NUREG/CR-4640
PNL-5784

Handbook of Software Quality Assurance Techniques Applicable to the Nuclear Industry

Prepared by J. L. Bryant, N. P. Wilburn

Pacific Northwest Laboratory
Operated by
Battelle Memorial Institute

Prepared for
U.S. Nuclear Regulatory Commission

NUREG/CR-4640
PNL-5784

Handbook of Software Quality Assurance Techniques Applicable to the Nuclear Industry

Manuscript Completed: June 1987
Date Published: August 1987

Prepared by
J. L. Bryant, N. P. Wilburn

Pacific Northwest Laboratory
Richland, WA 99352

Prepared for
Division of Licensee Performance and Quality Evaluation
Office of Nuclear Reactor Regulation
U.S. Nuclear Regulatory Commission
Washington, DC 20555
NRC FIN P2002

ABSTRACT

Pacific Northwest Laboratory is conducting a research project to recommend good engineering practices in the application of 10 CFR 50, Appendix B requirements to assure quality in the development and use of computer software for the design and operation of nuclear power plants for NRC and industry. This handbook defines the content of a software quality assurance program by enumerating the techniques applicable. Definitions, descriptions, and references where further information may be obtained are provided for each topic.

PREFACE

This publication has been prepared to provide general recommendations for software quality assurance (SQA) programs. It is intended to be used by the commercial nuclear power industry as an aid for structuring SQA programs and assessing the adequacy of existing software practices including its development and use.

This handbook describes a framework and overall approach for an SQA program as applied to software systems alone as distinct from other systems such as associated plant hardware or the man-machine interface. It recommends a set of topics to be addressed and describes some methods and references some tools that can be used to implement and evaluate such a program. It is not intended to supplant standards and does not prescribe specific procedures. The user of the handbook can tailor the information presented to fit the individual needs of the process under consideration.

SUMMARY

Computer software has become an increasingly important part of the design and operation of systems that perform complex and critical functions, including nuclear power plants. The growing role of software in supporting nuclear plant design and operation emphasizes the need for software integrity. Pacific Northwest Laboratory (PNL) is assisting the U.S. Nuclear Regulatory Commission (NRC) in identifying areas where recommendations are needed on how to control the development and use of software in nuclear power plants.

Software development and management practices that are necessary components of software quality assurance (SQA) programs are enumerated and discussed in this report. These practices are derived from the review and compilation of many sources: industry standards of SQA, current literature, established SQA programs, and experience with software development efforts. Checklists for specific areas, such as documentation, are provided where possible.

The principal conclusions drawn from this study are the following:

- While similarities exist between SQA and QA typically applied to the installation and use of equipment in nuclear power plants, hereafter, "hardware QA," a hardware QA program cannot be directly applied to software. Rather, hardware QA principles must be modified to fit the special needs of software QA.

- Use of the SQA techniques described in this report will yield good quality software. It has been shown that quality cannot be tested into software after development is complete; it must be incorporated into the design and construction processes.

- The majority of software currently in use was not originally designed and constructed using all the systematic methods described in this report. This does not imply that all such software is of inferior quality. However, specific techniques described in this document can be applied to software currently being implemented to assure that the future use of such software is controlled and technically correct. For example, planning, documenting, and carrying out adequate testing of software systems could define and demonstrate specific cases and parameter ranges in which the software performs satisfactorily.

- Adoption of a complete and systematic SQA program is imperative for producing reliable and maintainable software. The application of specific techniques for software already developed and used cannot fully replace such an overall program.

- To adequately implement an SQA program, the SQA function must be staffed with technically competent personnel cognizant of software engineering techniques. It is imperative, too, that upper management be firmly committed to SQA.

CONTENTS

FIGURES

TABLES

1.0 INTRODUCTION

This chapter presents the need for SQA, the scope and applicability of this handbook, and a discussion of the handbook's structure.

1.1 NEED FOR SOFTWARE QUALITY ASSURANCE

Software applications have become too prominent in the nuclear industry to be developed and maintained in the informal atmosphere that was so common in early software development. Software is now used in most aspects of nuclear plant licensing, from design, through construction, and in some cases throughout the world, in operation as well. Use of software to produce calculations critical to the design of safety-related components is one example of how software can have a direct impact on safety functions in nuclear power plants. The Code of Federal Regulations, Title 10, Part 50, Appendix B (U.S. NRC 1984) requires that a quality assurance (QA) program be implemented for all "structures, systems, and components that prevent or mitigate the consequences of postulated accidents that could cause undue risk to the health and safety of the public" in nuclear power plants and fuel reprocessing plants. For this reason, software used for these purposes is subject to the same kind of engineering control principles, including quality assurance, as other facets of plant design, construction, and operation.

Existing software quality assurance (SQA) programs established by vendors and utilities represent each organization's interpretation of what is required for control of software. Because SQA techniques are not widely known or practiced there is a tendency within the nuclear industry to apply hardware QA techniques even when they are inappropriate or a "force-fit." Because the principles of development and QA of hardware are different from those for software (see Chapter 2.0), the forced substitution of one for the other can be cumbersome and ineffective.

This situation is not unique to the nuclear industry. With the exception of the aerospace industry and the U.S. Department of Defense (U.S. Department of Defense 1985), most organizations are neither organized nor equipped to properly address formal SQA requirements. Many companies lack software policies, and SQA personnel lack a parity with hardware QA personnel. There has not been enough experience in software development within most organizations to fully understand the full ramifications of SQA.

The basic need for SQA concerns the potential for latent defects or errors in software. One of the main thrusts of an SQA program is to reduce the likelihood of defects ever getting into the executable code by applying appropriate, systematic techniques throughout the software life cycle.

Latent defects are not the only problem, however. Many computer programs do not do the job that they were specified to do. A program that is poorly documented or reflects complex rather than straightforward programming techniques is hard to understand, test, or "debug." The list of problems that

confront software development, operation, and maintenance also includes unreliable software; difficult-to-maintain software; poor requirements specifications; inefficient use of resources; lack of conclusive testing; and poor documentation (Brown 1979).

SQA results in a program of planned and systematic activities to achieve the required software qualities. These actions assure that the materials, data, supplies, and services conform to established technical requirements, and that they perform satisfactorily. The essence of SQA is to prevent problems, to remove defects as they are found, and to contribute to the usability and maintainability of the software (Fujii 1978).

1.2 SCOPE

The purpose of this handbook is to delineate those techniques that must be an integral part of the development, operation and management of software systems to be applied to the design and operation of facilities regulated by 10 CFR 50. This document does not prescribe an SQA program to be adopted by all facilities. Such a program would be too general to provide usable guidance. This document does contain a fairly comprehensive list of subjects to be addressed when structuring an SQA program. For this reason, the adequacy of existing nuclear industry practices can be assessed using this document as an aid.

1.3 REPORT CONTENTS

The structure of this document reflects the emphasis on nuclear utility requirements by first comparing SQA and hardware QA requirements (Chapter 2.0). Chapter 2.0 also correlates the criteria of 10 CFR 50, Appendix B with the SQA practices delineated in this document.

The next six chapters deal with the basic definitions and philosophy of SQA. Chapter 3.0 describes the software life cycle adopted for this document and references other life cycles suitable for utility use. Chapter 4.0 discusses the management philosophy and structure of an SQA organization. Also included is a discussion of SQA training and education. Chapter 5.0 presents the requirements for documentation of SQA functions and discusses records collection, maintenance, and retention. Chapters 6.0 and 7.0 provide the rationale for adoption of the practices enumerated for records collection, maintenance, and retention. Chapter 8.0 provides the basic tools and techniques that may be used in software development.

The final three chapters of the document deal with those activities of the software life cycle that are more common to the nuclear utility environment: Configuration Management and Code Control (Chapter 9.0), Verification and Testing (Chapter 10.0), and Control of Software Procurement (Chapter 11.0).

Because the subject of SQA is so broad (basically encompassing the whole area of software engineering), only a brief discussion of each topic is

presented. Guides, standards, and documents are referenced for further reading. The references are readily available in the open literature or from standards sources such as the Computer Society of the Institute of Electrical and Electronics Engineers, Inc. (IEEE).

2.0 OVERVIEW OF SOFTWARE QUALITY ASSURANCE

This chapter discusses a number of issues: what SQA is; software QA versus hardware QA; correspondence between 10 CFR 50, Appendix B criteria and SQA practices; the types of software products that should be subject to SQA including how the intended use of the software affects the degree of QA; and the elements which make up the attributes of quality software.

2.1 DESCRIPTION OF SOFTWARE QUALITY ASSURANCE

It cannot be overemphasized that an SQA program involves the entire software development process, not just inspection and testing of the end product. Although the removal and analysis of defects is an important function of SQA, it is the prevention of defects that demands most of SQA's attention. In the past, SQA programs have equated SQA to a test program i.e., a specification of test plans, procedures, categories, types of tests, and methods of testing. The major pitfall of such a test-oriented SQA program is that quality cannot be tested into a software product; quality must be built into the product.

Definition of criteria to be used to judge the quality of a software project establishes, in essence, the SQA processes and their degree for that project. Without concrete goals, the process never reaches an endpoint. A variety of methods and criteria can be used to determine the specific SQA techniques to be applied. For example, risk analysis can be used to assess the impact of software failure on the overall system. Whenever risks and consequences are considered great, an intensive SQA effort is merited.

Each organization needs to tailor an SQA program to fit its activities. Those that develop software must be more concerned with the design and testing process than organizations who apply acquired software products. The latter organizations must concentrate their SQA efforts on configuration management and code control, acceptance testing, and procurement practices. Most nuclear utilities fall into this second group. However, utilities must be able to audit and review the SQA practices of their software suppliers to assure compliance with SQA requirements. This subject is addressed in Chapter 11.0, Control of Software Procurement.

2.2 SOFTWARE QUALITY ASSURANCE VERSUS HARDWARE QUALITY ASSURANCE

Although many of the concepts of hardware QA applied throughout the nuclear industry are applicable to SQA, there are many differences. These differences must be considered in establishing any SQA program (Dunn and Ullman 1982).

- Hardware repairs restore the original condition. Software repairs establish a new piece of software.

- Unlike hardware, software failures are rarely preceded by warnings.
- Hardware components can be standardized. Software components have rarely been standardized.
- Hardware can usually be tested exhaustively. Software essentially requires infinite testing.
- Hardware quality can be established by product measurements such as ultrasonics, materials testing, and by accumulating statistics when multiple copies are available. In contrast, each piece of software is unique.

The consequence of the above considerations is that software quality must be built into the software during the development process. SQA serves as an independent instrument for assuring compliance with performance objectives and development and maintenance standards.

2.3 CORRESPONDENCE BETWEEN APPENDIX B CRITERIA AND SQA REQUIREMENTS

The 18 elements of a complete nuclear quality assurance program given in 10 CFR 50, Appendix B correspond in many ways to the practices and requirements of a complete SQA program. Table 2.1 identifies the chapters of this document that are applicable to the 18 criteria of Appendix B.

TABLE 2.1. Correspondence Between SQA Requirements and Appendix B Criteria

Report Chapter	Appendix B Criteria
3.0 Software Life Cycle	II. Quality Assurance Program III. Design Control X. Inspection
4.0 Management	I. Organization II. Quality Assurance Program
5.0 Documentation	II. Quality Assurance Program III. Design Control IV. Procurement Document Control V. Instructions, Procedures, and Drawings VI. Document Control XVII. Quality Assurance Records
6.0 Standards, Practices, and Conventions	II. Quality Assurance Program III. Design Control

TABLE 2.1. (contd)

Report Chapter		Appendix B Criteria	
7.0	Review, Audits, and Controls	I.	Organization
		II.	Quality Assurance Program
		III.	Design Control
		V.	Instructions, Procedures, and Drawings
		VI.	Document Control
		VIII.	Identification and Control of Materials, Parts, and Components
		X.	Inspection
		XVIII.	Audits
8.0	Tools and Techniques	III.	Design Control
		IX.	Control of Special Processes
9.0	Software Configuration Management and Code Control	I.	Organization
		II.	Quality Assurance Program
		V.	Instructions, Procedures, and Drawings
		VI.	Document Control
		VII.	Control of Purchased Material, Equipment, and Services
		VIII.	Identification and Control of Materials, Parts, and Components
		XIII.	Handling, Storage and Shipping
		XIV.	Inspection, Test, and Operating Status
		XV.	Nonconforming Materials, Parts, and Components
		XVI.	Corrective Action
		XVII.	Quality Assurance Records
10.0	Verification and Testing	II.	Quality Assurance Program
		III.	Design Control
		X.	Inspection
		XI.	Test Control
11.0	Control of Software Procurement	I.	Organization
		II.	Quality Assurance Program
		III.	Design Control
		IV.	Procurement Document Control
		VII.	Control of Purchased Material, Equipment, and Services

2.4 TYPES OF SOFTWARE PRODUCTS COVERED BY SQA

The type of software products that need to be covered by an SQA program are essentially specified by the requirements given in 10 CFR 50, Appendix B. The motivation for including software under a QA program is that software can be used in the design, analysis or operation of safety-related structures, systems, and components. Four types of software are commonly used in the nuclear power industry: application, support, test and maintenance, and training software.

- Application Software

  Examples of application software include computer codes written for reactor design, core physics studies, stress calculations, thermal calculations, hydraulic calculations, all reactor safety accident analyses establishing plant limiting conditions (such as power distributions and heat generation rates), for surveillance testing, for safety systems actuation, and (for potential future applications) plant control. Other areas in which application software is used include the determination of materials compatibility, plant accessibility for in-service inspection, and maintenance and repair scheduling.

- Support Software

  Support software includes those software items employed to create or use application software, such as compilers, assemblers, editors, testing programs, data bases, input parameters to the codes, debuggers, mathematical subroutine libraries, system libraries, and utility routines. Support software should be considered in SQA because the output of the support software can influence the outputs of application software.

- Test and Maintenance Software

  Test and maintenance software is used to carry out the testing, operation, and maintenance functions during the latter phases of the software life cycle, described in Chapter 3.0. The results of testing programs are directly affected by the particular set of software tools that are used.

- Training Software

  More and more, software systems are used to perform computer aided instruction (CAI) in the tasks associated with nuclear facilities. Training software consists of CAI as well as software for simulators built for training reactor operators and other personnel in the detailed operation of nuclear facilities. It is critical that the response of the simulator (or CAI system) very closely approximate that of the real plant or actual situation. This implies that the requirements specification for the software be exactingly written and implemented.

2.5 SOFTWARE QUALITY ATTRIBUTES

To adequately establish an SQA program, the definition of software quality must be considered. The concept of what constitutes software quality is not well formulated. Table 2.2 contains an abbreviated list of attributes that can be used to define software quality (Lipow et al. 1977; Caveno and McCall 1978; Boehm et al. 1978; McCall 1979). Appendix A contains a more extensive list with expanded definitions. There is presently no way of measuring these attributes. However, this list is included as one possible checklist for evaluating software quality.

Many of the individual characteristics of software quality are in conflict. For example, added efficiency is often gained at the price of portability, accuracy, understandability, and maintainability; added accuracy often conflicts with portability due to the dependence upon hardware constraints; conciseness can conflict with readability. Software users generally find it difficult to assess the relative values of these attributes in such situations.

To summarize, the measurement of quality of a software product varies with the needs and priorities of the prospective user. No measure can currently give a single composite rating for software quality. At best, a prospective user can develop a meaningful rating system with a thorough checklist and associated priorities. Attention to characteristics of software quality throughout the software life cycle can lead to increased software reliability and significant cost savings.

TABLE 2.2. Attributes of Quality Software

Correctness	Does it do what I want?
Efficiency	Does it run as well as it can?
Flexibility	Can it be modified?
Integrity	Is it secure from intrusion?
Interoperability	Does it interface well with other systems?
Maintainability	Can it be fixed?
Portability	Can it be moved to another computer?
Reliability	Does it always perform correctly?
Reusability	Does it consist of general modules?
Testability	Can it be tested?
Usability	Is it easy to use?

3.0 SOFTWARE LIFE CYCLE

A software life cycle provides a systematic approach to the development, use, and operation of any software system (Kastelein 1971). The software life cycle has been defined as follows (IEEE 1979c): "That period of time in which the software is conceived, developed and used." All organizations that have an effective SQA program use such a formal life-cycle development system. There are many different life cycle variations, as referenced in these documents: ANSI/IEEE 1984; Boehm 1976 and 1979; Carrow 1976; Holthouse and Greenberg 1978; Kerola and Freeman 1981; Lattanzi 1979; Peters and Tripp 1978; U.S. DOD 1979 Fairley 1985. Strict adoption and use of a life cycle ensures that software development will progress in a traceable, planned, and orderly manner.

The division of the SQA effort into well-defined tasks has additional benefits. Such a division provides a logical conclusion for each phase of development, use, and operation, usually with a document. The phases and activities of the software life cycle that have been chosen for this study are given below and are shown in Figure 3.1:

1. requirements specification
2. functional specification
3. detailed software design
4. coding and software generation
5. testing, installation, and commissioning
6. transfer of responsibility
7. operation/maintenance
8. project management.

The requirements specification phase (the WHAT) consists of identifying the requirements that the computer program must satisfy. The functional specification phase (the HOW) determines the design for the software. Together, these two phases produce a statement of the project objectives, system functional specifications, and design constraints.

The detailed software design phase continues the breakdown of the functions identified in the software requirements, providing a conceptual solution or blueprint for the phases that follow. During the detailed design phase, the software component definition, interfaces, and data definitions are generated and checked for accuracy against the requirements.

The coding and generation phase consists of both actual code generation and unit testing of the program by the developer. During the testing phase, system integration of the software components and system acceptance tests are performed against the requirements. Transfer of responsibility of the maintenance of the software from the developer to the user often takes place after installation and certification. (Procured software commonly enters an organization at this phase of the life cycle. Chapter 11.0 discusses this in more detail.) The operations/maintenance phase involves the use and maintenance of

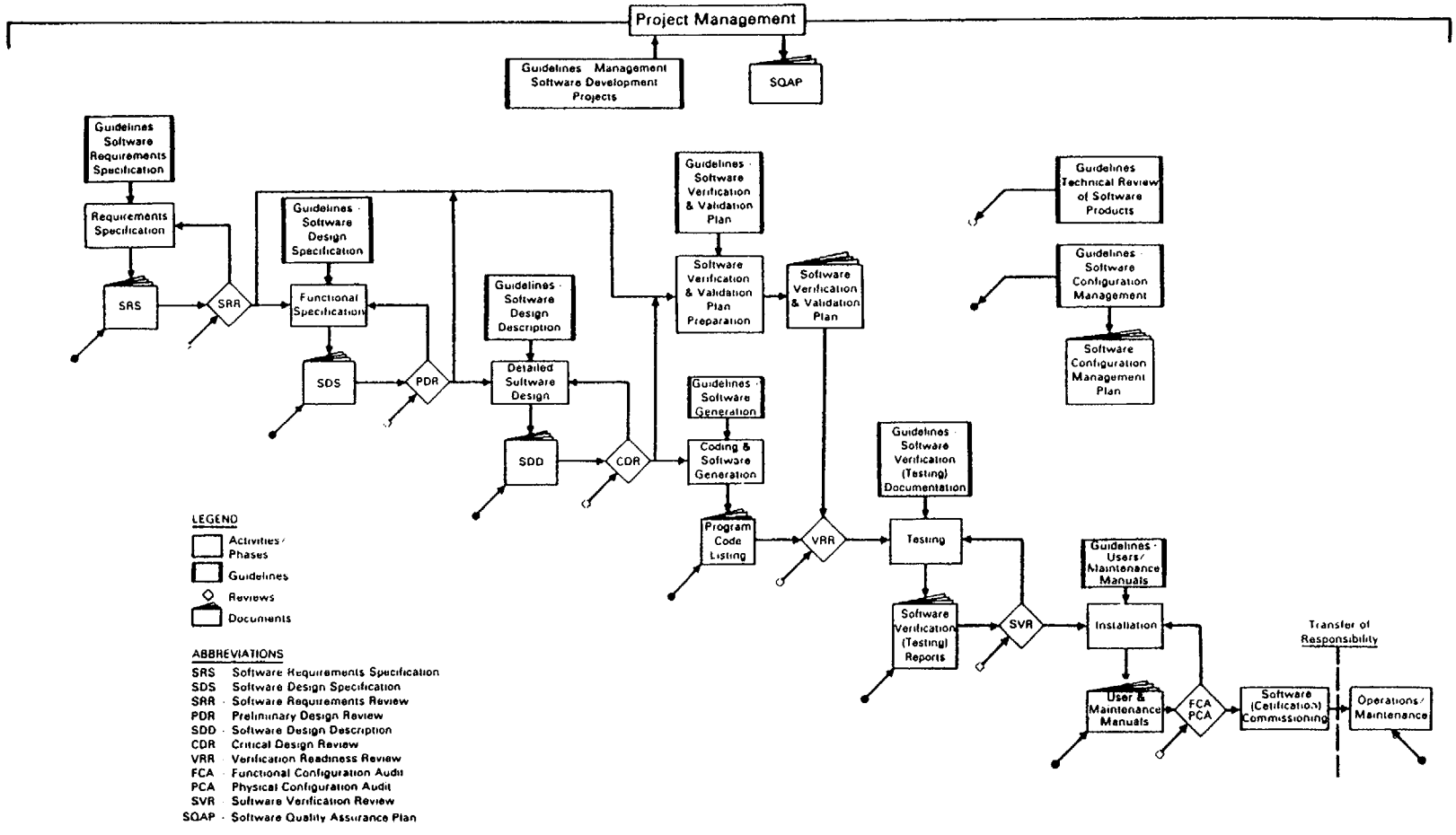

FIGURE 3.1. Software Life Cycle

the system. This includes the detection and correction of errors and incorporation of modifications to add capabilities and/or improve performance. Project management should govern throughout the entire life cycle (see Chapter 11.0).

3.1 REQUIREMENTS SPECIFICATION

During the requirements specification phase, the requirements that the computer program must satisfy are identified and recorded, usually in a document called the software requirements specification (SRS) (Wilburn 1982b). System requirements are analyzed to decide what is to be implemented by the software. Analyses determine which software functions are needed and the inputs, processing, and outputs required for each function. The requirements specification is the most significant phase of the overall project in terms of its effect on quality of the final product (Deutsch 1982). Critical errors need to be caught during the requirements analysis to avoid costly rework, reanalysis, and replanning during later development. If this phase is properly performed, the cost effectiveness ratios for requirements verification and validation and associated QA activities are probably greater than for any other activity throughout the life cycle.

The software requirements specification may take many forms. An SRS should include details of the quality of the software and its testability. It must contain enough information to frame the problem so that the software design can address the functions correctly. A type of specification that will lead to quality software is one that is 1) simply structured, 2) traceable to the specified system problem it is intended to solve, and 3) comprehensive and accurate (Dunn and Ullman 1982). The SRS should not specify how the implementation or the design is to be done (the latter is specified in the next step).

References with guidelines for how to write good software requirements specifications (Wilburn 1982b; IEEE 1984a) may be used to simplify the process.

3.2 FUNCTIONAL SPECIFICATION

Functional specifications determine the high level design for the software and are documented in a software design specification (SDS). At this point, "how" the software is to provide the requirements and to be implemented is specified. (Several acceptable methodologies are available to carry out this phase of the operation [Enos and Van Tilburg 1981; IEEE 1980b and 1983d; Jackson 1985; Yourdon and Constantine 1978].) The purpose of high level design is to separate the system into functional parts so that each part is a cohesive unit that carries out, as independently as possible, the functions specified in the software requirements specification. This is a key activity in developing the modular structure of the program. Modularity is a means of dividing a large and complex problem into a set of smaller, less complex ones. The division of the problem into hierarchies of related modules represents a major step

in the completion of the final design (Dunn and Ullman 1982). The success with which software can be made modular influences the quality of software over the entire life cycle.

3.3 DETAILED SOFTWARE DESIGN

Development of the detailed software design continues the logical separation of the functions identified in the software requirements specification (Glass 1979). The detailed design should include the definition of algorithms and equations, the detailed control logic, and data operations that are to be performed within the software. The detailed software design provides a conceptual solution or blueprint for the implementation phase that follows. All the ingredients that will ultimately make up final implementation are considered. Some of the specific considerations are defined at this time, including 1) the computer, 2) the computer resources to be used and the extent of use, 3) the computer language, 4) the modules, 5) the sequence of functions, 6) the data structures and 7) other items specific to the software product.

The primary output of this phase is a detailed design specification, which is usually designated as the software design description (SDD) It may consist of words, flowcharts, decision tables, program design languages, or other choices. Acceptable design methodologies are provided in several references (Enos and Van Tilburg 1981; IEEE 1980b and 1983d; Jackson 1975; Yourdon and Constantine 1978).

3.4 CODING AND SOFTWARE GENERATION

During this phase, the detailed software design is translated into a high level or assembly level programming language. Compilation and assembly errors are corrected and preliminary program checkout is begun by executing the individual program modules to remove obvious errors. Although much testing is performed by the developer in this phase of the life cycle, this testing does not formally constitute the testing phase of the software life cycle but is vital to the overall verification process as described in Chapter 10. The product of this phase is usually a computer program listing, the first item in the life cycle that is available in computer-readable and computer-processable form. Several guidelines have been prepared for this phase (ATC 1983 and 1985; Kernighan and Plauger 1978).

3.5 TESTING, INSTALLATION, AND COMMISSIONING

These phases of the life cycle include final testing by the developer, installation, acceptance testing, and commissioning (or certification) of the software system. During the testing phase, program components are combined into the overall software code, and testing is performed according to a developed Test (Software Verification and Validation) plan. This plan has been developed in parallel previous three phases and draws on the SRS, SDS and SDD. (Information on the specific processes to be followed during the testing

phase can be obtained from the following sources: Adrion et al. 1981; Beizer 1983 and 1984; Branstad et al. 1980; Computer Program Testing 1981; Glass 1979; IEEE 1978, 1986c; ANSI/IEEE 1987; Infotech 1979a and 1979b; McCabe 1982; Myers 1976 and 1979; Powell 1982a and 1982b.)

Testing during this phase determines whether all the requirements have been satisfied and is performed in accordance with the reviewed software verification and validation plan. Test results are evaluated and test and verification reports are prepared to describe the outcome of the process following the requirements of the Software Verification and Validation Plan (IEEE 1983f).

Part of the testing process is system integration, which brings together all system components, man, hardware and software. This testing is conducted to assure that system requirements in actual or simulated system environments are satisfied.

When the developer's testing and system installation have been completed, acceptance testing that leads to ultimate commissioning (or certification) is begun. Acceptance testing should be done by an independent organization. On completion of acceptance testing, a functional configuration audit (FCA) and a physical configuration audit (PCA) are conducted (see Figure 3.1 and Section 7.2), the official commissioning (or certification) of the software occurs, and the software is turned over to the user for implementation.

Concurrent with all of the previous phases is the preparation of the user and maintenance manuals. These documents require input from the SRS, SDS, SDD and the testing documentation. They should be reviewed ("tested") in the FCA/PCA for completeness and usability.

3.6 TRANSFER OF RESPONSIBILITY

The turnover of the software is a fairly short phase of the life cycle but is quite important. It involves the transfer of responsibility for the maintenance of the software from the developer to the user, and takes place after the FCA and PCA described in Section 7.2. At this point all the items to be given to the user for software implementation are assessed. It then becomes the user's responsibility to establish an appropriate SQA program to control and manage the software.

3.7 OPERATION/MAINTENANCE

The final phase in the software life cycle is operation and maintenance. At this point, the software has been accepted for operational use. Further activity consists of modifying the software to remove latent errors or to respond to new or revised requirements. Maintenance is defined as any change made to the software, either to correct a deficiency in performance, as required by the original software requirements specification, to compensate for

environmental changes, or to improve its operation (which is also called enhancement) (Barikh 1980; Glass and Noisex 1981; IEEE 1983e; NBS 1983).

Because changes are inevitable in this phase, a software configuration management (SCM) program following a SCM Plan must be established. SCM is discussed in Chapter 9.0. The following references are SCM standards, textbooks, and tutorials on software maintenance: Bersoff et al. 1979a and 1980; Doggett et al. 1983; IEEE 1980a, 1983b, and 1984b. Because maintenance also involves regression testing (the function required to determine that the software has not been affected by enhancement or the environment changes), systematic archiving must be implemented. These archived results can then be used for direct comparison (either automatically or manually) of software versions to determine that the software still correctly performs its originally specified tasks.

3.8 PROJECT MANAGEMENT

Project management is a critical element of SQA and covers the entire software life cycle, including both the development and operational phases. It includes management of the SQA function, the software configuration management functions, the establishment of standards, scheduling of all reviews and verification and validation, preparation of the Software Quality Assurance and the Software Configuration Management Plans.

Funding (a management function) also affects the quality of software. Typically, underfunded projects have little or no documentation. This inevitably leads to poor control over the product, resulting in poor performance. Similarly, inadequate funding significantly limits the amount of software product testing. The result is that the user performs the ultimate testing, too often by trial and error.

Project management controls the level of software quality because it determines the budget for software development. Upper management must consider total costs over the entire software life cycle from its inception to ultimate removal of the software from service and budget funds appropriately. Because low quality software results in systems that are difficult and costly to maintain, other considerations such as software reliability must be addressed as software is developed.

Many guides and standards are available for the project manager: Bruce and Pederson 1982; Cooper and Fisher 1979; DeMarco 1982; Fife 1977; IEEE 1979b; Tausworthe 1977 and 1979; and Yourdon 1979. Implementing the concepts detailed in these guides and tutorials will greatly enhance the quality and reliability of the software.

3.9 SOFTWARE VERIFICATION AND VALIDATION

Software Verification proceeds in parallel with the other elements of the life cycle. It consists of the preparation and implementation of the Software Verification and Validation Plan. The methods that can be applied are described in Section 10.1.3.

Software Validation consists of the whole process of verification throughout the software life cycle, whereas verification consists of the individual techniques and methods used.

4.0 MANAGEMENT

Software quality assurance consists of the application of procedures, techniques, and tools throughout the software life cycle to ensure that the software products conform to (meet or exceed) prespecified requirements. The objective of the SQA function is to train, plan, report, and control the software development process, so that this goal is met.[a] The SQA function must be managed with this objective in mind. The degree of quality in a program correlates strongly with the software quality objectives and priorities set by management (Boehm et al. 1976). Imposing plans and procedures that provide for well-defined milestones within the framework of software life cycle phases will allow the evaluation of software quality performance at each step (Cooper and Fisher 1979).

This section considers aspects of management of the SQA function, including overall SQA policy setting, SQA management organization, SQA program implementation, and SQA training and education.

4.1 SETTING OF OVERALL SQA POLICIES, GOALS, AND OBJECTIVES

A set of SQA management policies, goals, and objectives is necessary to guide the implementation and application of the SQA program. Upper levels of management must recognize that SQA is a vital part of the software development process and that software development, implementation, operation, and maintenance are similar to other engineering processes subject to QA requirements. This recognition by upper management must be translated into a commitment through policies that set software quality goals; establish SQA functions; and authorize the necessary resources in terms of people, money, and equipment to perform the tasks.

The SQA function must not make project management decisions. The issue of compliance or noncompliance to established standards and procedures should be the only issue in which SQA has the power to dictate a project's fate. The determination of compliance or noncompliance should be objective in nature. The consequences of noncompliance should be spelled out in the policies and procedures.

The SQA organization will be accepted more readily by the project team if its policy is one of assistance, rather than exclusively one of audit. SQA management should always be aware of the danger of overregulation. There is always the fear of empire-building associated with the SQA function, and that SQA will be a hindrance rather than an aid (Buckley and Poston 1984). An

(a) The SQA function encompasses those activities comprising an SQA program. These activities can take place internally within a project or be implemented by a separate organizational component.

organization that performs only as watchdog or policeman tends to breed resentment and will usually be unsuccessful. A spirit of cooperation cannot exist between the project team and the SQA function if the latter is always a source of bad news.

4.2 SQA MANAGEMENT ORGANIZATION

This section discusses the staffing, structure, interfaces, and authorities associated with an SQA organization. The makeup of the SQA function depends on the amount of software development performed. An organization that only uses acquired software needs a much different SQA function than an organization that extensively develops software for use by others. Likewise, software used in design and production requires a different SQA environment than that necessary for scientific research.

4.2.1 Competent Staffing

Competent staffing is the key to a successful SQA program. SQA staff must have the respect of the project staff with which they work. They must understand how the work whose quality they are assuring is actually accomplished; i.e., the SQA staff should be competent to recognize quality in software. SQA personnel should possess technical experience in software development, software specification, software design, and software testing. Senior technical staff are preferable to administrative project management staff. SQA personnel should have technical currency; they should be able to use current programming methodologies such as structured programming, top-down design, implementation, and testing methodologies. The personnel must also have the skills to communicate the concepts they are advocating (Gustafson and Kerr 1982). SQA personnel also need to be conversant with current QA practices, regulations, and standards. They must know how to construct an SQA program to meet the regulatory requirements of the nuclear industry.

4.2.2 Structure

The SQA organization should have a charter, with each element of the organization defined and its responsibility outlined. The elements responsible for SQA should be independent from those responsible for software development. The responsibilities and authorities for each element of the organization must be clearly delineated with the means established to measure the proficiency of the organization.

SQA personnel should be given sufficient responsibility, authority, and organizational freedom to identify problems in quality and to initiate, recommend, and provide solutions. The personnel performing SQA tasks should also be free to evaluate and recommend changes in the software design and production activities.

The SQA organization must not be subordinate in any way to software development activities or to software delivery. The SQA function can best be performed by a separate organization if the development of software constitutes a

significant portion of the organization's tasks. This provides the independence desired for the SQA function and can be cost-effective because all functions are maintained in a single organization. Many of the common functions that are necessary for software development projects can be implemented by one organization and in one system, assuming that a high level of technical competency is maintained by the SQA staff. However, if the projects are small enough, an SQA function that is integral to the development organization can be implemented, recognizing the danger that independence may not be maintained. Another possible mode of operation is to combine the software and hardware QA organizations. This can only be effective if the differences between hardware and software QA are recognized in the organization's policies.

4.2.3 Interfaces and Authorities

The interfaces between the SQA organization and the software development organization need to be carefully defined. It is important that project managers know when and how to bring in SQA resources.

4.3 SQA IMPLEMENTATION

Implementation of an SQA program requires that all the individuals involved understand what is happening, why it is being done, how they will benefit, how the organization will benefit, and exactly what is expected of them. Each individual involved in a software development or operation and maintenance program must be convinced that a systematic engineering methodology and an effective SQA program will help rather than hinder the software development process (Poston 1982).

4.3.1 SQA Organizational Tasks

For each project involving software development, operation, implementation and maintenance, a set of SQA tasks needs to be established. The input for and output from each task should be identified and the responsibility for the task defined (Gustafson and Kerr 1982; Boehm et al. 1976; Fisher 1978).

An SQA task list for a given project can be drawn from a number of sources: the general organizational SQA plan and policy guides, this document, or SQA plan standards that have been developed by others (ANSI/IEEE 1984; Bruce and Pederson 1982; DeMarco 1982; Fife 1977; IEEE 1979b; Tausworthe 1977 and 1979; Yourdon 1979). The software tasks may consist of the following:

- preparation of an SQA plan
- development of policies, procedures, and standards
- analysis and enforcement of policies, procedures, and standards
- certification and testing of software
- education and training of personnel performing SQA tasks
- SQA audits of
  - design
  - configuration management

- testing
- verification and validation.

Each task needs to be defined by entrance and exit criteria: i.e., what is needed to initiate the task and what is the output of the task? The output of each task should be defined in such a way that its achievement or completion can be objectively determined in a prescribed manner. Additionally, a table indicating the staffing levels for each of the tasks should be developed (ANSI/IEEE 1987).

4.3.2 Responsibilities for Tasks

The organizational elements responsible for each task listed should be identified. If two or more elements share responsibility for a task, their respective responsibilities should be identified, as well as the management position accountable for the overall project SQA.

It can be beneficial to arrange the elements of the SQA organization along task lines to clearly delineate responsibility. For example, separate elements of the organization might be assigned to perform education and training, audits, and development of policies and procedures. However, it is probably better for clear communications to assign specific personnel to each software element or major program.

4.4 TRAINING/EDUCATION

While training of personnel is not usually thought of as an SQA activity, it has been included here because quality of the software product is directly related to the competence of the individuals developing the product. The SQA program should provide in-depth training in the elements of software engineering and SQA for all personnel performing activities affecting quality. This includes training in software design and development techniques, as well as SQA procedures. The subject areas presented in this document can provide a framework for developing a training program specific to a facility's needs.

Classes and seminars can be conducted to train personnel in software development, software standards, and software engineering techniques. Seminars or short courses are available from companies in the software industry. Many of these courses are listed in trade journals. Since the seminars may be somewhat expensive, it may often be more practical to bring the seminar to the company itself. Other possibilities for training are videotape seminars, interactive laser-disk seminars, computer-aided instruction, and in-house training using in-house experts.

Training records (courses taken and dates completed) should be kept on each individual involved in software development, software maintenance, software testing, and SQA. This information is valuable in establishing when individuals should be trained or retrained. It also identifies individuals able to carry out the various phases of development throughout the software

life cycle. Tests are available through commercial organizations or through certification organizations that could be used to determine competence in the subject areas.

Training of personnel takes time and money. Some organizations have required up to a 6-month fulltime commitment by individuals to obtain adequate training in software engineering and SQA. Therefore a strong commitment by upper management to support training is necessary. This commitment should be a part of company policy.

5.0 DOCUMENTATION

Documentation issued during a software development project is essentially the only means by which progress through the software life cycle can be measured. This chapter presents minimal documentation requirements, possible additional documentation, documentation quality, and documentation control. It is recommended that the following standards and guidelines be followed when documentation is prepared: ANSI/ANS 1986; ATC 1985; IEEE 1986b; NBS 1976 and 1982; and Neumann 1982.

5.1 MINIMUM DOCUMENTATION REQUIREMENTS

For any project considered safety related and subject to 10 CFR 50, Appendix B criteria, the following documentation is considered by many to be the minimum necessary (ANSI/IEEE 1984):

- Software Quality Assurance Plan (SQAP)
- Software Design Documentation (SDS and SDD)
- Software Requirements Specification (SRS)(a)
- Software Verification and Validation Plan (SVVP)
- Software Verification and Validation Reports (SVVR)
- User Documentation.

5.1.1 Software Quality Assurance Plan (SQAP)

The SQAP should identify the documentation to be prepared during the development, verification and validation, use, and maintenance of the particular software system (ANSI/IEEE 1984; IEEE 1986b). The SQAP should identify the organizational elements responsible for the origination, verification and validation, maintenance, and control of the required documentation. It should also identify the specific reviews, audits, and the associated criteria required for each document. The SQAP should specify the tools, techniques, and methodologies to be followed during quality audits; checks and other functions that ensure the integrity of the software products; required documentation; and the management structure and methodology to be employed.

5.1.2 Software Requirements Specification (SRS)

The SRS should clearly describe each software requirement (function, performance, design constraints, and attributes of the software and external interfaces). Each requirement should be defined such that its achievement can be verified and validated objectively by a prescribed method (e.g., inspection,

(a) The SRS is mandatory for any software development project. The SRS describes what the software is to do and unless it is available, there is nothing by which software performance can be measured.

demonstration, analysis, or testing) (ANSI/IEEE 1984; Wilburn 1982b). The SRS should specify in detail the requirements agreed on by the software developer and the requester or user.

The particular form that the SRS should take is described in many standards and guidelines (ANSI/ANS 1986; IEEE 1984a; NBS 1976 and 1982; Neumann 1982).
However, it is a simple fact that the major quality problem is not the form of the software requirements specification but simply its lack or inadequacy.

5.1.3 Software Design Documentation (SDS, SDD)

The Software Design Specificaiton (SDS) should describe the major components of software design, including the data bases and internal interfaces (ANSI/IEEE 1984; IEEE 1986b). The SDS is a technical description of how the software will meet the requirements set forth in the SRS. It describes the major functions of the software such as data bases, diagnostics, external and internal interfaces, and the overall structure. The Software Design Description (SDD) involves detailed descriptions of the operating environment, monitors, timing, system throughput, tables, sizing, modeling, etc. For each component in the system, it should contain descriptions of component inputs, outputs, and calling sequences; function or tasks or algorithms; a list of all calling components; the allowable and tolerable range of values for all inputs; allowed and expected range values for all outputs; and assumptions, limitations, and effects on other components. The SDS and SDD documentation should follow the formats suggested in references on software design (Enos and Van Tilburg 1981; IEEE 1980b and 1983d; Jackson 1975; Yourdon and Constantine 1978).

5.1.4 Software Verification and Validation Plan (SVVP)

The SVVP should describe the following for each phase of the software life cycle: the verification and validation tasks; tools, techniques, methods and criteria; inputs and outputs; schedule; resources; risks and assumptions; and roles and responsibilities for accomplishing verification and validation of the software. The SVVP should identify all the test documentation that is to be prepared. The SVVP should include a verification matrix in which the requirements are referenced to their corresponding SVVP section. The IEEE and others have issued standards and guidelines useful for preparation of software verification and validation plans (Adrion et al. 1981; ANSI/ANS 1987; Deutsch 1982; IEEE 1986; Powell 1982a and 1982b; Wilburn 1983a).

5.1.5 Software Verification and Validation Reports (SVVR)

The SVVR should describe the results of the execution of the SVVP (ANSI/IEEE 1984). This includes the results of all reviews, audits, and tests required by the SQAP. The SVVR summarizes the status of the software as a result of the execution of the SVVP. It describes any major deficiencies found; provides the results of reviews, audits, and tests; and recommends whether the software is ready for operational use. The proposed IEEE standard for test documentation (IEEE 1983f) can be used to format the SVVR.

5.1.6 User Documentation

User documentation (e.g., operations and maintenance manuals, or guides) should specify and describe the required data, input sequences, options, program limitations, and other activities/items necessary for the execution of the software (ANSI/IEEE 1984; IEEE 1986b). All error messages should be identified in text meaningful to the user and possible corrective actions described. A method for transmitting user-identified errors to the software developer should be developed. User documentation should include the following items:

- user instructions that contain an introduction, a description of the user's interaction with the system, and a description of any required training for using the system
- a system narrative
- input/output specifications
- samples of original source documents and examples of all input formats, forms, or displays
- samples of all outputs, forms, reports, or displays
- data entry instructions for data preparation, data keying, data verification, data proofing, and error correction
- references to all documents or manuals intended for users
- a description of system limitations
- a description of all possible error situations and how the user should react to these situations (IEEE 1986b).

There are many user documentation guidelines and standards for preparing this documentation (ANSI 1980; NBS 1976 and 1982; Neumann 1982).

5.2 OTHER DOCUMENTATION

Other documentation that might be created during the course of a software development project includes the following (ANSI/IEEE 1984; IEEE 1986b):

- Software Development Plan (SDP)
- Software Configuration Management Plan (SCMP)
- Standards and Procedures Manual
- Training Manual
- Operations Manual
- Installation Manual
- Maintenance Manual
- Unit Development Folders
- Project File.

These additional items may be desirable for larger or more complex projects. Each item is outlined in the subsections that follow.

5.2.1 Software Development Plan

The software development plan describes the breakdown of the software development project into manageable tasks arranged into a hierarchical refinement of detail. The SDP should identify all technical and managerial activities associated with computer program development. It could specify the following items (ANSI/IEEE 1984; IEEE 1986b): an activity description, activity deliverables and associated completion criteria, prerequisite deliverables from prior activities (if any), interrelationship among the activities, and assignment of responsibilities for each activity. There can be a great deal of overlap between the SDP and the SQAP, as described above. Project management determines which section should be in which document.

5.2.2 Software Configuration Management Plan

The SCMP addresses the identification, control, status accounting, and configuration audit of the operational and support software needed to develop, produce, support, and test the software throughout its life cycle. The plan should make visible the configuration management process (ANSI/IEEE 1984; IEEE 1986b) for the installer and any regulatory agency.

5.2.3 Standards and Procedures Manual

The standards and procedures manual should provide details of the standards and procedures to be followed for software development. These standards and procedures can be derived from a general standards documentation used by the software development company or from national standards such as the IEEE (ANSI/IEEE 1984; IEEE 1986b).

5.2.4 Training Manual

The training manual should contain an introduction, instructions for using the system and preparing the input, data input descriptions, data control descriptions, instructions for running the system, and a description and interpretation of output data (ANSI/IEEE 1984; IEEE 1986b).

5.2.5 Operations Manual

The operations manual should be composed of the following items: run schedules, set-up requirements, job control procedures, error procedures, security procedures, distribution procedures, backup and recovery procedures, and restart procedures. In addition, the operations manual should contain specifications for the system, including all the environmental requirements, input/output specifications, and auditing controls (ANSI/IEEE 1984; IEEE 1986b).

5.2.6 Installation Manual

The installation manual should contain instructions for the installation of the software product, instructions for file conversion, use of user-controlled installation options, and instructions for performing an installation test (ANSI/IEEE 1984; IEEE 1986b).

5.2.7 Maintenance Manual

The maintenance manual should contain instructions for software product support and maintenance such as procedures for correcting defects and installing enhancements. This document should refer to both the procedures described in Section 9.3 and to the Software Configuration Management Plan (ANSI/IEEE 1984; IEEE 1986b).

5.2.8 Unit Development Folders

Unit development folders consist of the programmer's technical records during the design and testing work on the individual program modules. As standard project documents, these folders augment the project records and specifications by providing more technical documentation for review and inspection. The folders are especially important in large projects that are subject to frequent personnel changes or reassignments (IEEE 1983a).

5.2.9 Project File

For each project a file consisting of records, project plans, specifications, schedules, work assignments, budgets, and technical standards should be maintained. A central repository should be maintained for all current documentation associated with the project and should be available to project developers, users, and managers. It is appropriate that this file be indexed with an on-line computer system, possibly by means of a relational data base in which each word in the title can be scanned to identify documents pertinent to any requested subject. This indexing will allow the computer to do the organizing or sorting (IEEE 1983a).

5.3 DOCUMENTATION QUALITY

When considering the quality of the overall development project, the quality of the documentation itself must not be neglected. If the SQA Program is to be effective, company-wide standards should exist that specify uniform requirements for all project and software documents. These standards should define the scope and format of each document. The standards should also address the issue of technical writing style to improve document clarity and consistency. Because of the necessity for traceability, paragraph numbering by means of a decimal system is probably in order. A means of identifying changes to documents, such as bars in the right or left margins, can be used.

5.3.1 Application of Standards

Documentation should be formatted according to appropriate standards. Standards provide a means for the author to determine exactly what needs to be included in the document as well as the form it is to take. Their use promotes consistency in documentation among projects. Standards can also provide a checklist with which documents can be reviewed.

5.3.2 Review

Upon completion, all documentation should be reviewed, preferably by an independent party who has not been part of the documentation generation. Reviews will be covered in detail in Chapter 7.0. Most reviews are conducted after all documentation has been generated. However, documentation can also be reviewed piecemeal in draft form during the course of its generation.

5.3.3 Documentation Maintenance

One major problem in SQA is the maintenance of documentation. This seems to be an odious chore to most technical personnel. It has been recommended that all documentation associated with the software project be maintained on-line. This eliminates any distribution problem and the inevitable publication costs associated with revisions to documentation, especially for a large project. Maintenance of documentation on-line allows the developer to obtain the most up-to-date copy directly when it is needed. It also circumvents the problem of determining who should receive updated copies of the documentation. With the increasing cost of document reproduction and the decreasing cost of bulk storage on a computer system, this means of documentation maintenance is becoming more and more attractive. Programs to facilitate on-line documentation are available commercially.

5.4 DOCUMENTATION CONTROL

Control of documentation falls under the heading of software configuration management (see Chapter 9.0). Documentation can be considered a software product as much as the computer program itself, and is subject to the same configuration management and control. Use of an on-line computer system for documentation makes its control simpler because only one copy of the documentation need be controlled. No changes should be made to the documentation without the appropriate librarian or other responsible person's concurrence. The computer system also can provide appropriate tools such as software configuration control systems, as discussed in the following: Bersoff et al. 1979 and 1980; Doggett et al. 1983; IEEE 1980a, 1983b, and 1986a.

5.5 SOFTWARE RECORDS: COLLECTION, MAINTENANCE, AND RETENTION

This section deals with the records and data that should be collected and retained during the course of the software life cycle and the methods that should be used to assemble and maintain this documentation over the designated retention period.

5.5.1 Records to be Collected

The records that should be retained during a particular software development project and its follow-on operation and maintenance phases should be designated in the software quality assurance plan (ANSI/IEEE 1984). The types of records to be collected are determined by the overall recordkeeping objectives established during the project. Possible objectives are to provide 1) legal and contractual evidence that the software development process was performed in conformance with established professional practice or with the customer requirements, and 2) historical or reference data that could be used to discover long term trends in development techniques. The documents collected for historical or reference purposes should be capable of providing data for productivity, quality, and methodology studies.

The documents collected for legal or contractual purposes should provide evidence that 1) the SQA plan was followed and all the documents conform to the requirements of applicable standards, 2) the software meets the design intent and satisfies contractual requirements, 3) corrective action is effective, and 4) testing has been performed in accordance with test plans. The documents collected for trend analysis should provide sufficient design, implementation, and testing data so they will be useful for determining future development practices.

In addition to these kinds of documents, records should also include program media containing the exact version of programs and materials used in performing tests to assure test repeatability, and a central index listing all the documents associated with each code used in the design or safety analysis of the nuclear facility. This listing should contain all information pertinent to the documentation of the code and any data accumulated throughout code development, operation, and use. In addition, records should identify the approved list software users so that when any errors or defects are discovered, the users can be notified promptly.

The user of a critical piece of software should retain a record of how and when the code was used. This record could consist of date of use, the code's identification and version numbers, the project identification, any problems encountered in running the code, and any other pertinent information. The completed data sheet could be sent to a central location for retention. Records of use could be implemented successfully using a data base management system. If the data for each piece of software are collected in a relational data base, interrogations could determine trends and occurrences throughout the life of the software, such as problematic code modules. The collection of information on standardized forms during development and operation of software makes it easy to analyze the data using such a data base. Examples of standard

forms for this type of use are provided in the following references: Barikh 1980; Glass and Noisex 1981; IEEE 1983e and 1986d; NBS 1974 and 1983.

5.5.2 Records Maintenance

The SQA plan should specify acceptable methods of keeping records, (i.e., hard copy, computer file, microfiche, etc.) (ANSI/IEEE 1984). Maintaining records involves both physical media control, discussed in Chapter 9.0 and updating of the information contained there. Use of a data base management system or relational data base is a systematic way of accomplishing this maintenance. Specialized tools for this purpose could also be utilized effectively. See Section 8.1 for examples of such tools.

5.5.3 Records Retention

The length of retention for each type of record maintained should be specified in the SQA plan (ANSI/IEEE 1984). In addition, the retention length could be specified in the document or form itself. The date for destruction or review for possible destruction should be stated; a computerized system detailing this date could be included as part of the maintenance system.

5.5.4 Organizational Responsibility

The SQA plan should identify the organizational elements responsible for the origination, collection, maintenance, storage, and protection of records. Authorities responsible for changing, purging, and destroying records should be identified. Chapter 9.0 discusses control and management of software project records.

6.0 STANDARDS, PRACTICES, AND CONVENTIONS

The establishment, implementation, and enforcement of sound standards, practices, and conventions are essential to any SQA program. Software standards include procedures and rules employed and enforced that prescribe a disciplined, uniform approach to software development and utilization. A software standard specifies the methods and procedures that should be carried out to complete a specified software task. Practices are agreed-upon methods or techniques for developing and using software, established to ensure uniformity throughout a project. A software practice specifies the methods and techniques to be used to carry out a particular software related activity. Conventions are the uniform patterns or forms for arranging data or presenting information to provide consistency and to facilitate understanding (ANSI/IEEE 1984). (For readability, standards, practices, and conventions are referred to in this chapter as standards.)

Standards serve both technical and managerial functions. They facilitate program readability, software verification and validation, interface definition, and management review of software development. The use of standards is consistent with Appendix B of 10 CFR 50 (U.S. NRC 1984), which requires that activities affecting quality shall be prescribed by documented instructions, procedures, or drawings of a type appropriate to the circumstances.

A primary function of SQA consists of defining and recommending the software related standards, practices, and conventions for management approval and monitoring the software products and software development process to ensure that they comply with the adopted standards. The standards adopted should constitute a thread that links one event to another throughout the software life cycle and shows how the particular requirement has been implemented in the ultimate product.

The sections below consider recommended standards and their implementation, monitoring of compliance, and enforcement.

6.1 APPLICABLE STANDARDS

The project manager, in cooperation with the SQA organization, should select and establish a set of standards and procedures applicable to the particular project. These standards should be identified along with the life cycle phases to which they are applied. As a minimum, the standards should address documentation, requirements specification, design, coding, testing, and operations/maintenance.

Standards that are to be followed during the course of the project are specified in the SQA plan. If a standard or procedure is revised while the project is under way, the effect of the revised standard on the project should be evaluated and a decision made whether to continue to comply with the previous standard or with the new one. However, records should clearly state which procedure is being followed at all times during the course of the

project. As a practical matter, the standards pertinent to the particular project can be packaged in a single handbook. This can be part of the project file or maintained in on-line computer files.

6.1.1 Documentation Standards

The objective of imposing documentation standards is to ensure uniform quality. This does not mean that all software will be documented to the same level of detail. The detail needed depends on the application, complexity, and expected life span of the software. It does mean that the format of the document should be prescribed to minimize variation in style, notation, and terminology to make review, use, and control of the software documentation easier.

Documentation standards and procedures must be established early in the software development process and must be adhered to rigidly. The development of documentation standards is one of the initial activities of the SQA organization. The standards should adhere to industry standards as much as possible. Standards and guidelines for documentation are given in these references: ANSI/ANS 1986; ANSI/IEEE 1984; ATC 1983 and 1985; IEEE 1983b, 1983f, 1984a, 1986a, 1986b, and 1986e; NBS 1976 and 1982; Neumann 1982; Tausworthe 1979; Wilburn 1982b.

6.1.2 Design Standards

The basis for software reliability is design. It is a well known fact that reliability cannot be tested into a software system. Programs that are well designed in both data structure and control structure are the first defense against errors. Good design should be accompanied by careful proofreading.

The standards to be used during the design phases should be described in a design standard. Serious consideration should be given to the use of graphical techniques and the use of top-down design (Yourdon and Constantine 1978). Naming conventions and argument list standards should be addressed, and serious consideration should be given to requiring the use of program design languages.

Some attributes of software quality can be enhanced by appropriate design and implementation methodologies (Goodenough 1979). For example, robustness can be increased by the use of fault-tolerant design. Defensive programming is also a technique for increasing system robustness. Such programming consists of identifying and implementing assumptions whose violation would lead to critically unacceptable behavior. For example, if the effect of an out-of-range input would be severe, a procedure should check the range. Similarly, a program that expects input from an on-line terminal is more robust if it is designed to process arbitrary input sequences, even if the program specifications state that only certain sequences will actually be presented.

As part of the design standard, certain standards can be implemented that are specific to the design methodology, such as flowcharting standards, hierarchical chart standards, or the kind of methodology to be used [e.g., the Jackson (Jackson 1975; IEEE 1983d), the Nassi-Schneiderman (IEEE 1980b) or the

Yourdon-Constantine (Yourdon and Constantine 1978; Gane and Sarson 1977) methodologies of design]. It could also be specified that particular programming languages should be used or that certain high-level design languages are to be used.

6.1.3 Coding Standards

The practices and conventions to be used during the implementation and coding phases should be described in a coding standard. Coding standards provide for specifying quality attributes in a testable way. Implementing standards for structured code or use of structuring precompilers, local/global data access, and parameter passing will reduce the number of coding errors. Code maintenance will also be improved by using coding standards, particularly those that deal with the appearance and arrangement of the code as well as commentary. The standards should include criteria for module size, naming and numbering, header commentary, in-line commentary, local/global data access, parameter passing, and code formatting. Automated methods or manual methods for verifying compliance with programming standards can be implemented using software tools. Using these methods is cost-effective, based upon the authors' experience.

Coding standards should specify the coding language and format to be used for implementation. Available coding standards for each of the common languages are provided by Associated Technology, Inc. (ATC 1983 and 1985). Both high-level and assembly languages are available for computers. Assembly languages are used for systems programming and online systems and are not appropriate to scientific codes used in the nuclear industry. For most situations, a high-level language should be specified.

To provide uniformity in an organization, naming and labeling conventions should be established for each version and every component of the software. The program name should be included in all source code and each version derived from every element. Each version of a program must be given a unique version number. The version number should be referenced in any testing results obtained from the program as well as the date on which the program was tested. Naming and labeling conventions should be unique to each project but uniform in format throughout a company.

Use of appropriate layout conventions for each software module will result in higher quality software. Detailed specifications should be established that cover such programming conventions as indenting and spacing of the program statements, use of comments, and required use or restriction of certain features of the programming language. Layout conventions are important, particularly for maintenance. If the software throughout an organization always has the same format, a maintenance programmer can gain a great deal of information simply from familiarity with the particular format. This is the one area where a standard for an organization is a must.

Coding techniques tend to be specific to a particular programming language. All the basic structured constructs can be implemented in a standard fashion using any of the programming languages available and in use throughout

the nuclear industry. Standard constructs, followed rigorously, allow ease of translation of the so-called pseudo-coding or other design representations created during detailed design directly into the programming language.

One method for coding that allows in-line verification is assertion testing. Assertions are embedded within the code in the form of comments that can later be activated to determine the state of the processing variables at any point in the code. The assertions can check limitations on the variables that are physically realizable, such as ranges of temperature and pressure, and thereby provide a degree of verification while the code is operating. Pre- and post-processors can be used to embed assertions into almost any type of implementation language. For example, a FORTRAN assertion checker is available from the National Bureau of Standards, and other tools that perform the same function are available from commercial software vendors (Houghton 1981, 1982, and 1983; Houghton and Oakley 1980; Riddle and Fairley 1980).

Establishment of standards for in-line commentary will lead to uniformity in the amount of detail included in the commentary. Commentary should not simply reflect information that can be obtained more readily by looking at the logic flow itself, such as by saying, "branch on plus," when it is obvious from the computer coding. Commentary should add information about programming logic and can be used as a means to embed the detail design into the program listing.

6.1.4 Testing Guidelines

The standards, practices, and conventions to be used during the testing phase should be described in a set of guidelines for unit, integration, regression, and system testing. The test documentation required could follow that specified in IEEE Standard 829-1983 (IEEE 1983f) for software test documentation. The criteria for test repeatability and test coverage should be addressed, perhaps by including requirements that specify testing every requirement, user procedure, and programming statement.

The guidelines should indicate whether support software may be used. A testing guideline contains specific criteria governing the program testing be performed. It assures that programs are uniformly tested by all programmers. A draft software unit testing standard was recently developed by the IEEE that can be used in preparing such a testing guide (ANSI/IEEE 1987).

6.1.5 Code Operation/Maintenance Standards

A set of standards or guidelines should be prepared for code operation and maintenance. Many of the items in such a standard may be indirectly implemented by requiring the appropriate items in the design standards discussed in Section 6.1.3. However, some items can be considered unique to the operation and maintenance phase. Items that may be considered in preparing a code operations standard are as follows:

- All programs should be designed to print on each page of output the corresponding version number of the program, the current revision of the user's guide along with the output date, and the page number.

- All codes should print out the input data so that the input actually used by the code can be checked as part of the output verification.

- Each page of the paper or microfiche produced by a production, development, or test program should be identified by the letters PROD, DEV, or TEST, respectively.

There are two types of maintenance: repair and enhancement. Repair corrects a defect found in the software or incorporates changes required by a change in the environment; enhancement adds some feature to the requirement specification. When considering an operation/maintenance standard, a new kind of maintenance known as preventative maintenance might also be considered (Arthur 1984). Most organizations typically practice only the first two types of maintenance.

Once a program or module has been identified as a candidate for preventative maintenance, an editor (preferably not the programmer) should be chosen to review and revise the code. In programs where size is a problem, the editor should look for ways to eliminate redundant code. In a typical program, 10% to 20% of the code is probably redundant (Arthur 1984). Once the redundant code has been removed, the editor should attempt to reduce decision complexity. Automatic tools can be used to measure program complexity and indicate where improvements can be made. The editor should then look for ways to restructure the logic to reduce decision complexity. Such items should be considered when preparing an operation/maintenance standard.

6.1.6 Code Quality Requirements

A standard practice or guideline should be considered for specifying the code quality required. Chapter 2.0 and Appendix A can help determine which attributes should be included in the standard. While it is difficult to make a quantitative measurement of these attributes, a statement should be included regarding the importance of the particular attribute, a description of what it constitutes, and examples of how it can be obtained. A standard requiring that these quality attributes be included would be strong motivation toward improving software quality.

6.1.7 Other Standards

Other useful standards could be created for the following:

- software configuration management
- problem reporting and corrective action
- tools, techniques, and methodologies
- code control
- physical media control
- software supplier control
- records collection, maintenance, and retention
- training and education.

The recommendations given within this document and found in the literature (Foreman 1980; Glass 1981a; Poston 1984 and 1985) can be used to prepare standards for these areas. Such standards will provide management with a tool to evaluate how well a project is being carried out.

6.2 IMPLEMENTATION OF STANDARDS

The following sections discuss the procedures by which standards may be implemented within an organization. The first task is to determine who should create the standards and practices. It is suggested that the SQA organization be responsible; they must work with the technical staff who will ultimately use the standards and practices, however. The standards are the most important and visible result of an SQA program. It is imperative that the standards and conventions be acceptable to the software developers, to management, and to the user.

6.2.1 Use of Available Standards

Many standards have been created by companies, government agencies, and nongovernment agencies, several of which are referenced in this document and elsewhere (Wilburn 1983b). The standards can be used as guides for preparing the company's own in-house standards.

6.2.2 Creation and Review of Standards

An organization must have a structure in which to develop standards. There should be a limited number of standards and the standards themselves should be brief. However, they should not be so abbreviated that they do not cover the subject adequately. The standards should be organized systematically and be readily available, either in a looseleaf notebook or on an on-line computer system so that they can be maintained easily. The following is a suggested outline for a standards document (Glass 1981a):

- name and number of the standard
- effective date and expiration date
- objective and applicability
- method for verifying conformance
- degree of conformance required
- procedure for obtaining a waiver
- related standards and documents
- detailed statement of the standard
- explanatory comments
- indexes.

The following guidelines are suggested for creating software standards:

- follow a common outline
- use consistent terminology
- be brief

- check for overlap and inconsistency with other standards
- address the reader.

Procedures for the review and development of standards should be established. Two distinct organizations should review software standards: a technical group and a management group. Review of the standards can follow the procedures established for any other documentation review (see Chapter 7.0).

6.2.3 Maintenance and Control of Standards

Standards should be controlled like other documentation and be subject to the same software configuration management procedures as the documentation associated with that project. All standards should include a "sunset" clause by which the standard is automatically void unless reviewed and updated at periodic intervals (e.g., every 5 years).

6.3 COMPLIANCE MONITORING

The SQA organization must be involved in defining valid software development standards. They also must ensure that the software products and the processes used to develop them comply with these standards. An appropriate methodology to accomplish this is the review and audit process. This implies that SQA personnel must be competent to evaluate whether the standards are indeed being followed.

6.4 ENFORCEMENT OF STANDARDS

Associated with compliance monitoring is enforcement of the standards. A mechanism must be in place to keep management informed, and management in turn must take the steps necessary to assure that the standards are adhered to. This sometimes becomes difficult due to conflicting criteria, e.g., software quality versus production milestones. At this point, it is again necessary to reaffirm that standards are established in the interest of productivity, performance, user acceptability, predictability, and control.

7.0 REVIEWS, AUDITS, AND CONTROLS

Software development, operation, and maintenance efforts should be reviewed and audited periodically to determine conformance to SQA requirements. Technical reviews and audits should be periodically conducted to evaluate the status and quality of the engineering efforts and to assure the generation of required engineering documentation and adherence to appropriate standards. The review of software under development is the primary method used by SQA groups to assure quality.

The specific technical reviews and audits of software development plans and schedules should be identified in the SQA plan (ANSI/IEEE 1984). The procedures to be used in reviews and audits should be described in a guideline (see Freedman and Weinberg 1979; Wilburn 1982a and 1983a; Yourdon 1978). The participants and their specific responsibilities are to be identified as well. As a minimum, the following reviews and audits should be conducted (see Figure 3.1):

- software requirements review (SRR)
- preliminary design review (PDR)
- critical design review (CDR)
- software verification review (SVR)
- functional configuration audit (FCA)
- physical configuration audit (PCA)
- in-process audit
- managerial reviews (ANSI/IEEE 1984).

7.1 TECHNICAL REVIEWS

Technical reviews serve many purposes beyond helping to establish software quality. They allow several individuals to share their experience with the creators of a product. The software review has the effect of improving the technical capabilities of the individuals, as well as the team associated with the development project. The members of the group gradually come to know and understand their colleagues, how they think in certain situations, where they routinely make mistakes, etc. Such mutual understanding creates a better technical team and can keep the same types of problems from recurring. The organization of people into teams allows projects to proceed smoothly. The process of assembling the teams and assigning work can compensate for differences in individual capabilities. A team can often find defects overlooked by individuals.

7.1.1 Review Team Members

The review should be performed by individuals having sufficient technical expertise to provide a thorough review of all activities. Independent checking should be performed by an engineering or technical group rather than by an SQA

organization, which normally performs the auditing function. Review participants should be independent of those developing the program logic and technically competent in areas related to the program tasks.

7.1.2 Review Procedures

Methods of software review are provided in the following: Freedman and Weinberg 1979; Wilburn 1982a and 1983a; Yourdon 1978. The reviews and audits should be clearly identified, scheduled, and properly sequenced.

The procedures to be used for reviews and audits should identify the participants, their specific responsibilities, and the types of information to be collected and reviewed. They should also specify the preparation of a written report for each review and identify who is to prepare the reports. In addition, the report format, who is to receive the reports, and the associated management responsibilities are to be described along with any follow-up actions assure that recommendations made during the reviews and audits are properly implemented. The time interval between the review and the follow-up action should be prescribed, as well as the personnel responsible for performing the follow-up actions.

Checklists can be effectively used in the course of the technical review (ANSI/ANS 1979, Wilburn 1983a). The participants in the review should inspect all available documentation in light of these checklists before the formal review meeting. It is almost impossible to conduct an effective review during the course of the meeting itself.

7.1.3 Review Types

Table 7.1 lists the types of reviews appropriate in the software development phases of the life cycle. These reviews, which are recommended by the IEEE in their SQA plan guide (IEEE 1986b), are described in the subsections below.

7.1.3.1 Software Requirements Review

The software requirements review (SRR) takes place at the end of the life cycle phase in which the software requirements specification (SRS) (ANSI/IEEE 1984) is generated. The SRR constitutes an evaluation of the SRS. It is conducted to assure the adequacy, technical feasibility, and completeness of the requirements stated in the SRS. The SRR is held to evaluate the SRS to ensure that it is complete, verifiable, consistent, maintainable, modifiable, traceable, and usable during the operation and maintenance phases. The review ensures that sufficient detail is available to complete the software design. All organizational elements affected or impacted by the requirements should participate in this review. These may include software design personnel, software testing personnel, SQA personnel, systems engineering personnel, customers, users, and marketing and manufacturing personnel. The results of the SRR should be documented and include a record of all deficiencies identified, and a plan and schedule for corrective action. After the SRS is updated to correct these deficiencies, the document should be placed under configuration

TABLE 7.1. Checklist of Potential Reviews Throughout the Software Life Cycle

Name of Review[a]	Acronym	Reference Section[b]
Software Requirements Review*	SRR	7.1.3.1
Preliminary Design Review	PDR	7.1.3.2
Critical Design Review*	CDR	7.1.3.3
Software Verification Review*	SVR	7.1.3.4
Formal Management Reviews	--	7.1.3.5

(a) An asterisk (*) indicates those reviews required for all software development projects. Other reviews in the list may be required, depending on the nature of the software project and final product(s).
(b) The section of this document that discusses the review listed.

control, establishing the baseline to be used for software design and other efforts throughout the life cycle. During software design and its implementation, make further changes to the SRS. In such instances, the broader and far-reaching effects of such changes should be assessed.

7.1.3.2 Preliminary Design Review

The preliminary design review (PDR) is held at the end of the functional specification phase (ANSI/IEEE 1984). The PDR evaluates the technical adequacy of the preliminary design as a prelude to the detailed design. The review assesses the technical adequacy of the selected design approach; checks the design compatibility with the functional and performance requirements of the SRS; and verifies the existence and compatibility of the interfaces between software, hardware, and user.

All organizational elements that impose requirements or that are impacted by the design should send representatives to participate in this review. Documentation of the results should contain a record of all deficiencies identified in the review, and a plan and schedule for their corrective action. The updated SDS document should then be placed under configuration control, establishing a baseline for the detailed software design effort. Changes to the high level design that become necessary during detailed design, implementation or testing should be incorporated into the design documentation, with appropriate reviews made to determine the impact of these changes.

7.1.3.3 Critical Design Review

The critical design review (CDR) is held at the end of the detailed software design phase (ANSI/IEEE 1984). The CDR evaluates the technical adequacy, completeness, and correctness of the detailed design before the start of actual coding. The purpose of the CDR is to evaluate the acceptability of the

detailed design depicted in the software design description (SDD) to establish that the detailed design satisfies the requirements of the SRS; to review compatibility with other software and hardware with which the product is required to interact; and to assess the technical, cost, and schedule risks of the product's design.

The organizational elements that impose requirements or that are impacted by the design should participate in the review. Documentation of the results of the review should identify the discrepancies found during the review and should present schedules and plans for their resolution. The updated SDD is then placed under configuration control to establish a baseline for the next phase of implementation and coding.

7.1.3.4 Software Verification Review

The software verification review (SVR) constitutes an evaluation of a completed software verification and validation plan (SVVP) (ANSI/IEEE 1984). Since this plan may be developed incrementally as the requirements specification, high level design, and detailed design proceed, multiple reviews may be necessary. These reviews are held to assure that the methods described in the SVP are adequate and will provide an acceptable verification of the software. Documentation of the results of the review should record all deficiencies noted in the review, and schedules and plans for their resolution. The updated SVP, when placed under configuration control, establishes the baseline for the software verification (or testing) effort.

7.1.3.5 Managerial Reviews

These reviews are held periodically to assess the status and implementation of the SQA plan and program development plan (ANSI/IEEE 1984). The planned frequency and structure of the managerial reviews should be stated in the SQA plan and should be conducted under the direction of the program manager. Each review should be documented by a report summarizing the review findings, including any exceptions to the process stated in the SQA plan and any recommended changes or improvements.

7.2 AUDITS

The following sections describe audits of the SQA program and the SQA function.

7.2.1 Functional Configuration Audit

A functional configuration audit is held prior to software delivery to verify that all requirements specified in the software requirements specification (SRS) have been met (ANSI/IEEE 1984). The functional audit compares the code with the requirements stated in the current SRS. Its intent is to determine that the code addresses all documented requirements. Documentation of the

results should include any discrepancies and the plan and schedule for their resolution. Once the discrepancies have been resolved, the software can be delivered to the user.

7.2.2 Physical Configuration Audit

The principal purpose of a physical configuration audit is to determine if all the technical products of the computer program development effort are complete and formally acceptable to the user (ANSI/IEEE 1984). The material audited during a physical audit includes the technical products related to the computer program to be delivered to the customer, such as the final SRS, the software design description, and all other documentation formally prepared for the user and identified in previous sections.

7.2.3 In-Process Audits

Walk-throughs and inspections may be included as part of the in-process audit activity (ANSI/IEEE 1984). The objective of these audits is to verify the consistency of the product as it evolves during development or as it is changed during the maintenance phase. The results of all the in-process audits should be documented and should identify all discrepancies found and the plans and schedule for their resolution.

7.2.4 SQA Audits

These audits should evaluate the adherence to and effectiveness of the prescribed procedures, standards, and conventions provided in SQA program documentation. The internal procedures, the project SQA plans, configuration management, and contractually required deliverables from both the physical and functional aspects should be audited throughout the life cycle. The SQA audit consists of visual inspection of documents to determine if they meet accepted standards and requirements (Tausworthe 1977). The SQA audit is not intended to review the conceptual approach to a solution of a problem or to a design. Rather, the auditor should check the format of each document for conformance with its prescribed outline as well as for omissions, apparent contradictions, and items that may be sources of confusion in later work. The auditors should verify the existence of all required documents and that the quality of each is acceptable. A formal SQA audit report should be generated and submitted to the cognizant project manager for information and action. When such audits are carried on concurrently with design, coding, documentation, etc., they decrease the possibility of oversights or inadvertent misconceptions that could result in major rework and cost overruns.

7.3 CORRECTIVE ACTION

Plans and schedules for correction of deficiencies are necessary to complete the review and audit process. Corrective action should take place within a short time (specified by project management) after the review or the audit. Corrective actions are best implemented by assignment of an individual or team to carry out the corrections. If it has been decided that the corrective

action is not necessary or can be deferred, software users should be notified. The problem reporting and corrective actions detailed in Chapter 9.0 may be utilized to inform users of identified software problems.

8.0 TOOLS AND TECHNIQUES

Application of software tools and techniques in the development/operation of software systems and SQA functions can significantly improve the quality and reliability of the software.

8.1 TOOLS

The following tools can be used to develop software systems or in SQA functions:

- interrupt analyzers
- debuggers
- data base analyzers
- language processors
- text editors
- dynamic simulators
- requirements tracers
- decision tables
- hardware monitors
- structural test analyzers
- logic analyzers
- library handlers
- cross reference generators
- test drivers
- timing analyzers
- source comparitors
- instruction tracers
- editors
- dynamic analyzers
- consistency checkers
- test beds
- standards analyzers
- test result processors
- flow charters
- interface checkers
- automated test generators
- static analyzers
- software monitors
- management information systems

These tools are described in the following references: Brown 1979; Fisher 1978; Houghton 1981, 1982, and 1983; Houghton and Oakley 1980; IEEE 1979a and 1983c; NBS 1981; Osterweil 1982; Powell 1982a; Reifer 1979a; and Riddle and Fairley 1980.

Another method that can be used to improve reliability is to create for each production program run a run log that contains a record of everything that happened during the run. This could include operator commands; time and cycle of restart dumps; timing statistics showing where the CPU, I/O, and system times are being used; and a record of all errors together with diagnostic snapshots detailing the cause of the problem.

8.2 TECHNIQUES

Listed below are techniques that support various software quality assurance functions:

- auditing
- code inspection
- design inspection
- error-prone analysis
- functional testing
- logical testing
- path testing
- reviewing
- simulation
- standardization
- static analysis
- stress testing
- walkthroughs
- statistical recordkeeping

Statistical recordkeeping merits further discussion here. It has been demonstrated repeatedly (Dunn and Ullman 1982) that a few modules in any given system contribute to the observed failures far out of proportion to their number. These modules are candidates for further analysis to determine if the most appropriate action would be to redesign and recode them. Records can be used for a trend analysis and review of the effectiveness of the corrective action program.

Furthermore, it is advantageous to collect data to compute statistics about software for comparison with other project software with similar attributes. Without data to analyze, identifying effective and ineffective methods used in the development and operation of software is not possible. Methods cannot then evolve into efficient techniques and tools. Recommendations are given by NBS (1983) regarding the type of statistical data that should be collected during software development and operation. Use of statistical data is a tool with which to evaluate the effectiveness of the SQA plan itself. Quantifying the efficacy of the plan is of primary importance to assuring software quality.

One method of statistical data collection is to use automatic tools that operate in the computer on which the software is being developed. The metrics developed by McCabe (1982) can be used in collecting data with such automatic tools. Software metrics based on mechanized analysis of code systems can provide a means to quantify many important characteristics before a component module is compiled or tested. Dynamic analysis helps to identify a module's efficiency; operational analysis measures its reliability; and change management tracking (i.e., how frequently the module is required to be changed) measures its maintainability, flexibility, and reliability. Software quality measurements of this type can be applied to both the developmental and operational phases of the software life cycle.

At the conclusion of each software development project or after a period of time has elapsed while the software has been in active use, the data collected should be analyzed to determine the quality of the particular software module. Calculations can be made of the number of errors occurring as a function of the number of lines of code, the number of errors per module, and the number of errors versus the size of the module. Data analysis could involve comparison with similar data that have been accumulated from other software

systems. The data analysis could also help determine whether it might be more cost-effective to completely rewrite a piece of software than to continue to maintain inferior software.

8.3 EVALUATION OF TOOLS AND TECHNIQUES

The following factors may be used to evaluate software development and quality assurance tools and techniques (Lipow et al. 1977):

APPLICABILITY	Is the proposed tool or technique well-suited for its task? Does the proposed tool or technique have a sound quality basis?
COST-BENEFIT	Is there an explicit benefit to be gained from each of the proposed tools or techniques? Do the benefits of the proposed tool or technique equal or exceed the cost to the project?
RISK	What is the risk involved in implementing the proposed tool or technique?
STATE OF THE ART	Is the proposed tool or technique appropriate with respect to the current state of the art?
CONTROLLABILITY	How easily can the proposed tool or technique be controlled by either management or quality assurance personnel?
PAST EXPERIENCE	What has been done to show that the proposed tool or technique can be developed or implemented? Is there a good resource base from which to draw?
DELIVERY	Will the tool be delivered on time with enough information to make it easy to use and maintain?

8.4 CONTROL OF TOOLS AND TECHNIQUES

The design, development, testing, and documentation of tools and techniques must entail the same rigor and level of detail as other deliverable software. The tools and techniques need to be placed within a central repository that has management responsibility and funds for configuration management, maintenance, documentation, and dissemination, making them available for wide distribution and use by many projects and organizations. Most tools and techniques will need to be modified to fit specific projects. Therefore, consideration of their maintenance and modification is an important part of any development effort. Tools should be coded in high level languages so that portability from one computer to another does not entail major rework.

9.0 SOFTWARE CONFIGURATION MANAGEMENT AND CODE CONTROL

This chapter presents a general discussion of software configuration management (SCM) and code control, including problem reporting and corrective actions. Control of physical media and computer security (access control) are also discussed.

9.1 PROBLEM REPORTING AND CORRECTIVE ACTION

A formal procedure of software problem reporting and corrective action should be established for all "critical" software.[a] Measures should be established to promptly identify failures, malfunctions, deficiencies, deviations, defective materials and equipment, and nonconformances. The problem reporting system should interface with software configuration management procedures to ensure formal processing to resolve these problems. For any software defect identified, a time frame should be specified in line with NRC requirements in which to determine if a potential safety concern has arisen and if so, whether it is reportable to NRC.

Problems encountered during software development or operation may result from defects in software, in hardware, or in system operations. Because of the large number of possible defects, defect sources, and means of detection, a centrally located system for monitoring software defects is necessary. The objectives of the software problem reporting and tracking system are (IEEE 1986b):

- to assure that the defects are documented, corrected, and not forgotten
- to assure that the defects are assessed for their validity
- to assure that all defect corrections are approved by a review team or change control board before changes to the software configuration are made
- to facilitate measuring the defect correction process
- to inform the designer and user of the defect's status
- to provide a method of setting priorities for defect correction and scheduling appropriate actions
- to provide management with knowledge of the status of software defects

---

(a) "Critical" denotes software whose failure could cause a monetary loss or physical loss, or would have impact on public health or safety.

- to provide data for measuring and predicting software quality and reliability.

Standard forms or documents are encouraged for reporting problems and proposed changes for critical software. These forms should include the following items as a minimum (IEEE 1986b):

- a description of the problem and proposed corrective action
- authorization to implement the change
- a list of all items expected to be affected by the change
- an estimate of the resources required for the change
- identification of the personnel involved in the origination and disposition of the problem report and in the resolution of the problem
- an identification number and date.

9.1.1 Corrective Action Procedures

Corrective action procedures deal with the process of correcting software discrepancies. All corrective actions must be supported by software development and testing. Corrective actions must allow developers enough latitude so that their productivity and creativity are not encumbered. Significant negative impacts on the cost and reliability of software can occur if corrective action is not timely or is improperly administered. Software errors that go uncorrected until the system is implemented cost far more to correct than those that are uncovered during software development. The corrective action process must be established early in the development cycle. Prompt detection and early correction of software deficiencies cannot be overemphasized.

Corrective action procedures should aid rather than hinder the systematic identification and correction of software discrepancies and anomalies. The baselines established in the SCM system should permit systematic incorporation of corrective action procedures. These procedures should include steps for identifying the discrepancy in writing, documenting the proposed changes, independently reviewing the proposed changes for adequacy and retesting of the affected code and all interfacing modules.

Corrective action procedures should establish a mechanism for feedback to users on the error analysis of individual problems, and information about recurrent types of problems. Conversely, corrective action procedures should require software users to inform the program developer when errors are discovered in the computer program, so that the developer can examine and assess the overall effects of the error. Users should be provided with sufficient information to determine what effect the defect has had on previous calculations or decisions.

The program developer is ultimately responsible for the resolution of errors discovered during software development and use. Furthermore, the developer should decide if the error can be corrected with a minor change, or if a significant revision that requires reverification of the software is necessary. After the significance of the error is assessed, the developer should inform all users of the corrective action planned and the effect of the changes on the results already obtained with the defective program.

Effective corrective action procedures require input from software designers, developers, and testers, as well as SQA and configuration management organizations. This input helps determine what in the original development process went wrong. Existing methodologies should then be reexamined by project management to determine actions to be taken to minimize recurrence of such defects. In particular, any points in the software development life cycle that tend to be error-prone should be identified. This function should be incorporated as part of records collection, maintenance, and retention, discussed in Chapter 5, Section 5.

9.1.2 Organizational Responsibilities

Validating, tracking, and resolving software problems require the coordination of various groups within the organization. The SQA plan should specify the groups responsible for authorizing and implementing problem reporting and corrective actions. The groups should be composed of software designers, developers, and testers, as well as SQA and SCM personnel. These groups should be vested with the authority to enforce the program.

The relationship between the corrective action program and the overall SQA program, SCM system, and program management plan should be clearly defined. The SQA plan should also identify the point in the development process at which the generation of problem reports is required. The program plan should cover the organization of the SCM operation; management responsibilities; the interfaces between SQA, program development, and the SCM organization; SCM implementation; and applicable management policies. Each of these topics is covered in detail by Bersoff et al. (1980), and IEEE's Guide for Software Configuration Management (IEEE 1986a).

9.2 SCM ACTIVITIES

SCM activities consist of the following: configuration identification; configuration change control; configuration status accounting and reporting; configuration audits and reviews; use of SCM tools, techniques, and methodologies; supplier SCM control; and collection and retention of SCM records (Bersoff et al. 1980; Doggett et al. 1983; IEEE 1980a and 1986a).

9.2.1 Configuration Identification

Labeling the components, units, or documents associated with software can be accomplished several ways. Numbering schemes can identify the components, or a hierarchy of names can be used to organize and identify components with

mnemonics or key English labels. The concept of baselines is important in this function because it allows everyone associated with a project to have a common point of reference when they are defining, developing, or changing a software product.

9.2.2 Configuration Change Control

Configuration change control must provide the controls necessary to manage and control the change process. The mechanics of processing changes need to be defined by the SCM plan. Appropriate signoff procedures must be incorporated. A change control board (CCB) has proven to be most effective in SCM of large projects and critical software. A plan needs to be established to define the formal structure of the CCB; most importantly, the scope of the CCB authority must be established.

9.2.3 Configuration Status Accounting and Reporting

Configuration status accounting (CSA) is used to develop and maintain records of the status of software as it moves through the software life cycle. CSA may be thought of as an accounting system. It must be established early enough in the software development life cycle to allow firm control to be applied.

9.2.4 Configuration Audits and Reviews

As with any established SQA procedure, the SCM process should be audited and reviewed. The configuration items can be audited when the baseline is released. The amount of audits involved will vary according to the baseline being released. The criteria for the audit, including the roles of its participants, should be set in the SCM plan. At a minimum, audits should be performed whenever a product baseline is established, whenever the product baseline is changed, or whenever a new version of the software is released.

9.2.5 Supplier SCM Control

The subcontractor or software supplier must implement an SCM system compatible with the buyer's SCM system. The buyer's SCM group should perform an SCM audit of each major subcontractor used to ensure satisfactory compliance. Further discussion of this important activity is found in Chapter 10.0.

9.2.6 Collection and Retention of SCM Records

The general collection and retention of SCM records fall are discussed under the topic documentation (Chapter 5.0). Specific items that should be retained under code control are user-supplied items and the baseline and tests library.

User-Supplied Items

When a code is specified, items are developed that need to be retained and controlled as the software is being developed. Included in this category are

documentation providing the equations for the model, data to be included in the data base, parameters to be incorporated into the model, and possibly, previously coded subroutines and software. These items are listed at the outset of code development and placed under configuration control.

Baseline and Tests Library

The items enumerated in Chapter 5.0 that are pertinent to a particular project should be maintained, using SCM procedures in the baseline and tests library. Most of these items will change little during the course of operation of the code itself.

A computer program library system provides an effective means to control software documentation and operating programs that may be stored on several kinds of media (cards, tapes, disks, etc.). Documentation and program storage, retrieval, and change processing are essential activities in the library function. SQA policies should provide for monitoring of the library control system to ensure that correct procedures are followed.

Test documentation that has been prepared during software development should be maintained for regression testing whenever changes are made. Doing so provides confidence that the software is still reliably producing the same results as when originally tested upon completion of development. These tests can be repeated and compared manually or with appropriate file comparison routines on-line to determine where any changes have occurred in the results of the calculation or in any function that the software is to carry out.

The amount of material may seem large. However, with the improving storage media such as laser disks, videotapes, and other storage media having large capacity capabilities, the storage of such documentation on-line becomes quite practical. For updating and maintaining documentation, the advantages of these media far outweigh the inconvenience of storing them.

9.3 CODE CONTROL

Code control encompasses the procedures necessary to distribute, protect, and ensure the validity of the operating software and associated documentation. Once a code baseline has been established, the operating code should be put under SCM and placed in a centralized computer program library. The SQA plan should require that adequate controls and security measures are established for software changes and for protection from inadvertent alteration after the code has been baselined.

The software to be controlled can include computer-readable documentation and executable code. The particular types to be controlled on a given project should be specified by general SQA policy. In the nuclear industry, these types are typically involved in the design or analysis of operation of safety systems. However, any software considered critical can be a candidate for control.

New version implementation should generally follow the procedures mentioned in Section 9.2.6. It is the responsibility of the baseline and tests librarian to maintain a user list as a formal record and to notify users when a new version becomes available, alerting them when any changes have been made that might affect their calculations. It is beneficial to identify where the code or documentation has been modified by bars in the margins, or lists of pages or lines that have been modified. The easiest way to maintain a user list is by simply employing system software that identifies when a particular piece of software is being used. However, when software is distributed for use on more than one type of computer, maintenance of a user list becomes somewhat more difficult.

Accurate and unique identification of all versions of a computer program should be ensured. Controls must be established to record the changing of source or object code or related material. The software library should assign and track identification numbers of computer programs and documentation, including revisions. The library should also provide documentation of release authorization. An authorized signature list needs to be in place for this purpose. The software library should assist with the arrangements for marking, labeling, and packing software shipments, and should maintain logs and records of the distribution, inventory, and configuration control/status accounting for deliverables. A central index should be established that lists the documents composing the project file.

9.4 PHYSICAL MEDIA CONTROL

The control of physical media and associated services is the performance of functions that assure that the stored data or software is physically retrievable and cannot be lost or compromised by day-to-day operations or catastrophic events.

Typical storage media includes magnetic disks, magnetic tapes, large-scale integrated circuits, punch paper tape, program cards, magnetic diskettes, and computer listings. As technology evolves, the media will probably also include videocassette tapes, laser disks, compact disks, and other media of the audio-video industry.

9.4.1 Access Authorization and Security

Control of physical media must be provided to assure that the stored software is accessible only to authorized persons that can demonstrate need of access. Greater attention has been focused lately on physical media control because of recent violations of many computer systems by "hackers" and other unauthorized individuals. Adequate protection from unauthorized access to computer program media is available through several methods. The primary method is password control or hardware access protection, including limited-access program libraries, encryption, external markings, and proprietary statements identifying the controlled programs. Modern computer operating systems are being designed with extensive security features, especially when access is permitted by telephone lines and associated hardware modems. The following

standards and guidelines have been developed for physical security of computer media: NBS 1974, 1979, and 1980; Ruder and Madden 1978; Shankar 1977; and Steinauer 1985.

Operating computer codes are usually controlled and maintained by the code librarian. The code librarian is responsible for assuring that only the approved versions are distributed and used for analysis, and that any code modifications are made in accordance with established procedures. The computer system used should have the necessary software tools to capture all the information essential to produce distribution records and status reports on the software.

9.4.2 Protection from Damage, Alteration, and Degradation

The physical media upon which the software is stored must be controlled so that the software is not damaged, altered, or degraded. This can be accomplished by providing adequate SCM techniques, controlled software libraries, and safe storage techniques such as fireproof and waterproof vaults that are anti-static and anti-magnetic in design. Periodic physical checks of the media to ensure the use of such controlled environments will minimize degradation.

It is recommended that there be at least two backup copies of any software considered critical. These backups should be stored in separate locations to preclude the possibility that the same catastrophic event could damage both copies. One common practice is to implement a common storage facility for use by many different organizations, with each organization having an additional local facility for software storage.

A second operating copy of critical software should be provided to allow ease of access to the user in case the first operating copy is somehow degraded. The second copy is maintained on the same central machine so that the user can access it readily if it becomes evident that the primary copy has not given a correct result. Periodically, the two copies should be compared to assure that no degradation has taken place.

In addition, to safeguard against physical damage, protection from inadvertent damage during routine operations must be available. This protection can be provided by using library facilities in which access is limited by means of controlled passwords. The system manager or librarian is the only person to allow access, write, or delete privileges. Procedures must be provided to guide the librarian in providing backups and in rare instances, to only authorized changes to the software itself. There have been many cases of people, including code librarians, who have inadvertently destroyed software by not following established procedures. The routine functions of library management are described in the references listed in Section 9.4.1.

9.4.3 Verification of Physical Transmittal

When software is accessed from the central library, it is important that there be an established way to verify that the software was transmitted correctly. Several practices can be used, such as using check sums, parity

checking, and multiple transmissions with ensuing file comparisons. Appropriate test cases should be transmitted along with the software. These test cases can be run to verify that the software performs correctly.

10.0 VERIFICATION AND TESTING

As software becomes an increasingly important part of many different kinds of systems that perform complex and critical functions in the nuclear industry, the risk of software-caused failures has increased dramatically. There is now general agreement on the need to increase software reliability and quality by eliminating errors made during software development. Industry and academic institutions have responded to this need by improving development methods in the technology known as software engineering, and by employing systematic checks for detecting errors in software during and in parallel with the development process. This second technique for achieving reliable software is called verification. Software testing is a subset of software verification and will also be dealt with in this chapter. Validation is a broader term than verification and includes the whole process of verification throughout the software life cycle.

10.1 VERIFICATION

Verification concepts and principles for software development and use have typically not been widely implemented in the nuclear industry. Until recently no guidelines or standards have been available that directly address verification (except in the aerospace industries), although there are guidelines covering various types of reviews and related activities. The IEEE, NBS, and ANS are currently addressing this lack of guidelines (Adrion et al. 1981; ANS/ANSI 1987; IEEE 1986e).

Confidence in the performance of stand-alone codes has traditionally been established by benchmarking the results from code computations or empirical data, and by comparing computed results with less complex models. The verification methods described in the following sections have seldom been implemented during the other phases of code development.

Programming is done primarily by scientists or engineers, who have little training in the formal aspects of software development. These groups are highly motivated to get a program running in the shortest time possible. The results of this expediency is that the users find the bugs in a software system, after the system is put into production. While this costs the developer very little, it potentially costs the user orders of magnitude more than it would cost the developer to fix the defect during the development phase. The cost of fixing an error, both in time and money, increases dramatically as the life cycle progresses. Figure 10.1 illustrates this point (Wilburn 1983a).

10.1.1 Effects of Verification

Implementation of a verification methodology results in systematic review, analysis, and testing employed throughout the software life cycle. Verification ensures the production and maintenance of reliable and high quality software. There are two fundamental criteria for reliable software. The first is that the software adequately and correctly performs all intended functions. The second, and more subtle, criterion is that the software does not perform

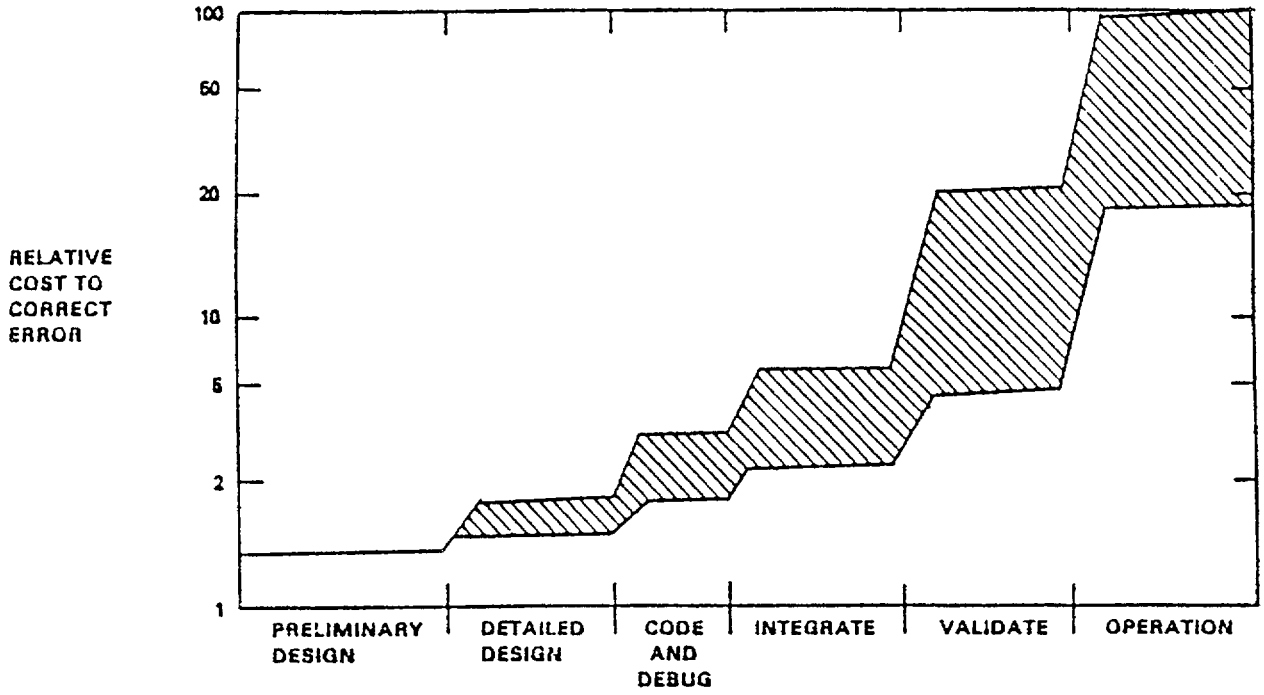

FIGURE 10.1. Software Error Cost Versus Software Development Phase

any function that either by itself or in combination with other functions can degrade the performance of the entire system.

Verification has caused major changes in the practice of software developers. "Black-art" programming practices have been replaced with planned, systematic program development. Each phase of development is considered complete only when the phase has been documented and reviewed sufficiently so that an independent person can easily understand and evaluate the documentation.

One criticism of verification programs is that they substantially increase the cost of software development. However, when the total cost of software is considered (i.e., the costs throughout the total software life cycle, from inception to decommissioning), verification actually results in a reduction in the overall cost of software development.

10.1.2 Verification Concepts

This section describes a number of concepts associated with the verification process (Wilburn 1983a). In general, these concepts will affect at least one of the phases of the software life cycle. Table 10.1 presents these concepts, which are discussed below.

TABLE 10.1. Verification Concepts

- SRS Is Required
- Baselines Must Be Congruent
- Verification Is Not Only Testing
- Verification Should Be Applied to All Components
- Verification Should Be Applied to All Changes
- Verification Should Be Independent of Development
- Verification Costs Can Be Reduced Using Automated Tools
- Verification Requires Training, Judgment, and Experience
- Verification Must Be Done by More than One Method
- A Verification Plan Is Required
- Verification Should Include a Metrics Group
- Verification Must Be Tailored to the Project
- Organization Standards and Guidelines Are Required
- Enforcement Is Required
- SCM Is Required
- Accurate Records Must Be Kept
- Management Must Show Commitment

SRS Is Mandatory: The Software Requirements Specification (SRS), which is the first product in the software life cycle, is a requirement for any verification program. The SRS forms the foundation for determining the correctness of a software system by specifying what the software is supposed to do. Unless the tester, developer, and user know what the program is supposed to do, the program is essentially impossible to verify.

Baselines Must Be Congruent: Verification must check the consistency between successive levels of detail within and between successive baselines (i.e., products of successive life cycle phases). The extent to which this can be accomplished depends on the information contained at each level in the respective baselines. The design specification, for example, can only be verified against an unambiguous and complete SRS. In this manner, verification ensures that what is intended in one baseline or life cycle phase is actually achieved in the succeeding one. In other terms, the verification process must establish traceability between life cycle phases. A systematic method for carrying out this traceability should also be included within a software configuration management program, which is described in Chapter 9.0.

Verification Is Not Only Testing: Verification should be integrated into all phases of the software development life cycle, rather than isolated in a separate testing stage, which takes place long after the requirements specification

and design phases. Testing is one aspect of verification, but it cannot do the whole job. Verification is most effective and efficient when applied from the beginning of the development process.

Verification Should Be Applied to All Components: Many software products are created within each software life cycle phase and include intermediate products, support software, or tools that have been created for the particular development process. Verification should be applied to these components, as well as to the end products, to accomplish a quality end-product.

Verification Should Be Applied to All Changes: Because of their high cost, documentation and verification of software changes are sometimes omitted, with severe consequences to the overall project. If the changes are significant (i.e., a modification to the original requirements), the change should be implemented as though a new piece of software were being developed. Modification or correction to the software structure at later phases requires reverification of the original structure produced during previous phases.

Verification Should Be Independent of Development: Independent verification by a group separate from the development group is usually necessary. A software developer has a vested interest in showing that the piece of software works because it reflects on his other skills as a developer. A group independent of the development process is likely to do a more thorough and objective job of planning and executing the software verification, producing a series of complex tests and verification methods. Another motive for an independent verification team is its freedom from preconceived ideas that may create blind spots in the evaluation.

Verification Costs Can Be Reduced by Using Automated Tools: Many activities of the verification process throughout the software life cycle can be reduced in cost using automated tools. Government and industry publications are available that give extensive lists of these tools (see Houghton 1980, 1981, 1982; IEEE 1979a, NBS 1981).

Verification Requires Training, Judgment, and Experience: The use of verification does not of itself guarantee success. Success depends heavily on the use of judgment, training, and experience by the individuals involved. It is best to use people who have experience in software development projects that have employed software engineering and verification methodologies.

Verification Must Be Done By More Than One Method: Traditionally, testing has been the only methodology of software verification. However, a single method of verification cannot provide sufficient substantiation of the correctness and reliability of the software.

A Verification Plan Is Required: A plan must be created to describe the verification process in detail. A software verification plan describes the verification approach and methods of performance, specifies how errors will be reported and documented, specifies the level of detail, and establishes the degree of rigor to be imposed in accordance with system criticality (ANSI/IEEE 1984).

Verification Should Include a Metrics Group: A metrics group is responsible for quantitative data collection, and the metric analysis and forecasting of the expected number of errors. This group defines useful metrics and uses them to forecast results with maximum effectiveness. A discussion of the types of metrics and data that are appropriate to collect are covered by Wilburn (1983a).

Verification Must Be Tailored to the Project: The criticality of the software project determines the amount of verification necessary. The decision as to how much verification should be used is basically one to be made by project management. Verification should, however, always be applied to critical areas of a particular piece of software.

Organization Standards and Guidelines are Required: For the verification process to proceed systematically, company standards and guidelines need to be developed to guide the development process. These standards and guidelines can either be developed in-house or by an organization such as IEEE.

Enforcement Is Required: The lack of enforcement of appropriate standards and guidelines on the earlier products of the life cycle make code verification difficult, time-consuming, and almost impractical.

SCM Is Required: A software configuration management (SCM) system (described in Chapter 9.0) is required that identifies and controls approved and implemented changes. It is vital that any changes found to be necessary to the verification process are correctly implemented.

A configuration control librarian is given the responsibility for ensuring that all development materials (such as the SRS and other products, tape and card decks, and program listings) are complete, current, and unaltered. Verification materials such as tools, test data, and test results are similarly controlled by SCM procedures and the configuration control librarian.

Accurate Records Must Be Kept: Many documents may be generated during the software life cycle that record verification activities. These documents include review reports such as the software requirements specification reviews, design reviews, and the verification readiness review; inspection reports that result from desk checks of software or other baseline documents; software verification reports that describe the tests that have been run on the system; and any data collected by a software metrics group.

Management Must Show Commitment: For verification to be an effective process in software development, management must be committed to the idea. Unless top management is committed to verification, there is little incentive for project management to follow verification practices. In fact, verification may be perceived as undesirable because of the additional short-term cost of verification efforts.

Management cannot assume that programmers know how to carry out software development and verification properly. Most inexperienced programmers and software developers tend to generate complex and poorly documented codes.

Therefore, training programs, in addition to appropriate software development standards, are required. In association, management incentives must be provided to project management and software developers to encourage use of these training programs.

10.1.3 Verification Methods Across the Software Life Cycle

The sections below describe how verification methods can be implemented in each phase of the life cycle. Appendix A of Wilburn's work (1983a) references more than 30 verification methods that can be used throughout the software life cycle. These verification methods are summarized below.

Requirements Specification

Software Requirements Analysis: Software requirements analysis is one of the most important verification methods because the derivation of formal specifications is one of the most error-prone of all programming activities. Requirements analysis is performed by the development team to ensure that each software requirement is completely and correctly defined. The checklist given in Appendix C (Wilburn 1982b) can also be used effectively in this analysis.

Unique Tagging of Requirements: The verification process throughout the software life cycle is substantially easier if each requirement is given a unique identification or tag.

Writing of Testable Requirements: An adequate verification process begins in the SRS activity with the writing of valid testable requirements. The verification should specify criteria that can be measured to determine whether they have been successfully met, rather than simply stating general requirements.

Use of Requirements Specification Languages: Many of the mistakes (defects) of the SRS can be eliminated by using better methods of problem definition, i.e., using specification languages. Languages such as SREM and PDL are being developed to address problem definition. The use of these languages makes each requirement more quantitative and testable, which, as noted above, is required for proper verification.

Use of Structured Methods: By systematically breaking down a complex problem into a number of intellectually simpler problems, solutions can be constructed for each "subproblem." These solutions are probably more correct and easily verifiable than those from the total problem. Similarly, because of these simpler problem pieces, tests can be generated more easily. This is the essence of the structured approach.

Model Verification: Part of the requirements definition phase in scientific and engineering software development is definition and incorporation of mathematical models to describe physical processes. To assure that these models are adequate, a model verification methodology should be incorporated. The following approach may be used:

1. establish the limits of the system inputs over which the model is believed to be valid due to approximations used in modeling and/or physical constraints

2. determine the variability in performance of similar software systems given the same inputs

3. establish prediction error tolerances for the software system being considered

4. run a simulation of the software system and establish acceptable bands around each simulation

5. superimpose any experimental data on simulation results

6. identify data points that fall outside the bands.

Functional Specification/Detailed Software Design

Several methodologies can be incorporated into the function specification/design phase of the software life cycle which will result in software that is easily verifiable during the software construction and software verification (or testing) activities, and which also will lead to higher quality software with less propensity to failure. Some of these methods are identified below.

Defensive Design: Defensive design is basically the use of design methodologies known to result in high quality software. Examples are use of appropriate standards and guidelines; use of design margins; design that anticipates defects; avoidance of intertwined control constructs; use of a hierarchical design structure; use of a program design language; and use of principles of modular design with coherent, cohesive modules.

Fault-Tolerant Design: The impact of program failures can be reduced most effectively during the design phase by first explicitly identifying assumptions whose violations would be critical to acceptable program operation. The designer should then specify how the program should behave if any of these assumptions are violated. Such a "fault-tolerant" design makes software continue to function successfully in spite of failures when faults occur.

Use of Structured Techniques: A higher quality and more easily verifiable product is usually achieved by applying approaches popularly known as structured techniques. The objective of these techniques is to reduce the complexity of the design and verification of the software by dividing the system into intellectually manageable components.

Completeness of Design Documentation: The form and completeness of design documentation are a significant part of the verification process. They determine the feasibility of 1) verifying that the design is consistent with and has satisfied the requirements, 2) performing consistency and completeness checks

within the design itself, 3) verifying the consistency of the code with the design, and 4) providing a more thorough testing of the code based on the design.

Threading of Design to SRS: Tracing and verifying requirements as they are interpreted into the design and then into code is a major problem. One way of tracking requirements is to note the driving requirement for each design element or section of code in the design representation, or as comments in the code or listing. A master requirements tracking document can summarize for each requirement the location of the related design elements or code sections.

Design Analysis: Design analysis ensures that the computer program design is correct and that it satisfies the defined software requirements. The first step in design analysis is to check for design completeness by correlating design elements with their source requirements. Techniques are then applied to verify design elements such as mathematical equations, algorithms, and control logic. Techniques for verifying the mathematical elements include independent derivation, dimensional analysis, and comparison to outside references. To verify certain algorithms, such as those for estimation and automatic control, simulation models are used to evaluate the algorithm's response to external stimuli. Control logic is more difficult to verify; it is best analyzed by determining the set of conditions for which the program must execute correctly, then manually analyzing the logic paths for each condition.

Coding and Software Generation

Many verification methods can be incorporated into the software coding and generation phase to improve quality, reduce error rate, and increase reliability. The following sections present some of these methodologies.

Team Efforts: Software is best developed by teams. An advantage of team development is that it can compensate for individual differences. A team can find defects overlooked by individual members in their own work and can keep the same problems from resurfacing. The exchange of information at team meetings keeps all members up to date on various problems.

Peer Review: Peer review is a technique of evaluating programs in terms of overall quality, maintainability, extensibility, and usability.

Coding Standards: The use of coding standards in the development of software permits reviewers to be on common ground when they are verifying a software module. If each software module throughout the project is formatted like every other, a reviewer will always be in familiar territory. A similar format expedites the review process and makes possible the relatively easy identification of errors in format and deficiencies. It is strongly recommended that a coding standard be utilized and developed for each software development project.

Self-Descriptive Programs: Self-descriptive programs incorporate documentation (whether it is design or requirements specifications) into the source program itself. Documentation internal to the program makes the verification

and testing easier, and is a powerful incentive for proper maintenance and an assurance that documentation will be accessible to the user. In the case of scientific software commentary which references the source of the equations, the models, and the logic are of great help to reviewers and users in verification and validation of the software or in establishing the adequacy or applicability of the software.

Code Analysis: Code analysis is performed to verify that the computer program, as coded, correctly implements the specified design. Code analysts examine the program's source language and its compiled or assembled object code using a variety of techniques. The equations and logic of the source language program are reconstructed, either manually or using automated aids, and compared to those specified in the design to identify errors made in translating the design into programming language. Violations of programming standards are also identified.

Assertions and Assertion Checkers: The use of assertions and associated assertion checkers come under the general heading of self-validating programs. The program is instrumented with dynamic assertions, and then usually a preprocessor is used to generate the appropriate code in the high level language that is being used to check the assertions during code operation. Assertions should be placed between statements such that every loop and every branch are cut by at least one assertion. Assertions are a claim that the stated relations hold at this point each time the program control reaches that point.

Parallel Design of Module Tests: An effective means of validation during software construction is to design the module tests in parallel with the construction of the module. When applying criteria to ensure that the module is effectively tested, logic errors will often become readily apparent to the developer.

Data Flow Analysis: If, in the design of a program module, each subroutine parameter is classified as input, output, or computational, data flow analyses can then be used to ensure that 1) all input variables are only referenced and never assigned values, and 2) all output values are always assigned a value along some path through the program. In data flow analysis, the goal is to trace the behavior of program variables as they are initialized or modified while the program executes. Data flow analysis is performed by associating at each node in the data flow graph values for the tokens that represent program variables, and by indicating whether the corresponding variable is referenced, unreferenced, or defined with the execution of the statement represented by that node. Some data flow analysis methods can be automated.

Code Instrumentation: Code instrumentation is inserted into the program solely to measure program characteristics. Knowledge of these characteristics can be useful for program verification. For medium-sized and large projects, tools can be acquired or developed to do instrumentation automatically. For small projects, the programmer can do his or her own instrumentation. Examples of the type of analyses that can be performed using code instrumentation include the following: auxiliary coding such as checking array boundaries,

checking loop control variables, determining if key data values are within permissible ranges, tracing the execution, and counting the number of times a group of statements is executed.

Static and Dynamic Analysis: Static analysis focuses on the form and structure of the programming module, but not on the functional or computational aspects. It detects classes of errors or error-prone constructs or anomalies. Dynamic analysis usually consists of a three-step process: 1) static analysis plus instrumentation of the program, 2) execution of the instrumented program, and 3) analysis of the instrumented data. Often this process is accomplished interactively through automated tools.

10.2 TESTING

Software testing is the final verification activity in the software development phases of the software life cycle and includes software unit, subsystem, and system testing. This activity should follow the procedures detailed in the software verification plan. The subject of software testing is very broad. Many books, reports, and papers have been written on this subject (see Adrion et al. 1981; Beizer 1983 and 1984; Branstad et al. 1980; Computer Program Testing 1981; Glass 1979; IEEE 1978, 1983f; ANSI/IEEE 1987; Infotech 1979a and 1979b; McCabe 1982; Myers 1976 and 1979; Powell 1982a and 1982b).

The objective of testing during software development is to provide assurance that the software performs as specified by its technical and operational requirements, which are detailed in the SRS and design documentation. Testing activities should be designed to assure that these objectives are achieved in an orderly, cohesive, clear, and controlled fashion. An effective SQA testing program must start with the requirements definition phase and address any testing performed throughout the software life cycle, including the operation and maintenance phases.

10.2.1 Planning

A test plan document should include (Lipow et al. 1977):

- a description of the purpose and scope of each level of testing to be conducted on each deliverable item or support item
- identification of the organization responsible for each level of testing
- identification and description of the pre- and post-test documentation to be generated for each level of testing, including test specifications, procedures, and logs
- test methods to be used to establish compliance (i.e., test by function or structure)

- identification and use of the support software and computer hardware to be used in testing

- test standards and quality criteria for acceptance to be employed.

10.2.2 Performance

The performance of testing should follow the developed test plan in detail, keeping appropriate records. Individual tests that are appropriate to specific cases can be designed using recommendations from the references identified in Section 10.2.

10.2.3 Review

The SQA plan should identify the activities for review of software testing which should include (U.S. DOD 1979):

- review of the software requirements to determine their testability

- review of the test plans and procedures for compliance with appropriate standards and satisfaction of contractual requirements

- review of the test requirements and criteria to be used to determine their adequacy, feasibility, and the satisfaction of the requirements specification

- monitoring of the test and certification processes to establish that the test results are indeed the actual findings

- review and certification of test reports

- assurance that test-related documentation is retained to allow repeatability of the tests.

Review procedures should follow the recommendations given in Chapter 7.0 and be incorporated into established milestones.

10.2.4 Acceptance Testing and Certification

Acceptance testing and certification are related to testing performed during software development. In fact, many tests used in acceptance testing are identical to those performed during development testing; however, acceptance testing/certification is more formalized than development testing.

Acceptance testing is defined as "formal testing conducted to determine whether a software system satisfies its acceptance criteria and to enable the customer to determine whether to accept the system" (Powell 1982b).

Formal testing includes the planning and execution of several kinds of test, (e.g., functional, volume, performance tests to demonstrate that the implemented software satisfies customer requirements for the software system.

Acceptance testing consists of three activities and many sub-activities. Reference should be made to Wallace (1986) for an overview of software acceptance testing and an extensive bibliography on the subject.

The first activity, test planning, determines, from the software requirements, what tests should be performed for each software function and what tests will exercise the entire computer program's functions or modules. An acceptance test plan is developed from these findings and the test procedures prepared to specify the actual acceptance tests in detail. Second, the acceptance testing is conducted to establish the proper execution of each software function. Third, analysis is performed to demonstrate that the integrated software has operated correctly in the use environment.

Performing adequate acceptance testing requires that each software requirement be identified according to some numerical or other scheme. This allows a specific requirement to be tested with an appropriate result, so that it can be recorded that the function was indeed performed correctly. Detailed, adequate testing can be expensive and time-consuming. However, in the long run, the time spent and the cost involved are justified. Certification of the software is indicated by signatures of the concerned parties that testify the software has indeed performed its functions as specified and is ready for operational use.

10.2.5 Operation/Maintenance Testing

During the course of day-to-day code operation the software system should be routinely tested, following the same procedures established in the test planning documentation described above in Section 10.2.1. The results should be compared with the original results which are to be maintained under configuration management. Such routine testing (especially after any maintenance activities or operating system changes) is known as regression testing and may identify either software degradation or hidden changes in the environment which compromise the validity of the software.

11.0 CONTROL OF SOFTWARE PROCUREMENT

Procured software generally consists of two types. The first type of software is developed specifically for a particular organization and is new code. This type is dealt with in Sections 11.1, 11.2, and 11.3. The second type of software is that which has been developed previously and is being provided "off the shelf" by the supplier (see Section 11.4).

It is essential that appropriate SQA requirements be imposed upon all suppliers of software to a nuclear utility (Lipow et al. 1977). This can be achieved by including appropriate supplier SQA requirements in the Request for Proposal and monitoring the supplier's conformance to these requirements.

11.1 REQUIREMENTS FOR THE SUPPLIER'S SQA PROGRAM

It is recommended that any organization supplying software to nuclear utilities have a defined SQA program. The supplier's SQA program must include the following:

- definition of a software life cycle with intermediate milestones
- commitment to specific documentation to be supplied to the user
- commitment regarding the level of detail to be contained in the documents
- established review procedures
- existence of a verification and validation effort
- identification of software development tools and techniques used in the effort
- system of software configuration management
- methods to provide assurance that the SQA program is actually being implemented as written.

The purchasing organization should evaluate its choice of suppliers (Lipow et al. 1977) based on the following considerations:

- the extent of and specific interactions between the software development organization (the developer) and the purchasing organization (user)
- description and assurance of implementation of the software life cycle utilized by the developer

- description and implementation of the developer's software problem reporting and corrective action processes
- description and implementation assurance of configuration control and management of the software throughout its life cycle
- developer's methods of assuring that the user's requirements for the software have been met
- documentation included in the software package delivered to the user.

Criteria for evaluating each area given above will be established by the purchasing organization and will be highly dependent on the end use of the software to be developed. Examples of questions to be addressed in the procurement process are provided in Appendix B.

11.2 AUDITING OF THE SUPPLIER'S SQA PROGRAM

The purchasing organization should audit the supplier organization to assure that each item that the supplier specified will be performed was performed adequately. The supplier's SQA plan and procedures for its implementation should be reviewed as well. At specified intervals, the supplier's control activities should be reviewed, including applicable records. These activities should establish that problems identified are corrected quickly and that the results of the corrective action are documented. Sufficient records should be maintained to demonstrate the effectiveness of the SQA program.

11.3 NONCONFORMANCE OF A SUPPLIER

Penalty clauses should be written into procurement documentation to enforce the conformance of the supplier (developer) to the specified SQA program. The penalty clauses should be strong enough to deter the supplier from deviating from the plan established when the contract was established. In this manner, preventative rather than punitive actions will be taken to assure compliance to the specified SQA program.

11.4 TRANSFER OF RESPONSIBILITY

Procured software typically enters the organization's life cycle at the operational phase, where responsibility for configuration management and code control is transferred to the buyer/user.

This approach to using software "off-the-shelf" has several disadvantages, including lack of control over the initial phases of the life cycle. The software package procured from outside suppliers must meet the same QA requirements as software designed within the organization. Verifying that software indeed meets the specified criteria for its code class relies on establishment that the design process for that software has been carried out in the structured,

systematic manner described in this document. Requisite documentation must be included as part of the delivered software package. Acceptance tests on the purchasing organization's computer must be planned, designed, and carried out in accordance with the software's requirements specifications.

After the code has been tested and/or verified on the purchasing organization's system, the software must be placed under configuration management. From that point forward, the code is handled and treated as software developed by the organization, and the software life cycle is implemented as described.

Another common situation is for facilities to purchase use of software via a "software clearinghouse." These companies provide read-only access to software used by the nuclear community. In this case, the clearinghouse places the software under configuration control and allows access to software on a contractual basis. However, this does not absolve the user facility of responsibility for controlling use of the software and knowing specific information about that software. For example, it is important to know version numbers of the software used to perform calculations, the dates they were run, and who ran the code. Furthermore, it is imperative that the purchasing organization have a systematic means of informing all past code users of updates, bugs that have been identified and fixed, and planned changes to the software. To do this, a contractual obligation must be established that requires the clearinghouse to inform the user organization of such conditions. Furthermore, the user facility must assure that someone is responsible for getting this information distributed to the appropriate people within the organization.

Purchasing off-the-shelf software does not absolve the user facility of responsibility for accuracy of calculational results, identification of software errors, and assessment of impacts caused by software errors identified by other users.

REFERENCES

ANS. 1982. Application Criteria for Programmable Digital Computer Systems of Nuclear Power Generating Stations. ANS/IEEE-7.4.3.2-1982, American National Standards Institute/Institute of Electrical and Electronic Engineers, Inc., New York.

ANSI/ANS. 1987. American National Standard Guidelines for the Verification and Validation of Scientific and Engineering Computer Programs for the Nuclear Industry. Approved Draft Standard, ANSI/ANS 10.4, American Nuclear Society, La Grange Park, Illinois.

ANSI/ANS. 1986. American National Standard Guidelines for the Documentation of Digital Computer Programs. ANSI/ANS 10.3-1986, American Nuclear society, La Grange Parke, Illinois.

ANSI/ANS. 1979. Guidelines for Considering User Needs in Computer Program Development. ANSI/ANS-10.5-1979, American Nuclear Society, LaGrange Park, Illinois.

ANSI/IEEE. 1987. Standard for Software Unit Testing. NASI/IEEE Standard 1008-1987, American National Standards Institute/Institute of Electrical and Electronic Engineers, Inc., New York.

ANSI/IEEE. 1984. IEEE Standard for Software Quality Assurance Plans. ANSI/IEEE Standard 730-1984, American National Standards Institute/Institute of Electrical and Electronics Engineers, Inc., New York.

ATC. 1983. A FORTRAN Coding Standard. Associated Technology Co., Estill Springs, Tennessee.

ATC. 1985. A Product Level Software Documentation Guide. Associated Technology Co., Estill Springs, Tennessee.

Adrion, W. Richard, Martha A. Branstad, and John C. Cherniavsky. 1981. Validation, Verification and Testing of Computer Software. NBS Special Publication NBS-SP-500-75, National Bureau of Standards, Washington, D.C.

Arthur, Jay. 1984. "Software Quality Measurement." Datamation, p. 115 (December issue).

Barikh, Girish. 1980. Techniques of Program and Systems Maintenance. Ethnotech Inc., Lincoln, Nebraska.

Beizer, Boris. 1983. Software Testing Techniques. Van Nostrand-Reinhold, New York.

Beizer, Boris. 1984. Software System Testing and Quality Assurance. Van Nostrand-Reinhold, New York.

Bersoff, E. H., V. D. Henderson, and S. E. Siegel. 1979a. "Software Configuration Management--A Tutorial." IEEE Computer 11(1):6.

Bersoff, E. H., V. D. Henderson, and S. E. Seigel. 1979b. "Attaining Software Product Integrity." In Proceedings of the Computer Software and Applications Conference 1979 (COMPSAC-79), p. 680. IEEE Computer Society Catalog 79CH1515-6C, Institute of Electrical and Electronics Engineers, Inc., New York.

Bersoff, E. H., V. D. Henderson, and S. E. Siegel. 1980. Software Configuration Management--An Investment in Product Integrity. Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Boehm, B. W., J. R. Brown, and N. Lipow. 1976. "Quantitative Evaluation of Software Quality." In Proceedings of the Second International Conference on Software Engineering, p. 592. IEEE Computer Society Catalog 76CH1125-4C, Institute of Electrical and Electronics Engineers, Inc., New York.

Boehm, Barry W. et al. 1978. Characteristics of Software Quality. North Holland Publishing Co., New York.

Boehm, B. W. 1976. "Software Engineering." IEEE Transactions on Computers C-25(12):1226-1241.

Boehm, B. W. 1979. "Software Engineering As it Is." In Proceedings of the Fourth International Conference on Software Engineering, p. 11. IEEE Catalog 79CH1479-5C, Institute of Electrical and Electronics Engineers, Inc., New York.

Branstad, Martha A., John C. Cherniavsky, and W. Richard Adrion. 1980. "Validation, Verification and Testing for the Individual Programmer." Computer 13(12):24.

Brown, John R. 1979. Programming Practices for Increased Software Quality." In Software Quality Management, p. 197. Petrocelli, New York/Princeton.

Bruce, P. and S. M. Pederson. 1982. The Software Development Project--Planning and Management. J. Wiley and Sons, Inc.

Buckley, F. J. and R. Poston. 1984. "Software Quality Assurance." In IEEE Transactions on Software Engineering SE-10(1):36. Institute of Electrical and Electronics Engineers, Inc., New York.

Carrow, J. C. 1976. "Structured Programming: From Theory to Practice." In Proceedings of the Second International Conference on Software Engineering, p. 370. IEEE Catalog 76CH1125-4C, Institute of Electrical and Electronics Engineers, Inc., New York.

Computer Program Testing. 1981. North Holland Publishing Co., Amsterdam, New York.

Cooper, John D. and Matthew J. Fisher, eds. 1979. Software Quality Management. Petrocelli, New York/Princeton.

DeMarco, Tom. 1982. Controlling Software Projects--Management Measurement and Estimation. Yourdon Press, Inc., New York.

Deutsch, Michael S. 1982. Software Verification and Validation--Realistic Approaches. Prentice-Hall Inc., Englewood Cliffs, New Jersey.

Doggett, R. B., Z. E. Carey, and N. P. Wilburn. 1983. Guidelines--Software Configuration Management. HEDL-TC-2263, Westinghouse-Hanford Co., Richland, Washington.

Dunn, Robert and Richard Ullman. 1982. Quality Assurance for Computer Software. McGraw-Hill, New York.

Enos, Judith L. and R. L. Van Tilburg. 1981. "Tutorial Series 5: Software Design." Computer 14(2):61.

Fairley, Richard E. 1985. Software Engineering Concepts. McGraw Hill Book Co., New York.

Fife, Dennis W. 1977. Computer Science and Technology: Computer Software Management, A Primer for Project Management and Quality Control. NBS Special Publication NBS-SP-500-11, National Bureau of Standards, Washington, D.C.

Fisher, Curt F. 1978. "Software Quality Assurance Tools: Recent Experience and Future Requirements." In Proceedings of the Software Quality Assurance Workshop, November 15-17, 1978, San Diego, California, p. 116. ACM, New York.

Foreman, J. J. 1980. "Implementing Software Standards." IEEE Computer 13(6):67.

Freedman, Daniel P. and Gerald M. Weinberg. 1979. Ethnotech Review Handbook. Ethnotech, Inc., Lincoln, Nebraska.

Fujii, Marilyn S. 1978. "A Comparison of Software Assurance Methods." In Proceedings of the Software Quality Assurance Workshop, November 15-17, 1978, San Diego, California, p. 27. ACM, New York.

Gane, Chris and Trish Sarson. 1977. Structured Systems Analysis: Tools and Techniques. McDonnell-Douglas Automation Co., St. Louis, Missouri.

Glass, Robert L. 1979. Software Reliability Guide Book. Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Glass, Robert L. 1981a. "Standards for Standards Writers." In Proceedings of Software Engineering Standards Applications Workshop (SESAW-I), San Francisco, California, p. 144. IEEE Computer Society Catalog 81CH1633-7, Institute of Electrical and Electronics Engineers, Inc., New York.

Glass, Robert L. and Ronald A. Noisex. 1981. Software Maintenance Guide Book. Prentice-Hall Inc., Englewood Cliffs, New Jersey.

Goodenough, John B. 1979. "A Survey of Program Testing Issues." In Research Directions in Software Technology, p. 316. MIT Press, Cambridge, Massachusetts.

Gustafson, G. G. and R. J. Kerr. 1982. "Some Practical Experience with a Software Quality Assurance Program." Communications of the ACM 25(1).

Holthouse, M. A. and S. G. Greenberg. 1978. "Software Technology for Scientific and Engineering Application." In Proceedings of the IEEE Computer Software and Applications Conference (COMPSAC 78), p. 814. IEEE Catalog 78CH1338-3C, Institute of Electrical and Electronics Engineers, Inc., New York.

Houghton, Raymond C., Jr. 1981. Features of Software Development Tools. NBS Special Publication 500-74, National Bureau of Standards, Washington, D.C.

Houghton, Raymond C., Jr. 1982. Software Development Tools. NBS Special Publication NBS-SP-500-88, National Bureau of Standards, Washington, D.C.

Houghton, Raymond C., Jr. 1983. "Software Development Tools: A Profile." Computer 16(5):63.

Houghton, Raymond C., Jr. and Karen A. Oakley. 1980. NBS Software Tools Data Base. NBS Special Publication NBS-IR-80-2159, National Bureau of Standards, Washington, D.C.

IEEE. 1978. Tutorial: Software Testing and Validation Techniques. IEEE Computer Society Catalog EH0138-8, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1979a. Tutorial: Automated Tools for Software Engineering. IEEE Computer Society Catalog EH0150-3, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1979b. Tutorial: Software Management. IEEE Computer Society Catalog EH0146-1, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1979c. Standard Computer Dictionary. IEEE Computer Society, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1980a. Tutorial: Software Configuration Management. IEEE Computer Society Catalog EH0169-3, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1980b. Tutorial on Software Design Techniques. IEEE Computer Society Catalog EH0161-0, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1983a. IEEE Standard Glossary of Software Engineering Terminology. IEEE Standard 729-1983, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1983b. IEEE Standard For Software Configuration Management Plan. IEEE Standard 828-1983, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1983c. SOFTFAIR: A Conference on Software Development Tools, Techniques, and Alternatives. IEEE Computer Society Catalog 83CH1919-0, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1983d. Tutorial JSP and JSD: The Jackson Approach to Software Development. IEEE Computer Society Catalog EH0206-3, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1983e. Tutorial on Software Maintenance. IEEE Computer Society Catalog EH0201-4, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1983f. IEEE Standard for Software Test Documentation. IEEE Standard 829-1983, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1984a. IEEE Guide to Software Requirements Specification. IEEE Standard 830-1984, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1984b. Model Program in Computer Science and Engineering. IEEE Computer Society, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1986a. Guide for Software Configuration Management. IEEE Computer Society Approved Draft Guide P1042, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1986b. Guide for Software Quality Assurance Plans. IEEE Guide 983-1986, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1986c. Standard for Software Engineering Standards Taxonomy. IEEE Computer Society Approved Draft Standard P1002, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1986d. A Standard for Measurements to Produce Reliable Software. IEEE Computer Society Draft Standard P982, Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE. 1986e. Standard for Software Verification and Validation Plans. IEEE Computer Society Approved Draft Standard P1012, Institute of Electrical and Electronics Engineers, Inc., New York.

Infotech. 1979a. Infotech State of the Art Report--Software Testing, Volume 1: Analysis and Bibliography. Infotech International Ltd., Berkshire, England.

Infotech. 1979b. Infotech State of the Art Report--Software Testing, Volume 2: Invited Papers. Infotech International Ltd., Berkshire, England.

Jackson, M. A. 1975. Principles of Program Design. Academic Press, New York.

Kastelein, J. E. 1971. Quality Assurance Requirements During Flight Software Development. TRW-SS-71-05, TRW Systems Group.

Kernighan, Brian W. and P. J. Plauger. 1978. The Elements of Programming Style. McGraw-Hill, New York.

Kerola, P. and P. Freeman. 1981. "A Comparison of Life Cycle Models." In Proceedings of the Fifth International Conference on Software Engineering, p. 90. IEEE Catalog 81CH1627-9, Institute of Electrical and Electronics Engineers, Inc., New York.

Lattanzi, L. D. 1979. "An Analysis of the Performance of a Software Development Methodology." In Proceedings of the Computer Software and Applications Conference, (COMPSAC 79), p. 7. Institute of Electrical and Electronics Engineers, Inc., New York.

Lipow, H., B. B. White, and B. W. Boehm. 1977. Software Quality Assurance, An Acquisition Guidebook. TRW-SS-77-07, TRW Systems Group.

McCabe, T. J. 1982. Structured Testing. IEEE Computer Society Catalog EH0200-6, Institute of Electrical and Electronics Engineers, Inc., New York.

McCall, James A. 1979. "An Introduction to Software Quality Metrics." In Software Quality Management, p. 127. Petrocelli, New York/Princeton.

Myers, Glenford J. 1976. Software Reliability Principles and Practices. Wiley-Interscience, New York.

Myers, Glenford J. 1979. The Art of Software Testing. Wiley-Interscience, New York.

NBS. 1974. "Guidelines for Automatic Data Processing Physical Security and Risk Management." Federal Information Processing Standards Publication 31, National Bureau of Standards, Washington, D.C.

NBS. 1976. "Guidelines for Documentation of Computer Programs and Automated Data Systems." Federal Information Processing Standard Publication 38, National Bureau of Standards, Washington, D.C.

NBS. 1979. "Guidelines for Automatic Data Processing Risk Analysis." In Federal Information Processing Standards Publication 65, National Bureau of Standards, Washington, D.C.

NBS. 1980. "Guidelines for Security of Computer Applications." Federal Information Processing Standards Publication 73, National Bureau of Standards, Washington, D.C.

NBS. 1981. Computer Model Documentation Guide. NBS Special Publication 500-73, National Bureau of Standards, Washington, D.C.

NBS. 1981. Proceedings of the NBS/IEEE/ACM Software Tool Fair. NBS Special Publication NBS-SP-500-80, National Bureau of Standards, Washington, D.C.

NBS. 1982. Proceedings of the NBS/FIPS Software Documentation Workshop. NBS Special Publication NBS-SP-500-04, National Bureau of Standards, Washington, D.C.

NBS. 1983. Guidance on Software Maintenance. NBS Special Publication NBS-SP-500-106, National Bureau of Standards, Washington, D.C.

Neumann, Albrecht J. 1982. Management Guide for Software Documentation. NBS Special Publication NBS-SP-500-87, National Bureau of Standards, Washington, D.C.

NRC. 1982. Inspection and Enforcement Manual-Computer Code Development and Use. USNRC Inspection Procedure 37998.

Osterweil, L. J. 1982. "TOOLPACK--An Experimental Software Development Environment Research Project." In Proceedings of Sixth International Conference on Software Engineering, p. 166. IEEE Computer Society Catalog 82CH1795-4, Institute of Electrical and Electronics Engineers, Inc., New York.

Peters, L. J. and L. L. Tripp. 1978. "A Model of Software Engineering." In Proceedings of the Third International Conference on Software Engineering, p. 63. IEEE Catalog 78CH1317-7C, Institute of Electrical and Electronics Engineers, Inc., New York.

Poston, R. M. 1982. "Software Quality Assurance Implementation." In Proceedings of Computer Software and Application Conference 1982 (COMPSAC-82), p. 356. IEEE Computer Society Catalog 82CH1810-1, Institute of Electrical and Electronics Engineers, Inc., New York.

Poston, R. M. 1984. "Determining a Complete Set of Software Development Standards." Software 1(3):87.

Poston, R. M. 1985. "Software Standards." IEEE Software 2(1):83.

Powell, Patricia B., ed. 1982a. Software Validation, Verification and Testing: Technique and Tool Reference Guide. NBS Special Publication NBS-SP-500-93, National Bureau of Standards, Washington, D.C.

Powell, Patricia B., ed. 1982b. Planning for Software Validation, Verification, and Testing. NBS Special Publication NBS-SP-500-98, National Bureau of Standards, Washington, D.C.

Reifer, Donald J. 1979a. "Software Quality Assurance Tools and Techniques." In Software Quality Management, p. 209. Petrocelli, New York/Princeton.

Riddle, W. E. and R. E. Fairley. 1980. Software Development Tools. Springer-Verlag, New York.

Ruder, Bryan and J. D. Madden. 1978. An Analysis of Computer Security Safeguards for Detecting and Preventing Intentional Computer Misuse. NBS Special Publication NBS-SP-500-25, National Bureau of Standards, Washington, D.C.

Shankar, K. S. 1977. "The Total Computer Security Problem: An Overview." IEEE Computer 10(6):50.

Steinauer, Dennis D. 1985. Security of Personal Computer Systems: A Management Guide. NBS Special Publication NBS-SP-500-120, National Bureau of Standards, Washington, D.C.

Sheron, B. W. and A. R. Rosztocsy. 1980. Report on Nuclear Industry Quality Assurance Procedures for Safety Analysis Computer Code Development and Use. NUREG-0653, U.S. Nuclear Regulatory Commission, Washington, D.C.

Tausworthe, Robert C. 1977. Standardized Development of Computer Software: Part I, Methods. Prentice-Hall Inc., Englewood Cliffs, New Jersey.

Tausworthe, Robert C. 1979. Standardized Development of Computer Software: Part II, Standards. Prentice-Hall Inc., Englewood Cliffs, New Jersey.

U.S. Department of Defense. 1979. Software Quality Assurance Program Requirements. Military Specification MIL-S-52779A, U.S. Department of Defense, Washington, D.C.

U.S. Department of Defense. 1985. Military Standard-Defense System Software Development. DOD-STD-2167, U.S. Department of Defense, Washington, D.C.

U.S. Nuclear Regulatory Commission. 1984. "Quality Assurance Criteria for Nuclear Power Plants and Fuel Reprocessing Plants." Appendix B of Code of Federal Regulations, Title 10, Energy; Part 50, Domestic Licensing of Production and Utilization Facilities. U.S. NRC, Washington, D.C. (10 CFR 50).

Wallace, D. R. 1986. An Overview of Computer Software Acceptance Testing. NBS Special Publication 500-136, U.S. Department of Commerce-National Bureau of Standards, Washington, D.C.

Wilburn, N. P. 1982a. Guidelines for Technical Reviews of Software Products. HEDL-TC-2132, Westinghouse-Hanford Co., Richland, Washington.

Wilburn, N. P. 1982b. Guidelines--Software Requirements Specification (SRS) Document Preparation. HEDL-TC-2159, Westinghouse-Hanford Co., Richland, Washington.

Wilburn, N. P. 1983a. Guidelines--Software Verification. HEDL-TC-2425, Westinghouse-Hanford Co., Richland, Washington.

Wilburn, N. P. 1983b. Standards and Guidelines Applicable to Scientific Software Life Cycle. HEDL-TC-2314, Westinghouse-Hanford Co., Richland, Washington.

Yourdon, Edward. 1978. Structured Walkthroughs, 2nd ed. Yourdon Press, Inc., New York.

Yourdon, Edward. 1979. Managing the Structured Techniques. Yourdon Press, Inc., New York.

Yourdon, Edward and Larry L. Constantine. 1978. Structure Design: Fundamentals of a Discipline of Computer Program and Systems Design. Yourdon Press, Inc., New York.

BIBLIOGRAPHY

ANSI/ANS. 1980. Standard Criteria for the Application of Programmable Digital Computer Systems in Safety Systems of Nuclear Power Generating Stations. ANSI/ANS-4.3.2, Proposed American National Standard, Draft 8, American National Standards Institute/American Nuclear Society, La Grange Park, Illinois.

Ackerman, A. F., A. S. Ackerman, and R. G. Ebenau. 1982. "A Software Inspection Training Program." In Proceedings of IEEE Computer Society's Sixth International Computer Software and Applications Conference 1982 (COMPSAC-82), p. 443. IEEE Computer Society Catalog 82CH1810-1, Institute of Electrical and Electronics Engineers, Inc., New York.

Barnes, Kate. 1985. "Doing It Yourself--A Blueprint for Training." PC Magazine, p. 147 (August 6 issue).

Basili, V. R. and B. T. Perricone. 1984. "Software Errors and Complexity: An Empirical Investigation." Communications of the ACM 27(1).

Basili, Victor R. and Robert W. Reiter, Jr. 1980. "Evaluating Automatable Measures of Software Development and Engineering." In Tutorial on Models and Metrics for Software Management, p. 280. IEEE Computer Society Catalog EH0167-7, Institute of Electrical and Electronics Engineers, Inc., New York.

Bryan, William L. and Stanley E. Siegel. 1984. "Product Assurance: Insurance Against a Software Disaster." Computer 17(4):75.

Buckley, F. J. 1982. Software Quality Assurance--A Tutorial. RCA Government Systems Division.

CSA. 1984. Software Quality Assurance Program. Canadian Standards Association Standard Q396.1, Canadian Standards Association.

Cain, J. T., G. G. Langdon, and M. R. Varanasi. 1984. "The IEEE Computer Society Model Program in Computer Science and Engineering." Computer 17(4):8.

Caveno, Joseph P. and James A. McCall. 1978. "A Framework for Measurement of Software Quality." In Proceedings of the Software Quality and Assurance Workshop, p. 133. ACM, New York.

Cooke, C. M. 1984. "Lessons from Implementing a Software QA Section." In Proceedings of Third Software Engineering Standards Application Workshop (SESAW-III), p. 68. IEEE Computer Society, Institute of Electrical and Electronics Engineers, Inc., New York.

Daughtrey, H. T., S. Y. Horn, and C. A. Schamp. 1984. "Independent Verification and Validation for Nuclear Plant Safety." In Proceedings of Third Software Engineering Standards Application Workshop (SESAW-III), p. 92. IEEE Computer Society, Institute of Electrical and Electronics Engineers, Inc., New York.

DeMarco, Tom. 1978. Structured Analysis and System Specification. Yourdon Press, Inc., New York.

EAI. n.d. EAI Software Quality Assurance. Electronic Associates, Inc.

EAI. 1977. EAI Software Methodology. Electronic Associates, Inc.

ESD. 1977. Software Acquisition Management Guidebook: Software Quality Assurance. ESD-TR-77-255 (NTIS AD-A047318), National Technical Information Service, Springfield, Virginia.

Endres, Albert. 1975. "An Analysis of Errors and Their Causes in System Programs." In IEEE Transactions on Software Engineering SE-1(2).

Fagan, M. E. "Design and Code Inspection to Reduce Errors in Program Development." In Tutorial on Structured Programming: Integrated Practices, p. 216. IEEE Computer Society Catalog EH0178-4, Institute of Electrical and Electronics Engineers, Inc., New York.

Foster, K. A. 1980. "Error Sensitive Test Case Analysis (ESTCA)." IEEE Transactions on Software Engineering SE-6(3):258.

Gannon, Carolynn 1979. "Error Detection Using Path Testing and Static Analysis." Computer 12(8):26.

Gannon, Carolyn. 1983. "Software Error Studies." In Proceedings of National Conference on Software Testing and Evaluation, p. I-1. National Security Industrial Association.

Glass, Robert L. 1981b. "Persistent Software Errors." IEEE Transactions on Software Engineering SE-7(2):162-168.

Greene, J. J., C. P. Hollocker, M. A. Jones, and T. C. Pingel. 1982. "Developing a Software Quality Assurance Program Based on the IEEE Standard 730-1981." In Proceedings of IEEE Computer Society's Sixth International Computer Software and Applications Conference 1982 (COMPSAC-82), p. 257. IEEE Computer Society Catalog 82CH1810-1, Institute of Electrical and Electronics Engineers, Inc., New York.

Guideline for Lifecycle Validation, Verification, and Testing of Computer Software, Federal Information Processing Standards Publication (FIPS PUB 101), U.S. Department of Commerce, National Bureau of Standards, June 6, 1983.

Howden, W. E. 1981. "Errors, Design Properties and Functional Program Tests." In Computer Program Testing, p. 115. North Holland Publishing Co., Amsterdam, New York.

Huang, J. C. 1977. "Error Detection Through Program Testing." In Current Trends in Programming Methodology--Volume 2, Program Validation, p. 16. Prentice-Hall Inc., Englewood Cliffs, New Jersey.

Keefe, Patricia. 1979. "Quality Assurance Shows Top Growth in Corporate DP." Computer World, p. 17 (January 16 issue).

Malsbury, J. 1983. "Educational Support for the Standards Process." In Second Software Engineering Standards Application Workshop (SESAW-II), p. 119. IEEE Computer Society Catalog 83CH1884-6, Institute of Electrical and Electronics Engineers, Inc., New York.

McGill, J. P. 1984. "The Software Engineering Shortage." IEEE Transactions on Software Engineering SE-10(1):42.

Meekel, J. and R. Troy. 1984. "Comparative Study of Standards for Software Quality Assurance Plans." In Software Engineering Standards Workshop (SESAW-III). IEEE Computer Society, Institute of Electrical and Electronics Engineers, Inc., New York.

Miller, Edward. 1979. "Software Quality Assurance." Computer 12(8):7.

Mills, H. D. 1979. "Software Development." In Research Directions in Software Technology, p. 87. MIT Press, Cambridge, Massachusetts.

Mills, H. D. 1980. "Software Engineering Education." Proceedings IEEE 68(9):1158.

Mizuno, Yukio. 1983. "Software Quality Improvement." Computer 16(3):66.

Neumann, A. J. 1982. Management Guide for Software Documentation. NBS Special Publication 500-87, U.S. Department of Commerce, National Bureau of Standards, Washington, D.C.

NRC Inspection and Enforcement Manual: Inspection Procedure 37998 - "Computer Code Development and Use."

Orr, Ken. 1981. Structured Requirements Definition. Ken Orr and Associates, Inc., Topeka, Kansas.

Osborne, W. M. 1986. Executive Guide to Software Maintenance. NBS Special Publication 500-130, U.S. Department of Commerce, National Bureau of Standards, Washington, D.C.

Osterweil, L. J. and L. D. Fosdick. 1978. "DAVE--A Validation Error Detection and Documentation System for FORTRAN Programs." In Tutorial: Software Testing and Validation Techniques, p. 473. IEEE Catalog EH0138-8, Institute of Electrical and Electronics Engineers, Inc., New York.

Perry, William F. 1981. Effective Methods of EDP Quality Assurance. QED Information Sciences, Inc., Wellesley, Massachusetts.

Poetschat, G. R. 1981. "Review of ANS-10 Standards and Activities." In Proceedings of the International Topical Meeting on Advances in Mathematical Methods for the Solution of Nuclear Engineering Problems, p. 567. Munich, Federal Republic of Germany.

Powell, P. B., ed. 1982. Planning for Software Validation, Verification, and Testing. NBS Special Publication 500-98, Computer Science and Technology, National Bureau of Standards, U.S. Department of Commerce, Washington, D.C.

Powell, P. B., ed. 1982. Software Validation, Verification and Testing Technique and Tool Reference Guide. NBS Special Publication 500-93, National Bureau of Standards, Washington, D.C.

"QA Survey Results: Views Vary Significantly." 1985. Government Computer News, p. 22 (April 26 issue).

Raskin, Robin. 1985. "Individual Training: A Matter of Style." PC Magazine, p. 121 (August 6 issue).

Reifer, Donald J. 1979b. "The Software Engineering Checklist." In Tutorial: Software Management, p. 70. IEEE Computer Society Catalog EH0146-1, Institute of Electrical and Electronics Engineers, Inc., New York.

Rice, John R. 1979. "Software for Numerical Computation." In Research Directions in Software Technology, p. 688. MIT Press, Cambridge, Massachusetts.

Roderique, G., E. D. Giroux, and M. Pratt. 1980. "Perspectives on Large-Scale Scientific Computation." Computer 13(10):65.

Schneiderman, Ben. 1980. Software Psychology: Human Factors in Computer and Information Systems. Winthrop Publishers Inc., Cambridge, Massachusetts.

Schneidewind, N. S. and H. M. Hoffman. 1979. "An Experiment in Software Data Collection and Analysis." IEEE Transactions on Software Engineering SE-5(3):276.

Scholten, Roger W. 1977. "Software Quality Assurance." In Proceedings Western Regional Conference ASQC, Seattle, Washington.

Shen, V. Y., T. J. Yu, S. M. Thebaut, and L. R. Paulsen. 1985. "Identifying Error-Prone Software--An Empirical Study." In IEEE Transactions on Software Engineering SE-11(3):302.

Sheron, B. W. and Z. R. Rosztoczy. 1980. Report on Nuclear Industry Quality Assurance Procedures for Safety Analysis, Computer Code Development and Use. NUREG-0653, U.S. Nuclear Regulatory Commission, Washington, D.C.

Shooman, M. L. and M. I. Bolsky. 1975. "Types, Distribution, and Test and Correction Times for Programming Errors." In Proceedings of the International Conference on Reliable Software, p. 347. IEEE Computer Society, Institute of Electrical and Electronics Engineers, Inc., New York.

Silling, S. A. 1983. Final Technical Position on Documentation of Computer Codes for High-Level Waste Management, U.S. Nuclear Regulatory Commission Report NUREG-0856, June 1983.

Standard for Software Quality Assurance Plans, Institute of Electrical and Electronic Engineers (IEEE) Standard 730-1984.

Straker, E. A. 1985. "Software Quality--How Is It Achieved?" In Proceedings of Nuclear Power Plant Safety Control Technology Seminar.

Thomas, Nina C. and Henry L. Reeves, Jr. 1980. "Experience from Quality Assurance in Nuclear Power Plant Protection System Software Validation." IEEE Transactions on Nuclear Science NS-27(1).

Tice, George D., Jr. 1980. "Software Quality Control--Roadbed for the Bullets." In Proceedings 1980 ASQC Western Regional Conference, Seattle, Washington.

WHC. 1980. Computer Software Development and Use. Hanford Engineering Development Laboratory Quality Assurance Bulletin 79-3, Westinghouse-Hanford Co., Richland, Washington.

Wallace, D. R., and J. C. Cherniavsky. 1987. Report on the NBS Sortware Acceptance Test Workshop--April 1-2, 1986. NBS Special Publication 500-146, U.S. Department of Commerce, National Bureau of Standards, Washington, D.C.

Wasserman, A. I. and Peter Freedman. 1980. "Software Engineering Education: Status and Prospects." In Tutorial on Software Design Techniques, 3rd ed., p. 445. IEEE Computer Society Catalog EH0161-0, Institute of Electrical and Electronics Engineers, Inc., New York.

Westermeier, J. T., Jr. 1979. "Nuclear Near-Disaster." Data Management, p. 30 (June issue).

Wilkinson, G. F. 1985. Quality Assurance Plan for Computer Software Supporting the U.S. Nuclear Regulatory Commission's High-Level Waste Management Program, Sandia National Labortory [DRAFT] Report NUREG/CR-4369 (SAND85-1774), September 1985.

APPENDIX A

CRITERIA FOR ASSESSING SOFTWARE QUALITY

APPENDIX A

CRITERIA FOR ASSESSING SOFTWARE QUALITY

ACCOUNTABILITY	The ability to measure use of computer resources by a module or program. Critical segments of code can be instrumented with probes to measure timing, to determine whether specified branches are exercised, etc. Codes or subroutines used for probes are preferably invoked by conditional assembly or compilation.
ACCURACY	The extent that the code's outputs are sufficiently precise to satisfy their intended use.
AUGMENTABILITY	The extent that the code can easily accommodate expansion of computational functions within components or data storage requirements.
COMMUNICATIVENESS	The extent that the form and content of the code's inputs and outputs facilitate assimilation, usefulness, and understanding. Communicativeness also includes those attributes of the software that provide standard protocols and interface routines required to couple the system with another independent system.
COMPLETENESS	The extent that the code's required functions are present and fully developed. External reference documents must be available and the required functions coded and present as designed.
CONCISENESS	The absence of redundant or excessive coding and the assurance that the required functions are implemented with a minimum amount of coding. Conciseness implies that the program is not excessively fragmented into modules, overlays, functions, and subroutines; and that the same sequence of coding is not repeated in numerous places (rather than defining a subroutine or macro).
CONSISTENCY	The extent that the code contains uniform notation, terminology, comments, symbology, and implementation techniques. Internal consistency implies that coding standards are uniformly adhered to; e.g., comments are not unnecessarily wordy in one place, while being scanty at another; the number of arguments in subroutine calls match with the subroutine header, etc.

	External consistency refers to the extent that the code's contents are traceable and conform to the requirements and design. External consistency implies that variable names and definitions, including physical units, are consistent with a glossary, or that there is a one-to-one relationship between functional flowchart entities and coded routines or modules.
CORRECTNESS	The ability of the software to produce specific outputs when given the specific inputs, and the extent to which a program satisfies its specifications and fulfills the user's mission.
DEVICE EFFICIENCY	The extent that the operations, functions, or instructions provided by the code are performed without waste of computer resources (CPU time, I/O channel capacity, core memory, etc.). Thus a program may be efficient with respect to one device (e.g., CPU time) but not to another (e.g., core memory), implying that it is not efficient with respect to the overall set of resources it employs.
DEVICE INDEPENDENCE	The ability of the code to be unaffected by changes to the computer hardware or peripheral equipment. For independence, coding directly related to a specific hardware device should be minimized, isolated, and identified.
EFFICIENCY	The extent to which the code performs its required functions without waste of resources. Choices of source code construction must be made to produce the minimum number of words of object code; where alternate algorithms are available, those taking the least time should be chosen; information-packing density in the core should be high, etc.
ERROR HANDLING CAPABILITY	The code's ability to handle errors due to hardware or software failures in a way that the resulting system performance degrades gracefully rather than catastrophically.
HUMAN ENGINEERING	The extent that the code fulfills its purpose without wasting the users' time and energy or degrading their morale. Inputs and outputs should be self-explanatory, understandable, unambiguous, and designed to avoid misinterpretation. This attribute implies robustness and communicativeness.
INTEGRITY	The extent to which access to software or data by unauthorized persons can be controlled.

INTEROPERABILITY	The effort required to couple the code system to another independent code system.
MAINTAINABILITY	The extent that the code facilitates updating to satisfy new requirements, to correct deficiencies, or to move to a similar computer system. This implies that the code is understandable, testable, and modifiable.
MODIFIABILITY	Characteristics of the design and implementation of the code that facilitate incorporation of changes, once the nature of the desired change has been determined.
PORTABILITY	The extent that the code can be operated easily and well on computer configurations other than its current one.
READABILITY	The extent that the code's function and design can be easily understood by reading (e.g., complex expressions having mnemonic variable names and parentheses even if they are unnecessary).
RELIABILITY	The extent that the code can be expected to perform its intended functions satisfactorily under normal conditions. In a reliable system, abnormal conditions may cause degraded performance but will not result in erroneous performance masked as correct performance. Reliability implies that the program will compile, load, and execute, producing answers of the requisite accuracy; and that the program will continue to operate correctly, except for a tolerably few instances, while in use. Reliability also implies that the code is complete and externally consistent.
REUSABILITY	The extent to which a program or its pieces can be used in other applications. Reusability is related to the packaging and scope of the functions that programs perform.
ROBUSTNESS	The extent that the code can continue to perform despite some violation of the assumptions in its specification. Robustness implies, for example, that the program will handle inputs or intermediate calculated variables that are out of range or in different format or type than specified, without degrading the performance of functions not dependent on the inputs or variables.
SELF-CONTAINEDNESS	The extent that the code performs its explicit and implicit functions within itself. Examples of implicit functions are initialization, input checking, and diagnostics.

SELF-DESCRIPTIVENESS	The extent that the code listing contains enough information for a reader to determine or verify its objectives, assumptions, constraints, inputs, outputs, components, and revision status. Commentary and traceability of previous changes by transforming previous versions of code into nonexecutable but available code provide for self-descriptiveness. Self-descriptiveness is necessary for both testability and understandability.
SIMPLICITY	Implementation of functions in the most understandable manner, usually requiring avoidance of practices that increase complexity.
STRUCTUREDNESS	The extent that the code possesses a definite pattern within its interdependent parts. Structuredness implies that the program design has proceeded in an orderly and systematic manner, that standard control structures have been followed in coding the program, etc.
TESTABILITY	The extent that the code facilitates the establishment of test plans, designs, procedures, and implementation.
TRACEABILITY	Those attributes of software that provide a thread from requirements to implementation, with respect to the specific development and operational environment.
UNDERSTANDABILITY	The extent that the code's functions are clear to the reader. Understandability implies that variable names or symbols are used consistently, modules of code are self-descriptive, and the control structure is simple or in accordance with a prescribed standard. The program should contain no hidden meanings or operating characteristics that come to light only after months of use.
USABILITY	The effort required to learn, operate, prepare input, and interpret output of a program.

APPENDIX B

EXAMPLE QUESTIONS TO BE ADDRESSED FOR PROCURED SOFTWARE

APPENDIX B

EXAMPLE QUESTIONS TO BE ADDRESSED FOR PROCURED SOFTWARE

DEVELOPER-USER INTERACTIONS

- To what extent does the user participate in the following steps of the software development process:
  - software requirements specification?
  - software design specification?
  - software verification/validation?
  - reviews and audits?
- After transfer of the software to the user, what further obligations/responsibilities reside with the developer? Who in the user organization will monitor these actions to assure performance?

SOFTWARE LIFE CYCLE

- Does the developer ascribe to and actually utilize a software life cycle for software development?
- If so, what is the life cycle, and does it meet the user's needs and requirements?
- What assurance does the developer provide to the user that the life cycle has been implemented as represented; i.e., what traceability is provided?

PROBLEM REPORTING/CORRECTIVE ACTION

- How does the developer report software problems and corrective actions to the user?
- Conversely, how does the user report problems to the developer?
- What positions within each organization represent the point of contact for problem reporting?
- Will the developer/supplier address the magnitude of problems, corrective actions, and possible consequences, or are these the responsibilities of the user?

CONFIGURATION CONTROL AND MANAGEMENT

- What methods of configuration control and configuration management are used by the developer?
- When are new versions of software issued by the developer? Does the user receive updated versions, or must they be purchased?
- Does the user receive a read-only version of the software?
- How is the source code protected and stored?

VERIFICATION AND VALIDATION OF THE SOFTWARE

- How is verification and/or validation (V and/or V) performed?
- What documentation is provided on the V and/or V process?
- To what extent has V and/or V been systematically conducted; i.e., is V and/or V equated with acceptance testing by the developer?

DOCUMENTATION

- What documentation is included along with the software?
  - User's manual?
  - Theory and algorithms used?
  - V and/or V documentation?
  - Requirements and design specifications?
  - Acceptance tests?
  - Other?

APPENDIX C

SRS REVIEW CHECKLIST

APPENDIX C

SRS REVIEW CHECKLIST[a]

A. Is the software requirements specification (SRS) in conformance with the SRS documentation guideline (Wilburn 1982b) and any other company guidelines?

1. Does a formal SRS document exist?
2. Are the necessary sections present?
3. Does each section contain the required information?
4. Is the SRS document in the recommended format?
5. Does the SRS conform to documentation guidelines?
6. Are the technical requirements in concurrence with administrative and contract requirements?

B. Does the SRS reflect an understanding of the problem to be solved?

1. Are the requirements consistent with the Statement of Work for the program?
2. Are the models that are specified appropriate for the problem being solved?
3. Are the numerical techniques that are specified appropriate for the problem being solved?
4. Are the algorithms that are specified appropriate for the problem being solved?
5. Have the program functions been partitioned in a manner consistent with the problem to be solved?
6. Will the program, as specified, solve the problem?
7. Are the equations scientifically correct and consistent with the requirements?

(a) As adapted from Wilburn 1983a.

8. Is the full scope of software development understood, and are problem areas explicitly noted?

9. Is the operational environment correctly understood?

C. Are the requirements complete?

1. Are the ultimate software products completely defined and is adequate documentation required?

2. Are documentation standards established for all deliverable and nondeliverable software?

3. Is all software to be used, identified?

4. Are system startup, restart, and batch or interactive program execution procedures identified?

5. Are user requirements addressed?

6. Are human engineering requirements and problem areas identified?

7. Are goals for the software identified?

8. Have the expected level of change in the system and the time required to implement changes been considered?

9. Is each functional requirement explicitly, quantitatively, and testably defined in terms of inputs, processing, outputs, data requirements, interfaces, accuracy, timing, exception handling, constraints, and performance?

10. Are the requirements mapped from system specifications into correct software requirements specifications?

11. Are the software requirements identified? If not, are possible approaches described well enough so that possible software requirements are indicated?

12. Do the requirements include the functions implied by the Statement of Work?

Input/Output

1. Are display contents and layouts described?

2. Are program inputs identified and described to the extent needed to design the program?

3. Are required program outputs identified and described to the extent needed to design the program?

4. Does the SRS include required behavior in the face of improper inputs and other anomalous conditions?

Data

1. Are procedures identified for purging and updating data bases?

2. Are data security and protection against data loss provided for?

3. Are the logic data base and its access mechanisms defined?

4. Is each entity or relationship that is mentioned in the requirements also defined in the data dictionary, and vice versa?

5. Are requirements specified for security, accuracy, who requires access to the information, and how quickly it is needed?

Interfaces

1. Are the person-machine interfaces and operational procedures clearly defined?

2. Is adequate attention given to both the hardware-to-software and software-to-software interfaces?

3. Is conformance with system accuracy control and interface control specifications (i.e., other equipments, operators, other software and data/data bases) stated?

4. Are external system interface definitions accurate and complete?

5. Are operational interfaces with the computer program, including both hardware and software, identified, or are there references to the specifications that define those interfaces?

6. Are applicable nonoperational interfaces that are related to computer program support and code generation identified, such as specific programming language, compiler, data base management system, loaders, other utility programs, or unique support hardware? Are references to appropriate documentation of these interfaces identified?

7. Do the requirements identify external interfaces and fully specify required program behavior with respect to each?

Performance

1. Are performance requirements for each function described in separate paragraphs? Do these paragraphs include source and type of inputs, and destination and type of outputs?

2. Are requirements for system resource margins adequately specified?

Error Processing

1. Are provisions made for transition to degraded or manual modes if the system or subsystem fails?

2. Are adequate provisions made for system backup and redundancy?

3. Are the software and hardware diagnostic capabilities adequate?

4. Is error processing logic described; i.e., does the software indicate improper, incorrect, or out-of-range inputs?

Environment

1. Are the software tools required for development, testing, and maintenance of the software described, and are they deliverables?

2. Does the SRS describe the operational environment into which the program must fit?

3. Do the specifications tell what the computer program must do, how well, and under what conditions, and do they describe the environment in which it is to operate?

4. Have software support and modification requirements been initially identified?

5. Have support tools, facilities, and recruitment and training of support personnel been addressed?

Constraints

1. Are explicit limits for the system (i.e, what it should and should not do) defined, and are constraints identified?

2. Are the volume and throughput expectations for the system identified?

3. Are the system protection and security requirements identified?

4. Does the SRS include applicable timing and sizing constraints?

D. **Are the requirements correct?**

1. Are the requirements consistent with the program's Statement of Work?

2. Are the requirements consistent with documented descriptions and known properties of the operational environment into which the program must fit?

3. Do interface requirements agree with document descriptions and known properties of the external interfacing elements?

4. Do input requirements correctly describe inputs whose format, content, data rate, etc. are not at the discretion of the designer?

5. Do output requirements correctly describe outputs whose format, content, data rate, etc. are not at the discretion of the designer?

6. Do requirements concerning models, algorithms, and numerical techniques agree with standard references, where applicable?

7. Do the project manager and user management have any differences over interpretation of the requirements?

E. **Are the requirements consistent?**

1. Is the SRS free of internal contradictions?

2. Are the models, algorithms, and numerical techniques that are specified mathematically compatible?

3. Are input and output formats consistent, to the extent possible?

4. Are the requirements for similar or related functions consistent?

5. Are the accuracies required of inputs, computations, output, etc. compatible?

6. Are the style of presentation and the level of detail consistent throughout the document?

7. Is the mapping of software requirements from the system specifications consistent, complete, and accurate?

8. Are software system limits and capacities compatible with the system specification?

9. Are vertical and horizontal consistency and compatibility achieved between requirements?

10. Are system availability requirements consistent with the system's intended operation, and will they require reasonably priced hardware or software?

11. Is any information entity defined more than once unnecessarily (i.e., is there redundancy)?

12. Can each input entity be related to a source of information?

13. Is each input entity related to a derived entity or an output entity?

14. Is each output entity related to a derived or input entity?

F. Are all requirements clear and unambiguous?

1. Can all requirements be adequately interpreted?

2. Can all requirements be interpreted in only one way?

3. Are the requirements sufficiently detailed to prevent misinterpretation?

4. Are the requirements organized and presented in a way that promotes clarity (for example, use of tables and lists in place of text, where applicable)?

5. Does the SRS differentiate between program requirements and other information provided in the specifications?

6. Are the data base and data requirements clearly stated?

7. Are the requirements for software structure, etc., clearly stated?

8. Are requirements stated singularly, clearly, and concisely?

9. Are the performance requirements stated in a manner that will support unambiguous design and test?

10. Is each definition and description of an entity or relationship understandable and consistent with requirements specification guidelines?

11. Are system/subsystem limitations and restrictions clearly stated?

G. Are the requirements feasible?

1. Is the necessary technology fully developed and are the approaches to its utilization mature?

2. Are the specified models, algorithms, and numerical techniques state of the art?

3. Can the models, algorithms, and numerical techniques be implemented within the constraints imposed on the system and on the development effort?

4. Are the quality attributes specified for the program achievable, when viewing the program both by its parts and as a whole?

5. Are the required functions attainable within the available resources?

6. Is the hardware available?

7. Is the hardware of sufficient size to perform debugging?

8. Are adequate development and test facilities available?

9. Are hardware/firmware/software tradeoffs sufficiently discussed, and are adequate flexibility and growth potential retained in the result?

H. Does the SRS contain adequate provision for program verification and acceptance?

1. Are requirements stated in testable terms?

2. Are acceptance criteria specified for each requirement?

3. Can quantitative terms (e.g., ranges, accuracies, tolerances, rates, boundary values, and limits) used in stating requirements be recorded as evidence of satisfaction?

4. Are the acceptance criteria consistent with use of any of the following:

   - results obtained from similar computer programs
   - classical solutions
   - accepted experimental results
   - analytical results published in the technical literature
   - benchmark problems?

5. Is the source of each requirement or derivation shown, and are all requirements traceable to or derived from a higher level specification?

I. Does the SRS avoid placing undue constraints on program design and implementation?

1. Is there justification for including in the SRS constraints in the SRS on design or implementation?

2. Is the specification limited to defining requirements, and how will it be done?

J. Does the SRS have sufficient quality requirements?

1. Does the SRS include the desired quality requirements (e.g., requirements for performance, reliability, accuracy, portability, maintainability, user friendliness)?

2. Are the factors that lead to quality and reliable software sufficiently well defined? Are modularity, structuredness, descriptiveness, consistency, simplicity, expandability, testability, device independence, robustness/integrity, and accessibility required?

NRC FORM 335 (2-84) NRCM 1102, 3201, 3202

U.S. NUCLEAR REGULATORY COMMISSION

BIBLIOGRAPHIC DATA SHEET

SEE INSTRUCTIONS ON THE REVERSE

1. REPORT NUMBER (Assigned by TIDC, add Vol. No., if any)
NUREG/CR-4640
PNL-5784

2. TITLE AND SUBTITLE
Handbook of Software Quality Assurance Techniques Applicable to the Nuclear Industry

3. LEAVE BLANK

4. DATE REPORT COMPLETED
MONTH: August YEAR: 1987

5. AUTHOR(S)
J. L. Bryant
N. P. Wilburn

6. DATE REPORT ISSUED
MONTH: August YEAR: 1987

7. PERFORMING ORGANIZATION NAME AND MAILING ADDRESS (Include Zip Code)
Pacific Northwest Laboratory
Richland, Washington 99352

8. PROJECT/TASK/WORK UNIT NUMBER

9. FIN OR GRANT NUMBER
FIN P2002

10. SPONSORING ORGANIZATION NAME AND MAILING ADDRESS (Include Zip Code)
Division of Licensee Performance and Quality Evaluation
Office of Nuclear Reactor Regulation
U.S. Nuclear Regulatory Commission
Washington, DC 20555

11a. TYPE OF REPORT
Technical

b. PERIOD COVERED (Inclusive dates)

12. SUPPLEMENTARY NOTES

13. ABSTRACT (200 words or less)

Pacific Northwest Laboratory is conducting a research project to recommend good engineering practices in the application of 10 CFR 50, Appendix B requirements to assure quality in the development and use of computer software for the design and operation of nuclear power plants for NRC and industry. This handbook defines the content of a software quality assurance program by enumerating the techniques applicable. Definitions, descriptions, and references where further information may be obtained are provided for each topic.

14. DOCUMENT ANALYSIS – a. KEYWORDS/DESCRIPTORS

Software Quality Assurance in the Nuclear Industry

b. IDENTIFIERS/OPEN-ENDED TERMS

Computer Software, Software Quality Assurance, Nuclear Industry Appendix B Criteria, Software Life Cycle, Software Verification and Validations, Configuration Management and Code Control.

15. AVAILABILITY STATEMENT
Unlimited

16. SECURITY CLASSIFICATION
(This page) Unclassified
(This report) Unclassified

17. NUMBER OF PAGES

18. PRICE

UNITED STATES
NUCLEAR REGULATORY COMMISSION
WASHINGTON, D.C. 20555

OFFICIAL BUSINESS
PENALTY FOR PRIVATE USE, $300

SPECIAL FOURTH-CLASS RATE
POSTAGE & FEES PAID
USNRC
WASH. D.C.
PERMIT No. G-67

TO THE NUCLEAR INDUSTRY