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

LIST OF ILLUSTRATIONS

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

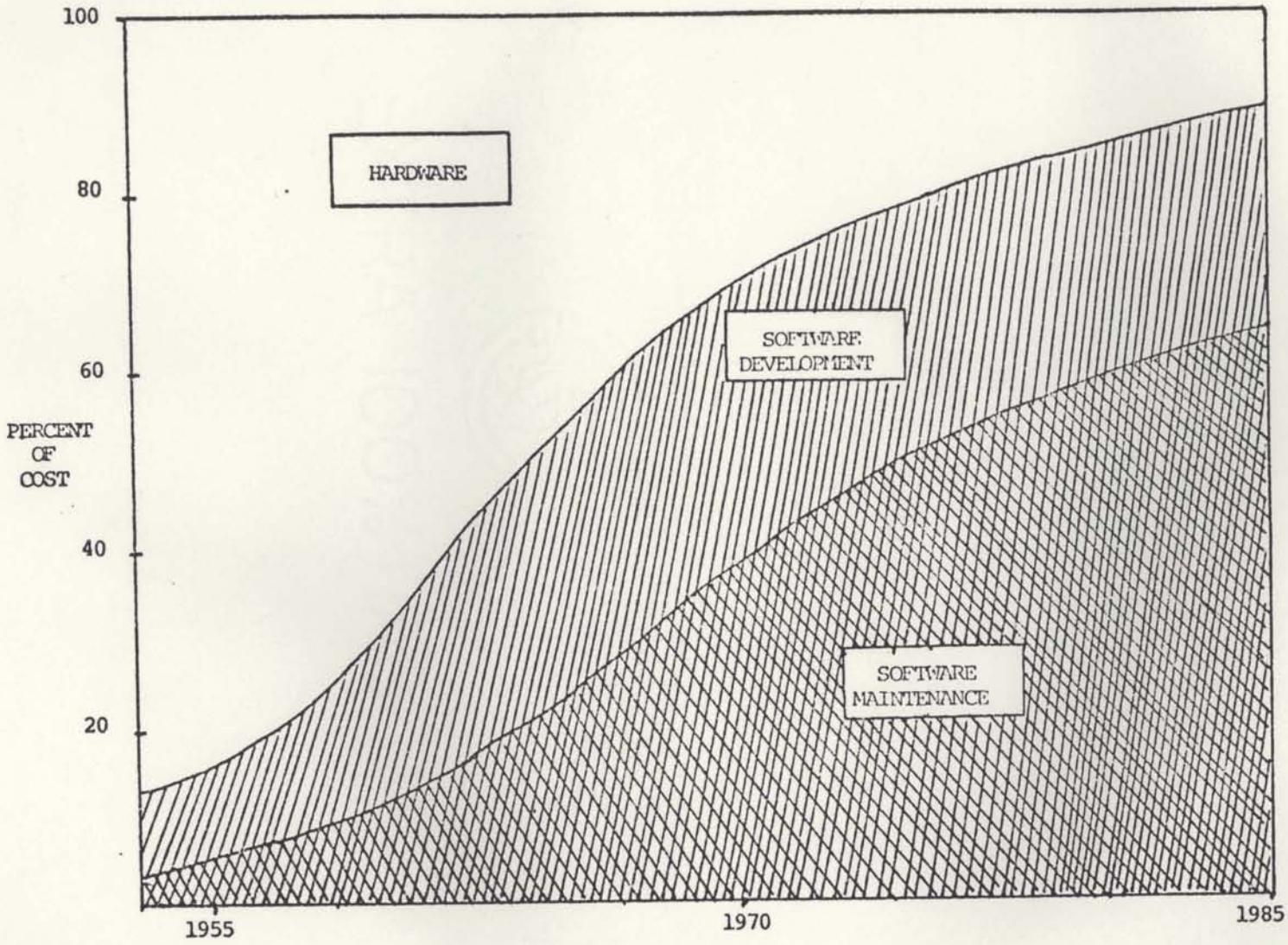


Fig. 1. Hardware / Software Cost Trends

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

- Is unreliable

  --Exhaustive test is impossible
  --More money, more tests

- Is inadequately documented

  --There were no standards
  --You got what you paid for

- 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. Diijkstra 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 propogates 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 COMPUTATIONAL 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 development, 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 characteristics 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 computer, it was felt that qualitative characteristics such as reliability and maintainability were meaningless. Recent experience, however, has shown the opposite to be true (12). With minor redefinition 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 heretofor 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.

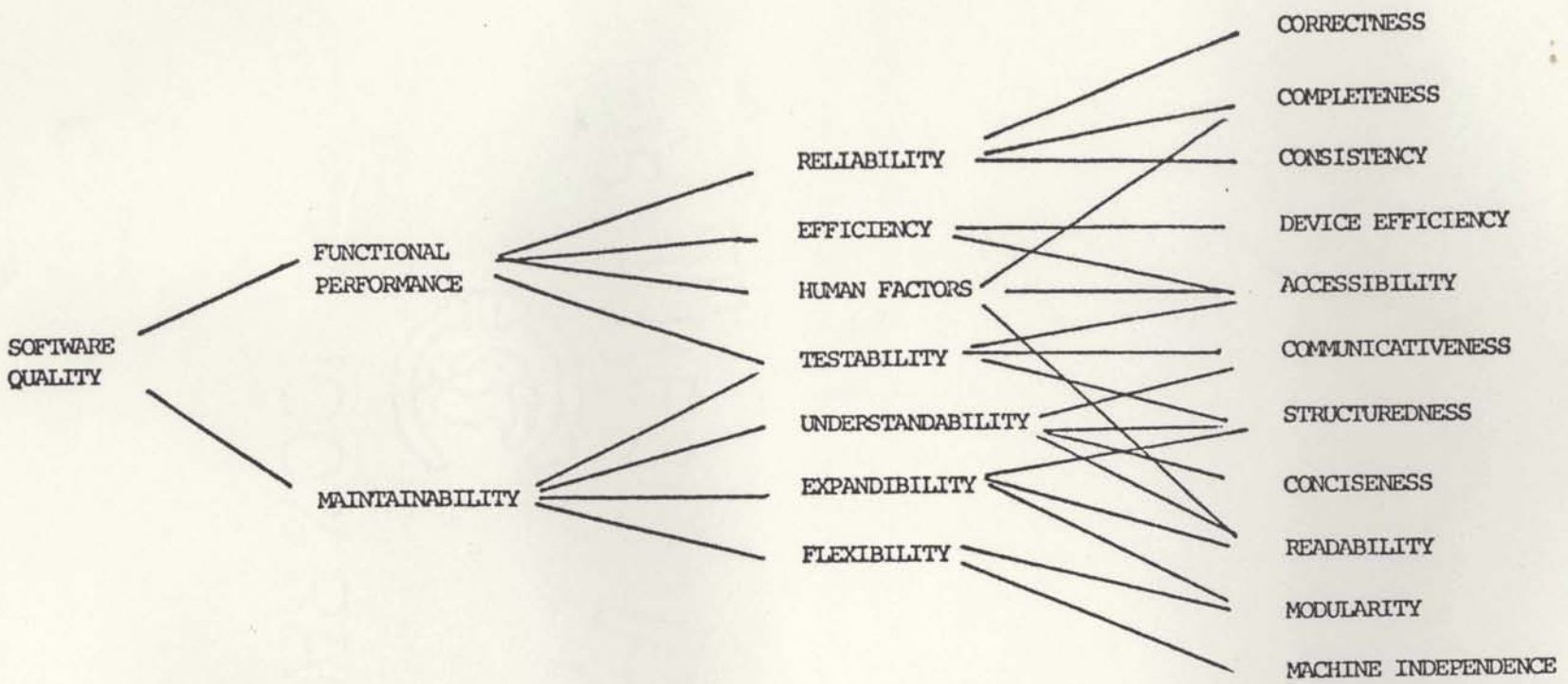
Figure 2 also aligns the various life cycle phases to the measures of quality shown. For example, the figure shows that during the Conceptual and Validation phases of development, the primary goal is to establish system requirements with respect to functional performance and maintainability.

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



GOAL:	SYSTEM REQUIREMENTS	MEASURABLE AND CONTROLLABLE QUALITIES	SOFTWARE ENGINEERING STANDARDS AND TECHNIQUES
LIFE CYCLE:	CONCEPT AND VALIDATION	DETAIL DESIGN AND TEST	CODE AND DEBUG

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 visability 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 visability 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 established 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, essentially all agencies involved in the procurement or development of software agree on the generically defined types of software documentation 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 implemented. 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.

- o 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:

- o The Requirements Definition Phase
- o The Development Phase
- o The Test Phase
- o 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.

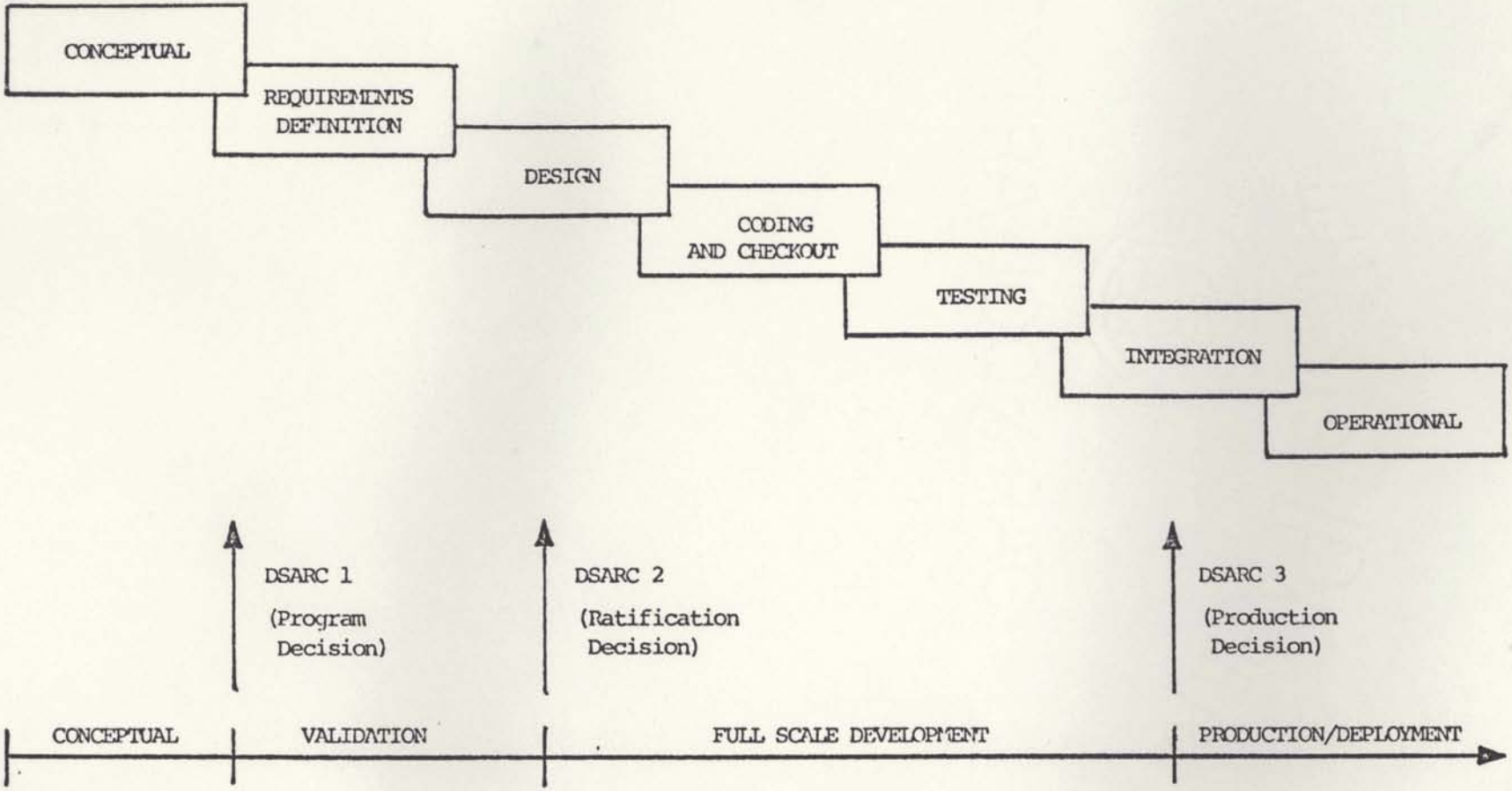


Fig. 3. Software Life Cycle and Major Decision Points