Software validation
- a tool for better software quality

1. PREFACE

In the years of 1992 to 1998 the US Food and Drug Administration, FDA analyzed more than 3100 medical device recalls. This investigation revealed that 242 of them (7.7%) were 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.

The nature of software makes it quite different from hardware especially because software problems are mostly 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.

Software validation and other related good software engineering practices are principal means of avoiding such defects and resultant recalls by focusing on the development life-cycle of the software.

2. WHAT IS SOFTWARE VALIDATION?

Software validation is a part of the overall design validation for a finished medical device.
The 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 occurs both at the beginning, during and at the end of the software development life cycle to ensure that all requirements have been fulfilled.

The validation of software typically includes evidence that all software requirements have been implemented correctly and completely; that all software requirements is traceable to system requirements and that a comprehensive software testing, inspections, analyses, and other verification tasks has been 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 medical 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.

3. REGULATORY REQUIREMENTS

The FDA's Quality System Regulation requires compliance with the validation requirements for software as expressed in 21 CFR (Code of Federal Regulations):

- 820.30(g) Design validation
- 820.70(i) Automated processes
- 820.75 Process validation and
- 21 CFR 11 Electronic records and signatures

In contrast the EU / Medical Devices Directive is less specific and requires only compliance with "electronic programmable systems" in Annex I, section 12.1 with respect to medical devices:

- *Devices incorporating electronic programmable systems must be designed to ensure the repeatability, reliability and performance of these systems according to the intended use. In the event of a single fault condition (in the system) appropriate means should be adopted to eliminate or reduce as far as possible consequent risks*

The interpretation of this requirement was established when the EU-Commission harmonized the standard: EN 60601-1-4 "General requirements for Safety, Programmable Electrical Medical Systems" in November 2001 to the MDD.

Automated processes and software process validation is not regulated through the Medical Devices Directives. However, the new draft international standard ISO/DIS 13485 does specify in section 6.1 "Infrastructure", in 7.5.2.2 "Record keeping" and in 7.6 "Control of monitoring and measuring devices" requirements for software validation.

4. SOFTWARE LIFE CYCLE

Software validation takes place within the environment of an established software life cycle. The software life cycle is divided into phases which contain software engineering tasks and documentation necessary to support the software validation effort. Especially, the software life cycle contains specific verification and validation tasks based on well defined input and output specification with integral processes for Risk Management. Typical activities would include:

Process implementation

- Definition of activities, software units and subunits, tasks, deliverables and risk management plan

Establishing of requirements

- Definition of software requirements including architecture, input and output specification

Software design

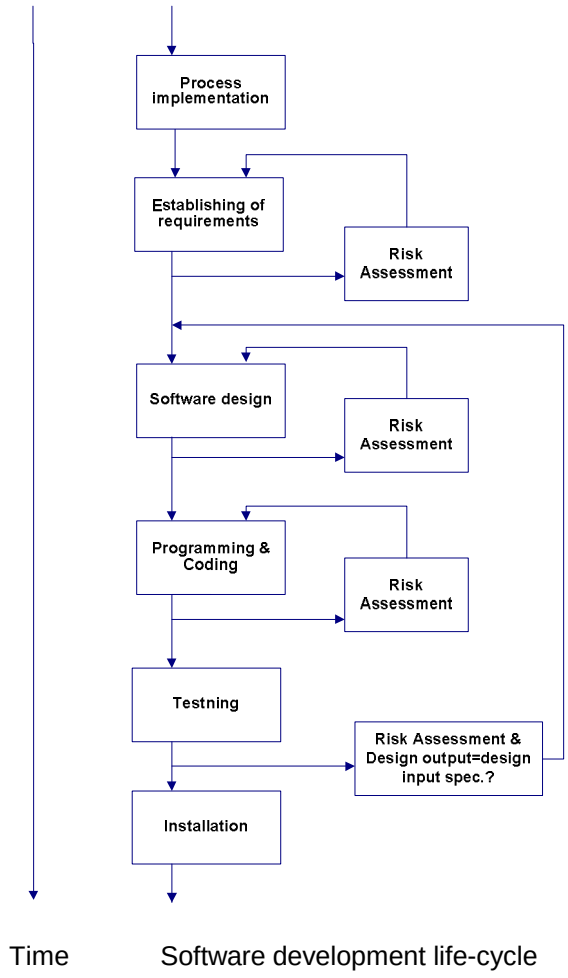
- Software detailed design; incl. in –and output description of software units, verification & validation methods and interface to hardware….

Software coding

- Establishing of strategy and methods for verifying each software unit

Software testing

- Establishing of integration plan including testing of software units. Test plan should include activities that can expose software behavior in normal and exceptional cases (Safety issues)



5. ESTABLISHING OF 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.

This user or system requirement should then be "translated" into the software requirements specifications with definition of 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.

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.

6. RISK MANAGEMENT

A comprehensive risk management approach includes hazard analysis and mitigation that continues iteratively throughout the life of the product. The manufacturer is expected to perform a specific software hazard analysis as a part of a medical device (system) hazard analysis.

Software safety requirements are derived from the 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.

7. 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.

The completed software design specification constrains the programmer/coder to stay within the intent of the agreed upon requirements and design.

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. 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.

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.

8. 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.

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.

The manufacturer frequently adopt specific coding guidelines that establish quality policies and procedures related to the software coding process. 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 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.

9. TESTING

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.

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.

Testing of all program functionality and all program code does not mean the program is 100% correct! Furthermore, 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.

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.

9.1. White box testing

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

9.2. Black box testing

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.
- **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.

The black box testing concept is frequently used as means for assessing the suitability of Off the Self software in cases where the source code can not be made available from the original program developer.

10. CONCLUTION

Validation of Software using the principles and tasks listed above has been conducted in many segments of the software industry for more than 20 years. What is new is therefore not the technique of software validation but the regulatory enforcement of its application in the medical devices industry as indicated by the recent publication of FDA guidance documents, the harmonization of the EN 60601-1-4 "General requirements for Safety, Programmable Electrical Medical Systems" and the preparation of the new Notify Body recommendation NB-MED/2.2/Rec4 on how to handle software in the context of CE-marking of programmable medical devices.

Farum 04-01-2003

Poul Schmidt-Andersen

11. REFERENCES:

- Guidance for Industry, FDA Reviewers and Compliance on Off-The-Shelf Software Use in Medical Devices, September 9, 1999
- Guidance for the Content of Premarket Submissions for Software Contained in Medical Devices, May 29, 1998
- General Principles of Software Validation; Final Guidance for Industry and FDA Staff, January 11, 2002
- EN 60601-1-4 "General requirements for Safety, Programmable Electrical Medical Systems"
- ANSI/AAMI SW68:2001 "Medical devices software – Software life cycle processes"
- NB-MED/2.2/Rec4 (Draft) "Software and Medical devices"

12. ABOUT THE AUTHOR

Poul Schmidt-Andersen B.Sc.E.E; B.Comm.

Poul spent 17 years in the fields of R&D and marketing of medical devices for Simonsen & Weel AS. As chief engineer he was responsible for all R&D efforts within the active medical devices field such cardiac output computers, neonatal monitoring, resuscitation and diathermy equipment.
In 1987 he left the company to set up S&W Medico Teknik AB (A subsidiary of Vickers PLC, UK) in Gothenburg, Sweden. As Managing Director he was responsible for marketing resuscitation systems, neonatal therapeutic and monitoring equipment.

In 1994, Poul started the DGM - the Danish Medical Devices Certification Organization, and Notified Body for the IVDD and MDD Directives. As Managing Director of DGM he was responsible for its development until July 2001 where he resigned to start his own consultancy firm Copenhagen Medical Devices Consulting aps.

Until leaving DGM he acted as Vice-Chairman for the: "Notified Bodies Medical Devices Group" which is designed to promote co-operation between Notified Bodies, medical device manufacturers and the EU-Commission. He also worked as a convener of several working groups developing NB-MED Recommendations" including the newly finished recommendations on software requirements.

Poul is a Lead Auditor for EN ISO 9001 / EN ISO 13485 quality systems and for the Medical Devices Directive (MDD). He is a recognized FDA inspector with respect to the US requirements of the Code of Federal Regulation 21 Part 820 and has been trained by the FDA as a Pre-market Notification 510(k) reviewer under the Mutual Recognition Agreement (MRA - Medical Devices Annex) between the European Commission and the US.
In addition, he has extensive experience in the review of quality system documentation; technical files risk analysis and clinical data with the view to conformity assessment against MDD and FDA requirements.