AD-A118 211 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
A MODEL FOR ESTIMATING TACTICAL SOFTWARE MAINTENANCE REQUIREMEN--ETC(U)
JUN 82 W H MERRING

UNCLASSIFIED NL

1 OF 1
AD
A118211

END
DATE
FILMED
9-82
DTIC

②

AD A118211

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A MODEL FOR ESTIMATING
TACTICAL SOFTWARE MAINTENANCE
REQUIREMENTS

by

William H. Merring, III

June 1982

Thesis Advisors: D. C. Boger

R. Modes

82 08 16 171

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A118211	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A MODEL FOR ESTIMATING TACTICAL SOFTWARE MAINTENANCE REQUIREMENTS		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June 1982
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) William H. Merring, III		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June 1982
		13. NUMBER OF PAGES 51
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release: distribution unlimited

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

18. SUPPLEMENTARY NOTES

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

Software Maintenance
Tactical Software Maintenance
Software Microestimation
Software Metrics

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

Recent studies have pointed to the increasing burden that is software maintenance. The maintenance of tactical systems software will demand resources that exceed those expended during the development phase as their numbers and time-in-service increase. This increased demand for resources requires more effective management of the maintenance phase and development of the software with maintenance in mind.

This thesis presents those items that should be considered and utilized

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE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

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (continued)

during the development phase to reduce maintenance costs over the life-cycle of the system. It also presents a model that uses the known configuration of the program to estimate the maintenance personnel requirements for that system. These requirements will be estimated from the beginning of the maintenance phase to its completion. The model utilizes the technique of measuring the characteristics of the software to obtain the estimation.

A Model for Estimating Tactical Software Maintenance Requirements

by

William H. Merring, III
Captain, United States Marine Corps
B.S.I.E., Rutgers University, 1976
M.S., University of Southern California, 1981

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS
from the
NAVAL POSTGRADUATE SCHOOL
June 1982

Author: William H. Merring III

Approved by: Dan C Boger
Thesis Co-advisor

[illegible]
Thesis Co-advisor

[illegible]
Second Reader

[illegible]
Chairman, Department of Administrative Sciences

W. M. Woods
Dean of Information and Policy Sciences

ABSTRACT

Recent studies have pointed to the increasing burden that is software maintenance. The maintenance of tactical systems software will demand resources that exceed those expended during the development phase as their numbers and time-in-service increase. This increased demand for resources requires more effective management of the maintenance phase and development of the software with maintenance in mind.

This thesis presents those items that should be considered and utilized during the development phase to reduce maintenance costs over the life-cycle of the system. It also presents a model that uses the known configuration of the program to estimate the maintenance personnel requirements for that system. These requirements will be estimated from the beginning of the maintenance phase to its completion. The model utilizes the technique of measuring the characteristics of the software to obtain the estimation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

A. CONTEXT OF THE STUDY AND BACKGROUND

The maintenance of software that has been acquired and made operational by the government is a growing concern. The number of tactical computer programs in service and those under development will increase the future requirement for more effective and detailed planning of all resources in the software maintenance area. Not only the number of these systems, but also the length of time that these systems remain operational will add to the requirement for more effective maintenance management. This management will be especially demanding with regards to personnel requirements since this will potentially be the most expensive and difficult single resource to manage.

The Marine Corps Tactical Systems Support Activity (MCTSSA) has the Marine Corps responsibility for maintenance of the software of tactical systems [Ref. 1]. The objective of this research is to present the current ideas in software engineering and maintenance. This study was completed with MCTSSA's function in mind, but this should not be construed as limiting the tenets outlined in this paper as being limited to MCTSSA. They should be applicable in varying degrees to all software projects.

It is important at this point to realize that good maintainability begins very early in the software development process and can only be planned from the beginning of the system life-cycle. Attempts to improve maintainability late in the project are potentially hazardous to the integrity of the software and extremely can be expensive. The inclination on the part of the developers to reduce near-term costs at the expense of maintainability almost ensures that the effort expended later in the life-cycle will far exceed any immediate advantage.

Most of the literature available on software maintenance and its related aspects falls into the area of general purpose computing. Little of the current information available specifically addresses tactical computer software. While this initially appeared to restrict sources of information, it became apparent that, even though there are some definite differences between tactical and general purpose computing, they are less important than the similarities. The particularly demanding parts of tactical systems are the real-time requirements or scheduling constraints, and the "critical" or "life-and-death" nature of their decisions. In the bulk of general purpose software these features are either not present or not present to the same degree. [Ref. 2]. The software will to a great extent be similar in composition even if the requirements differ. Additionally, many of the processors that the tactical systems are based on were originally designed for a commercial or general purpose systems.

Software maintenance, for the purposes of this paper, will be defined as those actions which are taken to repair or beneficially alter the software in a system that leaves the majority of the instructions in the program unaltered and the designed function of the system intact. Software maintenance has been divided into two types, corrective maintenance and adaptive maintenance [Ref. 3].

Corrective maintenance is that maintenance which is required to repair errors or improve the functioning of the software without altering its primary functions. The latter part of this type of maintenance would concern itself with essentially perfecting the programs or making them more machine efficient. The repair portion would consist of fixing errors. These errors can range in degree from an error that results in total failure of the system to an error that is bothersome but has no effect on performance [Ref. 4]. The second type of maintenance is that of enhancement or adaptive maintenance. This occurs when a minor change in the function of software is desired. This change can improve the system by increasing its capabilities or changing its operation to the form that the user desires at some time after the system has been made operational. The changes made should leave most of the system's software as it was.

Software maintainability is that degree to which the software can be changed or corrected. It is the degree of ease the maintainer has in understanding the software and then applying the proper corrective action or integrating the desired enhancement. Software reliability is the extent to which the program performs its designed functions accurately and in a timely manner. These two concepts will be related to each other throughout the life of the software.

B. FORMAT OF THE THESIS

This paper has been broken into two main sections. The first section will describe those methods that should be used early in the life-cycle of the software. The objective of this section is to present those ideas that should have their greatest effect on reducing the total life-cycle cost of the software with emphasis on reducing the costs of the maintenance phase. It is important to stress that extra time or money spent early in the project can reduce costs later in the life-cycle of the software at an extremely favorable rate.

The second section of the paper will deal with presenting a proposed model for the predicition of the personnel required for maintenance of a system's software once MCTSSA assumes full responsibility for this function. An important feature to note in this model is that it is based on the known and quantifiable parts of the software. The programs have already been written and thus MCTSSA will know what they consist of. This fact alone should aid the predictive capabilities of the model.

II. DESIGNING SOFTWARE FOR MAINTAINABILITY

A. SOFTWARE LIFE-CYCLE

The software life-cycle can be divided into six distinct areas. These are Requirements Analysis, Specification, Design, Coding, Testing, and Operations and Maintenance (see Figure 2.1) [Ref. 5]. Each of these areas can be further broken into sub-areas and often overlap. Only those items that are pertinent to maintenance will be developed.

B. REQUIREMENTS ANALYSIS

During the requirements analysis the type of system and its capabilities are identified. This process takes place between the user and the developer. Although in the past this phase has been considered less important, it is both critical to successful maintenance and difficult. It is critical because one must ensure that the user's requirements are met and it is difficult because this phase has not lent itself to the kind of structuring that allows a cookbook or step-by-step approach. The user and developer during this phase attempt to determine the user's needs, a step which is often difficult to accomplish with the technical tools available to conduct the project [Ref. 6].

Items that are important to consider during this phase which improve subsequent maintainability include, first, identifying the maintenance group (in this case MCTSSA) and including them in the review of the system requirements [Ref. 7]. The purpose of this is twofold, first, to determine the maintenance facility's capability to support maintenance and to allow the maintenance managers to begin

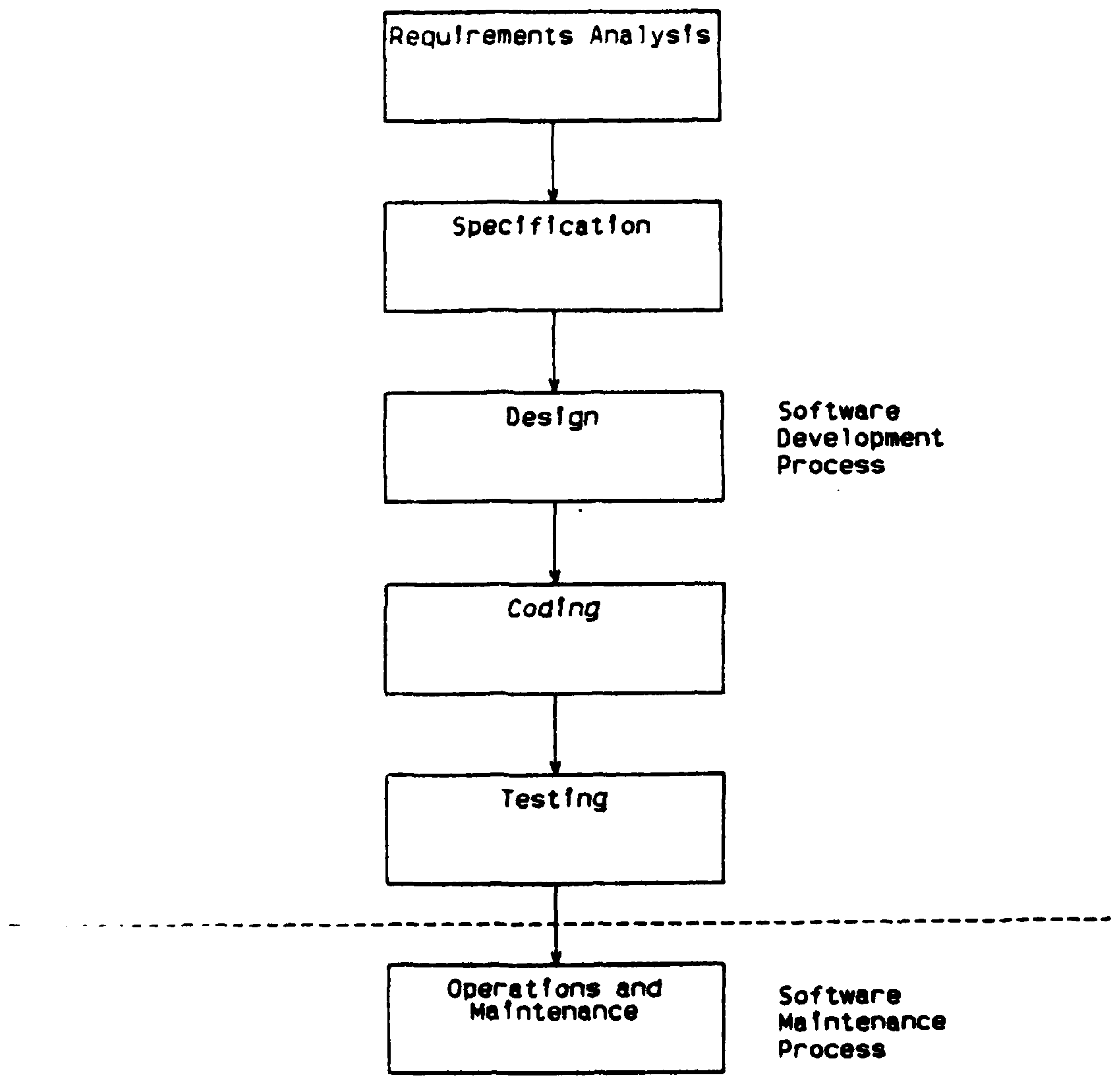

Requirements Analysis
Specification
Design
Coding
Testing
Operations and Maintenance
Software Development Process
Software Maintenance Process

FIGURE 2.1 Software Life Cycle Phases.

Source: McClure, Carma L., _Managing Software Development and Maintenance._ Van Nostrand Reinhold Company, Figure 3.1, p. 33.

planning for the maintenance phase of the project in relation to the other systems currently operational. Second, it is at this time that thought should be given to developing schedules of priorities, enhancements, and resource allocation. The enhancement portion would be extremely difficult to define precisely, but planning for it needs to be conducted in some form as experience has shown that this accounts for, on the average, forty-two percent of the maintenance effort [Ref. 8]. This early identification of enhancements should afford the maintenance facility enough lead time to begin considering its support requirements. The previous experience of the facility in dealing with enhancements should also allow it to make an estimate of the enhancement rate. The development of enhancement estimation should include data gathered from both the vendor and the facility. This requires the establishment of a database that deals with these areas and is readily available to the managers of the software maintenance facility.

C. SPECIFICATION PHASE

It is during the specification phase that the functions of the system are defined. This has to be done with the user to ensure that his requirements will be met. Failure to accomplish this will result in costly corrections either before acceptance when discrepancies arise during testing or later when adaptive maintenance has to be done to bring the system in line with the user's needs. Figure 2.2, the information for which was obtained from the Software Maintenance Guidebook by R. L. Glass and R. A. Noiseux, shows how the cost per error increases as the life cycle progresses.

This phase should be conducted between the user and the developer to ensure that both understand the potentials of

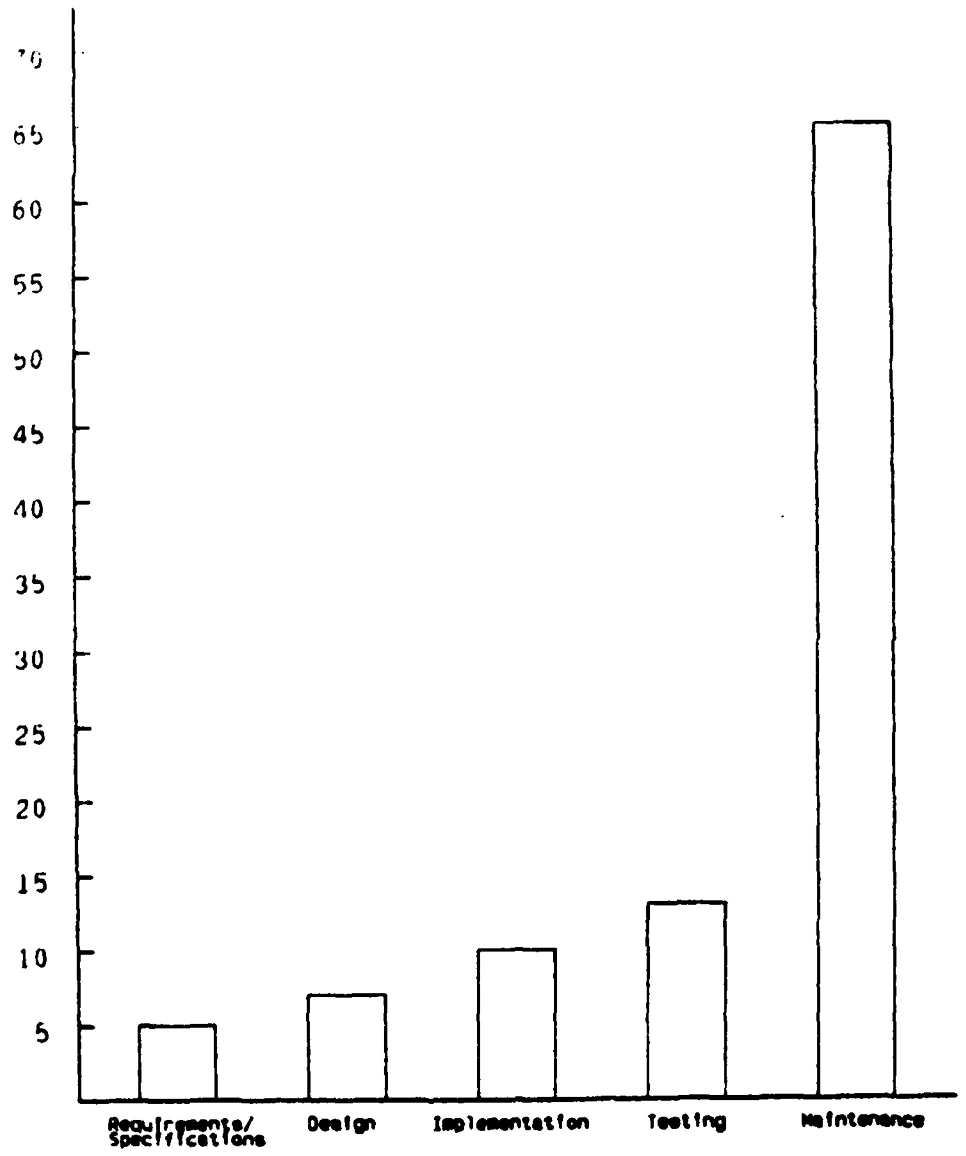


Figure 2.2. Software life cycle:
per error fix cost per phase.

Source: Glass, R. L., and Noiseux, R. A., Software Maintenance Guidebook, Prentice-Hall, Inc., Figure 1.1-4, p. 10.

the system. The user so that unreasonable demands or expectations are not made and the developer to ensure he understands what the user requires and explain what is within the capabilities of the current technology.

The specifications give a concise description of the system's functions to both the user and the developer [Ref. 9]. The completeness and correctness of these specifications will govern the entire project. Good specifications will afford management better estimates of the magnitude of the project for scheduling purposes. Conversely, poor specifications will result in software that cannot be expected to perform adequately or even be useful for the increased costs that it will create. It is during this phase that structured programming should be incorporated into the project to specify the structure of the software for the design phase.

During this phase the maintainer should be included in review of the specifications and he should evaluate the impact on current systems [Ref. 10]. The latter allows the maintenance facililty to refine the planning for maintenance begun during the requirements analysis phase and the former allows the maintainer to provide insight as to what sections of the program can be provided through reused code. It is at this time the identification of those areas that could be implemented through the utilization of reused code should begin.

D. DESIGN PHASE

The design phase is when the structure of the software is actually delineated. The designer develops in detail a structured hierarchy of modules. This phase is critical as studies have identified that almost sixty-four percent of the software's errors have arisen from poor design

[Ref. 11]. A well-structured design will either eliminate errors or facilitate detection of design errors early in the life-cycle of the system when they are least expensive to fix. This is also the phase during which the most widely accepted methods of ensuring maintainability are instituted.

Structured design is the first of these methods. Structured design consists of following an established set of procedures for accomplishing the design phase. It establishes the framework for the software and, when properly completed, facilitates maintenance The structure or framework of the modules should be hierarchial with control flowing from top to bottom in a logical manner with no level calling on a higher level. The top level should be the highest level of control logic present [Ref. 12].

Part of this structure is the design modules, which are those modules that are constructed during the design phase. They have been determined to improve maintainability by almost eighty-nine percent of their users [Ref. 13]. The key aspect of the modules is that they should perform a single function. This reduces their complexity and allows for both easier error checking and error correction. The aspect of a single function is important to maintenance as it makes the module easier for the maintainer to understand what the module is doing. The capacity of the system to use previously coded modules is increased through this method. Additionally, the flexiblity of the system is increased through the plug-in capabilities inherent in modular design.

During this phase it is again important for the developer and the prospective maintainer to maintain close contact. The maintainer is able to provide insight as to what and how the current systems are performing and especially to provide information on what functional modules have already been coded and are available from previous projects. The object here is to reduce the total effort and possibly eliminate any pitfalls that have been experienced in previous designs.

Documentation of the program increases in importance as the development of the system progresses. The developer should ensure that all steps of the process are explained fully. All parts of the hierarchy and modules should be explained completely to ensure adequate understanding of both their functions and methods of implementation. This will considerably ease the maintainer's burden when he has to correct errors or enhance the system much later in the life of the system when those who developed the system are not available for explanations as to why and how. The maintenance personnel can provide valuable input to the documentation design by providing information on those types that have proved especially helpful and easy to use.

E. CODING

This phase is possibly the best understood phase in the entire life-cycle. However, there are items that need to be particularly adhered to if maintainability is to be obtained. The first of these is widely recognized as aiding both development and maintenance. It is the use of a high order language. The greatest contribution of a high order language is that it adds readibility to the program, thus increasing the understanding of the code. Another technique that will facilitate understanding of the code is the use of structured programming. This is the grouping of similar modules and the use of such techniques as indentation of inside a program to increase readability.

Reused code has excellent potential to reduce the life cycle costs of software. It can do this through reducing coding and testing of modules, and consolidation of maintenance through reduction of the total number of unique modules that need to be maintained. It has been applied in limited situations with significant improvements in productivity and reductions in development and maintenance costs [Ref. 14]. The maintenance facility would be required to maintain a library of current modules that are available for use and in operation. It would also maintain the data on those modules in terms of error rates, system locations and other important information.

It is during this phase that reused code is inserted into the program where it was identified as being suitable. Through the reuse of code, not only the coding time but also the testing effort is reduced. The reused code has been tested and implemented in other systems and has been proven. There should be many areas in each new program that provide the opportunity for the reuse of code. A reduction in required maintenance should occur as one module is repaired, the change can be applied to all systems using that module [Ref. 15]. The primary advantage in terms of maintenance is the reduction in the probabilty of errors being generated by that module.

Upon completion of the coding and the checking of the module by the programmer, that module should be passed on to be checked by another individual. This checker should use a checklist that identifies the common errors that arise in programs [Ref. 16]. The use of the checklist will improve the productivity of the inspection process greatly. An example of an inspection checklist can be found in T. Gilb's book on Software Metrics on p. 59.

A technique that shows much promise in increasing both the maintainability and reliability of software is the use of dual code. The dual code technique would be implemented during this phase of the life-cycle and consists of utilizing the structure developed during the design phase to independently construct two sets of code. While it would appear to increase costs by a factor of two, it has in its limited application increased costs over the life-cycle from five to ten percent in cases where no future benefits have been obtained. In most cases, however, a net cost savings of up to fifteen percent or a substantial increase in the quality of the code produced has been realized [Ref. 17].

The advantages of dual code can be manifold. The first, and most important in terms of tactical systems, is the increased quality of the software produced. The reason for this is that the two sets of code will check each other. The results they produce can be checked to determine if there are differences between them and thus possible errors. Therefore a check on the quality of the coding is provided. This technique would have one program essentially do the work of the desk tester, thus automating this step in the process [Ref. 18]. This automation of the desk testing process should increase the dependability of the checking that is conducted. Dual code is also used in conjunction with the bebugging technique, which will be explained later.

F. TESTING PHASE

The testing phase can be broken into three parts: unit level, integration, and system testing. Unit level testing is done for each module to determine if it functions properly. Integration testing is completed next and is done in either a top-down or bottom-up fashion. It ensures that the modules will work properly in the program environment. The system test is completed next and is done to ensure that the system meets the specification it was designed for.

The maintainer should be involved in this entire phase to give advice on the test methodology. He has current information that concerns systems that are on-line and is able to highlight likely areas for extra attention during testing.

Most important during the testing phase is the documentation of the testing. This documentation should provide information on the error rate of each module, the type of error, and difficulties with the overall system. Also included should be data on the manpower and resources required on the project to date, broken down by modules if possible. Table I contains an example of the data that should be included on the program's test history. This information is important to the maintainer as it will give him some idea as to possible trouble spots in the software and an overall idea as to the difficulty he will experience in maintaining the entire program. A system that is difficult to maintain early in its life time will continue to be difficult throughout its entire life and needs to be identified as such as early as possible.

G. MOVEMENT INTO THE MAINTENANCE PHASE

During a study conducted by Lientz and Swanson, it was determined that the best organization for conducting maintenance is one that performs that function solely. This facility should be separate from that of development. They gave as possible reasons, increased efficiency and greater control of efforts and reduced costs that arise from this specialization [Ref. 19]. Additional benefits can occur through the possible career enhancement of programmers who become involved with maintenance.

A tradeoff of a separate maintenance function as compared to one integrated with development is that the productivity of the maintainers declines when fewer of the original developers are involved in maintenance [Ref. 20].

TABLE I

Program Test History.

Unit Test History

Number of Modules Unit Tested
Average Number of Unit Tests Executed per Module
Number of Errors Discovered during Testing
Average Number of Errors Discovered in a Module (UAEM)
Total Number of Statements Modified to Correct Errors
List of Modules in which the Number of Errors Discovered Exceeds UAEM
Types of Errors Discovered
- -Hardware Failure
- -Software Reaction to Hardware Failure
- -Coding Error
- -Design Error
- -Specification Error
- -Logic Error*
- -Computational Error*
- -Data Error*

Average Length of Time to Discover and Correct an Error

Integration Test History

Number of Integration Tests Executed
Number of Errors Discovered during Integration Testing
Average Number of Errors Discovered in a Module (IAEM)
List of Modules in which the Number of Errors Discovered Exceeds IAEM
Total Number of Statements Modified to Correct Errors
Total Number of Modules Modified to Correct Errors
Types of Errors Discovered
Average Length of Time to Discover and to Correct an Error

System (Acceptance) Test History

Number of System (Acceptance) Tests Executed
Number of Errors Discovered during System (Acceptance) Testing
Average Number of Errors Discovered per Module (SAEM)
List of Modules Modified to Correct Errors
Types of Errors Discovered
Average Length of Time to Correct an Error

---

*Error type added by author of this thesis

Source: McClure, Carma L., _Managing Software Development and Maintenance_, Van Nostrand Reinhold Company, Table 3.2, p. 64.

This can be partially compensated through the use of a "maintenance escort" [Ref. 21]. This escort will take part in the development of the software and, when the system's responsiblity for maintenance is transferred, he goes along with it to provide the needed experience to reduce the initial maintenance effort and improve the learning of the maintainers.

Since programmers will not be constantly in demand on a specific project, the organization should be constructed such that departments with experienced personnel are organized around a specific functional area of software, such as input, arithmetic, output, signal processing or data display types of software. This would allow the programmers to become experts on specific areas. The departments would consist of functional areas that are common to all projects and allow programmers to become experienced with that type of function. The departments would send the required people to the requesting projects on an as-requested basis. To ensure familiarity with the project, specific programmers would be allocated to certain projects on a consistent basis. The objective of this system is to obtain greater utilizaton of experienced programmers, the alternative being their complete devotion to a specific project with the resulting under utilization of their skills and abilities.

The management of software maintenance is unique in many respects and requires attention to some special areas. First, it should be realized that about twenty percent of the time the maintainer will actually be employed in corrective maintenance [Ref. 22]. The remainder will be employed in conducting adaptive maintenance. This is important, in that, while it may be difficult to fix bugs in programs, it is more difficult and costly to add enhancements to the software such that it meets the same stringent requirements of the original product. The maintenance of software is a repeating cycle that requires the same steps, although on a smaller scale, that were conducted during the development phase. Figure 2.3 shows the recommended cycle for including enhancements.

It is important that the addition of an enhancement take place in much the same manner as the development process to ensure that the quality of the software is maintained. The maintainer needs to document changes, employ structured programming and modularization in much the same way. Failure to do so will throughly destroy a good program and even shorten its possible useful life. Critical to conducting adaptive maintenance is the determination of whether it is worth the effort to add the enhancement. An evaluation should be conducted on each proposed enhancement, to determine if the potential benefits exceed the possible long term costs. This process when used on small scale enhancements may prove infeasible, but it should definitely be required for all enhancements that have the potential for consuming large amounts of resources.

Reused code will show an additional benefit in the area of changes. That is, if a change is required in one module that change can possibly be instituted in all other instances of that module. The correction of an error would then only have to be detected and corrected once, rather than waiting for the remainder of similar modules to err and require correction.

Essential to the maintenance function is the accumulation of data on the various types of projects that are being maintained by the activity. These data need to be tabulated from the beginning of the project to its end and require completeness. That is the data need to include information on the functions and modules of the program. For instance,

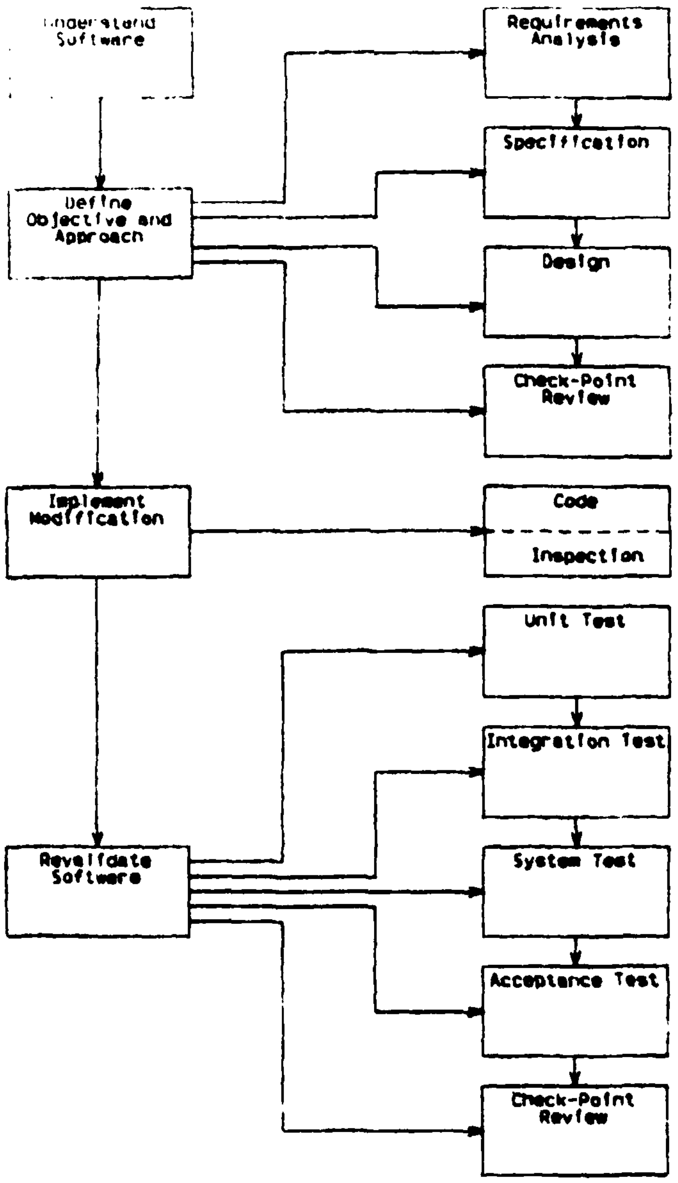


Figure 2.3. Structuring the maintenance process.

Source: McClure, Carma L., Managing Software Development and Maintenance, Van Nostrand Reinhold Company, Figure 6.2, p. 151.

they should include the error rate, what types of errors, when they were found, and especially how long it took to find and repair them. These data will enable management to gain better insight into the maintenance process and allow them to form better estimates on personnel requirements to conduct this function. Models can prove useful in this respect, but will prove even more useful when there are data available to determine their validity. The data accumulated will provide much information on the maintenance process and its usefulness will cross over into other projects because of a large degree of common properties in software.

H. SUMMARY

One of the ideas that should have become apparent from the preceding discussion is that maintenance, quality and reliability are intrinsically related. Designing for reliability and quality, while not necessarily increasing maintainability, will reduce the maintenance costs, if for no other reason than the elimination of errors. These three are more deeply related because the use of structured programming and modularity not only increases maintainability by decreasing complexity but also increases quality and reliablity for the same reason. The human programmer is able to comprehend only a limited amount of a program. These techniques allow him to understand what he is working on and in what context, reducing the probabilty of errors early in the life of the system. That is the key to all of maintenance, making the software as easily understandable as is possible, thus increasing the capability of the maintainer to find and repair or add the desired changes. It is possible to develop reliable software that is relatively complex without making it maintainable; it has probably been done more than once. It is easier to accomplish and certainly less expensive to conduct the software development process with the objectives of reliablity, quality, and maintainability when these tenets are adhered to than it would be if they are not considered.

The three potential methods to achieve the above objectives are dual code,reused code and bebugging. The latter will be explicitly treated in the following chapter. Dual code can increase reliability of code by providing a ready check on that code to ensure that it is error reduced. Reused code should reduce costs through reduction in testing of modules and coding required. It should additionally improve reliability through the use of previously tested and proven code. Bebugging will allow the manager to estimate the error rate of the code and its maintainability through some simple testing procedures. This method is important in that it provides a measure of the quality and maintainability of the program, thereby improving the planning for maintenance.

III. SUGGESTED MODEL FOR ESTIMATING PERSONNEL REQUIREMENTS DURING MAINTENANCE

A. INTRODUCTION

Prior to entering the maintenance phase, it is extremely important to have an estimate of the resources required to conduct it. An especially important part of these resources is the personnel requirements. The people who maintain the systems software will prove to be the largest single expense of the maintenance portion of the life-cycle. There have been a number of models developed to estimate the development costs of software and a few have attempted to extend their predictions to include the maintenance phase [Ref. 23], [Ref. 24], [Ref. 25], [Ref. 26]. The emphasis of these models is to utilize the functions, size, and applications of the software to estimate its life-cycle costs prior to the initiation of development and coding. This is essentially the macro approach to looking at the overall functions of the system and using them to estimate resource requirements.

While in many cases any model or estimating technique is better than no formalized technique at all, the construction of a model should be based on a thorough understanding of the components of the system. This could be considered the micro approach. The model presented here is one approach to understanding that portion of the life-cycle called maintenance. It attempts to explain the interactions of the components of software maintenance with a view toward predicting the resource requirements. Fundamental to the model is the fact that the development phase has been completed and that the system is in use.

Therefore, the size, functions, and applications of the program are well known and can be utilized in the estimation model. This approach should integrate well with MCTSSA's role as a maintenance facility as this is the point in time that they assume responsibility for the software system.

The presentation of the model covers the two types of maintenance that were defined in Chapter I, corrective maintenance and adaptive maintenance. The objective of the model is to determine the amount of effort required to conduct both types of maintenance.

The major assumptions made in this model are that the development of the system has been completed and it is currently in use. The maintenance facility is assuming responsibility for the software and, thus, knows its content. This allows for more accurate use of an estimation model based on an in-depth analysis of the code itself. Further assumptions are that the system has been developed in accordance with the guidelines presented earlier in this paper. While all guidelines may not have been followed in every instance, the model is presented such that the reader should be able to adjust its construction to suit his use.

B. DEVELOPMENT OF METRIC TO ESTIMATE CORRECTIVE MAINTENANCE WORKLOAD

1. Bebugging

"Bebugging" is a term coined by T. Gilb and is derived from a concept developed by H. Mills that introduces a number of known errors into a program to calibrate the error location process [Ref. 27]. The concept is that of introducing a known number of errors and then performing a debug exercise on the program. The objective is to compare the seeded number of errors detected to the errors that occurred naturally in the program and then use these figures to estimate the total number of bugs present in the program. Conducting this test over a specific period of time will afford a measure of the bug detection rate.

G. Schick and R. Wolverton in their article, "An Analysis of Competing Software Reliability Models" summarized the work of H. Mills and the later work of S. Basin and presented a formula that can be utilized in calculating an estimate of the maximum number of errors present in the software. That formula is:

$$N = \text{INT}\left[\frac{k(1) \times M - n(1) + 1}{r - k(2)}\right] \qquad \langle 1 \rangle$$

where,

N= maximum number of errors,

INT= integer value of evaluated expression,

r= number of statements in the test,

k(1)= number of statements in test in which indigenous errors were detected,

k(2)= number of statements in test in which seeded errors were detected,

n(1)= number of statements in which errors were introduced,

M= total number of machine executable statements in the system [Ref. 28].

This formula is based on a count of the statements with errors. The errors are seeded randomly in the entire program and in executable instructions only.

2. Implementation of Bebugging

The implementation of bebugging should be relatively simple and straightfoward. The seeding of the program needs to be done randomly. This can be done manually or automatically through the machine's use of a predetermined algorithm. The error type to be introduced should be considered at this point. The type of error introduced needs to represent the proper proportion of that error in relation to the total number of errors. [Ref. 29]. The type of bugs or errors considered in this test are semantic. The categories of semantic bugs are computational, logical, and data [Ref. 30]. The syntactic type of bug is not considered as it should be detected during complilation [Ref. 31] and design errors are generally considered too difficult to artificially introduce. The best method for obtaining the proportion of error types is to refer to the vendor supplied information on this project and to data that has been accumulated on other similar projects. The errors introduced should reflect this proportion in order to obtain a representative estimate. When the test is run each type of error should be calculated separately using the above formula. This is essential as each type of error will require a different degree of effort to repair.

Two methods are readily apparent for detecting errors after the program has been seeded. The first of these methods is manual detection by the programmer or programmers who will be involved in the maintenance of the system. It is important that those involved with the maintenance participate in the test to achieve calibration of the model to the programmer's capabilities and possibly eliminate any variance that could arise from differences in programmer skill. This will provide additional information on how long it takes for the programmer to detect and then correct the errors. The test should consist of timing how long it takes for the programmer to locate an error by type. This could be accomplished by maintaining a time-sequenced listing of when each error was found by type. The detection rate could then be established for each error type by obtaining the average number of errors detected per a specific time, in this case a man-hour. This rate is valid if the system is to be constantly reviewed for errors by these individuals. The alternative to this method is to develop a model which is able to predict an error rate valid for the operating cycle. This model would obtain a value that would show the rate at which errors appeared during operation of the system and required repair.

The second method of obtaining the error count is through the use of the dual code technique. Dual code provides two parallel implementations of the design specifications, in either the same or different languages and has been discussed earlier in section E of Chapter II. In the bebugging context the seeded or artificial errors are introduced into one of the code sets. The two code sets are run in parallel and their results are compared during running for discrepancies [Ref. 32]. The differences in the results will yield, since the two sets of code were coded independently, indicators as to where errors lie in the program. One set of code will, in effect, check the other through this process. The code set with the introduced errors will be used to obtain the error estimate. This method will yield only an estimation of the total number of errors and an error rate per lines of code. It will not allow one to determine the maintainabilty of the code through the use of programmers.

Since the dual code method will yield only an estimated number of errors, the two methods of manual and dual code should be used together. The reason is to obtain a check on the number of projected errors, and because only the manual method can provide an indicator as to the time required to detect and correct errors by a specific programmer. If only one method can be used due to resource constraints, the manual method is preferred as it provides three types of information, that of error rate, detectability of the errors, and the maintainability of the code.

Bebugging was selected as a possible method for the maintenance facility to evaluate the software for planning purposes for a number of reasons. It is conducted independently of the vendor and allows verification of his data and the techniques employed. While discrepancies between estimates are sure to arise, large discrepancies should be suspect and should subject either the facility's or vendor's methods to re-evaluation. The bebugging test is conducted under the conditions and with the people that will be prevalent during the maintenance phase. The test should be relatively simple to structure and implement by the facility. Additionally, the concepts involved are easy to understand by those participating.

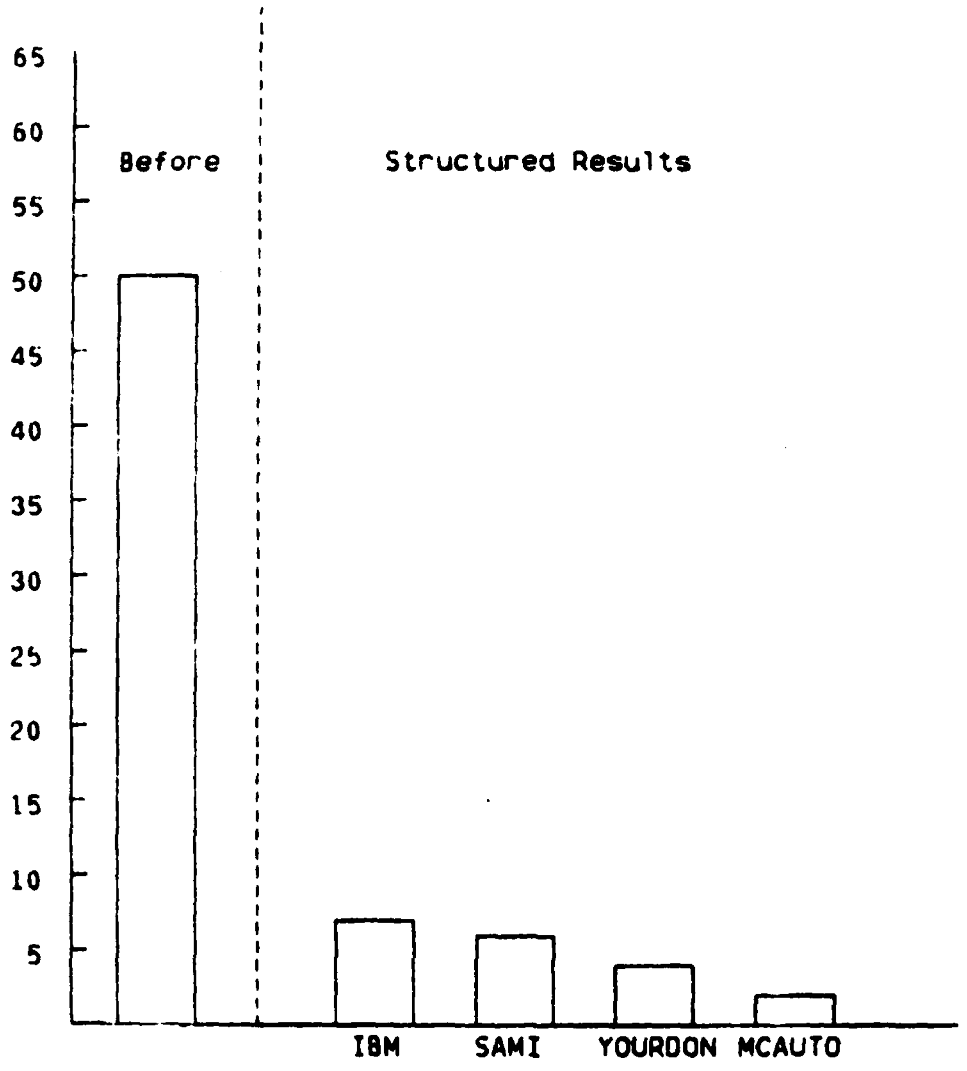
There are some distinct disadvantages to bebugging that should be discussed. The first is that it fails to identify the degree of error. The degree of an error can fall into one of five categories: 1) error which prevents the accomplishment of an essential function, 2) error which adversely affects the accomplishment of an essential function degrading performance, 3) error which adversely affects the accomplishment of an essential function degrading performance, but has a work-around solution, 4) error which is merely an operator inconvenience, and 5) all other errors [Ref. 33]. As can be seen from the above definitions, the degree of the error is the extent to which the system's functioning is affected and not the cause of the error or error type. The degree of error and design errors do not lend themselves to detection though the bebugging technique due to their complexity. Alternative methods need to be developed in this area.

An additional trouble area became apparent during research and that is the problems that occur when repairing a bug. The possibility always exists for the repair of the bug to introduce additional errors through unpredictable effects on other modules. The best insurance to insulate against these effects is the preservation of modularity and the use of information-hiding modules which do not allow the programmer to make any assumptions that could later prove dangerous to the program. Additionally, the use of structured programming and the techniques described in Chapter II, section C should work to reduce the design errors that may develop later. This is exemplified by Figure 3.1.

The use of bebugging will produce an estimate of the error detection rate that can be used for planning purposes, if it is recognized that this is just an estimate and not what will occur. The bebugging method can be used periodically to evaluate the current status of the software at various points along the maintenance path. The estimates derived therefrom can be used to refine or revise planning figures. Further, greater confidence in an estimate can be achieved through more testing, although at additional cost.

3. Estimation of Corrective Maintenance Workload

The corrective maintenance workload can be predicted using the number of estimated errors in the system and the rate of error detection established by the programmers during the bebugging test using equation <1>. The error detection rate as well as the number of errors should be divided into the three types of semantic bugs identified earlier. The resultant formula should estimate the amount of corrective maintenance that will be required on the system.



Errors Per 10,000 Lines of Code.

Figure 3.1. Software maintenance improvements with structured programming.

Source: McClure, Carma L., *Managing Software Development and Maintenance*, Van Nostrand Reinhold Company, Figure 3.10, p. 54.

The formula is:

$$CM = \frac{N(c)}{d(c)} + \frac{N(l)}{d(l)} + \frac{N(d)}{d(d)} \qquad <2>$$

where,

CM= total corrective maintenance required in man-hours,

N(i) = number of i type error estimated (c=computational, l=logical, d=data) using equation <1>, and

d(i) = the detection rate in errors per man-hour of i type error.

C. DEVELOPMENT OF METRIC TO ESTIMATE ADAPTIVE MAINTENANCE LOAD

Adaptive maintance, as previously defined, is that maintenance conducted to improve the system by increasing its capabilites or change its operation to a form that the user desires after the system is operational. These changes or enchancements are accomplished to improve the overall efficiency of the system, add new features, or provide interfaces with other systems that were not called for in the original design. The enhancements, in many cases, should not substantially alter the original design of the program. If a major redesign is warranted, the system should be returned to the development phase to ensure that the design is done properly. The adaptive maintenance discussed here will cover those cases where modules may be changed or added, but the structure of the original program essentially remains intact.

The addition of enhancements should be conducted along the lines of the original development process to ensure that, while the system is enhanced, the changes are integrated into the system with a minimal degrading effect on performance. An understanding of the system to be enhanced is required. This understanding is governed by the logical and structural complexities of the software. Adaptive maintenance involves two major types of enhancements. A portion of them involves alteration and a small addition of code and a portion requires the addition of a new module, replacement of an older module or a restructuring of the software's structure. The degree to which each portion presents itself during the life of a system is as yet indeterminate and will require in-depth study. Experience can provide some indication as to how often and to what extent these two degrees of enhancement are made.

1. Use of Halstead's Effort Metric as a Measure of the Program Complexity

In order that a modification may be made, the individual making the change needs to understand the system. The amount of time he takes before he can begin useful work on the system is governed by the complexity of that system. The degree of software complexity is inversely related to the level of understanding. The more complex the software, the less well understood it will be until more effort is expended in an effort to improve comprehesion.

Halstead's development of programming effort essentially realized this. Halstead's effort metric was developed to analyze the effort required to construct a program in a specific language from a preconceived algorithm. Its application in maintenance for using it to rate the complexity of the software should prove valid. To develop programming effort Halstead used the concepts of program level and volume [Ref. 34].

Program level refers to the level of a program's implementation. There is a minimum level of implementation where the fewest number of operands and operators possible can be used and the program will still function as intended. This most elegant of implementations is never realized in fact and some lower program level is achieved. The easiest language to use would have a program level of one, where any procedure desired would consist of merely a call on that procedure. This would require an infinite list of procedures and is not realizable. Implementations of programs will fall into an area of program level less than one. Use of this greater number of statements and the consequent explanation results in greater understanding of the implementation for the person less familiar with the system The difficulty of the comprehension of a program varies inversely with the level of that program [Ref. 35].

As presented by Halstead, the program level is affected by the operators present. The larger the number of operators employed, the lower the level of implementation. The minimum number of operators possible is two, where one would consist of a function designator and the other an assignment operator. The program level is therefore proportional to the minimum number of operators possible divided by the actual number of unique operators [Ref. 36].

Operands do not show a similar minimum over all implementions. In cases where an operand name is represented, the implementation is at a lower level than was possible if the operand was used only once. The program level is then proportional to the ratio of the number of unique operands to the total operand usage [Ref. 37]. Combining the two proportionalities and noting that the constant of proportionality is one, as this is the maximum defined value of program level, yields the program level, L, as

$$L = \frac{2}{n(1)} \times \frac{n(2)}{N(2)} \qquad \langle 3 \rangle$$

where,

n(1) = the number of unique operators,

n(2) = the number of unique operands,

N(2) = the total number of operands present and two is considered the minimum number of operators possible [Ref. 38].

Program level represents a meaure of how well the software has been implemented in relation to the capabilities of the language that has been used. The better the implementation the closer to one the value of L becomes.

Program volume recognizes the importance of obtaining a metric for the size of a algorithm that not only measures its physical length but also the number of distinct operations performed in the program. The objective is to allow application to a wide variety of languages. Volume V has been defined as:

$$V = \frac{N \ln(N)}{\ln 2} \qquad \langle 4 \rangle$$

where,

N is equal to length or N1+N2, the total number of operators and operands utilized, and

n is the vocabulary of unique operators and operands or n(1)+n(2) [Ref. 39].

This volume can be applied to any programming language and measures the size of the program in the language coded. It takes into account the capabilites of the language as presented by the number of unique operators and operands and its size as represented by the total number of operators and operands. Program volume represents an overall measure of the size of a program in relation to that program's comprehensibility.

To obtain the effort metric, E, Halstead uses the ratio of program volume to program level [Ref. 40].

$$E = V/L \qquad <5>$$

From this equation it can be seen that as the program volume increases, the effort or complexity will increase and that as the program level increases the effort decreases in kind. Executing the necessary substitutions to allow for calculation of the complexity of the program the equation becomes:

$$E = n(1) \times N(2) \times \left[\frac{(N(1) + N(2)) \times \ln(n(1) + n(2))}{\ln 2} \right] \qquad <6>$$

This formula, when used to determine the complexity of a program in relation to a programmer's debugging performance, accounted for over twice as much variance in performance as a metric that counted solely the total number of program statements [Ref. 41]. The resultant value, when applied to programs on board, will provide an estimate of the complexity of a program, thereby refining the estimate of the quantity of resources required to make alterations to the software.

Implicity treated in the above formula is the way in which modularity affects the complexity of the program. The use of modules such as functions, subroutines and macros will reduce the program volume through their inclusion. They are performed multiple times during execution of the program, but will be present only a single time when the software is reviewed or checked. Through their single inclusion, they reduce the total number of operators and operands present, directly reducing the program volume and increasing its comprehensibility. Additionally, as Halstead indicated, the number of unique operators will increase with the addition of subroutine or function calls, again reducing the overall complexity of the system. Interesting to note at this point is that Halstead, through further development, has stated that E will vary with the square of the volume and not linearly in relation to the program's potential minimum volume (the best implementation possible) [Ref. 42]. This also demonstrates that as modules are added the system's complexity is reduced, not linearly, but as some function of the square.

2. Estimation of Adaptive Maintenance Workload

The estimates of the amount of personnel effort needed will require the combination of the above complexity metric and the benefit of previous experience on similiar projects. The metric can, to a large extent, predict the amount of time required to understand a program, a factor that is critical to the proper conduct of adaptive maintenance. This effort should be required each time the system is to be enhanced. The shortcoming of the model is the requirement for a prediction of the frequency of enhancements and their degree. The degree of alteration is a definite consideration as it will govern the time and effort required to accomplish the changes. Major alterations will take greater time and effort than will minor ones, but a large number of minor changes can easily outweigh one major change in effort.

The only present method used to estimate the frequency of alterations required is experience. The degree of enhancement should be divided, at least initially, into major and minor. A major enhancement would consist of at least the replacement of an old module of the system or the addition of a new one. A minor enhancement would consist of an alteration to a module in which either a line of code is rewritten or replaced, or the module itself is rewritten with its function remaining as it was prior to the modification. Until further data is accumulated on the type of enhancements conducted, this initial distinction should be used to improve the estimation process.

Two methods are suggested for the use of the complexity metric. The first would consist of simply multiplying the average time of all enhancements by the ratio of the complexities for the new system to the average complexities of all the previous systems. The preferred method, though, is to use the average time to conduct the enhancement and the average complexity thereof broken out by the enhancement degree. Each resulting average by enhancement degree should then be multiplied by the ratio of the new system's complexity to the average of the previous system's complexity and the frequency of enhancements per project. This formula is:

$$AM = E \times \left[\frac{x(\text{maj}) \times N(\text{maj})}{E(\text{aver. for maj})} + \frac{x(\text{min}) \times N(\text{min})}{E(\text{aver. for min})} \right] \quad <7>$$

where,

AM= total adaptive maintenance required in man-hours,

x(maj) = the average time to add a major enhancement,

x(min) = the average time to complete a minor enhancement,

E(aver for maj) = the average complexity of major enhancements using equation <6>,

E(aver for min) = the average complexity of minor enhancements using equation <6>,

N(maj) = the average number of major enhancements,

N(min) = the average number of minor enhancements, and

E = the complexity of the program using equation <6>.

N(maj) and N(min) can be used in the equation in two ways. The first, as presented, is as the frequency that enhancements of the two degrees have occurred in the past. The other way could be to use an estimated number of enhancements, if management has some idea of special circumstances in which these numbers will vary from past events.

D. MODEL AGGREGATION

The entire maintenance effort required for the project from time the system is accepted at the maintenance facility can be calculated by adding the estimated corrective maintenance workload to the adaptive maintenance workload. This yields:

$$TM = AM + CM \qquad \langle 8\rangle$$

where,

TM= total combined maintenance required in man-hours.

This result should yield an estimate of the total maintenance effort required and needs to be subdivided into years to be more useful to management. One method of accomplishing this is to divide the total maintenance effort by the estimated number of years remaining in the project. This will yield a straight line average of maintenance that fails to show any variations that normally present themselves later. Its advantage is that it is extremely simple. Another method is to reevaluate the project yearly using the above formulas and actual experience. This may prove infeasible as the estimation needs to be conducted well in advance of that year for budgeting purposes.

The last method for developing annual personnel requirements is to return to the components of maintenance. Worse case corrective maintenance can be estimated as remaining at least constant if not decreasing throughout the remaining life of the software. The error rate will prove highly dependent on the enhancement rate. It seems reasonable to assume that, if a large number of enhancements are made, the system's error rate will increase correspondingly. Thus, the normal assumption that the error rate decreases as the project continues will not prove valid, if a sizeable number of enhancements are made. Additionally, if no enhancements are made the error detection rate will never disappear entirely and will remain much higher than expected. This variation should be insulated against by utilizing an annual detection rate where the initial number of estimated bugs is divided by the estimated annual detection rate. The estimated annual detection rate can be estimated from the bug detection rate established during the bebugging test. Periodic retesting of the system should be conducted especially when a major enhancement has been added to revalidate the error detection rate.

The adaptive maintenance phase, at over seventy percent of maintenance costs, accounts for the largest portion of the software maintenance budget in government activites [Ref. 43]. Therefore, it is this area that demands the greatest efforts to account for cyclic activity. Again, no predictive ability for the number of enhancements by type exists in this model and the only method is to use data obtained from previous projects to determine the enhancement rate at different stages in the maintenance phase. These estimates, broken out by year, could then be added to the anticipated corrective maintenance loading to obtain the annual figures to be used for planning. This model does develop a prediction of the quantity of resources required to implement each enhancement by type.

E. CHAPTER SUMMARY

The model has been developed considering the two elements of maintenance as defined here, corrective and adaptive maintenance. It has further been shown to yield an estimation of the total manpower requirements for the project from the time of assumption of maintenance responsibilities. These figures have been manipulated to provide annual estimates of manpower requirements. Halstead's effort metric and Gilb's bebugging provide the basis from which the model was developed.

The largest requirement for this model, or any model for that matter, is to obtain data with which the results of the model can be calibrated and tested. The requirement exists for the establishment of a data base on personnel expenditures during the maintenance cycle. Without this data any model developed cannot be tested fully. Additionally, the models developed will not be calibrated properly to allow for their fullest utility.

The model developed here has been designed to maintain, to some extent, simplicity in order to allow it to be employed in a working environment. It has been outlined so that the reasoning should be evident, allowing managerial personnel the capability to adjust it to fit their situation. The model is based on those concepts that have apparently received favorable reviews and have been employed in other similiar situations.

IV. CONCLUSIONS

As this research has shown, it is important to establish early in the life of a software project the desire to reduce maintenance costs. With this commitment, the development phase may take longer and cost more, but the long-term results will be worth the extra effort. Maintenance costs will continue to consume the largest portion of the resources allocated to software systems and only through the conscientious application of the tenets outlined in Chapter II can this portion be expected to be reduced.

Most important of these tenets is the use of structured design and structured programming to aid in the reduction and identification of potential errors early in the life of the project. This is the time when they are least expensive to repair. Reused code will provide benefits throughout the entire life of the project by reducing devlopment and maintenance costs by providing previously coded and tested modules for inclusion in the software being developed. The use of a high level language will increase the maintainabilty of the program by making it more understandable.

The estimation of the personnel requirements will aid management during the budgeting process. The number of unplanned occurrences, such as exceeding budget limitations or unexpected levels of maintenance, will decrease because of greater comprehension of the maintenance phase and its components.

This model is a first attempt at developing a system that will estimate the personnel requirements for maintenance. It has been presented in such a manner as to increase understanding of the items that affect maintenance in both favorable and unfavorable ways. The reader is encouraged to utilize the model and adapt it to his own use by applying it to his own situation and requirements.

An important item to note is that a data base has to be established that catalogs those items concerning the software that are important to the estimation and understanding of the software maintenance process. This information should include at least error detection rates, correction times, number of enhancements made as well as the estimated rates for each of these items.

LIST OF REFERENCES

1. Reyes Associates Inc., "Study of the Marine Corps Tactical Systems Support Activites", p. 1-2, July 1978.

2. Naval Underwater Systems Center Techincal Report #5783, A Comparison of Tactical Military and Commercial Data Processing Computer Architectural Requirements, by DeNoia, Lynn A. and Gordon, Robert L., p. 15, May 1978.

3. Lientz, B. P. and Swanson, E. B., Software Maintenance Management , p. 105, Addison-Wesley, 1980.

4. Schneidewind, Norman F., Analysis of Error Processes in Software, Naval Postgraduate School, p. 6-7, 1974.

5. McClure, Carma L., Managing Software Development and Maintenance, Van Nostrand Reinhold Company, p. 33, 1981.

6. Ibid, p. 41.

7. Ibid, p. 42.

8. Lientz, B. P. and Swanson, E. B., Software Maintenance Management, p. 68, Addison-Wesley, 1980.

9. McClure, Carma L., Managing Software Development and Maintenance, Van Nostrand Reinhold Company, p. 43, 1981.

10. Ibid, p. 44.

11. Ibid, p. 45.

12. Military Standard MIL-STD-1679(NAVY), Weapon System Software Development, p. 8, December 1978.

13. Lientz, B. P. and Swanson, E. B., Software Maintenance Management, p. 7, Addison-Wesley, 1980.

14. Lanergan, Robert G. and Dugan, Denis K., "Software Engineering with Reusable Designs and Codes," IEEE 1981 COMPCON Fall , p. 296, September 1981.

15. Ibid, p. 303.

16. Gilb, Tom, Software Metrics, p. 58, Winthrop Publishers Inc., 1977.

17. Ibid, p. 85.

18. Ibid, p. 87.

19. Lientz, B. P. and Swanson, E. B., Software Maintenance Management, p. 30 and 154, Addison-Wesley, 1980.

20. Ibid, p. 69.

21. Ibid, p. 154.

22. Ibid, p. 68.

23. TRW Defense and Space Systems Group, "Software Cost Analysis and Estimating," Airborne Systems Software Acquisition Engineering Guidebook, September 1980.

24. Rome Air Development Center RATC-TR-81-144, "An Evaluation of Software Cost Estimating Models", by Robert Thibodeau, June 1981.

25. Shooman, Martin L., "Tutorial on Software Cost Models", Workshop on Quantitative Software Models, October 1979.

26. Air Force Wright Aeronautical Laboratories AFWAL-TR-80-1056 Vol. II, "Predictive Software Cost Model Study", by Hughes Aircraft Company Support Systems, June 1980.

27. Gilb, Tom, Software Metrics, p. 28, Winthrop Publishers Inc., 1977.

28. Schick, George J. and Wolverton, Ray W., "An Analysis of Competing Software Reliability Models", IEEE Transactions on Software Engineering, Vol. SE-4, No. 2, p. 110, March 1978.

29. Gilb, Tom, Software Metrics, p. 37, Winthrop Publishers Inc., 1977.

30. Curtis, Bill, Sheppard, Sylvia B., and Milliman, Phil, "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metric", Software Engineering, p. 358, September 1979.

31. Ibid, p. 358.

32. Gilb, Tom, Software Metrics, pp. 46-48, Winthrop Publishers Inc., 1977.

33. Military Standard MIL-STD-1679(NAVY), Weapon System Software Development, p. 19, December 1978.

34. Halstead, Maurice H., Elements of Software Science, p. 47, Elsevier North Holland, 1977.

35. Ibid, p. 26.

36. Ibid, p. 27.

37. Ibid, p. 27.

38. Ibid, p. 27.

39. Ibid, p. 19.

40. Ibid, p. 47.

41. Curtis, Bill, Sheppard, Sylvia B., and Milliman, Phil, "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metric", Software Engineering, p. 358, September 1979.

42. Halstead, Maurice H., Elements of Software Science, p. 47, Elsevier North Holland, 1977.

43. General Accounting Office, AFMD-81-25, "Federal Agencies' Maintenance of Computer Programs: Expensive and Undermanaged", p. 42.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Defense Logistics Studies Information Exchange U. S. Army Logistics Management Center Fort Lee, Virginia 23801	2
3.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
4.	Department Chairman, Code 54 Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	2
5.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
6.	Dr. Dan C. Boger, Code 54BK Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	5
7.	LCDR Ronald Modes, Code 52MF Department of Computer Science Naval Postgraduate School Monterey, California 93940	3
9.	LtCol. A. E. Burns, III, USMC Marine Corps Tactical Systems Support Activity Camp Pendleton, California 92055	2
10.	Commandant of the Marine Corps (Code MPP-21) (ATTN: Captain W. H. Merring, III, USMC) Headquarters, United States Marine Corps Washington, D. C. 20381	4

DATE

ILME

—8