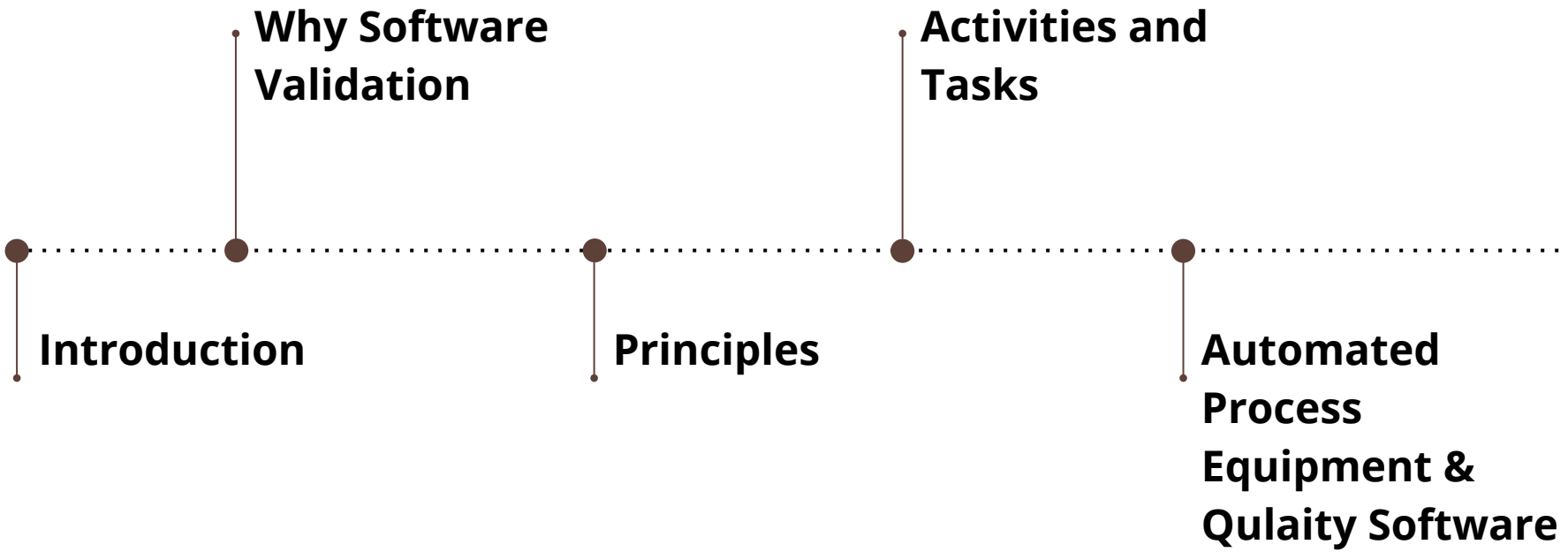
Medical Software Testing FDA

Abhishek Appaji

Topics


Introduction
Why Software Validation
Principles
Activities and Tasks
Automated Process Equipment & Qulaity Software

Usage

- 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).

Regulatory requirements for Software Validation

- 3140 medical device recalls conducted between 1992 and 1998
- 242 (7.7%) are software failures.
- 192 (79%) were caused by software defects after its initial production and distribution.
- Device production process or quality system should be validated
- Component acceptance, manufacturing, labeling, packaging, distribution, complaint handling, or to automate any other aspect of the quality system.
- create, modify, and maintain electronic records and to manage electronic signatures
- validated to ensure accuracy, reliability, consistent intended performance, and the ability to discern invalid or altered records.

”off-the-shelf” software

Even this has to be validated

Why software validation

Establish - Define, document and implement

Requirement - Need or expectation for a system or software

e.g., design, functional, implementation, interface, performance, or physical requirements

Specification - Document that states requirement

include drawings, patterns, or other relevant documents

System requirements specification, software requirements specification, software design specification, software test specification, software integration specification, etc

Verification & Validation

Verification

objective evidence that the design outputs of a particular phase of the software development life cycle meet all of the specified requirements

consistency, completeness, and correctness of the software and its supporting documentation,

Ex: static and dynamic analyses, code and document inspections, walkthroughs, etc

Validation

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.

During and at the end of the software development life cycle

comprehensive software testing, inspections, analyses, and other verification tasks performed at each stage

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 Testing - level of confidence

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

will vary depending upon the safety risk (hazard) posed by the automated functions of the device

Installation / Operational / Performance Qualification

IQ

OQ

PQ

Software development as part of system design

software validation must be considered within the context of the overall design validation for the system

demonstrate that completed software products comply with documented software and system requirements

confirmation of conformance to all software specifications

Confirmation of software requirements are traceable to the system specifications.

Software vs Hardware

Software doesn’t wear out unlike hardware

May improve over age

But continuous update may counter new problem

Software failure can occur without warning unlike hardware

Changes in software is easy and faster

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.

Software vs Hardware

Little changes can bring unexpected and significant problems elsewhere

Documentation is important to debug by someone else in future

Software changes are not standardized or interchangeable

In summary,

software engineering needs an even greater level of managerial scrutiny and control than does hardware engineering.

Software is different from Hardware

Software

Errors easily traceable during design and dev

Quality depends on design and Dev

Reproduction is easy (copy/paste)

Meeting specification is tough

Branching is easy (parallel execution) but complex

Testing alone cannot verify software to be complete or correct

Hardware

Tough to debug

Quality depends on design, dev & manufacturing

Reproduction is tough

Meeting specs is comparatively easy

Benefits of software validation

assure the quality of device software and software automated operations

increase the usability and reliability of the device,

resulting in decreased failure rates,

fewer recalls and corrective actions,

less risk to patients and users,

reduced liability to device manufacturers

reduce long term costs

Design Review

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

separately for the software, after the software is integrated with the hardware into the system, or both.

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

Design Review ...

primary tool for managing and evaluating development projects

it is recommended that multiple design reviews be conducted

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

Design Review... should answer

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?

Principles of Software Validation

Defect Prevention

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.

development environment, application, size of project, language, and risk.

begin early, i.e., during design and development planning and design input.

Software Life Cycle

the software life cycle contains specific verification and validation tasks that are appropriate for the intended use of the software.

plan defines “what” is to be accomplished through the software validation effort.

Software validation **plans** specify areas such as scope, approach, resources, schedules and the types and extent of activities, tasks, and work items.

Software Change

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.

Validation Coverage

Based on software’s complexity and safety risk – not on firm size or resource constraints.

Depends complexity of the software design and the risk associated

INDEPENDENCE OF REVIEW

Self-validation is extremely difficult.

an independent evaluation is always better, especially for higher risk applications.

Ex: third-party or internal staff members that are not involved in a particular design or its implementation

FLEXIBILITY AND RESPONSIBILITY

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.

FLEXIBILITY AND RESPONSIBILITY

application (e.g., in-house developed software, off-the-shelf software, contract software, shareware)

different forms (e.g., application software, operating systems, compilers, debuggers, configuration management tools)

commensurate with the safety risk associated with the system, device, or process.

Activities and Tasks

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

Quality Planning

· 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.

Typical Tasks – Quality Planning ·

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

Typical software Requirements

- 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.

Should be verified for

- 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.

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

Design

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.

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)

Construction or Coding

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)

Testing by the Software Developer

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.

Coverage Metrics

- **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.

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.

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.

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

User Site Testing

Terms such as beta test, site validation, user acceptance test, installation verification, and installation testing have all been used to describe user site testing.

Typical Tasks – User Site Testing

· Acceptance Test Execution

· Test Results Evaluation

· Error Evaluation/Resolution

· Final Test Report

Maintenance and Software Changes

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.

HOW MUCH VALIDATION EVIDENCE IS NEEDED?

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 lifesustaining / life-supporting device.

DEFINED USER REQUIREMENTS

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.

· 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 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.

Please note the following points

I have enclosed the questions.

- Students are expected to research to find answers to the questions
- They have to use the handout as one of the references
- Strictly no copy paste from Google. Use the references but explain in their own words (No plagiarism)
- They are expected to use examples in all the answers
- The evaluation may involve one on one Q&A session

Questions

1. Why testing is important?
2. HW vs SW vs Embedded testing – Highlight the differences
3. Verification vs Validation vs Testing – Highlight the differences
4. Write Short Notes: Defect Prevention in the context SW development and testing
5. White Box vs Black Box Testing – Highlight the differences
6. Refer to the Handout on Coverage Metrics. Explain the concepts with **examples**
7. In the above Coverage Metrics what are we achieving?
8. Explore Google to find out additional Coverage Metrics. Explain them in detail
9. Functional Testing: Explain the concept with examples
10. Explore Google to find out additional Functional Testing strategies. Explain them in detail
11. There are few techniques mentioned under robustness. Explain them with examples
12. Explore additional techniques for robustness testing. Explain them with examples
13. Write short Notes on System Level testing
14. Write short Notes on Site Level Testing
15. Explore Automated testing techniques. Explain them in detail
16. Study about bug tracking tools and explain one of them in detail. Focus on the purpose of the tool.
17. Choose an application and do the functional testing using the techniques that you have learnt. Document them
18. Use any common app you find and design test cases for the app.