

American National Standard



ANSI/AAMI SW91:2018

Classification of defects in
health software

Objectives and uses of AAMI standards and recommended practices

It is most important that the objectives and potential uses of an AAMI product standard or recommended practice are clearly understood. The objectives of AAMI's technical development program derive from AAMI's overall mission: the advancement of medical instrumentation. Essential to such advancement are (1) a continued increase in the safe and effective application of current technologies to patient care, and (2) the encouragement of new technologies. It is AAMI's view that standards and recommended practices can contribute significantly to the advancement of medical instrumentation, provided that they are drafted with attention to these objectives and provided that arbitrary and restrictive uses are avoided.

A voluntary *standard* for a *medical device* recommends to the manufacturer the information that should be provided with or on the product, basic safety and performance criteria that should be considered in qualifying the device for clinical use, and the measurement techniques that can be used to determine whether the device conforms with the safety and performance criteria and/or to compare the performance characteristics of different products. Some standards emphasize the information that should be provided with the device, including performance characteristics, instructions for use, warnings and precautions, and other data considered important in ensuring the safe and effective use of the device in the clinical environment. Recommending the disclosure of performance characteristics often necessitates the development of specialized test methods to facilitate uniformity in reporting; reaching consensus on these tests can represent a considerable part of committee work. When a drafting committee determines that clinical concerns warrant the establishment of *minimum* safety and performance criteria, referee tests must be provided and the reasons for establishing the criteria must be documented in the rationale.

A *recommended practice* provides guidelines for the use, care, and/or processing of a medical device or system. A recommended practice does not address device performance *per se*, but rather procedures and practices that will help ensure that a device is used safely and effectively and that its performance will be maintained.

Although a device standard is primarily directed to the manufacturer, it may also be of value to the potential purchaser or user of the device as a frame of reference for device evaluation. Similarly, even though a recommended practice is usually oriented towards healthcare professionals, it may be useful to the manufacturer in better understanding the environment in which a medical device will be used. Also, some recommended practices, while not addressing device performance criteria, provide guidelines to industrial personnel on such subjects as sterilization processing, methods of collecting data to establish safety and efficacy, human engineering, and other processing or evaluation techniques; such guidelines may be useful to health care professionals in understanding industrial practices.

In determining whether an AAMI standard or recommended practice is relevant to the specific needs of a potential user of the document, several important concepts must be recognized:

All AAMI standards and recommended practices are *voluntary* (unless, of course, they are adopted by government regulatory or procurement authorities). The application of a standard or recommended practice is solely within the discretion and professional judgment of the user of the document.

Each AAMI standard or recommended practice reflects the collective expertise of a committee of health care professionals and industrial representatives, whose work has been reviewed nationally (and sometimes internationally). As such, the consensus recommendations embodied in a standard or recommended practice are intended to respond to clinical needs and, ultimately, to help ensure patient safety. A standard or recommended practice is limited, however, in the sense that it responds generally to perceived risks and conditions that may not always be relevant to specific situations. A standard or recommended practice is an important *reference* in responsible decision-making, but it should never *replace* responsible decision-making.

Despite periodic review and revision (at least once every five years), a standard or recommended practice is necessarily a static document applied to a dynamic technology. Therefore, a standards user must carefully review the reasons why the document was initially developed and the specific rationale for each of its provisions. This review will reveal whether the document remains relevant to the specific needs of the user.

Particular care should be taken in applying a product standard to existing devices and equipment, and in applying a recommended practice to current procedures and practices. While observed or potential risks with existing equipment typically form the basis for the safety and performance criteria defined in a standard, professional judgment must be used in applying these criteria to existing equipment. No single source of information will serve to identify a particular product as "unsafe". A voluntary standard can be used as one resource, but the ultimate decision as to product safety and efficacy must take into account the specifics of its utilization and, of course, cost-benefit considerations. Similarly, a recommended practice should be analyzed in the context of the specific needs and resources of the individual institution or firm. Again, the rationale accompanying each AAMI standard and recommended practice is an excellent guide to the reasoning and data underlying its provision.

In summary, a standard or recommended practice is truly useful only when it is used in conjunction with other sources of information and policy guidance and in the context of professional experience and judgment.

INTERPRETATIONS OF AAMI STANDARDS AND RECOMMENDED PRACTICES

Requests for interpretations of AAMI standards and recommended practices must be made in writing, to the AAMI Vice President, Standards Policy and Programs. An official interpretation must be approved by letter ballot of the originating committee and subsequently reviewed and approved by the AAMI Standards Board. The interpretation will become official and representation of the Association only upon exhaustion of any appeals and upon publication of notice of interpretation in the Standards Monitor Online monthly newsletter. The Association for the Advancement of Medical Instrumentation disclaims responsibility for any characterization or explanation of a standard or recommended practice which has not been developed and communicated in accordance with this procedure and which is not published, by appropriate notice, as an *official interpretation* in the AAMI Standards Monitor Online.

Classification of defects in health software

Approved 9 October 2018 by
AAMI

Approved 22 October 2018 by
American National Standards Institute, Inc.

Abstract: Provides a common language for the classification of defects occurring in health software.

Keywords: medical device, software, defects, classification, taxonomy

AAMI Standard

This Association for the Advancement of Medical Instrumentation (AAMI) standard implies a consensus of those substantially concerned with its scope and provisions. The existence of an AAMI standard does not in any respect preclude anyone, whether they have approved the standard or not, from manufacturing, marketing, purchasing, or using products, processes, or procedures not conforming to the standard. AAMI standards are subject to periodic review, and users are cautioned to obtain the latest editions.

CAUTION NOTICE: This AAMI standard may be revised or withdrawn at any time. AAMI procedures require that action be taken to reaffirm, revise, or withdraw this standard no later than five years from the date of publication. Interested parties may obtain current information on all AAMI standards by calling or writing AAMI.

All AAMI standards, recommended practices, technical information reports, and other types of technical documents developed by AAMI are voluntary, and their application is solely within the discretion and professional judgment of the user of the document. Occasionally, voluntary technical documents are adopted by government regulatory agencies or procurement authorities, in which case the adopting agency is responsible for enforcement of its rules and regulations.

Published by

AAMI
4301 N. Fairfax Drive, Suite, 301
Arlington, VA 22203-1633
www.aami.org

© 2018 by the Association for the Advancement of Medical Instrumentation

All Rights Reserved

Publication, reproduction, photocopying, storage, or transmission, electronically or otherwise, of all or any part of this document without the prior written permission of the Association for the Advancement of Medical Instrumentation is strictly prohibited by law. It is illegal under federal law (17 U.S.C. § 101, et seq.) to make copies of all or any part of this document (whether internally or externally) without the prior written permission of the Association for the Advancement of Medical Instrumentation. Violators risk legal action, including civil and criminal penalties, and damages of \$100,000 per offense. For permission regarding the use of all or any part of this document, contact AAMI at 4301 N. Fairfax Dr., Suite 301, Arlington, VA 22203-1633. Phone: (703) 525-4890; Fax: (703) 525-1067.

Printed in the United States of America

ISBN 978-1-57020-707-5

Contents

Page

Committee representation	iv
Foreword	v
1 Scope	1
2 Normative references	1
3 Terms and Definitions	1
4 Defect Codes	2
5 Taxonomy	4
Annex A (informative) Uses of defect data	39
Annex B (informative) FDA Evaluation result code to SW91 defect code mapping	41
Annex C (informative) IEC/TR 80002-1 to SW91 Defect code mapping	44
Annex D (informative) Common weakness enumeration (CWE) to SW91 defect code mapping.....	54
Annex E (informative) Examples using SW91 during root cause analysis	58
Annex F (informative) SW91 Defect category and code table.....	63
Annex G (informative) Defect inclusion criteria	68
Bibliography	69

Committee representation

Association for the Advancement of Medical Instrumentation

Software Defect Classification Working Group

The publication of AAMI SW91 as a new American National Standard was initiated by the AAMI Software Defect Classification Working Group under the auspices of the AAMI Software and Information Technology Committee. At the time this standard was published, the **AAMI Software Defect Classification Working Group** had the following members:

Cochairs: Daniel Rubery
Lisa Simone

Members: Pat Baird, Philips
Frank Clay, Philips Healthcare
Lena Cordie, Qualitas Professional Services LLC
Plamena Entcheva-Dimitrov, Preferred Clinical Research LLC
Yimin Li, St Jude Medical Inc
Mary Beth McDonald, Mary Beth McDonald Consulting
Mark Miller, Medtronic Inc Campus
Andrew O'Keeffe, Draeger Medical Systems Inc
Dave Osborn, Philips Healthcare
Hamsini Ketheswarasarma Rajaram, Dundalk Institute of Tech.
Daniel Rubery, NxStage Medical Inc
Andrea Ruth, ALR Regulatory Consulting LLC
Ray Silkaitis, Amgen Inc
Lisa Simone, FDA/CBER
Fei Wang, Fresenius Medical Care
Daidi Zhong, Chongqing University

Alternates: Patty Krantz-Zuppan, Medtronic Inc Campus
Silvana Togneri MacMahon, Dundalk Institute of Tech.
John Murray, FDA/CDRH

NOTE—Participation by federal agency representatives in the development of this document does not constitute endorsement by the federal government or any of its agencies.

Foreword

Another set of health information is compromised, another medical device unexpectedly reboots, another set of patient data results are mixed up, another device fails to perform as intended. News stories bring reports of medical device failures, and increasingly, these failures appear to be related to software quality. Use of software across the sector continues to grow. The popularity of medical- and health-related applications is exploding due to the ubiquity of portable devices and platforms, creating new business opportunities and expanding the diversity of software developers and manufacturers.

Hacking, privacy, and stolen patient information are obvious newer threats. However, analysis of past failures over the last several decades shows that similar root causes of software failures continue to occur despite new tools and methods to improve software quality. It is difficult to determine what these failures have in common, in part because no common language has been adopted and used across the industry to discuss and characterize failures. Knowledge learned is not routinely shared or leveraged across the medical device sector to improve the quality of software. A purpose for this standard is to provide the method to help address this.

The Institute of Medicine Report on Health IT and Patient Safety called for the development of measures to assess and monitor the safety of health IT. Their third recommendation was that “the ONC should work with the private and public sectors to make comparative user experiences across vendors publicly available” [12].

The report also states that “[a]nother area necessary for making health IT safer is the development of measures. Inasmuch as the committee’s charge is to recommend policies and practices that lead to safer use of health IT, the nation needs reliable means of assessing the current state and monitoring for improvement. Currently, no entity is developing such measures” [12].

A common method to identify and report the software-related defects that lead to or are associated with medical device failures could be beneficial in gathering useful data on the causes of software-related device failures. Existing methods for classifying defects focus on capturing the attributes of the defect, such as the priority, severity, and probability of recurrence, and the process activities during which these defects are inserted into the software. The values of these attributes are easily bounded. When attempting to describe the type of defect, however, values to characterize the defect are less clearly defined.

Companies often have a method to describe the defects they identify in their medical devices. However, how defects are identified and characterized may not be the same from company to company. Without a common language, it is difficult to gather and analyze common defect and failure information in a broader context, which makes it very challenging to determine if the type and number of defects seen by a company is unusual. It is possible that all companies are experiencing the same types of failures and that industry-wide training and tools could be developed to reduce the incidence of defects. Without a way to gather such information, the strength and usefulness of such methods and tools cannot be fully realized.

To address this, AAMI established a Working Group to identify a common language to describe software defects. The goal was to develop a language that is comprehensive enough to capture the defects currently being identified, and at the same time, flexible enough to allow for expansion as technology and development methods evolve. The language is intended to be neutral with respect to development methodology, programming language, and application domain so as to make it useful for a diverse set of stakeholders. Although the standard has been developed for use with health software, there is nothing specific to health software in the taxonomy.

Once individuals, companies, and communities use the same language, means can be developed to aggregate the data in a way that benefits not only medical device manufacturers, but also tool developers, consultants, trainers, and regulators. As a consequence, data and methods can be compared with peers to identify where processes can be improved. Improvements can be monitored across multiple projects and years in order to ensure these improvements are effective. The end goal is higher quality and safer software for patients and for the users of the systems that we develop.

The content of the taxonomy was developed following several guiding principles. Existing taxonomies were leveraged as appropriate, and validation activities were performed on the taxonomy using several of these existing taxonomies. The inclusion criteria for defects were based on offering an optimal depth of detail that includes many types of defects without requiring a rigid hierarchy that quickly becomes outdated. These topics are explored in Annex G.

NOTE—This foreword does not contain provisions of the ANSI/AAMI SW91, *Classification of Defects in Health Software*, but it does provide important information about the development and intended use of the document.

Classification of defects in health software

1 Scope

This document identifies a defect classification system, or taxonomy, that can be used to classify the types of defects that might exist in software. The taxonomy applies to defect root causes in all types of software (including third-party software, medical device software, test software, and manufacturing software) throughout the software development lifecycle.

The classification system is meant to be neutral with respect to

- programming language,
- methodology,
- product,
- intended use,
- risk (the consequences of failure), and
- regulatory status.

The use of a common taxonomy allows industry-wide aggregation of defect occurrence data that can be used to improve software quality. Some examples of ways in which this may be accomplished are presented as use cases in Annex A.

The way in which the standard is to be used is not prescribed. The taxonomy does not attempt to classify the severity of defects, as the consequences of any defect can only be evaluated within the context of the software's intended use. It does not attempt to describe methodologies for analyzing root cause, managing defect resolution, or assigning risk. It does not attempt to categorize defects in quality system processes.

Several annexes are included for information. Annexes B, C, and D are provided to illustrate how the taxonomy can be used to categorize various types of software problems. Annex E provides examples of cause analysis using the taxonomy. Annex F provides a reference table of all defect categories and codes, and Annex G provides the rationale for the types of defects that are in scope for the standard.

2 Normative references

There are no normative references in this document.

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1

ANOMALY

any condition that deviates from the expected based on requirements specifications, design documents, standards, etc. or from someone's perceptions or experiences. ANOMALIES may be found during, but not limited to, the review, test, analysis, compilation, or use of SOFTWARE PRODUCTS or applicable documentation

[SOURCE: IEEE 1044:1993, definition 3.1; IEC 62304:2006, definition 3.2]

3.2

COMPONENT

a subroutine, function, macro, program, program segment, method, thread, process, or any other sensible processing component

3.3

DEFECT

an imperfection or deficiency in a work product where that work product does not meet its requirements or specifications

Note 1 to entry: Cases where software meets its requirements and specifications but does not meet a user's particular needs are not considered to be defects.

Note 2 to entry: DEFECT is a more specific term than ANOMALY. See IEEE 1044:2009 figure 1 for relationships among some common terms.

[SOURCE: IEEE 1044:2009, modified – By omitting both the statement “and needs to be either repaired or replaced” from the end of the sentence, and the accompanying note to the entry containing examples. Note 1 to entry and Note 2 to entry have been added.]

3.4

FAILURE

(a) termination of the ability of a product to perform a required function or its inability to perform within previously specified limits. (adapted from ISO/IEC 25000:2005); (b) an event in which a system or system component does not perform a required function within specified limits. (adapted from ISO/IEC 24765:2009)

NOTE—A failure may be produced when a fault is encountered.

[SOURCE: IEEE 1044:2009]

3.5

HEALTH SOFTWARE

software intended to be used specifically for managing, maintaining or improving health of individual persons, or the delivery of care

Note 1 to entry: HEALTH SOFTWARE fully includes what is considered software as a MEDICAL DEVICE.

[SOURCE: IEC 62304 Ed. 2 CD]

3.6

INVOKE

cause a COMPONENT to execute. The term “invoke” is used rather than “call” because there may be no actual call as such. A task order placed in a processing queue is an invocation, though (typically) not a call. A remote procedure call is not a direct call, but is still considered an invocation.

3.7

OBJECT

a data item including constants and variables of primitive types in addition to objects as instances of a class, struct, or union

4 Defect codes

4.1. Defect code hierarchy and scope

Defects are coded in a hierarchical fashion. Each category of defect is represented by a category description and a category number, or “defect code.” Defect codes for subcategories are represented by appending a period and a separate number to the category number. There is no limit to the number of subcategories in the code. For example,

Category	1
Sub Category of 1	1.1
Sub-Category of 1.1	1.1.1

Each category that also includes subcategories contains a “General” section describing the types of defects that are included in that category. The “General” sections are not associated with defect codes because they provide general characteristics for all subcategories in the entire parent category.

Because defect codes are strings, pattern matching languages such as regular expressions can be used to identify groups of defects. For example, all category two defects are represented as “2”. The subcategories immediately under

category two are represented as "2.x". The subcategories up to three levels below category two are represented as "2.x.x.x" (this should be valid even if some subcategories do not extend to three levels). All subcategories at all depths under category two are represented collectively as "2.*" or "2***". Note that "2" identifies only defects with a code of "2", where "2*" identifies all defects with a code of "2" plus all sub-categories of "2" such as "2.1", "2.2", "2.2.1", etc. Guidelines for adding new categories are given in section 4.3.

4.2. Defect code use

To use this taxonomy, root cause analysis is performed to the greatest depth possible. If an appropriate defect category exists, the defect code for that category is assigned to the identified software defect. In cases where the cause cannot be determined to the level of detail of an individual category, then the lowest level subcategory that best captures the type of defect should be selected. For example, root cause analysis has determined that a defect exists in a third-party device driver module, but further details are not available. Information about the possible defect is consistent with the category description in the General section for Third-Party Software (3.7). In this case, the subcategory Third-Party Software (3.7) itself may be selected rather than a more specific subcategory within 3.7, such as 3.7.1, 3.7.2, 3.7.3, or 3.7.4.

Identifying the best candidate for a root cause can be aided by considering at what point in the design cycle the defect was introduced. For example, a problem is identified with a data compression algorithm. Care should be taken when considering the cause(s) associated with the algorithm problem to ensure that deficiencies in requirements and implementations are not accidentally captured as design defects. If the identified problem stems from an unclear requirement, then Requirement Clarity (2.1.2) may apply, or if a requirement was not considered at all, then Requirement Scope (2.2.2) may apply. In general, design errors that result from defective requirements are not defects with the design, nor are errors in the resulting implementation if the implementation is otherwise true to the design. In a different situation, a designer reading a correct requirement may have misinterpreted the requirement and selected an algorithm that was inappropriate for the defined requirement. In this case, the defect is a design defect (Algorithm Selection (3.10)) even if the selected algorithm correctly implements the misunderstood requirement. Note that this does not preclude other defects from being introduced during implementation, which may also be identified and categorized.

Determining root causes in features such as security and interoperability that span multiple disciplines or vendors can be challenging. Using this taxonomy, cause analysis should discover why the defect exists without focusing on the functionality of the module that contains the defect. For example, the taxonomy does not contain a category for malware detected. Instead, it contains several causes that may lead to malware arriving on the system. One possible cause may be a buffer overrun in evaluating a Loop Exit Condition (4.3.3.3). In many cases, it may be impossible to identify the exact cause, especially those caused by maintenance updates and the use of third-party software. General categories for third-party software and software updates are provided for cases where more in-depth analysis is not feasible. It is up to the analyst to decide what level in the hierarchy is adequate to capture a defect. If it is determined that there is no value to be gained by selecting a particular subcategory in the hierarchy, the analyst should stop at the level where sufficient granularity is achieved. Groups within an organization may aim for different hierarchy levels depending on their needs.

In cases where the root cause analysis uncovers several issues, the analyst should report all appropriate categories. This includes when the defect is introduced as the result of a process error. For example, while investigating a system crash due to an external stimulus, the analyst realizes that there is no requirement for what the system should do when the stimulus occurs. One root cause is what actually caused the system to crash. The missing requirement would be listed as an additional root cause (Requirement Scope (2.2.2)). In another case illustrating multiple causes, a failure occurred because the functionality in legacy code used in the device was misunderstood. Both the implementation defect and any architecture and design defect would be captured.

It is important to understand that the root cause, for the purposes of using the taxonomy, is generally not an observable symptom or related process issue. For example, consider the situation where an off-by-one error causes the system to reset. The symptom is the reset but the reset itself is not the defect. Testing may have been inadequate (and an appropriate testing defect may also be identified), although this does not mean that inadequate testing caused the reset failure. The root cause of the system reset is the off-by-one error, or Reference to Wrong Element (4.2.2.3). It is left to the company to decide if and how to capture symptoms and whether to report how the defect manifested.

It is not required that a problem actually manifests or a vulnerability be exploited in order for many of the defect codes in this taxonomy to be assigned. For example, the categories may be used for defects identified during code review, before the software is ever released.

4.3. Adding defect codes

If, after due consideration, root cause analysis indicates that no suitable subcategory exists for the identified defect and use of a parent category is not suitable, companies are permitted to define their own categories. The taxonomy has been designed to support the addition of categories if nothing appropriate can be found.

Creating new categories comes with the understanding that until new categories become part of the standard, they are limited to local reporting and data analysis. Once categories become part of the standard and additional category numbers are assigned, the company would be expected to use the new assigned category numbers for subsequent and retrospective reporting.

If a new category must be added, the user should use Other (*.99) to extend the most appropriate parent category. For example, if an organization wanted to expand the Planning (1.*) category, they could create the following new categories and defect codes:

- Project Plan (1.99.1)
- Risk Management Plan (1.99.2)
- Verification and Validation Plan (1.99.3)
- Software Development Plan (1.99.4)
- Configuration Management Plan (1.99.5)

When this user's defined categories are aggregated, they will roll up into defect category Planning (1). Any category or subcategory in the taxonomy could be so extended if it was determined that a defect category is missing.

Note that new categories should be given explicit and descriptive category names. Categories named "Other" are discouraged as it suggests that a category is being created as a substitute for adequate root cause analysis. If sufficient information is not available to create a new and unique category, the defect should be categorized using a parent defect code as described in section 4.2.

5 Taxonomy

5.1. General

The categories and subcategories included in this taxonomy are presented in the sections below. When a category contains subcategories, a general description of all defects in the category is provided in the "General" section for the parent category. "General" is not a valid defect category.

Notes and Examples are provided to assist in interpretation and to illustrate many categories; however, the information provided should not be considered comprehensive or inclusive of all possible uses for each defect category.

5.2. Planning (1)

5.2.1. General

These categories are for defects related to the planning phase of development, whether for a new product, a change to an existing product, or the retirement of an existing product. Defects in this category are always accompanied by one or more defects in subsequent lifecycle phases.

5.2.2. Plan documentation (1.1)

The plan is not documented or the documentation is incomplete or inconsistent.

Example

A defect is missed because no integration testing was performed. If the plan does not specify that integration testing is required, then the defect falls into this category. If the plan specifies the testing, but the testing is not performed, then the correct category is Test Case Completeness (5.1.2).

5.2.3. Plan feasibility (1.2)

The plan is impossible to put into action, either because of insufficient time or resources, or because dependencies in the plan cannot be met.

5.3. Requirements (2)

5.3.1. General

These categories are for defects with software requirements. This category can also be used for problems that occur when requirements from standards (e.g., performance standards) are not correctly captured.

5.3.2. Requirement correctness (2.1)

5.3.2.1. General

These categories are for defects related to the correctness of the requirements.

5.3.2.2. Requirement accuracy (2.1.1)

The information included in the requirement deviates from a parent requirement or known user need.

Note 1

When a design or implementation satisfies inaccurate requirements, the defect lies with the requirements and not the implementation.

Note 2

This is not to be confused with an unknown user need, which would be captured as a defect in Requirement Scope (2.2.2).

5.3.2.3. Requirement clarity (2.1.2)

The information included in the requirement can have more than one interpretation or is not clear enough to constrain the required functionality.

Note

This can occur if terms are not defined, metrics are not provided, synonyms are used, or no units are provided for numeric values. Unlike the category for Requirement Adequacy (2.2.3), a reader is unclear what information the requirement is trying to convey.

5.3.2.4. Requirement relevance (2.1.3)

The requirement is no longer needed or the information provided is not relevant.

Note

This may occur when the scope of the functionality changes.

Example

Addition of new functionality or “scope creep” changes the scope of the software and the requirement becomes obsolete, or old functionality is not removed from the requirements documentation.

5.3.3. Requirement completeness (2.2)

5.3.3.1. General

These categories are for defects related to the completeness of the requirements.

5.3.3.2. Requirement uniqueness (2.2.1)

The requirement is specified in more than one location, or partially overlaps a requirement or requirements identified elsewhere.

5.3.3.3. Requirement scope (2.2.2)

An entire requirement containing information necessary to specify the functionality is missing from the requirements documentation.

Note

The system level or higher level requirement, or information from which this requirement would be derived, may or may not be known.

Example 1

An accuracy requirement for a parameter is completely missing.

Example 2

Requirements for an unanticipated user workflow state are completely missing.

Example 3

During the development of a project, no requirement is specified for the properties of auditory alarm signals. Since these requirements are clearly spelled out in IEC 60601-1-8, it is a Requirement Scope defect.

5.3.3.4. Requirement adequacy (2.2.3)

The information provided within an existing requirement is clear, but not sufficient to constrain the required functionality.

Example 1

An upper bound for a parameter is specified, but a lower bound is not specified.

Example 2

The function of an algorithm is described but the performance characteristics for that algorithm are not provided.

5.3.4. Requirement logic (2.3)

5.3.4.1. General

These categories are for defects related to the reasonableness and overall sense of the requirements. Logic defects can occur in a subsystem when system level requirements are inadequate or if design constraints are not applied consistently.

5.3.4.2. Requirement soundness (2.3.1)

The requirement is illogical.

Note 1

This is often due to a self-contradiction that can be exposed by an analysis of the requirements; for example, via case analysis or model checking.

Note 2

A requirement that contradicts one or more other requirements rather than itself suggests a defect in Requirement Compatibility (2.3.4) rather than in Requirement Soundness (2.3.1).

5.3.4.3. Requirement reasonableness (2.3.2)

The requirement is logical and consistent but unreasonable with respect to the environment, budgetary or time constraints, or with the software's intended functionality.

Note

A requirement that results in unacceptable performance may result from a defect in Requirement Reasonableness, but consider that defects in architecture and design may also lead to performance issues.

5.3.4.4. Requirement achievability (2.3.3)

The requirement is fundamentally impossible to implement.

Note

Unlike Requirement Reasonableness (2.3.2), an unachievable requirement would mean violating physical constraints. Some constraints include memory size, numbers of resources, or the need to move a device faster than is physically possible.

5.3.4.5. Requirement compatibility (2.3.4)

A requirement is incompatible with other requirements at any level when it is not possible for the design to satisfy all requirements.

Note

Problems with Requirement Compatibility create a functional contradiction. Incompatible requirements may occur within the component being specified or may also arise at the interface between two components and involve the specification of configurations, formats, and protocols. They may also arise from version sensitivity, including incompatibility with previous versions of the product; for example, the use of specific hardware platforms, operating systems, or operating environments. They may also occur with requirements for quantities, ranges, tolerances, or units of measure.

Example 1

One requirement specifies that a product is sterile and another requirement specifies that the same product is non-sterile.

Example 2

An instrument transmits treatment date in YYYY-MM-DD format while the hospital information system expects the date in MM-DD-YYYY format.

Example 3

One requirement for a vital signs monitor may state that alarms will sound whenever an error occurs. A second requirement may state that alarms will only sound when blood pressure is out of range.

5.3.4.6. Requirement specificity (2.3.5)

The requirement places unneeded constraints on features or functionality of the system.

Note

This category can also be used when a requirement actually specifies some aspect of the design.

Example

Instead of specifying the characteristics of a controlled signal, a specific type of control algorithm is specified.

5.3.5. Requirement verifiability (2.4)

The requirement is correct, clear, and logical, but cannot be verified using demonstration, analysis, inspection, or testing.

Example

The acceptance criteria may be unmeasurable, or the number of possible test conditions is too large to verify to an acceptable degree.

5.4. Architecture and design (3)

5.4.1. General

These categories are for defects related to the software architecture and high level design. The lower numbered categories are for defects not attributable to a particular component or interface, but rather involve the organization of components or dependencies across components. Architecture defects can be introduced whether or not the architectural decisions related to the defects are captured in a document. This category also includes defects associated more with the initial design of the overarching decomposition and structure; for example, detailed design of interfaces between the software components, and between software components and external interfaces (hardware and software).

5.4.2. System decomposition (3.1)

Defect related to the high level decomposition of functionality into modules, data flows and interfaces.

Note

This does not include the implementation of interfaces, but the decisions made during the high level architecture design process.

5.4.3. Segregation necessary for risk control (3.2)

Defect related to segregation between software items necessary for risk control.

Example

Software safety classification may require that different software units execute on different processors.

5.4.4. Scalability (3.3)

Defect related to the ability to add additional user capacity, storage capacity, network bandwidth, performance capabilities, or other attributes related to the scale on which the software operates, where scalability is a requirement.

5.4.5. Thread/task interactions (3.4)

5.4.5.1. General

These categories are for defects related to the architecture of threads and tasks. An operating system does not need to be present for defects in this category to apply. The use of interrupts can lead to the same types of threading issues that may be present in an operating system.

5.4.5.2. Thread reentrancy (3.4.1)

Defect related to reentrance of program components, such as a reentrant component that should not be reentrant, a non-reentrant component that should be reentrant, and a reentrant call that should be non-reentrant.

5.4.5.3. Thread safety (3.4.2)

Defect in the use of interlock mechanisms and interprocess communications facilities. This includes deadlock and livelock conditions, failure to protect a shared resource, failure to lock, failure to unlock, failure to set or reset a semaphore, and duplicate locking.

Note

These are defects in the use of the mechanisms and should not be confused with defects in the mechanisms themselves.

5.4.5.4. Thread priority (3.4.3)

Defect related to task, thread, process, or interrupt priority, such as priority too low or too high, the selected priority is not allowed, or priority conflicts.

Note

In some cases, defects in this category may manifest as defects in Performance (3.5).

5.4.5.5. Transactional integrity (3.4.4)

Defect related to the integrity of data during transactions. This includes database changes, authentication sequences, and not protecting multi-instruction access from interrupts (e.g., in critical sections).

Note

Transactional Integrity defects are related to the integrity rather than the consistency of data. If inconsistencies arise between multiple copies of the data, the problem is a defect in Coherency (4.2.4).

Example 1

Failure to perform a database update in a transaction when more than one change is required. The failure of one of the changes leaves the database in an inconsistent state.

Example 2

An embedded application running on a 16-bit processor has a 32-bit value that is shared between the main loop and an interrupt service routine. Because of the processor architecture, it takes two instructions to access the 32-bit value. If an interrupt service routine (ISR) interrupts the main loop between the two instructions and modifies the value, the main loop will be left with a corrupted value.

5.4.5.6. Resource management and control (3.4.5)

Defect related to the management of shared resources.

Examples

Not returning a buffer block after use, not getting an object, failure to clean up an object after use, getting the wrong kind of object, returning an object to the wrong pool, inefficient use of swap memory, inefficient sharing of disk space, improper thread termination, and not allocating memory to threads.

5.4.6. Performance (3.5)

5.4.6.1. General

These categories are for defects related to various performance characteristics of the software.

5.4.6.2. Throughput (3.5.1)

Software is not able to perform the required processing in the allocated time.

Note 1

This assumes that the software performs the processing correctly.

Note 2

This can be due to inadequate hardware or inefficient code. Many other types of defects can present as Throughput problems. Only through proper analysis can Throughput defects be differentiated from other more specific defects like Thread Priority (3.4.3), or from defects in requirements or architecture.

5.4.6.3. Response time or delay (3.5.2)

The response time to incoming events is too long or too short at the specified load, or the delay between outgoing events is too long or too short.

Note

If the response time is too long, the root cause of this defect may be the same as for Throughput (3.5.1). If the response time is too short, the software may not be correctly determining the appropriate interval.

5.4.6.4. Capacity (3.5.3)

The maximum specified number of simultaneous users or tasks cannot be accommodated.

Note 1

This can be caused when there is no requirement for what happens when the specified capacity is exceeded.

Note 2

This category is for failures related to overloading during execution. This is different from Scalability (3.3), which is about the ability of the architecture to add additional capacity.

5.4.6.5. Resource utilization (3.5.4)

Improper utilization of processing resources.

Note 1

This includes not correctly distributing threads to the processors of a multi-core system or to the servers in a server farm.

Note 2

Defects related to utilization issues of shared resources are captured in Resource Management and Control (3.4.5).

5.4.7. Operating environment selection (3.6)

Defect caused by the selection of the wrong technology, operating or computing environment.

Note

This defect may occur if the software was designed to be run on a machine with incorrect technological characteristics, insufficient memory size or processing power, a multi-core instead of a single-core processing unit, incorrect operating system version, incorrect system generation, or other host environment problems.

5.4.8. Third-party software (3.7)

5.4.8.1. General

These categories are for defects related to the use of third-party software. These categories are not to be confused with integration problems caused by improperly using the third-party software. In cases when external software is provided or available in source code format, the taxonomy user should determine how defects in third-party software should be captured; for example, it may be possible to identify the actual root cause and capture the defect in requirements, architecture, design, etc. The taxonomy user may wish to capture the actual root cause instead of (or in addition to) the third-party-specific defects listed in this category or the third-party cause categories in the section for Release (6) defects.

Because using third-party software often limits visibility to the actual source code, further selection among these categories may not be possible. For example, consider a product that includes a third-party floating point library. The third-party library is delivered as binary code. An error is found in certain calculations, resulting in a loss of accuracy. If no specific root cause in this section can be determined due to the opacity of the library, or if the defect was reported by the vendor, the taxonomy user may simply choose to classify it as a Third-Party Software (3.7) defect.

5.4.8.2. Third-party software functional and performance requirements (3.7.1)

Defect in third-party software that prevents it from fulfilling its stated functional and performance requirements.

5.4.8.3. Third-party software selection (3.7.2)

Defect caused by incorrect selection of third-party software.

Note

In this case, the specifications of the third-party software are correctly stated by the third-party vendor, but these specifications cannot fulfill the system requirements of the system into which the third-party software is being integrated.

Example

A random number generator is selected that is not cryptographically strong.

5.4.8.4. Third-party software documentation (3.7.3)

Defect caused by problems in the documentation supplied by the vendor.

Note

It is possible to develop the interface software to the vendor's supplied specification and still have defects if the vendor's specification is not correct.

Example 1

The memory requirements for the third-party software are greater than stated in the technical specifications, causing overall system resources to be exceeded.

Example 2

The third-party software defect list does not include an identified software defect or does not fully disclose information necessary to assess the functionality and risks of using the third-party software in the system.

5.4.8.5. Extraneous third-party software functionality (3.7.4)

The third-party software includes functionality that causes or contributes to a problem and the problem is not properly mitigated during the integration of the software.

Note 1

This includes when functionality that is not required for operation has unintended consequences, including when defects arise in the unrequired functionality.

Note 2

Some regulations require that there be no unused code in a product. If unused code is present, then this category is applicable.

Example 1

Extraneous functionality that is not needed consumes necessary system resources; for example, `printf()` functionality used for debugging that is not needed in a production release.

Example 2

Error checking associated with unnecessary functionality throws exceptions or performs unwanted actions that disrupt program flow.

Example 3

An off the shelf (OTS) package includes a phone-home feature that is not needed, but exposes the device to a security vulnerability.

5.4.9. Security (3.8)

Defect related to security issues in the architecture.

Note 1

This category should be used thoughtfully. Many issues related to security involve inadequate requirements, and many security vulnerabilities can be caused by poor design or implementation. Capturing all the causes and contributing factors for a security defect should involve identifying possible defects in Requirements (2.*), Architecture and Design (3.*), and Implementation (4.*) rather than categorizing every failure associated with security as a Security (3.8) defect.

Note 2

At this level in the hierarchy, the focus is on architectures that are insecure rather than on implementation details that lead to security problems.

Note 3

This category also includes failure to authenticate a remote node before granting trust, failure to encrypt a communications channel that carries sensitive information, or failure to use access control to authorize an action.

Example 1

Unsigned software is used in a device, and/or the manufacturer allows the introduction of unknown changes after the software is deployed.

Example 2

A manufacturer has developed a mobile app based on caregivers' requests that the app allow them to view their loved ones' test results from their phones. However, for convenience, the caregivers did not want to log in each time they wanted to use the app. Instead of requiring access control on a per-use basis, a password is only used the first time the app is run. However, if the phone is lost, stolen, or traded in for an upgrade, the new owner has the potential to access private data.

5.4.10. Code reuse (3.9)

Defect related to the reuse of modules and components.

Note 1

This includes defects caused by reuse of unmaintainable code, changes made to shared code, and may also include defects caused by misunderstanding legacy code.

Note 2

This category does not include defects that result from maintenance activities or use of third-party software.

5.4.11. Algorithm selection (3.10)

An inappropriate or incorrect algorithm is selected, although the algorithm itself is correctly implemented.

Note 1

This includes incorporating an algorithm that is not appropriate for the problem at hand or that is not sufficiently robust to satisfy the stated requirements.

Note 2

This category captures design errors, not implementation errors. A complete design does not leave the implementer to guess which algorithm is intended. Similarly, defects associated with algorithms may arise from requirements that do not adequately capture what the algorithm is intended to do. If a correctly-specified algorithm is incorrectly implemented, the implementation defect should be identified rather than using this defect category.

Example 1

Using an incorrect approximation, or using a shortcut string search algorithm that assumes string characteristics that do not apply.

Example 2

Using an inappropriate battery charge algorithm for the battery type or application that leads to damage to the battery.

5.4.12. Hardware interfaces (3.11)

5.4.12.1. General

These categories are for defects related to interfaces between software and hardware. For the purposes of this standard, hardware is defined as ports or peripherals that are collocated with the processor or are accessed through a communication interface. Examples of hardware include a serial port (e.g., UART), Real Time Clock (RTC) Chip, Ethernet Controller, wireless controller, Display Controller, and General Purpose I/O pins. Defects related to interfaces with device drivers are captured in Software Interfaces (4.4).

Although the taxonomy is primarily concerned with software defects, it is relevant to capture defects related to software interactions with hardware. True hardware-related root causes are outside the scope of the taxonomy.

5.4.12.2. Interrupts/exceptions (3.11.1)

Incorrect interrupt handling or setting up for interrupts.

Note

This includes invoking the wrong handler, failure to block or unblock interrupts, and unhandled interrupts.

5.4.12.3. Hardware usage (3.11.2)

Incorrect interfaces with hardware as described in section 5.4.12.1 **General** above.

Example 1

Configuring an incorrect number of wait states in a memory controller, configuring the incorrect polarity in a serial peripheral interface (SPI), failing to select an address increment in a Direct Memory Access (DMA) controller when one is needed, incorrect initialization of or failure to initialize a device register, or misinterpreting the data from the device.

Example 2

Incorrectly using the input/output (I/O) pins of a device; for example, configuring a pin as an output when it needs to be configured as an input, disabling the pull-up resistor for a pin when it needs to be enabled, misconfiguring a multiplexed pin.

5.4.12.4. I/O Timing or throughput (3.11.3)

Defect with timing and data rates for external devices.

Note 1

This also includes not meeting specified timing requirements (too long or too short), forcing too much throughput, and mismatched data exchange rates.

Note 2

Such a defect may also result from jitter in hardware signal timing coupled with software that is not robust in the presence of such environmental fluctuations.

Example

Communications cannot be established with a device because the software does not meet the setup and hold times specified in the external device datasheet.

5.4.13. Failure to capture design (3.12)

A defect occurred because some aspect of the design was not captured in design documentation or other source of documentation, such as in coding comments or revision history comments.

Note 1

This defect category also covers situations where the design does not match the requirements.

Note 2

It is not assumed that the only allowable location to capture information about the design is in formal design specifications documents.

Note 3

This category should not be used as the only root cause unless no other category better describes the actual defect that resulted from upstream issues with design specifications.

Example

Requirements for an algorithm are correct but some important aspect of the design of the algorithm is incorrect or missing, leading to an implementation that does not meet the requirements. See Annex E for a thorough analysis of an example of this defect category.

5.5. Implementation (4)

5.5.1. General

These categories are for defects related to the lower-level design and coding of the software. Some of these categories, such as those for standards and documentation, may not affect the actual workings of the software. They are included in the taxonomy because of their impact on maintenance.

5.5.2. Data definition (4.1)

5.5.2.1. General

These categories are for defects in the definition, structure, and initialization of data. They apply whether the data is declared statically in source code or created dynamically. There may be overlap between issues in the Data Definition (4.1) and Data Use and Processing (4.2) sections.

5.5.2.2. Scalar data type (4.1.1)

5.5.2.2.1. General

These categories are for defects that occur when the data object type, as declared, is incorrect for the purpose intended.

5.5.2.2.2. Scalar data operations (4.1.1.1)

One or more methods do not exist for all the operations to be performed on the type.

5.5.2.2.3. Scaled fixed-point precision (4.1.1.2)

For a scaled integer type, the value of the least significant bit is too large to represent the data to the required precision.

Note

This may lead to quantization errors in calculations.

5.5.2.2.4. Scaled fixed-point range (4.1.1.3)

For a scaled integer type, the value of the least significant bit is too small to represent the data to the required range.

Note

This may lead to overflow errors in calculations.

5.5.2.3. Scalar size (4.1.2)

5.5.2.3.1. General

These categories are for defects that occur when an object is declared with a size that is inappropriate for its intended use. Size refers both to the number of bits in an object of primitive type and to the dimensions of an array object.

5.5.2.3.2. Scalar precision (4.1.2.1)

A floating point scalar object is declared with an insufficient number of bits in the mantissa or an integer scalar object is declared with an insufficient number of bits.

Note 1

This may lead to quantization errors in calculations.

Note 2

If an object is declared with enough bits, but the number of significant bits is insufficient, see Rounding and Significant Digits (4.2.7.2).

5.5.2.3.3. Scalar range (4.1.2.2)

A string or character array object intended to hold UTF-16 or other multi-byte characters is declared as holding UTF-8 or other single-byte characters, an integer object is declared with an insufficient number of bits, or a floating point object is declared with an insufficient number of bits in the exponent.

5.5.2.3.4. Array declaration (4.1.2.3)

An object is declared as an array of arrays when the detailed design calls for a multi-dimensional array, or vice versa.

Example

The defect is declaring an array as `int *x[3]`, when `int x[3][4]` or `int x[3,4]` is intended. The first declaration allows creation of a ragged array, and if the dereferenced arrays are not all of dimension 4 or more, then an access to `x[k][3]` may be a Reference Outside Declared Bounds (4.2.2.2).

5.5.2.3.5. Array bounds (4.1.2.4)

An object is declared as an array, but the upper and/or lower dimensions are incompatible with the design.

Note

Boundary condition issues may arise from other sources; for example, issues with array declarations, problems with Object Referencing (4.2.2.*), and problems with Data Manipulations (4.2.7.*).

Example

In order to store the total number of adverse events received each month, the design specifies that an array should be declared to hold 12 values but the array is only declared to hold 11.

```
// Array to store the number of adverse events
// for each of 12 months

int num_adverse_events[11];
```

5.5.2.3.6. Array dimensioning (4.1.2.5)

An object is declared as an array, but the number of declared dimensions is incompatible with the design.

5.5.2.4. Data alignment (4.1.3)

An object is placed in storage at a location incompatible with hardware alignment requirements.

5.5.2.5. Data initial value correctness (4.1.4)

5.5.2.5.1. General

These categories are for defects that occur when an object is not assigned an appropriate initial value or state. They also apply to variables used in other parts of the taxonomy such as those used for control logic. They should only be used for defects related to the first time a variable is set.

5.5.2.5.2. Data uninitialized (4.1.4.1)

An object is not initialized at all.

Note

If a copy or move constructor does not initialize all members, the defect is Data Cloning (4.1.4.2).

Example

A constructor is defined but not invoked, or a value is never assigned to a scalar, or if a constructor is defined, the constructor does not initialize all members.

5.5.2.5.3. Data cloning (4.1.4.2)

Problems with the cloning of data.

Note

This includes object slicing and deep copying errors.

Example 1

The copy constructor below illustrates a cloning defect wherein a pointer to dynamic memory is copied. With the copy constructor as written, if the memory allocated for the original object is freed, any subsequent use of the clone creates a Use After Free (4.2.2.4) defect, and the destructor shown below creates a Free After Free (4.2.2.5) defect.

```
class Buffer
{
public:
    char *pData;

    Buffer()
    {
        pData = new char[100];
    }
};
```