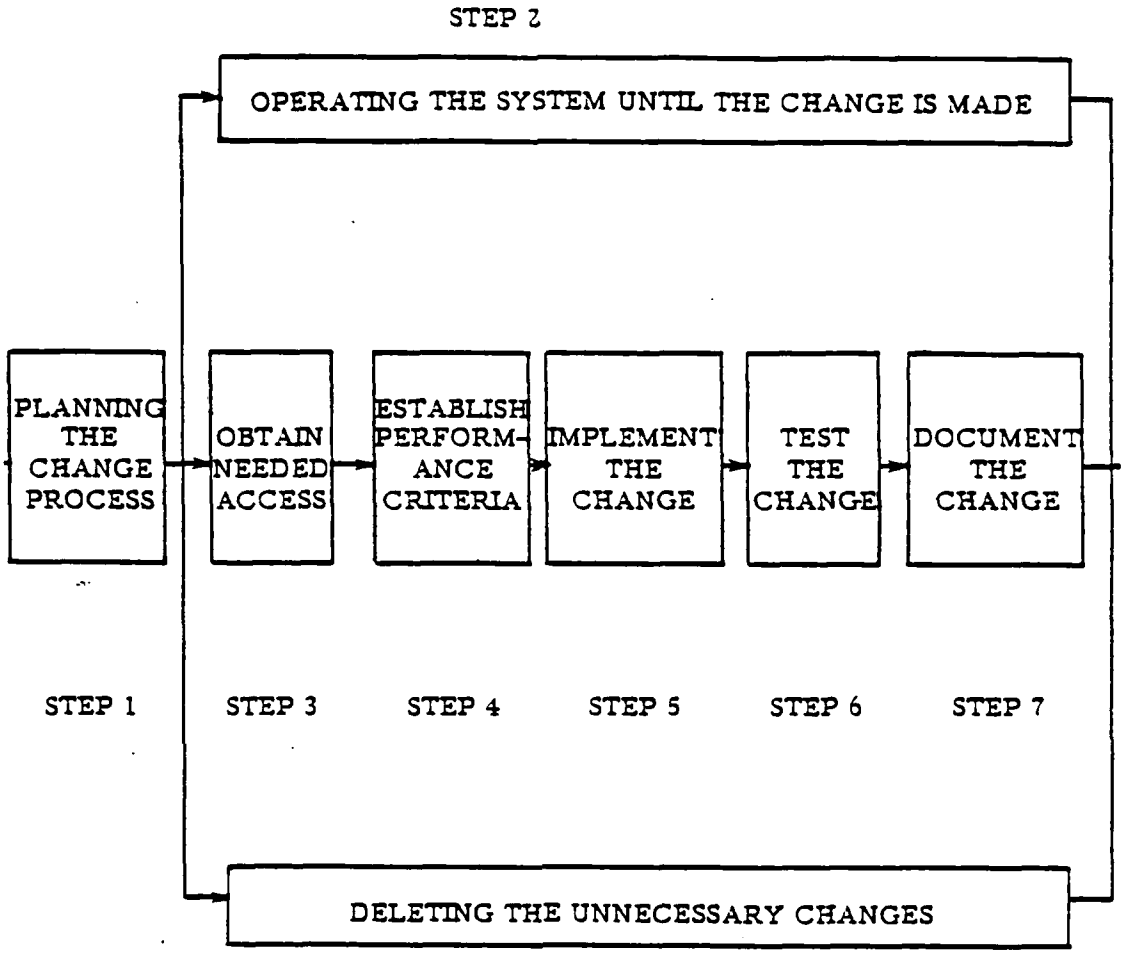


Figure 5.2 Perry's Change Control Model

Perry gives the system maintenance analyst the authority to eliminate any change or adjust the priority. According to Perry deleting the unnecessary changes extends from Step 3 to Step 7. Glass gives the CCB the authority to eliminate the change and this is done early in the change phase so only the approved changes will proceed further down the change process. A detailed discussion of Perry's seven steps to control the change follows.

1. Step 1--Planning the Change Process

The planning effort is one of providing the needed resources for a change to the responsible people who will do the change. If people are given the start and due date, they can normally do most of the detailed planning for their effort within the allotted time, and can do it by themselves. The change request should provide enough detailed information and the recommended solution should provide sufficient specifications so the change can be implemented without further specification. Perry suggests the use of a "planning the change" worksheet which provides a planning document for gathering resources. It is used to record the type of information, the resources needed, where they can be obtained, and whether or not they have been received.

2. Step 2--Operating During Change Period

The user would like to keep using the old system until the change is implemented and tested. This may or may

not be a problem for the maintenance group depending on the nature of the system and the type of change. In many cases, the old system can run as is until the requested change is installed. However, in some instances the system cannot run as is. Perry mentioned some of the conditions that require immediate action:

- Hang-ups. The system does not run at all in its current status. Potential temporary fixes are to eliminate specific types of data, to provide a temporary solution using a patch, and to operate the system in a manual mode.

- Installation deadline has passed. The change may be delayed. Perry gives an example: new federal tax withholding tables must be installed. Alternate solutions include patching the system, running the system in the current version to produce most of the data, and then rerunning it at a later point in time to provide the correct output. Another solution is to run it in the old mode and then do special programming later to make the necessary corrections.

- Detected error condition. If the system is operating but producing known errors, those errors should be corrected in future runs until the requested change is implemented. Some of the solutions include developing special programs to search and correct an error, manual searching and correcting the error, and inserting a patch in the program.

An essential part of the maintenance change process is to assure that the production version of the application will produce the proper result during the period when the change is being installed. The maintenance analyst must determine whether the change impacts the currect production status of the application.

3. Step 3--Obtaining Needed Access

Even though the maintenance team is assigned to maintain an application system, they are not authorized to have free unrestricted access to the programs and data. To control the maintenance process, access to programs should not be automatic. The access can be controlled by data management or a security officer. Each change or group of changes should require a new version of the program. The maintenance team should have no access to that program once that version has been tested and installed in production, until access is again authorized.

4. Step 4--Establish Performance Criteria

A criteria to measure the success of the change is needed. "Performance" in the broad sense is used to indicate that the system performs in accordance with user requirements. The areas of performance that need to be measured are:

- Functional: The system should perform those functions specified by the user. This includes accuracy, reliability, consistency, completeness, and protection against wrong input data.
- Regression: The change should not negatively impact unchanged portions of the system.
- Stress: The system can handle all the specified data volumes without problems.
- Economy/efficiency/effectiveness: The operating characteristics of the system, such as response time or turnaround time need to be measured before user needs are met.

These performance criteria should be specified in measurable terms. Determining measurable performance criteria can be difficult for users and project personnel. However, both the implementation of change and testing would be easier once the criteria has been established. These criteria should be established before implementing the change. These criteria are useful in determining the type of controls needed in the application system.

5. Step 5--Implementing the Change

Implementing the change includes designing detailed systems, specifying programs and coding. The concerns during this step are:

1) Installing the change in the proper part of the application system.

2) Changing all parts of the system that are impacted by the problem.

3) The continuous performance of the current corrected functions.

Perry gives some tips and techniques to assist the maintenance group in their function including:

- Identify affected data elements.
- Identify programs using those data elements.
- Identify the programs that either create, modify, or delete the affected data elements.
- Identify the external controls over the data elements.
- Identify the programs that edit or audit the affected data elements.

- Code the change in a single module if possible.
- Contain the entire change in one program if possible.

6. Step 6--Test the Change

After the change is made, a complete testing is needed to assure that the entire system will function properly. This is the only effective way to minimize the maintenance change problems. An exhaustive set of test data which is established at the time of development should be used and adjusted whenever a change is made to reflect the change and verify that the entire system still functions correctly after the change has been implemented.

7. Step 7--Documenting the Change

Each change implemented into the system has to be documented. The documentation will become outdated if this step is not performed. A check list of all the documentation that is affected by the change will help the programmers to update the documentation to reflect the change. It is preferred to update documentation after the change has been tested, the reason is that testing may alter the way in which the change is installed. Perry provided a "System Maintenance Documentation Work Plan" form. Its objectives are to identify all areas of the documentation that need to be changed, as well as the individual responsible for making the change.

VI. SYNTHESIS OF A NEW CHANGE CONTROL MODEL

A. INTRODUCTION

In this chapter we will compare and evaluate the two models presented in the previous chapter, then synthesize a new change control model which will combine the best aspects of the existing model with some new ideas.

B. COMPARISON AND EVALUATION

Perry in his change process dealt with most of the actions that are usually needed to control changes, but looking at Figure 5.2, he did not show what the action is if more information might be needed from the groups who request the change. He ignored the fact that some changes might be returned for further details before a decision can be made. Perry did not mention the need for keeping records and tracking the changes as Glass suggested by having a separate error tracking group for this purpose. Another difference: Glass made use of the configuration management group and especially the CCB to evaluate the change request and make decisions, while Perry leaves this entirely to the system analyst. The use of CCB seems to give better evaluation and control especially for large software systems. While Glass limits the initiation of SPR to the test group which in turn submits it to the tracking group for log and routing before

the change is analyzed and submitted for approval to CCB, Perry assumes that approval was granted before the first step (planning for change) is to start. But Perry did not mention how approval is reached and under what criteria or what to do in case of emergency changes which don't usually follow the normal change steps. As mentioned before, Perry included updating documentation as the last step in the change process, while Glass ended his SPR with testing and did not mention that several documents might be affected by the change and needed to be updated to keep the documentation reflecting the current system. Glass also has not discussed what procedure to follow during the time the change is implemented to assure that the old system is in an operational status. He failed to specifically mention that only changes that are within budget and time should be approved, and he seems to assume that it is the responsibility of the CCB to make that decision. Perry provided a good number of forms, checklists, plans, and instructions to help manage and control the change phase of maintenance.

C. NEW CHANGE CONTROL MODEL

It is clear from the previous discussion that neither the Glass nor the Perry change control model is complete and can be considered ideal. The author in an attempt to cover the unaddressed aspects of change control in both models would like to combine them in one model with some modifications which the author feels will help make the change

control process more clear and more effective when applied to large software systems. While acknowledging that there is no one ideal universal change control methodology which will apply to all types of organizations and software projects, the proposed model will include the essential steps that most people agree have to be taken when dealing with large software projects. Even though most of these steps already have been talked about in the literature, the author feels that the proposed model is more detailed and comprehensive. One might argue that this model is too expensive for small organizations who are dealing with small projects and have only a limited number of staff. For these organizations a subset of the model can be used to fit the organizational limitations, using the model as a general guide.

The proposed change control process model describes the procedures that can be used to control changes to software which are crucial to software configuration management. All documents and programs must be baselined and under the control of the software library. Only when approval is received by the librarian can programs or documents be released for revision, and only authorized revised programs and/or documents entered into the software library. The modified model is shown in Figure 6.1 and the steps in this procedure are as follows.

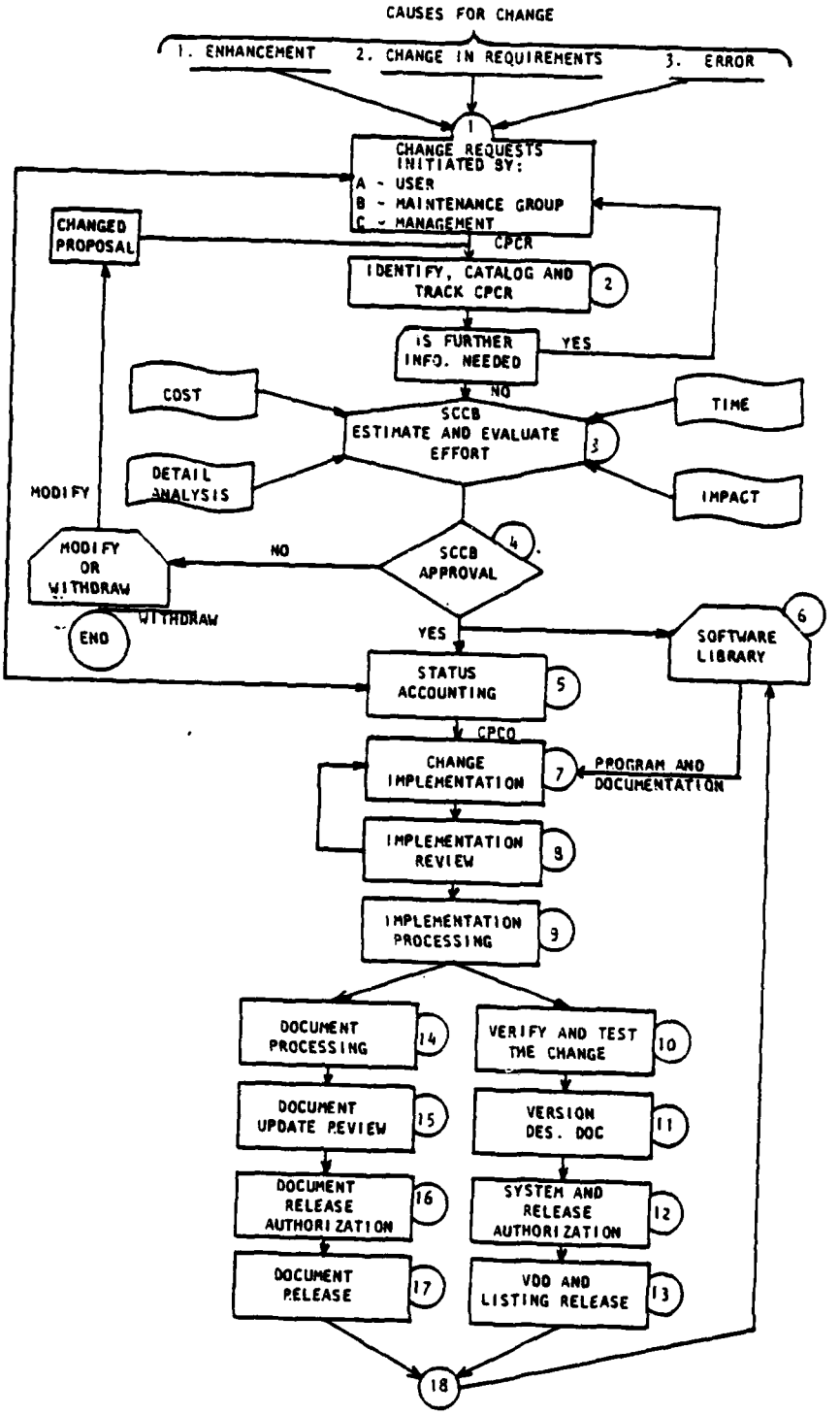


Figure 6.1 Improved Change Control Model

1. Step 1--Computer Program Change Request CPCR

The originator of the change request must submit a CPCR which should include useful information such as definition of the problem, the originator name and organization, reason for the change, the programs or documents affected, priority of this change, and proposed solution if applicable.

2. Step 2--Screen and Track

The problem and proposed solutions are subject to an initial screening to ascertain

a) Is the origin of the problem hardware, software, or both?

b) Is the problem statement sufficiently specific or is further information needed?

Identification of the problem/change together with status is entered into the status/accounting records.

3. Step 3--Evaluate and Estimate the Effort

The SCCB evaluates the problem and/or changes analysing the impact of the change on related software that are governed by the SCCB. Time and cost will be estimated to help in the decision making process.

4. Step 4--Change Approval/Rejection

The SCCB will review the evaluation and assess the criticality of the change consistent with its knowledge of customer needs and priorities. Approval of SCCB constitutes authority for implementing the change. Sometimes a change

request has to be modified in order to be approved. In this case the requester of the change either modifies the original change request and resubmits, or withdraws it.

5. Step 5--Status Accounting

The implementation status is recorded in the accounting data base following the approval for implementation. This action is taken by the SCCB through the software control center (SCC).

6. Step 6--Software Library Releases Programs and Documents

The software library will only release the programs and/or documents after the SCCB has authorized the change.

7. Step 7--Implement the Change

A Computer Program Change Order (CPCO) will be assigned to the programmer(s) responsible for the computer program modules affected by the change. The changed packages are submitted to SCCB for review and approval, at the completion of code, test and document change.

8. Step 8--Implementation Review

The SCCB examines the complete change package to ensure that the change satisfies all requirements and that all elements of the package are consistent with each other. SCCB approval at this point constitutes authority for the SCC to process the change for inclusion on the system program version.

9. Step 9--Implementation Processing

During this step the SCC updates applicable configuration controlled libraries and processes specification/documentation changes. SCC also prepares the preliminary Version Description Document (VDD) which includes a complete inventory of all computer program modules/elements that constitute the system tape together with descriptive information of changes that have been implemented in the to-be-released version of the system program.

10. Step 10--Verify and Test the Change

As a quality measure a series of regression tests are performed after the change to ensure that the system satisfies the performance requirements. Satisfactory completion of system verification is the basis for release of the system version.

11. Step 11--Version Description Document (VDD)

The SCC completes the VDD by incorporating the CPCR and/or CPCO corresponding to changes determined to be necessary as a result of verification testing. Where the preliminary VDD has been subjected to an audit, the results of the audit are addressed by the SCC by correcting discrepancies or obtaining clarification.

12. Step 12--System and VDD Release Authorization

Upon successful completion of the tests, the completed VDD will be reviewed and when satisfied that the

result of the testing and content of VDD reflects changes that had been authorized, the system and associated VDD will be released.

13. Step 13--VDD and Listing Release

The SCC has the VDD reproduced for distribution. In accordance with the established requirements, copies of listings for computer program modules that have been changed or are initial releases, are also obtained for distribution.

14. Step 14--Document Change Processing

Programmer prepared documentation that consists of initial material or changes to existing documents/ specifications are processed by documentation support. In order to release the documentation, a document change order is required.

15. Step 15--Document Review

The completed document changes are reviewed to ensure:

a) Transcription of programmer input is complete and accurate.

b) Final wording is consistent with performance requirements.

c) Audits for traceability to original statement of problem and approved design implementation.

16. Step 16--Document Release Authorization

Documentation will be reviewed for adherence to user agreements and management requirements. Only when the result of the review is satisfactory, documentation is authorized for release.

17. Step 17--Document Release

Final release processing includes the recording of the document release into the release record, reproducing the document, and distributing the copies to the organizations.

18. Step 18--Software Library Accepts New Release

The software library will only accept authorized releases of system, VDD and documents. New back-up copies will be generated and kept in a safe place.

D. SUMMARY

Change control is an essential element in managing a software system which involves many change requests. By controlling the change process using the model/procedures described above, the software maintenance process will be easier to understand and the cost and time involved will be reduced, thereby increasing software productivity.

VII. CHANGE CONTROL TOOLS

A. INTRODUCTION

Software is always changing. Many problems that arise during the software maintenance phase are due to a lack of adequate control and organization of program source code and documentation. In the maintenance phase there are always errors to fix, enhancements to add, and adaptation to new environments throughout the entire software life. This continual modification results in multiple versions of the system. There is not only the current version to change, but also last year's version (which is still supported) and next year's version.

Maintaining and controlling these versions is a difficult job unless a suitable technique and tool is applied. The Source Code Control System (SCCS) is one of the best known systems for dealing with this problem [Ref. 38].

Next we will discuss SCCS and outline its benefits in helping control changes to source code and tracking multi-versions of the system. Also a new tool, Revision Control System (RCS), which promises improvement over SCCS will be presented. Finally, a summary of useful future maintenance tools will be presented.

B. SOURCE CODE CONTROL SYSTEM

The Source Code Control System (SCCS) is a software tool designed to manage source code. It provides facilities for storing, changing, and retrieving all versions of individual modules. When a change is made, SCCS records what the changes are, why they were made, who made them and when. SCCS also keeps the old versions; this allows SCCS to retrieve any version of the text. SCCS handles synchronization of multiple readers and writers, as well as attempting to protect users against interruptions or crashes. In projects with more than one person, SCCS will ensure that no two persons can edit the same file at the same time.

There are two implementations of SCCS: one for the IBM 370 under OS and one for PDP-11 under UNIX [Ref. 39]. SCCS uses the operating system protection mechanism to control the creation and destruction of text.

SCCS treats each module as a set of related sequences of source code, each member of which represents one version of the module. Each set of changes to each module is stored as a discrete delta. The deltas resulting from a series of changes are strung together in a chain [Ref. 7]. Figure 7.1 shows a module which has been changed three times. The source code of the module is accessible at each of the four points at which deltas where added.

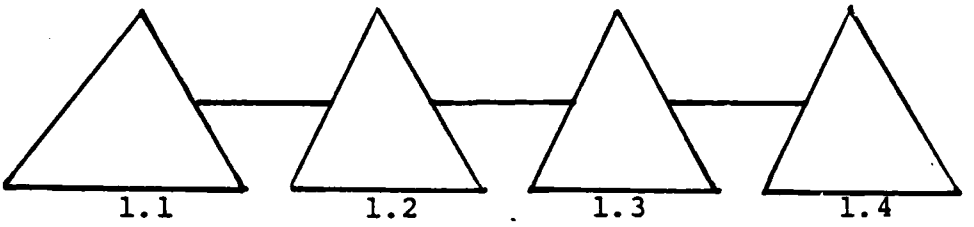


Figure 7.1 Release 1 with Four Levels

When coded, a new module is said to be at release 1. Each delta represents a new level. Deltas are named by their release and level numbers. In Figure 7.1, the first delta represents release 1, level 1, the second represents release 1, level 2, etc. When an enhancement is needed for this module, the programmer makes a copy of the most recent version of the module and then begins modifying that copy. The programmer just adds more deltas to the end of the chain when using SCCS, specifying that they belong to a new release, for example, release 2. As Figure 7.2 shows two

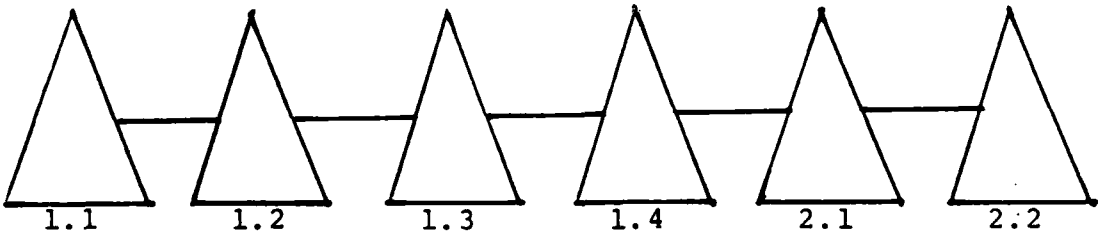


Figure 7.2 Release 2 with Two Levels

new deltas have been added to release 2. Deltas may only be added at the end of the release; the system will not permit a delta to be inserted between deltas 1.3 and 1.4 for example.

Two kinds of special deltas give flexibility in controlling the effect of deltas. The first is optional deltas. Optional deltas in all respects are like normal deltas, except that when added they are associated with an arbitrary option letter. An optional letter would be assigned to specific customers, and those optional deltas would be used to install "temporary fixes" appropriate only for one customer, with the idea that such fixes would be incorporated into the standard product in the next release. Optional deltas can be used for other similar purposes.

The second kind of special delta is one which, when applied, explicitly forces other deltas to be applied or not, by either including or excluding them. A list of deltas to be included or excluded is specified when such a delta is created. Most often the exclusion is used simply to correct mistakes. For example, if after delta 2.3 is added, it is found to be undesirable, the programmer might add delta 2.4 which excludes it. From the view point of control, this form of error correction is safer than allowing the programmer to actually delete a delta, since no potentially necessary information is lost.

A delta which includes and/or excludes other deltas may be optional. Additionally, a delta which includes and/or excludes other deltas may in turn be included or excluded by some other delta. If one delta includes another delta, and the other delta excludes that same delta, the chronologically newer of the two including/excluding deltas has precedence [Ref. 7].

1. Identification

The SCCS identification permits the correct version of source code to be determined from information such as version number, date, time, etc. The source code that was used to make the load module may later be retrieved from this information alone. On some systems where all code is maintained with SCCS, a user can easily identify the version of any program, without examining the source code.

2. Protection

Only authorized programmers can add deltas to certain modules. While a programmer is working in one release it is locked and no one can add a delta to a locked release. The only access to a module is through SCCS.

3. Documentation

The SCCS automatically records what the change is, who made it and when it was added (date and time to the nearest second).

C. REVISION CONTROL SYSTEM

Revision Control system (RCS) is a software tool that helps like SCCS in managing multiple revisions of text. It was developed by Walter Tichy at Purdue University and was intended to improve the deficiencies of SCCS [Ref. 8]. The basic function of RCS is that it manages revisions of text. RCS stores and retrieves multiple revisions of text, logs changes, identifies revisions, merges revisions, and controls access to them. The space overhead of storing multiple revisions is minimized by saving only the differences between successive pairs. The SCCS is limited in that it treats each system part in isolation and does not consider configurations of parts. RCS avoids this limitation, and corrects some other design flaws according to Tichy. SCCS is implemented with forward and merge deltas while RCS uses reverse and separate deltas which improves its performance considerably. Another improvement is that the user of RCS can specify the working file, or the revision file, or both when manipulating files rather than referring to the SCCS database file names. The commands were also more mnemonic.

The access control in SCCS is sometimes too strict. If a revision is locked, it is impossible to force the lock unless one has extra privileges which will leave no trace of the action. RCS has a more flexible approach for forcing

the lock with a special command which will always send a message to the mailbox of the user whose lock was broken. Thus, RCS allows work to proceed while delaying the resolution of the update conflict.

RCS identification is not complicated. Program versions can be named instead of being restricted to numbers as in SCCS. SCCS provides no symbolic revision names making it awkward to specify which revisions constitute a specific configuration if the revisions do not share the same numbers.

1. The Revision Tree

In some situations where two programmers modify the same revision and want their modifications to remain separate, RCS is instructed to maintain two revisions with a common ancestor. These two revisions may again be modified several times giving rise to a tree with two branches [Ref. 8]. Figure 7.3 illustrates an example tree with four branches not counting the trunk which is the main branch. The revisions are numbered 1.1, 1.2, . . . , 2.1, 2.2, etc.

Every revision in the tree consists of the following attributes: a revision number, a check-in date and time, the author's identification, a log message, and the actual text. All these items are determined at the time the revision is checked in.

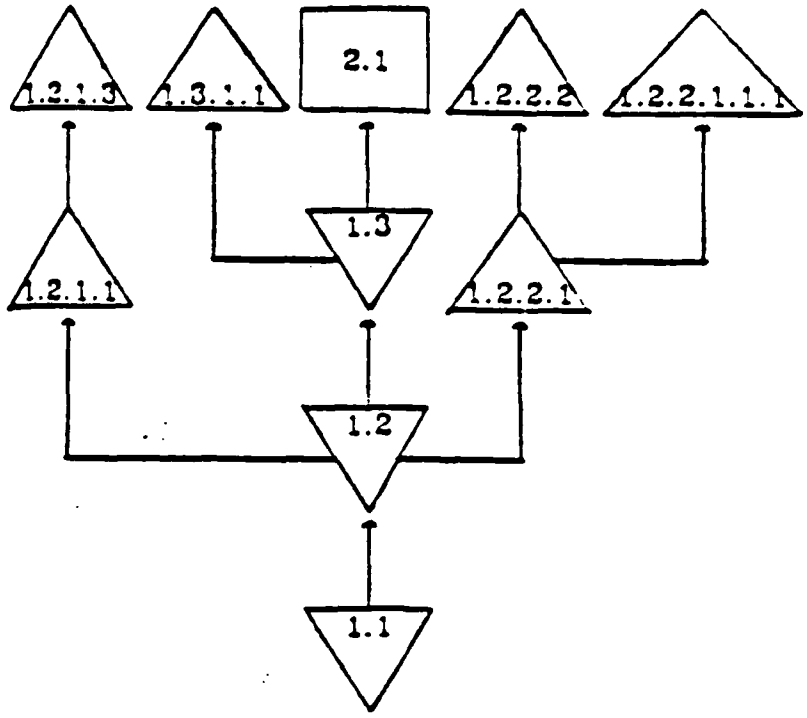


Figure 7.3 A Revision Tree with Forward and Reverse Deltas

2. RCS Auxiliary Commands

There are several auxiliary RCS commands. There is a command which displays the log entries and other information about revisions in a variety of formats. Also there are commands which shrink and expand the access list, change the symbolic tables, reset the state attributes of revisions, and delete revisions. There are facilities to lock and unlock revisions, as well as to "force" locks. Forcing a lock is sometimes necessary if a programmer forgets to release his locks.

A special option permits the joining of revisions. The resulting revision can be edited or checked back in as a new revision.

D. OTHER USEFUL MAINTENANCE TOOLS

In a project conducted for the Rome Air Development Center (RADC) by Advanced Information and Decision System (AI&DS) the final technical report [Ref. 40] defined and proposed some software maintenance tools that if implemented, would help increase productivity, improve reliability, and lower costs. The group surveyed the literature and conducted extensive interviews with maintenance programmers and managers at three Air Force C3I sites. Also a questionnaire, designed to assess maintenance problems in more depth was sent by the group to selected personnel at all the interview sites.

A summary of these tools is below:

- Programming Manager assists the programmer by systematically applying administrative and technical policies, as well as helping apply both general and application-specific programming techniques and methods.
- Intelligent Editor provides facilities for manipulating programs at several conceptual levels (e.g., textual, syntactic, semantic, and intentional), and provides an intelligent interface to other tools.
- Documentation Assistant is a tightly woven collection of tools for creating, structuring, maintaining, and accessing all forms of documentation.
- Style Analyzer checks programs for adherence to programming standards and style guidelines (which are expressed with a specification method that is independent of the analysis process itself).

- Metrics Tool Set provides tools for measuring, analyzing, and assessing various properties of software systems over their lifetime.

- Annotation Language is a method for extending a programming language by allowing annotations which specify state properties and other aspects of programs that cannot be conveniently expressed in the programming language itself.

- Change Propagation Detector analyzes a program for effects of program changes.

- Test Cast Analyzer allows the output produced by test runs to be automatically checked for correctness, based on a formal (or informal) specification of what the output should look like.

- Intelligent Tutor uses a knowledge-based approach to teach programmers about programming languages and programming environments, using the tools themselves.

Several of the proposed tools are less comprehensive, attempting to solve smaller problems. These tools provide capabilities that may already be available, but they also employ advanced techniques which provide much greater depth and sophistication than existing tools.

VIII. CONCLUSIONS AND RECOMMENDATIONS

Since software maintenance dominates the software life cycle in terms of effort and cost, it is vital to find effective ways to reduce or make more efficient the software maintenance effort. Otherwise, maintenance efforts may absorb all programming resources, leaving nothing for the development of new software.

Maintenance tasks are often more difficult than new development tasks and thus require very skillful programmers and good management control. Changes have to be managed carefully in order not to jeopardize the integrity of the software or increase its complexity.

Since it is critical that software change control be emphasized, accurately determining the precise type and amount of control for software maintenance is vital. Software configuration management techniques are the result of that determination and should be incorporated into software projects.

The management of software maintenance has to establish certain policies and procedures to control the maintenance process especially controlling the source change through the use of software tools which promise effective results. Changes should not be granted just because the users ask for

them. An evaluation of the effort, cost, impact on the rest of the system and the risks of doing the change, has to take place before implementing the change. Only those changes that are cost-effective and needed should be approved.

By implementing the change control model proposed by the author, maintenance can be managed with a high degree of control and visibility. Configuration change control is an essential element in managing projects which contain numerous change requests.

By implementing software tools which control access and changes to source code such as SCCS and RCS, control can be made easier. Changes can be tracked and different versions can be identified and retrieved. Also simultaneous modification of the same file by more than one programmer is prevented.

A framework has been proposed for program maintenance change control. This framework should be implemented and tested on various sizes and types of software systems to discover its effectiveness in achieving the organizational goals and how much savings in time and money is achieved.

One of the problems of maintenance is the lack of knowledge about maintenance. Maintenance awareness and knowledge must be transferred to students, programmers, and managers. Computer science curricula should present the

awareness that software must eventually be maintained. Computer science courses should be devoted to analyzing, debugging, and solving problems in existing software. One will appreciate good development methodologies after he has worked with poorly designed and coded programs.

Software maintenance management must be developed more fully to give the manager the techniques and knowledge needed to ensure that the task is done correctly and economically. Management principles can be applied to software management, but the applications differ because of time scales, urgent decisions, levels of effort, and the changing nature of software.

Further research is needed in software maintenance techniques and tools. There are few developed techniques and guidelines for organizing, working, or managing a maintenance effort. More data on software maintenance activities is needed to allow the development of a comprehensive process model of software maintenance. This would help in understanding what maintenance is and how to control it. Such an understanding would point the way to administrative and methodological improvements, as well as identify critical needs that could be addressed by maintenance tools.

LIST OF REFERENCES

1. Boehm, B. W., "Software Engineering," IEEE Transactions, Computers, pp. 1266-1271, December 1976.

2. Lintz and Swanson, Software Maintenance Management, Addison-Wesley, Reading, MA, 1980.

3. Martin, James and McClure, Carma, Software Maintenance: The Problem and Its Solutions, Prentice-Hall Publishing, Inc. 1983.

4. Boehm, B. W., Software Engineering Economics, pp. 35-55 and pp. 641-690, Prentice-Hall Publishing, Inc., 1981.

5. Bersoff, E. Henderson, V. and Siegel, S., Software Configuration Management, Prentice-Hall Publishing, Inc., Englewood Cliffs, NJ, 1980.

6. Buckel, J. K., Software Configuration Management, M.A., FBCS, 1982.

7. Rochkind, Mark, "The Source Code Control System," IEEE Transactions on Software Engineering, Vol. SE-1, No. 4, December 1975.

8. Tichy, Walter F., "Design, Implementation, and Evaluation of Revision Control System, IEEE, 6th International Conference on Software Engineering, pp. 58-67, September 1982.

9. Comptroller General Report to Congress of the United States, FGMSD-80-38, Wider Use of Better Computer Software Technology Can Reduce Cost, 29 April 1980.

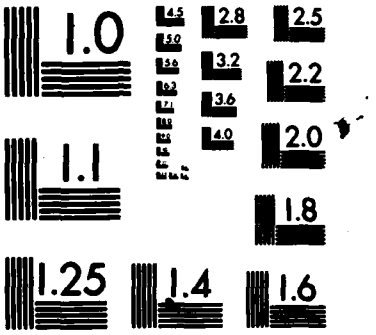
10. Parikh, Girish, "The World of Software Maintenance," IEEE Tutorial in Software Maintenance, p. 7, 1983.

11. Owitz, M. Zelk, Principle of Software Engineering and Design, pp. 2-11, Prentice-Hall Publishing, Inc., Englewood Cliffs, NJ, 1979.

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

AD-A171 391 CONTROL AND MANAGEMENT OF THE SOFTWARE MAINTENANCE CHANGES PROCESS(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA N A AL-SUBAIEI JUN 86 2/2

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

13. Lientz, B. P., Swanson E. B., and Tompkins, G. E., "Characteristics of Application Software Maintenance," Communication of the ACM, Vol. 21, No. 6, pp. 466-471, June 1978.

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

15. National Bureau of Standards Report NBS-500-106, Guidance on Software Maintenance, by Roger Martin and William Osborne, 1983.

16. McClure, C., Managing Software Development and Maintenance, Van Nostrand Reinhold Company, New York, 1978.

17. Myers, G., "Selection from Composite/Structure Design," Techniques of Program and System Maintenance, pp. 205-206, Ethnotech, Lincoln, NE, 1980.

18. Boehm, B., Brown, J., Kaspar, H., Lipow, M., MacLeod, J., and Menit, M., Characteristics of Software Quality, pp. 3-1 to 3-26, TRW/North-Holland Publishing Company, New York, 1978.

19. Brooks, F., The Mythical Man-Month, Addison-Wesley, Reading, MA, 1975.

20. Sommerville, Ian, Software Engineering, Addison-Wesley, London, 1982.

21. Kepner, Charles H., Tregoe, Benjmin B., The Rational Manager, McGraw-Hill Book Company, New York, 1965.

22. Reutter, J., "Maintenance is a Management Problem and Programmer's Opportunity," AFIPS Conference Proceedings on 1981 National Computer Conference, Chicago, Vol. 50, pp. 343-347, May 4-7, 1981.

23. Porter, W., and Perry, W., EDP Controls and Auditing, p. 10, Wadsworth Publishing, 1981.

24. Mitchell, David, Controlling Without Bureaucracy, McGraw-Hill Book Company, 1979.

25. Stout, Russell, Jr,, "Management or Control?", The Organizational Challenge, p. 14, Indiana University Press, Bloomington, 1980.

26. Merchant, Kenneth A., Control in Business Organization, Pitman Publishing, Inc., 1985.

27. Williamson, Oliver E., Corporate Control and Business Behavior: An Inquiry into the Effects of Organizational Form on Enterprise Behavior, Prentice-Hall Publishing, Inc., Englewood Cliffs, NJ, 1970.

28. Pope, A. Berett, "Software Configuration Management: A Quality Assurance Tool," IEEE Engineering Management Conference, pp. 56-66, 1983.

29. Samaras, Thomas T. and Czerwinski, Frank L., Fundamentals of Configuration Management, pp. 27-58, Wiley-Interscience, 1971.

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

31. Jensen, R. W. and Tonies, C. D., Software Engineering, pp. 39-40, Prentice-Hall Publishing, Inc., Englewood Cliffs, NJ, 1979.

32. Searle, L. V., Air Force Guide to Configuration Management, Systems Development Corp., Santa Monica, CA 1976.

33. Brayn, W., Siegel, S., and Whiteleather, G., "Auditing Throughout the Software Life Cycle: A Primer," Computer, Vol. 15, pp. 56-67, March 1982.

34. Department of the Air Force Military Standard MIL-STD-483A, Configuration Management, 4 June 1985.

35. Parnas, D., "Designing for Ease of Extension and Contraction," IEEE Trans. on Software Engineering, Vol. SE-5, No. 2, pp. 129-137, March 1979.

36. Perry, William E., Managing Systems Maintenance, Prentice-Hall Publishing, Inc., 1983.

37. Glass, Robert L. and Noiseux, Ronald A., Software Maintenance Guidebook, Prentice-Hall Publishing, Inc., 1981.

38. Glasser, A., "The Evolution of the Source Code Control System," Software Engineering Notes, 3(5), pp. 122-125, 1978.

39. Ritchie, B. M. and Thompson, K., "The Unix Time-Sharing System," Commun. Ass. Comput. Mach., Vol. 17, pp. 365-375, July 1974.

40. Dean, Jeffrey and McCune, Brian, Advanced Tools for Software Maintenance, RADC-TR-82-313, Final Technical Report, December 1982.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5000	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
4.	Dr. Gordon Bradley, Code 62BZ Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	3
5.	Dr. Bruce MacLennan, Code 52ML Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
6.	B. General Khaled Bin Sultan Bin Abdulaziz Commander, Royal Saudi Air Defense Forces Ministry of Defense and Aviation Riyadh, Kingdom of Saudi Arabia	2
7.	Commander, Center of Maintenance and Technical Support for Royal Saudi Air Defense Forces Post Office Box 5380 Jeddah, Saudi Arabia	2
8.	Commander, Air Defense Forces Institute Royal Saudi Air Defense Forces Jeddah, Saudi Arabia	1
9.	CAPT Nasser A. Al-Subaiei Post Office Box 5380 Jeddah, Saudi Arabia	2

10. Library 2
King Abdulaziz University
Jeddah, Saudi Arabia

11. Library 1
Kind Saud University
Riyadh, Saudi Arabia

END

DTIC

9-86