General Principles of Software Validation; Final Guidance for Industry and FDA Staff

Document issued on: January 11, 2002

This document supersedes the draft document, "General Principles of Software Validation, Version 1.1, dated June 9, 1997.

U.S. Department Of Health and Human Services
Food and Drug Administration
Center for Devices and Radiological Health
Center for Biologics Evaluation and Research

Preface

Public Comment

Comments and suggestions may be submitted at any time for Agency consideration to Dockets Management Branch, Division of Management Systems and Policy, Office of Human Resources and Management Services, Food and Drug Administration, 5630 Fishers Lane, Room 1061, (HFA-305), Rockville, MD, 20852. When submitting comments, please refer to the exact title of this guidance document. Comments may not be acted upon by the Agency until the document is next revised or updated.

For questions regarding the use or interpretation of this guidance which involve the Center for Devices and Radiological Health (CDRH), contact John F. Murray at (301) 594-4659 or email jfm@cdrh.fda.gov

For questions regarding the use or interpretation of this guidance which involve the Center for Biologics Evaluation and Research (CBER) contact Jerome Davis at (301) 827-6220 or email davis@cber.fda.gov.

Additional Copies

CDRH
Additional copies are available from the Internet at: www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/UCM085281.htm.
You may also send an e-mail request to dsmica@fda.hhs.gov to receive an electronic copy of the guidance or send a fax request to 301-847-8149 to receive a hard copy. Please use the document number (938) to identify the guidance you are requesting.

CBER
Additional copies are available from the Internet at: http://www.fda.gov/cber/guidelines.htm, by writing to CBER, Office of Communication, Training, and Manufacturers' Assistance (HFM-40), 1401 Rockville Pike, Rockville, Maryland 20852-1448, or by telephone request at 1-800-835-5709 or 301-827-1800.

Table of Contents

General Principles of Software Validation

This document is intended to provide guidance. It represents the Agency's current thinking on this topic. It does not create or confer any rights for or on any person and does not operate to bind Food and Drug Administration (FDA) or the public. An alternative approach may be used if such approach satisfies the requirements of the applicable statutes and regulations.

SECTION 1. PURPOSE

This guidance outlines general validation principles that the Food and Drug Administration (FDA) considers to be applicable to the validation of medical device software or the validation of software used to design, develop, or manufacture medical devices. This final guidance document, Version 2.0, supersedes the draft document, *General Principles of Software Validation, Version 1.1*, dated June 9, 1997.

SECTION 2. SCOPE

This guidance describes how certain provisions of the medical device Quality System regulation apply to software and the agency's current approach to evaluating a software validation system. For example, this document lists elements that are acceptable to the FDA for the validation of software; however, it does not list all of the activities and tasks that must, in all instances, be used to comply with the law.

The scope of this guidance is somewhat broader than the scope of validation in the strictest definition of that term. Planning, verification, testing, traceability, configuration management, and many other aspects of good software engineering discussed in this guidance are important activities that together help to support a final conclusion that software is validated.

This guidance recommends an integration of software life cycle management and risk management activities. Based on the intended use and the safety risk associated with the software to be developed, the software developer should determine the specific approach, the combination of techniques to be used, and the level of effort to be applied. While this guidance does not recommend any specific life cycle model or any specific technique or method, it does recommend that software validation and verification activities be conducted throughout the entire software life cycle.

Where the software is developed by someone other than the device manufacturer (e.g., off-the-shelf software) the software developer may not be directly responsible for compliance with FDA regulations.

In that case, the party with regulatory responsibility (i.e., the device manufacturer) needs to assess the adequacy of the off-the-shelf software developer's activities and determine what additional efforts are needed to establish that the software is validated for the device manufacturer's intended use.

2.1. APPLICABILITY

This guidance applies to:

- Software used as a component, part, or accessory of a medical device;
- Software that is itself a medical device (e.g., blood establishment software);
- Software used in the production of a device (e.g., programmable logic controllers in manufacturing equipment); and
- Software used in implementation of the device manufacturer's quality system (e.g., software that records and maintains the device history record).

This document is based on generally recognized software validation principles and, therefore, can be applied to any software. For FDA purposes, this guidance applies to any software related to a regulated medical device, as defined by Section 201(h) of the Federal Food, Drug, and Cosmetic Act (the Act) and by current FDA software and regulatory policy. This document does not specifically identify which software is or is not regulated.

2.2. AUDIENCE

This guidance provides useful information and recommendations to the following individuals:

- Persons subject to the medical device Quality System regulation
- Persons responsible for the design, development, or production of medical device software
- Persons responsible for the design, development, production, or procurement of automated tools used for the design, development, or manufacture of medical devices or software tools used to implement the quality system itself
- FDA Investigators
- FDA Compliance Officers
- FDA Scientific Reviewers

2.3. THE LEAST BURDENSOME APPROACH

We believe we should consider the least burdensome approach in all areas of medical device regulation. This guidance reflects our careful review of the relevant scientific and legal requirements and what we believe is the least burdensome way for you to comply with those requirements. However, if you believe that an alternative approach would be less burdensome, please contact us so we can consider

your point of view. You may send your written comments to the contact person listed in the preface to this guidance or to the CDRH Ombudsman. Comprehensive information on CDRH's Ombudsman, including ways to contact him, can be found on the Internet at:

http://www.fda.gov/cdrh/resolvingdisputes/ombudsman.html.

2.4. REGULATORY REQUIREMENTS FOR SOFTWARE VALIDATION

The FDA's analysis of 3140 medical device recalls conducted between 1992 and 1998 reveals that 242 of them (7.7%) are attributable to software failures. Of those software related recalls, 192 (or 79%) were caused by software defects that were introduced when changes were made to the software after its initial production and distribution. Software validation and other related good software engineering practices discussed in this guidance are a principal means of avoiding such defects and resultant recalls.

Software validation is a requirement of the Quality System regulation, which was published in the Federal Register on October 7, 1996 and took effect on June 1, 1997. (See Title 21 Code of Federal Regulations (CFR) Part 820, and 61 Federal Register (FR) 52602, respectively.) Validation requirements apply to software used as components in medical devices, to software that is itself a medical device, and to software used in production of the device or in implementation of the device manufacturer's quality system.

Unless specifically exempted in a classification regulation, any medical device software product developed after June 1, 1997, regardless of its device class, is subject to applicable design control provisions. (See of 21 CFR §820.30.) This requirement includes the completion of current development projects, all new development projects, and all changes made to existing medical device software. Specific requirements for validation of device software are found in 21 CFR §820.30(g). Other design controls, such as planning, input, verification, and reviews, are required for medical device software. (See 21 CFR §820.30.) The corresponding documented results from these activities can provide additional support for a conclusion that medical device software is validated.

Any software used to automate any part of the device production process or any part of the quality system must be validated for its intended use, as required by 21 CFR §820.70(i). This requirement applies to any software used to automate device design, testing, component acceptance, manufacturing, labeling, packaging, distribution, complaint handling, or to automate any other aspect of the quality system.

In addition, computer systems used to create, modify, and maintain electronic records and to manage electronic signatures are also subject to the validation requirements. (See 21 CFR §11.10(a).) Such computer systems must be validated to ensure accuracy, reliability, consistent intended performance, and the ability to discern invalid or altered records.

Software for the above applications may be developed in-house or under contract. However, software is frequently purchased off-the-shelf for a particular intended use. All production and/or quality system software, even if purchased off-the-shelf, should have documented requirements that fully define its intended use, and information against which testing results and other evidence can be compared, to show that the software is validated for its intended use.

The use of off-the-shelf software in automated medical devices and in automated manufacturing and quality system operations is increasing. Off-the-shelf software may have many capabilities, only a few of which are needed by the device manufacturer. Device manufacturers are responsible for the adequacy of the software used in their devices, and used to produce devices. When device manufacturers purchase "off-the-shelf" software, they must ensure that it will perform as intended in their chosen application. For off-the-shelf software used in manufacturing or in the quality system, additional guidance is included in Section 6.3 of this document. For device software, additional useful information may be found in FDA's *Guidance for Industry, FDA Reviewers, and Compliance on Off-The-Shelf Software Use in Medical Devices*.

2.4. QUALITY SYSTEM REGULATION VS PRE-MARKET SUBMISSIONS

This document addresses Quality System regulation issues that involve the implementation of software validation. It provides guidance for the management and control of the software validation process. The management and control of the software validation process should not be confused with any other validation requirements, such as process validation for an automated manufacturing process.

Device manufacturers may use the same procedures and records for compliance with quality system and design control requirements, as well as for pre-market submissions to FDA. This document does not cover any specific safety or efficacy issues related to software validation. Design issues and documentation requirements for pre-market submissions of regulated software are not addressed by this document. Specific issues related to safety and efficacy, and the documentation required in pre-market submissions, should be addressed to the Office of Device Evaluation (ODE), Center for Devices and Radiological Health (CDRH) or to the Office of Blood Research and Review, Center for Biologics Evaluation and Research (CBER). See the references in Appendix A for applicable FDA guidance documents for pre-market submissions.

SECTION 3. CONTEXT FOR SOFTWARE VALIDATION

Many people have asked for specific guidance on what FDA expects them to do to ensure compliance with the Quality System regulation with regard to software validation. Information on software validation presented in this document is not new. Validation of software, using the principles and tasks listed in Sections 4 and 5, has been conducted in many segments of the software industry for well over 20 years.

Due to the great variety of medical devices, processes, and manufacturing facilities, it is not possible to state in one document all of the specific validation elements that are applicable. However, a general application of several broad concepts can be used successfully as guidance for software validation. These broad concepts provide an acceptable framework for building a comprehensive approach to software validation. Additional specific information is available from many of the references listed in Appendix A.

3.1. DEFINITIONS AND TERMINOLOGY

Unless defined in the Quality System regulation, or otherwise specified below, all other terms used in this guidance are as defined in the current edition of the FDA *Glossary of Computerized System and Software Development Terminology*.

The medical device Quality System regulation (21 CFR 820.3(k)) defines "**establish**" to mean "define, document, and implement." Where it appears in this guidance, the words "establish" and "established" should be interpreted to have this same meaning.

Some definitions found in the medical device Quality System regulation can be confusing when compared to commonly used terminology in the software industry. Examples are requirements, specification, verification, and validation.

3.1.1 Requirements and Specifications

While the Quality System regulation states that design input requirements must be documented, and that specified requirements must be verified, the regulation does not further clarify the distinction between the terms "requirement" and "specification." A **requirement** can be any need or expectation for a system or for its software. Requirements reflect the stated or implied needs of the customer, and may be market-based, contractual, or statutory, as well as an organization's internal requirements. There can be many different kinds of requirements (e.g., design, functional, implementation, interface, performance, or physical requirements). Software requirements are typically derived from the system requirements for those aspects of system functionality that have been allocated to software. Software requirements are typically stated in functional terms and are defined, refined, and updated as a development project progresses. Success in accurately and completely documenting software requirements is a crucial factor in successful validation of the resulting software.

A **specification** is defined as "a document that states requirements." (See 21 CFR §820.3(y).) It may refer to or include drawings, patterns, or other relevant documents and usually indicates the means and the criteria whereby conformity with the requirement can be checked. There are many different kinds of written specifications, e.g., system requirements specification, software requirements specification, software design specification, software test specification, software integration specification, etc. All of these documents establish "specified requirements" and are design outputs for which various forms of verification are necessary.

3.1.2 Verification and Validation

The Quality System regulation is harmonized with *ISO 8402*:1994, which treats "verification" and "validation" as separate and distinct terms. On the other hand, many software engineering journal articles and textbooks use the terms "verification" and "validation" interchangeably, or in some cases refer to software "verification, validation, and testing (VV&T)" as if it is a single concept, with no distinction among the three terms.

Software verification provides objective evidence that the design outputs of a particular phase of the software development life cycle meet all of the specified requirements for that phase. Software verification looks for consistency, completeness, and correctness of the software and its supporting documentation, as it is being developed, and provides support for a subsequent conclusion that software is validated. Software testing is one of many verification activities intended to confirm that software development output meets its input requirements. Other verification activities include various static and dynamic analyses, code and document inspections, walkthroughs, and other techniques.

Software validation is a part of the design validation for a finished device, but is not separately defined in the Quality System regulation. For purposes of this guidance, FDA considers software validation to be "**confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled.**" In practice, software validation activities may occur both during, as well as at the end of the software development life cycle to ensure that all requirements have been fulfilled. Since software is usually part of a larger hardware system, the validation of software typically includes evidence that all software requirements have been implemented correctly and completely and are traceable to system requirements. A conclusion that software is validated is highly dependent upon comprehensive software testing, inspections, analyses, and other verification tasks performed at each stage of the software development life cycle. Testing of device software functionality in a simulated use environment, and user site testing are typically included as components of an overall design validation program for a software automated device.

Software verification and validation are difficult because a developer cannot test forever, and it is hard to know how much evidence is enough. In large measure, software validation is a matter of developing a "level of confidence" that the device meets all requirements and user expectations for the software automated functions and features of the device. Measures such as defects found in specifications documents, estimates of defects remaining, testing coverage, and other techniques are all used to

develop an acceptable level of confidence before shipping the product. The level of confidence, and therefore the level of software validation, verification, and testing effort needed, will vary depending upon the safety risk (hazard) posed by the automated functions of the device. Additional guidance regarding safety risk management for software may be found in Section 4 of FDA's *Guidance for the Content of Pre-market Submissions for Software Contained in Medical Devices*, and in the international standards *ISO/IEC 14971-1* and *IEC 60601-1-4* referenced in Appendix A.

3.1.3 IQ/OQ/PQ

For many years, both FDA and regulated industry have attempted to understand and define software validation within the context of process validation terminology. For example, industry documents and other FDA validation guidance sometimes describe user site software validation in terms of installation qualification (IQ), operational qualification (OQ) and performance qualification (PQ). Definitions of these terms and additional information regarding IQ/OQ/PQ may be found in FDA's *Guideline on General Principles of Process Validation*, dated May 11, 1987, and in FDA's *Glossary of Computerized System and Software Development Terminology*, dated August 1995.

While IQ/OQ/PQ terminology has served its purpose well and is one of many legitimate ways to organize software validation tasks at the user site, this terminology may not be well understood among many software professionals, and it is not used elsewhere in this document. However, both FDA personnel and device manufacturers need to be aware of these differences in terminology as they ask for and provide information regarding software validation.

3.2. SOFTWARE DEVELOPMENT AS PART OF SYSTEM DESIGN

The decision to implement system functionality using software is one that is typically made during system design. Software requirements are typically derived from the overall system requirements and design for those aspects in the system that are to be implemented using software. There are user needs and intended uses for a finished device, but users typically do not specify whether those requirements are to be met by hardware, software, or some combination of both. Therefore, software validation must be considered within the context of the overall design validation for the system.

A documented requirements specification represents the user's needs and intended uses from which the product is developed. A primary goal of software validation is to then demonstrate that all completed software products comply with all documented software and system requirements. The correctness and completeness of both the system requirements and the software requirements should be addressed as part of the design validation process for the device. Software validation includes confirmation of conformance to all software specifications and confirmation that all software requirements are traceable to the system specifications. Confirmation is an important part of the overall design validation to ensure that all aspects of the medical device conform to user needs and intended uses.

3.3. SOFTWARE IS DIFFERENT FROM HARDWARE

While software shares many of the same engineering tasks as hardware, it has some very important differences. For example:

- The vast majority of software problems are traceable to errors made during the design and development process. While the quality of a hardware product is highly dependent on design, development and manufacture, the quality of a software product is dependent primarily on design and development with a minimum concern for software manufacture. Software manufacturing consists of reproduction that can be easily verified. It is not difficult to manufacture thousands of program copies that function exactly the same as the original; the difficulty comes in getting the original program to meet all specifications.

- One of the most significant features of software is branching, i.e., the ability to execute alternative series of commands, based on differing inputs. This feature is a major contributing factor for another characteristic of software – its complexity. Even short programs can be very complex and difficult to fully understand.

- Typically, testing alone cannot fully verify that software is complete and correct. In addition to testing, other verification techniques and a structured and documented development process should be combined to ensure a comprehensive validation approach.

- Unlike hardware, software is not a physical entity and does not wear out. In fact, software may improve with age, as latent defects are discovered and removed. However, as software is constantly updated and changed, such improvements are sometimes countered by new defects introduced into the software during the change.

- Unlike some hardware failures, software failures occur without advanced warning. The software's branching that allows it to follow differing paths during execution, may hide some latent defects until long after a software product has been introduced into the marketplace.

- Another related characteristic of software is the speed and ease with which it can be changed. This factor can cause both software and non-software professionals to believe that software problems can be corrected easily. Combined with a lack of understanding of software, it can lead managers to believe that tightly controlled engineering is not needed as much for software as it is for hardware. In fact, the opposite is true. **Because of its complexity, the development process for software should be even more tightly controlled than for hardware, in order to prevent problems that cannot be easily detected later in the development process**.

- Seemingly insignificant changes in software code can create unexpected and very significant problems elsewhere in the software program. The software development process should be sufficiently well planned, controlled, and documented to detect and correct unexpected results from software changes.

- Given the high demand for software professionals and the highly mobile workforce, the software personnel who make maintenance changes to software may not have been involved in the original software development. Therefore, accurate and thorough documentation is essential.

- Historically, software components have not been as frequently standardized and interchangeable as hardware components. However, medical device software developers are beginning to use component-based development tools and techniques. Object-oriented methodologies and the use of off-the-shelf software components hold promise for faster and less expensive software development. However, component-based approaches require very careful attention during integration. Prior to integration, time is needed to fully define and develop reusable software code and to fully understand the behavior of off-the-shelf components.

For these and other reasons, software engineering needs an even greater level of managerial scrutiny and control than does hardware engineering.

3.4. BENEFITS OF SOFTWARE VALIDATION

Software validation is a critical tool used to assure the quality of device software and software automated operations. Software validation can increase the usability and reliability of the device, resulting in decreased failure rates, fewer recalls and corrective actions, less risk to patients and users, and reduced liability to device manufacturers. Software validation can also reduce long term costs by making it easier and less costly to reliably modify software and revalidate software changes. Software maintenance can represent a very large percentage of the total cost of software over its entire life cycle. An established comprehensive software validation process helps to reduce the long-term cost of software by reducing the cost of validation for each subsequent release of the software.

3.5 DESIGN REVIEW

Design reviews are documented, comprehensive, and systematic examinations of a design to evaluate the adequacy of the design requirements, to evaluate the capability of the design to meet these requirements, and to identify problems. While there may be many informal technical reviews that occur within the development team during a software project, a formal design review is more structured and includes participation from others outside the development team. Formal design reviews may reference or include results from other formal and informal reviews. Design reviews may be conducted separately for the software, after the software is integrated with the hardware into the system, or both. Design reviews should include examination of development plans, requirements specifications, design specifications, testing plans and procedures, all other documents and activities associated with the project, verification results from each stage of the defined life cycle, and validation results for the overall device.

Design review is a primary tool for managing and evaluating development projects. For example, formal design reviews allow management to confirm that all goals defined in the software validation plan have

been achieved. The Quality System regulation requires that at least one formal design review be conducted during the device design process. However, it is recommended that multiple design reviews be conducted (e.g., at the end of each software life cycle activity, in preparation for proceeding to the next activity). Formal design review is especially important at or near the end of the requirements activity, before major resources have been committed to specific design solutions. Problems found at this point can be resolved more easily, save time and money, and reduce the likelihood of missing a critical issue.

Answers to some key questions should be documented during formal design reviews. These include:

- Have the appropriate tasks and expected results, outputs, or products been established for each software life cycle activity?

- Do the tasks and expected results, outputs, or products of each software life cycle activity:

  - ✓ Comply with the requirements of other software life cycle activities in terms of correctness, completeness, consistency, and accuracy?

  - ✓ Satisfy the standards, practices, and conventions of that activity?

  - ✓ Establish a proper basis for initiating tasks for the next software life cycle activity?

SECTION 4. PRINCIPLES OF SOFTWARE VALIDATION

This section lists the general principles that should be considered for the validation of software.

4.1. REQUIREMENTS

A documented software requirements specification provides a baseline for both validation and verification. The software validation process cannot be completed without an established software requirements specification (Ref: 21 CFR 820.3(z) and (aa) and 820.30(f) and (g)).

4.2. DEFECT PREVENTION

Software quality assurance needs to focus on preventing the introduction of defects into the software development process and not on trying to "test quality into" the software code after it is written. Software testing is very limited in its ability to surface all latent defects in software code. For example, the complexity of most software prevents it from being exhaustively tested. **Software testing is a necessary activity. However, in most cases software testing by itself is not sufficient to establish confidence that the software is fit for its intended use.** In order to establish that confidence, software developers should use a mixture of methods and techniques to prevent software errors and to detect software errors that do occur. The "best mix" of methods depends on many factors including the development environment, application, size of project, language, and risk.

4.3. TIME AND EFFORT

To build a case that the software is validated requires time and effort. Preparation for software validation should begin early, i.e., during design and development planning and design input. The final conclusion that the software is validated should be based on evidence collected from planned efforts conducted throughout the software lifecycle.

4.4. SOFTWARE LIFE CYCLE

Software validation takes place within the environment of an established software life cycle. The software life cycle contains software engineering tasks and documentation necessary to support the software validation effort. In addition, the software life cycle contains specific verification and validation tasks that are appropriate for the intended use of the software. This guidance does not recommend any particular life cycle models – only that they should be selected and used for a software development project.

4.5. PLANS

The software validation process is defined and controlled through the use of a plan. The software validation plan defines "what" is to be accomplished through the software validation effort. Software validation plans are a significant quality system tool. Software validation plans specify areas such as scope, approach, resources, schedules and the types and extent of activities, tasks, and work items.

4.6. PROCEDURES

The software validation process is executed through the use of procedures. These procedures establish "how" to conduct the software validation effort. The procedures should identify the specific actions or sequence of actions that must be taken to complete individual validation activities, tasks, and work items.

4.7. SOFTWARE VALIDATION AFTER A CHANGE

Due to the complexity of software, a seemingly small local change may have a significant global system impact. When any change (even a small change) is made to the software, the validation status of the software needs to be re-established. **Whenever software is changed, a validation analysis should be conducted not just for validation of the individual change, but also to determine the extent and impact of that change on the entire software system.** Based on this analysis, the software developer should then conduct an appropriate level of software regression testing to show that unchanged but vulnerable portions of the system have not been adversely affected. Design controls and appropriate regression testing provide the confidence that the software is validated after a software change.

4.8. VALIDATION COVERAGE

Validation coverage should be based on the software's complexity and safety risk – not on firm size or resource constraints. The selection of validation activities, tasks, and work items should be commensurate with the complexity of the software design and the risk associated with the use of the software for the specified intended use. For lower risk devices, only baseline validation activities may be conducted. As the risk increases additional validation activities should be added to cover the additional risk. Validation documentation should be sufficient to demonstrate that all software validation plans and procedures have been completed successfully.

4.9. INDEPENDENCE OF REVIEW

Validation activities should be conducted using the basic quality assurance precept of "independence of review." Self-validation is extremely difficult. When possible, an independent evaluation is always better, especially for higher risk applications. Some firms contract out for a third-party independent

verification and validation, but this solution may not always be feasible. Another approach is to assign internal staff members that are not involved in a particular design or its implementation, but who have sufficient knowledge to evaluate the project and conduct the verification and validation activities. Smaller firms may need to be creative in how tasks are organized and assigned in order to maintain internal independence of review.

4.10. FLEXIBILITY AND RESPONSIBILITY

Specific implementation of these software validation principles may be quite different from one application to another. The device manufacturer has flexibility in choosing how to apply these validation principles, but retains ultimate responsibility for demonstrating that the software has been validated.

Software is designed, developed, validated, and regulated in a wide spectrum of environments, and for a wide variety of devices with varying levels of risk. FDA regulated medical device applications include software that:

- Is a component, part, or accessory of a medical device;
- Is itself a medical device; or
- Is used in manufacturing, design and development, or other parts of the quality system.

In each environment, software components from many sources may be used to create the application (e.g., in-house developed software, off-the-shelf software, contract software, shareware). In addition, software components come in many different forms (e.g., application software, operating systems, compilers, debuggers, configuration management tools, and many more). The validation of software in these environments can be a complex undertaking; therefore, it is appropriate that all of these software validation principles be considered when designing the software validation process. The resultant software validation process should be commensurate with the safety risk associated with the system, device, or process.

Software validation activities and tasks may be dispersed, occurring at different locations and being conducted by different organizations. However, regardless of the distribution of tasks, contractual relations, source of components, or the development environment, the device manufacturer or specification developer retains ultimate responsibility for ensuring that the software is validated.

SECTION 5. ACTIVITIES AND TASKS

Software validation is accomplished through a series of activities and tasks that are planned and executed at various stages of the software development life cycle. These tasks may be one time occurrences or may be iterated many times, depending on the life cycle model used and the scope of changes made as the software project progresses.

5.1. SOFTWARE LIFE CYCLE ACTIVITIES

This guidance does not recommend the use of any specific software life cycle model. Software developers should establish a software life cycle model that is appropriate for their product and organization. The software life cycle model that is selected should cover the software from its birth to its retirement. Activities in a typical software life cycle model include the following:

- Quality Planning
- System Requirements Definition
- Detailed Software Requirements Specification
- Software Design Specification
- Construction or Coding
- Testing
- Installation
- Operation and Support
- Maintenance
- Retirement

Verification, testing, and other tasks that support software validation occur during each of these activities. A life cycle model organizes these software development activities in various ways and provides a framework for monitoring and controlling the software development project. Several software life cycle models (e.g., waterfall, spiral, rapid prototyping, incremental development, etc.) are defined in FDA's *Glossary of Computerized System and Software Development Terminology*, dated August 1995. These and many other life cycle models are described in various references listed in Appendix A.

5.2. TYPICAL TASKS SUPPORTING VALIDATION

For each of the software life cycle activities, there are certain "typical" tasks that support a conclusion that the software is validated. However, the specific tasks to be performed, their order of performance, and the iteration and timing of their performance will be dictated by the specific software life cycle model that is selected and the safety risk associated with the software application. For very low risk applications, certain tasks may not be needed at all. However, the software developer should at least consider each of these tasks and should define and document which tasks are or are not appropriate for

their specific application. The following discussion is generic and is not intended to prescribe any particular software life cycle model or any particular order in which tasks are to be performed.

5.2.1. Quality Planning

Design and development planning should culminate in a plan that identifies necessary tasks, procedures for anomaly reporting and resolution, necessary resources, and management review requirements, including formal design reviews. A software life cycle model and associated activities should be identified, as well as those tasks necessary for each software life cycle activity. The plan should include:

- The specific tasks for each life cycle activity;
- Enumeration of important quality factors (e.g., reliability, maintainability, and usability);
- Methods and procedures for each task;
- Task acceptance criteria;
- Criteria for defining and documenting outputs in terms that will allow evaluation of their conformance to input requirements;
- Inputs for each task;
- Outputs from each task;
- Roles, resources, and responsibilities for each task;
- Risks and assumptions; and
- Documentation of user needs.

Management must identify and provide the appropriate software development environment and resources. (See 21 CFR §820.20(b)(1) and (2).) Typically, each task requires personnel as well as physical resources. The plan should identify the personnel, the facility and equipment resources for each task, and the role that risk (hazard) management will play. A configuration management plan should be developed that will guide and control multiple parallel development activities and ensure proper communications and documentation. Controls are necessary to ensure positive and correct correspondence among all approved versions of the specifications documents, source code, object code, and test suites that comprise a software system. The controls also should ensure accurate identification of, and access to, the currently approved versions.

Procedures should be created for reporting and resolving software anomalies found through validation or other activities. Management should identify the reports and specify the contents, format, and responsible organizational elements for each report. Procedures also are necessary for the review and approval of software development results, including the responsible organizational elements for such reviews and approvals.

<u>Typical Tasks – Quality Planning</u>

- Risk (Hazard) Management Plan
- Configuration Management Plan

- Software Quality Assurance Plan
  - Software Verification and Validation Plan
    - Verification and Validation Tasks, and Acceptance Criteria
    - Schedule and Resource Allocation (for software verification and validation activities)
    - Reporting Requirements
  - Formal Design Review Requirements
  - Other Technical Review Requirements
- Problem Reporting and Resolution Procedures
- Other Support Activities

5.2.2. Requirements

Requirements development includes the identification, analysis, and documentation of information about the device and its intended use. Areas of special importance include allocation of system functions to hardware/software, operating conditions, user characteristics, potential hazards, and anticipated tasks. In addition, the requirements should state clearly the intended use of the software.

The software requirements specification document should contain a written definition of the software functions. It is not possible to validate software without predetermined and documented software requirements. Typical software requirements specify the following:

- All software system inputs;
- All software system outputs;
- All functions that the software system will perform;
- All performance requirements that the software will meet, (e.g., data throughput, reliability, and timing);
- The definition of all external and user interfaces, as well as any internal software-to-system interfaces;
- How users will interact with the system;
- What constitutes an error and how errors should be handled;
- Required response times;
- The intended operating environment for the software, if this is a design constraint (e.g., hardware platform, operating system);
- All ranges, limits, defaults, and specific values that the software will accept; and
- All safety related requirements, specifications, features, or functions that will be implemented in software.

Software safety requirements are derived from a technical risk management process that is closely integrated with the system requirements development process. Software requirement specifications should identify clearly the potential hazards that can result from a software failure in the system as well as any safety requirements to be implemented in software. The consequences of software failure should be evaluated, along with means of mitigating such failures (e.g., hardware mitigation, defensive programming, etc.). From this analysis, it should be possible to identify the most appropriate measures necessary to prevent harm.

The Quality System regulation requires a mechanism for addressing incomplete, ambiguous, or conflicting requirements. (See 21 CFR 820.30(c).) Each requirement (e.g., hardware, software, user, operator interface, and safety) identified in the software requirements specification should be evaluated for accuracy, completeness, consistency, testability, correctness, and clarity. For example, software requirements should be evaluated to verify that:

- There are no internal inconsistencies among requirements;
- All of the performance requirements for the system have been spelled out;
- Fault tolerance, safety, and security requirements are complete and correct;
- Allocation of software functions is accurate and complete;
- Software requirements are appropriate for the system hazards; and
- All requirements are expressed in terms that are measurable or objectively verifiable.

A software requirements traceability analysis should be conducted to trace software requirements to (and from) system requirements and to risk analysis results. In addition to any other analyses and documentation used to verify software requirements, a formal design review is recommended to confirm that requirements are fully specified and appropriate before extensive software design efforts begin. Requirements can be approved and released incrementally, but care should be taken that interactions and interfaces among software (and hardware) requirements are properly reviewed, analyzed, and controlled.

Typical Tasks – Requirements

- Preliminary Risk Analysis
- Traceability Analysis
  - Software Requirements to System Requirements (and vice versa)
  - Software Requirements to Risk Analysis
- Description of User Characteristics
- Listing of Characteristics and Limitations of Primary and Secondary Memory
- Software Requirements Evaluation
- Software User Interface Requirements Analysis
- System Test Plan Generation
- Acceptance Test Plan Generation
- Ambiguity Review or Analysis

5.2.3. Design

In the design process, the software requirements specification is translated into a logical and physical representation of the software to be implemented. The software design specification is a description of what the software should do and how it should do it. Due to complexity of the project or to enable

persons with varying levels of technical responsibilities to clearly understand design information, the design specification may contain both a high level summary of the design and detailed design information. The completed software design specification constrains the programmer/coder to stay within the intent of the agreed upon requirements and design. A complete software design specification will relieve the programmer from the need to make ad hoc design decisions.

The software design needs to address human factors. Use error caused by designs that are either overly complex or contrary to users' intuitive expectations for operation is one of the most persistent and critical problems encountered by FDA. Frequently, the design of the software is a factor in such use errors. Human factors engineering should be woven into the entire design and development process, including the device design requirements, analyses, and tests. Device safety and usability issues should be considered when developing flowcharts, state diagrams, prototyping tools, and test plans. Also, task and function analyses, risk analyses, prototype tests and reviews, and full usability tests should be performed. Participants from the user population should be included when applying these methodologies.

The software design specification should include:

- Software requirements specification, including predetermined criteria for acceptance of the software;
- Software risk analysis;
- Development procedures and coding guidelines (or other programming procedures);
- Systems documentation (e.g., a narrative or a context diagram) that describes the systems context in which the program is intended to function, including the relationship of hardware, software, and the physical environment;
- Hardware to be used;
- Parameters to be measured or recorded;
- Logical structure (including control logic) and logical processing steps (e.g., algorithms);
- Data structures and data flow diagrams;
- Definitions of variables (control and data) and description of where they are used;
- Error, alarm, and warning messages;
- Supporting software (e.g., operating systems, drivers, other application software);
- Communication links (links among internal modules of the software, links with the supporting software, links with the hardware, and links with the user);
- Security measures (both physical and logical security); and
- Any additional constraints not identified in the above elements.

The first four of the elements noted above usually are separate pre-existing documents that are included by reference in the software design specification. Software requirements specification was discussed in the preceding section, as was software risk analysis. Written development procedures serve as a guide to the organization, and written programming procedures serve as a guide to individual programmers. As software cannot be validated without knowledge of the context in which it is intended to function, systems documentation is referenced. If some of the above elements are not included in the software, it

may be helpful to future reviewers and maintainers of the software if that is clearly stated (e.g., There are no error messages in this program).

The activities that occur during software design have several purposes. Software design evaluations are conducted to determine if the design is complete, correct, consistent, unambiguous, feasible, and maintainable. Appropriate consideration of software architecture (e.g., modular structure) during design can reduce the magnitude of future validation efforts when software changes are needed. Software design evaluations may include analyses of control flow, data flow, complexity, timing, sizing, memory allocation, criticality analysis, and many other aspects of the design. A traceability analysis should be conducted to verify that the software design implements all of the software requirements. As a technique for identifying where requirements are not sufficient, the traceability analysis should also verify that all aspects of the design are traceable to software requirements. An analysis of communication links should be conducted to evaluate the proposed design with respect to hardware, user, and related software requirements. The software risk analysis should be re-examined to determine whether any additional hazards have been identified and whether any new hazards have been introduced by the design.

At the end of the software design activity, a Formal Design Review should be conducted to verify that the design is correct, consistent, complete, accurate, and testable, before moving to implement the design. Portions of the design can be approved and released incrementally for implementation; but care should be taken that interactions and communication links among various elements are properly reviewed, analyzed, and controlled.

Most software development models will be iterative. This is likely to result in several versions of both the software requirement specification and the software design specification. All approved versions should be archived and controlled in accordance with established configuration management procedures.

Typical Tasks – Design

- Updated Software Risk Analysis
- Traceability Analysis - Design Specification to Software Requirements (and vice versa)
- Software Design Evaluation
- Design Communication Link Analysis
- Module Test Plan Generation
- Integration Test Plan Generation
- Test Design Generation (module, integration, system, and acceptance)

5.2.4. Construction or Coding

Software may be constructed either by coding (i.e., programming) or by assembling together previously coded software components (e.g., from code libraries, off-the-shelf software, etc.) for use in a new application. Coding is the software activity where the detailed design specification is implemented as source code. Coding is the lowest level of abstraction for the software development process. It is the last stage in decomposition of the software requirements where module specifications are translated into a programming language.

Coding usually involves the use of a high-level programming language, but may also entail the use of assembly language (or microcode) for time-critical operations. The source code may be either compiled or interpreted for use on a target hardware platform. Decisions on the selection of programming languages and software build tools (assemblers, linkers, and compilers) should include consideration of the impact on subsequent quality evaluation tasks (e.g., availability of debugging and testing tools for the chosen language). Some compilers offer optional levels and commands for error checking to assist in debugging the code. Different levels of error checking may be used throughout the coding process, and warnings or other messages from the compiler may or may not be recorded. However, at the end of the coding and debugging process, the most rigorous level of error checking is normally used to document what compilation errors still remain in the software. If the most rigorous level of error checking is not used for final translation of the source code, then justification for use of the less rigorous translation error checking should be documented. Also, for the final compilation, there should be documentation of the compilation process and its outcome, including any warnings or other messages from the compiler and their resolution, or justification for the decision to leave issues unresolved.

Firms frequently adopt specific coding guidelines that establish quality policies and procedures related to the software coding process. Source code should be evaluated to verify its compliance with specified coding guidelines. Such guidelines should include coding conventions regarding clarity, style, complexity management, and commenting. Code comments should provide useful and descriptive information for a module, including expected inputs and outputs, variables referenced, expected data types, and operations to be performed. Source code should also be evaluated to verify its compliance with the corresponding detailed design specification. Modules ready for integration and test should have documentation of compliance with coding guidelines and any other applicable quality policies and procedures.

Source code evaluations are often implemented as code inspections and code walkthroughs. Such static analyses provide a very effective means to detect errors before execution of the code. They allow for examination of each error in isolation and can also help in focusing later dynamic testing of the software. Firms may use manual (desk) checking with appropriate controls to ensure consistency and independence. Source code evaluations should be extended to verification of internal linkages between modules and layers (horizontal and vertical interfaces), and compliance with their design specifications. Documentation of the procedures used and the results of source code evaluations should be maintained as part of design verification.

A source code traceability analysis is an important tool to verify that all code is linked to established specifications and established test procedures. A source code traceability analysis should be conducted and documented to verify that:

- Each element of the software design specification has been implemented in code;
- Modules and functions implemented in code can be traced back to an element in the software design specification and to the risk analysis;
- Tests for modules and functions can be traced back to an element in the software design specification and to the risk analysis; and
- Tests for modules and functions can be traced to source code for the same modules and functions.

Typical Tasks – Construction or Coding

- Traceability Analyses
  - Source Code to Design Specification (and vice versa)
  - Test Cases to Source Code and to Design Specification
- Source Code and Source Code Documentation Evaluation
- Source Code Interface Analysis
- Test Procedure and Test Case Generation (module, integration, system, and acceptance)

5.2.5. Testing by the Software Developer

Software testing entails running software products under known conditions with defined inputs and documented outcomes that can be compared to their predefined expectations. It is a time consuming, difficult, and imperfect activity. As such, it requires early planning in order to be effective and efficient.

Test plans and test cases should be created as early in the software development process as feasible. They should identify the schedules, environments, resources (personnel, tools, etc.), methodologies, cases (inputs, procedures, outputs, expected results), documentation, and reporting criteria. The magnitude of effort to be applied throughout the testing process can be linked to complexity, criticality, reliability, and/or safety issues (e.g., requiring functions or modules that produce critical outcomes to be challenged with intensive testing of their fault tolerance features). Descriptions of categories of software and software testing effort appear in the literature, for example:

- NIST Special Publication 500-235, *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*;
- NUREG/CR-6293, *Verification and Validation Guidelines for High Integrity Systems*; and
- IEEE Computer Society Press, *Handbook of Software Reliability Engineering*.

Software test plans should identify the particular tasks to be conducted at each stage of development and include justification of the level of effort represented by their corresponding completion criteria.

Software testing has limitations that must be recognized and considered when planning the testing of a particular software product. Except for the simplest of programs, software cannot be exhaustively tested. Generally it is not feasible to test a software product with all possible inputs, nor is it possible to test all possible data processing paths that can occur during program execution. There is no one type of testing or testing methodology that can ensure a particular software product has been thoroughly tested. Testing of all program functionality does not mean all of the program has been tested. Testing of all of a program's code does not mean all necessary functionality is present in the program. Testing of all program functionality and all program code does not mean the program is 100% correct! Software testing that finds no errors should not be interpreted to mean that errors do not exist in the software product; it may mean the testing was superficial.

An essential element of a software test case is the expected result. It is the key detail that permits objective evaluation of the actual test result. This necessary testing information is obtained from the corresponding, predefined definition or specification. A software specification document must identify what, when, how, why, etc., is to be achieved with an engineering (i.e., measurable or objectively verifiable) level of detail in order for it to be confirmed through testing. The real effort of effective software testing lies in the definition of what is to be tested rather than in the performance of the test.

A software testing process should be based on principles that foster effective examinations of a software product. Applicable software testing tenets include:

- The expected test outcome is predefined;
- A good test case has a high probability of exposing an error;
- A successful test is one that finds an error;
- There is independence from coding;
- Both application (user) and software (programming) expertise are employed;
- Testers use different tools from coders;
- Examining only the usual case is insufficient;
- Test documentation permits its reuse and an independent confirmation of the pass/fail status of a test outcome during subsequent review.

Once the prerequisite tasks (e.g., code inspection) have been successfully completed, software testing begins. It starts with unit level testing and concludes with system level testing. There may be a distinct integration level of testing. A software product should be challenged with test cases based on its internal structure and with test cases based on its external specification. These tests should provide a thorough and rigorous examination of the software product's compliance with its functional, performance, and interface definitions and requirements.

Code-based testing is also known as structural testing or "white-box" testing. It identifies test cases based on knowledge obtained from the source code, detailed design specification, and other development documents. These test cases challenge the control decisions made by the program; and the program's data structures including configuration tables. Structural testing can identify "dead" code

that is never executed when the program is run. Structural testing is accomplished primarily with unit (module) level testing, but can be extended to other levels of software testing.

The level of structural testing can be evaluated using metrics that are designed to show what percentage of the software structure has been evaluated during structural testing. These metrics are typically referred to as "coverage" and are a measure of completeness with respect to test selection criteria. The amount of structural coverage should be commensurate with the level of risk posed by the software. Use of the term "coverage" usually means 100% coverage. For example, if a testing program has achieved "statement coverage," it means that 100% of the statements in the software have been executed at least once. Common structural coverage metrics include:

- **Statement Coverage** – This criteria requires sufficient test cases for each program statement to be executed at least once; however, its achievement is insufficient to provide confidence in a software product's behavior.

- **Decision (Branch) Coverage** – This criteria requires sufficient test cases for each program decision or branch to be executed so that each possible outcome occurs at least once. It is considered to be a minimum level of coverage for most software products, but decision coverage alone is insufficient for high-integrity applications.

- **Condition Coverage** – This criteria requires sufficient test cases for each condition in a program decision to take on all possible outcomes at least once. It differs from branch coverage only when multiple conditions must be evaluated to reach a decision.

- **Multi-Condition Coverage** – This criteria requires sufficient test cases to exercise all possible combinations of conditions in a program decision.

- **Loop Coverage** – This criteria requires sufficient test cases for all program loops to be executed for zero, one, two, and many iterations covering initialization, typical running and termination (boundary) conditions.

- **Path Coverage** – This criteria requires sufficient test cases for each feasible path, basis path, etc., from start to exit of a defined program segment, to be executed at least once. Because of the very large number of possible paths through a software program, path coverage is generally not achievable. The amount of path coverage is normally established based on the risk or criticality of the software under test.

- **Data Flow Coverage** – This criteria requires sufficient test cases for each feasible data flow to be executed at least once. A number of data flow testing strategies are available.

Definition-based or specification-based testing is also known as functional testing or "black-box" testing. It identifies test cases based on the definition of what the software product (whether it be a unit (module) or a complete program) is intended to do. These test cases challenge the intended use or functionality of a program, and the program's internal and external interfaces. Functional testing can be applied at all levels of software testing, from unit to system level testing.

The following types of functional software testing involve generally increasing levels of effort:

- **Normal Case** – Testing with usual inputs is necessary. However, testing a software product only with expected, valid inputs does not thoroughly test that software product. By itself, normal case testing cannot provide sufficient confidence in the dependability of the software product.

- **Output Forcing** – Choosing test inputs to ensure that selected (or all) software outputs are generated by testing.

- **Robustness** – Software testing should demonstrate that a software product behaves correctly when given unexpected, invalid inputs. Methods for identifying a sufficient set of such test cases include Equivalence Class Partitioning, Boundary Value Analysis, and Special Case Identification (Error Guessing). While important and necessary, these techniques do not ensure that all of the most appropriate challenges to a software product have been identified for testing.

- **Combinations of Inputs** – The functional testing methods identified above all emphasize individual or single test inputs. Most software products operate with multiple inputs under their conditions of use. Thorough software product testing should consider the combinations of inputs a software unit or system may encounter during operation. Error guessing can be extended to identify combinations of inputs, but it is an ad hoc technique. Cause-effect graphing is one functional software testing technique that systematically identifies combinations of inputs to a software product for inclusion in test cases.

Functional and structural software test case identification techniques provide specific inputs for testing, rather than random test inputs. One weakness of these techniques is the difficulty in linking structural and functional test completion criteria to a software product's reliability. Advanced software testing methods, such as statistical testing, can be employed to provide further assurance that a software product is dependable. Statistical testing uses randomly generated test data from defined distributions based on an operational profile (e.g., expected use, hazardous use, or malicious use of the software product). Large amounts of test data are generated and can be targeted to cover particular areas or concerns, providing an increased possibility of identifying individual and multiple rare operating conditions that were not anticipated by either the software product's designers or its testers. Statistical testing also provides high structural coverage. It does require a stable software product. Thus, structural and functional testing are prerequisites for statistical testing of a software product.

Another aspect of software testing is the testing of software changes. Changes occur frequently during software development. These changes are the result of 1) debugging that finds an error and it is corrected, 2) new or changed requirements ("requirements creep"), and 3) modified designs as more effective or efficient implementations are found. Once a software product has been baselined (approved), any change to that product should have its own “mini life cycle,” including testing. Testing of a changed software product requires additional effort. Not only should it demonstrate that the change was implemented correctly, testing should also demonstrate that the change did not adversely impact other parts of the software product. Regression analysis and testing are employed to provide

assurance that a change has not created problems elsewhere in the software product. Regression analysis is the determination of the impact of a change based on review of the relevant documentation (e.g., software requirements specification, software design specification, source code, test plans, test cases, test scripts, etc.) in order to identify the necessary regression tests to be run. Regression testing is the rerunning of test cases that a program has previously executed correctly and comparing the current result to the previous result in order to detect unintended effects of a software change. Regression analysis and regression testing should also be employed when using integration methods to build a software product to ensure that newly integrated modules do not adversely impact the operation of previously integrated modules.

In order to provide a thorough and rigorous examination of a software product, development testing is typically organized into levels. As an example, a software product's testing can be organized into unit, integration, and system levels of testing.

1) Unit (module or component) level testing focuses on the early examination of sub-program functionality and ensures that functionality not visible at the system level is examined by testing. Unit testing ensures that quality software units are furnished for integration into the finished software product.

2) Integration level testing focuses on the transfer of data and control across a program's internal and external interfaces. External interfaces are those with other software (including operating system software), system hardware, and the users and can be described as communications links.

3) System level testing demonstrates that all specified functionality exists and that the software product is trustworthy. This testing verifies the as-built program's functionality and performance with respect to the requirements for the software product as exhibited on the specified operating platform(s). System level software testing addresses functional concerns and the following elements of a device's software that are related to the intended use(s):

    - Performance issues (e.g., response times, reliability measurements);
    - Responses to stress conditions, e.g., behavior under maximum load, continuous use;
    - Operation of internal and external security features;
    - Effectiveness of recovery procedures, including disaster recovery;
    - Usability;
    - Compatibility with other software products;
    - Behavior in each of the defined hardware configurations; and
    - Accuracy of documentation.

Control measures (e.g., a traceability analysis) should be used to ensure that the intended coverage is achieved.

System level testing also exhibits the software product's behavior in the intended operating environment. The location of such testing is dependent upon the software developer's ability to produce the target operating environment(s). Depending upon the circumstances, simulation and/or testing at (potential) customer locations may be utilized. Test plans should identify the controls needed to ensure that the

intended coverage is achieved and that proper documentation is prepared when planned system level testing is conducted at sites not directly controlled by the software developer. Also, for a software product that is a medical device or a component of a medical device that is to be used on humans prior to FDA clearance, testing involving human subjects may require an Investigational Device Exemption (IDE) or Institutional Review Board (IRB) approval.

Test procedures, test data, and test results should be documented in a manner permitting objective pass/fail decisions to be reached. They should also be suitable for review and objective decision making subsequent to running the test, and they should be suitable for use in any subsequent regression testing. Errors detected during testing should be logged, classified, reviewed, and resolved prior to release of the software. Software error data that is collected and analyzed during a development life cycle may be used to determine the suitability of the software product for release for commercial distribution. Test reports should comply with the requirements of the corresponding test plans.

Software products that perform useful functions in medical devices or their production are often complex. Software testing tools are frequently used to ensure consistency, thoroughness, and efficiency in the testing of such software products and to fulfill the requirements of the planned testing activities. These tools may include supporting software built in-house to facilitate unit (module) testing and subsequent integration testing (e.g., drivers and stubs) as well as commercial software testing tools. Such tools should have a degree of quality no less than the software product they are used to develop. Appropriate documentation providing evidence of the validation of these software tools for their intended use should be maintained (see section 6 of this guidance).

Typical Tasks – Testing by the Software Developer

- Test Planning
- Structural Test Case Identification
- Functional Test Case Identification
- Traceability Analysis - Testing
  - Unit (Module) Tests to Detailed Design
  - Integration Tests to High Level Design
  - System Tests to Software Requirements
- Unit (Module) Test Execution
- Integration Test Execution
- Functional Test Execution
- System Test Execution
- Acceptance Test Execution
- Test Results Evaluation
- Error Evaluation/Resolution
- Final Test Report

5.2.6. User Site Testing

Testing at the user site is an essential part of software validation. The Quality System regulation requires installation and inspection procedures (including testing where appropriate) as well as documentation of inspection and testing to demonstrate proper installation. (See 21 CFR §820.170.) Likewise, manufacturing equipment must meet specified requirements, and automated systems must be validated for their intended use. (See 21 CFR §820.70(g) and 21 CFR §820.70(i) respectively.)

Terminology regarding user site testing can be confusing. Terms such as beta test, site validation, user acceptance test, installation verification, and installation testing have all been used to describe user site testing. For purposes of this guidance, the term "user site testing" encompasses all of these and any other testing that takes place outside of the developer's controlled environment. This testing should take place at a user's site with the actual hardware and software that will be part of the installed system configuration. The testing is accomplished through either actual or simulated use of the software being tested within the context in which it is intended to function.

Guidance contained here is general in nature and is applicable to any user site testing. However, in some areas (e.g., blood establishment systems) there may be specific site validation issues that need to be considered in the planning of user site testing. Test planners should check with the FDA Center(s) with the corresponding product jurisdiction to determine whether there are any additional regulatory requirements for user site testing.

User site testing should follow a pre-defined written plan with a formal summary of testing and a record of formal acceptance. Documented evidence of all testing procedures, test input data, and test results should be retained.

There should be evidence that hardware and software are installed and configured as specified. Measures should ensure that all system components are exercised during the testing and that the versions of these components are those specified. The testing plan should specify testing throughout the full range of operating conditions and should specify continuation for a sufficient time to allow the system to encounter a wide spectrum of conditions and events in an effort to detect any latent faults that are not apparent during more normal activities.

Some of the evaluations that have been performed earlier by the software developer at the developer's site should be repeated at the site of actual use. These may include tests for a high volume of data, heavy loads or stresses, security, fault testing (avoidance, detection, tolerance, and recovery), error messages, and implementation of safety requirements. The developer may be able to furnish the user with some of the test data sets to be used for this purpose.

In addition to an evaluation of the system's ability to properly perform its intended functions, there should be an evaluation of the ability of the users of the system to understand and correctly interface with it. Operators should be able to perform the intended functions and respond in an appropriate and timely manner to all alarms, warnings, and error messages.

During user site testing, records should be maintained of both proper system performance and any system failures that are encountered. The revision of the system to compensate for faults detected during this user site testing should follow the same procedures and controls as for any other software change.

The developers of the software may or may not be involved in the user site testing. If the developers are involved, they may seamlessly carry over to the user's site the last portions of design-level systems testing. If the developers are not involved, it is all the more important that the user have persons who understand the importance of careful test planning, the definition of expected test results, and the recording of all test outputs.

Typical Tasks – User Site Testing

- Acceptance Test Execution
- Test Results Evaluation
- Error Evaluation/Resolution
- Final Test Report

5.2.7. Maintenance and Software Changes

As applied to software, the term maintenance does not mean the same as when applied to hardware. The operational maintenance of hardware and software are different because their failure/error mechanisms are different. Hardware maintenance typically includes preventive hardware maintenance actions, component replacement, and corrective changes. Software maintenance includes corrective, perfective, and adaptive maintenance but does not include preventive maintenance actions or software component replacement.

Changes made to correct errors and faults in the software are corrective maintenance. Changes made to the software to improve the performance, maintainability, or other attributes of the software system are perfective maintenance. Software changes to make the software system usable in a changed environment are adaptive maintenance.

When changes are made to a software system, either during initial development or during post release maintenance, sufficient regression analysis and testing should be conducted to demonstrate that portions of the software not involved in the change were not adversely impacted. This is in addition to testing that evaluates the correctness of the implemented change(s).

The specific validation effort necessary for each software change is determined by the type of change, the development products affected, and the impact of those products on the operation of the software. Careful and complete documentation of the design structure and interrelationships of various modules, interfaces, etc., can limit the validation effort needed when a change is made. The level of effort needed

to fully validate a change is also dependent upon the degree to which validation of the original software was documented and archived. For example, test documentation, test cases, and results of previous verification and validation testing need to be archived if they are to be available for performing subsequent regression testing. Failure to archive this information for later use can significantly increase the level of effort and expense of revalidating the software after a change is made.

In addition to software verification and validation tasks that are part of the standard software development process, the following additional maintenance tasks should be addressed:

- **Software Validation Plan Revision** - For software that was previously validated, the existing software validation plan should be revised to support the validation of the revised software. If no previous software validation plan exists, such a plan should be established to support the validation of the revised software.

- **Anomaly Evaluation** – Software organizations frequently maintain documentation, such as software problem reports that describe software anomalies discovered and the specific corrective action taken to fix each anomaly. Too often, however, mistakes are repeated because software developers do not take the next step to determine the root causes of problems and make the process and procedural changes needed to avoid recurrence of the problem. Software anomalies should be evaluated in terms of their severity and their effects on system operation and safety, but they should also be treated as symptoms of process deficiencies in the quality system. A root cause analysis of anomalies can identify specific quality system deficiencies. Where trends are identified (e.g., recurrence of similar software anomalies), appropriate corrective and preventive actions must be implemented and documented to avoid further recurrence of similar quality problems. (See 21 CFR 820.100.)

- **Problem Identification and Resolution Tracking** - All problems discovered during maintenance of the software should be documented. The resolution of each problem should be tracked to ensure it is fixed, for historical reference, and for trending.

- **Proposed Change Assessment** - All proposed modifications, enhancements, or additions should be assessed to determine the effect each change would have on the system. This information should determine the extent to which verification and/or validation tasks need to be iterated.

- **Task Iteration** - For approved software changes, all necessary verification and validation tasks should be performed to ensure that planned changes are implemented correctly, all documentation is complete and up to date, and no unacceptable changes have occurred in software performance.

- **Documentation Updating** – Documentation should be carefully reviewed to determine which documents have been impacted by a change. All approved documents (e.g., specifications, test procedures, user manuals, etc.) that have been affected should be updated in accordance with configuration management procedures. Specifications should be updated before any maintenance and software changes are made.

SECTION 6. VALIDATION OF AUTOMATED PROCESS EQUIPMENT AND QUALITY SYSTEM SOFTWARE

The Quality System regulation requires that "when computers or automated data processing systems are used as part of production or the quality system, the [device] manufacturer shall validate computer software for its intended use according to an established protocol." (See 21 CFR §820.70(i)). This has been a regulatory requirement of FDA's medical device Good Manufacturing Practice (GMP) regulations since 1978.

In addition to the above validation requirement, computer systems that implement part of a device manufacturer's production processes or quality system (or that are used to create and maintain records required by any other FDA regulation) are subject to the Electronic Records; Electronic Signatures regulation. (See 21 CFR Part 11.) This regulation establishes additional security, data integrity, and validation requirements when records are created or maintained electronically. These additional Part 11 requirements should be carefully considered and included in system requirements and software requirements for any automated record `keeping systems. System validation and software validation should demonstrate that all Part 11 requirements have been met.

Computers and automated equipment are used extensively throughout all aspects of medical device design, laboratory testing and analysis, product inspection and acceptance, production and process control, environmental controls, packaging, labeling, traceability, document control, complaint management, and many other aspects of the quality system. Increasingly, automated plant floor operations can involve extensive use of embedded systems in:

- programmable logic controllers;
- digital function controllers;
- statistical process control;
- supervisory control and data acquisition;
- robotics;
- human-machine interfaces;
- input/output devices; and
- computer operating systems.

Software tools are frequently used to design, build, and test the software that goes into an automated medical device. Many other commercial software applications, such as word processors, spreadsheets, databases, and flowcharting software are used to implement the quality system. All of these applications are subject to the requirement for software validation, but the validation approach used for each application can vary widely.

Whether production or quality system software is developed in-house by the device manufacturer, developed by a contractor, or purchased off-the-shelf, it should be developed using the basic principles

outlined elsewhere in this guidance. The device manufacturer has latitude and flexibility in defining how validation of that software will be accomplished, but validation should be a key consideration in deciding how and by whom the software will be developed or from whom it will be purchased. The software developer defines a life cycle model. Validation is typically supported by:

- verifications of the outputs from each stage of that software development life cycle; and
- checking for proper operation of the finished software in the device manufacturer's intended use environment.

6.1. HOW MUCH VALIDATION EVIDENCE IS NEEDED?

The level of validation effort should be commensurate with the risk posed by the automated operation. In addition to risk other factors, such as the complexity of the process software and the degree to which the device manufacturer is dependent upon that automated process to produce a safe and effective device, determine the nature and extent of testing needed as part of the validation effort. Documented requirements and risk analysis of the automated process help to define the scope of the evidence needed to show that the software is validated for its intended use. For example, an automated milling machine may require very little testing if the device manufacturer can show that the output of the operation is subsequently fully verified against the specification before release. On the other hand, extensive testing may be needed for:

- a plant-wide electronic record and electronic signature system;
- an automated controller for a sterilization cycle; or
- automated test equipment used for inspection and acceptance of finished circuit boards in a life-sustaining / life-supporting device.

Numerous commercial software applications may be used as part of the quality system (e.g., a spreadsheet or statistical package used for quality system calculations, a graphics package used for trend analysis, or a commercial database used for recording device history records or for complaint management). The extent of validation evidence needed for such software depends on the device manufacturer's documented intended use of that software. For example, a device manufacturer who chooses not to use all the vendor-supplied capabilities of the software only needs to validate those functions that will be used and for which the device manufacturer is dependent upon the software results as part of production or the quality system. However, high risk applications should not be running in the same operating environment with non-validated software functions, even if those software functions are not used. Risk mitigation techniques such as memory partitioning or other approaches to resource protection may need to be considered when high risk applications and lower risk applications are to be used in the same operating environment. When software is upgraded or any changes are made to the software, the device manufacturer should consider how those changes may impact the "used portions" of the software and must reconfirm the validation of those portions of the software that are used. (See 21 CFR §820.70(i).)

6.2. DEFINED USER REQUIREMENTS

A very important key to software validation is a documented user requirements specification that defines:

- the "intended use" of the software or automated equipment; and
- the extent to which the device manufacturer is dependent upon that software or equipment for production of a quality medical device.

The device manufacturer (user) needs to define the expected operating environment including any required hardware and software configurations, software versions, utilities, etc. The user also needs to:

- document requirements for system performance, quality, error handling, startup, shutdown, security, etc.;
- identify any safety related functions or features, such as sensors, alarms, interlocks, logical processing steps, or command sequences; and
- define objective criteria for determining acceptable performance.

The validation must be conducted in accordance with a documented protocol, and the validation results must also be documented. (See 21 CFR §820.70(i).) Test cases should be documented that will exercise the system to challenge its performance against the pre-determined criteria, especially for its most critical parameters. Test cases should address error and alarm conditions, startup, shutdown, all applicable user functions and operator controls, potential operator errors, maximum and minimum ranges of allowed values, and stress conditions applicable to the intended use of the equipment. The test cases should be executed and the results should be recorded and evaluated to determine whether the results support a conclusion that the software is validated for its intended use.

A device manufacturer may conduct a validation using their own personnel or may depend on a third party such as the equipment/software vendor or a consultant. In any case, the device manufacturer retains the ultimate responsibility for ensuring that the production and quality system software:

- is validated according to a written procedure for the particular intended use; and
- will perform as intended in the chosen application.

The device manufacturer should have documentation including:

- defined user requirements;
- validation protocol used;
- acceptance criteria;
- test cases and results; and
- a validation summary

that objectively confirms that the software is validated for its intended use.

6.3. VALIDATION OF OFF-THE-SHELF SOFTWARE AND AUTOMATED EQUIPMENT

Most of the automated equipment and systems used by device manufacturers are supplied by third-party vendors and are purchased off-the-shelf (OTS). The device manufacturer is responsible for ensuring that the product development methodologies used by the OTS software developer are appropriate and sufficient for the device manufacturer's intended use of that OTS software. For OTS software and equipment, the device manufacturer may or may not have access to the vendor's software validation documentation. If the vendor can provide information about their system requirements, software requirements, validation process, and the results of their validation, the medical device manufacturer can use that information as a beginning point for their required validation documentation. The vendor's life cycle documentation, such as testing protocols and results, source code, design specification, and requirements specification, can be useful in establishing that the software has been validated. However, such documentation is frequently not available from commercial equipment vendors, or the vendor may refuse to share their proprietary information.

Where possible and depending upon the device risk involved, the device manufacturer should consider auditing the vendor's design and development methodologies used in the construction of the OTS software and should assess the development and validation documentation generated for the OTS software. Such audits can be conducted by the device manufacturer or by a qualified third party. The audit should demonstrate that the vendor's procedures for and results of the verification and validation activities performed the OTS software are appropriate and sufficient for the safety and effectiveness requirements of the medical device to be produced using that software.

Some vendors who are not accustomed to operating in a regulated environment may not have a documented life cycle process that can support the device manufacturer's validation requirement. Other vendors may not permit an audit. Where necessary validation information is not available from the vendor, the device manufacturer will need to perform sufficient system level "black box" testing to establish that the software meets their "user needs and intended uses." For many applications black box testing alone is not sufficient. Depending upon the risk of the device produced, the role of the OTS software in the process, the ability to audit the vendor, and the sufficiency of vendor-supplied information, the use of OTS software or equipment may or may not be appropriate, especially if there are suitable alternatives available. The device manufacturer should also consider the implications (if any) for continued maintenance and support of the OTS software should the vendor terminate their support.

For some off-the-shelf software development tools, such as software compilers, linkers, editors, and operating systems, exhaustive black-box testing by the device manufacturer may be impractical. Without such testing – a key element of the validation effort – it may not be possible to validate these software tools. However, their proper operation may be satisfactorily inferred by other means. For example, compilers are frequently certified by independent third-party testing, and commercial software products may have "bug lists", system requirements and other operational information available from the vendor that can be compared to the device manufacturer's intended use to help focus the "black-box" testing effort. Off-the-shelf operating systems need not be validated as a separate program. However, system-level validation testing of the application software should address all the operating system services used, including maximum loading conditions, file operations, handling of system error

conditions, and memory constraints that may be applicable to the intended use of the application program.

For more detailed information, see the production and process software references in Appendix A.

APPENDIX A - REFERENCES

Food and Drug Administration References

Design Control Guidance for Medical Device Manufacturers, Center for Devices and Radiological Health, Food and Drug Administration, March 1997.

Do It by Design, An Introduction to Human Factors in Medical Devices, Center for Devices and Radiological Health, Food and Drug Administration, March 1997.

Electronic Records; Electronic Signatures Final Rule, 62 Federal Register 13430 (March 20, 1997).

Glossary of Computerized System and Software Development Terminology, Division of Field Investigations, Office of Regional Operations, Office of Regulatory Affairs, Food and Drug Administration, August 1995.

Guidance for the Content of Pre-market Submissions for Software Contained in Medical Devices, Office of Device Evaluation, Center for Devices and Radiological Health, Food and Drug Administration, May 1998.

Guidance for Industry, FDA Reviewers and Compliance on Off-the-Shelf Software Use in Medical Devices, Office of Device Evaluation, Center for Devices and Radiological Health, Food and Drug Administration, September 1999.

Guideline on General Principles of Process Validation, Center for Drugs and Biologics, & Center For Devices and Radiological Health, Food and Drug Administration, May 1987.

Medical Devices; Current Good Manufacturing Practice (CGMP) Final Rule; Quality System Regulation , 61 Federal Register 52602 (October 7, 1996).

Reviewer Guidance for a Pre-Market Notification Submission for Blood Establishment Computer Software, Center for Biologics Evaluation and Research, Food and Drug Administration, January 1997

Student Manual 1, Course INV545, Computer System Validation, Division of Human Resource Development, Office of Regulatory Affairs, Food and Drug Administration, 1997.

Technical Report, Software Development Activities, Division of Field Investigations, Office of Regional Operations, Office of Regulatory Affairs, Food and Drug Administration, July 1987.

Other Government References

W. Richards Adrion, Martha A. Branstad, John C. Cherniavsky. *NBS Special Publication 500-75, Validation, Verification, and Testing of Computer Software,* Center for Programming Science and Technology, Institute for Computer Sciences and Technology, National Bureau of Standards, U.S. Department of Commerce, February 1981.

Martha A. Branstad, John C Cherniavsky, W. Richards Adrion, *NBS Special Publication 500-56, Validation, Verification, and Testing for the Individual Programmer*, Center for Programming Science and Technology, Institute for Computer Sciences and Technology, National Bureau of Standards, U.S. Department of Commerce, February 1980.

J.L. Bryant, N.P. Wilburn, *Handbook of Software Quality Assurance Techniques Applicable to the Nuclear Industry*, NUREG/CR-4640, U.S. Nuclear Regulatory Commission, 1987.

H. Hecht, et.al., *Verification and Validation Guidelines for High Integrity Systems.* NUREG/CR-6293. Prepared for U.S. Nuclear Regulatory Commission, 1995.

H. Hecht, et.al., *Review Guidelines on Software Languages for Use in Nuclear Power Plant Safety Systems, Final Report*. NUREG/CR-6463. Prepared for U.S. Nuclear Regulatory Commission, 1996.

J.D. Lawrence, W.L. Persons, *Survey of Industry Methods for Producing Highly Reliable Software*, NUREG/CR-6278, U.S. Nuclear Regulatory Commission, 1994.

J.D. Lawrence, G.G. Preckshot, *Design Factors for Safety-Critical Software,* NUREG/CR-6294, U.S. Nuclear Regulatory Commission, 1994.

Patricia B. Powell, Editor. *NBS Special Publication 500-98, Planning for Software Validation, Verification, and Testing,* Center for Programming Science and Technology, Institute for Computer Sciences and Technology, National Bureau of Standards, U.S. Department of Commerce, November 1982.

Patricia B. Powell, Editor. *NBS Special Publication 500-93, Software Validation, Verification, and Testing Technique and Tool Reference Guide*, Center for Programming Science and Technology, Institute for Computer Sciences and Technology, National Bureau of Standards, U.S. Department of Commerce, September 1982.

Delores R. Wallace, Roger U. Fujii, *NIST Special Publication 500-165, Software Verification and Validation: Its Role in Computer Assurance and Its Relationship with Software Project Management Standards*, National Computer Systems Laboratory, National Institute of Standards and Technology, U.S. Department of Commerce, September 1995.

Delores R. Wallace, Laura M. Ippolito, D. Richard Kuhn, *NIST Special Publication 500-204, High Integrity Software, Standards and Guidelines*, Computer Systems Laboratory, National Institute of

Standards and Technology, U.S. Department of Commerce, September 1992.

Delores R. Wallace, et.al. *NIST Special Publication 500-234, Reference Information for the Software Verification and Validation Process*. Computer Systems Laboratory, National Institute of Standards and Technology, U.S. Department of Commerce, March 1996.

Delores R. Wallace, Editor. *NIST Special Publication 500-235, Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. Computer Systems Laboratory, National Institute of Standards and Technology, U.S. Department of Commerce, August 1996.

International and National Consensus Standards

ANSI / ANS-10.4-1987, *Guidelines for the Verification and Validation of Scientific and Engineering Computer Programs for the Nuclear Industry*, American National Standards Institute, 1987.

ANSI / ASQC Standard D1160-1995, *Formal Design Reviews,* American Society for Quality Control, 1995.

ANSI / UL 1998:1998, *Standard for Safety for Software in Programmable Components*, Underwriters Laboratories, Inc., 1998.

AS 3563.1-1991, *Software Quality Management System, Part 1: Requirements.* Published by Standards Australia [Standards Association of Australia], 1 The Crescent, Homebush, NSW 2140.

AS 3563.2-1991, *Software Quality Management System, Part 2: Implementation Guide*. Published by Standards Australia [Standards Association of Australia], 1 The Crescent, Homebush, NSW 2140.

IEC 60601-1-4:1996, *Medical electrical equipment, Part 1: General requirements for safety, 4. Collateral Standard: Programmable electrical medical systems.* International Electrotechnical Commission, 1996.

IEC 61506:1997, *Industrial process measurement and control – Documentation of application software*. International Electrotechnical Commission, 1997.

IEC 61508:1998, *Functional safety of electrical/electronic/programmable electronic safety-related systems*. International Electrotechnical Commission, 1998.

IEEE Std 1012-1986, *Software Verification and Validation Plans*, Institute for Electrical and Electronics Engineers, 1986.

IEEE Standards Collection, Software Engineering, Institute of Electrical and Electronics Engineers, Inc., 1994. ISBN 1-55937-442-X.

ISO 8402:1994, *Quality management and quality assurance – Vocabulary*. International Organization for Standardization, 1994.

ISO 9000-3:1997, *Quality management and quality assurance standards - Part 3: Guidelines for the application of ISO 9001:1994 to the development, supply, installation and maintenance of computer software*. International Organization for Standardization, 1997.

ISO 9001:1994, *Quality systems – Model for quality assurance in design, development, production, installation, and servicing*. International Organization for Standardization, 1994.

ISO 13485:1996, *Quality systems – Medical devices – Particular requirements for the application of ISO 9001*. International Organization for Standardization, 1996.

ISO/IEC 12119:1994, *Information technology – Software packages – Quality requirements and testing*, Joint Technical Committee ISO/IEC JTC 1, International Organization for Standardization and International Electrotechnical Commission, 1994.

ISO/IEC 12207:1995, *Information technology – Software life cycle processes*, Joint Technical Committee ISO/IEC JTC 1, Subcommittee SC 7, International Organization for Standardization and International Electrotechnical Commission, 1995.

ISO/IEC 14598:1999, *Information technology – Software product evaluation*, Joint Technical Committee ISO/IEC JTC 1, Subcommittee SC 7, International Organization for Standardization and International Electrotechnical Commission, 1999.

ISO 14971-1:1998, *Medical Devices – Risk Management – Part 1: Application of Risk Analysis.* International Organization for Standardization, 1998.

Software Considerations in Airborne Systems and Equipment Certification. Special Committee 167 of RTCA. RTCA Inc., Washington, D.C. Tel: 202-833-9339. Document No. RTCA/DO-178B, December 1992.

Production Process Software References

The Application of the Principles of GLP to Computerized Systems, Environmental Monograph #116, Organization for Economic Cooperation and Development (OECD), 1995.

George J. Grigonis, Jr., Edward J. Subak, Jr., and Michael Wyrick, "Validation Key Practices for Computer Systems Used in Regulated Operations," *Pharmaceutical Technology*, June 1997.

Guide to Inspection of Computerized Systems in Drug Processing, Reference Materials and

Training Aids for Investigators, Division of Drug Quality Compliance, Associate Director for Compliance, Office of Drugs, National Center for Drugs and Biologics, & Division of Field Investigations, Associate Director for Field Support, Executive Director of Regional Operations, Food and Drug Administration, February 1983.

Daniel P. Olivier, "Validating Process Software", *FDA Investigator Course: Medical Device Process Validation*, Food and Drug Administration.

GAMP Guide For Validation of Automated Systems in Pharmaceutical Manufacture,Version V3.0, Good Automated Manufacturing Practice (GAMP) Forum, March 1998:

Volume 1, Part 1: User Guide
Part 2: Supplier Guide
Volume 2: Best Practice for User and Suppliers.

Technical Report No. 18, Validation of Computer-Related Systems. PDA Committee on Validation of Computer-Related Systems. PDA Journal of Pharmaceutical Science and Technology, Volume 49, Number 1, January-February 1995 Supplement.

Validation Compliance Annual 1995, International Validation Forum, Inc.

General Software Quality References

Boris Beizer, *Black Box Testing, Techniques for Functional Testing of Software and Systems*, John Wiley & Sons, 1995. ISBN 0-471-12094-4.

Boris Beizer, *Software System Testing and Quality Assurance*, International Thomson Computer Press, 1996. ISBN 1-85032-821-8.

Boris Beizer, *Software Testing Techniques*, Second Edition, Van Nostrand Reinhold, 1990. ISBN 0-442-20672-0.

Richard Bender, *Writing Testable Requirements, Version 1.0*, Bender & Associates, Inc., Larkspur, CA 94777, 1996.

Frederick P. Brooks, Jr., *The Mythical Man-Month, Essays on Software Engineering*, Addison-Wesley Longman, Anniversary Edition, 1995. ISBN 0-201-83595-9.

Silvana Castano, et.al., *Database Security*, ACM Press, Addison-Wesley Publishing Company, 1995. ISBN 0-201-59375-0.

Computerized Data Systems for Nonclinical Safety Assessment, Current Concepts and Quality Assurance, Drug Information Association, Maple Glen, PA, September 1988.

M. S. Deutsch, *Software Verification and Validation, Realistic Project Approaches*, Prentice Hall, 1982.

Robert H. Dunn and Richard S. Ullman, *TQM for Computer Software*, Second Edition, McGraw-Hill, Inc., 1994. ISBN 0-07-018314-7.

Elfriede Dustin, Jeff Rashka, and John Paul, *Automated Software Testing – Introduction, Management and Performance,* Addison Wesley Longman, Inc., 1999. ISBN 0-201-43287-0.

Robert G. Ebenau and Susan H. Strauss, *Software Inspection Process*, McGraw-Hill, 1994. ISBN 0-07-062166-7.

Richard E. Fairley, *Software Engineering Concepts*, McGraw-Hill Publishing Company, 1985. ISBN 0-07-019902-7.

Michael A. Friedman and Jeffrey M. Voas, *Software Assessment - Reliability, Safety, Testability,* Wiley-Interscience, John Wiley & Sons Inc., 1995. ISBN 0-471-01009-X.

Tom Gilb, Dorothy Graham, *Software Inspection*, Addison-Wesley Publishing Company, 1993. ISBN 0-201-63181-4.

Robert B. Grady, *Practical Software Metrics for Project Management and Process Improvement*, PTR Prentice-Hall Inc., 1992. ISBN 0-13-720384-5.

Les Hatton, *Safer C: Developing Software for High-integrity and Safety-critical Systems*, McGraw-Hill Book Company, 1994. ISBN 0-07-707640-0.

Janis V. Halvorsen, *A Software Requirements Specification Document Model for the Medical Device Industry*, Proceedings IEEE SOUTHEASTCON '93, Banking on Technology, April 4th -7th, 1993, Charlotte, North Carolina.

Debra S. Herrmann, *Software Safety and Reliability: Techniques, Approaches and Standards of Key Industrial Sectors*, IEEE Computer Society, 1999. ISBN 0-7695-0299-7.

Bill Hetzel, *The Complete Guide to Software Testing*, Second Edition, A Wiley-QED Publication, John Wiley & Sons, Inc., 1988. ISBN 0-471-56567-9.

Watts S. Humphrey, *A Discipline for Software Engineering*. Addison-Wesley Longman, 1995. ISBN 0-201-54610-8.

Watts S. Humphrey, *Managing the Software Process*, Addison-Wesley Publishing Company, 1989. ISBN 0-201-18095-2.

Capers Jones, *Software Quality, Analysis and Guidelines for Success*, International Thomson Computer Press, 1997. ISBN 1-85032-867-6.

J.M. Juran, Frank M. Gryna, *Quality Planning and Analysis*, Third Edition, , McGraw-Hill, 1993. ISBN 0-07-033183-9.

Stephen H. Kan, *Metrics and Models in Software Quality Engineering*, Addison-Wesley Publishing Company, 1995. ISBN 0-201-63339-6.

Cem Kaner, Jack Falk, Hung Quoc Nguyen, *Testing Computer Software*, Second Edition, Vsn Nostrand Reinhold, 1993. ISBN 0-442-01361-2.

Craig Kaplan, Ralph Clark, Victor Tang, *Secrets of Software Quality, 40 Innovations from IBM*, McGraw-Hill, 1995. ISBN 0-07-911795-3.

Edward Kit, *Software Testing in the Real World*, Addison-Wesley Longman, 1995. ISBN 0-201-87756-2.

Alan Kusinitz, "Software Validation", *Current Issues in Medical Device Quality Systems*, Association for the Advancement of Medical Instrumentation, 1997. ISBN 1-57020-075-0.

Nancy G. Leveson, *Safeware, System Safety and Computers*, Addison-Wesley Publishing Company, 1995. ISBN 0-201-11972-2.

Michael R. Lyu, Editor, *Handbook of Software Reliability Engineering*, IEEE Computer Society Press, McGraw-Hill, 1996. ISBN 0-07-039400-8.

Steven R. Mallory, *Software Development and Quality Assurance for the Healthcare Manufacturing Industries*, Interpharm Press,Inc., 1994. ISBN 0-935184-58-9.

Brian Marick, *The Craft of Software Testing*, Prentice Hall PTR, 1995. ISBN 0-13-177411-5.

Steve McConnell, *Rapid Development*, Microsoft Press, 1996. ISBN 1-55615-900-5.

Glenford J. Myers, *The Art of Software Testing*, John Wiley & Sons, 1979. ISBN 0-471-04328-1.

Peter G. Neumann, *Computer Related Risks*, ACM Press/Addison-Wesley Publishing Co., 1995. ISBN 0-201-55805-X.

Daniel Olivier, *Conducting Software Audits, Auditing Software for Conformance to FDA Requirements*, Computer Application Specialists, San Diego, CA, 1994.

William Perry, *Effective Methods for Software Testing*, John Wiley & Sons, Inc. 1995. ISBN 0-471-06097-6.

William E. Perry, Randall W. Rice, *Surviving the Top Ten Challenges of Software Testing*, Dorset

House Publishing, 1997. ISBN 0-932633-38-2.

Roger S. Pressman, *Software Engineering, A Practitioner's Approach*, Third Edition, McGraw-Hill Inc., 1992. ISBN 0-07-050814-3.

Roger S. Pressman, *A Manager's Guide to Software Engineering*, McGraw-Hill Inc., 1993 ISBN 0-07-050820-8.

A. P. Sage, J. D. Palmer, *Software Systems Engineering*, John Wiley & Sons, 1990.

Joc Sanders, Eugene Curran, *Software Quality,* Addison-Wesley Publishing Co., 1994. ISBN 0-201-63198-9.

Ken Shumate, Marilyn Keller, *Software Specification and Design, A Disciplined Approach for Real-Time Systems*, John Wiley & Sons, 1992. ISBN 0-471-53296-7.

Dennis D. Smith, *Designing Maintainable Software*, Springer-Verlag, 1999. ISBN 0-387-98783-5.

Ian Sommerville, *Software Engineering*, Third Edition, Addison Wesley Publishing Co., 1989. ISBN 0-201-17568-1.

Karl E. Wiegers, *Creating a Software Engineering Culture*, Dorset House Publishing, 1996. ISBN 0-932633-33-1.

Karl E. Wiegers, *Software Inspection, Improving Quality with Software Inspections*, Software Development, April 1995, pages 55-64.

Karl E. Wiegers, *Software Requirements*, Microsoft Press, 1999. ISBN 0-7356-0631-5.

APPENDIX B - DEVELOPMENT TEAM

Center for Devices and Radiological Health

Office of Compliance	Stewart Crumpler
Office of Device Evaluation	James Cheng, Donna-Bea Tillman
Office of Health and Industry Programs	Bryan Benesch, Dick Sawyer
Office of Science and Technology	John Murray
Office of Surveillance and Biometrics	Howard Press

Center for Drug Evaluation and Research

Office of Medical Policy	Charles Snipes

Center for Biologics Evaluation and Research

Office of Compliance and Biologics Quality	Alice Godziemski

Office of Regulatory Affairs

Office of Regional Operations	David Bergeson, Joan Loreng