To the Graduate Council:

I am submitting herewith a thesis written by Roger D. Pelham entitled "The Role and Implementation of Software Quality Assurance in the Implementation of a Quality Improvement Program." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

S. I. Hariharan.

S. I. Hariharan, Major Professor

We have read this thesis
and recommend its acceptance:

W. C. Armstrong

F. W. Brooks

Accepted for the Council:

Lew Minkel

The Graduate School

4

THE ROLE AND IMPLEMENTATION OF SOFTWARE QUALITY ASSURANCE IN THE IMPLEMENTATION OF A QUALITY IMPROVEMENT PROGRAM

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Roger D. Pelham

December 1984

ACKNOWLEDGMENTS

The author wishes to express sincere thanks to his thesis committee members, Dr. S. I. Hariharan, Dr. F. W. Donaldson, and Mr. W. C. Armstrong, for their personal guidance, direction, and constructive criticism which made this study possible.

A special thanks to Mrs. Peggy Lominac for her assistance in proofreading this material for grammatical correctness, and to Mrs. Jeannie Harper for her dependable assistance and aid in typing the final manuscript.

ABSTRACT

As software has become an increasingly important part of systems performing complex and critical functions, the risks of having a software-caused failure have dramatically increased. Software personnel now generally agree on the need for methods of increasing software reliability by eliminating errors made during software development. Industry and academic institutions have responded to this need through the use of improved development methods, and rigorous methodology employing systematic checks for detecting errors in the software during its development process.

This work addresses the contribution that can be made by quality assurance by outlining the role and implementation of software quality assurance employed during software development and maintenance. The approach taken is to develop a means of controlling and managing software development through the use of quality assurance factors and criteria and their quantification. Guidelines and procedures are presented to aid quality assurance personnel and software developers in making decisions relative to factor, criteria, and quantification specification to meet the needs of a software development effort. Tools and techniques which are available are categorized according

to quality assurance concepts presented to aid in the specification of those most suited for a given software production effort.

The results of this study are a method for determining the role and the implementation of a quality assurance plan in a software development effort. This work provides guidelines for quality assurance personnel and software developers to follow in establishing their own quality assurance plan to meet the needs of their own software production requirements.

TABLE OF CONTENTS

CHAPTER		PAGE
I.	INTRODUCTION.	1
II.	FUNDAMENTAL DEFINITIONS AND CONCEPTS FOR SOFTWARE QUALITY ASSURANCE.	4
	Software Quality.	4
	Quality Control	4
	Quality Management.	5
III.	SOFTWARE QUALITY ASSURANCE FACTORS AND CRITERIA.	7
	Conceptualization of Software Quality Assurance Factors	7
	Relationships of Quality Assurance Factors with Life Cycle Phases.	9
	Relationships of Criteria to Quality Assurance Factors	10
	Relationships Between Quality Assurance Factors	12
IV.	SOFTWARE QUALITY METRICS.	15
	The Need for Quantification	15
	Applications of Software Quality Metrics	16
	Measuring Program Complexity.	16

CHAPTER PAGE

V. QUALITY ASSURANCE TOOLS AND TECHNIQUES. 27

Tool and Technique Taxonomy 27

Assessment of Tool and Technique Effectiveness 29

VI. THE SOFTWARE DEVELOPMENT PROCESS. 31

Preliminary Design Phase. 34

System Requirements 35

Software Requirements 35

Preliminary Program Design. 36

Preliminary Design Review 37

Detailed Design Phase 39

Analysis. 39

Detailed Program Design 40

Critical Design Review. 41

Implementation and Operation. 42

Translating Design Into Code. 43

Operation 44

Transition Support. 44

Operator Training 45

Operational Use 46

Maintenance 47

The Testing Process 48

Program Unit Testing. 49

Module Integration Testing. 50

CHAPTER PAGE

VI. (continued)

Verification Testing. 52

Acceptance Testing. 53

Software Acceptance Review. 54

VII. DEVELOPING A QUALITY ASSURANCE PROGRAM. 56

Guidelines and Procedures 58

Quality Factor Specification. 64

Quality Quantification Specification. 65

LIST OF REFERENCES. 66

APPENDIX. 69

VITA. 108

LIST OF TABLES

TABLE		PAGE
1.	Quality Assurance Factor Specification and Measurement in the Software Life Cycle	11
2.	Software Quality Factor Definitions.	70
3.	Quality Assurance Criteria Definitions and Their Related Factors.	72
4.	Software Quality Metric Definitions.	75
5.	Software Quality Assurance Tools and Techniques	94
6.	Software Quality Assurance Technique Descriptions	95
7.	Software Quality Assurance Tool Descriptions	97
8.	Technique Effectiveness in Assessing Quality Factors.	100
9.	Tool Effectiveness in Assessing Quality Factors.	103
10.	Fundamental System Characteristics and Related Critical Quality Factors	107

CHAPTER I

INTRODUCTION

Computer software today represents a significant achievement in a highly technological world. Society entrusts a large proportion of its most important functions to the automatic processing and control by a computer and its associated hardware and software. Yet with all of society's dependence on computer software, the reliability of the software in many systems leaves much to be desired. Excessive downtime resulting from software errors is tolerated in many cases because no one seems to be able to understand the software design and to correct the situation. Unreliable, unmaintainable, error-prone software will continue to plague the industry if software continues to be developed using a "code and debug" method of development.

Many areas of industry spend a tremendous number of man-hours and dollars attempting to maintain software which was developed using this "code and debug" development process. With the increase of software costs within a system development, greater attention is being focused on the quality of software being produced as a means of reducing costs. An integral part of software development is the implementation of a well-defined Software Quality

Assurance (QA) Program. Quality must be engineered into software development; it cannot be added at the end. The correctness, reliability, efficiency, usability, and maintainability, among other factors, are directly related to the degree of quality implemented during the software development.

The basic problem in many areas of the software industry is that the industry is neither organized nor equipped properly to address QA requirements. Many companies lack sufficient software-related policy. In order to be competitive in the source selection, customer influence has created a need for companies to plan, organize, and demonstrate integrated hardware and software systems through the implementation of a well-defined QA program.

This work outlines the role and implementation of quality assurance to be employed in software development and maintenance. This quality assurance program may be only part of an overall industrial quality improvement program. The following topics relative to software QA and its implementation are addressed:

1. fundamental QA concepts and definitions relative to software development,
2. software QA factors and criteria and their relationships,

3. means of quantifying QA criteria and factors through software quality metrics,
4. tools and techniques available for measuring software quality,
5. QA applications in the software development process, and
6. guidelines and procedures for developing a QA program.

The intent of this work is to provide software developers and QA managers with insight into the role and applications of software QA in the software life cycle. It is also hoped that this work will provide guidelines for QA personnel to follow in developing their own QA plan to meet the needs of their specific software development requirements.

CHAPTER II

FUNDAMENTAL DEFINITIONS AND CONCEPTS FOR SOFTWARE QUALITY ASSURANCE

Software Quality

Webster defines quality as "that which makes something what it is; the degree of excellence which a thing possesses." By applying this definition to software, the quality of software would be the collection of all attributes which describe its degree of excellence. These attributes might include user-related factors, cost-related factors, and correlation characteristics to other software products.

One single factor or attribute cannot convey the definition of software quality. Depending upon the application of and environment related to software development, software quality is a composite of several factors. Quality assurance encompasses the identification of the important factors in a given environment, contract specification of those factors, and the measurement of the degree of factor presence during the software life cycle.

Quality Control

The foundation of software quality assurance is quality control. Quality control includes meeting

standards, delivery of the product on schedule, staying within budget, and fulfilling specifications. Quality control is the process through which these standards are established and met. In general, any type of control is established using the following steps [1]:[1]

1. Choose what is to be regulated.
2. Choose the unit of measure.
3. Specify the standard value or quality characteristic.
4. Create a device to measure the characteristic in the specified unit of measure.
5. Conduct the characteristic measurement.
6. Interpret any differences between the standard and the actual measurement.
7. Reconcile any variation which is detected.

This set of steps describes the general procedure by which any process is controlled. Software QA will involve the same regulatory process in the control of software development and maintenance.

Quality Management

Software QA management includes the procedures and disciplines to be used in the development, management, and

---

[1]Numbers in brackets refer to similarly numbered references in the List of References.

maintenance of a software product. Quality assurance provides the framework within which software development and maintenance operates. This framework includes the laws to be obeyed for the duration of the software life cycle. The function of QA management is to monitor and enforce compliance to the policies, regulations, guidelines, and standards which constitute this framework [2].

The majority of the time spent in software development comes during the design phase. For this reason, the development of a QA plan should concentrate on the techniques and methods for the early detection and elimination of design errors.

The management process which is used to maintain a program after completion of design is normally referred to as "software configuration management." This process allows management to control the manufacture and maintenance of a software product by dictating control of modifications to the final software design. After configuration control takes effect, all design, code, and documentation changes must be accompanied by management-approved forms indicating the reason for the change, the change being made, and verified test results of the effected change. This type of software change control is continued throughout the duration of the software maintenance cycle.

CHAPTER III

SOFTWARE QUALITY ASSURANCE FACTORS AND CRITERIA

In order to implement quality assurance in a software production environment, factors and criteria must be developed for the defining of and the adherence to quality standards. Software quality factors should be identified and further refined into measurable criteria to be applied in the software production effort.

Conceptualization of Software Quality Assurance Factors

The interaction of support groups with an operational software system can be pictured involving three distinct activities--product operation, product revision, and product transition. Figure 1 provides a conceptual scheme with these three activities and some questions and quality factors [1].

In the past, most software production efforts have emphasized the initial product operation. Correctness and reliability were the factors stressed in specifications and testing. Flexibility and maintainability of a software product have come to be important factors as a result of the growing recognition of the long-term implications in

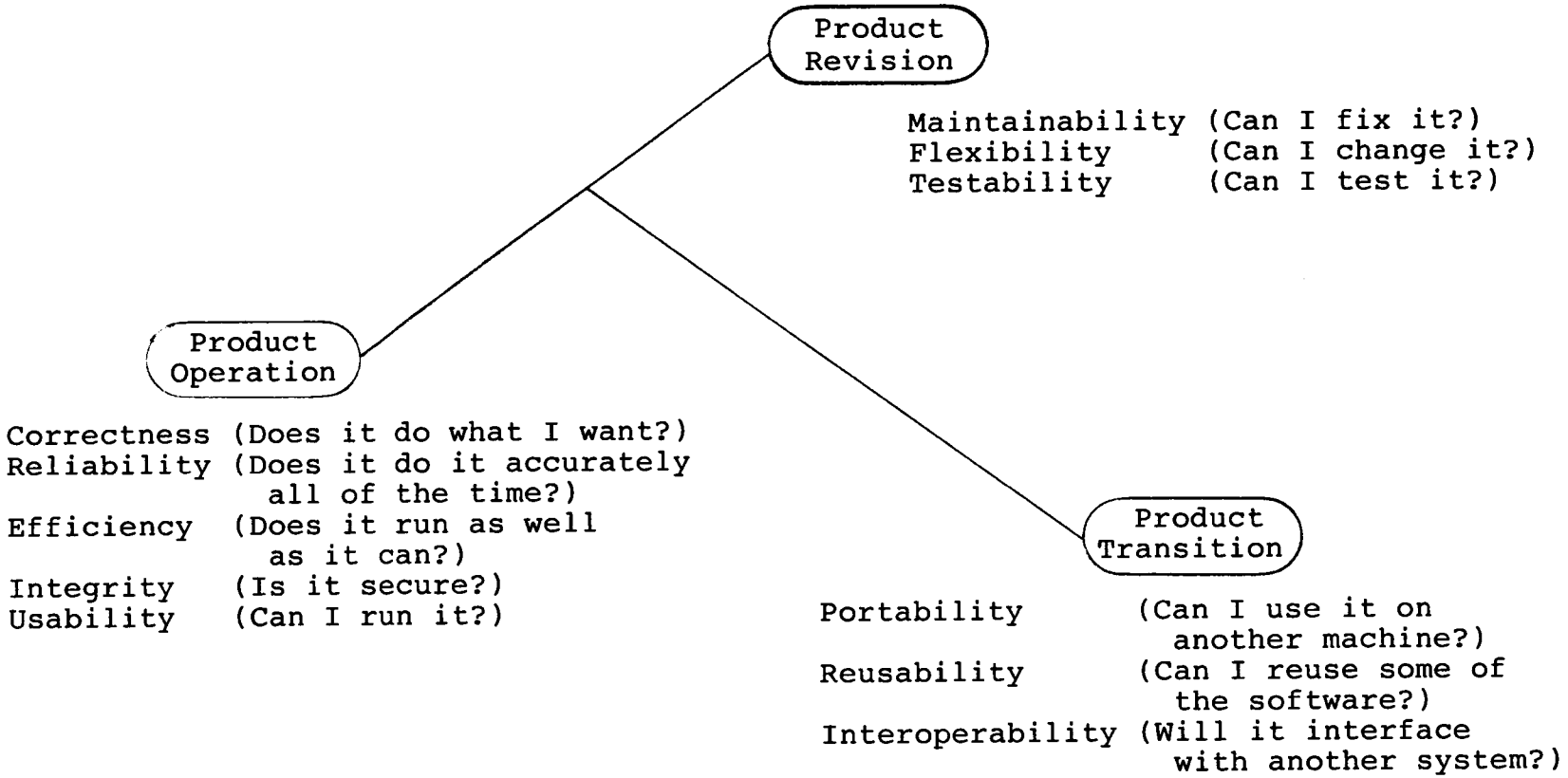


Figure 1. Software Quality Factor Allocations to Product Activity.

software development. Life cycle cost analyses have shown that maintenance and redesign too often far exceed the cost of initial development. Studies have shown that the cost of correcting errors in an operational system can be as high as 30 times greater than if the errors were detected during the system design and testing [3]. A prime objective of software QA is to place required emphasis on such factors as maintainability, flexibility, and portability during system design and development. Quality factors must be defined and through the development and implementation of a QA program, selected factors are chosen to be assured which will meet the needs of the software development effort. The most important quality factor to be considered in any software development project is reliability. If software does not function reliably, then the software and the effort spent to develop it are of far less value to anyone. Table 2, Appendix, provides formal descriptions of software quality factors which should be considered for inclusion in a QA plan.

Relationships of Quality Assurance Factors with Life Cycle Phases

Timing of quality factor measurement can be of critical importance in the software life cycle. All factors cannot and should not be measured during all phases

of the software life cycle. Table 1 shows the relationship between some of the factors and the system life cycle [1]. The table identifies where the factors should be measured relative to the system life cycle and where the greatest impact due to poor quality can be expected. From the standpoint of software QA, modifications and enhancements to software made during the maintenance cycle can be thought of as starting a new life cycle process. When substantial modifications are made, desired factors should be specified and measured during the requirements, analysis, design, and coding phases of the software change.

Relationships of Criteria to Quality Assurance Factors

Defining quality criteria is critical to understanding and communicating what is meant by the quality factors. Criteria which affect more than one factor will help describe the relationships between factors. Criteria will also aid in establishing a one-to-one relationship between metrics and quality factors. Table 3, Appendix, provides definitions of quality criteria and their relationships to the QA factors which have been presented [1]. These criteria are derived by examining the definitions of each factor and expanding the factor into independent attributes to which objective measures can be associated.

Table 1. Quality Assurance Factor Specification and Measurement in the Software Life Cycle

Factor	Life Cycle Phases						
	Requirements	Design	Code	Test	Operation	Maintenance	Transition
Reliability	o	o	o	x	x	x	
Efficiency		o	o		x		
Usability	o	o		x	x	x	x
Correctness	o	o	o	x	x		
Testability		o	o	x		x	x
Maintainability		o	o			x	x
Flexibility		o	o			x	x
Reusability		o	o				x
Integrity	o	o	o		x		
Portability		o	o				x
Interoperability		o			x		x

Legend

o Life cycle phase where factor should be measured

x Life cycle phase where greatest impact of poor quality is realized

Relationships Between Quality Assurance Factors

The specification of factors to meet the needs of a software production effort implies some relationships between the factors. Different types of software development will require different sets of quality factors during the development. For instance, the factors to be measured during the development of a real-time controls program will differ somewhat from those required for the development of a data reduction or analysis program.

Criteria, especially those common to more than one factor, will provide a detailed understanding of the factors and their relationships to one another. When choosing the factors to be assured, one must take into consideration those factors which have common criteria. If a criterion has a high degree of importance in a software development effort, then each factor related to that criterion should be considered for inclusion in the quality assurance plan.

It would be virtually impossible to assure all quality factors which have been mentioned in every software production effort. Certain trade-offs must be made when choosing those factors to be assured. Some of those trade-offs which must be considered are as follows [1]:

1. Integrity versus Efficiency: Time and storage are usually increased due to additional code and processing required to control software and data access.
2. Usability versus Efficiency: To ease operator tasks and provide more usable output, additional code and processing are required, hence lengthening run time and requiring additional storage.
3. Maintainability versus Efficiency: Maintenance problems are usually introduced when optimized code, incorporating intricate coding techniques and direct code, are implemented. Less efficient operation is introduced due to overhead built into a system through modular, high-level code.
4. Testability versus Efficiency: Modular, high-level code which is easily testable could be less efficient than an optimized code.
5. Portability versus Efficiency: Programs developed for efficiency through direct code and optimizing methods decreases the likelihood of software portability.

6. Flexibility versus Efficiency: Increased system overhead resulting from generality required for flexible software decreases the efficiency of execution.
7. Reusability versus Efficiency: Generality required for a reusable system increases overhead and decreases the system efficiency.
8. Interoperability versus Efficiency: Interface routines for conversion from standard protocol and standard data representations decreases the operating efficiency of the system.
9. Interoperability versus Integrity: Security problems are compounded when coupled systems share data and software. These types of systems allow for increased potential for sensitive data access whether accidental or deliberate.

One aspect of software portability which should be considered is that of software performance as a result of system hardware modifications or upgrades. If system modification or upgrades are anticipated, portability should be a QA factor consideration since those modifications may present the same challenges to software operation as transporting software from one system to another.

CHAPTER IV

SOFTWARE QUALITY METRICS

The Need for Quantification

In order to fully understand and communicate what is meant by various factors, metrics must be defined to give a quantification. If the reliability of software can be described by a single number, or set of numbers, or an equation for generating a set of numbers, the determination of its acceptability as a completed product can be drawn in much the same way other products are accepted or rejected.

The culmination of data acquired through applications of quality metrics provides a system model which will assist management in arriving at software development decisions. Whether for the determination of the reliability of complex systems, the acceptability of software, or the need for further testing, attaching quality metric numbers to a computer program will construct a success/failure model for the system. Trouble spots in software development which may have long-term implications can be detected early in the development cycle through proper interpretation of quality metric data [4].

Applications of Software Quality Metrics

The term "metric" can be defined as a measure of the extent or degree to which a product possesses and exhibits certain characteristics. Here the characteristics are those quality factors to be measured in the software life cycle. Table 4, Appendix, presents a number of metrics, methods of measurement, and their related criteria and quality factors [1]. Each metric is related to the requirements, design, or implementation phase of a project. Entries made in the table under the project phases indicate to which phase each individual metric can be applied. An "X" in a respective column indicates that the metric applies to this phase of the project. A "V" in a respective column indicates that the metric applies to this phase and that a maximum value should be forced upon the corresponding metric during the phase. As will be seen from the table, some factors lend themselves well to measurable metrics, while others are quite awkward to employ. The intent is to present the metrics in order to illustrate the methods of quantification widely used in industry.

Measuring Program Complexity

The program complexity measure is probably the most capable measure for both quantitative and objective

control of a software project. Many kinds of measures to evaluate program complexity have been developed and tested. Various applications of program complexity measure include estimating programming time and the number of software bugs, testing methodology based on complexity measure, measuring unstructuredness of program codes, software quality measurement, and evaluating programmer productivity. It is widely recognized that there is a lack of appropriate measures for evaluating software characteristics. Such measures are needed to effect a more quantitative and objective control of software development.

Various techniques for measuring program complexity from source code are available. These include Step Count, Halstead's E Measure [5], McCabe's Complexity Measure based on program control flow [6], and Weighted Statement Count. The greater need for a program complexity measure is to provide a means of controlling software testability and maintainability.

The Step Count is a measure of the steps contained in a program module. The Step Count is subdivided into the following measures:

1. STEPmax: The number of steps including comment lines.
2. STEPmid: The number of steps excluding comment lines.

3. STEPmin: The number of steps excluding comment lines and data declaration lines.

The Step Count merely provides a measure of the size of a program by indicating the number of lines of source code. Some software development efforts place a limit on the size of any one program module by limiting the number of lines of source code in the module. This would be one application of the Step Count complexity measure.

The Halstead E Measure is designed to calculate the amount of effort required to generate a program [5]. It also can be considered as indicating program complexity.

Halstead begins by defining two fundamental concepts for software development (i.e., program volume, V, and program level, L). The assumption is made that any implementation of any algorithm consists of N selections from a vocabulary of n elements. Each selection from the vocabulary n must be made on a nonrandom basis. One of the most efficient searches of an ordered list is a binary search in which the list is repetitively halved until the desired element is found. The result is that the desired number of comparisons required is equal to the binary logarithm of the number of elements in the list. An efficient process, equivalent to a binary search, therefore requires ($\log_2$ n) comparisons for selection of each element. It follows that a program is generated by making (N $\log_2$ n)

mental comparisons. Halstead defines this measure as program volume and is therefore given by the formula:

$$V = N \log_2 n ,$$

where,

$N = N_1 + N_2$,

$n = n_1 + n_2$,

N_1 = total operators ,

N_2 = total operands ,

n_1 = unique operator count ,

n_2 = unique operand count .

Halstead further assumes that the least possible number of unique operators that could ever be employed is two, where the pair would consist of a function designator and an assignment or grouping operator. On the other hand, there appears to be no limit to the number of unique operators that might actually be employed, because they are not limited by any language which permits the definition of new subprocedures, subroutines, or transfers to labeled points. Therefore, with respect to operators, Halstead defines the following program level proportionality:

$$L_1 = \frac{2}{n_1} .$$

Operands, on the other hand, exhibit no unique minimum over all possible algorithms. The program level with respect to operands is then defined by taking the ratio of unique to total operand usage, giving the second proportionality:

$$L_2 = \frac{n_2}{N_2} .$$

Program level is then defined as the product of the program levels with respect to operators and operands, giving the following equation:

$$L = L_1 L_2 = \frac{2}{n_1} \frac{n_2}{N_2} .$$

Since the volume, V, is a count of the mental comparisons, and the reciprocal of program level, or 1/L, is a measure of the average number of elementary mental discriminations required for each mental comparison, it follows that the total number of elementary mental discriminations, E, required to generate a given program, should be given by:

$$E = \frac{V}{L} .$$

Substituting the equations for V and L gives the following resultant equation for E, a measure of the amount of effort required to generate a program [5]:

$$E = \frac{n_1 n_2 (N_1+N_2) \log_2 (n_1+n_2)}{2n_2} .$$

The Weighted Statement Count views program complexity as based on the difficulty to understand each statement. The weight of a program statement is given by the formula:

$$W_i = DR_i + NS_i ,$$

where,

DR_i = number of data items referred to in the i^{th} statement ,

NS_i = number of subsequent statements to be executed after the execution of the i^{th} statement .

The Weighted Statement Count measure for a program module is then given by the summation of all the individual statement weights:

$$WSC = \sum_{i=1}^{n} W_i = \sum_{i=1}^{n} (DR_i + NS_i) ,$$

where,

n = number of program statements .

The McCabe Complexity Measure is based on the cyclomatic number for a program control flow graph [6].

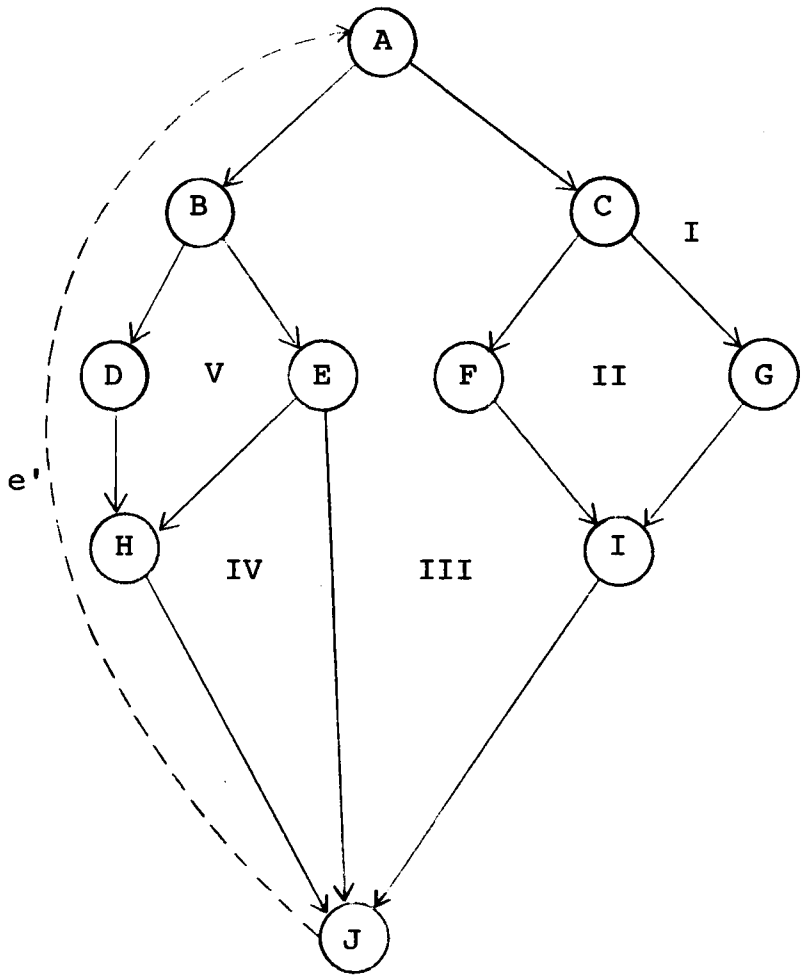
The measure indicates the number of basic paths through the program.

The McCabe Complexity Measure requires the following preliminary definition and theorem:

1. Definition: The cyclomatic number V(G) of a graph G with n vertices or nodes, e edges, and p connected components is V(G)=e-n+p.
2. Theorem: In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits.

Given a structured program, McCabe associates with it a directed graph that has unique entry and exit nodes. As shown in Figure 2, each node in the graph corresponds to a block of code in the program where the flow is sequential and the arcs correspond to branches taken in the program. This graph is classically known as the program control graph and it is assumed that each node can be reached by the entry node and each node can reach the exit node.

The above theorem can be applied to the graph in Figure 2 when the edge e' is drawn to branch exit node J back to entry node A. The control graph is now strongly connected (there is a path joining any pair of arbitrary distinct vertices). McCabe assumes that the way a program control graph is defined (unique entry and exit node, all



Legend

e = 13
n = 10
V(G) = e-n+2
V(G) = 13-10+2=5

Figure 2. Sample Complexity Measure Flow Graph.

nodes reachable from the entry, and the exit reachable from all nodes) would result in all program control graphs having only one connected component (i.e., p=1). Also, an extra edge, e', must be drawn connecting the exit node to the entry node in order to make the graph strongly connected. Based on these assumptions and applying them to the cyclomatic number definition given above, a cyclomatic complexity measure expression is defined as follows [6]:

$$V(G) = e - n + 2 ,$$

where,

e = number of edges in the original graph ,

n = number of nodes .

It should be noted that the extra edge, e', is not considered in the total number of edges in the formula. The assumption that p=1 and the extra edge, e', being drawn is represented by the constant value 2 in the formula. For this reason, the edge, e', need not be drawn when applying the formula to program control graphs.

Applying the formula to the graph in Figure 2 gives a complexity measure V(G)=5 since n=10 and e=13. This measure indicates that at least five tests must be made in order to execute each line of code.

McCabe's Complexity Measure value can also be found by counting the number of planar regions formed by the generated control graph. As indicated by the regions labeled with Roman numerals in Figure 2, five planar regions are formed by the graph, hence giving the same complexity value as generated by the formula. Again, edge e' is not considered when counting the number of regions formed.

McCabe's measure can provide a general indication of how testable, understandable, and maintainable a program module is without actually looking at the source code. A large value for program complexity measure may indicate that the module should be divided into smaller modules to make the code more testable and maintainable. In developing and illustrating his complexity measure, McCabe states that part of software development control must be the establishment of a maximum complexity measure for any program module [6]. When a program module exceeds this limit, module redesign or subdividing should be considered to preserve modularity, testability, and maintainability of the program module.

Of the program complexity measure calculations presented, McCabe's measure seems to be the most practical and applicable. It not only provides management with a

measure of modularity and testability, but it also provides software developers with a tool for effective path testing.

The Step Count measure provides little indication of program complexity other than a measure of source code length. There is not always a direct relation between program length and complexity. The translation of any given algorithm of a given complexity into source code will usually be highly dependent upon programmer style and method of implementation.

Halstead's E Measure is built on sound mathematical reasoning and concepts but it would prove to be quite difficult to employ. Too much must be known beforehand about a program (i.e., the number of operators and operands) in order to apply this measure. It would be virtually impossible to know this sort of detail about a program until the source code is completed. This method could be applied to given programming requirements to provide an approximation of the effort required where information is known about similar programming requirements.

Like the Step Count, the Weighted Statement Count relies too heavily on source code length. The weight of a given statement would not only depend upon the complexity of the statement itself but its physical location in the program. It would seem quite difficult to interpret anything meaningful from this type of weighted measure.

CHAPTER V

QUALITY ASSURANCE TOOLS AND TECHNIQUES

Available literature contains countless automated tools and techniques developed to check out design, flow charts, code, and even documentation systematically and quickly. These tools and techniques are most applicable to the detection of errors in code and simple sorts of logic errors rather than major flaws in program logic or design approach. Nevertheless, they can contribute significantly to increased productivity, earlier detection, and hence some reduction of the "ripple" effect (errors introduced as a result of correcting other errors).

Tool and Technique Taxonomy

A wide variety of tools and techniques are used for program design and validation. Two general categories of these tools and techniques can be defined:

1. execution tools and techniques by which the program is exercised using the actual hardware or simulations, and
2. analysis tools and techniques by which the behavior and structure of the program are examined without actually executing the program.

Execution tools are most familiar to development programmers. Validation-oriented automatic aids often are used to generate test inputs and interpret test results. Analysis methods, which are somewhat less familiar, range from edit programs, through which complex cross-reference listings are generated and specific code sequences are detected, to automatic flow charters, through which a detailed high-level and graphic representation of the program is constructed. The ultimate analysis tools and techniques involve correctness or correspondence proofs and program restructuring.

Quality assurance personnel employ methods of inspection, analysis, demonstration, and test. Inspection involves the verification of product or procedure compliance to stated requirements through examination. Analysis studies the product in detail to confirm an answer or a result analytically. Demonstration provides tangible evidence of compliance to stated requirements. Test provides objective assurance of compliance by making trial output acceptable for review and comparison against expected results.

Quality assurance tools and techniques can be classified based upon the method(s) which they support. Table 5, Appendix, lists several QA tools and techniques under the categories which have been mentioned and which

they support. Some overlapping is indicated in Table 5, but the taxonomy is structured to minimize any overlapping between categories. Tables 6 and 7, Appendix, provide formal descriptions of those techniques and tools, respectively, which are given in Table 5 [7].

Assessment of Tool and Technique Effectiveness

Many of the methods which have been presented can be expected to have a broad impact on design, development, and implementation. In many cases, their contribution to quality is rather straightforward, particularly for error detection tools. However, for those which promise the most sweeping reforms, essentially those dealing with management or design effectiveness, measurement is rather difficult.

Automated aids need to be implemented to support individuals making the critical decisions relative to evaluating and optimizing alternate approaches to problems. Care should be taken in choosing the techniques and tools to be employed in a QA program. Only those techniques and tools should be used which provide the highest effectiveness payoff for measuring those quality factors which have been selected for assurance during the software development. Table 8, Appendix, categorizes quality factors into low,

medium, and high effectiveness relative to each of the QA techniques which have been presented. Table 9, Appendix, provides a similar categorization of quality factors relative to QA tools. These tables will aid in the selection of tools and techniques to be employed based upon those QA factors which were selected for assurance [1].

CHAPTER VI

THE SOFTWARE DEVELOPMENT PROCESS

The software development process begins with the identification of the need for a computer software product and ends with the successful operation of the developed software in the user environment. The development process must be well defined both from the user's and developer's points of view. The required quality assurance documentation usually takes the form of a Software Development Plan which specifies the methods of development, review, and implementation to be employed in the project. This chapter describes a structured approach to the organization of the activities outlined in the Software Development Plan and required throughout the complete design, development, and installation cycles.

The simplest analysis of the software development process might yield a three-phase approach: (1) analysis of the problem, (2) coding, and (3) operational use. These three steps are common to the development and use of all computer programs, independent of the size, complexity, or application. In fact, these steps may be all that are required if the program is very small and used exclusively by the implementer. However, an implementation plan

designed around these steps cannot succeed for larger software development efforts [8].

The major problem with the three-phase approach is the lack of intermediate measurable milestones which provide checkpoints in the development process. The approach given below is an attempt to introduce such checkpoints:

1. definition of the problem,
2. analysis of the problem,
3. design of the software,
4. coding of the software,
5. testing of the software, and
6. operational use.

Each step ends with a measurable milestone; furthermore, each step of the process may require iteration with adjacent steps, but, hopefully, not with steps further back in the process.

The definition, refinement, and formalization of products and the monitoring techniques for these processes are the subject of this chapter. Figure 3 provides an overview of the software development process to be presented. The process is divided into three major phases:

1. Preliminary Design,
2. Detailed Design, and
3. Implementation and Operation.

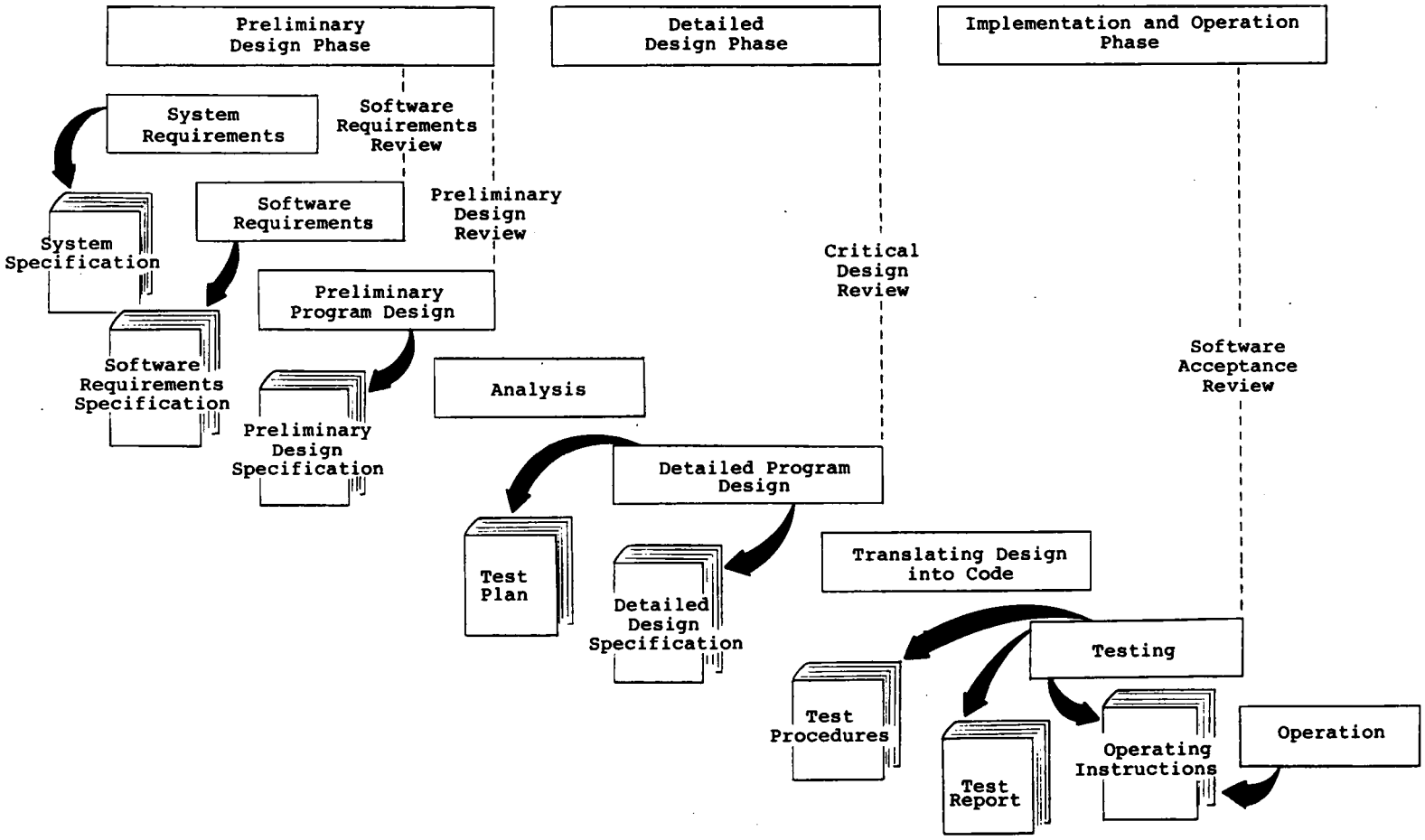


Figure 3. The Software Development Process.

Tasks performed within each phase of the software development process produce documents required to control and monitor the software design and produce coded programs, verified and delivered to the user. Reviews are used to formalize the development control process. Separate sections are dedicated in this chapter to the testing processes which are implemented during the various steps in the development cycle.

Preliminary Design Phase

The first phase of the development process is the preliminary design phase. The goals of this phase are to define what job the software must perform and generate a first cut at how the software should be structured to fulfill the job.

Defining the task the software must perform is generally labeled "requirements definition." The two basic objectives of requirements definition are to define the user requirements of the software and to derive any implied requirements for the software which might be necessary to meet the user requirements.

The second goal of the preliminary design phase is to generate an initial computer program structure, the preliminary program design, to perform the tasks. This work is performed by the software developers prior to the

detailed analysis that will be required to fully design and implement the software system. Since the designers are working from the requirements alone, the preliminary design may be in error, but the top-level system considerations (memory requirements, data storage, timing, I/O requirements, and interfaces) should be solidified [9].

System Requirements

The system requirements definition activity consists of requirements analyses and trade studies to determine the following:

1. system functions to be performed,
2. how well the functions are to be performed,
3. how the system will be structured or segmented, and
4. allocation of requirements to individual segments.

The results of these tasks are documented in the System Requirements Specifications. The customer or user is normally responsible for preparation of the System Requirements Specification [8].

Software Requirements

The functional, performance, interface, design, and verification requirements for each computer program to

be developed is normally documented in the Software Requirements Specification. The requirements must be specified to the level of detail necessary to establish the limits of design.

This activity extracts software requirements from the specifications and derives additional detailed requirements. The derivation of software requirements is a complex activity requiring the active participation and cooperation of systems and software personnel, along with other affected design and staff support personnel.

The major product of the software requirements definition task is a Software Requirements Specification for each of the defined deliverable computer programs (often called a Computer Program Configuration Item, or CPCI) making up the software system. The specification defines the functional, quantitative performance, interface, and verification requirements of each CPCI [8].

Preliminary Program Design

The purpose of the preliminary program design is to establish the functional design of the software at the computer program, or subroutine, level. It is the basis for subsequent detailed design of the computer program. It must therefore provide sufficient design information to accomplish the goals of the preliminary design review.

The Preliminary Program Design Specification (PDS) should provide the following content [10]:

1. Purpose,
2. Scope,
3. Design Overview,
4. Applicable References,
5. Computer Program Description,
6. External Interfaces,
7. Internal Interfaces,
8. Operational Concept,
9. Operational Configurations,
10. Operational Modes,
11. Operational Control,
12. Operational Timelines,
13. Data Flow,
14. Data Base Design,
15. Inputs and Outputs,
16. Limitations,
17. Method and Processing Sequence,
18. Design Conventions, and
19. Coding Conventions.

Preliminary Design Review

The preliminary design phase culminates in a Preliminary Design Review (PDR). The purpose of the PDR is

to evaluate the basic design approach prior to proceeding with the detailed design effort. This review is conducted to determine whether the design approach satisfies the requirements of the Software Requirements Specification and whether the interfaces are compatible with other interfacing systems. The review is held when design has progressed to the point where functions have been allocated to computer program modules, and when operational flows, showing the data flow between functions, have been completed.

The PDR meeting generally consists of a presentation addressing, as a minimum, the following issues [8]:

1. an overview of the design, identifying software structure, supporting design rationale, software operation in the system environment, and user interface;
2. an overview of the implementation and test plans; and
3. critical technical and contractual issues, followed by a resolution of these issues and agreement to proceed with the detailed design phase.

Detailed Design Phase

The major goal of the second phase of the software development process, the detailed design phase, is to develop and document a detailed program design to form the baseline for software implementation. The most important part of managing this phase is to enforce the detailed documentation requirements; the documentation is the design.

The basic activities of the detailed design phase are analysis and detailed program design. For scientific, highly analytical computer programs, the overlap of the analysis task with the detailed program design may be considerable; for logic-oriented programs with no critical resource problems, the analysis phase may be nonexistent.

Analysis

The analysis activity includes equation or algorithm derivation, data base content analysis, data storage access analysis, throughput analysis, and software design analysis. The equation or algorithm derivation task has traditionally been approached in the past by using the following concepts [8]:

1. the problem is defined,
2. a solution is defined (derivation of techniques or equations),

3. the solution is documented and reviewed by experts in the field,
4. the solution is tested analytically or with a simulation, and
5. the solution is released for implementation in the software.

Data base analysis involves studies of timing, access constraints, format problems, error handling, and recover design features to support the implementation of the PDR concepts during the detailed design process. The analysis activity also includes a detailed system throughput analysis based on the PDR and any additional software design analysis tasks that may be required to support the detailed program design activity.

Detailed Program Design

Detailed program design is the process of developing a complete design and operational description of each computer program, down to the subroutine level, based on the design concepts presented in the PDS and reviewed at the PDR.

The detailed software design description, presented in the form of the Detailed Design Specification (DDS), is reviewed at the Critical Design Review (CDR). The DDS establishes a detailed design from which the code will be

generated. The DDS should provide the following content [9]:

1. Purpose,
2. Scope,
3. Design Overview,
4. Applicable References,
5. Functional Description,
6. Usage,
7. Subroutines Required,
8. Inputs,
9. Outputs,
10. Processing Methods,
11. Limitations,
12. Sizing and Timing Estimates, and
13. Detailed Flow Diagrams or Structure Charts.

Critical Design Review

The detailed design phase culminates with the Critical Design Review (CDR). The CDR is a formal technical review of the detailed design, verifying design adequacy. The CDR generally consists of a presentation of at least the following topics [8]:

1. an overview of the detailed design, including detailed flow diagrams or structure charts, component interfaces, software operation in

the system environment, and detailed user interface descriptions;

2. an overview of the implementation and test plans; and
3. resolution of critical technical and contractual issues, leading to an agreement with the customer to proceed with the coding phase and to an agreed-to test program.

For a large, complex software project, the CDR is sometimes conducted on a progressive basis, with individual computer programs or subroutines reviewed and authorization granted to proceed with implementation and testing at a series of separate reviews. Care must be taken that the design of the separate modules is not dependent on the design of modules to be reviewed at a later date.

Implementation and Operation

The implementation and operation phase of the software development process includes:

1. translation of the documented, reviewed, detailed design into executable code;
2. validation of code through various levels of testing; and

3. formal verification that the results satisfy the documented requirements of the Software Requirements Specification.

The key events in this phase are the completion of the implementation and testing of the software and the formal acceptance of the software by the customer.

Translating Design Into Code

The first step in the implementation process consists of tasks involving code generation, visual inspection, and code checking. These activities are directed at preparing the code for testing.

The coding process is the conversion of the logic flows from the DDS to higher order language statements for input to the compiler. The coding process should be completed following the guidelines set forth in the adopted coding conventions. The coding activity also includes time required for correction of all compilation errors.

An important step in the software development process is to compare the compiled code with the logic flows. This simple visual inspection can save hours of computer programmer time in the verification process by locating many simple logic translation errors.

The code walk-through, sometimes called peer code reviews, is also an important concept which should be

implemented in the coding process. In addition to aiding the discovery of errors in coding, interfaces, and requirements implementation, the process also reduces management dependence on the support of individual programmers.

Operation

System operation consists of utilizing the software in the operational environment to produce the results or control the processes that were originally defined in the system specification. The operations phase typically involves four major activities:

1. transition support,
2. operator training,
3. operational use, and
4. maintenance.

Transition Support

After acceptance of the software or, in some cases, prior to acceptance, a period of parallel operation and tuning may occur. When a software system is replacing an old software or manual system, parallel operation may be required to insure that the outputs of the new system agree with the old system. During this type of operation, the capability of reverting back to the old system in the event of errors in the new system should be maintained.

During the transition period, software tuning may be required to provide optimal operations support. It is important that all software changes during this period be formally controlled. Hastily implemented changes in an operational system could prove disastrous.

Operator Training

If properly structured, the PDS, DDS, and operating instructions documents should provide the major elements required for operator training. The purpose of the operating instructions documents is to describe the operating procedures to load, start, operate, restart, and stop the computer program. Complete operating instructions include machine requirements, inputs, outputs, control card images or keyboard input formats, options, limitations, halt and recovery procedures, on-line listings, and special parameters. Complete operating instructions documents should provide the following content [10]:

1. Purpose,
2. Scope,
3. Operational Overview,
4. Applicable References,
5. System Components,
6. Hardware Description,
7. Software Operation,

8. System Configuration,
9. Equipment Setup,
10. Computer Operator Interaction,
11. The Operating System,
12. External Interfaces,
13. Operator Messages,
14. Operating Procedures,
15. Initialization,
16. Monitoring,
17. Shutdown,
18. Standby Operations,
19. Backup,
20. Recovery,
21. Analysis of Error Conditions,
22. Program Options,
23. Inputs,
24. Outputs, and
25. Limitations.

Operational Use

When the software is finally put into operational use, it is usually exercised by personnel who were not part of the development effort. For this reason, misunderstandings can surface in the early operational phase. One method of minimizing these misunderstandings is to

design the software messages with the user in mind. Messages which are meaningful and easily understood by the operator can aid tremendously in the smooth transition of the new software system into its operational environment. Whenever possible, software development personnel should provide on-site assistance during this time to answer questions, cope with unusual conditions, and clarify system usage to the operational personnel.

Maintenance

The final key activity during the operation phase is software maintenance. The major components of software maintenance are the configuration management and document control processes.

Software maintenance involves correcting errors that show up during operational use of the software. Software changes after delivery will often be made by persons other than those who developed the software; therefore, the documentation is extremely important. Clear, explicit DDS documents and tight configuration control are necessary to support continuing software maintenance activities [10].

In addition to fixing errors encountered during software usage, the software maintenance activity may include software enhancements. As with correcting errors, a key element in reducing enhancement risks is the quality

of the design documentation. Extensive software changes during the maintenance period should be thought of as beginning a new development process. Care should be taken that the specifications, design, code, review, and testing involved in the affected changes are controlled using the same guidelines as those employed during the original software development process.

The Testing Process

The testing process is performed at various phases of a software development project and consists of three basic activities [8]:

1. the generation of detailed test plans and procedures;
2. the performance of testing activities, including resolution and retest of deficient software; and
3. the documentation of test results for progress tracking.

The Test Plan describes the complete software test program based on the DDS. The Test Plan is delivered in final form at the CDR, and after the CDR, the Test Plan is under formal configuration change control.

The most influential organization in the testing process is the quality assurance organization for the

group producing the software. This organization may consist of a single person on the development team or a group of people not responsible for the development of the product and possibly from a totally different organization within the company.

Program Unit Testing

Program unit testing involves code path testing on the subroutine level. The program unit tests are completed when:

1. the programmer has completed his planned test program and is satisfied that the code matches the design,
2. all known errors have been eliminated, and
3. the code has been reviewed by a review team in a code walk-through.

A checklist to confirm that program unit testing is complete includes the following:

1. each equation has been tested using all applicable parameters with nominal values, values at the specified limits, and values outside the limits where appropriate;
2. every program branch has been executed;
3. all known coding and design errors have been corrected;

4. all successful tests have been recorded in the Test Report; and
5. all paperwork for design changes has been initiated.

Once the program unit testing is completed, the programmer submits the code to the computer program librarian to be controlled under the auspices of configuration management. The unit is then combined with related units when module integration testing begins [8].

Module Integration Testing

Module integration testing consists of building the entire software subsystem by iteratively adding program units; then, as the program units are integrated, testing to ensure that the implemented software subsystem matches the one described in the design documentation. If software is implemented using a top-down approach, module integration testing consists of integrating the top-level program units first and using the tested modules as a framework for testing each successive program unit.

Integration progresses in stages according to a schedule based on the functions to be performed by the modules, their interdependencies, their availability, and input interface data. Stubs are substituted for low-level routines to isolate the functions under test. The stubs

are replaced in stages by actual modules until the software subsystem is complete.

The purpose of module integration testing is to verify that [9]:

1. the individual program units interface with each other to form the modules described in the DDS;
2. the modules have been coded properly, so that all equations function properly and all logic paths operate as designed;
3. the modules interface with each other to form the software subsystem described in the DDS; and
4. there is an orderly progression toward the final software capabilities.

The Integration Test Plan is made up of procedures prepared jointly by software development and software quality assurance groups and are documented, reviewed, and approved. Software QA should insure that all tests are conducted in compliance with documented procedures; any deviation should be documented. The Integration Test Plan should include all steps involved in building the final software product. The expected results as the modules are added and tested should be listed, verified, and signed by the test group.

Module integration tests are conducted using the static and dynamic input data necessary to execute the intermodule and executive interfaces and to exercise operational functions. It is desirable to have access to the actual or target processor during module integration testing; at this point, the operating system should be fully capable of supporting these tests.

Verification Testing

Verification testing consists of tests, demonstrations, and analyses performed to confirm that the software satisfies all the requirements set forth in and derived from the Software Requirements Specification. Verification tests are conducted in the same test environment as module integration tests. It is desirable to have the full complements of support software, peripheral hardware, and operator devices, plus the target processor, with full operating system capabilities [9].

When verification testing begins, the computer program code and test software should be under configuration control and test procedures should be approved. All verification testing should be conducted under the surveillance of the project quality assurance group.

At the conclusion of testing, all verified requirements and test results should be analyzed and released.

The test results must verify external interfaces, and demonstrate that the computer program satisfies the specified functional and performance requirements under either simulated or actual host system operational conditions.

Acceptance Testing

Many of the verification tests are repeated during the acceptance test cycle to verify that the software performs in the operational environment as it did in the test environment. In many instances, acceptance testing is the only testing in which the software operates with the actual external interfaces, rather than with test software and hardware. As with integration testing, acceptance testing should be done under planned procedures of testing to verify specific program requirements. As the tests are completed, the results should be checked for validity, approved, and signed by the testing group.

Final documentation should be planned for delivery after successful execution of all acceptance tests so that corrections necessitated by errors detected late in the formal test program can be included. At the conclusion of acceptance testing and after the test report has been written, the Software Acceptance Review (SAR) is held.

Software Acceptance Review

The Software Acceptance Review (SAR) is a formal meeting held to audit the software product prior to user acceptance. Completion of this audit signifies that the software is ready for operation and that the development phases are complete.

The objective of the SAR is to verify that the software performance complies with the software specification performance requirements. This is accomplished by reviewing the test data accumulated during software testing. The review should include the following:

1. an audit of the test plans, procedures, and results to confirm that the test data and results comply with the plans and procedures;
2. an audit of simulations to confirm the validation of software and hardware test simulations;
3. an audit of analysis methods to confirm the results of any analyses made;
4. an audit of configuration control to confirm that the final software configuration agrees with the final documentation; and

5. an audit of the design documentation to confirm that they represent exactly the software product which was to be delivered.

The design documentation examination should be directed toward insuring adequate definition of data base characteristics, storage allocation provisions, timing and sequencing characteristics, and necessary cross-checking of flows to computer program listings [8].

CHAPTER VII

DEVELOPING A QUALITY ASSURANCE PROGRAM

Findings indicate that it is increasingly advantageous, from both product quality and cost-effectiveness standpoints, to have an explicit quality assurance activity on software development projects. The manager of this activity generally reports to the project manager and is purposely held from producing any of the deliverable products in order to provide an independent view of the product. Tasks included in this quality activity should be tailored to the project and depend upon the size and scope of the project. This approach should prove effective in insuring that the project is responsive to the quality requirements of the customer and the system application. The responsibilities of the quality assurance activity generally include [11]:

1. Planning: Preparation of a software QA plan which interprets quality program requirements and assigns task, schedules, and organizational responsibilities.
2. Policy, Practices and Procedure Development: Preparation of standards for all phases of software production, including requirements,

design, coding, and testing, tailored to specific project requirements.

3. Software QA Aids Development: Adaptation and development of manual and automated procedures for verification of compliance to software functional and performance requirements and project quality standards.
4. Test Surveillance: Reporting of software problems, analysis of error causes, and assurance of corrective action.
5. Records Retention: Retention of design and software problem reports, test cases, test data, logs verifying QA reviews, and other actions.
6. Physical Media Control: Inspection of disks, tapes, cards, and other program-retaining media for verification at all times of physical transmittal or retention, and assurance that contents are not destroyed or altered by environment or mishandling.

Most of these requirements involve considerations to quality factors and product compliance to standards and requirements specifications.

Guidelines and Procedures

Before assuming the task of developing a QA plan, key elements must be defined which should be addressed in the contractor quality assurance program. Quality assurance elements which should be considered include the following [2]:

1. initial quality planning,
2. work tasking procedures,
3. authorization procedures,
4. configuration management and control,
5. requirements standards,
6. testing procedures,
7. document control,
8. design standards,
9. coding standards,
10. documentation standards,
11. review procedures,
12. QA tools and techniques,
13. factor and criteria to be assured, and
14. quantification methods.

Contributions from all contractor organizations associated with the project are required for the implementation of an effective software QA program. Efficiency in QA applications can best be realized when a QA organization is established with the responsibility of overseeing

the QA program implementation. This element of the overall QA program must be evaluated by the software developer in light of the following questions:

1. Have the organizational elements of QA responsibility been defined?
2. Does the organization have sufficient freedom of action, authority, and responsibility to evaluate the software design and development activities?
3. Are the QA personnel qualified?

The effectiveness and success of a QA program will be directly related to the quality of the tools available as well as the qualifications of those individuals who are attempting to use the tools.

One of the first activities of a software development effort which will directly affect nearly all personnel who will be involved in the production effort are the work tasking and authorization procedures. Quality assurance procedures for issuing work tasking instructions should provide for task definitions and authorizations, task tracking and progress reporting, allocation of task resources, and methods for closing out completed tasks. These QA procedures should provide answers to the following questions:

1. Are the organizational levels of work tasking and authorization defined?
2. Can all tasks be monitored?
3. Is a Work Breakdown Structure (WBS) provided?
4. Is there a direct correlation between the WBS and task definitions?
5. Does each task have a definite completion date?
6. Is the responsible person identified for each task?
7. Have the available resources been allocated appropriately?
8. What procedures are used as followup when a task is closed out?

Once development tasks have been defined, the software QA program should require establishment of technical control and evaluation of the product as it is being developed. This requires the software developers to establish procedures for evaluating and reviewing design, code, tests, and documentation as development progresses. Other integral parts of this procedure are the definition of controls on baselined products and corrective action procedures when anomalies are detected. In the light of these aspects, the QA plan should provide answers to the following questions:

1. Have preliminary and detailed design reviews been incorporated into the development cycle?
2. Have provisions been made for corrective action items as a result of anomalies discovered during preliminary and detailed design reviews?
3. Are standards identified for all required documentation and coding?
4. Are all documents reviewed prior to release?
5. Are provisions addressed for the control of document/code changes?
6. Are provisions made for change traceability?
7. Have product testing procedures been defined?
8. Have various test levels been identified and scheduled?
9. Have test plan reviews been scheduled?
10. Are provisions made for test results monitoring, reporting, and certification?

The answers to the above questions should also provide insight into the methods to be employed (tools and techniques) during the software development. A great deal of time, effort, and dollars can be spent attempting to implement QA tools and techniques which are either unfamiliar to personnel attempting to use them or provide small return with respect to the amount of effort required

to employ them. To get the maximum return from tools and techniques in measuring quality, the following criteria should be followed:

1. Choose only those tools and techniques which are high effectiveness measures of those factors which were selected to be assured.
2. Concentrate on those tools and techniques requiring the least amount of effort with respect to the same amount of return.
3. Use tools and techniques which are either most familiar to personnel requiring them or those which require minimal training for implementation.
4. Establish standards of comparison for tool and technique output or results.

Consideration should be given to all areas which have been mentioned when developing the QA plan to be implemented during software development. The following outline is suggested as a working model illustrating the topics to be addressed in the QA plan:

I. INTRODUCTION

A. Purpose

B. Scope

II. APPLICABLE DOCUMENTS

A. Project Specifications

B. Project Plans

1. Work Plan

2. Configuration Management Plan

3. Software Development Plan

4. Coding Conventions

III. REQUIREMENTS

A. Software QA Program

1. QA Staffing

2. Authorities and Responsibilities

a. QA Representative

b. Internal Auditor

B. Software QA Program Requirements

1. Initial Quality Planning

2. Software Development

3. Configuration Management

4. Standards of Software Quality

a. Quality Factors

b. Quality Criteria

c. Quality Metrics

d. Tools and Techniques

5. Reviews and Audits

C. Corrective Action

D. Documentation

Quality Factor Specification

The explicit specification of quality factors should be considered in the initial specification for the software development. The project manager should identify and assign priorities to critical quality factors during the system requirements specification. Since there are no unique quality requirements for every system, no specific categorization of applications can be related to certain levels of quality. Each system must be evaluated against its own fundamental characteristics. Some of the fundamental system characteristics and related critical quality factors are identified in Table 10, Appendix.

Consideration must also be given as to how the quality factors selected impact the system life cycle. It should be recognized, for instance, that an extensive modification during the maintenance stage could restart the life cycle process; therefore, the appropriate factors should be specified and measured during the development of such modifications. Chapter II of this work addresses the topic of software quality factors and their relationships and impacts during the software life cycle.

Once the critical quality factors have been identified and priorities assigned, they should be included in the contractual specifications and the developer should be required to comment on how the software will be developed

to exhibit the qualities specified. Detailed explanations should be provided for each quality factor included. For example, if portability is a major concern, then as precise a description as possible should be included as to the types of environments to which the system might be transported.

Quality Quantification Specification

Quality quantification specification requires precise statements of the levels of quality required of the software. After critical quality factors are identified, specific performance levels or ratings for each factor should be specified through the selection of quality metrics. To comply with this specification, the software would have to exhibit characteristics which, when present, give an indication that the software will perform to this rating. The measurements are inserted in a mathematical relationship and a predicted rating is obtained. Only those specific metrics should be identified which will be applied to various software products to provide an indication of progress and a measurement of the quality factors which were specified. A detailed explanation of various metrics available and their associated quality factors are provided in Chapter IV of this work.

LIST OF REFERENCES

LIST OF REFERENCES

1. McCabe, Thomas J. Software Quality Assurance: A Survey. Columbia, Maryland: Thomas J. McCabe and Associates, Inc., 1980.

2. McClure, Carma L. Managing Software Development and Maintenance. New York: Litton Educational Publishing, Inc., 1981.

3. Alberts, David S. "The Economics of Software Quality Assurance," Tutorial: Software Testing and Validation Techniques. Los Alamitos, California: IEEE Computer Society Press, 1981. Pp. 415-423.

4. Metzger, Philip W. Managing a Programming Project. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1973.

5. Halstead, Maurice H. Elements of Software Science, New York: Elsevier North-Holland, Inc., 1977.

6. McCabe, Thomas J. "A Complexity Measure," IEEE Transactions on Software Engineering, SE-2(No. 4): 308-319, December, 1976.

7. Reifer, D. J. "Automated Aids for Reliable Software," Proceedings of the 1975 International Conference on Reliable Software. Los Alamitos, California: IEEE Computer Society Press, 1975. Pp. 131-142.

8. Bruce, Phillip, and Sam M. Pederson. The Software Development Project--Planning and Management. New York: Wiley-Interscience, 1982.

9. Enger, Normal L. Documentation Standards for Computer Systems. Fairfax Station, Virginia: The Technology Press, Inc., 1976.

10. Hines, V. Douglas. "Designing Software Documentation for Non-technical Users," National Bureau of Standards Federal Information Processing Standard: Software Documentation Workshop. U.S., Department of Commerce Publication No. 500-94. Washington: Government Printing Office, 1982. Pp. 225-229.

11. Bryan, William, and Christopher Chadbourne. "Source Selection and Contractor System Definition for CM, DM, and QA," Tutorial: Software Configuration Management. Los Alamitos, California: IEEE Computer Society Press, 1981. Pp. 38-41.

APPENDIX

TABLES

Table 2. Software Quality Factor Definitions

Factor	Definition
Reliability	The program will compile, load, and execute, producing answers of the requisite accuracy; and that the program will continue to operate correctly, while in operational use.
Efficiency	The extent to which a program fulfills its purpose without waste of resources, including minimum code, efficient algorithms, and conservation of memory resources in data storage.
Usability	The effort required to learn, operate, prepare input, and interpret output of a program.
Correctness	The extent to which a program satisfies its specifications and fulfills the user's objectives.
Testability	The extent to which a program facilitates the establishment of verification criteria and supports evaluation of its performance.
Maintainability	The extent to which a program facilitates updating to satisfy new requirements or to correct deficiencies.
Flexibility	The effort required to modify an operational program for the additions of required changes.
Reusability	The extent to which a program can be used in other applications related to the scope of the program functions.

Table 2 (continued)

Factor	Definition
Integrity	The extent to which access to software or data by unauthorized persons can be controlled.
Portability	The extent to which a program can be operated easily on computer configurations other than its current one.
Interoperability	The effort required to interface one software system with another.

Table 3. Quality Assurance Criteria Definitions and Their Related Factors

Criterion	Definition	Related Factor(s)
Error Tolerance	Those attributes which provide for continued operation under abnormal conditions.	Reliability
Accuracy	Those attributes which provide the required calculation, input, and output precision.	Reliability
Conciseness	Those attributes which provide for minimal code usage for function implementation.	Maintainability
Simplicity	Those attributes which provide for the implementation of required functions in the most understandable manner.	Reliability Maintainability Testability
Consistency	Those attributes which provide for the use of uniform techniques and notation.	Correctness Reliability Maintainability
Completeness	Those attributes which provide for full implementation of all user required functions.	Correctness
Traceability	Those attributes which provide for the location of all requirements as implemented.	Correctness
Modularity	Those attributes which provide for a structure of independent modules.	Maintainability Flexibility Testability Portability Reusability Interoperability
Expandability	Those attributes which provide for expansion of computational functions and data storage requirements.	Flexibility

Table 3 (continued)

Criterion	Definition	Related Factor(s)
Generality	Those attributes which provide for a broad variation in the functions performed.	Flexibility Reusability
Self-descriptive	Those attributes which provide for explanation of module functions.	Flexibility Maintainability Testability Portability Reusability
Execution Efficiency	Those attributes which provide for minimum use of processor time.	Efficiency
Storage Efficiency	Those attributes which provide for minimum storage requirements and access time.	Efficiency
Access Control	Those attributes which provide for the control of software and data access.	Integrity
Access Audit	Those attributes which provide for a software and data access audit.	Integrity
Communications Commonality	Those attributes which provide for for the use of standard interface routines and protocol.	Interoperability
Data Commonality	Those attributes which provide for the use of standard data representations.	Interoperability
Machine Independence	Those attributes which provide for determination of software dependency upon its associated hardware.	Portability Reusability

Table 3 (continued)

Criterion	Definition	Related Factor(s)
Software System Independence	Those attributes which provide for determination of software dependency upon its associated operating system and related utility programs.	Portability Reusability
Communi-cativeness	Those attributes which provide for the implementation of useful inputs and outputs which can be interpreted.	Usability
Training	Those attributes which provide for the determination of the effort required for transition from a current operation or for initial familiarization.	Usability
Operability	Those attributes which provide for the determination of operational procedures required to operate the software.	Usability

Table 4. Software Quality Metric Definitions

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
Traceability	Correctness	Cross referencing of modules to requirements: $\frac{\text{\# itemized requirements traced}}{\text{total \# requirements}}$		V	V
Completeness	Correctness	Completeness checklist:			
		1. Unambiguous function and I/O references.	X	X	X
		2. All defined functions are used.	X	X	X
		3. All referenced functions are defined.	X	X	X
		4. All conditions and processing defined for each decision point.	X	X	X
		5. All defined and referenced calling sequence parameters agree.		X	X
		6. Design agrees with requirements.		X	
		7. Code agrees with design.			X

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
Consistency	Correctness Reliability Maintainability	Procedure Consistency Checklist:			
		1. Standard design representation.		X	
		2. Standard calling sequence conventions used.		X	X
		3. Standard I/O conventions used.		X	X
		4. Standard error handling conventions used.		X	X
Error Tolerance	Reliability	Error Tolerance Control Checklist:			
		1. A definitive statement of error tolerance control.	X		
		2. Errors fixable and processing continues.		X	X
		3. Upon error detection, control passed to calling routine.		X	X
		4. Input data range values specified and checked.		X	X

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
Error Tolerance	Reliability	Error Tolerance Control Checklist (continued):			
		5. Determination of data availability before processing begins.	X		X
		6. Loop and multiple transfer index parameter ranges checked before use.	X		X
		7. Subscript checks.	X		X
		8. Critical output parameters are checked for reasonableness during processing.	X		X
		9. Recovery from hardware faults.	X		X
Design Simplicity	Reliability Maintainability Testability	Design Structure Measure:			
		1. Design organized in top-down fashion.	X		X
		2. No duplicate functions.	X		X
		3. Independence of modules:	V		V
		$1 - \frac{\text{\# modules violate rule}}{\text{total \# modules}}$			

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
Design Simplicity	Reliability Maintainability Testability	Design Structure Measure (continued):			
		4. Module descriptions include I/O processing limitations:	V		
		$1 - \frac{\text{\# modules violate rule}}{\text{total \# modules}}$			
		5. Each module has single entry, single exit:	V		V
		$1 - \frac{\text{\# modules violate rule}}{\text{total \# modules}}$			
Code Simplicity	Reliability Maintainability Testability	Measure of Code Simplicity:			
		1. Module flow top to bottom.			X
		2. Jumps in and out of loops:			V
		$1 - \frac{\text{\# single en./ex. loops}}{\text{total \# loops}}$			

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req. Des. Imp.
Code Simplicity	Reliability Maintainability Testability	Measure of Code Simplicity (continued):	
		3. Loop index modified: $1 - \frac{\text{\# loop indices modified}}{\text{Total \# loops}}$	V
		4. Module not self-modifying.	X
		5. Unique variable names.	X
		6. Number of statement labels: $1 - \frac{\text{\# labels}}{\text{\# executable statements}}$	V
		7. Single use of variables.	X
		8. Limited use of mixed mode expressions.	X
		9. Module nesting level: $\frac{1}{\text{max. nesting level}}$	V

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
Code Simplicity	Reliability Maintainability Testability	Measure of Code Simplicity (continued):			
		10. Number of branches: $1 - \frac{\#\ \text{branches}}{\#\ \text{executable statements}}$		V	
		11. Number of GO TO's: $1 - \frac{\#\ \text{GO TO's}}{\#\ \text{executable statements}}$		V	
		12. Variable density: $1 - \frac{\#\ \text{variables}}{\#\ \text{executable statements}}$		V	

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
Modularity	Maintainability Flexibility Testability Portability Reusability Interoperability	Modularity Measure:			
		1. All modules do not exceed standard module size: $1 - \frac{\#\ \text{mod.} > \text{standard}}{\text{total}\ \#\ \text{modules}}$			V
		2. All modules represent one function: $1 - \frac{\#\ \text{modules violate rule}}{\text{total}\ \#\ \text{modules}}$	V		V
		3. Controlling parameters defined by calling module: $1 - \frac{\#\ \text{modules violate rule}}{\text{total}\ \#\ \text{modules}}$	V		V
		4. Input data controlled by calling module: $1 - \frac{\#\ \text{modules violate rule}}{\text{total}\ \#\ \text{modules}}$	V		V

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
Modularity	Maintainability Flexibility Testability Portability Reusability Interoperability	Modularity Measure (continued):			
		5. Output data provided to calling module: $1 - \frac{\text{\# modules violate rule}}{\text{total \# modules}}$	V	V	
		6. Control returned to calling module: $1 - \frac{\text{\# modules violate rule}}{\text{total \# modules}}$	V	V	
		7. Modules do not share temporary storage.	X	X	
Generality	Flexibility Reusability	Extent to which a module is referenced by other modules: $\frac{\text{\# common modules}}{\text{total \# modules}}$	V	V	

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
Module Testing Support	Testability	Module Testing Measure:			
		1. Path coverage: $\frac{\text{\# paths to be tested}}{\text{total \# paths}}$	V		V
		2. Input parameter boundary tests: $\frac{\text{\# par. to be tested}}{\text{total \# parameters}}$	V		V
Integration Testing Support	Testability	Integration Testing Measure:			
		1. Module interfaces tested: $\frac{\text{\# int. to be tested}}{\text{total \# interfaces}}$	V		V
		2. Performance requirements (timing and storage) tests: $\frac{\text{\# req. to be tested}}{\text{total \# perf. req.}}$	V		V

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
Comment Effectiveness	Flexibility Maintainability Testability Portability Reusability	Comment Effectiveness Measure:			
		1. Modules have standard formatted comments which describe:			X
		Module Name/Version #			
		Author			
		Date			
		Purpose			
		Inputs			
		Outputs			
		Function			
		Assumptions			
		Limitations			
		Accuracy Requirements			
		Error Recovery			
		Referencing Modules			
		Modules Referenced			
		Revision Information			

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
Comment Effectiveness	Flexibility Maintainability Testability Portability Reusability	Comment Effectiveness Measure (continued):			
		2. Uniform code commenting: $1 - \frac{\text{\# modules violate rule}}{\text{total \# modules}}$		V	
		3. Transfer of control and destinations commented: $1 - \frac{\text{\# modules violate rule}}{\text{total \# modules}}$		V	
		4. Machine dependent code commented: $1 - \frac{\text{\# modules violate rule}}{\text{total \# modules}}$		V	
		5. Variable attributes are commented: $1 - \frac{\text{\# modules violate rule}}{\text{total \# modules}}$		V	

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
Execution Efficiency	Efficiency	Performance requirements allocated to design.	X		
Iterative Processing	Efficiency	Iterative Processing Efficiency Measure:			
		1. Invariants kept out of loop:	V		V
		$1 - \frac{\text{\# inv. stmnts. in loop}}{\text{total \# loop stmnts.}}$			
		2. Optimizing compiler used.			X
		3. Free of nonfunctional executable code:			V
		$1 - \frac{\text{\# nonf. exe. stmnts.}}{\text{total \# exe. stmnts.}}$			
		4. Decision statements are efficiently coded:			V
		$1 - \frac{\text{\# ineff. dec. stmnts.}}{\text{total \# dec. stmnts.}}$			

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
Data Usage	Efficiency	Data Usage Efficiency Measure:			
		1. Data grouped for efficient processing.		X	X
		2. Variable initialization:			V
		$\frac{\text{\# variables initialized}}{\text{\# var. req. initial.}}$			
		3. Mixed-mode expression measure:			V
		$1 - \frac{\text{\# mix-mode exp.}}{\text{\# executable stmnts.}}$			
Storage Efficiency	Efficiency	Storage Efficiency Measure:			
		1. Storage requirements allocated to design.		X	
		2. Common data defined once:			V
		$1 - \frac{\text{\# var. def. > once}}{\text{total \# variables}}$			
		3. Program segmentation:		V	V
		$1 - \frac{\text{max. segment length}}{\text{total prog. length}}$			

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
Storage Efficiency	Efficiency	Storage Efficiency Measure (continued):			
		4. Data segmentation	V		V
		$1 - \frac{\text{amt. of unused data}}{\text{total amt. of data}}$			
		5. Memory management used.			X
		6. Data packing used.			X
		7. Free of nonfunctional code:			V
		$1 - \frac{\text{\# nonfun. stmnts.}}{\text{total \# statements}}$			
		8. No duplicate code:			V
		$1 - \frac{\text{\# duplicate stmnts.}}{\text{total \# statements}}$			
		9. Optimizing compiler used.			X
		10. Free of redundant data elements:			
		$1 - \frac{\text{\# red. data elements}}{\text{\# data elements}}$			

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
Access Control	Integrity	Access Control Checklist:			
		1. User I/O access control provided (ID, password).	X	X	X
		2. Data base access control provided (privacy locks, protection codes, etc.).	X	X	X
		3. Memory protection across tasks provided.	X	X	X
		4. Provisions for recording and reporting access.	X	X	X
		5. Provisions for immediate indication of access violation.	X	X	X
Operability	Usability	Operability Checklist:			
		1. All operation steps are described.	X	X	X
		2. Descriptions of error conditions and responses.	X	X	X
		3. Provisions for operator interrupts.	X	X	X
		4. Operator messages are consistent and responses standard.		X	X

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
Training	Usability	Training Checklist:			
		1. Training materials developed for operators, end users, and maintainers.			X
		2. Realistic simulated exercises provided.			X
		3. Help and diagnostic information available on-line.			X
User I/O Interfaces	Usability	User I/O Interface Checklist:			
		1. Default values defined.			X
		2. Input formats uniform.			X
		3. Input records self-descriptive.			X
		4. Input can be verified by user prior to execution.		X	X
		5. Provision for specifying input media.	X	X	X
		6. Selective output controls.	X	X	X
		7. Output contains descriptive user-oriented labels.		X	X

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
User I/O Interfaces	Usability	User I/O Interface Checklist (continued):			
		8. Uniform output formats.		X	X
		9. Output data logically grouped for user examination.		X	X
		10. Provision for specifying output media.	X	X	X
Software System Independence	Portability Reusability	Software System Independence Measure:			
		1. Dependence on software system utility programs: $1 - \frac{\text{\# mods. req. utl. pgm.}}{\text{total \# modules}}$		V	V
		2. Dependence on software system libraries: $1 - \frac{\text{\# lib. routines used}}{\text{total \# modules}}$		V	V
		3. Standard language usage: $1 - \frac{\text{\# mods. violate rule}}{\text{total \# modules}}$			V

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
Software System Independence	Portability Reusability	Software System Independence Measure (continued):			
		4. Freedom of operating system references: $1 - \frac{\#\ \text{mods. w/OS ref.}}{\text{total \# modules}}$			V
Machine Independence	Portability Reusability	Machine Independence Measure:			
		1. Availability of programming language on other machines.	X		X
		2. Free from machine dependent I/O references: $1 - \frac{\#\ \text{mods. with I/O ref.}}{\text{total \# modules}}$			V
		3. Code is independent of word and character size: $1 - \frac{\#\ \text{modules violate rule}}{\text{total \# modules}}$			V
		4. Machine independence of data representations: $1 - \frac{\#\ \text{modules violate rule}}{\text{total \# modules}}$			V

Table 4 (continued)

Criterion	Factor(s)	Metric	Project Phase Req.	Des.	Imp.
Commonality	Interoperability	Commonality Checklist:			
		1. Definitive statement of the requirement for communication with other systems.	X		
		2. Protocol standards used.		X	X
		3. Single module interface for input.		X	X
		4. Single module interface for output.		X	X

Table 5. Software Quality Assurance Tools and Techniques

Class	Techniques	Tools
Inspection	Auditing Code Inspection Design Inspection Reviewing	Consistency Checker Editor Requirements Tracer Standards Analyzer
Analysis	Correctness Proof Error-prone Analysis Execution Analysis Simulation Standardization	Accuracy Study Processor Comparator Consistency Checker Cross-Referencers Decision Tables Editor Flow Charter Interface Checker Interrupt Analyzer Logic Analyzer Simulators Structure Analyzer Timing Analyzer
Demonstration	Functional Testing Walk-throughs	Dynamic Simulator Hardware Monitor Software Monitor Standards Analyzer Test Bed
Test	Correctness Proofs Functional Testing Logical Testing Path Testing Stress Testing Symbolic Execution	Comparator Debugger Flow Charter Simulators Instruction Trace Software Monitor Hardware Monitor Test Drivers Test Result Processor

Table 6. Software Quality Assurance Technique Descriptions

Technique	Description
Auditing	A process to examine and verify through inspection either program development status or the adherence to established procedures and requirements.
Code Inspection	A process through which visual inspections of code are accomplished to insure design compliance and to identify any coding errors.
Design Inspection	A process through which visual inspections of designs are accomplished to insure design compliance to requirements and to identify any design errors.
Reviewing	A process to examine and verify through inspection either final design or code status and its documentation along with adherence to required specifications.
Correctness Proof	Applications of axioms and theorems derived to establish the validity of the program with respect to a precise specification of its purpose.
Error-prone Analysis	A technique employed during coding to identify areas of the program that have required abnormally frequent corrections/changes.
Execution Analysis	A technique employed during test to investigate program behavior errors and to identify areas in the code that were either untested or not fully tested.
Simulation	The process of studying specific system characteristics by use of models exercised with a variety of conditions for the purpose of evaluating alternatives, timing, system capacities, performance, and constraints within the system.
Standardization	A process used to create a model against which a product or procedure can be compared in order to determine its quality.

Table 6 (continued)

Technique	Description
Functional Testing	A process used to demonstrate that the software performs satisfactorily under normal operating conditions.
Walk-throughs	A form of review in which the responsible programmer discusses his product with his peers and solicits their constructive advice.
Logical Testing	A technique used to confirm that the code performs its computations correctly (i.e., precision, accuracy, error handling, interfaces, and timing).
Path Testing	A process used to insure that a sufficient number of tests have been made on the code, exercising all paths through the code.
Stress Testing	A process used to confirm that a program performs its specifications under abnormal operating conditions.
Symbolic Execution	A technique that employs symbolic data to confirm that the software performs properly.

Table 7. Software Quality Assurance Tool Descriptions

Tool	Description
Consistency Checker	A program used to determine the consistency and completeness of program design and related data bases.
Editor	A program used to analyze source code for coding errors and scan to detect violations of specific programming practices and standards.
Requirements Tracer	A program used to provide traceability from requirements through design and implementation of the software product.
Standards Analyzer	A program used to automatically determine adherence to prescribed programming standards and practices by checking for such items as program size, commentary, structure, etc.
Accuracy Study Processor	A program which accepts mathematical equations and data as inputs and performs calculations to assist in determining if program variables are computed with required accuracy.
Comparator	A program used to compare two versions of the same computer source program to identify any differences in the two versions.
Cross-Referencer	A program which aids in assessing the impact of changes to one area of a program by providing cross-reference information to other system components.
Decision Tables	A mechanism used to represent information on program conditions, rules, and actions in tabular form that can be automatically translated to executable code by a processor.
Flow Charter	A program which shows the detail of the logical structure of a computer program based on the operations in the program.

Table 7 (continued)

Tool	Description
Interface Checker	A computer program used to check the range and limits of variables as well as source program compliance to module interfaces and verification of adherence to modularity rules.
Interrupt Analyzer	A program used to determine system actions and reactions as a result of an interrupt.
Logic Analyzer	A program which automatically reconstructs program equations and translates program flow into flow charts.
Simulator	A program which provides a system with inputs and responses that resemble the actual device being simulated.
Structure Analyzer	A program which parses source code to determine if structuring rules governing code and data manipulation are followed.
Timing Analyzer	A program which monitors execution time of a program, subroutine, or function.
Dynamic Analyzer	A program which monitors and keeps track of how thoroughly various portions of a program have been exercised.
Hardware Monitor	A hardware unit which monitors computer hardware signals to obtain data on system and/or program performance by providing data on CPU activity, channel utilization, disk I/O, terminal I/O, printer I/O, etc.
Software Monitor	A program used to monitor software performance by providing data concerning system table access, core usage, queue lengths, program timing, etc.
Debugger	Program execution aids which provide compile and execution checkout data such as program execution traces, memory dumps, and program execution breakpoint information.

Table 7 (continued)

Tool	Description
Instruction Tracer	A program used to record instances of certain classes of operations when they occur by providing event-driven data collection.
Test Drivers	Programs or input data files used to generate program data to represent the test situation or event in a program at the proper execution time.
Test Result Processor	A program used to perform test output data reduction, formatting, analysis, and printing.

Table 8. Technique Effectiveness in Assessing Quality Factors

Technique	Low Effectiveness Factors	Medium Effectiveness Factors	High Effectiveness Factors
Auditing		Efficiency Integrity Portability	Correctness Maintainability Modifiability Reliability Testability Usability
Code Inspection	Efficiency Integrity	Correctness Maintainability Modifiability Portability Testability Usability	Reliability
Design Inspection			Correctness Efficiency Integrity Maintainability Modifiability Portability Reliability Testability Usability
Reviewing		Efficiency Integrity Portability	Correctness Maintainability Modifiability Reliability Testability Usability
Correctness Proof	Efficiency Integrity Maintainability Modifiability Portability Usability	Reliability Testability	Correctness

Table 8 (continued)

Technique	Low Effectiveness Factors	Medium Effectiveness Factors	High Effectiveness Factors
Error-prone Analysis	Integrity Maintainability Modifiability Portability Usability	Correctness Reliability Testability	Efficiency
Execution Analysis	Portability Testability Usability	Integrity Maintainability Modifiability Reliability	Correctness Efficiency
Simulation		Integrity Maintainability Modifiability Reliability Testability	Correctness Efficiency Portability Usability
Standardi-zation			Correctness Efficiency Integrity Maintainability Modifiability Portability Reliability Testability Usability
Functional Testing	Usability	Maintainability Modifiability Portability Reliability	Correctness Efficiency Integrity Testability

Table 8 (continued)

Technique	Low Effectiveness Factors	Medium Effectiveness Factors	High Effectiveness Factors
Walk-throughs		Correctness Efficiency Integrity Maintainability Modifiability Portability Usability	Reliability Portability
Logical Testing	Integrity Maintainability Modifiability Usability	Correctness Efficiency Portability Testability	Reliability
Path Testing	Efficiency Integrity Portability Usability	Correctness Maintainability Modifiability	Reliability Testability
Stress Testing	Portability Usability	Correctness Integrity Maintainability Modifiability Testability	Efficiency Reliability
Symbolic Execution	Usability	Correctness Efficiency Integrity Maintainability Modifiability Portability Testability	Reliability

Table 9. Tool Effectiveness in Assessing Quality Factors

Tool	Low Effectiveness Factors	Medium Effectiveness Factors	High Effectiveness Factors
Consistency Checker	Efficiency Integrity Maintainability Modifiability Portability Usability	Testability	Correctness Usability
Editor	Efficiency Integrity Portability Usability	Correctness Maintainability Modifiability Reliability Testability	
Requirements Tracer	Efficiency Integrity Usability	Maintainability Modifiability Portability Reliability	Correctness Testability
Standards Analyzer	Efficiency	Integrity Portability Reliability Testability Usability	Correctness Maintainability Modifiability
Accuracy Study Processor	Efficiency Integrity Maintainability Modifiability Portability Testability Usability	Correctness	Reliability

Table 9 (continued)

Tool	Low Effectiveness Factors	Medium Effectiveness Factors	High Effectiveness Factors
Comparator	Correctness Efficiency Integrity Maintainability Modifiability Testability Usability	Portability Reliability	
Cross-Referencer	Efficiency Integrity Portability Testability Usability	Correctness Maintainability Modifiability Reliability	
Decision Tables	Efficiency Maintainability Modifiability Portability Usability	Correctness Integrity Reliability Testability	
Flow Charter	Efficiency Integrity	Maintainability Modifiability Portability Reliability	Correctness Testability Usability
Interface Checker	Efficiency Portability Testability Usability	Integrity Maintainability Modifiability	Correctness Reliability
Interrupt Analyzer	Efficiency Portability Testability Usability	Integrity Maintainability Modifiability	Correctness Reliability
Logic Analyzer	Integrity Maintainability Modifiability Portability Usability	Correctness Efficiency Testability	Reliability

Table 9 (continued)

Tool	Low Effectiveness Factors	Medium Effectiveness Factors	High Effectiveness Factors
Simulators		Integrity Maintainability Modifiability Portability Reliability Testability	Correctness Efficiency Usability
Structure Analyzer	Integrity	Efficiency Maintainability Modifiability Portability Reliability Testability Usability	Correctness
Timing Analyzer	Efficiency Portability Testability Usability	Integrity Maintainability Modifiability	Correctness Reliability
Dynamic Simulator	Integrity Maintainability Modifiability Portability Testability Usability	Efficiency	Correctness Reliability
Hardware Monitor	Integrity Maintainability Modifiability Portability Usability	Correctness Testability	Efficiency Reliability
Software Monitor	Integrity Maintainability Modifiability Portability Usability	Correctness Testability	Efficiency Reliability

Table 9 (continued)

Tool	Low Effectiveness Factors	Medium Effectiveness Factors	High Effectiveness Factors
Debugger	Integrity Maintainability Modifiability Portability Testability Usability	Correctness Efficiency	Reliability
Instruction Trace	Correctness Integrity Maintainability Modifiability Portability Usability	Efficiency Testability	Reliability
Test Drivers	Efficiency Integrity Maintainability Modifiability Portability Usability	Correctness	Reliability Testability
Test Result Processor	Efficiency Integrity	Correctness Maintainability Modifiability Portability Reliability	Testability Usability

Table 10. Fundamental System Characteristics and Related Critical Quality Factors

Fundamental Characteristic	Critical Quality Factors
Frequent modifications required for different applications	Flexibility Testability Reusability Maintainability
To be used by nonprofessionals	Usability Reliability Integrity
Human lives are affected	Reliability Correctness Testability
Long life cycle	Maintainability Flexibility Portability
Real-time application	Efficiency Reliability Correctness
Other agency maintains software	Maintainability Testability Usability
Software is part of a larger system	Reusability Portability Interoperability
To be used in different machines	Portability Flexibility Reusability Interoperability
On-board computer application	Efficiency Reliability Correctness
Processes classified information	Integrity
Interrelated systems	Interoperability

VITA

Roger D. Pelham was born on March 13, 1952, in Manchester, Tennessee. He attended elementary school and high school in Cannon County and was graduated from Woodbury Central High School in May, 1970. The following August he enrolled at Middle Tennessee State University. In August, 1972, he enrolled at the University of Tulsa, where he graduated in December, 1973, with a Bachelor of Science degree in Education with a major in Mathematics.

In January, 1974, he accepted a teaching position in the Tulsa Public School System, where he taught Basic Math, Advanced Algebra, and Analytic Geometry. In the fall of 1974, he accepted a teaching position in the Coffee County School System at Manchester, Tennessee, where he taught Basic Math, Algebra I, Algebra II, and Unified Geometry for five years.

In June, 1979, Mr. Pelham began work toward a Master of Science degree in Computer Science, entering The University of Tennessee Space Institute as a part-time student. In July, 1979, he accepted a position with ARO, Inc., at the Arnold Engineering Development Center, Tullahoma, Tennessee. In January, 1981, he accepted a position with Arvin/Calspan Field Services, Inc., at the

Arnold Engineering Development Center. He received the Master of Science degree with a major in Computer Science in December, 1984.