NASA-TM-109706

(NASA-TM-109706) SOFTWARE ASSURANCE GUIDEBOOK (NASA) 55 p

N94-71748

Unclas

Z9/61 0001766

SOFTWARE ASSURANCE GUIDEBOOK

SEPTEMBER 1989

Office of Safety, Reliability, Maintainability,
and Quality Assurance
Software Management and Assurance Program (SMAP)
Guidebook Working Group

SOFTWARE ASSURANCE GUIDEBOOK

Approvals

Lawrence E. Hyatt 3 Aug 89

Lawrence E. Hyatt
Chairman, Software Assurance
Guidebook Working Group

Date

Donald W. Sova 8/8/89

Donald W. Sova
Manager, SMAP

Date

PREFACE

The growth in cost and importance of software has made it necessary for NASA to establish software standards and guidance for use in the development and acquisition of software. The Software Management and Assurance Program (SMAP), established in the Office of Safety, Reliability, Maintainability, and Quality Assurance of NASA Headquarters, focuses the NASA activities in defining standards for software management, engineering, and assurance. One of the products of the SMAP is a series of guidebooks that define a NASA concept of the processes that are used to manage, engineer, and assure software. This is the Software Assurance Guidebook which describes software assurance in a way that is compatible with practices at NASA Centers.

TABLE OF CONTENTS

Section		Page
	PREFACE	
I.	**OVERVIEW**	1
	A. Concepts and Definitions	1
	B. Goals of Software Assurance	1
	C. Purpose of this Guidebook	2
	D. Changes to this Guidebook	2
II.	**ESTABLISHING A PROJECT SOFTWARE ASSURANCE ACTIVITY**	3
	A. Concepts and Definitions	3
	B. Tailoring Software Assurance to the Project	3
	C. Creating the Software Assurance Plan	4
	D. Project Structure Considerations	4
	E. Completion Criteria	5
	F. Implementation of the Software Assurance Plan	5
	G. Sources of Help	6
	H. Summary	6
III.	**SOFTWARE QUALITY ASSURANCE**	7
	A. Concepts and Definitions	7
	B. Standards and Procedures	7
	C. Software Quality Assurance Activities	8
	D. SQA Relationships to Other Assurance Activities	8
	1. Configuration Management Monitoring	8
	2. Verification and Validation Monitoring	9
	3. Formal Test Monitoring	10
	E. Software Quality Assurance During the Software Acquisition Life Cycle	11
	1. Software Concept and Initiation Phase	11
	2. Software Requirements Phase	11
	3. Software Architectural (Preliminary) Design Phase	11
	4. Software Detailed Design Phase	12
	5. Software Implementation Phase	12
	6. Software Integration and Test Phase	12
	7. Software Acceptance and Delivery Phase	13
	8. Software Sustaining Engineering and Operations Phase	13
	F. Techniques and Tools	13

TABLE OF CONTENTS (Cont.)

Section		Page
IV.	**SOFTWARE QUALITY ENGINEERING**	15
	A. Concepts	15
	B. Software Qualities	15
	1. Reliability	15
	2. Maintainability	15
	3. Transportability	16
	4. Interoperability	16
	5. Efficiency	16
	C. Metrics	16
	D. A Software Quality Engineering Program	16
	1. Qualities and Attributes	16
	2. Quality Evaluations	17
	3. Nonconformance Analysis	17
	4. Fault Tolerance Engineering	18
	E. Techniques and Tools	18
V.	**VERIFICATION AND VALIDATION**	21
	A. Concepts and Definitions	21
	B. Activities	21
	1. Reviews, Inspections, and Walkthroughs	21
	2. Testing	22
	a. Informal Testing	22
	b. Formal Tests	23
	C. Verification and Validation During the Software Acquisition Life Cycle	23
	1. Software Concept and Initiation Phase	23
	2. Software Requirements Phase	24
	3. Software Architectural (Preliminary) Design Phase	24
	4. Software Detailed Design Phase	24
	5. Software Implementation Phase	25
	6. Software Integration and Test Phase	25
	7. Software Acceptance and Delivery Phase	25
	8. Software Sustaining Engineering and Operations Phase	25
	D. Independent Verification and Validation	26
	E. Techniques and Tools	26
VI.	**NONCONFORMANCE REPORTING AND CORRECTIVE ACTION**	27
	A. Concepts and Definitions	27
	B. Activities	27
	1. Nonconformance Detection and Reporting	27
	2. Tracking and Management Reports	28
	3. Impact Assessment and Corrective Action	28
	C. Interrelationships	29
	D. Techniques and Tools	29

TABLE OF CONTENTS (Cont.)

Section		Page
VII.	**SOFTWARE AND SYSTEM SAFETY**	31
	A. Concepts and Definitions	31
	B. Software Problems	31
	C. Methods for Improving Software Safety	31
	D. Software Safety Program (Example)	32
	1. Requirements Analysis	32
	2. Design Analysis	32
	3. Code Analysis	33
	4. Safety Testing	33
	E. Techniques and Tools	33
VIII.	**SECURITY ASSURANCE**	35
	A. Concepts and Definitions	35
	B. Automated Information Security	35
	C. Security Assurance Activities	35
	D. Techniques and Tools	36
APPENDIX A	Software Assurance Activities by Software Development Phases	A-1
APPENDIX B	References	B-1
APPENDIX C	Acronyms and Abbreviations	C-1
INDEX		I-1

PRECEDING PAGE BLANK NOT FILMED

I. OVERVIEW

A. Concepts and Definitions

Software assurance is the planned and systematic set of activities that ensures that software processes and products conform to requirements, standards, and procedures. "Processes" include all of the activities involved in designing, developing, enhancing, and maintaining software; "products" include the software, associated data, its documentation, and all supporting and reporting paperwork.

The three mutually supportive activities involved in the software life cycle are management, engineering, and assurance. Software management is the set of activities involved in planning, controlling, and directing the software project. Software engineering is the set of activities that analyzes requirements, develops designs, writes code, and structures databases. Software assurance makes sure the management and engineering efforts put forth result in a product that meets all of its requirements.

Software assurance is not an organization, but a set of related activities. It is unlikely that any NASA Center or NASA contractor has a single organizational entity that performs all of the functions defined in this guidebook. The guidebook should be read as guidance to activities that are vital to the success of a software project. Some considerations for organizational structuring to enhance the probability of success are given in the section on establishing a software assurance activity.

B. Goals of Software Assurance

Software development, like any complex development activity, is a process full of risks. The risks are both technical and programmatic; that is, risks that the software will not perform as intended or will be too difficult to operate, modify, or maintain are technical risks, while risks that the project will overrun cost or schedule are programmatic risks. The goal of software assurance is to reduce these risks. For example, coding standards are set to specify a minimum quality of code. If no standards are set, there exists some risk that the code will not come up to a minimum usable standard, and that the code will require rework. If standards are set but there is no explicit process for assuring that all code meets the standards, then there is some risk that some coders will produce code that does not meet the standards. The assurance process involved is quality assurance, and to have no quality assurance activity is to increase the risk that unacceptable code will be produced.

Similarly, the lack of a nonconformance reporting and corrective action system increases the risk that problems in the software will be forgotten and not corrected, or that important problems will not get priority attention. Other risk-related examples can be provided to support all of the activities in this guidebook. The point is that software assurance activities can help to reduce risks.

C. Purpose of this Guidebook

The purpose of this Software Assurance Guidebook is to provide assistance, in the form of guidance, to NASA managers responsible for software acquisition and development and for establishing software assurance requirements.

The style of the guidebook is intended to be tutorial rather than directive. It is hoped that the reader will find the following sections an easily understood introduction to software assurance and a useful guide to formulating and addressing software project needs related to assurance.

The remainder of this guidebook will touch on each major activity within software assurance: software quality assurance, software quality engineering, verification and validation, nonconformance reporting and corrective action, safety, and security. Section II, Establishing a Project Software Assurance Activity, is designed to assist managers in starting a new assurance activity or improving an existing assurance program.

D. Changes to this Guidebook

Suggestions for improvement may be made on the form provided at the back of this guidebook.

II. ESTABLISHING A PROJECT SOFTWARE ASSURANCE ACTIVITY

A. Concepts and Definitions

Every software development, enhancement, or maintenance project includes some assurance activities. The types, amount, and formality of such activities are decisions of the project manager, based on an assessment of the project, its risks, and its development and operational environments.

Even a simple, one person development job has assurance activities embedded in it, even if the programmer denies that "quality assurance" plays any part in what is to be done. Each programmer has some idea of how code should be written, and this idea functions as a coding standard for that programmer. Likewise, each of us has some idea of how documentation should be written, and this is a personal documentation standard. Each programmer reviews his/her products to make sure they meet their internal standards, and this is an assurance review or audit. Each programmer tests and inspects his/her own work, and these are verification and validation processes. The list could go on, but the idea should be clear. A project software assurance program involves the processes that each programmer goes through, but requires the planning and formal establishment of project, rather than personal, standards and processes.

B. Tailoring Software Assurance to the Project

Specific project characteristics and risks influence assurance needs, and assurance planning should be tailored to reflect this fact. Characteristics that should be considered include safety and mission criticality of the software, schedule and budget, size and complexity of the product to be produced, and size and organizational complexity of the development staff.

The relationship of criticality to assurance is as one would expect: the more critical the software, the more important and formal the software assurance effort must be. The relationship of schedule and budget is not intuitive, however; the tighter the budget and schedule, the more critical it is to have a well planned and effective assurance effort. This does not mean that projects with more resources can afford to be lax, it just means that tight resources increase risks that should be offset by a strong assurance program.

The projected size of the software to be produced influences the level of assurance required. A large project requires explicit and detailed standards for all of the products in order to get at least a minimum standard of quality from the varied ideas and experience of many different programmers. In addition, a large project requires significant efforts in testing and other verification activities, which have to be planned and the plans followed. In short, just due to the size of the activity, a significant and formal assurance program must be established or risks of poor quality products must be accepted. On the other hand, a small project may require little formal assurance, and on a very small one, the assurance efforts may be left to the programmer involved if adequate, informal planning is done.

Another factor that influences assurance planning is the project's organizational structure. A small, centralized development staff can easily participate in reviews and inspections, keep each other informed on the status of nonconformances, and help each other in meeting coding and documentation standards. A large or dispersed staff will have many different ideas of the best ways of doing things and many more difficulties in communicating them. In the latter case, a more formal assurance program and a larger assurance effort will be needed.

A last but very important characteristic is the difference between the requirements of a software providing organization and a software acquiring organization. A software provider actually develops the products by developing designs and writing code, etc., and therefore needs a full assurance program. An acquirer does not develop software and thus can limit its assurance activities to those that ensure that the provider is adhering to agreed-to methods and standards and producing the agreed-to products.

C. Creating the Software Assurance Plan

An effective assurance program requires planning and follow through; it cannot simply evolve along with the project. Adequate assurance planning ensures that the assurance activities are focused on the quality requirements and risks associated with the specific project.

The purpose of creating a software assurance plan is to document/specify the conduct of the activities that will comprise software assurance for a specific project. Armed with information about the project and the available software assurance resources, the project manager is ready to develop the plan. A useful guide for documenting assurance plans is provided in the assurance sections of the SMAP Management Plan Documentation Standard. In addition, the following should be considered:

- Plan software assurance in conjunction with management and engineering planning, i.e., during the project concept and initiation phase.
- Phase assurance activities properly. For example, design standards must be produced well before design is to be done.
- Complete tool development or procurement before the tools are needed. Especially important is the development of test tools and test data sources.

A summary of software assurance activities grouped by software development phases is provided in Appendix A.

D. Project Structure Considerations

In planning and establishing a software assurance program, one consideration is the software project organization and the location in that organization of the assurance activities. Experience has indicated, both in hardware and software, that some assurance functions are best done by organizational entities that are separate from the ones doing engineering activities. Software Quality Assurance (SQA) is one activity that should be organizationally separated from the producing organizations. Administratively, the SQA

organization should report no lower than the project manager; indeed, many large successful software producing organizations have the SQA organization report administratively to top corporate management and interface with the project manager. The reason for this separation of function is that the SQA organization is management's arm that assures that standards are met and that procedures are followed. If SQA is not independent of the development activity, clear and impartial assessment will be difficult.

In addition, many organizations have had success using an independent test team, or at least an independent test development team. The team is responsible for developing test plans, procedures, and test cases for formal acceptance tests. Independence is required because these tests should be requirements driven and not influenced by the design structure and coding details.

E. Completion Criteria

Because of the nature of software, it is difficult to ascertain the status of a development or maintenance activity. It is important, therefore, to define criteria for the completion of specific development stages. For example, during the implementation phase, one has to do the lowest level detailed design of small program elements, code the elements, and unit test them. When a significant number of program elements are involved, it is difficult for anyone to ascertain the status of the units without specific completion criteria. For example, if there is a criterion that detailed design is complete only after the rework that finishes a design inspection, then the design can be said to be either complete or incomplete depending on the status of the rework.

The setting of completion criteria is a management activity, but the audit of records is an SQA activity. The accuracy of the reported status can then be determined. This is important to both providers and acquirers of software, and this "status auditing" is an important SQA function.

F. Implementation of the Software Assurance Plan

Once the project needs have been determined and the software assurance planning is complete, the plan must be implemented. Qualified, trained staff must be obtained, and special training must be made available where needed. If standards and procedures are not available for reuse on this project, they must be written. Staff must be trained in the standards and procedures, since merely writing them down does not guarantee compliance. All of the above are management activities, but the assurance staff is a resource to help complete them.

Staff devoted purely to assurance activities is usually small compared to the project staff. On the other hand, it is important to have people with specific assurance responsibilities, even if they must be shared organizationally with other duties. Too often the truism that "quality is everybody's business" becomes "quality is nobody's business" if specific responsibilities are not assigned.

G. Sources of Help

In addition to this guidebook, there are other sources of help in planning and implementing a software assurance program. First, there is a NASA software planning requirement, stated in NMI 2410.6. In addition, there are Center requirements and guidance documents. Many of these are listed in Appendix B, which also contains other useful reference material used in the development of this guidebook. All NASA Centers have assurance organizations that provide varying degrees of support, assistance, and actual performance of software assurance activities.

H. Summary

Software assurance is an essential part of the development and maintenance of software. Software assurance forms part of the triad of activities, along with software management and software engineering that, taken together, can provide a successful software development, enhancement, or maintenance activity. This guidebook is intended to increase the general understanding in NASA of what comprises software assurance and how it is to be planned and implemented.

III. SOFTWARE QUALITY ASSURANCE

A. Concepts and Definitions

Software Quality Assurance (SQA) is defined as a planned and systematic approach to the evaluation of the quality of and adherence to software product standards, processes, and procedures. SQA includes the process of assuring that standards and procedures are established and are followed throughout the software acquisition life cycle. Compliance with agreed-upon standards and procedures is evaluated through process monitoring, product evaluation, and audits. Software development and control processes should include quality assurance approval points, where an SQA evaluation of the product may be done in relation to the applicable standards.

B. Standards and Procedures

Establishing standards and procedures for software development is critical, since these provide the framework from which the software evolves. Standards are the established criteria to which the software products are compared. Procedures are the established criteria to which the development and control processes are compared. Standards and procedures establish the prescribed methods for developing software; the SQA role is to ensure their existence and adequacy. Proper documentation of standards and procedures is necessary since the SQA activities of process monitoring, product evaluation, and auditing rely upon unequivocal definitions to measure project compliance.

Types of standards include:

- *Documentation Standards* specify form and content for planning, control, and product documentation and provide consistency throughout a project. The NASA Data Item Descriptions (DIDs) are documentation standards (see Appendix B).

- *Design Standards* specify the form and content of the design product. They provide rules and methods for translating the software requirements into the software design and for representing it in the design documentation.

- *Code Standards* specify the language in which the code is to be written and define any restrictions on use of language features. They define legal language structures, style conventions, rules for data structures and interfaces, and internal code documentation.

Procedures are explicit steps to be followed in carrying out a process. All processes should have documented procedures. Examples of processes for which procedures are needed are configuration management, nonconformance reporting and corrective action, testing, and formal inspections.

If developed according to the NASA DID, the Management Plan describes the software development control processes, such as configuration management, for which there have

to be procedures, and contains a list of the product standards. Standards are to be documented according to the Standards and Guidelines DID in the Product Specification. The planning activities required to assure that both products and processes comply with designated standards and procedures are described in the QA portion of the Management Plan.

C. Software Quality Assurance Activities

Product evaluation and process monitoring are the SQA activities that assure the software development and control processes described in the project's Management Plan are correctly carried out and that the project's procedures and standards are followed. Products are monitored for conformance to standards and processes are monitored for conformance to procedures. Audits are a key technique used to perform product evaluation and process monitoring. Review of the Management Plan should ensure that appropriate SQA approval points are built into these processes.

Product evaluation is an SQA activity that assures standards are being followed. Ideally, the first products monitored by SQA should be the project's standards and procedures. SQA assures that clear and achievable standards exist and then evaluates compliance of the software product to the established standards. Product evaluation assures that the software product reflects the requirements of the applicable standard(s) as identified in the Management Plan.

Process monitoring is an SQA activity that ensures that appropriate steps to carry out the process are being followed. SQA monitors processes by comparing the actual steps carried out with those in the documented procedures. The Assurance section of the Management Plan specifies the methods to be used by the SQA process monitoring activity.

A fundamental SQA technique is the audit, which looks at a process and/or a product in depth, comparing them to established procedures and standards. Audits are used to review management, technical, and assurance processes to provide an indication of the quality and status of the software product.

The purpose of an SQA audit is to assure that proper control procedures are being followed, that required documentation is maintained, and that the developer's status reports accurately reflect the status of the activity. The SQA product is an audit report to management consisting of findings and recommendations to bring the development into conformance with standards and/or procedures.

D. SQA Relationships to Other Assurance Activities

Some of the more important relationships of SQA to other management and assurance activities are described below.

1. Configuration Management Monitoring

SQA assures that software Configuration Management (CM) activities are performed in accordance with the CM plans, standards, and procedures.

SQA reviews the CM plans for compliance with software CM policies and requirements and provides follow-up for nonconformances. SQA audits the CM functions for adherence to standards and procedures and prepares reports of its findings.

The CM activities monitored and audited by SQA include baseline control, configuration identification, configuration control, configuration status accounting, and configuration authentication. SQA also monitors and audits the software library. SQA assures that:

- Baselines are established and consistently maintained for use in subsequent baseline development and control.

- Software configuration identification is consistent and accurate with respect to the numbering or naming of computer programs, software modules, software units, and associated software documents.

- Configuration control is maintained such that the software configuration used in critical phases of testing, acceptance, and delivery is compatible with the associated documentation.

- Configuration status accounting is performed accurately including the recording and reporting of data reflecting the software's configuration identification, proposed changes to the configuration identification, and the implementation status of approved changes.

- Software configuration authentication is established by a series of configuration reviews and audits that exhibit the performance required by the software requirements specification and the configuration of the software is accurately reflected in the software design documents.

- Software development libraries provide for proper handling of software code, documentation, media, and related data in their various forms and versions from the time of their initial approval or acceptance until they have been incorporated into the final media.

- Approved changes to baselined software are made properly and consistently in all products, and no unauthorized changes are made.

2. Verification and Validation Monitoring

SQA assures Verification and Validation (V&V) activities by monitoring technical reviews, inspections, and walkthroughs. The SQA role in formal testing is described in the next section. The SQA role in reviews, inspections, and walkthroughs is to observe, participate as needed, and verify that they were properly conducted and documented. SQA also ensures that any actions required are assigned, documented, scheduled, and updated.

Formal software reviews should be conducted at the end of each phase of the life cycle to identify problems and determine whether the interim product meets all applicable requirements. Examples of formal reviews are the Preliminary Design Review (PDR), Critical Design Review (CDR), and Test Readiness Review (TRR). A review looks at the overall picture of the product being developed to see if it satisfies its requirements. Reviews are part of the development process, designed to provide a ready/not-ready decision to begin the next phase. In formal reviews, actual work done is compared with established standards. SQA's main objective in reviews is to assure that the Management and Development Plans have been followed, and that the product is ready to proceed with the next phase of development. Although the decision to proceed is a management decision, SQA is responsible for advising management and participating in the decision.

An inspection or walkthrough is a detailed examination of a product on a step-by-step or line-of-code by line-of-code basis to find errors. For inspections and walkthroughs, SQA assures, at a minimum, that the process is properly completed and that needed follow-up is done. The inspection process may be used to measure compliance to standards.

3. Formal Test Monitoring

SQA assures that formal software testing, such as acceptance testing, is done in accordance with plans and procedures. SQA reviews testing documentation for completeness and adherence to standards. The documentation review includes test plans, test specifications, test procedures, and test reports. SQA monitors testing and provides follow-up on nonconformances. By test monitoring, SQA assures software completeness and readiness for delivery.

The objectives of SQA in monitoring formal software testing are to assure that:

- The test procedures are testing the software requirements in accordance with test plans.
- The test procedures are verifiable.
- The correct or "advertised" version of the software is being tested (by SQA monitoring of the CM activity).
- The test procedures are followed.
- Nonconformances occurring during testing (that is, any incident not expected in the test procedures) are noted and recorded.
- Test reports are accurate and complete.

- Regression testing is conducted to assure nonconformances have been corrected.

- Resolution of all nonconformances takes place prior to delivery.

Software testing verifies that the software meets its requirements. The quality of testing is assured by verifying that project requirements are satisfied and that the testing process is in accordance with the test plans and procedures.

E. Software Quality Assurance During the Software Acquisition Life Cycle

In addition to the general activities described in subsections C and D, there are phase-specific SQA activities that should be conducted during the Software Acquisition Life Cycle. At the conclusion of each phase, SQA concurrence is a key element in the management decision to initiate the following life cycle phase. Suggested activities for each phase are described below.

1. Software Concept and Initiation Phase

SQA should be involved in both writing and reviewing the Management Plan in order to assure that the processes, procedures, and standards identified in the plan are appropriate, clear, specific, and auditable. During this phase, SQA also provides the QA section of the Management Plan.

2. Software Requirements Phase

During the software requirements phase, SQA assures that software requirements are complete, testable, and properly expressed as functional, performance, and interface requirements.

3. Software Architectural (Preliminary) Design Phase

SQA activities during the architectural (preliminary) design phase include:

- Assuring adherence to approved design standards as designated in the Management Plan.

- Assuring all software requirements are allocated to software components.

- Assuring that a testing verification matrix exists and is kept up to date.

- Assuring the Interface Control Documents are in agreement with the standard in form and content.

- Reviewing PDR documentation and assuring that all action items are resolved.

- Assuring the approved design is placed under configuration management.

4. **Software Detailed Design Phase**

SQA activities during the detailed design phase include:

- Assuring that approved design standards are followed.

- Assuring that allocated modules are included in the detailed design.

- Assuring that results of design inspections are included in the design.

- Reviewing CDR documentation and assuring that all action items are resolved.

5. **Software Implementation Phase**

SQA activities during the implementation phase include the audit of:

- Results of coding and design activities including the schedule contained in the Software Development Plan.

- Status of all deliverable items.

- Configuration management activities and the software development library.

- Nonconformance reporting and corrective action system.

6. **Software Integration and Test Phase**

SQA activities during the integration and test phase include:

- Assuring readiness for testing of all deliverable items.

- Assuring that all tests are run according to test plans and procedures and that any nonconformances are reported and resolved.

- Assuring that test reports are complete and correct.

- Certifying that testing is complete and software and documentation are ready for delivery.

- Participating in the Test Readiness Review and assuring all action items are completed.

7. **Software Acceptance and Delivery Phase**

As a minimum, SQA activities during the software acceptance and delivery phase include assuring the performance of a final configuration audit to demonstrate that all deliverable items are ready for delivery.

8. **Software Sustaining Engineering and Operations Phase**

During this phase, there will be mini-development cycles to enhance or correct the software. During these development cycles, SQA conducts the appropriate phase-specific activities described above.

F. Techniques and Tools

SQA should evaluate its needs for assurance tools versus those available off-the-shelf for applicability to the specific project, and must develop the others it requires. Useful tools might include audit and inspection checklists and automatic code standards analyzers.

PRECEDING PAGE BLANK NOT FILMED

PRECEDING PAGE BLANK NOT FILMED

IV. SOFTWARE QUALITY ENGINEERING

A. Concepts

Software Quality Engineering (SQE) is a process that evaluates, assesses, and improves the quality of software. Software quality is often defined as the degree to which software meets requirements for reliability, maintainability, transportability, etc., as contrasted with functional, performance, and interface requirements that are satisfied as a result of software engineering.

Quality must be built into a software product during its development to satisfy quality requirements established for it. SQE ensures that the process of incorporating quality in the software is done properly, and that the resulting software product meets the quality requirements. The degree of conformance to quality requirements usually must be determined by analysis, while functional requirements are demonstrated by testing. SQE performs a function complementary to software development engineering. Their common goal is to ensure that a safe, reliable, and quality engineered software product is developed.

B. Software Qualities

Qualities for which an SQE evaluation is to be done must first be selected and requirements set for them. Some commonly used qualities are reliability, maintainability, transportability, interoperability, testability, useability, reusability, traceability, sustainability, and efficiency. Some of the key ones are discussed below.

1. Reliability

Hardware reliability is often defined in terms of the Mean-Time-To-Failure, or MTTF, of a given set of equipment. An analogous notion is useful for software, although the failure mechanisms are different and the mathematical predictions used for hardware have not yet been usefully applied to software. Software reliability is often defined as the extent to which a program can be expected to perform intended functions with required precision over a given period of time. Software reliability engineering is concerned with the detection and correction of errors in the software; even more, it is concerned with techniques to compensate for unknown software errors and for problems in the hardware and data environments in which the software must operate.

2. Maintainability

Software maintainability is defined as the ease of finding and correcting errors in the software. It is analogous to the hardware quality of Mean-Time-To-Repair, or MTTR. While there is as yet no way to directly measure or predict software maintainability, there is a significant body of knowledge about software attributes that make software easier to maintain. These include modularity, self (internal) documentation, code readability,

and structured coding techniques. These same attributes also improve sustainability, the ability to make improvements to the software.

3. Transportability

Transportability is defined as the ease of transporting a given set of software to a new hardware and/or operating system environment.

4. Interoperability

Software interoperability is the ability of two or more software systems to exchange information and to mutually use the exchanged information.

5. Efficiency

Efficiency is the extent to which software uses minimum hardware resources to perform its functions.

There are many other software qualities. Some of them will not be important to a specific software system, thus no activities will be performed to assess or improve them. Maximizing some qualities may cause others to be decreased. For example, increasing the efficiency of a piece of software may require writing parts of it in assembly language. This will decrease the transportability and maintainability of the software.

C. Metrics

Metrics are quantitative values, usually computed from the design or code, that measure the quality in question, or some attribute of the software related to the quality. Many metrics have been invented, and a number have been successfully used in specific environments, but none has gained widespread acceptance.

D. A Software Quality Engineering Program

The two software qualities which command the most attention are reliability and maintainability. Some practical programs and techniques have been developed to improve the reliability and maintainability of software, even if they are not measurable or predictable. The types of activities that might be included in an SQE program are described here in terms of these two qualities. These activities could be used as a model for the SQE activities for additional qualities.

1. Qualities and Attributes

An initial step in laying out an SQE program is to select the qualities that are important in the context of the use of the software that is being developed. For example, the highest priority qualities for flight software are usually reliability and efficiency. If revised flight software can be up-linked during flight, maintainability may be of interest, but considerations like transportability will not drive the design or implementation. On the other hand, the use of science analysis software might require ease of

change and maintainability, with reliability a concern and efficiency not a driver at all.

After the software qualities are selected and ranked, specific attributes of the software that help to increase those qualities should be identified. For example, modularity is an attribute that tends to increase both reliability and maintainability. Modular software is designed to result in code that is apportioned into small, self-contained, functionally unique components or units. Modular code is easier to maintain, because the interactions between units of code are easily understood, and low level functions are contained in few units of code. Modular code is also more reliable, because it is easier to completely test a small, self contained unit.

Not all software qualities are so simply related to measurable design and code attributes, and no quality is so simple that it can be easily measured. The idea is to select or devise measurable, analyzable, or testable design and code attributes that will increase the desired qualities. Attributes like information hiding, strength, cohesion, and coupling should be considered.

2. Quality Evaluations

Once some decisions have been made about the quality objectives and software attributes, quality evaluations can be done. The intent in an evaluation is to measure the effectiveness of a standard or procedure in promoting the desired attributes of the software product. For example, the design and coding standards should undergo a quality evaluation. If modularity is desired, the standards should clearly say so and should set standards for the size of units or components. Since internal documentation is linked to maintainability, the documentation standards should be clear and require good internal documentation.

Quality of designs and code should also be evaluated. This can be done as a part of the walkthrough or inspection process, or a quality audit can be done. In either case, the implementation is evaluated against the standard and against the evaluator's knowledge of good software engineering practices, and examples of poor quality in the product are identified for possible correction.

3. Nonconformance Analysis

One very useful SQE activity is an analysis of a project's nonconformance records. The nonconformances should be analyzed for unexpectedly high numbers of events in specific sections or modules of code. If areas of code are found that have had an unusually high error count (assuming it is not because the code in question has been tested more thoroughly), then the code should be examined. The high error count may be due to poor quality code, an inappropriate design, or requirements that are not well understood or defined. In any case, the analysis may indicate changes and

rework that can improve the reliability of the completed software. In addition to code problems, the analysis may also reveal software development or maintenance processes that allow or cause a high proportion of errors to be introduced into the software. If so, an evaluation of the procedures may lead to changes, or an audit may discover that the procedures are not being followed.

4. Fault Tolerance Engineering

For software that must be of high reliability, a fault tolerance activity should be established. It should identify software which provides and accomplishes critical functions and requirements. For this software, the engineering activity should determine and develop techniques which will ensure that the needed reliability or fault tolerance will be attained. Some of the techniques that have been developed for high reliability environments include:

- *Input data checking and error tolerance.* For example, if out-of-range or missing input data can affect reliability, then sophisticated error checking and data interpolation/extrapolation schemes may significantly improve reliability.

- *Proof of correctness.* For limited amounts of code, formal "proof of correctness" methods may be able to demonstrate that no errors exist.

- *N-Item voting.* This is a design and implementation scheme where a number of independent sets of software and hardware operate on the same input. Some comparison (voting) scheme is used to determine which output to use. This is especially effective where subtle timing or hardware errors may be present.

- *Independent development.* In this scheme, one or more of the N-items are independently developed units of software. This helps prevent the simultaneous failure of all items due to a common coding error.

E. Techniques and Tools

Some of the useful fault-tolerance techniques are described under subsection D, above. Standard statistical techniques can be used to manipulate nonconformance data. In addition, there is considerable experimentation with the Failure Modes and Effects Analysis (FMEA) technique adapted from hardware reliability engineering. In particular, the FMEA can be used to identify failure modes or other assumable (hardware) system states which can then lead the quality engineer to an analysis of the software that controls the system as it assumes those states.

There are also tools that are useful for quality engineering. They include system and software simulators, which allow the modeling of system behavior; dynamic analyzers,

which detect the portions of the code that are used most intensively; software tools that are used to compute metrics from code or designs; and a host of special purpose tools that can, for example, detect all system calls to help decide on portability limits.

PRECEDING PAGE BLANK NOT FILMED

V. VERIFICATION AND VALIDATION

A. Concepts and Definitions

Software Verification and Validation (V&V) is the process of ensuring that software being developed or changed will satisfy functional and other requirements (validation) and each step in the process of building the software yields the right products (verification). The differences between verification and validation are unimportant except to the theorist; practitioners use the term V&V to refer to all of the activities that are aimed at making sure the software will function as required.

V&V is intended to be a systematic and technical evaluation of software and associated products of the development and maintenance processes. Reviews and tests are done at the end of each phase of the development process to ensure software requirements are complete and testable and that design, code, documentation, and data satisfy those requirements.

B. Activities

The two major V&V activities are reviews, including inspections and walkthroughs, and testing.

1. Reviews, Inspections, and Walkthroughs

Reviews are conducted during and at the end of each phase of the life cycle to determine whether established requirements, design concepts, and specifications have been met. Reviews consist of the presentation of material to a review board or panel. Reviews are most effective when conducted by personnel who have not been directly involved in the development of the software being reviewed.

Informal reviews are conducted on an as-needed basis. The developer chooses a review panel and provides and/or presents the material to be reviewed. The material may be as informal as a computer listing or hand-written documentation.

Formal reviews are conducted at the end of each life cycle phase. The acquirer of the software appoints the formal review panel or board, who may make or affect a go/no-go decision to proceed to the next step of the life cycle. Formal reviews include the Software Requirements Review, the Software Preliminary Design Review, the Software Critical Design Review, and the Software Test Readiness Review.

An inspection or walkthrough is a detailed examination of a product on a step-by-step or line-of-code by line-of-code basis. The purpose of conducting inspections and walkthroughs is to find errors. The group that does an inspection or walkthrough is composed of peers from development, test, and quality assurance.

2. Testing

Testing is the operation of the software with real or simulated inputs to demonstrate that a product satisfies its requirements and, if it does not, to identify the specific differences between expected and actual results. There are varied levels of software tests, ranging from unit or element testing through integration testing and performance testing, up to software system and acceptance tests.

a. Informal Testing

Informal tests are done by the developer to measure the development progress. "Informal" in this case does not mean that the tests are done in a casual manner, just that the acquirer of the software is not formally involved, that witnessing of the testing is not required, and that the prime purpose of the tests is to find errors. Unit, component, and subsystem integration tests are usually informal tests.

Informal testing may be requirements-driven or design-driven. Requirements-driven or black box testing is done by selecting the input data and other parameters based on the software requirements and observing the outputs and reactions of the software. Black box testing can be done at any level of integration. In addition to testing for satisfaction of requirements, some of the objectives of requirements-driven testing are to ascertain:

- Computational correctness.
- Proper handling of boundary conditions, including extreme inputs and conditions that cause extreme outputs.
- State transitioning as expected.
- Proper behavior under stress or high load.
- Adequate error detection, handling, and recovery.

Design-driven or white box testing is the process where the tester examines the internal workings of code. Design-driven testing is done by selecting the input data and other parameters based on the internal logic paths that are to be checked. The goals of design-driven testing include ascertaining correctness of:

- All paths through the code. For most software products, this can be feasibly done only at the unit test level.

- Bit-by-bit functioning of interfaces.
- Size and timing of critical elements of code.

b. Formal Tests

Formal testing demonstrates that the software is ready for its intended use. A formal test should include an acquirer-approved test plan and procedures, quality assurance witnesses, a record of all discrepancies, and a test report. Formal testing is always requirements-driven, and its purpose is to demonstrate that the software meets its requirements.

Each software development project should have at least one formal test, the acceptance test that concludes the development activities and demonstrates that the software is ready for operations.

In addition to the final acceptance test, other formal testing may be done on a project. For example, if the software is to be developed and delivered in increments or builds, there may be incremental acceptance tests. As a practical matter, any contractually required test is usually considered a formal test; others are "informal."

After acceptance of a software product, all changes to the product should be accepted as a result of a formal test. Post acceptance testing should include regression testing. Regression testing involves rerunning previously used acceptance tests to ensure that the change did not disturb functions that have previously been accepted.

C. Verification and Validation During the Software Acquisition Life Cycle

The V&V Plan should cover all V&V activities to be performed during all phases of the life cycle. The V&V Plan Data Item Description (DID) may be rolled out of the Product Assurance Plan DID contained in the SMAP Management Plan Documentation Standard and DID.

1. Software Concept and Initiation Phase

The major V&V activity during this phase is to develop a concept of how the system is to be reviewed and tested. Simple projects may compress the life cycle steps; if so, the reviews may have to be compressed. Test concepts may involve simple generation of test cases by a user representative or may require the development of elaborate simulators and test data generators. Without an adequate V&V concept and plan, the cost, schedule, and complexity of the project may be poorly estimated due to the lack of adequate test capabilities and data.

2. **Software Requirements Phase**

V&V activities during this phase should include:

- Analyzing software requirements to determine if they are consistent with, and within the scope of, system requirements.
- Assuring that the requirements are testable and capable of being satisfied.
- Creating a preliminary version of the Acceptance Test Plan, including a verification matrix, which relates requirements to the tests used to demonstrate that requirements are satisfied.
- Beginning development, if needed, of test beds and test data generators.
- The phase-ending Software Requirements Review (SRR).

3. **Software Architectural (Preliminary) Design Phase**

V&V activities during this phase should include:

- Updating the preliminary version of the Acceptance Test Plan and the verification matrix.
- Conducting informal reviews and walkthroughs or inspections of the preliminary software and data base designs.
- The phase-ending Preliminary Design Review (PDR) at which the allocation of requirements to the software architecture is reviewed and approved.

4. **Software Detailed Design Phase**

V&V activities during this phase should include:

- Completing the Acceptance Test Plan and the verification matrix, including test specifications and unit test plans.
- Conducting informal reviews and walkthroughs or inspections of the detailed software and data base designs.
- The Critical Design Review (CDR) which completes the software detailed design phase.

5. **Software Implementation Phase**

V&V activities during this phase should include:

- Code inspections and/or walkthroughs.
- Unit testing software and data structures.
- Locating, correcting, and retesting errors.
- Development of detailed test procedures for the next two phases.

6. **Software Integration and Test Phase**

This phase is a major V&V effort, where the tested units from the previous phase are integrated into subsystems and then the final system. Activities during this phase should include:

- Conducting tests per test procedures.
- Documenting test performance, test completion, and conformance of test results versus expected results.
- Providing a test report that includes a summary of nonconformances found during testing.
- Locating, recording, correcting, and retesting nonconformances.
- The Test Readiness Review (TRR), confirming the product's readiness for acceptance testing.

7. **Software Acceptance and Delivery Phase**

V&V activities during this phase should include:

- By test, analysis, and inspection, demonstrating that the developed system meets its functional, performance, and interface requirements.
- Locating, correcting, and retesting nonconformances.
- The phase-ending Acceptance Review (AR).

8. **Software Sustaining Engineering and Operations Phase**

Any V&V activities conducted during the prior seven phases are conducted during this phase as they pertain to the revision or update of the software.

D. Independent Verification and Validation

Independent Verification and Validation (IV&V) is a process whereby the products of the software development life cycle phases are independently reviewed, verified, and validated by an organization that is neither the developer nor the acquirer of the software. The IV&V agent should have no stake in the success or failure of the software. The IV&V agent's only interest should be to make sure that the software is thoroughly tested against its complete set of requirements.

The IV&V activities duplicate the V&V activities step-by-step during the life cycle, with the exception that the IV&V agent does no informal testing. If there is an IV&V agent, the formal acceptance testing may be done only once, by the IV&V agent. In this case, the developer will do a formal demonstration that the software is ready for formal acceptance.

E. Techniques and Tools

Perhaps more tools have been developed to aid the V&V of software (especially testing) than any other software activity. The tools available include code tracers, special purpose memory dumpers and formatters, data generators, simulations, and emulations. Some tools are essential for testing any significant set of software, and, if they have to be developed, may turn out to be a significant cost and schedule driver.

An especially useful technique for finding errors is the formal inspection. Formal inspections were developed by Michael Fagan of IBM. Like walkthroughs, inspections involve the line-by-line evaluation of the product being reviewed. Inspections, however, are significantly different from walkthroughs and are significantly more effective. Inspections are done by a team, each member of which has a specific role. The team is led by a moderator, who is formally trained in the inspection process. The team includes a reader, who leads the team through the item; one or more reviewers, who look for faults in the item; a recorder, who notes the faults; and the author, who helps explain the item being inspected.

This formal, highly structured inspection process has been extremely effective in finding and eliminating errors. It can be applied to any product of the software development process, including documents, design, and code. One of its important side benefits has been the direct feedback to the developer/author, and the significant improvement in quality that results.

VI. NONCONFORMANCE REPORTING AND CORRECTIVE ACTION

A. Concepts and Definitions

The purpose of a Nonconformance Reporting and Corrective Action (NRCA) system or procedure is to report, analyze, and correct nonconformances and collect information from which reports on the overall status of nonconformances can be made. A nonconformance, often called a problem, discrepancy, anomaly, fault, or error, is any failure of any software document, code, or data structure to meet its requirements or standards. Corrective action is a general name for the process by which nonconformances are corrected and controlled.

The need for a NRCA system arises early in the software life cycle, as soon as the first documents and other products are developed. A NRCA system should track nonconformances, assign priorities, record their dispositions, note the version of the product in which they are corrected, notify the originator of the nonconformance about the actions taken, and produce management reports.

B. Activities

1. Nonconformance Detection and Reporting

By definition, a nonconformance is a deviation of any product from its requirements or standards. Nonconformance reports may be filed against any product in any phase of the software life cycle by anyone associated with the project. Normally, the NRCA system is used after a product is first approved or baselined by its developer and released for wider use. For example, while a developer is unit testing his/her code, errors found may be tracked only locally. After the code is declared correct and released for integration, the NRCA system is used to inform users of the code about nonconformances and to assure that the nonconformances are corrected and not overlooked.

Usually, a special form is used to make the nonconformance report. Examples of the information the form might contain are:

- Date and time of detection of the nonconformance.
- Error identification (report number and title).
- Reporting individual and organization.
- Individual responsible for corrective action.
- Criticality of the nonconformance.
- Statement of the nonconformance.

- Proposed fix for the nonconformance.
- Identifier of the unit of code, data, or documentation in which corrective action must be taken.
- Life cycle phase in which the nonconformance was introduced.
- Life cycle phase in which the nonconformance was detected.
- Final closure resolution.
- Date and/or version of the configuration item in which the correction will be included.
- Date on which the nonconformance is closed.

A DID for a discrepancy report is given in the Management Control and Status Reports volume of the NASA Documentation Standard.

2. Tracking and Management Reports

After the report is prepared by the individual who found the nonconformance, the data are entered into some form of controlling system. Data base management systems are often used to help automate the otherwise laborious clerical effort of tracking the nonconformances and providing management reports.

A nonconformance tracking and reporting system should be able to provide management reports containing such information as error and correction status, the number of errors found per product, and the criticality of open problems. The data enable the impact of nonconformances to be evaluated so that the use of resources may be prioritized.

3. Impact Assessment and Corrective Action

Nonconformance reports should be evaluated for criticality and level of importance. Factors to be considered include:

- The impact of not correcting the nonconformance.
- The resources required for correcting the nonconformance.
- The impact on other baselined items if the nonconformance is corrected.

If the decision is made to correct a nonconformance, there should be procedures to control the corrective action process. Such procedures should include followup to ensure the nonconformance has been

documented and corrected in the appropriate version of software, and to assure that adequate testing, including regression testing, is done.

C. Interrelationships

NRCA is a basic and fundamental tool for project management and for software assurance. As such, it impacts and interacts with many software management, development, and assurance activities. For example, CM has to track the product changes and versions that result from correcting nonconformances. In addition, some nonconformance reports will contain requirements changes disguised as nonconformances. These reports should result in the opening of a change request.

If the nonconformance is in a code or data product, those responsible for V&V activities must develop tests to ensure that the problem is indeed satisfactorily corrected. In addition, regression testing is needed to make sure that no new problems have been introduced by the fix.

SQA must assure that proper procedures are followed in processing nonconformances, and that shortcuts are not taken since they would threaten product integrity. SQA must also ensure that the modified product still meets its standards. SQA may use the numbers of nonconformances detected in specific areas of the system or that occurred in specific life cycle phases to identify process or product areas that might benefit from an audit.

The NRCA system is a useful tool for SQE, since it is often true that a product or component of a product with a large number of nonconformances is of poor quality and/or reliability and needs to be reexamined. In systems with significant safety and/or security requirements, the safety and security staffs will review nonconformances both to assure that their requirements are not compromised and to look for weaknesses that the problems might have uncovered.

D. Techniques and Tools

Each project should consider an automated tracking system for nonconformance reports and an automated updating capability to identify and record the product changes that occur as a result of the resolution of the nonconformances.

PRECEDING PAGE BLANK NOT FILMED

VII. SOFTWARE AND SYSTEM SAFETY

A. Concepts and Definitions

System safety is concerned with the possibility of catastrophic failure of systems in such a way as to compromise the safety of people or property, or result in mission failure. Software safety is definable only in the system context. Software has no inherent dangers; however, systems controlled or monitored by software do fail, and some failures of some systems will have safety impacts. To the extent that system failures can be caused or fail to be prevented by software, there is a need for an activity called "software safety."

If we are to be concerned with the safety of software only in a system context, we must then be concerned with nonconformances in the software and with the software requirements as well. Indeed, the most serious problems with software-based systems are those that develop when the software requirements are incorrect, inappropriate, or incomplete for the system situation.

B. Software Problems

System failures that are caused by software are due to one of two types of software problems: nonconformances (or failures to satisfy requirements) or an error or omission in the software requirements. A nonconformance may be simple (the most common is a coding error or "bug"), or more complex (i.e., a subtle timing error that delays a shuttle launch). The important point about nonconformances is that verification and validation techniques are designed to detect them and assurance techniques are designed to prevent them; improvements in these methods and a safety program based on specialized application of them are improving the safety and reliability of software controlled systems.

An error or omission in requirements is less tractable. The software may perform exactly as required, but the requirements do not correctly deal with some system state. When the system enters the undefined state, unexpected and undesirable behavior may result. This type of problem cannot be handled within the software discipline; it results from a failure of the system and software engineering processes which developed and allocated the system requirements to the software.

C. Methods for Improving Software Safety

Improving the software development process and building better software are ways to increase system reliability, i.e., by producing software with fewer faults. Intuitively, more reliable software is probably safer software, but from a safety standpoint more concentration on safety-related software functions is needed. A first order approach is to identify the critical software that controls system safety-related functions and give it special attention through the development and testing process. This is just a special case of the "build it better" method, but it focuses scarce resources on critical areas.

D. Software Safety Program (Example)

System hazard analysis may indicate that some software requires a more formal safety program because it is included in a safety critical system component. The software safety program begins with a preliminary software safety analysis. The purpose of the preliminary software safety analysis is to identify software controlled functions that affect the safety critical component and the software components that execute the functions. These software components are safety critical. When a safety critical software component is identified, then software safety activities are initiated on that component and continued through the requirements, design, and code analyses and testing phases in the software development process.

1. Requirements Analysis

Software safety requirements analysis forms the basis for subsequent software safety activities. The process of requirements analysis evaluates both software and interface requirements. The analysis is intended to identify errors and deficiencies in the software requirements that could result in the identified hazardous system states. Techniques employed in performing requirements analysis include criticality analysis; specification analysis; and timing, sizing, and throughput analysis.

- *Criticality analysis* evaluates each requirement in terms of the safety objectives derived for a given software component. This evaluation is to determine whether the requirement has safety implications. If so, the requirement is deemed critical and must be tracked throughout the software development cycle; that is, through design, coding, and testing. It must be traceable from the highest level specification all the way to the code and documentation.

- *Specification analysis* evaluates the completeness, correctness, consistency, and testability of identified software safety critical requirements. Specification analysis considers each requirement singly and all requirements as a set.

- *Timing, sizing, and throughput* analysis evaluates software requirements that relate to execution time, memory allocation, and channel usage. Timing, sizing, and throughput analysis focuses on noting and defining program constraints based on maximum required and allowable execution times, maximum memory usage and availability, and throughput considerations based on I/O channel usage.

2. Design Analysis

Design analysis verifies that the program design correctly implements safety critical requirements.

- *Design logic analysis* evaluates the equations, algorithms, and control logic of the software design.

- *Design data analysis* evaluates the description and intended usage of each data item used in design of the critical component. Interrupts and their effect on data must receive special attention in safety critical areas to verify that interrupts and interrupt handling routines do not alter critical data items used by other routines.

- *Design interface analysis* verifies the proper design of a software component's interfaces with other components of the system, including hardware, software, and operators.

- *Design constraint analysis* evaluates the design solutions against restrictions imposed by requirements and real-world limitations. The design must be responsive to all known or anticipated restrictions on the software component. These restrictions may include timing, sizing, and throughput constraints, equation and algorithm limitations, input and output data limitations, and design solution limitations.

3. Code Analysis

Code analysis verifies that the coded program correctly implements the verified design and does not violate safety requirements. The techniques used in the performance of code analysis mirror those used in design analysis.

4. Safety Testing

Software safety testing verifies analysis results, investigates program behavior, and confirms that the program complies with safety requirements. Special safety testing, conducted in accordance with the safety test plan and procedures, establishes the compliance of the software with the safety requirements. Safety testing focuses on locating program weaknesses and identifying extreme or unexpected situations that could cause the software to fail in ways that would cause a violation of safety requirements. The safety testing effort is limited to those software requirements classified as safety critical items.

E. Techniques and Tools

In the last few years, there has been much effort to adapt methods used in hardware safety and reliability to software. Tools like fault tree analysis and sneak circuit analysis have been applied to software with some success. Modeling of software using Petri nets has been tried, and other modeling techniques have been advocated, but with only limited success to date. While some techniques may have some limited usefulness, their success depends heavily on the ability of the analyst that applies them.

 PRECEDING PAGE BLANK NOT FILMED

VIII. SECURITY ASSURANCE

A. Concepts and Definitions

NASA policy states that automated information resources shall be provided with a level of security and integrity consistent with the potential harm from their loss, inaccuracy, alteration, unavailability, or misuse. Software is itself a resource and thus must be afforded appropriate security. Software also contains and controls data and other NASA resources; it must be designed and implemented to protect those resources. Software security assurance is the process of ensuring that the above requirements are satisfied during all phases of the software life cycle.

B. Automated Information Security

Policy for automated information security (AIS) is contained in NMI 2410.7A, "Assuring the Security and Integrity of NASA Automated Information Systems." Very briefly, the policy states that security protection provided for a system must be appropriate to its sensitivity. It also states that the sensitivity of a system is based on the sensitivity of the information being handled by the system. Sensitivity is based on the impact on NASA of inaccurate, altered, disclosed, or unavailable information.

The AIS process begins by considering and categorizing the information that is to be contained in the system. The information, including both programs and data, should be categorized according to its sensitivity. For example, in the lowest category, the impact of a security violation is minimal; the impact on NASA's missions, functions, or reputation is negligible, or result in the loss of no tangible asset. For a top category, however, the impact may pose a threat to human life; may have an irreparable impact on NASA's missions, functions, image, or reputation; or may result in the loss of significant assets or resources.

Based on the categorization, security requirements should be developed. The security requirements should encompass system access control, including network access and physical access; data management and data access; environmental controls (power, air conditioning, etc.) and off-line storage; human resource security; and audit trails and usage records.

C. Security Assurance Activities

Security assurance activities are directed to ensuring that information being (or to be) processed by an automated information system has been assigned a proper sensitivity category and that the appropriate protection requirements have been developed and met in the system being developed or maintained. In addition, security assurance activities include ensuring the control and protection of the software being developed and/or maintained, and of software support tools and data.

A minimum security assurance program should ensure that:

- A security evaluation has been performed.
- Security requirements have been established for the software and data being developed and/or maintained.
- Security requirements have been established for the development and/or maintenance process.
- Each software review and/or audit includes evaluation of security requirements.
- The configuration management and corrective action processes provide security for the existing software and that the change evaluation processes prevent security violations.
- Physical security for software and data is adequate.

D. Techniques and Tools

Off-the-shelf packages are available to be used to support security requirements. If used, they must be evaluated and their effectiveness assured.

APPENDIX A

SOFTWARE ASSURANCE ACTIVITIES BY SOFTWARE DEVELOPMENT PHASES

This appendix provides a list of key assurance activities within each of the NASA software acquisition life cycle phases. The list shows the guidebook section in which the activity is described and a short phrase describing the activity.

1. Software Concept and Initiation Phase

The software concept is defined in this phase. Management strategy and constraints are developed. Procurement of the software development is initiated.

SQA	Review the Software Management Plan.
	Write the SQA section of the Management Plan.
SQE	Select qualities for evaluation during the development process.
V&V	Develop and review system test concept.
Security	Review security requirements concepts.

2. Software Requirements Phase

In this phase, procurement of the software development and IV&V is finalized. Requirements analysis is performed. The development process is defined. The Software Requirements Review is the end-of-phase review.

SQA	Assure that software requirements are complete, testable, and properly expressed as functional, performance, and interface requirements.
	Assure that preliminary test plans are under development.
SQE	Establish a fault tolerance activity.
V&V	Develop a preliminary version of the Acceptance Test Plan, including a verification matrix, which relates requirements to tests.
	Analyze software requirements to determine if they are consistent with, and within the scope of, system requirements.
	Conduct the Software Requirements Review.
Safety	Perform safety and criticality analyses.

Security Assure that proper sensitivity levels are assigned.

Assure that software is in compliance with protection requirements.

3. Software Architectural (Preliminary) Design Phase

In this phase, the software high-level architectural design is developed. Allocation of the software component's requirements to its architectural design elements takes place. The Software Preliminary Design Review (PDR) is the end-of-phase review.

SQA Assure adherence to approved design standards.

Review allocation of software requirements to software components.

Ensure existence of an up-to-date verification matrix.

Conduct assurance audits of processes, procedures, and status.

SQE Assure that during its development, a software product satisfies the quality requirements established for it.

V&V Update the preliminary versions of the Acceptance Test Plan and the verification matrix.

Participate in informal reviews and walkthroughs or inspections of the preliminary software and data base designs.

Review and approve allocation of requirements to the software architecture at the PDR.

Safety Identify and analyze safety critical components.

Security Assure that the system being developed meets minimum protection requirements.

Assure that the PDR includes evaluation of security requirements.

4. Software Detailed Design Phase

In this phase, the unit level software design is developed. The Software Critical Design Review (CDR) is the end-of-phase review.

SQA Assure that approved design standards are followed.

Assure the design is complete, all modules are developed, and all requirements are included.

SQA (Cont.)	Conduct assurance audits of processes, procedures, and status.
	Review CDR documentation to assure all action items are resolved.
V&V	Complete the Acceptance Test Plan and the verification matrix, including test specifications and unit test plans.
	Conduct informal reviews and walkthroughs or inspections of the detailed software and data base designs.
	Conduct the CDR.
Security	Assure that the CDR includes evaluation of security requirements.

5. Software Implementation Phase

In this phase, the software components are coded and unit tested.

SQA	Assure that coding and documentation standards are followed.
	Audit the status of all deliverable items, configuration management activities, the software development library, and the NRCA system.
SQE	Analyze the project's nonconformance records for trends.
V&V	Audit code inspections and/or walkthroughs.
	Perform unit testing of software and data structures.
	Locate, correct, and retest errors.
	Develop detailed test procedures for the next phases.
Safety	Perform code analysis.
Security	Assure that reviews include evaluation of security requirements.

6. Software Integration and Test Phase

In this phase, the software components (units) are integrated. Integration testing is performed. The Test Readiness Review (TRR) is the end-of-phase review.

SQA — Assure readiness for testing of all deliverable items.

Verify readiness of test data.

Assure that all tests are run according to test plans and procedures, and nonconformances are reported and resolved.

Certify completeness and correctness of reports.

Assure readiness of software for delivery.

V&V — Conduct tests per test procedures.

Document performance, test completion, and conformance of test results versus expected results.

Provide test report and NRCA reports.

Locate, correct, and retest nonconformances.

Conduct the test readiness reviews.

Safety — Conduct software safety testing for software requirements classified as safety critical items in accordance with the safety test plan and procedures.

Security — Assure that satisfaction of security requirements are evaluated at the test readiness reviews.

7. Software Acceptance and Delivery Phase

In this phase, the software component is validated as having met its documented requirements and user needs. The software is prepared for delivery. The Software Acceptance Review is the end-of-phase review.

SQA — Assure performance of final configuration audit.

V&V — Demonstrate by test, analysis, and inspection, that the developed system meets its functional, performance, and interface requirements.

Locate, correct, and retest nonconformances.

Participate in the Acceptance Review (AR).

Security — Assure that the AR includes evaluation of security requirements.

8. Software Sustaining Engineering and Operations Phase

In this phase, the software is sustained, maintained, and enhanced. Enhancement must be within the scope of the original concept; otherwise, an entire new life cycle should be begun. This process is known as evolutionary acquisition.

To maintain the integrity and quality of the software, the same types and levels of assurance must be conducted during sustaining engineering as were performed during development (i.e., previous phases of the life cycle). Reviews are done for each product that is modified. Retirement of the software is the end-of-phase activity.

PRECEDING PAGE BLANK NOT FILMED

APPENDIX B

REFERENCES

NASA Documents

NASA Software Management Requirements for Flight Projects, NMI 2410.6.

SMAP Products

NASA, Office of Safety, Reliability, Maintainability, and Quality Assurance, Software Management and Assurance Program (SMAP), Washington, DC, Information System Life-Cycle and Documentation Standards, Release 4.3, February 28, 1989.

NASA, Office of Safety, Reliability, Maintainability, and Quality Assurance, Software Management and Assurance Program (SMAP), Washington, DC, Management Plan Documentation Standard and Data Item Descriptions (DID), Volume of the Information System Life-Cycle and Documentation Standards, Release 4.3, February 28, 1989.

NASA, Office of Safety, Reliability, Maintainability, and Quality Assurance, Software Management and Assurance Program (SMAP), Washington, DC, Product Specification Documentation Standard and Data Item Descriptions (DID), Volume of the Information System Life-Cycle and Documentation Standards, Release 4.3, February 28, 1989.

NASA, Office of Safety, Reliability, Maintainability, and Quality Assurance, Software Management and Assurance Program (SMAP), Washington, DC, Assurance Specification Documentation Standard and Data Item Descriptions (DID), Volume of the Information System Life-Cycle and Documentation Standards, Release 4.3, February 28, 1989.

NASA, Office of Safety, Reliability, Maintainability, and Quality Assurance, Software Management and Assurance Program (SMAP), Washington, DC, Management Control and Status Reports Documentation Standard and Data Item Descriptions (DID), Volume of the Information System Life-Cycle and Documentation Standards, Release 4.3, February 28, 1989.

NASA, Office of Safety, Reliability, Maintainability, and Quality Assurance, Software Management and Assurance Program (SMAP), Washington, DC, NASA Software Acquisition Life Cycle, Version 4, 1989.

NASA Information System Life-Cycle Inter-Relationships - A Phased-Delivery Model.

SMAP Training Courses

Software Acquisition Management (D-TRG-21)
Software Quality Assurance (D-TRG-22)
Software Project Management (D-TRG-23)
Software Configuration Management (D-TRG-24)
Software Verification and Validation (D-TRG-25)
Introduction to the Ada Programming Language
Software Test Workshop
Automated Information System Security

European Space Agency (ESA) Documents

Roy Atkins presentation in Washington, December 5, 1988. Topics: Reliability, Software, Non-Electronic Reliability, Human Dependability, Preventive Maintenance.

ESA Software Product Assurance Standard, PSS-01-21, Working Copy, November 1988.

European Space Software Development Environment (ESSDE) Requirements Document and Comments, WME/87-409/SPP, Issue 1, Rev 0, February 9, 1988.

Hazard Analysis - November 1988.

Hierarchical Object Oriented Design (HOOD) Manual.

Lists of Software Product Assurance Activities, Appendix C, taken from PSS-01-2, Issue 21, Draft 4 (Deleted from Subsequent Drafts), March 1988.

List of SRM&QA Activities Comparison of Development and PA Activities.

Outlines of the Level 3 Software PA Documents.

Risk Assessment.

Software Configuration Identification for ESA Spacecraft and Associated Equipment, ESA PSS-01-111, Issue 0, Draft 1, December 1988.

Software Configuration Control for ESA Spacecraft and Associated Equipment, ESA PSS-01-112, Issue 0, Draft 1, December 1988.

Software Configuration Status Accounting for ESA Spacecraft and Associated Equipment, ESA PSS-01-113, Issue 0, Draft 1, December 1988.

Software Configuration Verification for ESA Spacecraft and Associated Equipment, ESA PSS-01-114, Issue 0, Draft 1, December 1988.

Software Criticality Classification Working Papers (October and November 1988).

Ames Research Center

Software Assurance Management Manual, AMM 5333-2, October 11, 1984.

Configuration Management Requirements - Basic Operations Manual, Appendix 13 (0-1.013), Rev. 1, February 1988.

Requirements for Establishment and Implementation of Software Assurance Programs, AHB 5333-1, September 1984.

Software Management Requirements - Basic Operations Manual, Appendix 14 (0-1.014), Rev. 1, April 1987.

Work Order Process - Basic Operations Manual, Appendix 16 (0-1.016), Rev 0, July 1986.

Goddard Space Flight Center

Ronald W. Durachka, Software Engineering Standards and Practices, TM-82148, June 1981.

Guidelines for Software Assurance Requirements for Flight Projects, SAR-1, January 1986.

L. E. Hyatt, Procedures for Conducting Software Audit, February 1985.

Systems Management Policy, MDOD-8YMP/0485, July 1986.

Lyndon B. Johnson Space Center

National Space Transportation System, Methodology for Conduct of NSTS Hazard Analyses, NSTS 22254, May 20, 1987.

Jet Propulsion Laboratory (JPL)

A Discussion for the Development and Implementation of a Distributed Database Architecture, Rev. 1.

Julian O. Blosiu, Software Independent Verification and Validation Cost Effectiveness Study, September 26, 1985.

Julian O. Blosiu, Independent Verification and Validation of Computer Software: Methodology, JPL D-576, February 9, 1983.

Cost-Effectiveness of Software Independent Verification and Validation, October 15, 1985 (Prepared Under RTOP #323-51-72).

JPL Software Standard Practice, 500-152, Implementation and Management of JPL Software, October 1, 1982.

John F. Kennedy Space Center

Test, Control, and Monitor System (TCMS) Software Management Plan, KSC-STA-70 BASIC, Space Station Project, July 15, 1988.

Lewis Research Center

Office of SR&QA, Software Product Assurance Work Plan, September 1987.

Marshall Space Flight Center

MSFC Software Management and Development Requirements, MA-002-006-2H. Science and Engineering Configuration Management Plan, MM 8040.1.

Software Quality Assurance (SQA) Requirements for Flight Projects, MSFC-CQ-5330.1.

Other Documents

ANSI/IEEE Std 729-1983, An American National Standard IEEE Standard Glossary of Software Engineering Terminology, 1983.

Bell Laboratories Software Quality Group, Technology Training Department, Software Reliability Measurement.

Draft Standard for Software Verification and Validation Plans, STD P1012/D5.0, July 31, 1985, Sponsor Software Engineering Technical Committee of the IEEE Computer Society.

Nancy G. Leveson, Software Safety: Why, What, and How, ACM Computing Surveys, Vol. 18, No. 2, June 1986.

Dr. Siba N. Mohanty, Software Verification and Validation: A Tutorial.

Space Station Freedom Program Office, Reston, VA, Space Station Program Definition and Requirements Document, SSP 30000.

Test and Evaluation Handbook, Third Edition, July 1985, edited by W.J. Riegger, Test and Evaluation Coordinator for the Space & Naval Warfare Systems Command.

Richard Zultner, The Deming Approach to Software Quality Engineering, Quality Progress, November 1988.

PRECEDING PAGE BLANK NOT FILMED

APPENDIX C

ACRONYMS AND ABBREVIATIONS

AIS	Automated Information Security
AR	Acceptance Review
CDR	Critical Design Review
CM	Configuration Management
DID	Data Item Descriptions
FMEA	Failure Modes and Effects Analysis
FRR	Flight Readiness Review
IV&V	Independent Verification and Validation
MTTR	Mean-Time-To-Repair
MTTF	Mean-Time-To-Failure
NMI	NASA Management Instruction
NRCA	Nonconformance Reporting and Corrective Action
PDR	Preliminary Design Review
SMAP	Software Management and Assurance Program
SQA	Software Quality Assurance
SQE	Software Quality Engineering
SRR	Software Requirements Review
TRR	Test Readiness Review
V&V	Verification and Validation

PRECEDING PAGE BLANK NOT FILMED

INDEX

Term	Page
Acceptance and Delivery Phase	13, 25
Acceptance Review	25
Acceptance Test Plan	24
Acquirer	4,5
Analysis	15, 33
Analyzers	13, 18-19
Architectural Design Phase	11, 24
Audit	3, 5, 7, 8, 9, 12, 13, 17, 18, 29, 36
Automated Information Security	35
Baseline	9, 27, 28
Black Box Testing	22
Change Request	29
Code Analysis	33
Code Standards	7, 17
Completion Criteria	5
Components	11, 32, 33
Testing	22
Concept and Initiation Phase	4, 11, 23
Configuration Audit	13
Configuration Authentication	9
Configuration Control	9
Configuration Identification	9
Configuration Item	28
Configuration Management	8-9, 10, 12, 29, 36
Relationship to SQA	8-9
Configuration Status Accounting	9
Corrective Action	1, 7, 12, 27-29, 36
Criteria	5, 7
Critical Design Review (CDR)	10, 12, 21, 24
Criticality Analysis	32
Design Analysis	32, 33
Design Constraint Analysis	33
Design Data Analysis	33
Design Interface Analysis	33
Design Logic Analysis	33
Design Standards	4, 7, 12
Detailed Design Phase	12, 24
Development Plan	12
Documentation Standards	7, 17
Efficiency	15, 16-17

INDEX (Cont.)

Term	Page
Failure Modes and Effects Analysis	18
Fault	18, 27, 33
Fault Tolerance Engineering	18
Formal Inspection	26
Formal Reviews	10, 21
Formal Test Monitoring	10
Formal Tests	23
Hazard Analysis	32
Impact Assessment	28
Implementation Phase	5, 12, 25
Independent Development	18
Independent Verification and Validation	25, 26
Informal Reviews	21, 24
Informal Testing	22, 26
Input Data Checking	18
Inspections	9, 10, 12, 21, 24-26
Integration and Test Phase	12, 25
Interoperability	16
Life Cycle	1, 7, 10, 11-13, 21, 23-25, 26, 27, 28, 29, 35
NRCA Activities during	27-28
Security Assurance Activities during	35
SQA Activities during	11-13
V&V Activities during	23-25
Maintainability	15, 16-17
Management Plan	7-8, 11
Management Plan Documentation Standard	4, 23
Management Reports	8, 27, 28
Mean-Time-To-Failure (MTTF)	15
Mean-Time-To-Repair (MTTR)	15
Metrics	16, 19
Modular Software	17
N-Item Voting	18
NASA	1, 2, 6, 7, 28, 35
Nonconformance Analysis	17
Nonconformance Detection	27
Nonconformance Reporting and Corrective Action (NRCA)	1, 12, 27-29
Relationship to CM	29
Relationship to SQE	17-18, 29
Relationship to V&V	29
Relationship to SQA	29

INDEX (Cont.)

Term **Page**

Nonconformance Tracking and Reporting System 28

Preliminary Design Review 10, 12, 21, 24
Procedure 1, 5, 7-12, 17, 18, 21, 23, 25, 27, 28, 29, 33
Process Monitoring 7, 8
Processes 1
Product Evaluation 7, 8
Products 1, 21
Proof of Correctness 18
Provider 4, 5

Qualities 16-17
Quality Evaluations 17

Readability 16
Regression Testing 11, 23, 29
Reliability 15, 16-17
Reports 8-10, 12, 27-19
Requirements 1, 21, 22
Requirements Analysis 32
    Criticality Analysis 32
    Specification Analysis 32
    Timing, Sizing, and Throughput Analysis 32
Requirements Phase 11, 24
Reviews 9, 10, 21
Risks 1, 3, 4

Safety Critical 32, 33
Safety Testing 33
Security Assurance 35, 36
Software Acquisition Life Cycle 7, 11, 23
Software and System Safety 31-33
Software Assurance 1-6, 29
    Definition 1
    Establishing Project Activity 3
    Goals 3
Software Assurance Plan 4, 5
Software Attributes 16-17
Software Development Libraries 9
Software Engineering 1, 6
    Definition 1
Software Life Cycle 1, 27, 35
Software Management 1, 6
    Definition 1

INDEX (Cont.)

Term **Page**

Software Managment and Assurance Program 4, 23
Software Qualities 15-17
Software Quality 4, 7-13
Software Quality Assurance (SQA) 2, 4-5, 7-13, 29
    Definition 7
    Activities 8
Software Quality Engineering (SQE) 15-19, 29
    Definition 15
Software Quality Engineering Program 16
Software Safety 31-33
Software Safety Analysis 32
Software Safety Program 32-33
Specification Analysis 32
Standards 1, 3, 4, 5, 7-12, 17, 27, 29
    Code 7, 17
    Design 4, 7, 12
    Documentation 7, 17
Standards and Procedures 5, 7-9
Subsystem Integration Tests 22
Sustainability 15, 16
Sustaining Engineering and Operations Phase 13, 25
System Safety 31-33

Tailoring 3-4
Test Plan 23, 24, 33
Test Readiness Review (TRR) 10, 21, 25
Testability 15, 32
Testing 3, 10-11, 12, 15, 21, 22-23, 25, 26, 27, 29, 31, 32, 33
Timing, Sizing, and Throughput Analysis 32
Traceability 15
Transportability 16

Unit Test 5, 22, 24
Useability 15

Verification and Validation 9, 21-26, 29
    Definition 21
Verification Matrix 11, 24

Walkthroughs 9, 10, 21, 24, 25
White Box Testing 22-23

SOFTWARE ASSURANCE GUIDEBOOK CHANGE SUGGESTION FORM

From:_____ Code/Contractor:_____ Phone:_____

Version:_____ Chapter:_____ Section:_____ Page:_____

Suggested Change:

Rationale:

Forward to Donald W. Sova
Manager, SMAP
NASA Headquarters, Code QR
Washington, DC 20546