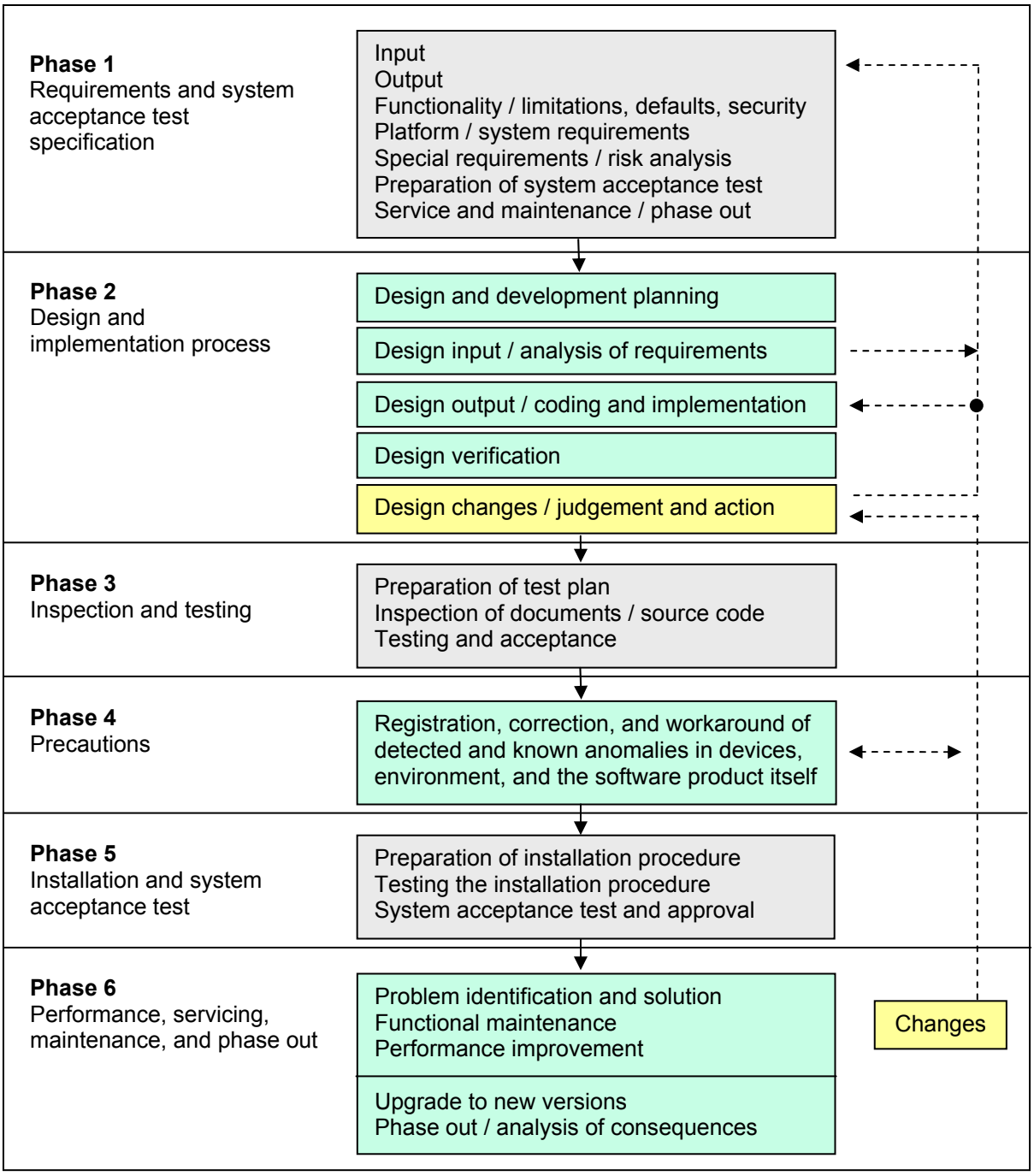

Software life cycle model

Abstract

Validation is the confirmation by examination and the provision of objective evidence that the particular requirements for a specific intended use are fulfilled [5]. Thus, validation of software is not just testing. Requirements must be specified and evidence covering the intended use must be provided. This method recommends a working strategy based on a common software life cycle model and presents the validation problems in a clear and systematic way. This method will help to establish documented evidence, which provides a high degree of assurance that the validated software product will consistently produce results meeting the predetermined specifications and quality attributes.

Table of contents

Introduction

This method is basically developed to assist accredited laboratories in validation of software for calibration and testing. The main requirements to the laboratories are stated in the Standard ISO/IEC 17025 [5]. The Danish Accreditation Body has prepared a DANAK guideline RL 10 [1] which interprets the requirements in ISO/IEC 17025 with respect to electronic data processing in the accredited laboratories. That guideline and this method are closely related.

If the laboratories comply with the requirements in ISO/IEC 17025 they will also meet the requirements of ISO 9001. The goal of this method was also to cover the situation where an accredited laboratory wants to develop and sell validated computer software on commercial basis. Therefore the Guideline ISO 9000-3 [2], which outlines requirements to be met for such suppliers, is taken into account.

Furthermore, the most rigorous validation requirements come from the medical and pharmaceutical industry. In order to let this method benefit from the ideas and requirements used in this area, the guidance from U.S. Food and Drag Administration (FDA) “General principles of software validation” [3] and the GAMP Guide [4] are intensively used as inspiration.

This method is not a guideline. It is a tool to be used for systematic and straightforward validation of various types of software. The laboratories may simply choose which elements they want to validate and which they do not. It is their option and their responsibility.

1 Definition of terms

In order to assure consistency, conventional terms used in this document will apply to the following definitions:

- *Computer system*. A group of hardware components and associated software designed and assembled to perform a specific function or group of functions [4].
- *Software*. A collection of programs, routines, and subroutines that controls the operation of a computer or a computerized system [4].
- *Software product*. The set of computer programs, procedures, and associated documentation and data [2].
- *Software item*. Any identifiable part of a software product [2].
- *Standard or configurable software packages*. Standard or configurable software packages are commercial products, which typically are used to produce customized applications (e.g. spreadsheets and executable programs). Even if the software packages themselves do not require validation, new versions should always be treated with caution and be approved before use. The applications they make should always be validated [4].
- *Custom built or bespoke systems*. Software products categorized as custom built or bespoke systems are applications that should be validated in accordance with a validation plan based on a full life cycle model [4].
- *Testing*. The process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies requirements or to identify differences between expected and actual results [4].
- *Verification*. Confirming that the output from a development phase meets the input requirements for that phase [3].
- *Validation*. Establishing by objective evidence that all software requirements have been implemented correctly and completely and are traceable to system requirements [3].
- *Revalidation*. Repetition of the validation process or a specific portion of it [4].
- *Retrospective validation*. Establishing documented evidence that a system does what it purports to do based on analysis of historical information [4].
- *Reverse engineering*. Preparing retrospective validation tasks to be conducted on existing software products (in contrast to software products under development).
- *Life cycle model*. A framework containing the processes, activities, and tasks involved in the development and maintenance of a software product, spanning the life of the software from the definition of its requirements to the termination of its use, i.e. from concept to retirement [2].
- *Design process*. Software life cycle process that comprises the activities of input requirements analysis, architectural design, and detailed function design. The design process is that which transforms the requirements into a software executable.
- *Development process*. Software life cycle process that comprises the activities of system requirements analysis, design, coding, integration, testing, installation, and support for acceptance. The development process is that which transforms the requirements into a software product [2].
- *System acceptance testing*. Documented validation that the software performs as defined in the requirements throughout anticipated operating ranges in the environment in which it will be used.
- *Dynamic testing*. Testing performed in the development process to ensure that all statements, functions, cases, and loops have been executed at least once.

- *Regression testing*. Testing to determine that changes made to correct defects have not introduced additional defects. [2]
- *Replication*. Copying a software product from one medium to another. [2]

2 Scope

Persons who use, develop, and validate software - especially software products used for calibration and testing in accredited laboratories - may use this method. Most of such software products require validation and are commonly categorized as custom built or bespoke systems. They are programs and spreadsheets that the laboratory itself develops or purchases.

This method is based on a common life cycle model and takes in consideration most aspects of normal (prospective) and retrospective validation. This method may be used for validation of:

- *Purchased software products that are not standard or configurable software packages*
- *Self-developed or purchased software products where the source code is available and known*
- *Software being developed in control of the laboratory*

2.1 Purchased software products

Purchased software products are generally subject to retrospective validation. Depending on the available information about the products, a more or less formal validation should be conducted (including at least specification of requirements and testing). In calibration and testing, as well in developing, supplying, installing and maintaining software products, purchased products may include:

- *Commercial off-the-shelf software*
- *Subcontracted development*
- *Tools to assist in the development of programs*

Purchased software products are validated to the extent required by their intended use. Large software packages may thus be just partly validated provided that the reason to do that can be documented.

2.2 Self-developed software products

Self-developed software products (including spreadsheets) developed by the laboratory by means of some commercial standard or configurable software package, require full validation. The software packages themselves do not require validation, but new versions should always be treated with caution and should be tested and approved before use. An advice: never use beta-releases.

It should especially be noted that spreadsheets are programs, and that they as such require validation. Spreadsheets may be validated as other programs, but there should be paid special attention to the fact that spreadsheets have a wide-open user interface and therefore are very vulnerable to unintentional changes.

2.3 Development, verification, and validation

While new software is being developed it may sometimes be necessary to test parts of the software. These tests have to be recorded in order to document that the development proceeded as planned.

Software products require validation. For a software product regarded as an encapsulated functional unit, the purpose of validation is to establish evidence that its requirements are met and that it performs adequately in its actual or expected surroundings.

Computer systems require validation in the environment in which they are used. The final validation may combine the individual validation tasks conducted on all the software products forming the complete computer system.

This method is designed to benefit these requirements.

3 Software life cycle model

This method recommends the use of a general life cycle model to organize the validation process. In this way, the software product can be subjected to validation in all phases of its lifetime, from the initial specification of requirements to phase out. This general life cycle model includes the following phases:

3.1 Requirements and system acceptance test specification
3.2 Design and implementation process
3.3 Inspection and testing
3.4 Precautions
3.5 Installation and system acceptance test
3.6 Performance, servicing, maintenance, and phase out

The life cycle model may thus be regarded as a validation schedule that contains the information necessary to make a proper assessment. It outlines the tasks to be performed, methods to be used, criteria for acceptance, input and output required for each task, required documentation, and the persons which are responsible for the validation.

3.1 Requirements and system acceptance test specification

The requirements describe and specify the software product and are basis for the development and validation process. A set of requirements can always be specified. In case of retrospective validation (where the development phase is irrelevant) it can at least be specified what the software is purported to do based on actual and historical facts. If the requirements specification is made in more versions, each version should be clearly identified.

When specifying requirements for spreadsheets, it should be noted that the user interface is wide-open for erroneous input and hereby provides a great risk for inadvertent changes. Thus, the spreadsheet requirements should specify input protection and/or some detailed documentation on how to use the spreadsheet program. Furthermore, it should be required that new spreadsheets should be based on templates and never on old modified spreadsheets.

3.1.1 Requirements specification

The requirements should encompass everything concerning the use of the software:

- *Version of requirements*. Information that identifies the actual version of, and changes applied, to the requirements specification.
- *Input*. All inputs that the software product will receive. Includes specification of ranges, limits, defaults, response to illegal inputs, etc.
- *Output*. All outputs that the software product will produce. Includes data formats, screen presentations, storage medium, printouts, generation of documents, etc.
- *Functionality*. All functions that the software product will provide. Includes performance requirements such as data throughput, reliability, timing, user interface features, etc.
- *Traceability*. Measures taken to ensure that critical user events are recorded and traceable.
- *Hardware control*. All device interfaces and equipments to be supported.
- *Limitations*. All acceptable and stated limitations in the software product.
- *Safety*. All precautions taken to prevent overflow and malfunction due to incorrect input or use.
- *Default settings*. All settings automatically applied after power-up, such as default input values, default instrument or program control settings, and options selected by default. Includes information on how to manage and maintain the default settings.

- *Version control*. How to identify different versions of the software product and to distinguish output from the individual versions.
- *Dedicated platform*. The operating hardware and software environment in which to use the software product, e.g. laboratory or office computer, the actual operating system, network, third-party executables such as Microsoft® Excel and Word, etc.
- *Installation*. Installation requirements, e.g. how to install and uninstall the software product.
- *How to upgrade*. How to upgrade to new versions of platforms, support tools, etc.
- *Special requirements*. Requirements stated by the International Standards to which the laboratory is committed. Security requirements, traceability, change control and back-up of records, protection of code and data, confidentiality, precautions, risks in case of errors in the software product etc.

The requirements also specify which software items must be available for correct and unambiguous use of the software product.

- *Documentation*. Description of the modes of operation and other relevant information about the software product.
- *User manual*. How to use the software product.
- *On-line help*. On-line Help provided by Windows programs.
- *Validation report*. Additional documentation stating that the software product has been validated to the extent required for its application.
- *Service and maintenance*. Documentation of service and support concerning maintenance, future updates, problem solutions, requested modifications, etc.
- *Special agreements*. Agreements between the supplier and the end-user concerning the software product where such agreements may influence the software product development and use, e.g. special editions, special analysis, or extended validation, etc.
- *Phase out*. Documentation on how (and when) to discontinue the use of the software product and how to avoid impact on existing systems and data.
- *Errors and alarms*. How to handle errors and alarms.

3.1.2 System acceptance test specification

The system acceptance test specification contains objective criteria on how the software product should be tested to ensure that the requirements are fulfilled and that the software product performs as required in the environment in which it will be used. The system acceptance test is performed after the software product has been properly installed and thus is ready for the final acceptance test and approval for use.

3.2 Design and implementation process

The design and implementation process is relevant when developing new software and when handling changes subjected to existing software. The output from this life cycle phase is a program approved and accepted for the subsequent inspection and testing phase.

The design phase may be more or less comprehensive depending on whether it is a simple spreadsheet or a large, complex program which is about to be developed, if there are many or few persons involved, or if there are special requirements for robustness etc. The design and implementation process may be divided into a number of sub-phases, each of which focusing on specific development activities and tasks.

Anomalies found and circumvented in the Design and implementation process should be described in phase 4, Precautions.

3.2.1 Design and development planning

In compliance with the complexity and schedule of the software project, a more or less detailed development plan is prepared, reviewed and approved. It is planned which part of the program should be reviewed and which criteria to use for acceptance.

Before coding and compiling it is decided, which software development tools (e.g. code generators, interpreters, compilers, linkers, and debuggers) to use. These decisions may be evident (part of the laboratory QA-system) or may be made by the persons who are in charge of the development project. If the development tools themselves can be regarded as common standard or configurable software packages, they are not subject to explicit validation. However, it should always be judged whether or not the tools are safe to use, e.g. if it is safe to use the same compiler or code generator to produce both the system-code and the test-code, which is used to test the system-code.

3.2.2 Design input

The design input phase establishes that the requirements can be implemented. Incomplete, ambiguous, or conflicting requirements are resolved with those responsible for imposing these requirements.

In the design input phase, requirements are translated into a description of the software to be implemented. The result of the design input phase is documented and reviewed as needed, which is the case if more persons are working on the project. The input design may then be presented as a detailed specification, e.g. by means of flow-charts, diagrams, module definitions etc.

Design improvements based on good interface design practice and normal utilization of programming facilities are considered as a natural part of the software solution.

3.2.3 Design output

The output from the design activity includes:

- *Architectural design specification*
- *Detailed design specification*
- *Source code*
- *User guides*

The design output must meet the design input requirements, contain or make references to acceptance criteria, and identify those characteristics of the design that are crucial to the safe and proper functioning of the product. The design output should be validated prior to releasing the software product for final inspection and testing.

3.2.3.1 Implementation (coding and compilation)

The software development tools (assemblers, basic interpreters, and high level compilers) used to produce the software executables are specified in the development plan. From the design output it should arise how they were actually used and how module and integration tests should be performed.

Support software such as Microsoft® Excel and its build-in Visual Basic for Applications (VBA) macro interpreter, C++ compilers, and other software development systems are categorized as standard or configurable software packages and are used as they are, i.e. they are not subject to explicit validation. However, all anomalies and errors that have been workaround to avoid harm to the software solution should be reported in the source code documentation.

It is recommended to keep a log of known anomalies and acquired experience that can be used by other programmers.

3.2.3.2 Version identification

As stated above, it is required that software products are identified by unambiguous version identification. This could for instance be a three-digit version number of the form "Version 1.0.0" where each digit informs about the revision level (e.g. new version, major and minor changes).

3.2.3.3 Tips on good programming practice

This section outlines the meaning of the phrase "good programming practice". It is the purpose of this requirement to obtain software that is well structured, understandable, readable, printable and inheritable (re-usable). If these simple programming rules are violated the program validation may become very difficult and maybe even impossible.

- *Modularization*. If a software solution implies different programs that perform identical measurement tasks, the identical operations should be collected in common modules. Such modules, static libraries (.LIB) and dynamic link libraries (.DLL), are easier to maintain and safer to use than inserted copies of identical source code.
- *Encapsulation*. Each object or module should be designed to perform a well-defined encapsulated function. Aggregation of non-familiar functions in the same module, or familiar functions spread over different modules, will make the source code unnecessary complex and impenetrable.
- *Functional division*. Functionality should be broken down into small manageable and testable units. Often used operations and calculations should be isolated so that identical performances are executed by the same code.
- *Strict compilation*. If a compiler offers optional error checking levels, the most rigorous level should be used. Aggressive optimizations and syntactical compiler assumptions should be avoided. Function prototypes and strict type specification should always be used.
- *Revision notes*. Programming revisions and changes to released executables should always be documented in the source code even if the changes are documented elsewhere.
- *Source code comments*. Source code should be properly documented. All relatively complex functions should have their purpose, operation, input and output parameters described. Irrelevant and temporary notes, experimental code etc. should be removed from the final edition of the source code.
- *Naming conventions*. Function and parameter names should express their meaning and use.
- *Readable source code*. Source code should be readable. Word-wrap in the text makes it difficult to read.
- *Printable source code*. Source code should be printable since it quite often will be the printout of the source code that will be used for validation. Pagination, headings, lines, and visual separation of sections and functions makes the printout easier to read.
- *Fail-safe*. The program should issue an error message whenever an error is detected and respond accordingly. Debugging options that can be used to catch run-time error conditions should never be used in released executables.

3.2.3.4 Tips on Windows® programming

Programs developed for the Windows® platform are expected to look and operate like common Windows programs known by the user. Windows programs should be intuitively and unambiguously operated by means of ordinary, self-explanatory Windows interface elements. Programs that are operated in some non-Windows conformable manner have, from a validation point of view, a great potential risk of being operated incorrectly.

Windows® allows executables to run in more than one instance, unless the programmer explicitly prevents the start of another instance when one is already running. The programmer should be aware that multiple instances will have access to the same files and data and that this may cause problems and sometimes even errors.

3.2.3.5 Dynamic testing

Source code evaluations are often implemented as code inspection and code walkthroughs. However, another aspect of good programming practice is dynamic testing performed during the implementation:

- *Statements*. All statements shall be executed at least once
- *Functions*. All functions shall be executed at least once
- *Cases*. All case segments shall be executed at least once
- *Loops*. All loops shall be executed to their boundaries

All parts of the program should be tested step-by-step during the implementation process using debugger, temporary modification and other means that can be used to avoid potential run-time errors. The programmer should explicit document if parts of the program have *not* been subject to dynamic testing.

3.2.3.6 Utilities for validation and testing

When convenient and possible, the program may be equipped with routines or functions that can be used to test or verify critical sequences and data management.

The requirements for test and evaluation should be kept in mind while the program is being developed. Without rejecting the ultimate test of all corners of the program, a well-organized structure may itself provide an adequate test of the basic issues of validation:

- Data are commonly read from a measuring device, shown graphically, and then stored in a datafile. Facilities that can read-back the stored data for review may be used to test the data-flow. If the reviewed data form an artificial recognizable pattern, the graphic display itself is tested as well.
- The simplest way of testing calculations is to prove that given input values produce the expected results. It may sometimes be convenient to create special supplementary test programs to assist in validation of complex calculations. Such test programs should also be validated.
- The condition, under which a program is operating, is normally controlled by a number of more or less predetermined parameters. By making these parameters accessible and retrievable via user interface facilities, the integrity of the program setup can be verified.

3.2.3.7 Tips on inactive code

In general, code segments and functions that are not used (dead source code) should be removed from the final software product. However, verified code intended for internal error detection, preventive testing, recovery, or future enhancements may remain in the source code provided that is properly documented.

3.2.3.8 Documentation

Human readable source code printouts are valid documentation. Programs should be properly documented so that all necessary information becomes available for the user to operate the software product correctly. The preparation of a user manual may be specified in the requirements, but additional user manuals and/or an On-line Help facilities may be produced if required.

3.2.4 Design verification

At appropriate stages of design, formal documented reviews and/or verification of the design should take place before proceeding with the next step of the development process. The main purpose of such actions is to ensure that the design process proceeds as planned.

3.2.5 Design changes

This sub-phase serves as an entry for all changes applied to the software product, also software products being subjected to retrospective validation.

Design changes and modifications should be identified, documented, reviewed, and approved before their implementation. Request for design changes may arise at any time during the software life cycle and may be imposed by detection of errors, inadequacy, revision of basic standards etc. Dealing with changes, the following tasks should be taken in consideration:

- *Documentation and justification of the change*
- *Evaluation of the consequences of the change*
- *Approving the change*
- *Implementing and verifying the change*

Minor corrections, updates, and enhancements that do not impact other modules of the program are regarded as changes that do not require an entire revalidation, since they just lead to a new updated version. Major changes leading to brand-new editions should be reviewed in order to decide the degree of necessary revalidation or even updating of the initial requirements and system acceptance test specification.

If changes are introduced as result of detected anomalies, these anomalies and the workarounds should additionally be described in phase 4, Precautions.

3.3 Inspection and testing

The inspection and testing of the software product is planned and documented in a test plan. The extent of the testing is in compliance with the requirements, the system acceptance test specification, the approach, complexity, risks, and the intended and expected use of the program.

The following elements are examined by inspection:

- *Design output*. Coding structure, documentation and compliance with the rules for good programming practice. Documentation of the design verification and review results and, if relevant, the design change control report.
- *Documentation*. The presence of program documentation, user manuals, test results etc. If required, the contents of the manuals may be approved as well.
- *Software development environment*. Data integrity, file storage, access rights, and source code protection against inadvertent damage to the program. Includes testing of installation kits and replication and distribution of the software product media.

A test plan should explicitly describe what to test, what to expect, and how to do the testing. Subsequently it should be confirmed what was done, what was the result, and if the result was approved. A test plan should take the following aspects in consideration:

- *Test objectives* Description of the test in terms of what, why, and how
- *Relevancy of tests* Relative to objectives and required operational usage
- *Scope of tests* In terms of coverage, volumes, and system complexity
- *Levels of tests* Module test, integration test, and system acceptance test
- *Types of tests* Input, functionality, boundary, performance, and usability
- *Sequence of tests* Test cases, test procedures, test data and expected results

- *Configuration tests* Platform, network, and integration with other systems
- *Calculations tests* To confirm that known inputs lead to expected outputs
- *Regression tests* To ensure that changes do not cause new errors
- *Traceability tests* To ensure that critical events during use are recorded and traceable
- *Special concerns* Testability, analysis, stress, repeatability, and safety
- *Acceptance criteria* When is the testing completed and accepted
- *Action if errors* What to do if errors are observed
- *Follow-up of test* How to follow up the testing
- *Result of testing* To approve or disapprove the testing

The test plan should be created during the development or reverse engineering phase and identify all elements that are about to be tested. It may be a good idea always to assume that there are errors – and then be happy if the assumption was wrong.

3.4 Precautions

When operating in a third-party software environment, such as Microsoft® Windows and Office, some undesirable, inappropriate, or anomalous operating conditions may exist. In cases where such conditions impact the use of the software product in some irregular way or cause malfunction, they must be clearly registered, documented, and avoided (if possible). All steps taken to workaround such conditions should also be verified and tested.

Precautionary steps may also be taken in case of discrepancies between the description of the way an instrument should operate, and the way it actually does. In either case it is a good idea to maintain a logbook of registered anomalies for other operators and programmers to use.

Minor errors in a software product may sometimes be acceptable if they are documented and/or properly circumvented.

3.5 Installation and system acceptance test

Purchased software products are normally supplied with an installation kit. Self-made software should, whenever possible, be installable via an installation kit. This will ensure that all software elements are properly installed on the host computer. The installation procedure should guide the user to obtain a safe copy of the software product. The general installation process should be validated.

A program should always be tested after being installed. The extent of the testing depends on the use of the product and the actual testing possibilities. The user could e.g. perform adequate testing following the guidelines in the validation test plan.

If the software product to install only contains small well-known updates, it may be sufficient to conduct only a partial test of the areas being updated. However, such partial testing should only be performed if the installation process previously has been completely tested and approved.

Sometimes it is recommendable to carry out the installation testing in a copy of the true environment in order to protect original data from possible fatal errors due to using a new program.

When the software product has been properly installed, the system acceptance test should be performed as required and planned in order to approve that the software product can be taken into use.

3.6 Performance, servicing, maintenance, and phase out

In this phase the software product is in use and subject to the requirements for service, maintenance performance, and support. This phase is where all activities during performance reside and where decisions about changes, revalidation, and phase out are made.

Maintenance activities for software products developed and/or used by the laboratory may typically be classified into the following:

- *Problem / solution*. This involves detection of software problems causing operational troubles. A first hand step could be to suggest or set up a well-documented temporary solution or workaround.
- *Functional maintenance*. If the software product is based on international standards, and these standards are changed, the software product, or the way it is used, should be updated accordingly.
- *Functional expansion and performance improvement*. User suggestions and requests should be recorded in order to improve the performance of the software product. Such records may provide influence on the development or evaluation of future versions of the software product.
- *New versions*. When a new version of the software product is taken into use, the effect on the existing system should be carefully analyzed and the degree of revalidation decided. The most common result of these considerations will be reentrance into the design changes sub-phase where further decisions will be made and documented. Special attention should be paid to the effect on old spreadsheets when upgrading the spreadsheet package.
- *Phase out*. Considerations should be taken on how (and when) to discontinue the use of the software product. The potential impact on existing systems and data should be examined prior to withdrawal.

Corrective actions due to errors detected in a released software product are addressed under the discipline described in the design changes clause.

4 Validation report

All validation activities should be documented and that may seem to be an overwhelming job. However, if the recommendations in this method are followed systematically, the work will become reasonable and it will be quite easy to produce a proper validation report.

This method provides a Word 2000 template "Nordtest Software Validation Report.dot" which is organized in accordance with the life cycle model stated above. There are two main tasks associated with each life cycle phase:

- *Preliminary work*. To specify/summarize the requirements (forward/reverse engineering for prospective/retrospective validation), to manage the design and development process, make the validation test plan, document precautions (if any), prepare the installation procedure, and to plan the service and maintenance phase. All documents and actions should be dated and signed.
- *Peer review and test*. To review all documents and papers concerning the validation process and conduct and approve the planned tests and installation procedures. All documents and actions should be dated and signed.

It is recommended always to mark topics that are excluded from the validation as "not relevant" or "not applicable" (n/a) – preferably with an argument – so it is evident that they are not forgotten but are deliberately skipped. Additional rows may optionally be inserted into the tables if required.

It is the intention that the validation report shall be a "dynamic" document, which is used to keep track on all changes and all additional information that currently may become relevant for the software product and its validation. Such current updating can, however, make the document more difficult to read, but never mind – it is the *contents*, not the *format*, which is important.

When validating software used in accredited work, the laboratories must be aware of the requirements specified by their National Accreditation Body and especially how to handle the option to include or exclude validation tasks. Excluded validation tasks should never be removed, but always marked as excluded with an explanatory statement. Thus, the laboratories themselves are responsible for using this method in a way, which can be accepted by their National Accreditation Body.

The software product should be designed to handle critical events (in terms of when, where, whom, and why) applied during use. Such events should be traceable through all life cycle phases and measures taken to ensure the traceability should be stated in the validation report.

It may be good validation practice to sign (by date and initials) the different parts of report as the validation proceeds, e.g. Requirements specification should be approved and signed before the Design is done, Test specifications should be approved and signed before the tests are carried out, etc. It is also important to identify the persons who are involved in the validation and are authorized to approve and sign the report, e.g.

- Other persons than those who built the software product should do the testing.
- Acceptance test should be done by the system user/owner rather than by the development team.
- The persons approving documents should not be the same as those who have authored them.

Tips on using the Software Validation Report (Word 2000 template)

A selected row in a table may be set to break across pages if its Table Properties | Row | Options check box Allow row to break across pages is checked.

The Software Validation Report contains a number of active check boxes (known as ActiveX components) used to make on/off decisions faster and easier. This implies that the documents contain macros. The check box located in section 4 "Conclusion" contains macrocode, which can lock editing of all other check boxes and hereby protect them from being inadvertently changed. However, if the actual report's ThisDocoment VBA code page is edited, the macrocode may accidentally be deleted and the lock/unlock facility will no longer work. To reestablish the facility the following macrocode should be inserted in the CheckBox46 click method:

```
Private Sub CheckBox46_Click()
    LockAllCheckBoxes Me, CheckBox46
End Sub
```

The lack of confirmation messages when clicking this check box indicates that the macro does not work properly.

5 References

[1] DANAK retningslinie, Anvendelse af edb i akkrediterede laboratorier, RL 10 af 2002.01.01

[2] DS/EN ISO 9000-3, 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, Second edition, 1997-12-15

[3] U.S. Food and Drug Administration: General Principles of Software Validation, Draft Guidance Version 1.1, June 9, 1997 (www.fda.gov/cdrh/ode/swareval.html)

[4] GAMP Guide. Validation of Automated Systems in Pharmaceutical Manufacture. Version: V3.0, March 1998

[5] DS/EN ISO/IEC 17025, General requirements for the competence of testing and calibration laboratories, First edition, 2000-04-27

[6] ISO/DIS 15189.2, Medical laboratories – Particular requirements for quality and competance, Draft 2002. (*It is intented that the method can be used by laboratories committed to this Standard, but that is, however, not documented*).