1-1-1983

A software quality assurance plan for a 'special' manufacturing environment.

David Henry Taylor

A SOFTWARE QUALITY ASSURANCE PLAN FOR A 'SPECIAL' MANUFACTURING ENVIRONMENT

by
David Henry Taylor

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of

Master of Science
in
Computer Science

Lehigh University
1983

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

May 13, 1983

Professor in Charge

Chairman of Department

TABLE OF CONTENTS

ABSTRACT

This thesis develops a software quality assurance plan for a defined environment. The environment involves a small manufacturing facility producing custom designed machinery. The machinery is controlled by real-time, process control software executing on a microprocessor based computer.

The need for a software quality assurance process in the environment is justified based on both error liability costs and software development costs. The concept of reuseable software in the defined environment and its affect on these costs is described.

Desired software attributes are defined and are ranked in importance in the given environment. The affect of various phases of the software development process on these attributes is discussed.

The software quality assurance process is defined to be an organized, systematic application of design, development, and verification approaches which build in software quality as well as test out software error. Various software design, development and verification techniques are reviewed.

A quality assurance plan for the defined environment is presented in an IEEE standard format. The plan includes specific requirements definition, design, code, and test

review procedures. It also incorporates a design philosophy based on top-down, functionally organized decomposition approaches which include information hiding concepts.

The problems of start-up implementation of the plan are discussed. The growth potential of the plan is also analyzed.

1. INTRODUCTION

Software quality assurance is the process by which computer programs are developed to ensure a level of performance which is acceptable based on reasonable engineering and managerial criteria. This thesis will examine the need for and process of software quality assurance in a small manufacturing environment.

Standard quality assurance plans as described in the literature apply to large scale (in both time and personnel) development. Based on the lack of literature on the subject, the problem of quality assurance in "small" systems development would seem to be insignificant. In the case of the environment to be examined, this is not so. The environment involves a small manufacturing facility which develops embedded computer systems to control the real-time process operation of the manufactured product. The problems solved in software by the real-time process control program are far from trivial. Likewise the impact of failure of the program may be significant in terms of public safety or economics. As a result, the programs developed need to be of "good" quality.

In developing the proposed quality assurance process, the question of what is "quality" software is examined, and current software quality assurance techniques are studied. Appropriate techniques are then organized into a software

1. INTRODUCTION

Software quality assurance is the process by which computer programs are developed to ensure a level of performance which is acceptable based on reasonable engineering and managerial criteria. This thesis will examine the need for and process of software quality assurance in a small manufacturing environment.

Standard quality assurance plans as described in the literature apply to large scale (in both time and personnel) development. Based on the lack of literature on the subject, the problem of quality assurance in "small" systems development would seem to be insignificant. In the case of the environment to be examined, this is not so. The environment involves a small manufacturing facility which develops embedded computer systems to control the real-time process operation of the manufactured product. The problems solved in software by the real-time process control program are far from trivial. Likewise the impact of failure of the program may be significant in terms of public safety or economics. As a result, the programs developed need to be of "good" quality.

In developing the proposed quality assurance process, the question of what is "quality" software is examined, and current software quality assurance techniques are studied. Appropriate techniques are then organized into a software

quality assurance plan for the given environment. The plan is presented in a format compatible with software quality assurance plan standards which are reviewed.

1.1 SCOPE

In the given environment, the current quality assurance process is of an ad hoc nature. No consistent, implemented plan for program quality exists. This thesis, then, attempts to develop a quality assurance process which will meet the objectives of productively producing functional, reliable software in the "small."

1.1.1 ENVIRONMENTAL TERMINOLOGY

In describing the environment, certain terms will be used repeatedly. These terms need to be defined in the context in which they are being used.

1.1.1.1 DEVELOPMENT IN THE "SMALL"

The concept of "small" versus "large" program development is not one that is well defined in the literature. Yourdon [1] describes five levels of programs based on size, manpower, and complexity of function. A paper by DeRemer and Kron [2] discusses "small" versus "large" in the context of the scope of program logic being designed as the program is developed. For the purpose of this thesis, "small" development refers to a program development environment in which: (a) the full range of program documentation is not normally produced; (b) program development

usually lasts less than six months; and (c) the program is usually developed by one person.

1.1.1.2 "EMBEDDED"

The term "embedded" is used in a dual context. First, it is used to describe an "E-program" type as defined by Lehman [3]. That is, an embedded program "...has become a part of the world it models...conceptually at least the program as a model contains elements that model itself, the consequences of its execution."[1] In other words, the program must recognize its own interaction with the environment as part of its logic. Secondly, embedded is used in the sense of something which is hidden. The program is not ostensibly a part of the delivered product. It is there, the customer pays for it, but it is not the primary reason for purchase of the product.

1.1.1.3 "REUSEABLE" OR "COMPONENT" SOFTWARE

The phrases "reuseable software" and "component software" seem inherently to promise something good. The idea of getting "code" for "free" to be used in a new system has

[1]Meier M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution," Proceedings of the IEEE, Sept. 1980, p-1063.

obvious value. Unfortunately in most systems, the idea has questionable practicality given current software technology. In the environment under study, however, the concept has both merit and practicality. In fact, the idea of reuseability begins to extend to that discussed by Freeman [4]. In particular, the product for which the software is developed is of a component nature. This should allow reuse of design, code and test materials. In current application the code, at least, is reused. Several "generic" programs have been developed which configure themselves to a particular application based on input parameters. In addition, the same basic machine control process code has been used on machines which were radically different in both appearance and function. The intent, then, of this thesis is to use a meaning of "reuseability" which is in between that defined by Freeman and one which implies only reuse of code.

1.1.1.4 REAL-TIME PROCESS CONTROL SOFTWARE

The type of software being developed in the environment is given the generic name "real-time process control software." This type of program is intended to be a continuously executing process, examining input data, generating output controls, monitoring response, and displaying status. The program has response and recognition time

requirements. The program as a result of its execution is controlling some external, physical process. Typical examples of this type of program usage are missile guidance systems, air traffic control systems, and nuclear power plant control systems.

1.1.2 THE ENVIRONMENT

As stated before, the environment being analyzed is a small manufacturing facility. This facility presents a specific atmosphere in which program development takes place. In addition, the programs being developed and the development process itself are relevant data to the quality assurance analysis. The facility and the program development process are described in the following paragraphs.

1.1.2.1 THE FACILITY

The facility is first and foremost a manufacturing plant. There is a hardware, production orientation to all aspects of work. The monthly production figures are "the bottom line."

The product being produced is a device in which several general purpose component functions are joined in a customized fashion. In addition to these component functions a significant percentage of the customers desire functions

which are unique to their operation. These functions are integrated into the standard operations. Also, as the market requires, new components are developed and added to the available options. The facility thus mass produces "specials." The end result from an engineering viewpoint is one of constantly designing and shipping "prototypes."

Software is developed for the device which allows real-time control and implementation of the desired functions. The software executes on a custom designed microcomputer which becomes part of the shipped product. Device control has been an evolving process from individually designed logic circuit control cards, through a 4 bit microcomputer programmed in assembly language, to a 16 bit microcomputer being programmed in a dialect of the programming language Pascal. It is this last evolution that will be examined in developing the quality assurance plan.

The engineering staff is small in number and is applications oriented. Each engineer is expected to work with 2-4 machines per month. The engineer does both hardware and software design and is responsible for the quality of both. The staff size combined with increasing production requirements has led to a need to produce more software that consumes less time for checkout. This has been the prime motivation for changing from assembly language development to Pascal development.

1.1.2.2 SOFTWARE DEVELOPMENT

The program development effort can be broken down into three categories reflecting the product produced. The first category consists of using totally pre-generated software and selecting required options via control parameters. The second involves using a majority of reuseable software and adding customer required special functions. The last category is the development of component code for a new function or product line.

The majority of programs developed fall into the second category. This category consumes 80-90 percent of the development effort.

The normal development sequence consists of the receipt of an informal functional specification which serves as a performance requirement for hardware, software, and production checkout. The program is generated by selecting an appropriate base standard package, adjusting the parameters for the functions required, designing any customer required special functions, and coding the designed functions. New component functions tend to be treated as a large size customer "special." The collected code is then translated by a program development system into machine format. The machine format data is transmitted into program memory chips. These chips are loaded into the custom microcomputer. The computer is inserted into the machine for

verification. The machine is checked out by production personnel, and any errors in machine performance due to the program are "debugged" by the engineer. A final checkout is performed, and the machine is shipped.

As stated before, the entire process was being done using assembly language programming and limited developmental aids. The development effort is now switching to Pascal with additional aids such as a syntax checking editor and a development system logic debugging package.

Typical assembly language programs' size were 4-6 thousand source lines of code (not including comment headers) with an average of 10-15 percent new code per program. Pascal programs are expected to be in the 1-3 thousand lines of source code (not including comments) and again include 10-15 percent new code.

1.1.2.3 THE ENVIRONMENTAL IMPACT ON DEVELOPMENT

In the manufacturing environment the software is not viewed as a produced product. Rather, the programs are only necessary components which are added on at the proper point in the manufacturing process.

Traditional software concepts of functional specifications, design documentation, and user's manuals are not considered useful or necessary.

What is of interest, in this environment, is productivity, functionality, and reliability. These properties reflect the desired properties of any other component of the manufactured product. The components must be available when in use at the customer facility. Also like other component parts, the customer should not need to know any operating instructions for the part.

1.2 OUTLINE

The thesis is organized according to the following general outline.

Chapter Two takes both a philosophical and practical look at software quality. What are the problems in program development, what constitutes "good" software, what is so tough about producing "good" software, and what is the impact of the specified environment on general software quality?

Chapter Three examines the software quality assurance process in a general development scheme. The reasoning behind an overall approach, the need for development standards, and the activities during various phases of a standard program "life cycle" are examined.

Chapter Four presents the specific software quality assurance plan for the given environment. The plan is presented in a standard format following a brief discussion of various format standards that have been developed.

Chapter Five analyzes part of the process of implementing the software quality assurance plan presented. This includes the need for a phased-in approach and an educational process dealing with the need for and implementation of software quality practices.

Chapter Six provides a summary of the thesis with respect to the goals outlined.

2. SOFTWARE QUALITY ASSURANCE

This chapter analyzes the concept of quality assurance for software. It examines the need for quality assurance in the environment, analyzes the concept of software design, looks at software quality "metrics," compares traditional quality assurance to software quality assurance, and discusses the special needs of the defined environment.

2.1 SOME QUESTIONS

This section examines the overall need for and basis of software quality assurance in the defined environment. It asks questions about the justification of a quality assurance plan and examines the nature of software failures.

2.1.1 WHY WORRY ABOUT QUALITY ASSURANCE IN THE DEFINED ENVIRONMENT?

The software being developed is embedded software (see Section 1.1.1.2). The implication for the product environment is that the software is the product. If the software does not work, the manufactured product is useless. If the software only partially works, the manufactured product has both short and long term performance liabilities. These liabilities are demonstrated to the manufacturer in a variety of ways.

Customer complaints are received on the performance of the product. Either the product does not perform as well as the customer expected, creates errors in the processing the customer desired, or does not perform the process the customer desired in the first place.

Poor software performance is reflected in the reliability of the product. The product may perform exactly as the customer desires except for an occasional "hiccup" that

is traceable to an occurence of a specific set of conditions. This set of conditions may occur infrequently leading the customer to question the overall reliability of the product. As another example, the product may perform well but have speed limitations which are lower than expected. These limitations may vary for various parts of the process. This again leads to inconsistent performance.

Poor software quality is reflected in the repairability of the product. Problems reported to the service organization may be difficult to trace to a root cause. Problems generated by a single software error may result in several vastly different physical symptoms causing the serviceman to think he has multiple problems. It is often difficult to isolate hardware problems from software problems thus requiring several skilled personnel to troubleshoot problems. Once found software errors that have to be fixed in the "field" can lead to even more problems. The "fix" may not be able to be tested at the factory. Thus the fix itself may have further errors. These errors may or may not be discovered when the software is changed.

Poor software may result in manufacturer product liabilities. Since the software is actually controlling a physical process, human safety may be imperiled. In addition, poor software/product performance can result in lost time, sales, or production for a customer who may then sue the manufacturer for the resultant financial losses.

Thus poor software quality in an embedded system can be expensive. It costs in marketability as customer complaints begin to be heard. It costs in potential liability suits over injuries or financial losses. It costs in maintenance costs for field repairs and updates.

Poor software quality, however, costs in a more direct fashion. Actual manufacturing costs increase. Poor software requires more time, and therefore cost, in the development process due to increased testing and rework. Errors, which must be corrected, cause delays in production while the "fix" is being designed and coded. Further delays occur as the product is now retested. This retest must be done not only for the previous error, but also to verify that the fix has not affected any other part of the software. In severe cases, hardware may have to be remanufactured to replace damaged equipment due to the failed program. Finally, these delays may result in a late shipment which often entails other financial penalties.

In summary, you worry about software quality assurance because good engineering and good management demand it. Poor quality embedded software is expensive through all phases of a product's life. To minimize these costs you need a quality assurance process.

2.1.2 WHAT IS WRONG WITH AN AD HOC APPROACH?

This question really asks what is wrong with what is currently being done in the environment under study? Do you really have to plan and work at quality assurance or can you just depend on "good" engineers and "good" managers and "good" production people?

One of the problems with this ad hoc concept is that you never know how good or how bad you are doing. This lack of knowledge prevents any possibility of management or engineering approaches to control or even monitor quality. This lack of observability and controllability, given the costs described in the previous section, is unacceptable.

In the ad hoc approach, quality becomes subjective and not objective. Concrete data is replaced with individual judgment. This lack of objectivity can lead to another problem. In cases of manufacturer liability on product safety, one issue examined is conformance to industry "standards." The ad hoc approach gives no data to prove a valid, on-going quality process.

A third problem with the ad hoc approach is that it has a bias towards failure. The ad hoc approach relies on individuals rather than on a process. This leads to several failure modes. First, people make mistakes and dependence on individuals alone means that when they fail, the product fails. Second, the required people are not

always available (they get sick, are on vacation, or quit). Third, people tend to use "intuitive" approaches to testing and, as reported by DeMillo et. al. [50] this can lead to a very poor choice of testing techniques.

Another problem is the inherent inconsistency of product delivered by the ad hoc approach. When left to "do it themselves" individuals will make different choices. All may be valid, but in an environment where reuseable software is a primary goal, inconsistencies are counter-productive.

In addition, the ad hoc approach can have problems in dealing with the complex nature of the environment. One of the reasons to use a planned and organized approach is to ensure adequate and complete coverage in testing. Ad hoc approaches tend to be incomplete and shallow.

Lastly, the ad hoc approach to software quality does not enhance future quality. Knowledge gained by individuals has no defined vehicle to be transmitted to others. Thus the same mistakes can be made over and over again. This creates recurring costs which are definitely avoidable.

2.1.3 WHAT CAN GO WRONG THAT'S SO BAD?

Given that you have to worry about software quality because of cost, and given that you need something more

than an ad hoc approach, what is the nature of the problem that needs to be solved? What things can go wrong in software development?

In their classic paper, Goodenough and Gerhart [6] classify two basic types of errors. They define "performance" errors as errors in which the software fails to produce results in the allowed time or memory space. They define "logic" errors as errors in which incorrect results are produced independent of time and space. They further break down "logic" errors into requirements, design, specification, and construction subtypes. Requirements errors occur when the system fails to satisfy a real requirement of the user. A design error occurs when the system fails to accomplish a known user requirement. A specification error exists when the written specification fails to meet the design. A construction error occurs when the program fails to meet the written specification.

From a software development standpoint, and for the purpose of discussion, these error categories will be somewhat rearranged and an additional error type recognized.

The earliest error that can occur, and usually the costliest, is a failure to understand the true requirements of the user. For a variety of reasons, it is possible to misunderstand or not know of a real customer requirement. This failure can be catastrophic. It can perpetuate itself

from the development group to the end product. When inadequate user review exists through the development phases, this error lies undiscovered until delivery. The costs at that time are potentially very large. Customer rejection, product remanufacture, and customer lawsuits to recover financial loss are among the many expensive results of these errors.

The second type of error again involves a misunderstanding. In this case the program designer does not understand the known customer requirements. As a result, the design is faulty. These errors may be found if an independent test is done by a group other than the design group. If not, the results can be very similar to those of the previous error.

Also in the design phase, the designer may develop an incorrect program design. This includes algorithms that do not give the proper response, improper handling of out-of-range input data, algorithms that require too much time or space, and program structural designs which are inadequate or difficult to understand. Some of these errors may be found during testing.

The last type of error is one in which the previous errors are not found. Inadequate testing due to lack of time, understanding, or material can lead to disastrous results at the customer site.

In addition to the above "errors," there is an additional problem that must be considered to be reflection of poor software quality assurance. The problem is one of the "schedule nightmare." In this nightmare, a program can be ninety percent complete for a very long time. Schedule dates come and go and the program still is not ready. Brooks [7] describes the problem as one of "milestones" versus "millstones." Indeed, part of the quality equation must include the availability of the program when it is needed.

All of the above types of errors will be discussed further and typical examples presented in the next chapter.

To summarize, almost anything can go wrong.

2.1.4 WHAT CAUSES THESE PROBLEMS?

Why do all these, and other problems exist? Why does software especially seem to be such a problem? There are several basic reasons which this section will explore. Among these are human frailty, the complexity of the problem being solved, the concept of "soft"ware development, and the problem of changes.

As the saying goes, "nobody's perfect." Human beings make mistakes. The problem, as Brooks [7] points out, is that computers demand perfection. The computer does not

understand what the programmer meant to type, it only knows what it actually read and performs accordingly.

Human errors can come from a wide variety of sources. Lack of training, lack of experience, lack of information, weariness, and lack of time to name a few. Errors also come from a source that may best be termed "blind spots." As noted by Yourdon [1], these blind spots are reflected by similar errors occurring repeatedly in a program. It may be as simple as confusing two similar instructions in the assembly language or always mistyping a given word in the language. These blind spots can be recognized and avoided, but they do exist.

Also related to the concept of human error is the nature of "soft"ware development. As observed by Myers [8], software development is really an information translating process. This process involves the mind analyzing data in one format (the problem to be solved) and, through a series of transformations, developing another format of information (a code program solving the problem). Software errors are the result of mistakes made in this transformation process. As pointed out by Myers, these translation errors occur for several reasons. First, there is the mind's ability to "read between the lines." We look at a problem or specification and make assumptions about what it really means. What we create is based on our interpretation of

what was requested. Second, there is a problem of lack of comprehension of complex problems. The mind has limitations on the amount of information and relationships that it can understand. The way the individual breaks down the complexities of the problem affects the resultant transformation. Thus the way a person thinks about a problem can affect the performance of the program. Another problem is one of natural forgetfulness. The mind, in the process of translating, does not always remember everything relevant to the problem solution. As a result, some portion of the transformation is not performed, and the program has a lost function. A last problem in this area is one of communication skills. Just as the mind "reads between the lines," so it often assumes the ability of others to do so correctly. Actual written or verbal communication regarding the program may be vague leading to further errors by the user of the data.

Myers [8] presents another cause for software problems as the complexity of software development itself. The amount of input data provided for a software development project is, in general, much larger than most hardware projects. Beyond the problem to be solved, there are the vast number of options in program language selection, language feature utilization, and design scheme selection and usage. To prove his point, Myers compares the

documentation required for the hardware of a computer to the documentation required for the operating system which executes on that hardware.

As a final cause, there is the problem of change. Change is inevitable, but in software projects it seems to flourish with greater vigor. Something about the idea of "it's only software" versus "we've already got the prototype built" allows change to be more acceptable in programs than in hardware.

None of these reasons, or others that could be presented, justify poor software quality. However, they do explain why problems will occur and help provide clues as to how to detect and minimize errors.

2.1.5 WHAT ABOUT THE DEFINED ENVIRONMENT?

What about the defined environment? Are there any special problems presented in embedded process control systems that can add to the problem of developing good software? Are there any special needs created by the reusing of software on new machines? The answer to both of the last two questions is "yes."

First of all, embedded control systems are faced with the problem of the environment in which they work. Time constraints exist on the speed at which they must recognize changes to the physical environment. This may require

the use of special hardware in the systems computer. This hardware allows the processing being performed to be interrupted by the external environment. The software must then respond to the change in a way which does not corrupt any non-related functions that are being performed. Likewise, the returned to functions must not override the actions performed by the interrupt. If interrupts are not available, the software must be written so that it can poll the inputs fast enough to meet the recognition criteria.

In addition to the recognition speed requirement, there is often a response time requirement. The software may have to meet a need to present a result within a given time frame from when the external change occurred. This can place restrictions on the algorithms which are developed or on the way in which those algorithms are implemented.

The software may be required to meet a "window" during which an action must be performed. Testing of certain data or implementation of a control function may only be valid during that window. Again this affects the style and nature of program design and evaluation.

Most embedded systems have a human interaction requirement. The program must present information to an operator. The operator must be allowed to enter data and control overall operation. These requirements place further speed and processing demands on the software. Data displays must

be updated at a reasonable rate. Display devices tend to require that the data be modified to a specific format prior to transmission to the device. The operator expects a reasonable recognition rate on entered data. Input switches may have to be debounced to prevent normal mechanical switch action from causing multiple responses to a single entry. Problems such as keyboard "rollover" and track-ball granularity must be analyzed and proper action taken in the software.

All of these interaction requirements reflect into the nature and style of the software. The operating executive of the program must be capable of meeting the demands placed on it by the environment. Processing routines need to be concerned about proper interaction with each other, the data base, and the outside world.

Besides these embedded systems problems, the attempt to reuse software by developing component functions, joining the required functions into a base program, modifying these functions as required, and adding new software to meet a special need presents further problems. Component functions may interact in unforseen ways unless proper design approaches are used. Coding techniques and language processors must be chosen to prevent confusion over the same name being used in two different components which are now being joined into one program. Conflicting formats or

sequence of data transfer transfers among components must be avoided by design and coding approaches which recognize these problems and prevent them.

Finally, there is the challenge of constant revision. Customers change their minds about what functions are required; budget cutting requires a modification in what is purchased; invalid information is corrected; and experience gained as development progresses requires a modification to be made. These changes can cause errors. Original design constraints are violated as the reasons for a particular structure or algorithm (which weren't written down because "everybody" knew them or there wasn't time) are forgotten. As an example, it was an obscure change to a program early in the development phase which caused the problem in launching the first orbital shuttle flight. In reporting on that "bug," Garman [9] discusses the problems of change, reuseable software and embedded systems. He provides a summary for his paper and for this section:

> The lesson from "the bug" that I plea is directed to the academic and software engineering community; help us to find ways to reliably modify software with minimum impact in time and cost. Not perfect reliability, because projects will always back off to trade for time and cost. Maintaining software systems in the field, absorbing large changes or additions in the middle of development cycles, and reconfiguring software systems to "fit"

never-quite-identical vehicles or missions are our real problems today. It's easy to say "don't break the rules." It's impossible not to without inverting the relative position of software in embedded systems - and that's wrong! Software may be the "soul" in most complex systems, but it is still just part of the supporting cast... a very flexible part.[2]

[2]John P. Garman, "The "Bug" Heard 'Round the World," Software Engineering Notes, October 1981, p-10.

2.2 WHAT IS QUALITY SOFTWARE?

Knowing that there is a need for quality software, how do you know when you have it? What are some of the parameters of "good" software? What parameters are most important? Can you measure software quality?

There are, at present, no objective measures of software quality. There are no tests to which you may submit a program which will say that it is 3.4 times better than the average. There are, however, a lot of desirable "qualities" which have been defined. A program may be subjectively judged against these qualities. Program development schemes may be used to enhance a program with respect to these qualities.

The choice of which of these qualities is most important is dependent upon the given environment. The resultant prioritized qualities can and should impact the quality assurance process.

2.2.1 "QUALITIES"

This section presents various "qualities" as they have been described in a variety of references.

2.2.1.1 "CORRECTNESS" VS. "WORKING" VS. "VALIDITY"

Correctness is the extent to which a program fulfills its intended function. The problem with this definition is knowing the intended function. Most often this quality therefore relates only to the intended function as defined by the program specification. A program is thus defined to be correct if it completely addresses its specification. Clearly a program may therefore be "correct" and still not do what is really required.

A program is defined as "working" if it meets the real operational requirements of the user. A program may be objectively tested for correctness but only subjectively tested for working if the program specification is incorrect.

The goal, obviously, is to develop programs in a fashion such that these two terms are synonymous, i.e., to ensure the specification completely and truly reflects the user needs and the program is "correct." The measure of how closely the two terms are to being identical is the "validity" of the program.

2.2.1.2 RELIABILITY

Reliability, in the general sense, is the extent to which a program can be expected to perform its intended functions without a detactable error. Again there is the

question of defining the "intended functions" and the desireability of convergence between "correctness" and "working." A somewhat more restrictive and better definition is given by Myers [8] in which he defines reliability as the probability that the program will run for a given period of time without failing weighted by the cost of a given failure to the user.

2.2.1.3 EFFICIENCY

The efficiency of a program deals with how well the program utilizes the resources needed to perform its required functions. These resources include a variety of memory types and the processing time used. Efficiency of programs can be compared since memory usage and processing time requirements are usually measurable quantities. This quality can, however, be misleading since its basis interacts with other qualities.

2.2.1.4 INTEGRITY OR SECURITY

Integrity is defined to be the extent to which access to program code or data is protected. This quality is most often related to multi-user operating systems where it is desireable to protect one user's data from access by another

user. The term may be extended to include how well protected the software is from unauthorized copying.

2.2.1.5 USEABILITY OR USER CLARITY

This quality examines the "user friendly" nature of the program human interface. How much effort is required to learn how to operate controls, input data, and interpret results? Are the input and output data in a natural language format? Are operational sequences intuitive or naturally prompted? Is "help" information readily available? Are error messages clear and do they prompt appropriate action?

2.2.1.6 MAINTAINABILITY OR SERVICEABILITY

These involve the amount of effort required to locate and fix an error in the operational program. To what extent have debugging or diagnostic aids been built into the program? This measure is in a sense the "mean time to repair" of traditional quality assurance measures. This quality is also directly related to the next quality - clarity.

2.2.1.7 CLARITY

Clarity is a measure of the effort required to understand the logic and data of the program itself. How difficult is it for another programmer to figure out what the program is doing and why? Program documentation plays a major role in determining clarity. Also involved are the type of logic and data structures used in the construction of the program, the quality and quantity of comments included in the code, naming conventions for routines and data, and consistency of format and logic approaches through the program.

2.2.1.8 TESTABILITY

Testability examines the ease with which the program may be tested. It involves whether or not all aspects of the program structure and function are both controllable and observable. It accounts for the effort required to exercise the controllability and monitor the observability.

2.2.1.9 FLEXIBILITY, EXTENSIBILITY, PORTABILITY, REUSABILITY, CONFIGURABILITY, AND GENERALITY

These concepts are closely related and deal with the ease with which the program may be changed from its present usage to another usage. Flexibility is looked upon as the

ability to modify an operational program to perform slightly different functions. Extensibility examines the ability to add to the current operations being performed in a given environment. Portability is the effort required to transfer the program from one software or hardware environment to another. Reuseability has been discussed earlier, but to reiterate, it deals with the extent to which a program may be used in other applications. Configurability reflects the ease with which multiple uses within a single program may be selected. Generality implies that the program was either written to handle the general case of the problem being solved (rather than a specific subset) or is easily extended to the general case.

2.2.1.10 INTEROPERABILITY

This factor examines the effort required to integrate one system with another. This quality would be most important in areas such as computer centers or weapons control systems where multiple hardware and software suites interact.

2.2.1.11 ROBUSTNESS, RECOVERABILITY, STRESS RESISITANCE, AND VOLUME TOLERANCE

These concepts analyze the performance of the program at and beyond its defined limits. Robustness implies that

the program is capable of receiving unacceptable or inconsistent input without detrimental results. Recoverability examines the ease with which the program recovers from hardware or software failures. Stress resistance implies the ability of the program to handle overload processing demands over a given short period of time. Volume tolerance is the ability of the program to handle maximum or near maximum loading over an extended period of time.

2.2.1.12 AVAILABILITY

As discussed earlier, the availability of the software, when it is required, must be considered as part of the quality equation.

2.2.2 QUALITIES FOR THE DEFINED ENVIRONMENT

Which of these qualities are relevant to the defined environment? Are some more important than others? Why?

Of maximum importance is that the program works and is available in time for shipment. As part of working, the program must meet defined volume processing requirements and handle customer initiated stress operation. This is because customers inevitably try to overextend the machine operation. Part of the historic marketability of the given product is its ability to handle such operation.

Reuseability and clarity are related to the production requirements. Without clarity, reuseability becomes difficult to achieve and without reuseability production needs cannot be met. Finally, maintainability and reliability are important because customer down-time and field maintenance are extremely expensive.

Efficiency is important to the extent that the program's time or space requirements impact its working and to the extent that space requirements affect costs of required memory. Useability is of concern due to potential impact on marketability. Testability is considered due to potential production time costs required to evaluate the software. Flexibility, extensibility, and configurability are of interest to the extent that they affect the overall reuseability of the software. Robustness and recoverability are of concern because of their potential impact on customer acceptance and therefore on marketability of the product.

Correctness and the associated validity are of interest only to the level that they impact any of the other important qualities. They are not given greater importance due to the nature of "small" development as defined earlier. Likewise generality is of concern only as it affects reuseability.

Integrity is of interest only in the need to protect the copyright interests of any proprietary software.

Portability and interoperability are not relevant to the defined environment.

Table 2.2.2-1 provides a summary of the evaluation of the relative importance of the various qualities in the defined environment. As indicated, a rating of "5" implies maximum importance and effort should be assigned to achieving the associated quality. A "4" implies concern for the quality, but not of an overriding nature, and an effort should be made to achieve the quality. A "3" means interest exists, and some effort may be expended to achieve the goal. A "2" implies a passing interest but only minimal effort should be used to achieve the quality. A "1" implies no concern exists and no effort should be expended. The evaluation of qualities, and the associated ratings, is based on the author's experience and judgment.

QUALITY	RATING	QUALITY	RATING
Correctness	3	Extensibility	4
Working	5	Portability	1
Validity	3	Reuseability	5
Reliability	5	Configurability	4
Efficiency	4	Generality	3
Integrity	2	Interoperability	1
Useability	4	Robustness	4
Maintainability	5	Recoverability	4
Clarity	5	Stress Resistance	5
Testability	4	Volume Tolerance	5
Flexibility	4	Availability	5

Note: 5 - Maximum Importance 1 - No Concern

TABLE 2.2.2-1: QUALITY IMPORTANCE EVALUATION

2.3 SOFTWARE QUALITY VS. TRADITIONAL QUALITY CONTROL

This section will compare the needs of software quality control to the traditional approaches of hardware quality control. In doing so, two areas will be examined. First, a historical perspective on the growth and approach of traditional quality control will be compared to a corresponding growth in software quality assurance approaches. Second, the features of hardware and software will be compared from a quality control viewpoint.

2.3.1 A HISTORICAL PERSPECTIVE

As discussed by Dunn and Ullman [10], traditional quality control has its roots in the craftsman examining his handmade product prior to sale. In software, this represents the individual programmer whose software quality is dependent solely on his own ability and standards of quality. Unfortunately this is a very poor approach. As discussed by Yourdon [1] and Mizuno [12], individual programmer abilities vary greatly.

Dunn and Ullman describe the next phase of traditional quality control as one involving the needs on mass production. As factories began to develop, quality control evolved into a final testing or checkout responsiblity. This eventually expanded to parts inspection and stastically

based testing. In software growth, this phase represents more involvement by individuals other than the programmer. Members of the design group participate in design review. Management becomes more involved in establishing standards to be followed by the group in design, coding, documentation, and testing.

The last phase discussed by Dunn and Ullman involves the establishment of a separate quality control function within the corporate structure. This group, in both hardware and software evolution, begins to reap the advantages of independent verification and validation. Independent individuals examine the entire production process from a quality assurance viewpoint. Their responsibilities are to ensure conformance to established standards and to develop new standards which will improve overall quality.

2.3.2 HARDWARD VS. SOFTWARE FROM A QUALITY CONTROL POINT OF VIEW

As discussed by Dunn and Ullman [10], there are feature differences between the nature of hardware and software which make different quality assurance approaches necessary. These differences are discussed in the following paragraphs.

With respect to failure, hardware eventually degrades and must be replaced, software can get better as errors are found and corrected. Hardware tends to give warning

indications before failure (degrading of signal strength, stress cracks, etc.), software usually gives no warning. Repair of hardware consists of restoring it to its original form (sometimes by replacement), software repair consists of creating a new and different "baseline" program.

With respect to manufacturing concerns, hardware quality testing consists of verifying that the average part on the assembly line conforms to the original design, copied software is always the same and copying software is not the normal production mode - creating new software is the problem. Hardware reliability for manufactured equipment can be established based on the component parts reliability, there are no guarantees on this being true in software. Hardware being tested off a production line can usually be examined over the total range of its intended use, in software the possible combinations of input are generally so large as to prevent this level of testing.

2.3.3 TESTING ANALYSIS

How then to test software to ensure quality? The traditional approach of statistical testing of parts does not make sense based on the differences discussed in the last section.

The historical approach of end product checkout is too expensive for a variety of reasons. Time spent correcting

errors is more costly in the end stages of production because more manpower is wasted waiting on the corrections. Errors found during testing can be more expensive since they may involve changing a basic design constraint. This can lead to analysis, design, code and debug effort which must be performed before testing can continue. Lastly, the cost of errors that slip by because the shipping date is reached and the product is shipped without the thorough testing required is staggering. Boehm [12] reports that it costs 100 times more to fix an error in the field than in the requirements phase of development.

The conclusion is that quality needs to be built in and not tested out [8, 10, 11].

2.4 SOFTWARE QUALITY AND REUSEABLE SOFTWARE

As discussed in Section 2.2, there are a variety of factors which can influence the quality reuseability. Based on the last section, the approach required to achieve reuseability (and control the factors that influence it) consists of building reuseability in. How can this be achieved?

To assure reuseability and the other qualities recognized as valuable in Section 2.2, an overall quality assurance plan must be developed. This plan will address the techniques required for use in the various stages of the software development life cycle. The stages of this life cycle and the relevant techniques are presented in the next chapter. The quality assurance plan which integrates the process is presented in the subsequent chapter.

3. THE SOFTWARE QUALITY ASSURANCE PROCESS

This chapter examines software quality assurance as it affects program development. The "standard" software life cycle is briefly presented. The influence of this life cycle on the "qualities" presented in the previous chapter is discussed. The rationale for a software quality assurance process which builds in quality rather than testing out inferiority is detailed, and the need for standards is discussed. The software life cycle is presented a second time with a discussion of the problems that can arise in each phase, the methods used to develop software which minimizes these errors, and the techniques used to verify the quality of efforts in the life cycle phase. Finally a brief section on the merits of independent verification and validation is presented.

3.1 AN OVERALL APPROACH

As discussed in the previous chapter, software quality assurance must be viewed as inherently different from hardware quality control.

> The quality solution for computer software rests on the foundation of those technological and managerial techniques and practices that support orderly, predictable, and controllable development and maintenance... One cannot assure the quality of software by adding gussets to stiffen it, or by derating its power dissipation, or by expediting deliveries with a private messenger service. The quality must be built in, and the only way to do so is to ensure that all phases of the development and maintenance are organized to that end.[3]

The problems of software development and quality measurement are compounded by the fact that, unlike other engineering disciplines, software is not derivable from the natural sciences. As a result, the software product is not realizable in the physical sense and therefore is not physically observable. This lack of direct observability forces software quality measurements to be qualitative and derivable only to the extent that the software development process is systematic [10].

[3] Robert Dunn and Richard Ullman, Quality Assurance for Computer Software, McGraw-Hill, 1982, p-81.

It appears that the particular choice of a systematic development technique, from the many available, is less important than the mere usage of any systematic technique [10,13]. In other words, software quality is improved anytime a systematic approach is followed rather than an ad hoc approach. Moreover it is obvious that it is useful within an organization to choose a single approach to ease costs of training, documentation, development, and maintenance [13]. The problem with choosing an approach to follow is that there are few complete methodologies in existance and very little objective data to choose one as best for a given environment [14]. Most of the methodologies that do exist are tied to a particular phase of the "software life cycle."

In addition to promoting a systematic development approach, the quality assurance process must incorporate procedures which recognize the "qualities" chosen as desirable and know what phases of the given development cycle potentially affect these "qualities." The process must then utilize those development and testing techniques which enhance the desired qualities.

The process, through all phases of development, must also provide measurable milestones of development quality and monitor quality performance. This means defining and implementing an evaluation process and error logging, error

analysis, and error follow-up procedures which are religiously followed. Without this error feedback loop, no permanent quality gains are possible for the organization.

3.1.1 THE SOFTWARE LIFE CYCLE

To enable further discussion of the varying techniques and to provide a framework for analysis of how software "qualities" (as defined in Section 2.2.1) are affected by software development, this section discusses the "standard" software life cycle. This life cycle, while used in most references in one form or another, is not universally accepted as valid or desirable [15, 16]. In addition, most references point out that the phases described are not totally discrete; that is, some feedback as well as look ahead occurs as development progresses through the phases and, as a result, in an actual programming project, the phases overlap. With these considerations, the life cycle consists of: (1) system requirements analysis and definition, (2) architectural and detailed design, (3) code implementation and debug, (4) testing and verification, and (5) maintenance. A brief explanation of these phases is presented in the following paragraphs. Greater detail on the activities, errors, and techniques applied during these phases is presented later in the chapter.

During system requirements analysis and definition, the "what" of the software system is determined. User needs are

analyzed and a proposed functional system is developed and documented via a system specification.

During the architectural and detailed design phase, the "How" of the software system is developed. Based on the system specification and other constraints (such as machine size and speed), architectural design decides the structure of the program that will be written. Detailed design develops the algorithms required to perform the logical functions of the system specification as assigned to the structures defined during architectural design.

Code implementation creates the software in a given programming language based on the design phase information. During debugging, the program undergoes preliminary testing by development group to remove coding errors or "bugs."

Testing and verification evaluates the developed software with respect to defined criteria including the original system specification.

Maintenance involves all follow-up activities after delivery of the software product. These include correction of residual errors and minor performance modifications as requested by the user and accepted by the developer.

3.1.2.1 CORRECTNESS, WORKING, VALIDITY

The correctness of a program is affected by the clarity and detail of the requirements specification. Vague, incomplete, or general specifications make correctness

difficult to measure or achieve. The true correctness of the program is obviously the accuracy of the mapping done during the design and coding phases. In this sense, the completeness of coverage of all functions defined by the specification, the accuracy of the algorithms used, and the accuracy of the code implementation produce the basis for correctness. The testing phase provides the visible measure of correctness as modified by the accuracy of the testing process.

Working is influenced by requirements analysis (if the program is valid) and by the design phases communications with the user. As with correctness, working is also affected by the accuracy of coding, debug, and test effort. Working is also influenced by the maintenance phase as user feedback begins to cause program changes during installation.

Validity is affected by all those areas which make a program correct and working but is most of all a reflection on the requirements phase.

3.1.2.2 RELIABILITY

Reliability is, in some cases, an inherent part of the requirements phase. User needs, the cost of particular kinds of failures, and the complexity of the system can and should lead to a specification of the minimum reliability required by the user and viable by production. Reliability is affected by the design structure approach taken, the

algorithms selected, the coding practices used, and the completeness of debug and test. The maintenance phase can provide feedback on reliability and force changes.

3.1.2.3 EFFICIENCY

Efficiency bounds can be forced by the requirements phase as decisions are often made regarding response time needs, maximum computer capacity, and other relevant parameters which are then included in the specification.

Program design affects efficiency as program structures, algorithms and design standards are selected and specified. Choices such as whether data will be global or passed as parameters and what functions will be placed in subroutines and which will be in-line code affect code and data memory size and program speed.

Program code has perhaps the largest direct impact, as the choices of a given language's constructs may cause great variations in program size or execution speed. For example, data storage in an array may be packed (thus using less memory at the expense of speed) or unpacked (thus executing faster but requiring more memory.) In addition, as code is corrected during debug or test, the style of correction can affect efficiency. Corrections often are made as "patches" which work, and correct the

symptons of failures, but which are not consistent with the intent and flow of the code causing inefficient execution.

3.1.2.4 INTEGRITY OR SECURITY

The integrity level required should be determined and specified during the requirements phase, designed in as required, ensured by code selection and debug, and verified by testing.

3.1.2.5 USEABILITY OR USER CLARITY

The requirements phase has the greatest impact on useability. The analysts must recognize the importance of user friendliness and cause appropriate requirements to be placed in the specification. Design and code phases need to implement the requirements properly. Test and maintenance phases need to provide feedback on the actual useability and cause changes to be made if necessary.

3.1.2.6 MAINTAINABILITY OR SERVICEABILITY

Maintainability can be a part of the requirements specification but is a difficult function to verify if included. Maintainability is affected more by the design strategies used and the coding practices and standards employed during development than by anything else. This

is due to the dependence of this quality on clarity. Also affecting this quality is feedback on the ways in which errors were found during testing and maintenance. This information can provide insights on enhancements that can be made in standard diagnostic routines, design approaches, and error alert and recovery procedures.

3.1.2.7 CLARITY

As with maintainability, clarity is primarily affected by the types of design strategies used (function based, data based, decomposition, synthesis) and coding practices employed (comments requirements, mnemonic conventions, data usage). This quality can be monitored and overall performance upgraded through techniques such as walkthroughs and code inspections. (These strategies, practices and techniques are described in greater detail later in this chapter.)

3.1.2.8 TESTABILITY

Testability is affected by the clarity of the requirements specification. It is also influenced by the design approaches used (levels of fragmentation of functions, input and output control, and complexity of algorithms used all affect the ease of testing). Coding practices and

language structures utilized can affect testability. In this area, several metrics, such as McCabe's numbers and Halstead measures, have been developed which attempt to relate program design and code parameters to probable required testing time [10].

3.1.2.9 FLEXIBILITY, EXTENSIBILITY, PORTABILITY, REUSEABILITY, CONFIGURABILITY, AND GENERALITY

These qualities can be affected by the requirements phase in the generality or specificity of the specification. The specification can (but usually does not) require that the program allow certain types of functional growth or require that an amount of memory or processing time be reserved for possible future expansion.

Design strategies and coding techniques tend to have a greater impact on these qualities. The designer needs to have these qualities in mind when he develops the program structure. Likewise, algorithms need to accept the general input case, functions and subfunctions need to be as uncoupled as possible. Code needs to avoid self modification, assumptions on input states, and restrictive usage of hardware. Code and design reviews can improve programs with respect to these qualities.

Maintenance processes need to use the same restraint when adding functions or correcting latent defects.

3.1.2.10 INTEROPERABILITY

Interoperability is a concern of the requirements phase. Design and code need to properly implement the specification. Test and maintenance need to verify the operation.

3.1.2.11 ROBUSTNESS, RECOVERABILITY, STRESS RESISTANCE, AND VOLUME TOLERANCE

These qualities relate to the development process in a fashion similar to flexibility and the other "growth" qualities. Robustness and the other "tolerance" qualities can be affected by the requirements specification and should be of concern to the analysts. The qualities are affected to a much greater degree by the design and code phase strategies. Again, code and design reviews are important.

3.1.2.12 AVAILABILITY

This quality is controlled by the performance of the development process with respect to the other qualities, the development environment (with respect to available tools), and the overall software management process (its realism, attitude, and performance).

3.1.3 ERROR MONITORING AND RECORDING

In addition to recognizing the impact of various phases of the development cycle on the desired qualities and choosing appropriate requirements, design, coding, testing and maintenance strategies, the quality assurance process must include a systematic error monitoring and recording process. Without this process, software development becomes an open loop control system. There needs to be feedback to cause adjustments in the techniques and standards being employed. Designers and programmers need to be made aware of errors being made, especially those of either re-occuring or catastrophic nature.

In general, this error monitoring process crosses phase boundaries as the output of one phase is used as the input to the next phase. The process of error reporting can, therefore, lead to conflicts between different groups within an organization. Thus, the process must be presented as one which is not an evaluation but rather an educational vehicle which fosters an overall good. As such, error monitoring and reporting is best not handled by management. Guarantees must, however, be made to insure correction of errors. Dunn and Ullman [10] present a discussion of some of the potential problems in this area. This area is also discussed further in Chapter 5 of this thesis.

3.2 THE NEED FOR STANDARDS

The software quality assurance process has thus far been defined to incorporate several things. First, its goal is to ensure a level of software performance which is acceptable based on engineering and managerial criteria. Second, it involves a process by which quality is systematically built into the program rather than a process in which final testing alone is used to remove errors. Third, the process must recognize those qualities which are most important and select the techniques in the phases of the development cycle which will promote those qualities. Finally, the process must incorporate a feedback loop which incorporates error reporting and correction monitoring.

In addition to these component concepts, the quality assurance process must include a final "glue." This "glue" consists of the adoption of development standards which will serve as guidelines throughout program development. These standards provide a visible symbol of the systematic development philosophy behind the process. They also provide starting points for evaluation in the error recording and correction process. In addition, they can directly affect software qualities such as clarity. Finally, as mentioned earlier, standards can ease the costs of training, documentation, development, and maintenance.

In a general sense, these standards consist of: guidelines in areas such as contents and format of requirements specifications; design methodologies to be used in creating the structure, and detailed logic content of program design and format of the appropriate design documentation; program languages to be used and allowed language structures as well as program format and comment conventions; program testing and validation techniques to be applied along with test reporting conventions; and maintenance logging and configuration management techniques to be applied after shipping. The specific selection of guidelines to be used is a function of the types of programs being developed and the development environment. The next section presents some of the methods, philosophies, and standards which have been applied in the various phases of the life cycle. The next chapter will present those standards selected for the environment under study.

3.3 THE SOFTWARE QUALITY ASSURANCE LIFE CYCLE

This section presents software quality assurance procedures in the program life cycle. The subsections present types of errors that can occur, various development methods to avoid these errors, and methods used to verify quality. Throughout typical documentation is identified.

Before describing the various quality efforts, it is useful to note that within the area of quality verification techniques, there are two subcategories which are used within the literature (and to a certain degree this thesis). These categories are static analysis and dynamic analysis [18]. Static analysis techniques are those which analyze system performance based on system documentation (requirements documents, design documents, source code) and do not require program execution. These techniques are applied throughout the development cycle. Dynamic analysis methods require execution of the program to analyze desired qualities. As such, dynamic analysis techniques can be applied only in the code and debug, testing, and maintenance phases of the development cycle. While within this thesis the specific category a technique belongs to is not always identified, the concepts presented by the categories are useful when developing an overall SQA process.

3.3.1 REQUIREMENTS ANALYSIS AND DEFINITION

As discussed earlier, this phase of the life cycle involves an analysis of user needs and development of a system functional definition. Without proper development of a requirements specification, the rest of the software quality assurance process has no foundation.

> Software design can be characterized as allocating requirements to the components of an architecture. This characterization stresses that a design consists of parts (modules) and their interconnections (interfaces) for the purpose of realizing a given set of requirements. Clearly this presupposes that the software requirements are defined and analyzed prior to the design activity. Without first satisfying this important presupposition all subsequent efforts to assure a quality product are, at best, misguided.[4]

One problem in developing a good specification in many development projects is that not enough time is put into the requirements phase and, as a result, quality suffers.

To be effective the requirements phase also requires good communication between the development group and the user group. Poor communication decreases information availability and reduces the quality of the performance

[4]John B. Goodenough and Clement L. McGowan, "Software Quality Assurance: Testing and Validation," Proceedings of the IEEE, Sept. 1980, p-1096.

specification. Even when communication is good, the user group may not be certain of its needs. Lack of user understanding coupled with development time problems leads to incomplete and changing specifications. Finally, errors in specification are often caused by lack of available system analysis tools and procedures.

3.3.1.1 COMMON ERRORS

The problems discussed above create errors that manifest themselves in a variety of ways.

Logic "holes" may exist in the specification. If the logic of the specification is drawn as a decision table, blank areas exist in the table. These "blanks" may deal with handling of input data that is out of range; program initialization or termination sequences; interaction between system functions; or system state transitions.

Beside logic holes, the specification may have errors in the functional definition itself. Functions may have been omitted, may be erroneously defined, may not be feasible with current technology, may be unnecessary, or may be inconsistent with other functions.

Another area for errors is the system's human interface. This interface may be cumbersome, totally undefined, or only partially defined.

Finally, the specification may be weak in the area of defining performance requirements or system environment. System specifications often specify only the functions to be performed and not any overall performance criteria such as total program size, program recognition and response times, or spare processing time. In addition, program environmental concerns such as the expected scope of operation are not defined.

In addition to these performance related errors, the specification format may cause problems. The way in which the information is presented may be confusing or difficult to modify.

3.3.1.2 METHODS OF DEVELOPMENT

Development techniques used to aid in the creation of better requirements specifications range from individual development tools to fully developed analysis processes which incorporate various tools into a cohesive approach which attempts to guarantee quality.

Wasserman [14] presents an excellent overview of the various process approaches which have been defined and used. Among these are: Structured Systems Analysis (SSA), which uses a combination diagrams, database elements, a design language, and decision tables and is discussed by Gane and Sasson [19] and DeMarco [20]; Structured Analysis and

Design Techniques (SADT), a diagrammatic modelling approach presented by Ross [21] and Ross and Schoman [22]; Problem Statement Language (PSL), a formal language with an automated analyzer developed by Teichroew and Hershey [23]; Software Requirements Engineering Methodology (SREM), a large systems based technique using a variety of notations and tools presented by Alford [24]; and Higher Order Software (HOS), a system of laws and a language consistent with the laws that may be applied to any design process and is discussed by Hamilton and Zeldin [25].

In addition to these techniques, some techniques usually applied to the design phase have been used in requirements definition. Included in this category are HIPO (Hierarchy-Input-Process-Output), a means of diagrammatic structured decomposition discussed by Stay [26] and the IBM report [27]; and the Warnier-Orr approach [28, 29] an output based on logical analysis process.

Finally, there are some complete life cycle development approaches which include requirements specification approaches. An example is the Software Development System (SDS), which uses an SREM based requirements phase and is described by Davis and Vick [30].

Besides the above approaches taken as packages, the tools used by them may be customized into a given

environment. These "tools" include data dictionaries, data flow diagrams, logic flow diagrams, and requirements statement languages (RSL).

As part of the overall development scheme, the beginning of the configuration management process described by Dunn and Ullman [10] should be started. This process is needed to maintain ordered, documented upgrades to the software system.

Also useful is the beginning of a form of documentation known as the project notebook as discussed by Brooks [7]. This notebook serves as a repository for relevant memos, design notes, and other project data which needs to be available to all members of the development team.

3.3.1.3 METHODS OF VERIFICATION

As Howden [31] indicates, requirements quality verification is dependent on analysis of the requirements specification. This analysis may be done as part of a formalized walkthrough of the document by the development group or as part of an independent verification of the requirements using simulation, modeling, and other mathematical analysis techniques.

If a requirement specification language such as PSL has been used, it is also possible to do some automated verification such as analyzing the specification for "completeness."

In addition to the above technical analysis of the requirements document, a final check of the specification involves user approval. Ideally this includes a review of the document by the user and a formal sign-off approval. If this is not possible, an internal quality control group should serve as a surrogate "user." After approval, the document needs to be placed under a configuration management process which allows change only as approved by the user, the specification group, the design group, the quality group, and needed other development groups depending on the state of the project.

3.3.2 ARCHITECTURAL AND DETAILED DESIGN

This portion of the development cycle develops the structure, logic, and algorithms to be used in generating the coded program. Based on the requirements specification, an information transformation occurs creating a design specification. How this transformation should proceed to develop "quality" software is the subject of many articles in the professional literature. At the heart of the controversy is the question of what is the proper basis for the development process and structure definition. Should this basis be data or function oriented? Should the designer first examine the required output and work backwards to the necessary input creating structure along the

way? Should the design start at the top with the overall functions to be performed or at the bottom with the hardware and interfaces that are to be used in the system? These concepts are presented or reviewed in a text edited by Freeman and Wasserman [32].

The question is: are there any of these philosophies which are better from a quality assurance viewpoint? As would be expected, each author claims he has the best approach. The authors reviewing the approaches are split. Goodenough and McGowan [13], for example, claim any approach which is consistent with the problem is valid. Dunn and Ullman [10] claim structure is absolutely necessary for the built-in quality and that structure should be of a layered, top-down functional nature.

Most of the philosophies are represented in design techniques. These techniques are listed as part of the development methods portion of this subsection.

3.3.2.1 COMMON ERRORS

Among categories of errors that develop in the design phase are logic errors, overload errors, timing errors, documentation errors, through-put or capacity errors, fallback or recovery errors, and standards errors.

Logic errors include: a process scheduling design which does not meet the system timing requirements;

algorithms that incorrectly compute data or are limited in range; improper handling of out-of-range input data; processes or algorithms that fail to complete; improper design of shared data controls; cumbersome/invalid/vague assignment of functions to program structures ("spaghetti logic"); overly complex formula or expressions; and poor sizing of modules.

Overload, timing, and capacity errors can be the result of logic errors or may involve improper utilization of resources.

Fallback and recovery errors include poor error condition definition, poor error alert indication, and various human interface definition errors.

Standards errors involve failure to use the selected design approach in developing a system architecture, failure to document properly, or failure to follow proper configuration management procedures.

Documentation errors would include documentation which follows standard format but may be vague, misleading, or incomplete in content.

3.3.2.2 METHODS OF DEVELOPMENT

Wasserman [14] presents a good overview of the various program design aids and approaches which have been developed and used in an attempt to design quality software. Among

these are: Structured Design, a modular design effort emphasizing single function modules and well defined data transfer between modules presented by Yourdon and Constantine [33]; HIPO, a means of diagrammatic decomposition discussed by Stay [26] and IBM documentation [27]; the Jackson Design Method (JDM), an input structure to output structure mapping based approach developed by Jackson [34]; Design Realization, Evaluation, And Modeling (DREAM), a behavioral object oriented modelling approach reported by Riddle [35]; structured flowcharts, a structured coding based diagramming method developed by Nassi and Shneiderman [36]; program design languages (PDL), a "structured" English module description tool reported by Caine and Gordon [37]; and the Warnier-Orr approach [28, 29], an output based logical analysis process with a diagrammatic description tool.

Other methods used include: developing with finite state machines, a finite state modelling approach, discussed by Salter [38]; designing with Petri nets, a directed graph based approach described by Peterson [39]; and designing using the Parnas concept, an information hiding approach to module selection and definition discussed by Parnas [40, 41].

Needless to say, the number of methods used is large and growing. Choice of an approach or combination of approaches and philosophies needs to be based on an

understanding of the strengths and weaknesses of the techniques and their applicability to the type of software being developed. Their impact on those qualities selected as important must be considered. The description of the approach to be used in the defined environment will be presented in the next chapter.

3.3.2.3 METHODS OF VERIFICATION

As with the requirements phase, verification consists of using static analysis approaches operating on the design documentation. This documentation, being a by-product of the design effort, is dependent on the design approach used. The various approaches create design artifacts in very different formats (data flow diagrams, logic flow diagrams, Petri net drawings, HIPO drawings, and PDL programs). How these are incorporated into a design specification affects the verification methods that can be used.

Techniques that have been applied include: a cross reference check between design elements and the requirements specification; verification of the interface portion of design elements to check consistency; analysis of logic paths through a top-down design; modelling and simulation based design data to verify requirements performance; verification of algorithms via simulation or comparison with independent equations; units analysis of equations;

structured walkthroughs of the design by the development group simulating execution of the design for various conditions; design inspections using standard error checklists; inductive assertion methods to verify algorithms; graph theory techniques applied to logic flow diagrams; and automated module interface checkers applied in certain PDL environments [31].

Most of the above methods are applied in a somewhat informal approach by the development group. A more formal approach is reflected in the widely practiced concepts of the Preliminary Design Review (PDR) and the Critical Design Review (CDR) [13]. The PDR consists of a formal review of the proposed design architecture. This review is performed by the management group, the users, and the quality assurance group, to verify the logic and feasibility of the design prior to proceeding to a detailed design level. The CDR is a similar formal evaluation of the detailed design for implementation and performance feasibility prior to code generation.

In addition to these techniques, there is a quality assurance approach which reflects into the design phase but is really concerned with overall program performance. This approach deals with self-testing programs. One self-testing approach method consists of including "dynamic assertions" about the properties and relationships of module input and

output. These assertions are incorporated into the design and verify proper operation of the program during execution [42,43,44]. A second approach, developed by Anderson and Kerr [45], consists of inclusion in the design and code of "recovery control blocks." These blocks evaluate a set of alternatives and return an error indicator to surrounding code [42].

The configuration management process and the project notebook begun during the requirements phase should continue through the design phase.

3.3.3 CODING

In the program development cycle, coding is the process of transformation of the design information into a format acceptable by a computer for eventual execution. In a typical software development project, this phase consumes less than 20 percent of the total effort. The process typically is assigned much greater significance due to the nature of the process and the importance of its output. The task of writing code is closely related to the design effort, and overall quality is strongly dependent on the proper interaction between these two efforts [31]. Also the style used in generating the code affects on the various software qualities (as noted earlier in this chapter).

Finally, it is the code that executes and generates the results that are visible to the world outside the development group.

The transformation of design into code includes efforts such as: selection of a programming language; development of logic and data structures within the language to support the algorithms of the design; selection of an implementation strategy (code from the top in layers with lower levels as "stubs," code individual modules and integrate them as they are developed, code individual modules and wait until all are available before integration); incorporation of a mnemonics or labelling convention; and inclusion of a comments standard [1, 14].

3.3.3.1 ERRORS

As presented earlier (see Section 2.1.3), Goodenough and Gerhart [6] provide a basic classification of error types that are evident in the generated code. Other references [1, 10, 46] provide sample lists of errors that commonly develop in the code phase. These errors are generally classified in groups which include: data reference errors, data declaration errors, computation errors, comparison errors, control flow errors, interfacing errors, language utilization errors, hardware utilization errors, and documentation or comments errors. The error lists can

form a basis for preventive approaches in the coding effort as well as checklists during code evaluation.

3.3.3.2 METHODS OF PREVENTION

Many different techniques have been applied to attempt to minimize coding errors. As indicated by Wasserman [14], some of the approaches are as simple as the development of a list of guidelines for programming style (strive for program readability, avoid programming tricks, restrict use of global data). Yourdon [1], in his discussion of ways to minimize coding errors, includes these guidelines but adds the concept of "antibugging" or including error traps in the code. These error traps are to catch standard coding errors and respond in some defined fashion.

The use of high level languages is another approach which is being used extensively. These languages often have "intelligent" compilers or syntax checking editors which can reduce errors or catch them earlier in the development phase. Specific languages have been developed (Pascal) and are being developed (ADA™) which incorporate concepts such as data typing and structured programming approaches in an effort to further reduce coding errors.

On larger projects, configuration management is aided by the use of "program libraries" with a "librarian" responsible for source and object file maintenance. Various

team programming approaches have also been used on these projects in an effort to utilize as much experience as possible to develop better code.

3.3.3.3 METHODS OF DETECTION / VERIFICATION

Several methods have been developed to aid in error detection in the coding phase. As indicated in the previous section, improved compilers and assemblers have been developed which provide greater analysis of data types and program syntax. These have been added to traditional tools such as the symbol cross-reference table output of compilers and assemblers. Tools such as automatic "flowchart generators" have been built which create a flowchart from the source code which can be compared to the design data.

Howden [31] reviews a group of other static analysis techniques which can be applied to the source code. Included in his review are: type and units analysis, reference analysis, expression analysis, and interface analysis. In addition, Howden reviews the new concept of symbolic execution and lists a variety of references. This technique involves utilization of a system which can "execute" the source program with program variables assuming symbolic values rather than numeric ones. Using algebraic and boolean logic, the system evaluates branch conditions or "predicates" to form "symbolic predicates." These are used

by the process to evaluate program logic and computations. The process has also been used to develop test data cases and, in some instances, been used to prove the correctness of the program.

Besides the various "tool" approaches, several review procedures have been developed and applied. Myers [46] presents a good review of these techniques which include: desk checking (a programmer self-testing process), code inspection (a group line-by-line analysis of the code for common errors), code walkthroughs (a group simulated "execution" examination of the code under specified states), and peer ratings (an anonymous group review of the code for style, clarity, extensibility and other qualities).

The configuration management efforts must continue to handle the inevitably staggered code development and to manage design change request impact on the code effort.

The project notebook provides a vehicle for recording reasons behind various coding decisions and dissemination of required data to the programming team.

3.3.4 DEBUG

Debugging is the process of removing errors from a program. The effort may be looked upon as a phase within the life cycle between coding and testing where the development group exercises a variety of processes to find and

remove errors prior to the formal testing phase. Debugging may also be looked upon as the effort which occurs as a result of a successful test in the testing phase (i.e. a test which has found an error) and which attempts to find the cause of the error and remove it. Whether debugging is defined as a separate phase or as a result of the test phase, or both, the process of debug provides an important opportunity for quality advancement. Debugging can produce data on error categories and error solution recognition techniques which can provide the design and coding phases with important feedback. Good debugging approaches are also required to enhance the availability of the program. An interesting history of debugging approaches and the changes in debugging philosophies through the years is given by Brooks [7]. He concludes his historical review by making a very important point: "... System debugging will take longer than one expects, and its difficulty justifies a thoroughly systematic and planned approach."[5]

3.3.4.1 PROBLEMS / ERRORS

The area of debugging can introduce errors into the program in the same way as the design and coding phases.

[5]Frederick P. Brooks, Jr., The Mythical Man-Month, Addison-Wesley, 1975, p-147.

The methods of introducing corrections to found errors can profoundly influence the clarity, efficiency, "growth" qualities, and "tolerance" qualities. Ideally, the correction process should restart the development cycle at the earliest required phase including the requirements phase if necessary. In reality, the pressures of deliveries tend to make corrections use the "band-aid" or "patch" approach. Needless to say, the quality assurance process must attempt to force usage of procedures which eliminate the patch approach.

In addition to these potential quality problems, the process of debugging, as Brooks [7] points out, is difficult. Yourdon [1] discusses debugging as an "art." Myers [46] lists some of the reasons for this difficulty: the psychological barriers inherent in admitting one's mistakes in the design and coding effort; the pressure to fix the problem as soon as possible; the nature of software (the "bug" can be anywhere - no line of code is sacred); and the lack of theory and technology on the methods of debugging.

In summary, just as the measurement of quality is hampered by the non-physical basis of software, so too is the process of error isolation hampered and made into a very mentally taxing effort.

3.3.4.2 METHODS OF APPROACH

Both Yourdon [1] and Myers [46] have excellent discussions on approaches to debugging. The methods can be somewhat loosely categorized as: brute force, analytical approaches, "rules of the road," and preventive medicine.

Brute force techniques include: data dumps (where you try to figure out what happened based on a snapshot of memory at the failure); print statement seeding (where you put print statements throughout the program to analyze, via the resultant printouts, what paths the program is executing); and the usage of traces and breakpoints (where a "debugger" tool is used to follow the sequence of routines being executed or stop the program at specified points). "Debuggers" have been expanded from traces and breakpoints to allow data and input/output monitoring and manipulation.

Analytical approaches include using inductive reasoning, using deductive reasoning, mentally "backtracking" from the error output, and hypothesis testing via executing various test cases.

The "rules of the road" category consists of a variety of error locating and error repairing principles. These rules include: check the obvious first, errors clump together, fix the error - not the sympton, determine if the error is repeatable and consistent, be thorough and methodological in data collection and hypothesis analysis, take

nothing for granted, think before you test, talk the problem over with others, if you reach an impasse - sleep on it, and know your own typical errors. Again, Myers and Yourdon present a good discussion on these and other principles.

Preventive medicine is the concept presented by Yourdon [1] (and discussed earlier - see Section 3.3.3.2) of "antibugging."

3.3.4.3 METHODS OF QUALITY ENHANCEMENT

As stated earlier in this section, debugging can provide important feedback information to the other development phase and improve the overall quality process. This feedback data results from following the error analysis process described by Myers [46] in which, for each error, the following questions are raised and answered: "When in the process was the error made?," "Who made the error?," "What was done wrong?," "How could the error have been prevented?," "Why was the error not found earlier?," "How could the error have been found earlier?," and "How was the error found?." The answers to these questions should be found and incorporated into changes in the development and quality assurance processes.

Configuration management is especially important during debug, as the temptation to incorporate changes rapidly to

fix the "bug," or try to isolate the bug, override the need to maintain orderly modifications and defined program versions. The project notebook continues to be a useful record keeping device.

3.3.5 TESTING AND VALIDATION

During testing and validation, the software is evaluated to determine if it performs all of the desired functions properly. Many different approaches have been taken to accomplish this task from informal programmer checkout to a series of independently operated tests leading to a formal operational certification.

Independent of the techniques is the basic goal of finding errors. As noted by several sources [1, 10, 46] this primary goal is sometimes forgotten as the meaning of a successful test becomes inverted (i.e. a successful test becomes one which discovers no errors rather than being one which does discover new errors).

The remainder of this section, through its subsections, will present: a discussion on the approaches used in developing software test procedures; a list of some of the errors made when developing those procedures; a discussion on some of the methods used in testing; and a presentation on the importance of error follow-up, configuration management, and the project notebook during testing.

3.3.5.1 TESTING APPROACHES

The primary approach to verifying programs is direct program testing using specific test plans. In addition to this approach, several other philosophies have been used including: symbolic execution, self-testing code, and program proofs. There has also been an approach used to test the quality of the tests called "mutation testing." Reviews of these categories may be found in a variety of references [18, 31, 42, 50]. The following sections examine these approaches.

3.3.5.1.1 SYMBOLIC EXECUTION

As discussed earlier (see Section 3.3.3.3), symbolic execution involves utilizing a system which "executes" the source code with the data variables assuming symbolic, rather than numerical, values. This approach is reviewed by Howden [31], discussed with reference to proving the correctness of programs by Hantler and King [47], and an example system called dissect is discussed by Howden [48].

3.3.5.1.2 SELF-TESTING CODE

The dynamic assertion method produces a program which is partially self testing (see Section 3.3.2.3). In this approach, code is inserted into the main program which

verifies the status of various data properties and relationships. The concept is reviewed by Howden [31], examined in terms of proving program correctness by Hantler and King [47], and presented as a concept by Stucki [43, 44].

The technique of Recovery Blocks [42, 45] provides for an evaluation of a set of alternatives to determine if an error exists. This information is then returned to surrounding code for appropriate action (see Section 3.3.2.3).

3.3.5.1.3 PROGRAM PROOFS

Some efforts have been made in the area of developing ways of mathematically proving the correctness of a program. Myers [46] reviews some of these approaches and lists various inductions and assertion proof methods that have been advanced. As Myers points out, there still is some question about the validity of the claim of guaranteeing no errors exist in anything but trivial programs.

3.3.5.1.4 MUTATION TESTING

Mutation testing is really a test of the tests. The process, as defined by Howden [49], involves defining a set of transformations to the program which should determine if the given test set will catch a specific type of error. The process is described as a method of determining whether

a given test set is complete. This concept provides a more complete and theoretical background to an effort known as "error seeding." Error seeding has been used to add known errors to a program to verify a test's ability to find a given error.

3.3.5.1.5 PROGRAM TESTING

The primary approach of program evaluation involves the execution of the program with a given input test set and an analysis of the output results. This process requires the selection of input data test sets for utilization during execution as well as the definition of the expected output. The criteria for selection of these test sets have been broken down into categories of requirements based, design based, program based, and error based. Reviews of these criteria and the reasoning behind an individual basing selection can be found in several references [5, 10, 13, 18, 31, 42, 46]. Actual test plans should include elements of tests from each of the basing methods since each provides some portion of quality testing which is not available in the others. The following paragraphs briefly examine the categories and provide some relevant references.

3.3.5.1.5.1 REQUIREMENTS BASED TESTING

This type of testing, often called "black box" testing, develops input test data to evaluate the program as defined by the requirements specification. Included in this analysis are: the functions to be performed; analysis of the input domain; extreme case analysis of input and output data; special value analysis; and analysis of the output domain. Howden [51] discusses functional based testing and reviews the overall concepts [42].

3.3.5.1.5.2 DESIGN BASED TESTS

Design based testing uses data about algorithms, data structures, modules and module interfaces as described in the design document to develop test cases. These characteristics are looked upon as abstract operators and abstract data elements. Functional style testing may then be developed based on these abstract elements and appropriate input data selected. Howden [42] reviews the concepts of design based tests, Goodenough and Gerhart [6] discuss some of the implications on test data selection, and Weyuker and Ostrand [52] provide further analysis of the use of program design information in the development of test sets.

3.3.5.1.5.3 PROGRAM BASED TESTS

Program based testing uses the specific program logic and data structures as a basis for the test data cases. This type of testing is often called "white box" or "glass box" testing since it utilizes all of the data on the specific construction methods used in creating the program. Much of the theory behind the methods used in developing test cases for this strategy is derived from graph theory as applied to logic or data flow graphs derived from the program. Included in the approaches used to develop the test data are: branch testing (each branch of the program, where a branch corresponds to an edge on the program flow graph, is traversed at least once); statement testing (each statement of the program is executed at least once); path testing (each "logical" path through the program is executed at least once); expression testing (where the various algebraic cases of the expressions in the program are tested); and data flow testing (where the data paths through the program are evaluated). Howden [42] and Dunn and Ullman [10] review these approaches and several papers [53, 54, 55] look at path testing. Statement testing is mentioned in several references as being unreliable. Expression testing and data flow testing require additional test cases based on other criteria to provide adequate test coverage.

3.3.5.1.5.4 ERROR BASED TESTS

Error based tests use typical programming error classes as the basis for test cases. This approach is obviously a supplemental one, but does provide interesting additional test sets. Gerhart and McGowan [13] briefly discuss the concept and indicate a need for research in the area and Gerhart and Yelowitz [56] indicate some examples of the types of categories that should be tested.

3.3.5.2 "ERRORS" IN TESTING

In the process of developing test procedures many possible "errors" or problems can develop. Among these are: poorly defined test objectives; tests that are vague or disorganized; inadequate time allowance for tests to be performed or results analyzed; inadequate planning for availability of test hardware or support software; lack of definition of authority or responsibility for tests; inadequate record keeping procedure definitions; incomplete retest procedures after repairs are made; undefined or erroneous expected results; incomplete test coverage; and undefined test completion criteria.

3.3.5.3 TESTING METHODS

With the knowledge of the various testing philosophies, how are these approaches implemented? How are they organized into a cohesive process to evaluate program quality? The answers to these questions are typically found in a project's test plan. This document should list the types of approaches to be used, when they should be applied, who should oversee the tests, what tools are required, and how their results are to be evaluated.

Most test plans will list a sequence of tests to be executed. These tests are, in general, ordered in a sequence that is compatible with the development process philosophy (bottom-up, top-down, or a mix). Usually included in this sequence, in one form or another, are: module related tests; integration type tests (as modules are joined together or as a new portion of a module is added to the system); function tests (where an overall system function is evaluated); system tests (where the entire system is evaluated by the development group); acceptance tests (where the customer or an outside quality control group conditionally accepts the system); and installation tests (where the system is checked out and accepted by the customer on the customer site). These tests, and the way they should be organized, are discussed in several references [1, 10, 46].

Tools have been developed to be used with the various test philosophies and approaches. In the area of program testing, Yourdon [1] describes some of these tools including: automated test data generators; automated output data checkers; automated test harnesses (an executive which controls the generation of the test data, the execution of the test, and the operation of the output checker); automated retesting of repaired software; and automated logging of test coverage and results via a monitor. Myers [46] describes several other tools including: module drivers; static flow analyzers; program correctness provers; symbolic execution "machines;" environmental simulators; and virtual machines. Dunn and Ullman [10] describe tools such as a standards analyzer and a system performance monitor.

Obviously, to maintain quality and to aid in avoidance of the test errors listed earlier, test plans need to be reviewed by a quality control group to verify content and conformance to standards.

3.3.5.4 ERROR REPORTING, CONFIGURATION MANAGEMENT, AND THE PROJECT NOTEBOOK

The process of error reporting and processing during the test and validation phase has the same importance ascribed to the error reporting and processing efforts in the debug phase (see Section 3.3.4.3). As noted by Dunn

and Ullman [10], this process can and must affect all software development not just a given project. To this end, Dunn and Ullman discuss various means of fault classification and present some sample data from published reports.

The configuration management process is again put under pressure as successful tests uncover errors causing the debug process to occur. Keeping track of changes and what tests have been executed with what version of the software can become difficult.

The project notebook continues to serve as a storehouse of data on the history and status of the project.

3.3.6 MAINTENANCE

The maintenance phase of the development cycle begins after product installation. The quality assurance process continues to have a major role as field repairs, program revisions, and customer requests for changes affect the delivered software.

Repairs due to latent defects need to be monitored to verify the quality of the change and the potential impact of the error on other systems and the overall development standards.

Customer requests for program enhancements need to be processed as a "mini" life cycle with appropriate quality measures being applied.

Configuration management becomes a major concern as the problems associated with maintaining a potentially very large number of different versions of software become great. These problems reflect the effort required to maintain accurate documentation, source files and program listings, and other information relevant to a given installation. Changes made to that installation for whatever reason must start with this data and modify it as required after the change has been successfully installed.

3.4 INDEPENDENT VERIFICATION

Independent verification uses an outside group to monitor software quality [10]. This approach has several strong advantages. First, it relieves the development group of the burden of additional, non-production related, effort. Second, the outside group should provide a more objective viewpoint on the quality and therefore the resultant product should eventually improve. Third, the independent group can provide an additional source of information about the overall product status to management. Finally, the process may take less time if the development group needs to be trained in quality assurance or is understaffed.

There are several disadvantages to independent verification. First, it costs more over a given time frame. Second, it can lead to personnel problems over differences of professional opinion on project quality between the reviewers and the development group. Finally, it can lead to potential conflict of interest by the outside group depending on their other activities.

3.4 INDEPENDENT VERIFICATION

Independent verification uses an outside group to monitor software quality [10]. This approach has several strong advantages. First, it relieves the development group of the burden of additional, non-production related, effort. Second, the outside group should provide a more objective viewpoint on the quality and therefore the resultant product should eventually improve. Third, the independent group can provide an additional source of information about the overall product status to management. Finally, the process may take less time if the development group needs to be trained in quality assurance or is understaffed.

There are several disadvantages to independent verification. First, it costs more over a given time frame. Second, it can lead to personnel problems over differences of professional opinion on project quality between the reviewers and the development group. Finally, it can lead to potential conflict of interest by the outside group depending on their other activities.

3.5 SUMMARY

This chapter has reviewed the software "qualities" and discussed which phases of the software life cycle affect the performance of the developed software with respect to these qualities. The chapter has presented various philosophies, techniques and tools which have been used in the phases of the life cycle to improve software with respect to these qualities.

The next chapter presents a software quality assurance plan, which incorporates some of the approaches presented, for the defined environment.

4. A SOFTWARE QUALITY ASSURANCE PLAN

Previous chapters have discussed a specific development environment and demonstrated a need for a software quality assurance process in that environment. A discussion of the various ways in which a quality assurance process affects the software development life cycle has been presented. The question is now: how should the types of techniques presented in the last chapter be applied to the defined environment?

The presentation of the software quality assurance process for this, or any other environment, should be done via a software quality assurance plan. As indicated by several references, a defined organized plan is, in fact, an inherent part of the quality assurance process that it documents. The plan serves as a guideline to the development and quality groups and their management during the implementation of the process. Several standards have been developed defining the form and content of software quality assurance plans. Among these are: MIL-STD-1679, a Navy document; MIL-S-52779A, a tri-service document; FAA-STD-018, a Federal Aviation Administration document; AQAP-13, a NATO document; DLAM-8200.1, a Department of Defense document; and IEEE-P730, an IEEE standard. Dunn and Ullman [10] review these documents and discuss their similarity in

content as an indication of the maturing of the concepts of software quality assurance.

This thesis, through the sections in this chapter, presents a quality assurance plan in a format based on the IEEE standard as described by Buckley [57]. Some liberties have been taken with the defined content and format based on the desire to develop a plan that is in keeping with the concept of the "small" (see Section 1.1.1.1) development environment being examined.

Before presenting the plan, a caveat must be stated. Using this plan (or any other known plan) does not guarantee that software developed will be perfect. The plan presents a process which, it is believed, will improve the quality of software currently being developed as measured by the qualities defined in section 2.2.1.

In addition to this caveat, it must be noted that: (1) the plan is intended as a guideline to indicate the processes that should be included in the development effort and (2) the plan is intended as a starting point for a dynamic quality assurance process which can and should adjust to changing needs in the development environment and in the level of quality assurance effort required.

Given the above notes, the following sections present the proposed quality assurance plan. Each section corresponds to a like-named section in IEEE-P730.

4.1 PURPOSE

The purpose of this document is to present a software quality assurance plan for the development of "small," embedded, process control software in a specific manufacturing environment. This environment develops three types of software: a standard product, which uses previously developed software and adjusts allowed parameters; a customer special product, which is based on a standard product but modifies it for a customer requested special function; and a new system product, which creates new standard functions for inclusion in the product line.

This plan examines the entire development cycle and is relevant to the following produceable items: a customer application memo, a software requirements memo, a program design document, a program test memo, a program error report, a program listing, the program itself, and a program change form.

4.2 REFERENCE DOCUMENTS

Table 4.2-1 lists documents referenced by this plan.

(1) Development Standards and Procedures Manual

(2) H. D. Mills, Mathematical Foundations for Structured Programming, FSC 72-6012, Gaithersburg, Md.: Federal Systems Division, IBM, 1972.

(3) E. Yourdon, Techniques of Program Structure and Design, Englewood Cliffs, N.J.: Prentice-Hall, 1975.

(4) H. D. Mills, "How to Write Correct Programs and Know It," Tutorial on Structured Programming, New York, N.Y.: IEEE Press, 1975.

(5) V. R. Basili and A. J. Turner, "Iterative Enhancement: A Practical Technique for Software Development," IEEE Transactions on Software Engineering, Vol. SE-1, Dec. 1975.

(6) D. L. Parnas, "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering, Vol. SE-5, March 1979.

(7) J. F. Stay, "HIPO and Integrated Program Design," IBM Systems Journal, Vol. 15(2), 1976.

(8) G. J. Myers, The Art of Software Testing, New York, N.Y.: John Wiley & Sons, 1979.

TABLE 4.2-1: REFERENCED DOCUMENTS

4.3 MANAGEMENT

The development organization will be defined in this document in terms of functional group categories. These functional groups are described in the following paragraphs.

The sales interface group provides communication with the company sales force. This group receives sales data, formats it into a generally distributed document called a "customer application memo," and provides a point of communication between engineering and sales.

The applications engineering functional group develops the hardware design, the software design, and the software code required for an individual customer machine.

The applications engineering management group is responsible for overseeing the proper operation of the applications engineering group and for interfacing that group with the other functional groups.

The engineering staff functional group is responsible for the development of new concepts and standards within the overall engineering operation. With the applications engineering group and its management, it is also responsible for developing new systems for utilization on the manufactured product.

The engineering management group oversees the operation of the applications engineering group, its management, and the engineering staff group. The engineering management

group is also responsible for interface with the production management group and the group management functional group.

The production staff functional group produces the manufactured product and performs system checkout and final product test.

The production management group oversees the operation of the production staff and interfaces with engineering management and group management.

Group management is responsible for the overall operation of development and production and oversees the operation of engineering and production management.

4.3.1 TASKS

Table 4.3-1 presents the tasks associated with the software development cycle covered by this quality assurance plan.

4.3.2 RESPONSIBILITIES

Table 4.3-1 also presents the organizational functional groups responsible for the tasks associated with the software development cycle covered by this plan.

Task	Description	Responsible Group (1)
Customer Application Memo (CAM)	Generation of memo incorporating sales information in format consistent with standards	SIG
CAM Review	Review CAM for clarity, software development category, errors, conformance to standards	AEM,AE,SIG,
Standard Development:		
Parameter selection and review	Selection of standard software parameters by assigned engineer, review of selection by second applications engineer	AE
Test Type Confirmation	Generation of program test memo to production confirming standard test approach	AE,AEM
Test Application and Report	Execution of required standard tests, report generation, system acceptance	PS,PM
Test Report Review	Confirm test report receipt, log errors	AE,AEM,ES
Error Procedure	Generation of error report Review report, plan response Confirm valid response	PS AE AEM
Customer Special Dev.:		
Requirements Memo (RM) Definition	Generation of software requirements reflecting special software requirements needed to accomplish customer requested, non-standard function	AE
RM Review	Review requirements memo for content, completeness, clarity, feasibility	AEM,ES

TABLE 4.3-1: TASKS AND RESPONSIBILITIES

Task	Description	Responsible Group (1)
Customer Spec: (Continued)		
Approach Definition	Develop proposed software modification plan indicating new modules, changed modules, and basic solution concept	AE
Approach Review (PDR)	Review proposed approach for feasibility, clarity, flexibility	AEM,AE
Detailed Design	Develop specific modification algorithms and data structures for custom software in manner consistent with standards	AE
Detailed Design Review (CDR)	Review detailed design document for feasibility, clarity, conformance to standards	AEM,AE
Code Development	Generate code based on design documentation and coding standards	AE
Code Review	Review generated code for clarity, standards incorporation, and comments by second application engineer	AE,AEM
Test Type Memo Dev. and Review	Develop proposed test set to verify new function(s), confirm other standard base tests, generate memo reflecting test plan	AE
	Review memo for validity	AEM
Test Application	Execution of required tests and report completion, system acceptance	PS,PM
Error Procedure	Report errors, test solutions	PS
	Review error & dev. solution	AE
	Review soln., confirm correction	AEM
	Log error	AEM,ES

TABLE 4.3-1: TASKS AND RESPONSIBLITIES (CONT.)

Task	Description	Responsible Group (1)
New "System" Development:		
Requirements Memo (RM) Definition	Develop requirements needed for software to create new system, examine requirements from defined qualities standpoint, examine performance requirements, write requirements memo in concert with format and content standards	AE,ES
RM Review	Review RM for content, clarity, extensibility, useability, other relevant qualities, and feasibility	AEM,ES,EM, SIG
Approach Development	Develop proposed solution concept and software architectural structure, create relevant portion of program design document consistent with content and format standards	AE,ES
Approach Review (PDR)	Review approach as defined in documentation. Evaluate feasibility, incorporation of desired qualities, and standards compliance	AEM,ES,EM
Detailed Design	Develop data and logic structures consistent with standard development approach, document following required content and format standards	AE,ES
Detailed Design Review (CDR)	Review program design document for feasibility, content and format. Examine for qualities desired.	AEM,ES,EM
Code Development	Develop code based on approved design document using code standards	AE,ES
Code Review	Review developed code for conformance to design document, coding standards, required qualities. Do code walkthrough	AEM,ES,EM

TABLE 4.3-1: TASKS AND RESPONSIBILITIES (CONT.)

Task	Description	Responsible Group (1)
New "System" Development: (Continued)		
Debug Plan	Develop a preliminary check-out plan designed to verify as much code as possible off the machine and an organized engineering check-out on the machine	AE,ES
Debug Plan Review	Review debug plan for feasibility and level of coverage	AEM,ES,AE
Test Memo Development	Develop test procedures to evaluate new software using requirements, input domain, output domain, and path based approaches. Document clearly via test memo	AE,ES
Test Memo Review	Review test memo for clarity, test coverage, organization, results definition, estimated testing time, and feasibility	AEM,ES,EM, SIG,PM
Test Application	Execute required tests as defined Generate appropriate error reports, system acceptance	PS,PM
Error Procedure	Report errors, test solutions Review error & dev. solution Review soln, confirm correction Log error	PS AE AEM AEM,ES
Error Summary Analysis	Review error log, analyze error types, propose procedure and standards changes to reduce errors	ES
Change Control Process:		
In Development	Evaluate proposed sales change Evaluate impact on current status Develop plan, implement on OK Approve plan	SIG,AEM,EM AEM,EM,AE,ES AE,ES AEM,EM

TABLE 4.3-1: TASKS AND RESPONSIBILITIES (CONT.)

Task	Description	Responsible Group (1)
Change Control Process: (Cont.)		
Field Change	Evaluate proposed change Approve change Go to customer specials	EM,AEM,SIG EM,AEM
Process Monitor	Evaluate conformance to quality and development process, recommend changes, approve changes	EM,AEM,ES, GM,PM
Arbitration	Settle disputes on process implementation	ES,EM,PM,GM

Notes: (1) Responsibilities are listed in abbreviated form where:

SIG - Sales Interface Group
AE - Applications Engineers
AEM - Application Engineering Management
ES - Engineering Staff
EM - Engineering Management
PS - Production Staff
PM - Production Management
GM - Group Management

TABLE 4.3-1: TASKS AND RESPONSIBILITIES (CONT.)

4.4 DOCUMENTATION

4.4.1 PURPOSE

This section describes the documents to be used in controlling software development and how they are audited.

A customer application memo (CAM) is developed for each machine and is reviewed as part of the minimum required audit described in section 6 of this plan. The format and content requirements of the CAM are reviewed as required by applications engineering management, engineering management, production management, and group management.

The software requirements memo (SRM) is created for customer special machines and for new systems and is reviewed as part of the required audits described in section 6 of this plan. The format and content standards for this document are reviewed as required by the applications engineering management, engineering staff, and engineering management groups.

The design document (DD) is developed for customer special machines and for new systems and is reviewed as part of the required audits described in section 6 of this plan. The format and content standards for this document are reviewed as required by the applications engineering management, engineering staff, and engineering management group.

The program listing is reviewed as described in the in-process audit and physical audit portions of section 6 of this plan. The format and content requirements are reviewed as required by applications management, engineering staff, and engineering management.

The test memo (TM) and the error report (ER) are reviewed as part of the in-process audits described in section 6 of this plan. The format and content requirements are reviewed as required by applications management, engineering staff, production management, and engineering management.

User's manuals, on those projects that require them, are reviewed as part of the in-process audits described in section 6 of this plan. The format and contents requirements are reviewed as required by the applications management, engineering staff, sales interface, and engineering management groups.

Standards documents, the program change form, and this plan are reviewed for content and format as required by applications management, engineering staff, engineering management, and group management.

4.4.2 MINIMUM DOCUMENTATION REQUIREMENTS

This section details the minimum documentation products required.

4.4.2.1 CUSTOMER APPLICATIONS MEMO (CAM)

The customer applications memo defines at a high level the functional and operational requirements of the system. This document provides the basis for the software requirements memo and therefore must reflect all required system functions, operational modes, and relevant hardware information.

4.4.2.2 SOFTWARE REQUIREMENTS MEMO (SRM)

On customer requested specials and on new systems, this document clearly and precisely defines the essential functions, design constraints and attributes of the software to be developed to meet the customer application memo. Included in this description is a discussion on: input required, functional processing used, generated output, operational modes included, mode selection logic used, and operator interface provided. It indicates any special limitations or considerations in the target environment.

4.4.2.3 DESIGN DESCRIPTION (DD)

On customer required specials and on new systems, this document describes the major components of the software design including the data requirements, the internal module communications, and the algorithms used to meet the software requirements memo defined needs. The components documentation includes an input/processing/output description and references the feature of the software requirements being supported.

4.4.2.4 TEST MEMO (TM)

This document clearly defines the test processes to be used in verifying the embedded software's proper operation. For standard products, this memo references the appropriate normal test procedure. For customer special systems and for new systems, this document references any standard plan used as a base, indicates what base plans are no longer valid, and adds those procedures which are needed to evaluate the special function. These added procedures verify the software with respect to the CAM, the SRM, and the DD. This plan includes test input data, test procedures, and expected results.

The test memo format includes the area required for test result reporting. This area is filled in during testing and references any generated error reports.

4.4.2.5 PROGRAM LISTING

The program listing is included in the documentation to allow examination of program code with respect to defined quality aspects such as clarity and to support configuration management functions in the maintenance phase of the development cycle.

4.4.2.6 ERROR REPORT (ER)

The error report form is to be filled out for all errors located in the debugging and test phases of the development cycle. This form incorporates content to allow subsequent error analysis for development of relevant error prevention procedures.

4.4.3 OTHER DOCUMENTATION

On customer requested specials and on new systems, a user's manual may be required. This manual includes clear and precise operating instructions. These instructions include set-up procedures, normal operational procedures, allowed options, alert conditions, recovery procedures and shutdown procedure descriptions.

A standards and procedures manual [1] is to be developed incorporating: document format and content descriptions, coding conventions, comments requirements, and error checklists.

The program change form is utilized as part of the configuration management process for sales and field requested modifications to the system. The document clearly defines the functional change required, provides for an estimated change time and includes an approval authorization area.

4.5 STANDARDS, PRACTICES, AND CONVENTIONS

4.5.1 PURPOSE

The following paragraphs list a set of standards, practices and conventions to be used in the development cycle and how they will be verified.

Documentation format and content standards exist for various documents listed in Section 4. Conformance to these standards is as defined in the audit processes in Chapter 6.

Logic structure standards exist and compliance is verified via the audit processes defined in Section 6 for the design document and program code products.

Coding standards and commentary standards will be followed and verified via the code audit processes described in Chapter 6.

4.5.2 CONTENT

The documentation content and format standards are defined in a separate document [1].

Logic structure utilized in design and code will conform to those structures allowed in the structured programming approach as defined by Mills [2] and as implementable in the standard language. These structures are further described in the standards document [1].

Coding standards require the use of the high level language Microprocessor Pascal™ for all code, except where its use prevents the operational feasibility of the software. In those cases, the assembly language of the computer will be allowed. Indentation of nested loops and conditional statement predicates is to be used to aid readability. Mnemonics are to be as descriptive as possible and may include only alphabetic characters and the underscore. Mnemonics must be unique within the first six characters and avoid utilization of any operating system standard function names.

Comments requirements include a standard header for all programs, processes, procedures, and functions. This header includes a description of the routine, an author's identification, a revision indication, a source date, a copyright indication, and a description of all input, output, and called routines. Comments are to be used to highlight compound statement groups in nested conditional statements. Comments are used to clarify algorithms. Comments are used to describe required detail for defined data elements. Comments shall be used to cross reference requirements memo functions to code sections.

The above standards on logic structures, coding requirements, and comments usage are examples of standards incorporated in the coding practices and procedures manual [1].

4.6 REVIEWS AND AUDITS

4.6.1 PURPOSE

During the development cycle, the various products being developed are reviewed. These audits examine the product for potential errors, feasibility, conformance to content and format standards, and performance with respect to defined software qualities [1].

The audit process is divided up into: a minimum set of design reviews, a functional test process, a physical software products review, and a series of in-process audits. The CAM, SRM and DD products are reviewed as part of the minimum design review set. The test memo and error reports are reviewed as part of the in-process audits. The program listing is reviewed in both the physical audit and the in-process audits. The program itself is evaluated during the functional audits.

The following subsections to this section describe how the various audit processes are accomplished.

4.6.2

The following paragraphs describe a minimum set of design reviews for the software development cycle in the environment of this test plan.

4.6.2.1 CUSTOMER APPLICATION MEMO REVIEW

This review is held to ensure the adequacy of the functional and operational data presented in the customer applications memo. The review also evaluates the requirements specified to determine the nature of software development to be used in implementing the CAM, i.e. standard software, customer special, or new systems development. The review includes, at a minimum, the sales interface group and the applications engineer. If required, the review also includes the applications engineering management, engineering staff and engineering management personnel. Changes made as a result of the review are re-examined until approved.

4.6.2.2 SOFTWARE REQUIREMENTS REVIEW

On customer specials and on new systems, the software requirements memo is examined to ensure the adequacy of the requirements specified. This review examines the SRM for completeness, feasibility of implementation, conformance to standards, and impact on qualities such as testability, useability, correctness, the "growth" and "tolerance" qualities, and reliability. The review is performed by the applications engineering management group with assistance as required by engineering staff. On new systems, the review includes engineering management and the sales interface group.

4.6.2.3 PRELIMINARY DESIGN REVIEW (PDR)

On customer specials and on new systems, the PDR evaluates the technical adequacy of the preliminary design of the software as given in a preliminary version of the design document (DD). The preliminary DD is reviewed for: coverage of requirements specified in the SRM, feasibility of the architecture described to implement the defined requirements, clarity of architecture description, conformance to development philosophies, and impact on qualities such as testability, clarity useability, the "growth" and "tolerance" qualities, maintainability, efficiency, reliability, and correctness. The review is done by the applications engineering management group in concert with the applications engineer. On new systems the review also includes engineering staff and engineering management. This review must be performed prior to initiation of detailed design development.

4.6.2.4 CRITICAL DESIGN REVIEW (CDR)

On customer specials and on new systems, the CDR determines the acceptability of the detailed software design as described in the detailed design document. The DD is examined for: incorporation and proper implementation of requirements given in the SRM, feasibility of algorithms

and data structures used in terms of meeting the requirements and in terms of being implemented in the target language, conformance of the DD to the content and format standards, and impact on all the defined qualities. The review is done by the applications engineering management group along with the applications engineer. On new systems the review also includes the engineering staff and the engineering management.

4.6.3 FUNCTIONAL AUDIT

The functional audit consists of machine checkout by the production staff. On customer specials and new systems, this includes special tests as defined in the test memo. On all systems, this testing includes normal machine operation checkout including a final sytems run with appropriate actual production requirements of the machine being utilized and examined. This audit is performed by the production staff and serves as an independent verification of the operation of the software with respect to the functions and operations defined in the CAM. Production management is responsible for overseeing the audit and completion of the test report portion of the test memo.

4.6.4 PHYSICAL AUDIT

On customer specials and on new systems, the program documentation including the CAM, SRM, DD and program listing are evaluated for consistency and for conformance to the content and format standards. This review is performed by the applications engineering management group and by engineering staff. On new systems the review includes the engineering management.

4.6.5 IN-PROCESS AUDIT

During the development process various other audits are performed as part of the quality assurance process. The following paragraphs detail these reviews.

On standard development software, the parameter selection is reviewed for accuracy by a second applications engineer.

On customer specials and on new systems, the code is reviewed prior to testing. The code is examined for clarity, conformance to standards, and comments usage. A code walkthrough is performed. This audit is done by the applications engineers and the engineering staff.

On new systems, a review of a proposed debug plan is done. This review evaluates the planned effort to check out the code prior to machine usage as well as preliminary

engineering checkout of the machine operation. The plan is examined with respect to feasibility and level of coverage. The review is performed by the applications engineer, the applications engineering management, and the engineering staff.

The test memo is reviewed prior to sending it to production. The memo is examined for validity and clarity. On customer specials and on new systems, the memo is also examined for feasibility, completeness of coverage, organization, results expected definitions, and estimated testing time. The test memo audit is performed by the applications engineering management. On new systems the audit includes, as required, the engineering staff, the engineering management, the sales interface group and production management.

4.7 CONFIGURATION MANAGEMENT

The configuration management process is concerned with two areas: identification of software products and change control and reporting.

Product identification involves both individual customer products and standard software. For customer software, all documentation is identified with the customer name and the machine serial number. In addition, program listings are identified with respect to the revision of the standard software used as a base. Customer specials and new systems may require their own revision information. For both customer software and standard software, revision information consists of a revision identifier and data with each revision change indicating the changes made to create the revision. Guidelines on what constitutes a revision are included in the coding practices and procedures manual [1]. Beside the listing and other documentation, the customer software is also presented in the hardware memories placed in the computer. To aid in identification, checksums of these chips are recorded and referenced to customer name, machine number, and engineer.

The change control process is concerned with changes made in two segments of the development cycle. These segments are separated by the actual shipment of the machine. Changes requested prior to shipment will be evaluated by

the sales interface group, applications engineering management and engineering management for feasibility, cost, and impact on current status. If accepted, the request initiates a change order which describes in detail the desired change. This change order is reviewed and an implementation plan is developed by applications engineering and engineering staff. This plan must then be approved by applications management and engineering management. The change is then implemented by the applications engineer and/or engineering staff.

Field change requests are evaluated by the sales interface group, applications engineering management and engineering management for feasibility and cost. On acceptance by management and by the customer, a change order is initiated describing clearly and in detail the desired functional change. This change is reviewed in much the same fashion as a new customer special machine with a new CAM and other documentation being developed.

4.8 PROBLEM REPORTING AND CORRECTIVE ACTION

Errors discovered during the reviews of the CAM, SRM, DD, and program listing are referred back to the originator of the document for correction. The document is then reviewed again. This process repeats until acceptance.

Errors found during debug and testing are recorded using the error report form. These errors are reported to the responsible engineer for correction in all applicable areas including both documentation and code. Corrections made must be noted on the error report form. Applications and engineering management are responsible for insuring that all error reports are reviewed for correction and appropriate changes are made. Engineering staff reviews all error reports and is responsible for developing new procedures and modifying standards as required to attempt to eliminate commonly reported error types.

4.9 TOOLS, TECHNIQUES, AND METHODOLOGIES

This section describes the tools, techniques and methodologies to be used in software development to aid in quality assurance. Included in this group are design philosophies, review techniques, development tools, and approaches in language utilization.

The development philosophy to be used is the "top-down" design concept described by Yourdon [3]. Also to be incorporated are the concepts described by Mills [4], Basili and Turner [5], and Parnas [6]. These philosophies should promote the general "growth" qualities as well as clarity and maintainability. These ideas involve the concept of a functional-based structure where the stepwise refinement technique is used as the design moves from the general to the specific. The Parnas ideas of choosing modules which protect volatile areas of the design are important in an environment which is based on hardware and functional modularity.

The HIPO approach [7] will be used in design documentation to aid in clarity.

Desk review [8] should be used during the code development process to aid in individual correction of errors.

The Pascal based language will be used with structured programming techniques to aid in clarity and maintainability. The syntax check feature of the language editor is to be

used after every edit session to find and correct syntax errors early and improve availability.

The debugger of the host development system and the target debugger are to be used to speed up analysis of errors and enhance availability.

The methods, techniques and tools described are incorporated in a separate development guide [1].

4.10 CODE CONTROL

Specific versions of the code need to be controlled and maintained. Copies of the final customer software, standard "generic" software, and currently valid versions of software in development need to be protected from loss.

Final customer software is copied onto flexible diskettes and stored in a protected area.

Standard software is likewise copied and stored in a protected area. In addition, engineering staff and applications engineering management are responsible for storing and additional copy of the standard software.

In progress software is backed up by engineering staff on an every other day basis as part of normal system backup procedures.

4.11 MEDIA CONTROL

Software physical media for working software is stored in a controlled access, environmentally controlled computer room. Backup copies of shipped programs are stored in an environmentally protected safe.

4.12 SUPPLIER CONTROL

This section is not relevant to the defined development environment.

4.13 RECORDS

This section discusses the retention of software records.

The CAM, SRM, DD, user's manual and program listing are retained until the machine is modified in the field. At that time, they are replaced with the new documents. The test memo and error reports are retained until reviewed and incorporated by the error analysis process.

5. PLAN IMPLEMENTATION

The last chapter presented a proposed quality assurance plan for the defined environment. Implementation of this plan in the defined environment is accompanied by several questions which need resolution for the plan to succeed. These questions reflect potential problems in three areas: plan acceptance, logistics, and growth potential.

5.1 PLAN ACCEPTANCE

To succeed the plan must be accepted by management, by the development group, and by production personnel.

An expected and legitimate question from management is the "cost" impact of the plan on development. The plan obviously calls for more work to be done and this can imply increased cost and increased time. The response that must be given is both philosophical and practical. The philosophical response deals with the entire question of the role of quality control in a business. The problem is reaching a point where:

> Management acceptance will stem from a philosophical point of view for which we may well look to Japan. There, quality control is considered a cost-saving measure; in the United States, it's generally regarded as a cost.[6]

Backing up this philosophical point of view are the items discussed in Chapter 2 of this thesis on the need for a quality assurance process. Specifically, costs can be reduced by the software quality assurance process in three ways. First, by using the design approaches, few errors will be required in testing. Second, errors that do occur

[6]Robert Dunn and Richard Ullman, Quality Assurance for Computer Software, McGraw-Hill, 1982, p-261.

will be found earlier in the development process due to the new and earlier reviews and, as documented in a wide variety of literature, these errors will therefore cost less to fix than if they were not caught until the testing phase. Finally, by using the reuseable software design concepts (in conjunction with the quality process to ensure the reliability of the reuseable code) the software development time will decrease as development becomes a process of selecting "building blocks" which will fit together to meet the customer's needs.

Acceptance by the development group is hindered by several items. First, there is a natural concern that this whole process is questioning the engineer's abilities to develop software. (What's wrong with what we're doing now?) The response to this concern needs to be based on the information presented in Chapter 2. (Nobody is perfect and we should always be looking for ways to improve the way we do things and try to decrease debugging on the production floor.) A second natural response results from looking at the whole process as more work to be done when there is not enough time now to develop code. The response here needs to be one similar to the discussion with management over increased costs and time. Finally, there may be inhibitions aroused by the concept of "pride of authorship," the desire not to use other engineer's code, or the anxiety over other

engineers seeing mistakes in your work as it is used or reviewed by them. This can only be resolved by an ongoing effort by management and other personnel to look at errors as being natural and not to demand individual perfection. Instead, the emphasis needs to be on the concept of "team" quality control (ala Japan's quality circles [11]).

Acceptance by the production personnel is hindered by the attitude: "It's not my problem - why should I worry about it?." In this case, the need again is to emphasize that quality is everyone's concern (the quality circle concepts). In addition, production personnel need to know the important role they are playing in the feedback process and in the role of independent evaluation.

Beyond the need for communication among and education of the various groups, plan acceptance and success also depends on the acceptance of the plan as policy. As noted by Dunn and Ullman [10], the informal approach to presenting the role of quality assurance in development does not work.

5.2 LOGISTICS OF IMPLEMENTATION

Implementation of the plan also faces some logistic problems. There currently exists a large amount of software design of the old style; software development is continuous and cannot be interrupted; the standards and procedures manual does not exist; the "standard" software does not exist; the various standard forms have not been created; and personnel are not familiar with the quality assurance or design concepts.

Obviously the plan cannot be implemented immediately. What is required is a phase-in process. This phase-in effort involves four basic concepts. First, implementation begins by working on parts of the plan. Those concepts which can be immediately implemented (CAM reviews, parameter selection reviews, the beginning of code reviews) are started. Subsequent sections of the plan are implemented in stages as soon as feasible. Second, this staging process must involve the development of the required new work habits. These habits involve not only the design and coding phases but also attitudes toward quality assurance as a team concept. Third, the initial development of the standards and procedures manual and its evolutionary review process must begin as soon as possible. This document forms the foundation of the quality assurance process and its sections need to be developed to support the relevant phase-in stage which

is to be initiated. Finally, the phase-in process involves beginning periodic training sessions to present the quality concepts, development methods and tools, and review procedures which will be used.

The phase-in concept has several drawbacks including slower overall progress to quality and potential confusion as only portions of new concepts are implemented. These drawbacks are offset by the advantages of maintaining production during phase-in and, at the same time, gradually increasing overall software quality.

5.3 GROWTH

The final question deals with the plan's ability to accept change. As new languages and new processors are used, new development tools (such as a source formatter or a program design language) are proposed or become available, or as new design philosophies (a data base approach for example) are proposed for review, how well will the plan respond?

The plan inherently includes a review process for the standards and procedures. New areas of development, new tools, and new philosophies should be examined as part of the review process. Changes to the process would probably undergo the same phase-in process proposed for the initial implementation of the plan.

5.4 CONCLUSION

Plan implementation involves modification of work habits and philosophies. This process should be a day by day evolution to quality.

> ...We can do a great deal to improve software immediately with the automation at hand. We do not need to set impossible, idealistic goals... We can do much by simply adjusting our everyday procedures.[7]

[7]Yukio Mizuno, "Software Quality Improvement," Computer, March 1983, p-72.

6. CONCLUSION

This thesis has examined the concepts of software quality assurance for a defined environment. In this analysis, the need for quality controls during the development of software has been examined. It has been found that there is a need to affect the entire development process, not just the program test and verification phase.

In analyzing the development process, the thesis has defined certain qualities of software which may be subjectively evaluated. These qualities were rated with respect to importance in the defined environment. The software development process itself was reviewed with respect to these qualities; and the methods, techniques, tools, and philosophies used in requirements definition, design code and testing reviewed.

A quality assurance plan was presented in a format compatible with IEEE-P730, a standard for software quality assurance plans. The plan itself consisted of: using the basic sequence of development steps of the defined environment; adding needed review steps; incorporating guidelines for design, coding and testing; and defining a design philosophy to be used.

Questions of implementation of the plan in the defined environment were reviewed and solutions to potential problems presented.

It is believed that this plan does present a viable approach to developing available, reliable, and reuseable software in the "small."

BIBLIOGRAPHY

[1] E. Yourdon, Techniques of Program Structure and Design, Englewood Cliffs, N.J.: Prentice Hall, 1975.

[2] F. DeRemer and H. H. Kron, "Programming-in-the-Large versus Programming-in-the-Small," IEEE Transactions on Software Engineering, Vol. SE-2, June 1976.

[3] M. M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution," Proceedings of the IEEE, Vol. 68, Sept. 1980.

[4] P. Freeman, "Reuseable Software Engineering: A Statement of Long-Range Research Objectives," Dept. of Information and Computer Science, University of California, Irvine, California, Tech. Rep. 159, Nov. 1980.

[5] R. A. DeMillo, F. J. Kipton, and F. G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," Computer, Vol. 11, April 1978.

[6] J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection," IEEE Transaction on Software Engineering, Vol. SE-1, June 1975.

[7] F. P. Brooks, Jr., The Mythical Man-Month, Reading, Mass.: Addison-Wesley, 1975.

[8] G. J. Myers, Software Reliability Principles and Practices, New York, N.Y.: John Wiley & Sons, 1976.

[9] J. R. Garman, "The "Bug" Heard 'Round the World," Software Engineering Notes, Vol. 6, October 1981.

[10] R. Dunn and R. Ullman, Quality Assurance for Computer Software, New York, N.Y.: McGraw-Hill, 1982.

[11] Y. Mizuno, "Software Quality Improvement," Computer, Vol. 16, March 1983.

[12] B. W. Boehm, "Software Engineering," IEEE Transactions on Computers, Vol. C-25, Dec. 1976.

[13] J. B. Goodenough and C. L. McGowan, "Software Quality Assurance: Testing and Validation," Proceedings of the IEEE, Vol. 68, Sept. 1980.

[14] A. I. Wasserman, "Information System Design Methodology," Journal of the American Society for Information Science, Vol. 31, Jan. 1980.

[15] R. Friehmelt, A. Jaeschke, and H. Tranboth, "Evolutionary Cyclic Model for Development of Complex Systems," COMPCON81 Proceedings, Sept. 1981.

[16] M. M. Lehman, "Programming Productivity - A Life Cycle Concept," COMPCON81 Proceedings, Sept. 1981.

[17] W. E. Howden, "Introduction to Software Validation," Tutorial: Software Testing and Validation Techniques, Second Edition, New York, N.Y.: IEEE Computer Society Press, 1981.

[18] E. Miller, "Introduction to Software Testing Technology," Tutorial: Software Testing and Validation Techniques, Second Edition, New York, N.Y.: IEEE Computer Society Press, 1981.

[19] C. Gane and T. Sarson, Structured System Analysis, Englewood Cliffs, N.J.: Prentice-Hall, 1979.

[20] T. DeMarco, Structured Analysis and System Specification, New York, N.Y.: Yourdon, 1978.

[21] D. T. Ross, "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Transactions on Software Engineering, Vol. SE-3, Jan. 1977.

[22] D. T. Ross and K. E. Schoeman, Jr., "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering, Vol. SE-3, Jan. 1977.

[23] D. Teichroew and E. A. Hershey III, "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, Vol. SE-3, Jan. 1977.

[24] M. W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Transactions on Software Engineering, Vol. SE-3, Jan. 1977.

[25] M. Hamilton and S. Zeldin, "Higher Order Software: A Methodology for Defining Software," IEEE Transactions on Software Engineering, Vol. SE-2, March 1976.

[26] J. F. Stay, "HIPO and Integrated Program Design," IBM Systems Journal, Vol. 15(2), 1976.

[27] HIPO-A Design Aide and Documentation Technique, White Plains, N.Y.: Data Processing Division, IBM Corporation, Document GC20-1851.

[28] J. D. Warnier, Logical Construction of Programs, New York, N.Y.: Van Nostrand Reinhold, 1974.

[29] K. E. Orr, Structured Systems Development, New York, N.Y.: Yourdon, 1977.

[30] C. G. Davis and C. R. Vick, "The Software Development System," IEEE Transactions on Software Engineering, Vol. SE-3, Jan. 1977.

[31] W. E. Howden, "A Survey of Static Analysis Methods," Tutorial: Software Testing and Validation Techniques, 2nd Ed, New York, N.Y.: IEEE Computer Society Press, 1981.

[32] P. Freeman and A. I. Wasserman, Tutorial on Software Design Techniques, 3rd Ed, New York, N.Y.: IEEE Computer Society Press, 1980.

[33] E. Yourdon and L. L. Constantine, Structured Design, Englewood Cliffs, N.J.: Prentice-Hall, 1979.

[34] M. A. Jackson, Principles of Program Design, London, England: Academic, 1975.

[35] W. Riddle, et. al., "DREAM - A Software Design Aid System," Information Technology, Proceedings of the 3rd Jerusalem Conference on Information Technology, Amsterdam: North-Holland, 1978.

[36] I. Nassi and B. Shneiderman, "Flowcharting Techniques for Structured Programming," ACM SIGPLAN Notices, Vol. 8, August 1978.

[37] S. H. Caine and E. K. Gordon, "PDL - A Tool for Software Design," Proceedings of the AFIPS 1975 NCC, Vol. 44, 1975.

[38] K. G. Salter, "A Method for Decomposing System Requirements into Data Processing Requirements," Proceedings of 2nd International Conference on Software Engineering, 1977.

[39] J. Peterson, Petri Nets: Theory and Practice, Englewood Cliffs, N.J.: Prentice-Hall, 1981.

[40] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, Dec. 1972.

[41] D. L. Parnas, "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering, Vol. SE-5, March 1979.

[42] W. E. Howden, "A Survey of Dynamic Analysis Methods," Tutorial: Software Testing and Validation Techniques, 2nd Ed, New York, N.Y.: IEEE Computer Society Press, 1981.

[43] L. G. Stucki, "New Directions in Automated Tools for Improving Software Quality," Current Trends in Programming Methodology, Vol. 2, Englewood Cliffs, N.J.: Prentice-Hall.

[44] L. G. Stucki, "The Use of Dynamic Assertions to Improve Software Quality," McDonnell Douglas, G6588, Nov. 1976.

[45] T. Anderson and R. Kerr, "Recovery Blocks in Action," Proceedings of Second International Conference on Software Engineering, IEEE Press, 1976.

[46] G. J. Myers, The Art of Software Testing, New York, N.Y.: John Wiley & Sons, 1979.

[47] S. L. Hantler and J. C. King, "An Introduction to Proving the Correctness of Programs," ACM Computing Surveys, Sept. 1976.

[48] W. E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System" IEEE Transactions on Software Engineering, Vol. SE-3, July 1977.

[49] W. E. Howden, "Completeness Criteria for Testing Elementary Program Functions," Proceedings, Fifth International Conference on Software Engineering, 1981.

[50] M. Schindler, "Software Testing - a Scarce Art Struggles to Become a Science," Electronic Design, July 1982.

[51] W. E. Howden, "Functional Based Testing," IEEE Transactions on Software Engineering, Vol. SE-6, March 1980.

[52] E. J. Weyucker and T. J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," IEEE Transactions on Software Engineering, Vol. SE-6, May 1980.

[53] W. E. Howden, "Reliability of the Path Analysis Testing Strategy," IEEE Transactions on Software Engineering, Vol. SE-2, Sept. 1976.

[54] L. J. White and E. I. Cohen, "A Domain Strategy for Computer Program Testing," IEEE Transactions on Software Engineering, Vol. SE-6, May 1980.

[55] E. Miller, M. R. Paige, J. P. Benson, and W. R. Wisehart, "Structural Techniques of Program Validation," Digest of Papers, COMPCON Spring 1974.

[56] S. L. Gerhart and L. Yelowitz, "Observations of Fallibility in Applications of Modern Programming Methodologies," IEEE Transactions on Software Engineering, Vol. SE-2, Sept. 1976.

[57] F. Buckley, "A Standard for Software Quality Assurance Plans," Computer, Vol. 12, August 1979.

[58] H. D. Mills, Mathematical Foundations for Structured Programming, FSC 72-6012, Gaithersburg, MD.: Federal Systems Division, IBM, 1972.

[59] H. D. Mills, "How to Write Correct Programs and Know It," Tutorial on Structured Programming, New York, N.Y.: IEEE Press, 1975.

[60] V. R. Basili and A. J. Turner, "Iterative Enhancement: A Practical Technique for Software Development," IEEE Transactions on Software Engineering, Vol. SE-1, Dec. 1975.

VITA

David Taylor was born August 23, 1948 in York, Pennsylvania to Henry E. and Evelyn P. Taylor. He received a B.S.E.E. from Lehigh University in 1970. He was employed by Vitro Laboratories in Silver Spring, Maryland where his interests were in real-time simulation and program development and test techniques. He is currently employed by Bell & Howell in Phillipsburg, New Jersey where his efforts involve real-time process control, component software, and high-level software development for microprocessor systems. He is a member of the I.E.E.E. and the A.C.M. He is married and has three children.