Assurance of Software Quality

SEI Curriculum Module SEI-CM-7-1.1 (Preliminary)

July 1987

Bradley J. Brown
Boeing Military Airplane Company

Carnegie Mellon University
Software Engineering Institute

This work was sponsored by the U.S. Department of Defense.

The Software Engineering Institute (SEI) is a federally funded research and development center, operated by Carnegie Mellon University under contract with the United States Department of Defense.

The SEI Education Program is developing a wide range of materials to support software engineering education. A **curriculum module** identifies and outlines the content of a specific topic area, and is intended to be used by an instructor in *designing* a course. A **support materials** package includes materials helpful in *teaching* a course. Other materials under development include textbooks and educational software tools.

SEI educational materials are being made available to educators throughout the academic, industrial, and government communities. The use of these materials in a course does not in any way constitute an endorsement of the course by the SEI, by Carnegie Mellon University, or by the United States government.

SEI curriculum modules may be copied or incorporated into other materials, but not for profit, provided that appropriate credit is given to the SEI and to the original author of the materials.

Requests for additional information should be addressed to the Director of Education, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213.

Comments on SEI materials are solicited, and may be sent to the Director of Education, or to the module author.

Bradley J. Brown
Manager, Software Quality Assurance
Boeing Military Airplane Company
P.O. Box 7730 M\S 32-24
Wichita, Kansas 67217

This technical report was prepared for the

SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position.
It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler
SEI Joint Program Office

ESD Report Number: ESD-TR-87-125

Assurance of Software Quality

Contents

Assurance of Software Quality

Module Revision History

Version 1.1 (July 1987)	format changes for title page and front matter
Version 1.0 (April 1987)	original version

Assurance of Software Quality

Capsule Description

This module presents the underlying philosophy and associated principles and practices related to the assurance of software quality. It includes a description of the assurance activities associated with the phases of the software development life-cycle (e.g., requirements, design, test, etc.).

Philosophy

This module presents the concepts underlying the assurance of software quality as a function of the software development process. Specifically, this module provides:

1. a basic understanding of the concept of quality as it relates to software
2. an explanation of the concept of software quality assurance as it relates to the software development process
3. an overview of the industry and government standards related to software
4. an examination of processes related to software defect reporting, resolution, and analysis
5. an explanation of requirements traceability and correlation, and the use of traceability to demonstrate of satisfaction of requirements
6. an examination of methods used to document assurance activities
7. an examination of the social factors involved in influencing the actions of persons despite an adversarial relationship

This module provides the concepts underlying the development, implementation, and maintenance of a software quality assurance program which assures that the process used in the development of software results in a product which complies with the specified requirements.

A module on Assurance of Software Quality is required in addition to the other modules treating software development because of the unique functional role inherent in the assurance of software quality. Most of the activities and functions in the software development process are product oriented and, as such, are nearly isolated from other activities and functions in the software development process. Assurance of software quality, on the other hand, is a process oriented function which is inherently involved with every activity and function of the software development process.

Objectives

A student who has worked through this module should be able to:

1. explain the concept of assurance of software quality, and discuss the relationship of software quality assurance to the phases of the software development process
2. discuss methods related to assurance of quality of software products
3. describe the concept of traceability, identify traceable products, and discuss schemes for implementing and using traceability
4. define the characteristics of software non-conformance reporting, identify related information analysis, and discuss non-conformance resolution including corrective action as to the cause of the non-conformance
5. demonstrate awareness of the social factors predominant in system non-conformance resolution and corrective action, and know how to achieve cooperation in spite of an adversarial relationship
6. identify the components of software docu-

mentation and related data, and specify control mechanisms for achieving the appropriate quality

7. be familiar with project, industry, and governmental standards, and understand their relationship to the software development process
8. discuss the components of a software quality assurance program, and understand how to align those components to a software development program with allowance for size, complexity, and other constraining factors associated with the program
9. demonstrate awareness of the social factors involved in implementing and maintaining a software quality assurance program

Since the actions associated with the assurance of software quality are highly dependent on the characteristics of the underlying software development program, this module cannot provide individual methods related to implementation or maintenance of a software quality assurance program. Instead, this module provides the concepts which serve as the basis for assurance of software quality and which provide the general knowledge required to develop specific software development project assurance methods.

Prerequisite Knowledge

Since the assurance of software quality is concerned with assuring the quality of the entire software development process, a general knowledge of the software development process is necessary to understand the relationship of software quality assurance functions to their associated software development functions. The minimum required knowledge should include a basic knowledge of:

1. software requirements definition and representation
2. software design methods and resulting documentation and data, specifically including maintenance and enhancement methods
3. software code representations and constraints
4. inspection, walkthrough, review, and audit conduct
5. test methods, including test case assessment
6. configuration management and configuration control
7. basic statistical methods
8. technical communication, including interpersonal and writing skills

Module Content

Outline

I. Introduction
  1. The Philosophy of Assurance
  2. The Meaning of Quality
  3. The Relationship of Assurance to the Software Life-cycle

II. Tailoring the Software Quality Assurance Program

III. Reviews
  1. Walkthrough
  2. Inspection
  3. Configuration Audits

IV. Evaluation
  1. Software Requirements
  2. Preliminary Design
  3. Detailed Design
  4. Coding and Unit Test
  5. Integration and Testing
  6. System Testing
  7. Types of Evaluations

V. Configuration Management
  1. Maintaining Product Integrity
  2. Change Management
  3. Version Control
  4. Metrics
  5. Configuration Management Planning

VI. Error Reporting
  1. Identification of Defect
  2. Analysis of Defect
  3. Correction of Defect
  4. Implementation of Correction
  5. Regression Testing
  6. Categorization of Defect
  7. Relationship to Development Phases

VII. Trend Analysis
  1. Error Quantity
  2. Error Frequency
  3. Program Unit Complexity
  4. Compilation Frequency

VIII. Corrective Action as to Cause
  1. Identifying the Requirement for Corrective Action
  2. Determining the Action to be Taken
  3. Implementing the Corrective Action
  4. Documenting the Corrective Action
  5. Periodic Review of Actions Taken

IX. Traceability

X. Records

XI. Software Quality Program Planning

XII. Social Factors
  1. Accuracy
  2. Authority
  3. Benefit
  4. Communication
  5. Consistency
  6. Retaliation

Annotated Outline

I. Introduction

1. The Philosophy of Assurance

The concept of Assurance of Software Quality is based on the principle of establishing good software engineering practices and monitoring adherence to those practices throughout the software development life-cycle. This results, to a large extent, in giving control of the software development process priority over control of the software product. It must be understood that quality cannot be the assigned function of any one person or organization; rather, it must be the primary responsibility of every person involved in the development of a product. The role of Software Quality Assurance, then, is to influence everyone to perform their function in a quality manner. The basis for this philosophy is that the consistent use of a quality process will result in a quality product.

2. The Meaning of Quality

A precise definition of *quality* is not important in order to understand the concept of *software quality assurance*. For the purpose of this module, *quality* is the presence of desired characteristics and the absence of undesirable characteristics in the product or

process. The preceding statement is not intended to be a definition of *quality* for use in all circumstances; it provides a basis for understanding which is necessary in order to discuss the concept of software quality assurance.

The *characteristics* whose absence or presence denote quality are completely dependent upon the situation surrounding each individual product. In essence, quality is relative. It is conceivable that a situation could occur where meeting schedule is more important than whether the item works. In this event, timeliness would be more important as a quality characteristic than would functionality. Although this may be an extreme example, the moral is that the actual quality characteristics are dependent upon each unique situation, and that *quality* is not a concrete, immutable concept referring to some unchanging characteristic.

3. The Relationship of Assurance to the Software Life-cycle

The function of Software Quality Assurance interacts to some degree with each phase of every software development process. Planning should occur in the initial phases of a software project and should address the methods and techniques to be used in each phase. A description of every product resulting from a phase and the attributes desired of each product should be defined in order to provide a basis for objectively identifying satisfactory completion of the phase.

II. Tailoring the Software Quality Assurance Program

Each software development effort is unique to some extent; even though some of the same methods and techniques can be used frequently, some differences between projects will almost always exist. Some factors that have a large impact on the software quality assurance program are:

- schedule requirements
- available budget
- technical complexity of the software product
- anticipated size of the software product
- relative experience of the labor pool
- available resources
- contract requirements

These and other factors determine the nature of the software quality assurance program. The initial planning of the software quality assurance program should identify how each of these factors will affect the program and determine how the program will be tailored to function effectively as a result.

III. Reviews

A technical review is a disciplined group process focused toward an extensive examination of a product or process. It derives a large portion of its efficacy from the combined expertise of the members of the group. A more extensive coverage of this subject can be found in The Software Technical Review Process module [Collofello86].

1. Walkthrough

A walkthrough is usually an informal, somewhat undisciplined, review of a software product; usually source code [Yourdon78].

2. Inspection

An inspection is a formal, disciplined review of all software products; not just source code [Fagan76], [Fagan86].

3. Configuration Audits

Final acceptance of a software product is frequently based on completing a set of configuration audits. These audits ensure that the product has satisfactorily met all of its applicable requirements.

a. Functional

The primary purpose of the Functional Configuration Audit is to ensure that the product that was tested to demonstrate compliance with contract requirements is essentially the same as the product that will be delivered. Conducting software tests frequently takes months or even years, during which time the software item being tested may undergo revisions and modifications. The Functional Configuration Audit should ensure that none of these revisions adversely affects the results of previous tests.

b. Physical

The primary purpose of the Physical Configuration Audit is to ensure that all of the requirements of the contract have been satisfied, with special emphasis on the documentation and data delivery requirements. This audit usually is performed after the Functional Configuration Audit has demonstrated that the item functions properly.

IV. Evaluation

An evaluation is usually performed by a single individual and is intended to ensure compliance with all applicable requirements for each software product. Although evaluation of software products is actually a software quality control function, it provides information regarding the software development process that may not be obtained effectively in any other manner. The actual evaluation of the software products may be performed by any organization or individual within the software development project. The following lists of evaluations to be performed during each phase of soft-

ware development are only suggestions. The actual products to be evaluated should be determined while planning the software quality assurance program.

1. Software Requirements

The activities and products of the software requirements phase should be examined throughout the conduct of this phase. This examination should evaluate the following:

- software development plan
- software standards and procedures manual
- software configuration management plan
- software quality program plan
- Software requirements specification
- interface requirements specification
- operational concept document

2. Preliminary Design

The activities and products of the preliminary design phase should be examined throughout the conduct of this phase. This examination should consist of the following evaluations:

a. All revised program plans

b. software top level design document

c. software test plan

d. operator's manual

e. user's manual

f. diagnostic manual

g. computer resources integrated support document

3. Detailed Design

The activities and products of the detailed design phase should be examined throughout the conduct of this phase. This examination should consist of the following evaluations:

- all revised program plans
- software detailed design document
- interface design document
- database design document
- software development files
- unit test cases
- integration test cases
- software test description
- software programmer's manual
- firmware support manual
- all revised manuals
- computer resources integrated support document

4. Coding and Unit Test

The activities and products of the coding and unit test phase should be examined throughout the conduct of this phase. This examination should consist of the following evaluations:

- all revised program plans
- source code
- object code
- software development folders
- unit test procedures
- unit test results
- all revised description documents
- integration test procedures
- software test procedure
- all revised manuals

5. Integration and Testing

The activities and products of the integration and testing phase should be examined throughout the conduct of this phase. This examination should consist of the following evaluations:

- all revised program plans
- integration test results
- all revised description documents
- revised source code
- revised object code
- revised software development files
- software test procedures
- all revised manuals

6. System Testing

The activities and products of the system testing phase should be examined throughout the conduct of this phase. This examination should consist of the following evaluations:

- all revised program plans
- system test report
- all revised description documents
- revised source code
- revised object code
- revised software development files
- software product specification
- version description document
- all manuals

7. Types of Evaluations

The following types of evaluations are based upon those found in DOD-STD-2168 Software Quality Program [DoD87]. Some or all of these evaluations apply to every software product. The software quality assurance program plan should specify which products are evaluated, and which evaluations are

performed on those products.

- adherence to required format and documentation standards
- compliance with contractual requirements
- internal consistency
- understandability
- traceability to indicated documents
- consistency with indicated documents
- appropriate requirements analysis, design, coding techniques used to prepare item
- appropriate allocation of sizing, timing resources
- adequate test coverage of requirements
- testability of requirements
- consistency between data definition and use
- adequacy of test cases, test procedures
- completeness of testing
- completeness of regression testing

V. Configuration Management

Software configuration management encompasses the disciplines and techniques of initiating, evaluating, and controlling change to software products during and after the development process. It emphasizes the importance of configuration control in managing software production. The Software Configuration Management module [Tomayko86] provides detailed information regarding the principles and procedures of software configuration management. An effective configuration management should control changes for all software products on a software development project, including specifically documentation, test reports, and software error reports. Software configuration management provides the foundation for all of the rest of the activities which occur during software development.

Although the need for a formalized methodology for software configuration management may not be apparent on smaller projects, the need quickly becomes crucial to success as the project grows even slightly in size. A team of three people may be able to maintain an *oral history* of the project, a feat which is patently impossible for a team of thirty people.

The functions of software configuration management that provide a basis for assuring software quality are:

1. Maintaining Product Integrity
2. Change Management
3. Version Control
4. Metrics
5. Configuration Management Planning

VI. Error Reporting

Unfortunately, no matter what software engineering techniques are used, errors seem to be a fact of life. Maintaining an effective error reporting system, however, will help minimize the potential impact of software errors. Every software development project should establish an error reporting system even if it consists of notes scribbled on the back of a dinner napkin. It takes valuable resources to detect each error, but they are wasted if they must be used to locate an error that had been previously detected.

An error reporting system should be tailored to the needs of the software development project. However simple or elaborate the system may be, it should address the following areas:

1. Identification of Defect

Each defect identified should be described in clear, precise terms. This description should usually be of the behavior of the system, although in some cases it may be more efficient to describe the actual defect in the software product. In any case, the description should be written to be understandable to persons somewhat unfamiliar with the specific software product, and to be understandable after time has passed. If appropriate care is taken in documenting errors, valuable data will be available in the future for analysis which could identify improved methods of developing or maintaining the software product.

2. Analysis of Defect

The severity of the defect and the difficulty of correcting the defect should be documented to provide a basis for determining resource allocation and scheduling defect correction priorities. Errors are frequently detected faster than they can be resolved, and performing an initial defect analysis can provide valuable information to project management for establishing priorities. Typically, the analysis can be performed much more rapidly than can identifying the actual correction and therefore should be considered as an independent operation within the error reporting system.

3. Correction of Defect

Documenting the correction of the defect is important to maintain proper configuration accounting. The description of the correction should include:

- a narrative description of the correction
- a list of program units affected
- the number, revision, and sections of all documents affected
- any test procedures changed as a result of the correction.

4. Implementation of Correction

Updates to software are frequently performed in *blocks* after a baseline has been established. What this essentially means is that error corrections are

identified by working with an engineering copy of the software, and then are incorporated *en masse* into the official baselined version of the software after a certain number of errors have been corrected. Identification of which error was corrected in which version of software is important. Recording the implementation of the correction makes this possible. The description of the implementation should include:

- the version in which the correction was incorporated
- the authority for incorporating the correction.

5. Regression Testing

Retesting the affected function is necessary after the change is incorporated since as many as 20 percent of all corrections result in additional errors. Frequently, additional functions will need to be tested to ensure that no latent defects were induced by the correction. In the event that latent defects were induced by the correction, one method of resolution would be to treat them as new errors and initiate a new error report. The description of regression testing should include:

- a list of test paragraphs/objectives retested
- the version of software used to perform regression test
- indication of successful/unsuccessful accomplishment of test.

6. Categorization of Defect

Errors can frequently be grouped into categories which will allow future data analysis of errors encountered. The most efficient time to categorize them is usually as they are resolved while the information is still fresh. Possible classifications for error categorization include:

- *error type*—requirements, design, code, test, etc.
- *error priority*—no work around available, work around available, cosmetic.
- *error frequency*—recurring, non-recurring.

7. Relationship to Development Phases

The software error reporting system should be designed to change in complexity as the complexity of the software development project changes. The configuration control requirements for the products of each phase increase with the maturity of that phase until, at phase completion, each product is placed under rigid change control with requested changes requiring a specified authorization prior to implementation. The software error reporting system should also change in complexity to match the program; the system should be simple for products under engineering control, more complete for products which have been baselined, and even more complete for products which have been delivered to a customer.

VII. Trend Analysis

Analysis of trends in the performance of work can help to avoid the development of a non-conforming product. Trend analysis is a passive activity in that it provides information indicating that corrective actions maybe necessary, and in some cases may even suggest appropriate corrective actions, but does not affect the software development process itself. Essentially, trend analysis essentially refers to a form of data analysis where time is represented as one of the elements of a report. The data that comprise the other elements of the *trend* report determine the nature and ultimate utility of the report. Trend reports are extremely phase dependent in that a report which is valuable while preparing requirement specifications may be useless during system testing. Some of the possible report types are:

1. Error Quantity

Error quantity reports frequently plot the quantity of errors versus the time of initiation and the time of closure. This report can be based on cumulative quantity or instantaneous quantity. Separating the errors into major functions or by responsible persons can be useful in locating the source of unusual quantities of errors. Statistical process control methods can even be used to provide upper and lower bounds, or *critical limits*, which can identify whether further training of the responsible persons is necessary or beneficial.

2. Error Frequency

Error frequency charts report the quantity of errors per unit of software product. The unit used may be a section of a requirements specification, a test procedure paragraph, a source code program unit, or any other objectively identifiable component of a software software. The utility of an error frequency report is based on the Pareto Principle of non-homogeneous error distribution. If errors are non-homogeneously distributed in the product to be examined, then units with high detected error frequencies will probably also have a larger than normal number of latent errors.

3. Program Unit Complexity

Various metrics have been developed for measuring the relative complexity of software source code and have been verified to have some correlation to error frequency, e.g., McCabe's cyclomatic complexity metric, or Halstead's information metric. A complexity metric which has demonstrated a correlation to error frequency could be used in the early phases of a project to identify units of unusually high complexity as candidates for simplification through

redesign.

4. Compilation Frequency

Compilation frequency is an example of a report which, on the surface, may appear to be trivial. DeMarco, however, indicates that for whatever reason—and there are identifiable reasons—program units which are compiled frequently during design are also compiled frequently during integration and system test. Therefore, a unit which has been compiled three standard deviations above the mean number of compilations would be a prime candidate for evaluation as to the reason for the unusually high compilation frequency.

VIII. Corrective Action as to Cause

The purpose of a corrective action system is to eliminate recurring errors by correcting the problem that caused the errors. Identifying the root cause of recurring defects is frequently difficult, but unless a concerted action is taken to correct the root cause that was identified, the effort is in vain. An organized corrective action system should provide for the following:

1. Identifying the Requirement for Corrective Action

The initial activity of any corrective action system is to determine when a corrective action would be beneficial. Several methods are available for establishing standard thresholds for non-conformances prior to a required corrective action. Some of these methods are as follows:

a. Comprehensive

Performing comprehensive corrective action on every non-conformance is probably the simplest method. No real decision needs to be made other than whether it is a non-conformance or not. The main disadvantage of this method is the inappropriate allocation of resources. Not all non-conformances are of the same magnitude and not all non-conformances have a recurring root cause; they are frequently non-recurring minor errors. However, depending on the criticality of the project, it may be desirable to at least examine every error for potential corrective action.

b. Error frequency

The frequency of errors within a unit of a software project may be used as a basis for a standard threshold. Statistical process control techniques can be used to determine the number of errors permissible before requiring corrective action. A simpler method, however, is to list the units in descending order by frequency of defects, then start from the top of the list and work down. This method ensures that corrective action resources are always focused on the errors having the greatest frequency. It might be desirable to establish a lower limit below which corrective action is not required.

c. Error magnitude

A standard threshold can also be established on the basis of error magnitude. This would focus the corrective action resources on the errors which have the largest impact on the project. An example would be to set a requirement that, after the start of system test, corrective action be mandatory for all errors which halt system test conduct. The difficulty with a system of this type lies in the inherent subjectivity of determining error magnitude.

d. Statistical sampling

A statistical sample of all errors could be selected at random for required corrective action. The benefit of this method would be reduction in resource allocation required over that of comprehensive corrective action. This benefit may be false, however, in that the number of errors not addressed and thus recur may require more resources than would a more focused corrective action system.

2. Determining the Action to be Taken

Determining the action to be taken to correct the root cause of a recurring defect requires disciplined analysis. Factors such as resource availability and anticipated political resistance should be taken into account during the assessment of the action to be taken. Possible actions could consist of additional training for the individuals involved, an improvement in unit test methods, or even scrapping the unit and starting over from scratch.

3. Implementing the Corrective Action

Implementing the identified corrective action is often the responsibility of someone other than the person who identified the action. Proper coordination should exist to ensure that the people involved understand the purpose and expected benefit of implementing the corrective action.

4. Documenting the Corrective Action

A disciplined method of documenting corrective actions is necessary to ensure that the corrective action system is effective. This method should ensure that the errors for which corrective action is required are identified, that the action to be taken is recorded, and that the implementation of the corrective action is documented. Provisions should also be considered for establishing an end date for corrective actions of a temporal nature.

5. Periodic Review of Actions Taken

No one likes to perform corrective action, and if the identified corrective action is not embedded into an existing system, it is easy to fail to comply. In order for a corrective action system to be effective, it must ensure that continued corrective action is maintained through a system of periodic review. Once the action has been implemented, provisions should exist to review the action after a week or maybe a month has passed to ensure that the individuals responsible for performing the action understand their responsibilities and are maintaining the corrective action.

IX. Traceability

The principle of traceability is that every software product should be traceable back to the product from which it was derived. With effective traceability, it should be possible to identify the requirement or design decision from which each algorithm in the software product was derived. Test procedures should be traceable to the requirement or design for which they demonstrate product compliance. Traceability provides for ease in determining phase completion and product completeness. It supports the accomplishment of reviews and evaluations, and provides for increased confidence in the accuracy of requirements verification. Also, effective traceability can assist in ensuring that test procedures are updated whenever errors are discovered which were undetected by the applicable procedure.

X. Records

Maintaining an effective software quality assurance program requires a disciplined method of handling the records processed—software error reports, product evaluation checklists, configuration records, review reports, corrective action records, etc. Each record should have a means of unique identification to ensure that it can be conveniently referenced. Retention requirements should be established for each type of record to ensure that they are stored in the appropriate manner and for the appropriate time necessary depending on their criticality.

XI. Software Quality Program Planning

Before starting a software project, and then throughout all phases of the project, planning should be done to ensure that the needs of the project are addressed. Every technique, method, record, and system should be established as necessary to support the project and then discontinued when no longer necessary. A technique that can be helpful in planning a software quality assurance program for a new project is to use an old plan or a data item description for a plan as a checklist to ensure that all possible items are considered in the new plan.

XII. Social Factors

Social factors play an important role in the application of quality assurance in a real world environment. As a function, assurance is not concerned with the software product per se, but with the process which produces that product. Since the software development process is composed of humans, the interaction with that process must therefore be through humans. Social factors are those concepts that must be considered when interacting with humans in the role of assuring software quality.

1. Accuracy

Accuracy of data is of paramount importance when presenting any information as a result of a software quality assurance activity. Credibility is difficult to achieve and can easily be lost through an inadvertent misstatement or a minor miscalculation. One of the most frequent defenses used to avoid changing the status quo, and unfortunately a very effective one, is that the data used to show the need for the change is inaccurate. This defense is often taken to the extreme where a mistake in a presentation years ago is used as a basis for saying that a current report is inaccurate. Under the pressure of a management review there is no time to demonstrate whether the data is correct or not, and the credibility of the organizations/people involved in the eyes of the management making the decisions is the determining factor in what those decisions are. Even though reports may frequently have to be prepared under a tight schedule, always take time to ensure the data is accurate.

2. Authority

Individuals involved in assuring software quality may be designated as having authority in some company procedure or policy statement, but the authority necessary to influence changes despite an adversarial relationship can rarely be enforced through management action. If the individuals involved are not competent to influence changes or the organization involved has insufficient credibility to gain management support—especially adversarial management support—then rarely can any amount of high-level management direction or corporate policy avail.

Each person has the authority that they are capable of assuming and for which they are willing to be responsible. Authority is gained through demonstrating one's competence to the persons whom one is interested in influencing. This competence is not just a matter of credentials, but a matter of whether the changes are really beneficial to the company as a whole or self serving to the person interested in getting them implemented.

3. Benefit

An effective program for assuring software quality will provide an overall benefit even though it will occasionally create temporary hardships for individ-

ual persons or organizations. It is important to emphasize the benefits which will result from each action. It is also important that persons responsible for software quality assurance be alert for opportunities to help individual persons and organizations whenever possible. If the process of collecting data to prepare a report results in the compilation of data necessary to produce certain system documentation, it might be beneficial to provide that data to the organization responsible for preparing the documentation, thus saving them duplication of effort. It is important, however, to maintain objectivity so that effective evaluations can be performed.

4. Communication

Assurance of Software Quality consists essentially of communicating information. Every evaluation or verification performed essentially consists of assimilating information and providing feedback as a result of that information; in other words, communication. Many forms of communication are used, but the primary forms are speech and writing. It is important that people who are responsible for assuring software quality be proficient at communication.

5. Consistency

It is important that actions taken and decisions made be consistent. If direction changes frequently, for whatever reason, the persons responsible for following that direction become confused and quickly learn not to follow any direction unless it is what they want. No procedure or policy regarding software quality assurance should be set in place unless the means and the desire exist to ensure that it is adhered to consistently and for as long as is necessary. Establishing standards and requirements and then allowing them to be disregarded or ignored damages the credibility of all standards and requirements.

6. Retaliation

A person responsible for assuring software quality will frequently encounter personal abuse merely because of the responsibilities of the position. Many opportunities are available to that person to retaliate due to the inherent responsibilities of the position. Although it is almost always difficult, retaliation should always be avoided. Personal vendettas usually result in tremendous losses to the company or project and vastly outweigh whatever personal satisfaction is gained. If revenge is frequently sought, the persons involved will soon lose whatever credibility they have and in so doing will destroy their value to the company.

Teaching Considerations

Exercises

These exercises will give the student a greater understanding of the concepts underlying the assurance of software quality. Although they can be accomplished individually, all of these exercises will benefit by association with a non-trivial software development project that involves many students and that demonstrates good software engineering principles.

Reviews. Have the student establish a system for performing walkthroughs and/or inspections on software products. The system should define what products are to be examined and should include criteria for completing the reviews. The records used to document accomplishment of reviews should be developed or described. The system should include provisions for identifying non-conformances and possibly even reassuring them. A method should exist to verify objectively the status of review conduct and to ensure that all applicable products are reviewed. The review of revisions and updates should be addressed.

Evaluations. Have the student prepare an evaluation plan for a single software product, e.g., a Software Requirements Specification. This plan should address methods for objectively verifying the presence or absence of the desired characteristics of the product. The forms and records used to document accomplishment of the review should be developed or described. Non-conformances should be addressed identified and possibly corrected.

Configuration Management. Have the student describe a system which would ensure that product integrity and change control are maintained throughout the software development process. This method should ensure that the appropriate change authorization has been received prior to implementing the change during each applicable phase of the software development project. If desired, this system could be restricted in application to source code to limit the magnitude of the system. This system should include records or forms used to document any verification activities.

Software Error Reporting. Have the student develop a system for collecting and reporting software error data. The system should accurately identify the software error, record the analysis of the cause of the error, document the correction of the defect, and record retest activities accomplished to demonstrate successful correction of the defect. The system should also provide for error status reporting. Additional refinements can be included, such as identification of errors in all software-related products (e.g., manuals, specifications) identification of the root cause of the defect, and corrective actions to eliminate the root cause.

If the system is implemented with a software development project, the student should prepare a summary report a suitable amount of time after the system is in use, describing the difficulties encountered in implementing the system and the location of suspected inaccurate data within the system. If grading pressure or other motivational influences are placed upon the software development project for schedule accomplishment, etc., the student should notice an associated influence on the accuracy of the software error data.

Trend Analysis. Have the student develop a system for data collection and analysis which will detect trends in the performance of work which could lead to a non-conforming product. The method used to verify the usefulness of the analysis in regard to correlation to software non-conformances should be addressed.

An alternative would be to have the student identify various methods of data analysis which could be used for trend analysis. Emphasis could be placed on using data other than software error data as the basis of the analysis.

Corrective Action. Have the student establish a methodology for determining the threshold for permissible non-conformances prior to required corrective action. A description of how *non-conformances* are identified and tracked should be included. The method should be objective, repeatable, and verifiable. The student should also develop a way of verifying that the proposed method correlates in some way with defect frequency.

As a more advanced exercise, have the student develop the entire corrective action system. Reports and forms should be prepared with procedures describing their initiation and use. The system should include provisions for evaluating its relative efficacy and should provide supporting data for a cost/benefit analysis.

Traceability. Have the student develop a standard

method for establishing traceability within a software development project's documentation and products. The method should include provisions for discrete identification of requirements and correlation of requirements to their applicable references. The method should provide for independent verification of the accuracy of the various data products.

Records. Have the student develop a system for identifying and controlling of records used for assurance of software quality. The system should ensure that the records are complete and accurate and should address retention requirements and storage.

Software Quality Program Plan. Developing an entire Software Quality Program Plan is an ambitious exercise, however, it could be accomplished as an incremental, multi-semester, team project. The initial team could develop the framework for the entire plan and the detailed procedures for the Requirements Analysis phase. Successive teams could then complete the detailed procedures for each of the successive phases. Attention should be given configuration management of the Software Quality Program Plan document as a software product. Correlation between the SQPP and the associated Software Development Plan, Software Configuration Management Plan, and Software Test Plan should be maintained to prevent conflicting provisions or unnecessary duplication of provisions.

Human Factors. Have the student observe an ongoing software development project for the effect of human factors on the project's productivity. Have the student conduct interviews and collect data (such as memos) that indicate the influence of human factors on the project. A report should be prepared describing instances observed; if possible, have the student evaluate possible methods of reducing adverse effects caused by human factors. A hint is that *human factors* can also be referred to occasionally as *politics*.

Bibliographies

Books

DeMarco78

DeMarco, T. *Structure Analysis and System Specification.* Yourdon Press, 1978.

DeMarco82

DeMarco, T. *Controlling Software Projects.* Yourdon Press, 1982.

This book is possibly one of the best treatises on the subject of metrics, even though it never actually uses the term *metric*. It also provides a realistic perspective on the psychology of software project management.

Evans84

Evans, M. W. *Productive Software Test Management.* John Wiley, 1984.

Machiavelli13

Machiavelli, N. *The Prince.* Bantam Books, Inc., 1981. First published in 1513.

This book presents several excellent concepts related to influencing people in spite of an adverse relationship. It was written to explain how a prince should control his principality, but with only a minor change in view point, it also provides a remarkably incisive commentary on interpersonal and interorganizational relationships.

Myers79

Myers, G. J. *The Art of Software Testing.* John Wiley, 1979.

This is a landmark book on the principles of software testing. The self-assessment given in the foreword of the book provides real enlightenment regarding the difficulty of developing comprehensive test cases.

Quirk85

Verification and Validation of Real-time Software. W. J. Quirk, ed. Springer-Verlag, 1985.

Yourdon78

Yourdon, E. *Structured Walkthroughs.* Yourdon Press, 1978.

Papers

Adrion82

Adrion, W. R., M. A. Branstad, and J. C. Cherniavsky. Validation, Verification, and Testing of Computing Software. *Computing Surveys 14*, 2 (June 1982), 334-367.

Abstract: Software quality is achieved through the application of development techniques and the use of verification procedures throughout the development process. Careful consideration of specific quality attributes and validation requirements leads to the selection of a balanced collection of review, analysis, and testing techniques for use throughout the life cycle. This paper surveys current verification, validation, and testing approaches and discusses their strengths, weaknesses, and life-cycle usage. In conjunction with these, the paper describes automated tools used to implement validation, verification, and testing. In the discusssion of new research thrusts, emphasis is given to the continued need to develop a stronger theoretical basis for testing and the need to employ combinations of tools and techniques that may vary over each application.

Boger85

Boger, D. C., and N. R. Lyons. The Organization of the Software Quality Assurance Process. *Data Base (USA) 16*, 2 (Winter 1985), 11-15.

Abstract: This paper discusses and analyzes approaches to the problem of software quality assurance. The approaches offered in the literature usually focus on designing in quality. This can be a productive approach, but there are also benefits to be gained by establishing an independent quality assurance (QA) group to review all aspects of the software development process. This paper discusses the organization of such a group using the function of an operations auditing group as a model.

Bowen80

Bowen, J. B. Standard Error Classification to Support Software Reliability Assessment. *Proc. AFIPS 1980 National Computer Conference.* May, 1980.

Abstract: A standard software error classification is viable based on experimental use of different schemes on Hughes Fullerton projects. Error classification schemes have prolifereated independently due to varied emphasis on depth of casual

traceability and when error data was collected. A standard classification is proposed that can be applied to all phases of software development. It includes a major casual category for design errors. Software error classification is a prerequisite both for feedback for error prevention and detection, and for prediction of residual errors in operational software.

Buckley86

Buckley, F. J. The Search for Software Quality, or One More Trip Down the Yellow Brick Road. *ACM Software Engineering Notes 11*, 1 (1986), 16-18.

Abstract: This paper takes a look at the current expressions of the need for increased Software Quality, and provides a transform for a portion of it into Software Productivity. Some of the quick responses to these concerns are examined and discarded, and an overall management approach to meet these needs is prescribed.

A lighthearted view of the pitfalls of a naive approach to software quality assurance. Buckley gives an almost cynical view of the role of software quality assurance in relation to a software development program.

Chusho83

Chusho, T. Coverage Measure for Path Testing Based on the Concept of Essential Branches. *J. Info. Processing 6*, 4 (1983), 199-205.

Abstract: A new coverage rate based on essential branches (full coverage of all branches) is proposed for efficient and effective software testing. The conventional coverage measure for branch testing has defects such as overestimation of software quality and redundant test data selection, because all branches are treated equally. In order to solve these problems, concepts of essential branches and nonessential branches for path testing are introduced. Essential branches and nonessential ones are called primitive and inheritor arcs, respectively, in a control flow graph of a tested program.

A reduction algorithm for transforming a control flow graph to a directed graph with only primitive arcs is presented and its correctness is proved. Furthermore, it is experimentally and theoretically ascertained that the coverage measure on this inheritor-reduced graph is nearly linear to the number of test cases and therefore suitable for software quality assurance.

Ciampi78

Ciampi, P. L. Software Error Patterns—A Personal Case History. *Proc. 3rd USA/JAPAN Computer Conference.* Oct., 1978, 176-181.

Abstract: This paper is a personal account of software errors and how they could have been avoided. It provides detailed data that supports the importance of reliable software techniques. The data represents 73 errors that occurred in a file management system and its related interface programs. These errors fall into 13 groups. The relations between each group and violation of software engineering ideas supports the major conclusions that; programmers should record and analyze their errors, and software engineering ideas reduce errors.

Collofello85

Collofello, J. S., and L. B. Balcom. A Proposed Causative Software Error Classification Scheme. *Proc. 1985 AFIPS National Computer Conference.* AFIPS, July, 1985, 537-545.

Abstract: Various tools, techniques, and methodologies have been developed by software engineers over the last 15 years. A goal of many of these approaches is to increase product reliability and reduce its cost by decreasing the number and severity of errors introduced by the software development process. The collection of software error data would appear to be a natural means for validation of these software engineering techniques. Yet, current software error collection efforts have had limited success in this area. A new causative software error classification scheme is introduced in this paper to refine these data collection efforts so that they can be better used in software engineering validation studies.

Collofello86

Collofello, J. S. *The Software Technical Review Process.* Curriculum Module SEI-CM-3.1.0, Software Engineering Institute, Carnegie-Mellon University, Sept., 1986.

Day85

Day, R., and T. McVey. A Survey of Software Quality Assurance in the Department of Defense During Life-Cycle Software Support. *Proc. IEEE Conf. on Software Maintenance.* IEEE Computer Society, Nov., 1985, 79-85.

Abstract: This paper summarizes the authors' research into the state-of-the-practice of software quality assurance (SQA) in software organizations throughout the Department of Defense (DoD). Information was obtained through personal visits to a limited number of software facilities and by utilizing a SQA questionnaire that was mailed to 27 DoD software organizations. Twenty questionnaires were returned from Army, Navy, Marine Corps, and Air Force organizations involved in the Life Cycle Software Support (LCSS) process. The survey developed information regarding such topics as:

staffing; personnel qualifications; quality standards used; SQA responsibilites; error data collection; adequacy of existing documentation; staff composition; distribution of SQA effort; and workforce mix. In addition, a list of SQA "Lessons Learned" was developed from comments made by DoD quality managers. The results of this study indicate a wide variance in the application of SQA within the DoD.

DoD87

Defense System Software Quality Program. U.S. Army Electronics Research and Development Command, 1987.

This Military Standard provides direction for establishing software quality assurance programs for all sponsored software development contracts sponsored by the Department of Defense.

DownsS85

Downs, T. A Review of Some of the Reliability Issues in Software Engineering. *J. Electrical and Electronic Eng. 5*, 1 (March 1985), 36-48.

Abstract: This paper commences with a detailed discussion of the problems and difficulties associated with software testing. It is shown that large software systems are so complex that software companies are obliged to terminate the testing process and release such systems with every expectation that the software still contains many errors. The possibility of using statistical models as an aid to deciding on the optimum time to release software is discussed and several such models are described. The idea of "disciplined" programming as a means of reducing software error content is also described, and ancillary topics such as formal specifications and program proofs are discussed. Other concepts, such as fault-tolerant software and software complexity measures, are also briefly described. Finally, the implications of the fact that hardware is cheap and reliable and software is expensive and unreliable are discussed. It is argued that many designs currently in use defy engineering common sense.

Duran81

Duran, J. W., and J. J. Wiorkowski. Capture-Recapture Sampling for Estimating Software Error Content. *IEEE Trans. Software Eng. SE-7*, 1 (Jan. 1981), 147-148.

Abstract: Mills' capture-recapture sampling method allows the estimation of the number of errors in a program by randomly inserting known errors and then testing the program for both inserted and indigenous errors. This correspondence shows how correct confident limits and maximum likelihood estimates can be obtained from the test results. Both fixed sample size testing and sequential testing are considered.

Fagan76

Fagan, M. E. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems J. 15*, 3 (1976).

This is the landmark paper on software inspections. It presents the basic methodology for a disciplined approach to identifying and correcting defects through a visual examination of the product.

Fagan86

Fagan, M. E. Advances in Software Inspections. *IEEE Trans. Software Eng. SE-12*, 7 (1986).

Abstract: This paper presents new studies and experiences that enhance the use of the inspection process and improve its contribution to development of defect-free software on time and at lower costs. Examples of benefits are cited followed by descriptions of the process and some methods of obtaining the enhanced results.

Software Inspection is a method of static testing to verify that software meets its requirements. It engages the developers and others in a formal process of investigation that usually detects more defects in the product—and at a lower cost—than does machine testing. Users of the method report very significant improvements in quality that are accompanied by lower development costs and greatly reduced maintenance efforts. Excellent results have been obtained by small and large organizations in all aspects of new development as well as in maintenance. There is some evidence that developers who participate in the inspection of their own product actually create fewer defects in future work. Because inspections formalize the development process, productivity and quality enhancing tools can be adopted more easily and rapidly.

Fay85

Fay, S. D., and D. G. Holmes. Help! I Have to Update an Undocumented Program. *Proc. IEEE Conf. on Software Maintenance.* Nov., 1985, 194-202.

Abstract: This paper discusses a method for documenting and maintaining an undocumented program. The paper provides guidance to junior personnel and management of areas that can alleviate the situation.

The paper specifically addresses:

- *First Impressions*
- *Resources, Who and What*
- *Approaches*

- *Schedule Assessment*

This paper is directed to those people in industry who are faced with documenting an undocumented program. However, it is also written with the hope that this will give the person supervising the maintainer a clearer view of the help which can be given by providing the resources and time necessary to maintain a program in the proper manner.

Hamlet82

Hamlet, R. Program Maintenance: A Modest Theory. *Proc. 15th Hawaii Intl. Conference on System Sciences.* Jan., 1982, 21-26.

Abstract: Design methods do not carry over into a program's life once it is released. The subsequent "maintenance phase" is thought to dominate the cost and poor quality of software. The only existing maintenance theory is mini-development: programs are changed in the same way they are designed, beginning with requirements and proceeding to testing. Maintenance programmers are impatient with such a view, because the constraints under which they work make it impractical.

The world defines maintenance as an activity with low unit cost, appropriate when development is too expensive. Some facts about real maintenance need explaining:

1. *Some people have a talent for it; others do not.*
2. *Some programs are much easier to maintain than others.*
3. *Maintenance becomes progressively harder to do as more is done, until finally any program becomes unmaintainable.*
4. *Testing of maintenance changes seems easier than initial development testing.*
5. *Maintenance documentation is different than design documentation.*

Herndon78

Herndon, M. A. Cost Effectiveness in Software Error Analysis Systems. *Proc. Second Software Life Cycle Management Workshop.* Aug., 1978.

Abstract: Software error analysis systems must have the capability of functioning as both a cost effective and valuable managerial tool. To achieve this capability, the design of the data collection must reflect the individual project's managerial concerns, and the resulting empirical analysis should be available for long term access.

IEEE81

ANSI/IEEE Std. 730-1981, IEEE Standard for Software Quality Assurance Plans. American National Standards Institute, 1981.

This is a list of section headings to give an outline of the content of a Software Quality Assurance Plan. It provides some detail as to suggested content of the sections.

McCall81

McCall J., D. Markham, M. Stosick, and R. McGindly. The Automated Measurement of Software Quality. *Proc. COMPSAC 81*. IEEE, 1981, 52-58.

Abstract: This paper describes the use of automated tools to support the application of software metrics. A prototype tool has been developed under contract to US Air Force Rome Air Development Center and US Army Computer Systems Command Army Institute for Research in Management Information and Computer Science. A brief description of the concept of software quality metrics, the tool, and its use during a large scale software development is provided.

Morse86

Morse, C. A. Software Quality Assurance. *Measurement and Control 19* (1986), 99-104.

This paper introduces the subject of software quality assurance to a wider audience of engineers so they may appreciate why software quality assurance has a place of importance in the software process and therefore must be considered seriously for all software projects.

Perry85

Perry, D. E., and W. M. Evangelist. An Empirical Study of Software Interface Faults. *Proc. Intl. Symp. on New Directions in Computing.* IEEE Computer Society, Trondheim, Norway, Aug., 1985, 32-38.

Abstract: We demonstrate through a survey of the literature on software errors that the research community has paid little attention to the problem of interface errors. The main focus of the paper is to present the results of a preliminary empirical study of error reports for a large software system. We determined that at least 66% of these errors arose from interface problems. The errors fell naturally into fifteen separate categories, most of which were related to problems with the methodology.

Pope83

Pope, A. B. Software Configuration Management: A Quality Assurance Tool. *Proc. 1983 IEEE Engineering Management Conference.* Nov., 1983.

Abstract: The literature of computer system development makes a strong case that many development failures or problems are caused by documentation. Many of these problems are the result of poorly

defined requirements which are changed without control. The original cost and schedule are based on developer perceptions as to the requirements and then changes are agreed to in an uncontrolled manner. Software Quality Assurance has the responsibility for ensuring complete requirements definitions as well as controlling changes to requirements and design; and tracking the resulting impact on cost and schedule. Software Configuration Management is the Quality Assurance tool for development project communications and tracking changes in cost and schedule.

Poston84
Poston, R. M. Implementing a Standard Software Quality Assurance Program. *Proc. Third Software Engineering Standards Application Workshop.* IEEE, 1984, 38-44.

Abstract: Software Quality Assurance Programs represent one approach to improving product quality and increasing productivity. Software Quality Assurance is defined in ANSI/IEEE 730 as the planned and systematic pattern of all *actions necessary to assure that products will function as specified. This implies that an SQA Program will affect everything involved in the creation of software. This paper describes one approach to implementing an SQA Program. The approach has evolved over seven different projects and has been only slightly modified on the last four projects.*

The approach presented in this paper is essentially *figure out what you want to do and then implement it incrementally.* This is a good approach as long as one understands that the basic Software Quality Assurance Program framework must still be determined before implementation.

Puhr83
Puhr-Westerheide, P., and B. Krzykacz. A Statistical Method for the Detection of Software Errors. *Proc. 3rd IFAC/IFIP Symp. on Software for Computer Control.* Oct., 1982, 383-386.

Abstract: A statistical software error detection method is presented that relies on some structural properties of the test object. The main idea is to select test paths by means of transition probabilities between the nodes of the control flow graph so that all paths have equal probabilities to be drawn. The transition probabilities can be determined from the control flow graph of a program in a simple way. This selection process turns out to have good statistical properties.

Sneed82
Sneed, H. M., and A. Merey. Automated Software Quality Assurance. *Proc. COMPSAC 82.* 1985, 909-916.

Abstract: This paper describes a family of tools which not only supports software development, but also assures the quality of each software product from the requirements definition to the integrated system. It is based upon an explicit definition of the design objects and includes: specification verification; design evaluation; static program analysis; dynamic program analysis; integration test auditing; and configuration management.

Soi78
Soi, I. M., and K. Gopal. Error Prediction in Software. *Microelectronics and Reliability 18* (1978), 433-436.

Abstract: Errors are introduced in software at all stages of the software production and the number of errors found during the software development phase affects significantly the cost of a project in terms of manpower and computer resources needed for correcting the errors. Early detection and correction of errors leads to substantial savings in cost. In this paper, causes, classifications and statistical behaviour exhibited by software errors have been discussed and a simple cost model which considers the use of a tool or technique to detect additional errors during the design phase and thereby save some of the greater expense of correcting the errors during the test phase has been discussed. It has been emphasized that inexpensive means of detecting and preventing errors applied during requirements analysis and design could significantly reduce the cost of a project.

Stamm81
Stamm, S. L. Assuring Quality, Quality Assurance. *Datamation ?* (1981), 195-200.

This paper describes an integrated Software Quality Assurance Program in use at General Electric's Space Division.

Tomayko86
Tomayko, J. E. *Software Configuration Management.* Curriculum Module SEI-CM-4.1.0, Software Engineering Institute, Carnegie-Mellon University, Sept., 1986.

Walker79
Walker, M. G. Auditing Software Development Projects: A Control Mechanism for the Digital Systems Development Methodology. *Proc. COMPCON 79.* IEEE, Spring, 1979, 310-314.

Abstract: This paper will introduce the audit as a control mechanism for the Computer Sciences Corporation (CSC) approach to system development.

The audit is performed on developing computer systems by a team independent of the developmental project. The independent audit is a cardinal feature of the Digital System Development Methodology (DSDM) which is the methodology CSC employs to engineer software systems. Audits are the most powerful mechanism for control built into the DSDM.

Yacobellis84

Yacobellis, R. H. Software and Development Process Quality Metrics. *Proc. 1st Data Engineering Conf.* IEEE Computer Society, 1984, 262-269.

Software projects that deliver a software or system product and involve from 30 to several hundred developers, testers, and project managers are considered. A description is given of a framework for gathering and reporting software and development process quality metrics, as well as data engineering issues related to data collection and report generation. The framework is described in project-independent terms, and a methodology for applying metrics to a given software project is included. A key aspect of this application is the use of project milestones predicted by a failure rate model.

Yamada86

Yamada, S., H. Narihisa, and H. Ohtera. Non-homogeneous Software Error Detection Rate Model: Data Analysis and Applications. *RAIRO Rech. Oper./Oper. Res. (France) 20*, 1 (1986), 51-60.

Abstract: A software reliability growth model called a nonhomogeneous error detection rate model is reviewed and applied to an actual data set of software failure occurrence time. In particular optimal software release policies with both software cost and software reliability requirements, i.e. cost-reliability optimal release policies, are discussed for the model. Using the numerical results of the data analyses, the cost-reliability optimal software release policy is illustrated.