Software Quality Assurance

Summer 1979

Edward C. Soistman

University of Central Florida, edsoistman@yahoo.com

Find similar works at: https://stars.library.ucf.edu/rtd

University of Central Florida Libraries http://library.ucf.edu

Part of the Industrial Engineering Commons

STARS Citation

https://stars.library.ucf.edu/rtd/447

This Masters Thesis (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of STARS. For more information, please contact lee.dotson@ucf.edu.
SOFTWARE QUALITY ASSURANCE

BY

EDWARD C. SOISTMAN
M.S., University of Central Florida, 1979

RESEARCH REPORT

Submitted in partial fulfillment of the requirements for the degree of Master of Science:
Industrial Engineering Management Systems
in the Graduate Studies Program of the College of Engineering
at the University of Central Florida; Orlando, Florida

Summer Quarter
1979
ABSTRACT

The problems associated with software development and use are investigated from a management point of view. Having identified the critical aspects of effective software management, an approach is suggested for the creation and implementation of a software quality assurance program. Particular attention is focused on the concept of Life Cycle Procurement as currently utilized by the Department of Defense.

The research was accomplished in two phases. The first consisted of an extensive literature search, seminar attendance and participation in several working groups assigned the responsibility for establishing software quality assurance guidelines. The second phase involved direct participation in the development of a formal software quality assurance program.

The report is written in a manner designed to guide a non-technically oriented manager through a complete analysis of software, its measures of quality, its problem sources and the most promising techniques which can be used to control and evaluate its development.
ACKNOWLEDGEMENT

I wish to thank my loving wife, Marianne and my wonderful children, Eddie, Laurie, Steven and Jeffrey for their understanding and patience throughout the duration of my research. Without their support the project could not have been accomplished.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS ........................................ iii

LIST OF ILLUSTRATIONS .................................. vi

Chapter

I. INTRODUCTION ............................................. 1

   Historical Observations .................................. 1
   The Need for Software Quality Assurance .................. 2

II. STATEMENT OF THE PROBLEM .............................. 6

   Purpose and Objectives .................................. 6
   Approach .................................................. 6
   Results Achieved ........................................ 7
   Literature Search ........................................ 7
   Seminars and Working Groups ................................ 8
   Development of a Software Quality Assurance Program .... 12
   Research Report ......................................... 13

III. SOFTWARE ............................................... 14

   Definitions ............................................ 14
   Software Quality ........................................ 16
   Error Types ............................................. 20
   Documentation ........................................... 23
   Development Process ..................................... 26
   Life Cycle Concepts ..................................... 27
   Need for Early Quality Assurance ........................ 29
   Need for Requirements Traceability ....................... 34

IV. SOFTWARE QUALITY ASSURANCE .......................... 41

   Historical Approach ..................................... 41
   Assurance Versus Control ................................ 42
   Analysis of the Development Process ....................... 49
## LIST OF ILLUSTRATIONS

1. Hardware/Software Cost Trends ........................................ 3
2. Characteristics of Software Quality .................................... 21
3. Software Life Cycle and Major Decision Points ....................... 30
4. Stages of the Software Development Process .......................... 31
5. Error Propagation During Software Development ................... 33
6. Critical Continuity Loop .................................................. 35
7. Classical Approach to Software Quality Control .................... 43
8. Quality Control versus Quality Assurance .............................. 45
9. Mil-S-52779 .................................................................. 48
10. Decomposition of the Software Development Cycle ................. 52
11. Activities Associated with a Software Stage ........................... 55
12. Total Quality Assurance of a Software Stage .......................... 56
CHAPTER I

INTRODUCTION

Historical Observations

On June 14, 1951 the UNIVAC I computer, built by J. P. Eckert and J. Mauchly was delivered to the United States Census Bureau (1). It was immediately heralded as a "brain," a wonder of the world and a panacea to cure all ills. The scientist saw it as a wondrous calculator capable of performing iterative mathematical functions with speed and accuracy never before imagined. The businessmen saw it as a super accountant capable of collecting, sorting, processing and disseminating thousands of facts. Dreamers saw it as a vehicle to a Utopian world where humans did little more than enjoy the fruits of the world while computer-controlled machines performed all unpleasurable tasks. Skeptics viewed it as the end of human domination of the world and felt that these unholy machines would self-propagate until they could outwit and eventually control mankind.

The first two prophecies have essentially been realized. The use of the computer has revolutionized both scientific and business data processing. In addition, technologies such as space exploration have become realities only because of the speed and precision available in today's computers. Fortunately, neither Utopia nor human subservience has occurred. The computer has indeed freed mankind of
many tedious and menial tasks and has opened the door to more creative and stimulating pursuits. The predictors of doom, however, may have the last laugh, but not in the sense of machine dominance. The world has learned not only to live with rapid technological growth but also to expect and demand it. This demand has generated an ever increasing demand for new computers, which has, in turn, spawned a multi-billion dollar industry, one on which we are becoming more and more dependent.

Computer programs, needed to cause computers to perform specific tasks, are an integral part of all computer systems. Although this fact has never been disputed, it has only recently been recognized that they represent a majority of the total costs associated with computer equipment. Figure 1 was taken directly from a special issue of the IEEE Transactions on Computers (2). As shown in the figure, software costs have already exceeded the price of the associated hardware. The trend is expected to continue until software eventually makes up 90% of the total costs associated with computer systems. In 1976, twenty-five short years after its introduction, the total annual costs of software in the United States exceeded twenty billion dollars. In its relatively short life, software has become a major and expensive commodity. Industry must establish techniques to manage and control its development or risk cancellation of the financial benefits of computer technology.

The Need for Software Quality Assurance

Cost alone is not necessarily bad. A premium price is gladly paid in many cases to obtain a premium product or service and computer
technology, despite its ever rising cost, has proven its value to society many times over. Unfortunately, while computer hardware costs have decreased due to state of the art advances in electronics, miniaturization and production technologies, software costs have skyrocketed despite equally impressive advances in programming capabilities. Furthermore, buyers of software products are frequently disappointed in the functional capabilities of the delivered products. Very large cost and schedule overruns are not uncommon (3).

Dr. Kenneth Iverson, the developer of APL, while at a conference in Pisa, Italy, reflected the feelings of many software customers by comparing a typical software project to the famous leaning tower:

It took 300 years to build and by the time it was 10% built, everyone knew it would be a total disaster. But by then the investment was so big that they felt compelled to go on. Since its completion, it has cost a fortune to maintain and is still in danger of collapsing. There are at present no plans to replace it since it was never really needed in the first place. (4)

In the Software Acquisition/Management Course, Software Enterprises Corporation presented a summary of typical software problems for which there are equally valid counterpoints (5):

- Exceed Cost Estimates
  --The request for proposal was vague
  --New requirements were added during the development

- Delivered late
  --Requirements were continually changing
  --Interfacing contractors caused delays

- Does not meet requirements
  --The requirements were vague
  --It passed the required tests
o Is unreliable
   --Exhaustive test is impossible
   --More money, more tests
o Is inadequately documented
   --There were no standards
   --You got what you paid for
o Does not interface
   --Our product is correct, the hardware is wrong
   --The requirements were not clear

Clearly, the prevalent problems associated with software development are associated with the management and control of the process rather than the technology of the science or the skills of the programmers. Unfortunately, it is only recently that this conclusion has been reached. While it is true that programming techniques and individual skills play a significant role in the production of high quality software, the failure to properly manage software development is responsible for the majority of serious shortcomings witnessed in the past. Quality Assurance and Control is equally, if not more, important to software development as it is to hardware production.
CHAPTER II

STATEMENT OF THE PROBLEM

Purpose and Objectives

The purpose of this study was twofold: first to research the nature of computer software, its measures of quality, its development process, its problems and potential solutions and to formulate an approach to the establishment of a Software Quality Assurance Program which can be generally applied to software development activities and second, to report the results of the research in a format which can be easily understood by managers unfamiliar with "jargon" and technical aspects of computer programming. An implied objective is to influence and encourage industrial implementation of software standards, management techniques and quality assurance activities.

Approach

The study was divided into two distinct phases. The first phase consisted of an extensive search of the literature and participation in numerous seminars and working groups for the purpose of gaining a complete understanding of the nature of software problems and the approaches taken by other individuals and organizations to solve or avoid them. The second phase consisted of direct participation in the development of a formal software quality assurance
program at Martin Marietta Corporation and the preparation of this report.

Results Achieved

All objectives were met or exceeded. Throughout the period of research it became increasingly obvious that there is an intense national (and international) interest in the problems associated with software development and quality assurance. Virtually every major software developer is seeking an effective solution to both problems. The following is a detailed summary of the specific results achieved.

Literature Search

It was anticipated that a wealth of literature would be available due to the interest in computer software and its ever increasing cost in relation to its associated hardware. Initial research confirmed an even greater interest than was expected. Literally thousands of articles have been written concerning software development and applications; however, most identify problems but very few offer constructive solutions. As a result of initial reviews, approximately one hundred articles were deemed to be pertinent to the study and were studied in depth. During the course of the research, it was found that the Department of Defense was particularly interested in the analysis of software development and had issued many directives, policies and guidebooks concerning software management and control. Many of these documents were reviewed and considered during development of this report.
In general, it was found that the trade journals were the best source of information. The proceedings of various societies such as ACM and IFIPS are generally research oriented and address a level of technology which will not be of use for some time. Additionally, it is the opinion of this author that too much emphasis is being placed on the mathematical "purity" of computer programming and not enough on the development of techniques for managing and controlling it. The problems associated with software development and implementation involve considerably more than simple programming techniques. The list of most common software complaints presented in Chapter I is an indication that the lack of effective communications and control are the prevalent sources of problems. Numerous studies show that the actual programming effort accounts for only 20% of the overall cost of software development (6).

The trade journals, unlike the professional societies, tend to address actual problems facing industry today. Unfortunately, the large preponderance of such articles address very specific solutions to very specific problems. Very few individuals and organizations have properly addressed the true nature and cause of software problems: the extremely rapid growth of computer technology has outpaced the development of management techniques to control it.

Seminars and Working Groups

The International Federation of Information Processors (IFIP) Congress 77 was held in Toronto, Ontario in August 1977. It was attended by nearly a thousand delegates from thirty countries.
Although the conferences held were oriented primarily toward theoretical and technical topics, there were strong indications of international recognition and concern over the apparent inability of the industry to define standards and techniques for managing software development. During one panel discussion, Dr. Edsger W. Dijkstra of the Netherlands and best known as the developer of structured programming techniques quipped: "I've never delivered a computer program with known errors" when asked about the need for software quality assurance (7). His remark was geared to emphasize two points.

First, software testing or verification is essential to find errors regardless of the expertise of the programmer. Secondly, that test or verification must be performed by someone other than the programmer himself. Obviously, no programmer would deliver a computer program with known errors! Since errors do occur, it is logical to assume that the programmer cannot thoroughly evaluate his own work. He will make the same logical errors when checking as he did during development. Other well known members of the panel presented similar views in their response to questions from the floor. One member, when questioned about software standards replied: "We can't even decide whether to slash an '0' or a zero." Obviously the lack of meaningful standards and testing methodologies are universally recognized.

Two related seminars given by Software Enterprises Corporation were also attended during the course of this research. The first was a four day course entitled "Software Acquisition/Development Management." It was an intensive review of the events which make up the "software life cycle." A typical software procurement was analyzed
from its inception to its eventual implementation by the procuring agency. At each stage of the development, problem sources were discussed and method or prevention were suggested. The course was directed primarily toward NASA and Department of Defense procurements but was attended by both military and commercial software producers. An exceptionally thorough presentation was made with regard to current government regulation, policies and specifications. The second seminar was a three-day course entitled "Software Quality Assurance." It too, was an intensive review of techniques which can be applied to various aspects of software development to control it and measure its quality against predetermined standards. Both courses proved to be invaluable to the author during later participation in various working groups. Both are highly recommended for anyone assigned with the responsibility for establishing software management or quality assurance programs within their organizations (3, 5).

While attending a forum presented by the Defense Contract Administration Service (DCAS), it was learned that a special team (Project Team Bravo) had been established and charged with defining the DCAS responsibilities with regard to administering government contracts for computer software. Despite the ever increasing recognition of the need for software quality assurance, it is still unclear as to whether a formal program should be contractually directed or if it should be self imposed as a matter of good business. In either case, the role to be assumed by DCAS is still unclear as to how to evaluate and monitor the effectiveness of the quality assurance program itself. Many companies have skirted the issue by simply
assuming that software and hardware can be controlled by the same quality organization and have ignored the special characteristics of software development. DCAS feels that specific programs and specific procedures are required for software quality assurance and feels that they have an obligation to review and approve them prior to contract award. Although the committee is still active and has not yet completed its charter, it is clear that prudent government contractors should be establishing internal programs both for their own internal benefit as well as for future contractual requirements (8).

Dealings with DCAS through both Project Team Bravo activities as well as through informal discussions suggested the need for the formation of a nationally based technical working group comprised of government, civilian and industry personnel to attempt to bring together and standardize all the diverse approaches to establishment of software quality assurance programs. The American Society for Quality Control (ASQC) was contacted and responded favorably to the idea. Working through the training branch, a working group has been created to define and present a two-day forum on the subject of software management and quality control. The committee includes representatives of several major corporations, the Defense Department and the Civil Aeronautics Board of Canada. At the initial planning meeting held June 1979, a charter was established and individual assignments were made. In the next two meetings of the group it is understood that the forum will be developed for presentation in 1980 to six different regions of the country. It is expected that a considerable interchange of ideas will be fostered by the activities
of the group. This author has been chosen as one of the researchers and presenters (9).

**Development of a Software Quality Assurance Program**

Martin Marietta Corporation is a major contractor for the Department of Defense. Although primarily a developer of large scale missile systems, Martin has become a major supplier of computer software both as a stand alone contract item and as a subsystem to entire missile systems. Like the rest of the industry, it was recognized that software costs are becoming increasingly significant even in procurements where software appears to represent a relatively small portion of the effort. In recognition of a lack of effective software quality control, a corporate working group was established in 1978 to define and formalize the policies, standards and procedures needed to insure that high quality software can be consistently produced at an acceptable cost. The committee included representatives of various Martin divisions and functional departments. They were selectively chosen for their software experience and technical expertise and were given the necessary authority to accomplish the necessary studies and evaluations of current and past software projects. The efforts of the group culminated in July, 1979, with the issuance of a corporate policy, divisional policies (defining functional responsibilities), software standards, quality procedures, departmental directives and organizational authorizations. Concurrently, two training programs were released: one for top management and one for software designers and programmers (10).
Research Report

The remainder of the report is divided into four major sections which are written in a manner designed to "lead" a non-technical reader from a basic introduction to software concepts through the implementation of a software quality assurance program. To accomplish this objective without unnecessary redundancy, the sections rely on definitions and conclusions made in previous sections. The following is a brief description of the general organization and intentions of the remainder of the report:

Chapter III. SOFTWARE--In this chapter, the characteristics of software are discussed beginning with definitions of various software terms and concepts and progressing through its development process. The subject of software quality is discussed in detail since it was discovered by the author that determining what to monitor proved to be one of the major frustrations experienced by project managers.

Chapter IV. SOFTWARE QUALITY ASSURANCE--In this chapter, essential activities are identified and described as they relate to the development process. A philosophical discussion of the control function is presented and followed by a more "down to earth" description of the actual activities of a software quality assurance function.

Chapter V. IMPLEMENTATION OF AN SQA PROGRAM--In this chapter, suggestions are offered for the creation and implementation of an SQA program. Recognizing the wide variance of organizational structures, the presentation makes maximum use of generic functional activities rather than precise functional or departmental structures.

Chapter VI. CONCLUSIONS--In this chapter, a brief summary of the activities associated with the research is presented.
CHAPTER III

SOFTWARE

Definitions

The term "software" itself is responsible for many of the problems associated with its management and control. The newness of computer science coupled with its rapid technological growth has created a "mystique" about it which is continually reinforced by the introduction of new terms and "computer jargon." In a guidebook prepared for Air Force procurement offices, Systems Development Corporation presents an interesting discussion of the term software and attempts to explain its evolution (11). They even point out, in a less than complimentary fashion, that the government propagates the confusion by using the terminology differently from one regulation to the next. Similar observations have been made by Logicon (12) and Software Enterprises Corporation (3). The confusion stems from the very nature of computer programs. In reality, software is a thought process or a sequence of commands which when issued to a computer (or any machine, or any person) will produce a desired result. As such, it cannot be held or touched or measured. It can be measured or evaluated only by observation and analysis of its physical manifestations such as listings of the commands themselves or graphic representations of the logical flow or sequence in which commands are
issued. Early attempts to establish management and quality techniques to computer programs were therefore, addressed toward evaluation and monitoring of the documentation of the programs rather than toward the programs themselves. Software was treated as though it were a data item rather than a deliverable contract item (11). Progress was measured by "lines of code completed" and quality was measured by a successful demonstration of computer responses to prescribed inputs. Presumably, if a program contained one hundred instructions last week and now has two hundred, progress has been made. Studies of empirical data, however, indicates that 42% of those instructions will be thrown away due to design and coding errors before the contract is complete (3). Obviously, the quantity of instructions and their physical appearance are not effective measures of software or its quality.

Any attempt to develop effective management and quality assurance techniques must begin with a clear understanding of what "software" is, how it is developed and what are the measurable parameters to be used in assessing its quality. For purposes of this report the following definitions apply to the term "software":

Computer--A machine used for carrying out calculations or transformations under control of a stored computer program.

Computer Program--A series of instructions or statements in a form acceptable to a computer, designed to cause the computer to execute an operation or a series of operations.

Computer Data--Basic elements of information used by computer equipment in responding to a computer program. Such data can be external or resident within the computer.

Computer Documentation--Technical data, including listings and printouts, in human-readable form, which documents the requirements, design, capabilities, functional relationships and operating characteristics of computer programs and data.
SOFTWARE--A COMBINATION OF ASSOCIATED COMPUTER PROGRAMS AND
COMPUTER DATA REQUIRED TO ENABLE A COMPUTER TO PERFORM COMPUTA-
TIONAL OR CONTROL FUNCTIONS:

Operational Software--Software required to operate the system,
i.e., the software developed to satisfy the need.

Support Software--Any software designed to support the develop-
ment, maintenance or modification of other software.

Utility Software--A developmental tool used for the generation of
operational or support software.

Test Software--Software that is used to test or demonstrate the
capabilities of hardware or other software.

Software Quality

The most difficult obstacle encountered in an attempt to
establish a Software Quality Assurance Program is the determination of
measurement criteria. The question of what constitutes software
quality is currently the subject of extensive research in both the
academic and business communities. Like hardware, many characteris-
tics can be specified in terms of functional performance requirements,
i.e., it either works or it doesn't. For a long time, this was the
only criteria against which software adequacy was measured. Since
software is a precise arrangement of discrete instructions to a com-
puter, it was felt that qualitative characteristics such as reli-
ability and maintainability were meaningless. Recent experience,
however, has shown the opposite to be true (12). With minor redefini-
tion of classical measurement characteristics it is easily seen that
qualitative measures of software are not only possible but highly
desirable. Figure 2 presents a summary of some of the quantitative
and qualitative measures of software quality which are currently in
use. It is important that the reader recognize the tree-like structure of the measures. The final objective or goal is the attainment of "software quality" but to achieve it the quality assurance activity must address two distinct types of quality, functional and qualitative. At each successive "level" of refinement, the terms progress from generality to more specific requirements. Continued refinements eventually lead to precise standards which can be imposed on software designers and programmers to insure the attainment of the ultimate goal.

At the first level, software quality can be categorically divided into measures of functional adequacy and maintainability. Functional adequacy is the degree to which the software satisfies performance requirements specified in contractual documents. Maintainability, with respect to software, is its built-in design characteristics which facilitates the task of modifying the software to correct deficiencies or to incorporate new or expanded requirements. Whereas the goal of hardware maintainability is the ability to retain or restore the original design, software maintenance implies an alteration to the design. This is an important distinction and one which until recently, has eluded recognition and attention. By the very nature of developmental contracts, a product is only delivered when it has been proven to be functionally acceptable. Hardware maintainability can be specified and measured as a functional requirement by introducing known malfunctions, timing the repair time and statistically computing a Mean Time To Repair (MTTR). Such a test cannot be devised for software since neither deficiencies nor future
requirements can be anticipated. Although maintainability is considered to be measurable, the calculation of MTTR is impossible and meaningless. This necessitates a further refinement before "maintainability" can be measured.

At the second level of refinement, functional adequacy can be measured with respect to the manner in which software interfaces with its "peers": the host computer (efficiency), the human operator (ease of use, self-protection, etc.) and the system of which it is a part (reliability). The last term requires explanation. Software "reliability" cannot be compared to hardware reliability. Since a computer program has no physical parts it cannot wear out and failure rates are meaningless. Any failures which are detected are necessarily "latent defects." The concept of Mean Time Between Failures (MTBF) is meaningless since the occurrence of failures is not related to the probabilistic accumulation of tolerances in time but rather the occurrence of a set of conditions which were not accounted for in the design of the software. Since these conditions are not time related, nor are they probabilistic in nature, the calculation of MTBF is impossible and meaningless. Despite its ambiguity, software reliability is frequently referred to as the ability of the software to operate throughout its range of operating conditions without any regard to time frame.

As mentioned previously, software maintainability is a characteristic of software quality which has heretofore escaped serious definition and control. It is also the area where the greatest overall cost savings can be realized (refer back to Figure 1 to see
the relative cost of software maintenance). Software procurements frequently extend the state of the art, at least with respect to the tasks they are designed to accomplish. Similarly, most large scale software projects are developed over a long time period relative to the speed at which the technology is expanding. It is not uncommon that during the course of software development and its subsequent implementation, both the customer and the developer become more educated with respect to the system being procured. When this occurs they are likely to expand, reduce or otherwise adjust system requirements to produce a better product. Software maintainability directly affects the cost of such adjustments. At the second level of refinement, maintainability is further defined in terms such as flexibility, readability and expandability to accommodate measurement. Unfortunately, until recently, software was not designed with these characteristics in mind. The Department of Defense recognizing the fact that they were continually procuring entirely new software each time requirements were expanded, issued a set of directives which instruct the individual services to utilize life cycle procurement techniques (13, 14, 15). The idea behind life cycle procurement is that the entire cost of a system would be considered before contract award. Since the life of a system includes its entire period of operational usage, and since operational usage is characterized by continual expansions and redefinition of requirements, system maintenance, including software maintenance became a critical contract consideration immediately. Software suppliers, at least in the aerospace industry, are now faced with the problem of creating software which
is maintainable and must develop techniques to measure and demonstrate it.

At the next level of refinement, measurements of programming style and technique appear. As is shown in Figure 2, some techniques such as "structuredness" serve to enhance more than one measure of software quality. It is at this level that new programming techniques can be related to overall quality. It is likewise, at this level that software standards can be formally defined and imposed.

Figure 2 also aligns the various life cycle phases to the measures of quality shown. For example, the figure shows that during the Conceptual and Validation phases of development, the primary goal is to establish system requirements with respect to functional performance and maintainability.

Error Types

Functional adequacy of computer software is measured by the successful demonstration of its response to prescribed input conditions in accordance with the requirements imposed upon it. Errors which cause a computer program to be considered functionally inadequate can generally be classified into three general categories:

- Coding or Syntactical Errors
- Execution Errors
- Logic Errors

Ultimately, digital computers relate to binary instructions stored within their memories. Communication of instructions to a computer from a programmer must necessarily be in the form of these
Fig. 2. Characteristics of Software Quality
binary instructions. To facilitate easier communication, translators and compilers are generally furnished with a computer. A compiler is itself, a computer program which accepts English-like commands and Algebra-like expressions as input and "translates" them into their binary equivalents. Computer programming, or coding, is the process whereby the programmer prepares his instructions to the computer in accordance with the prescribed rules or grammar imposed by the type of compiler which he is using. Generally, the keywords and rules of grammar associated with a particular compiler are referred to as a "language" and are given the name of the compiler. For example, the FORTRAN compiler accepts commands written in the FORTRAN language. The order of a language is associated with the resemblance of the language to spoken language. FORTRAN, COBOL and PL/1 are considered to be High Order Languages (HOL's).

Syntax errors occur when the programmer misuses or otherwise violates the language in which he is coding his instructions to the computer. Since the compiler is unable to translate the instruction, no binary command can be issued to the computer and the program will not operate. This class of software errors occur during the actual coding stage of software development. Since they will inhibit the computer from operating, they have very little cost impact on the program being developed. Since compilers are generally programmed to print out diagnostic messages whenever they cannot decipher the input, syntax errors are extremely easy to correct. They are usually the result of typographical errors or improperly trained programmers.
and are so common that the coding stage of software development is generally referred to as the code and debug stage.

Execution errors, as the name implies, occur while the computer is operating or "executing its instructions." These errors are generally harder to identify and more expensive to correct. They are generally caused by exceeding the capacity of the computer itself or by misusing interfaces with other equipment. In the latter case, the errors are typically not discovered until coding is "complete" and the software has progressed to the test phase. Depending on the magnitude of the problem, extensive re-coding or even re-design could be necessitated to correct such an error.

Logic errors are design deficiencies. They are characterized by the fact that they are successfully compiled and successfully executed but do not produce the desired result. These are, by far, the most prevalent and costly errors which confront software projects today (2, 3, 5, 11, 12). It should be recalled from Chapter I that four of the six most common software problems were related to design deficiencies. Furthermore, they represent a class of problems which are introduced early in the development cycle and are discovered late in the cycle.

**Documentation**

The only physical manifestations of software are the effects it causes to occur and the documents which are used to describe it. Since "effects" cannot be observed until the software is complete, the **only** means available to monitor, evaluate and control software
development is its documentation. Virtually every reference contained in the bibliography of this report identifies and stresses the importance of software documentation. In addition to furnishing management visibility into the development process, it also serves to provide a critical continuity between stages. Just as each stage represents a distinct activity, distinct talents are required of the software engineers performing the tasks required for each stage. Very few individuals are capable of maintaining a high level of expertise in such a wide range of disciplines. Systems designers must be able to envision and devise overall relationships between both hardware and software in order to achieve system goals. Their preoccupation with this critical activity precludes their development of programming skills. Likewise, a good programmer is good because of his comprehensive knowledge of the computer and its capabilities. Both are equally critical to the software development process. It has been the personal observation of this author that, except in the smallest of software tasks, a single person cannot effectively accomplish both tasks. Documentation is the vital communication between these and other personnel associated with software development. Unfortunately, documentation holds a very low priority in the minds of those who are actually performing software development. Designers and programmers are generally chosen for their creativity and problem solving talents. They are generally motivated by the challenge of the problem and consider documentation to be a distraction from their prime mission of solving the problem. Recognizing only the visibility aspects of documentation, it is rationalized that documentation can be
accomplished after the problem has been solved and proven. Quite to 
the contrary, numerous studies of past software development activities 
indicate that the most common software problem is caused by mistaken, 
ambiguous or misinterpreted requirements (2, 3, 5). The truth is, 
without adequate documentation at each stage, the problem solvers 
don't understand what they are supposed to solve. Software management 
and quality personnel must insure that the continuity between software 
documents and subsequently between software stages is clearly estab-
lished and maintained. Later in this chapter, the "traceability" of 
requirements from conception to implementation will be identified as 
one of the most critical requirements of a software quality assurance 
function.

Despite individual preferences for naming conventions, 
especially all agencies involved in the procurement or development 
of software agree on the generically defined types of software docu-
mentation essential to software documentation:

- **System Requirements Specification**--This document is used to 
define the overall goals of a system of hardware, software and 
personnel. It must clearly itemize all requirements and must 
"allocate" these requirements to various sub-systems, one of 
which is the software. All interfaces between various sub-
systems, including the software, must be clearly established.

- **Software Requirements Specification**--This document identifies 
all requirements of the software in specific "functional" 
terms. It must itemize all functions which the software must 
perform. It should not specify how it will perform these 
functions, but must describe precisely what the functions are.

- **Software Design Specification**--This document describes in 
detail the manner in which the design requirements are imple-
mented. It is generally prepared in two parts corresponding 
to the activities being performed, detailed design (logic 
diagrams) and coding (listings). It is the document which is 
the representation of the delivered product. Its completeness
and accuracy is essential for later modification of the software.

- Test Documentation—Generally a test plan and a family of test procedures are used to document the test requirements and results expected to verify that the software requirements placed on it.

The above list of documentation represents the least possible level of documentation needed to accomplish a successful software development. In large scale software efforts, the degree to which they are formally controlled critically affects the success of the project.

**Development Process**

All manufactured items evolve through a sequence of relatively discrete events: the requirements of the item are defined, the item is designed, it is produced and tested and finally reproduced in large quantities. These phases of the manufacturing process are usually described as:

- The Requirements Definition Phase
- The Development Phase
- The Test Phase
- The Production Phase

In a typical hardware procurement, the majority of quality assurance and control activities are performed during the production phase. Inspection and test procedures are established throughout the assembly process to identify, isolate and correct or discard items which do not conform to prescribed parameters and tolerances established during the development and test phases. Unlike hardware,
software has no production phase. Identical copies of computer programs are easily reproduced and verified. Software is a product of engineering, not a product of manufacturing. The Defense Logistic Agency Manual 8200.1, which directs the activities of government contract administrators, emphatically stresses this point:

Computer Software is never a "production item." All Procurement Quality Assurance actions must be completed during the "Development Phase." The Development Phase usually consists of four separate stages: Definition, Design, Programming and Test. (16)

As will be discussed in subsequent sections of this report, the four "stages" identified are recognized as the chronological sequence of software development activities which can and must be controlled. They are, however, only four of the seven major phases through which software passes during its "life cycle." Specifically, a phase must be considered prior to requirements definition to insure that software will perform its role as part of a larger system which, in all likelihood involves hardware, other software, people and procedures. Likewise, after software has been developed, it must be integrated into the system for which it was designed. Finally, the operational use of the software must be considered. It will be shown in Chapter IV that effective software management and quality assurance is possible only if all phases of the software life cycle are considered prior to and throughout the development portion.

Life Cycle Concepts

Alarmed by the ever increasing cost and schedule overruns on systems in general and computer software in particular, the Department of Defense issued a series of Directives in early 1977 to each of the
individual services (13, 14, 15). These directives require that each service develop and implement standards and regulations to govern all DoD procurements including software. The thrust of the new approach is to formally recognize and plan around the seven phases discussed in the previous section of this report. The concept is based on the principle that for any given system, its total costs must be considered before it is purchased. Total costs are defined as all expenses incurred from conception through dismantling. As shown in Figure 3, the software life cycle consists of seven distinct phases:

- **Conceptual Phase**--During this phase, system objectives are defined and trade-off studies are performed. A request for Proposal is generated and competing firms are invited to bid for the contract. A system specification is the final product of this phase.

- **Requirements Definition**--This is the most important phase in the overall design of the system. Detailed requirements and allocations must be firmly established and proven consistent with the overall objectives defined during the conceptual phase.

- **Design Phase**--It is during this phase that requirements are transformed into design configurations.

- **Coding and Checkout**--This phase consists of the implementation of the design created in the previous phase. Software is actually coded and debugged and checked for proper installation. It is interesting to note that within the last decade, this phase was thought to be the entire software development cycle.

- **Testing**--This phase consists of a considerable array of demonstrations, qualification tests and functional acceptance tests. It is here that the developed software is proven to meet the requirements defined earlier.

- **Integration**--When the software has been accepted as a functional subsystem, it is integrated into the system for which it was designed. The entire system is then tested for compliance with overall requirements.
Operation--This is the period of time for which the system is used for its intended purpose. Both the hardware and the software must be "maintainable" throughout this entire phase.

Need for Early Quality Assurance

The life cycle depicted in Figure 3 can be directly related to the logical and chronological evolution of software requirements into a software product. Figure 4 depicts the life cycle in a slightly different manner to highlight some very important aspects of software management and quality assurance. The process of software generation generally follows a sequence of events which repetitively allocate the requirements of a given item to those sub-items which make it up. In the figure it can be seen that the overall system requirements have been allocated to the three subsystems which comprise the overall system. At the next step, the computer program requirements are in turn allocated to the individual software modules which make it up. When successive "decomposition" reaches a level where further allocations are unnecessary, coding begins. After coding is complete, the reconstruction process begins. Individual modules are tested and integrated back into the overall software subsystem which is in turn tested and integrated with other subsystems. Finally, the overall system is tested, accepted and deployed. This is essentially a "top-down" design process, a "structured" or "modularized" coding and checkout process and a "bottom-up" test and verification process. Although this approach is not the only one used for system development, it highlights some very important aspects of the development cycle.
Fig. 3. Software Life Cycle and Major Decision Points
Fig. 4. Stages of the Software Development Process
Figure 5 dramatizes the effects of an error introduced into the life cycle at an early stage of the development process (i.e., during system requirements definition). This could occur as a result of a misinterpreted or ambiguously worded system specification. Once the error is introduced, it is passed to successive stages undetectable until ultimately the system is completely developed and integrated. The following key points deserve emphatic mention:

- An error introduced at any stage will be propagated into subsequent stages.
- An error introduced during any stage will be magnified by subsequent stages.
- The earlier in the development cycle an error is introduced, the later it will be detected.
- The earlier an error is introduced, the more serious will be the impact.

Sadly, all of the studies of software errors types which were reviewed during the course of this research as well as personal observations of the author confirm the worst: 65% of all software errors are introduced during the requirements and design stages (2, 3, 5). The messages to software managers and quality assurance personnel should be obvious: the greatest need for rigid controls and the most cost effective time to expend funds is during the earliest possible stages of the development cycle and effective control points must be established at each stage to insure that errors introduced during the stage are not allowed to propagate.
Fig. 5. Error Propagation During Software Development
Need for Requirements Traceability

Figure 6 adds the critical documentation which accompanies the software development cycle depicted earlier. After thorough analysis and feasibility studies, a system specification (A-level) is issued as part of the request for proposal. Functional Performance Specifications (B-level) are developed by contractor as the first activity of the awarded contract. This is accomplished during the requirement definition and functional design stages of the life-cycle. When the B-level specifications are approved, they become the contract documents which define the individual deliverable end items. It is these specifications which determine contract compliance. They are, therefore, the base line from which detailed design and testing documentation must be developed. It is virtually impossible to successfully satisfy the contract unless:

- The Functional Baseline (B-Specs) are correct and complete.
- Each performance requirement specified in the Functional Baseline (B-Specs) is correctly implemented in the Detailed Design (C-Specs).
- Each performance requirement specified in the Function Baseline (B-Specs) is verified (Test Plan).
- Each item of verification (Test Plan) must be adequately tested (Test Procedures).

All interconnecting lines depicted in Figure 6 are essential. It must be possible to trace every individual performance requirement to a corresponding portion of the detailed design. Likewise, it must be possible to trace the same requirement through the test plan to an individual test procedure which demonstrates its accomplishment.
Fig. 6. Critical Continuity Loop
Adequacy of the test procedure is determined only by a direct traceability between it and the detailed design.

A software quality system which could control the continuity between each of these documents would eliminate 65% of software errors which are in turn responsible for 80% of unplanned software costs (5). The remaining 35% of software errors are made during the module coding stage and discovered during subsequent low level module testing. Since coding errors are best controlled and corrected within the software engineering organization, Quality Assurance and Control should participate only in a monitoring and advisory role. If the contract specifically identifies a specific coding technique or characteristic, a sampling technique will be necessary to verify compliance. Otherwise Quality's role in this stage should be minimal. The remainder of this section will therefore address only the "critical" loop shown in Figure 6.

The responsibility for the generation of clear and unambiguous system concepts clearly lies with the procuring agency. It is not uncommon, however, for the customer to deliberately avoid discussing technique in his RFP. In addition to inviting price competition, he is also encouraging each competitor to choose and justify a technical approach. Quality participation in this stage of the development cycle should be advisory in nature.

Virtually every study and every advisory currently being published emphasizes the importance of the functional performance baseline specifications. Inaccurate and/or incomplete B-level specifications are without a doubt responsible for the most serious
and costly software problems (5). The system level specification provides a definition of the end items which compromise the system, their functional requirements and the interface requirements to cause them to act in harmony to meet overall system requirements. With respect to software requirements, the type of computer, language to be used and required coding techniques, if any, must also be clearly defined. In addition to the technical narrative, a "System Flowchart" should be included. A system flowchart depicts all components of the system with interconnecting lines to show interface requirements. Everyone associated with the project should be aware that this is the last time the system will be treated as a system until final integration and system testing. Any misconceptions uncorrected during this stage will be included in the subsequent design and construction of individual components.

B-level specifications are also produced for each subsystem. The Software Performance Specification is the contract document for the deliverable software. It describes in detail all of the performance requirements which must be designed into the software and identifies the Quality Assurance techniques which will verify each. It must supplement and be in complete accord with the system specification. It usually contains a "functional flow chart" to graphically portray the interactions between major program modules and data files. Once again, the accuracy of this document is of paramount importance. It is the document which will be used to determine acceptability of the software end item.
Armed with an approved functional baseline (B-5 Specification), the software design team begins the design process. The detailed design specification is prepared. This document is a comprehensive technical description of the developed computer program. As such, it is a documentary product of the development effort. Unlike the B-level specification, this specification does not have a role as a contractual compliance instrument. It is, however, a deliverable data item whose accuracy and completeness is essential for program "maintenance" during system development. It is also the primary indicator of software progress and compliance for in-house maintenance and Quality Assurance activities. In addition to the detailed technical parameters of each portion (module) of the program, detailed logic diagrams are essential elements of the C-level specification. Unlike system and functional flow diagrams which show components and the interactions, a logic diagram depicts the actual steps to be taken within the program and their logical sequence of execution. They will be used directly during the coding stage of the software development. Properly prepared, they are a "picture" or visualization of a computer program. Computer coding, although "readable," is not a useful tool for verifying logical correctness and completeness. By comparing logic diagrams contained in the C-level specification to the functional diagrams contained in the B-level specification, management can effectively monitor progress and Quality can maintain control.

Generally, deliverable computer programs are relatively large and involve distinct functional requirements. Therefore, programs are
typically segmented functionally into self-contained modules or sub-programs. The logic diagrams contained in the C-level specification generally consist of program logic diagrams which show the entire program logic diagrams for each individual module in terms of actual commands.

The system and computer program test plans are the documents used to verify contract compliance. With respect to software, compliance can be demonstrated in a variety of ways, primarily by demonstration. Unlike hardware, physical characteristics cannot be measured or observed. Observation is limited to observing predetermined effects of operating the software under specific conditions. Since the "effects" could be caused by any of a multitude of logic paths, their observance is meaningless unless the witness is thoroughly aware of both the specific conditions and the detailed design of the software. That is, the step by step test procedures must be identified and correlated to the software logic diagrams (C-specification) to insure that the expected effect actually demonstrates software compliance.

During the development phase, three distinct software development activities take place: detailed design, coding and checkout, and testing. Management resumes the role of monitoring and controlling the progress of the contract. Engineering designs and constructs the software product to be delivered in accordance with established functional requirements and software engineering standards. Quality must insure continuity of requirements throughout the phase. That is, each functional requirement must be traceable to a corresponding
detailed design and a corresponding test or demonstration. This continuation is essential and must be thoroughly controlled. Experience has shown that design inconsistencies and incompatibilities introduced during the phase are the most frequent and most expensive problems facing the software developer.
Historically, computer software has been "controlled" by a formal demonstration or test of each of its functional requirements. If properly accomplished, this approach does, in fact, guarantee that only functionally adequate software will escape the "develop, test, re-develop, re-test" loop shown in Figure 7. Two significant points can be made.

First, only the functional aspects of the software are being controlled. As was discussed in Chapter III, more than half of the overall life cycle costs of the software product will be spent maintaining it. Since the measurement criteria is based on functional adequacy, and since each time a test is failed the software is "repaired" and re-tested, it is not immediately obvious that each successive repair cycle is further degenerating the maintainability aspects of the software. Since the software eventually exits the loop by passing all functional requirements, it appears to be a quality product. Unfortunately when, during operational usage, it is determined that modifications to the software are needed, the user finds that it is impossible due to the many and varied "patches" which were incorporated prior to delivery. Software patches are small changes
to the software which correct a functional deficiency. Like patches on clothing, a few small patches do not significantly degrade the overall quality of the product. Many or large patches, on the other hand, render the software unreadable, inflexible and consequently, unmaintainable. Further, statistical evidence shows that the probability of introducing errors when fixing a known error is higher than it was when the software was being originally created (5). Obviously, it is better (and cheaper) to do it right the first time.

The second point to be made is based on simple economics. In Chapter III, it was shown that an error introduced in any stage is propogated into later stages. By allowing software development to proceed through its stages without timely reviews and checks, errors are allowed to propogate up to the test stage before they are recognized. Statistical evidence shows that up to 40% of good code is discarded because it reflected bad design (3). If the design had been verified prior to coding, considerable savings could have been realized.

Assurance Versus Control

The historical approach depicted in the previous section is one of control not assurance. Further it controls only the functional adequacy of the software product. Air Force Regulation 74-1 defined the scope of quality assurance:

A planned and systematic pattern of all actions necessary to provide adequate confidence that material, data, supplies and services conform to established technical requirements and achieve satisfactory performance. (17)
Fig. 7: Classical Approach To Software Quality Control
Quality Assurance covers the entire scope of activities associated with the creation and development of a product. In the case of software, it must encompass all stages of development.

Figure 8 presents a pictorial representation of the distinction between historical software quality control and software quality assurance. As stated earlier, the Department of Defense is a major customer of computer software. In response to its directives, the individual services have established a standard which will be used in all future procurements. The document is an extensive revision of an earlier Army Standard, MIL-S-52779 (AD). As of April 1979, the new version (same without the "AD") had been fully coordinated and was at the Pentagon awaiting final review by the Joint Chiefs of Staff (18).

This document dictates the requirements of a full scale software quality assurance program and will be cited in all future military procurements involving software. Similar standards for commercial software development are being prepared by various engineering societies (29). However, commercial standards are typically harder to impose since contracts are usually drawn to purchase existing software packages. Large scale software contracts are typically sold along with a "maintenance agreement" which is itself another contract to correct latent deficiencies and incorporate new requirements. It is the opinion of this author that eventually even commercial software will be purchased for its maintainability characteristics as well as its functional capabilities. MIL-S-52779 states:

The contractor shall develop and implement a Software QA Program to assure that all software delivered meets all contractual requirement. This program shall provide for detection, reporting,
Fig. 8. Quality Control Versus Quality Assurance
analysis and correction of software deficiencies. Contractor personnel performing quality functions shall have the responsibility and authority to evaluate software development activities and to recommend improvements. This program, including procedures, schedules, processes and products shall be documented. (18)

Within the text of the standard, ten activities are described to fulfill the requirements of a software quality assurance function. They are presented here since, although the terminology changes from one organization to the next, they are representative of the conclusions drawn by most of the authors of articles which were reviewed during this study. The requirements state that for any software procurement, a formal Software Quality Assurance Plan must be prepared and must present the supplier's intention and methodology for assuring the quality of each of the following:

- **Tools and Technologies**--The supplier must have a means of reviewing and evaluating all software support equipment and programs.

- **Program Design**--Provisions must be provided to allow design review for fulfillment of requirements, completeness and compliance with standards.

- **Work Certification**--The Software QA program must insure the work tasking is accomplished in accordance with pre-planned agreements and authorizations.

- **Documentation**--All software documentation will be prepared in accordance with pre-agreed standards and will be audited periodically for compliance.

- **Library Controls**--Procedures shall be established and implemented to assure that after acceptance testing, software is protected from unauthorized alteration.

- **Reviews and Audits**--The Software QA Program will provide for periodic independent reviews of the documentation, design, code and test results throughout the development cycle.

- **Configuration Management**--Periodic audits will be accomplished to insure proper management of all software configurations.
Test Monitoring—This activity includes analysis of requirements for testability, analysis of test plans and procedures for adequacy and evaluation of test results.

Corrective Action—Software QA will provide for quick detection, reporting and correction of software errors throughout the development.

Subcontractor Control—The Software QA program will impose similar requirements on all subcontractors and monitor them for compliance.

Figure 9 depicts the requirements of MIL-S-52779. Unfortunately, neither the standard nor the figure tell the vendors how to implement such a program. It is obvious from the previous analysis that the entire software development cycle must be engulfed by the Software Quality Assurance Program. The organization must devise plans, procedures, standards and reviews to impose on the development, but when, how and by whom? All of these questions are troublesome. Common sense suggests that SQA must occur throughout the development with special emphasis during the requirements phase. However, during early stages, the problems have not yet occurred (at least no one has seen them yet) and it is particularly difficult to inject an SQA engineer into the systems design organization to "look over their shoulder." They resent it and project management sees it as a waste of time and money since they don't anticipate any problems. In a speech presented in early 1979, Mr. Jeff Parnes of the Defense Contract Administration Service summed up a typical software development cycle as follows:

Wild enthusiasm is evident as the system requirements are developed. As the software requirements are finalized, disillusionment sets in. Preliminary design begins the search for quality. By the code and debug stage, punishment has been meted out. The promotion of the nonparticipants occurs during the
Fig. 9. MIL-STD-52779
test phase. By the time the operations and maintenance period has arrived, we are faced with late system delivery, cost overruns, degraded performance and an unreliable product. (20)

A meaningful Software Quality Assurance Program must begin during the very first stages of development and continue throughout all subsequent stages. The next question asks: How can SQA be imposed and by whom? The answer is not simple. The size and structure of organizations which develop software varies so widely that a simple list of required activities and a single organizational element would be meaningless to all but a few companies. The section which follows presents a somewhat philosophical approach to the concepts of software quality assurance. It is felt by this author that the approach could be applied to any organization even though the results could be quite different from one organization to the next.

Analysis of the Development Process

The requirements set forth in the previous section appear to describe all the activities needed to adequately assure software quality. They also appear to be impossible or at best, difficult to attain. For example, reviews are a good technique for monitoring progress and performance. But when should they be conducted, what critical parameters should be reviewed, who is to be reviewed and by whom? These types of questions have no universal answer due to organizational differences between companies. There is however, an approach to answering them which is universal.

The first step in the approach is the analysis of the software development process itself. Software development, like software
itself, is a sequential arrangement of activities which transforms software requirements into software products by way of a logical process. Top-Down design is a software technique which is currently receiving widespread attention and success. It begins with a simple description of the entire system design. Through a succession of refinements of what has been defined at each level, lower levels are specified. The process is continued until the design reduces to a set of easily implemented functions or design statements. These low level, easily verified functions are guaranteed to fit back together since at each step of the refinement process, the lower level was devised to satisfy all the requirements of the higher level. It is the opinion of this author that the same process can be applied to the creation and implementation of the software quality assurance function.

Figure 10 illustrates the first level of refinement. The software development activity has been broken down into its stages. If the quality of the overall software development cycle is to be assured, it is obvious that each stage requires assurance of its quality. Depending on their individual organizational structures and operating procedures, a company can further "decompose" any or all of these functions to isolate specific activities which occur during software development. An example of another refinement is also shown in Figure 10. It should be emphasized that whenever an activity is fragmented into lower level activities, care must be exercised to guarantee that the input to the first lower level activity is identical to the input to the high level activity being decomposed and
the output of the last low level activity must identically match the high level output. This will guarantee the completeness of the eventual low level description of the software development process. It should be noted also that up to this point, no attempt has been made to suggest where or how to impose quality assurance provisions. It is critically important to completely understand the individual company's software development process before a software quality assurance program can be devised for that particular company.

When the development process has been completely defined in simple discrete activities, the development of a software quality assurance program can be likened to the optimization of a multi-stage dynamic programming problem with additive returns. A model can be hypothesized wherein the "state" variables represent the current form of the software (e.g., requirements defined, functional design complete, etc.). The "returns" at each stage represent the errors which are introduced and uncorrected during the stage. The objective is to minimize the overall return subject to the cost, time and technological constraints imposed by the contract. In Chapter III, it was shown that software errors are multiplicative in that an error introduced at one stage will propagate through subsequent stages. Recognizing the impossibility of negative errors, it is intuitive that the return function defined is monotonically non-decreasing. Using the techniques of dynamic programming it can be shown that the overall software development model can be decomposed into its sequential stages for purpose of optimization (21). The problem of creating an optimal software quality assurance program reduces to many smaller
Fig. 10. Decomposition of the Software Development Cycle
problems of imposing software quality assurance on the individual stages of the development process. For any given stage, this task is considerably easier than would be immediately obvious.

When the organizational software development process has been decomposed into a logical sequence of individual stages, each can be considered for application of software quality assurance techniques which are "personalized" for the stage. In order to accomplish this, one need only to investigate the very nature of the stages themselves. Referring back to Figure 10, it is seen that each stage consists of the following:

- Inputs from the previous stage
- An activity to be performed on those inputs
- A product which results from the activity
- An output to the next stage

The task of imposing quality assurance on the stage reduces to one of analysis of each of these individual entities.

Since the input to this stage was the output of the previous stage, it can be assumed to be error-free since the same rules derived herein would have been imposed during that stage. Likewise, the output of this stage will be quality assured if the activity and the product are controlled.

An activity can be accomplished only if there is an agency to perform it. Since various stages are accomplished by various organizations, the generic term "developer" is used to denote the agency tasked with performing the activity. Similarly, an activity can be monitored to insure that it is being conducted in accordance with
pre-determined concepts of how it should be done. The generic name "monitor" is used to identify such a person or agency who oversees the developer.

A product is the result of the activity. To assure its adequacy, an "evaluator" must compare or test it against the requirements of the stage. Since this product, when evaluated successfully, will become the input to the next stage, it must be "controlled" to assure that it does not degrade after leaving the stage. Figure 11 illustrates the concepts of a stage with the various agencies depicted. The figure clearly shows that the stage has been "encapsulated" by a software quality assurance program. "How" is the only question which remains.

Consideration of the manner in which individual agencies are tasked with their respective responsibilities will reveal the manner in which quality is insured. The developer is tasked with the responsibility of converting the inputs into a prescribed product in accordance with certain pre-established guidelines or standards. He must understand what he is to do and how he is to do it. The monitor has a set of prescribed procedures for verifying that the developer is doing the correct activity and doing it within the constraints imposed by his standards. The evaluator checks the product in accordance with established standards by comparing it to the same requirements imposed on the developer. The controller follows established procedures for safeguarding the integrity of the product.

Replacing all generic terms with those commonly utilized by software developers, it is easily seen (refer to Figure 12) that
Fig. 11. Activities Associated with a Software Stage
Fig. 12 Total Quality Assurance of a Software Stage
software quality can be assured at any given stage by enveloping the stage in:

- Software Engineering Standards
- Quality Procedures
- Test Standards
- Configuration Management Procedures

The development of a Software Quality Assurance Program consists therefore of decomposing the development process into its most basic stages, establishing procedures and standards for each stage and then pulling together all of the results into a unified Software Standards Manual, a set of Quality Procedures, a set of Test Standards and a set of Configuration Management Procedures. These documents are subsequently imposed on the respective organizations.

Although four agencies are described above, it is unnecessary for all organizations. Many companies, particularly those where software represents a small portion of their overall business would find it impossible or prohibitively expensive to create separate agencies for each function. For example, many organizations do not have a separate testing organization. The engineers who design and perform the activity are expected to devise and perform tests to demonstrate the product. Although the personnel performing the test activity should be different from the developers, there is no reason for them to be from a separate organization. The process of decomposition described earlier can be used to separate the stage further if needed. That is, the test function itself can be considered to be a stage
itself. The input to the new stage is an untested product. The product of the stage is a tested product.

The process described can be continued to any degree desired. In its limit it reduces to a two agency organization of developers and monitors, neither of which can be dissolved by repetitive decomposition. If carried to an extreme, each developer would have his personal monitor and both would be totally controlled by their standards and procedures. Each individual organization must decide where to terminate the decomposition process to achieve maximum results for their software quality assurance program.
CHAPTER V

IMPLEMENTATION OF A SOFTWARE QUALITY ASSURANCE PROGRAM

Corporate Commitment

Software Quality Assurance is a relatively new concept and cannot be implemented without the full support of the top echelons of company management. As was shown in previous sections, considerable analysis is required to properly establish such a program let alone implement it. It is conceivable that the professional satisfaction of producing a high quality product is not sufficient motivation for implementing an effective program. Ultimately, all companies respond to monetary considerations. In many cases, the first task facing the software quality organization is justifying its own existence. Management must be "educated" to understand the special characteristic of software which distinguish it from hardware. It must be remembered that software is less than thirty years old. Most top managers attained their seniority at a time when software was either unheard of or considered to be unimportant in relation to hardware. When the software life cycle is explained and the critical aspects of its development cycle are compared to the design engineering aspects of hardware development, they readily accept the notion of imposing standards and controls throughout the process. A
"Monday morning quarterback" explanation of past software problems will further emphasize the need for effective quality assurance. As Fred Brooks asserts in his book "The Mythical Man-Month" many managers feel that software problems can be corrected by the addition of man-power. They are frustrated when their actions further degrade the project (22). It has been the personal observation of this author that management is very responsive to the concept of formalized software quality assurance when they understand it and can look back at the cost and schedule problems that could have been averted by its application.

In any case, an effective software quality assurance program cannot be implemented without a complete corporate commitment, usually in the form of a company policy. The policy should be a simple statement of commitment without expanding on the details for its accomplishment.

**Organizational Directive**

Armed with the support of top management, the individual organizations of a company must define responsibilities and authorizations. Based on the organizational structure of the company, the roles of each functional organization must be related to the software development process and the task of quality assurance. Since software quality assurance will necessarily cross at least two organizational boundaries, their respective areas of responsibility and authority must be precisely set forth in a mutually agreed upon directive. Without doubt, there will eventually be differences of opinion with
regard to the acceptability of a product or the degree to which it is
to be monitored or tested. It is essential that specific lines of
authority be established in the calm and professional atmosphere of a
working group rather than during the "heat of the battle" later on.

**Functional Directives**

When the various functional organizations have been tasked
with individual responsibilities, each should establish its own set
of "Umbrella" procedures or directives. These become the authority
documents within the organization. They direct the way in which their
individual standards and procedures are to be implemented and
enforced. They provide individual workers with instructions directly
related to their jobs. For example, in the quality organization, the
"umbrella procedure" may state that the corrective action function
will be the task of department "xxx" and will be performed in
accordance with quality procedure "yyy."

**Procedures and Standards**

As soon as the corporate commitment is made and organizational
responsibilities are defined, preparation of standards and procedures
can begin. The process described in Chapter IV can be applied to
insure that the results are complete or existing standards can be
modified to include provisions for software quality assurance. Sooner
or later, the standards and procedures must be compared to the organi-
zation of the software development process to insure that all aspects
are covered.
Training

A continuing training program should be established to acquaint all personnel involved in software engineering, design, management, test and quality assurance. Different courses should be created and developed for each specific audience. Project managers should be educated with respect to the need and importance of software quality assurance. This course should include a complete discussion of the software life cycle and appropriate comparisons to hardware development. Designers and engineers should be continually kept abreast of new techniques and practices. Test and quality personnel should be adequately trained in the standards and techniques being used by engineering personnel.
CHAPTER VI

CONCLUSIONS

Software Quality Assurance and Control is an attainable necessity. Lack of adequate control in the past has resulted in many large contract overruns of both cost and schedules. Adequate control however, requires an expansion of existing quality organizations to include "Quality Software Engineers" trained in all aspects of the software life cycle including the specification techniques and design techniques (flow charts and coding) as well as verification methods. As in hardware, the level of involvement by quality assurance personnel will determine (or be determined by) the degree of risk involved. On one end of the spectrum little or no formal quality assurance is exercised. Its cost is negligible but it serves no useful purpose. On the other end is "100% sampling." In effect, every detail of the software development is accomplished twice, once by the responsible engineer and once by the software quality engineer. From a cost point of view, this approach is equally as unattractive as the opposite extreme. The relative cost of the software, its complexity, its importance to the system and historical evidence of programming disciplines within the software engineering organization should allow the construction of a meaningful cost curve which
reflects the returns expected for a given level of quality assurance. Unfortunately, such data is unavailable due to the lack of effective programs in the past. Until historical evidence is accumulated, the judgment of experienced software experts who are, at least, familiar with the effects of ineffective quality assurance must be relied upon.

Establishment and operation of a meaningful software quality assurance function requires a joint understanding or policy between project management, software engineering and software quality organizations. Management must be prepared to assist in the risk determination and implement the consequent cost and schedule adjustments to accommodate the effort. Software standards and control techniques must be developed and implemented. Finally, training programs must be established to continually expose all software personnel to emerging software techniques, tools and methodologies.

All objectives of the study were met or exceeded. The literature search reaffirmed the existence of an intense interest in software management and quality assurance throughout the industry. Valuable insight was gained by reviewing the problems being experienced by others and their attempts to alleviate them. Subsequent participation in various working groups revealed that a considerable amount of work is being accomplished. Unfortunately, most work is being accomplished by individual groups and agencies and is not yet being published. Very little effort is being expended toward unifying their observations and conclusions. It is hoped that this research report as well as the inter-agency communications which it has prompted will serve to stimulate further cooperation between software
developers and users and will promote a concerted effort toward the
development and implementation of universally recognized software
quality assurance techniques.
LIST OF REFERENCES


