



Guidance on software aspects of dependability



This Australian Standard® was prepared by Committee QR-005, Dependability. It was approved on behalf of the Council of Standards Australia on 13 March 2014. This Standard was published on 4 April 2014.

The following are represented on Committee QR-005:

- Asset Management Council
 - Australian Organisation for Quality
 - Department of Defence (Australia)
 - Engineers Australia
 - Independent Transport Safety and Reliability Regulator
 - Institution of Professional Engineers New Zealand
 - New Zealand Society for Risk Management
 - Risk Management Institution of Australasia
 - The University of New South Wales
 - University of Wollongong
-

This Standard was issued in draft form for comment as DR AS/NZS IEC 62628.

Standards Australia wishes to acknowledge the participation of the expert individuals that contributed to the development of this Standard through their representation on the Committee and through the public comment period.

Keeping Standards up-to-date

Australian Standards® are living documents that reflect progress in science, technology and systems. To maintain their currency, all Standards are periodically reviewed, and new editions are published. Between editions, amendments may be issued.

Standards may also be withdrawn. It is important that readers assure themselves they are using a current Standard, which should include any amendments that may have been published since the Standard was published.

Detailed information about Australian Standards, drafts, amendments and new projects can be found by visiting www.standards.org.au

Standards Australia welcomes suggestions for improvements, and encourages readers to notify us immediately of any apparent inaccuracies or ambiguities. Contact us via email at mail@standards.org.au, or write to Standards Australia, GPO Box 476, Sydney, NSW 2001.

Australian Standard[®]

Guidance on software aspects of dependability

First published as AS IEC 62628:2014.

COPYRIGHT

© Standards Australia Limited

All rights are reserved. No part of this work may be reproduced or copied in any form or by any means, electronic or mechanical, including photocopying, without the written permission of the publisher, unless otherwise permitted under the Copyright Act 1968.

Published by SAI Global Limited under licence from Standards Australia Limited, GPO Box 476, Sydney, NSW 2001, Australia

ISBN 978 1 74342 687 6

PREFACE

This Standard was prepared by the Australian members of the Joint Standards Australia/Standards New Zealand Committee QR-005, Dependability. After consultation with stakeholders in both countries, Standards Australia and Standards New Zealand decided to develop this Standard as an Australian Standard rather than an Australian/New Zealand Standard.

The objective of this Standard is to provide Australian and New Zealand organizations who may be new to the areas of software development and assurance, with relevant guidance and methodology to facilitate the achievement of software dependability. It identifies the influence of management practices on dependability of software, and provides relevant technical processes to engineer software dependability into systems.

This Standard is identical with, and has been reproduced from IEC 62628 Ed 1.0 (2012), *Guidance on software aspects of dependability*.

Organizations should apply this Standard within the overall context of an AS/NZS ISO 31000 *Risk management—Principles and guidelines*, where aspects of software dependability (e.g. reliability, availability, and maintainability) have been clearly linked to the reduction of uncertainty with respect to business objectives.

This Standard is an entry-level Standard for those industries where a system lifecycle approach may not have previously been applied to software development and assurance. This Standard is not intended to revoke, supersede or be applied in preference to more sophisticated, industry-based or well-tried Standards for software dependability that are already used by some industries (e.g. rail, aerospace, automotive, and safety-related systems). Indeed, some aspects of this Standard may be seen to conflict with practices advocated by those other Standards..

For entry-level users, particular attention is drawn to Clauses 5.1, 5.2, 5.3 and Annex B where the overall system lifecycle framework for software dependability is described. Furthermore, Clauses 5.8, 6.1 and 7.4 suggest some fundamental strategies and practices for achieving software dependability.

As this Standard is reproduced from an International Standard, the following applies:

- (a) In the source text ‘this International Standard’ should read ‘this Australian Standard’.
- (b) A full point substitutes for a comma when referring to a decimal marker.

References to International Standards should be replaced by references to Australian or Australian/New Zealand Standards, as follows:

<i>Reference to International Standard</i>	<i>Australian Standard</i>
IEC	AS IEC
60300 Dependability management	60300 Dependability management
60300-3-15 Part 3-15: Application guide— Engineering of system dependability	60300.3.15 Part 3.15: Application guide— Engineering of system dependability

Only normative references that have been adopted as Australian or Australian/New Zealand Standard have been listed.

The term ‘informative’ has been used in this Standard to define the application of the annex to which it applies. An ‘informative’ annex is only for information and guidance.

CONTENTS

1	Scope	7
2	Normative references	7
3	Terms, definitions and abbreviations	7
3.1	Terms and definitions	7
3.2	Abbreviations	9
4	Overview of software aspects of dependability	9
4.1	Software and software systems	9
4.2	Software dependability and software organizations	10
4.3	Relationship between software and hardware dependability	10
4.4	Software and hardware interaction	11
5	Software dependability engineering and application.....	12
5.1	System life cycle framework	12
5.2	Software dependability project implementation	12
5.3	Software life cycle activities	13
5.4	Software dependability attributes.....	14
5.5	Software design environment	15
5.6	Establishing software requirements and dependability objectives	15
5.7	Classification of software faults	16
5.8	Strategy for software dependability implementation	17
5.8.1	Software fault avoidance	17
5.8.2	Software fault control.....	17
6	Methodology for software dependability applications	18
6.1	Software development practices for dependability achievement.....	18
6.2	Software dependability metrics and data collection.....	18
6.3	Software dependability assessment.....	19
6.3.1	Software dependability assessment process.....	19
6.3.2	System performance and dependability specification	20
6.3.3	Establishing software operational profile.....	21
6.3.4	Allocation of dependability attributes	21
6.3.5	Dependability analysis and evaluation	22
6.3.6	Software verification and software system validation	24
6.3.7	Software testing and measurement.....	25
6.3.8	Software reliability growth and forecasting.....	28
6.3.9	Software dependability information feedback	29
6.4	Software dependability improvement	29
6.4.1	Overview of software dependability improvement.....	29
6.4.2	Software complexity simplification	29
6.4.3	Software fault tolerance.....	30
6.4.4	Software interoperability	30
6.4.5	Software reuse	31
6.4.6	Software maintenance and enhancement	31
6.4.7	Software documentation	32
6.4.8	Automated tools	33
6.4.9	Technical support and user training	33

	<i>Page</i>
7 Software assurance	34
7.1 Overview of software assurance	34
7.2 Tailoring process	34
7.3 Technology influence on software assurance	34
7.4 Software assurance best practices	35
Annex A (informative) Categorization of software and software applications	37
Annex B (informative) Software system requirements and related dependability activities	39
Annex C (informative) Capability maturity model integration process	43
Annex D (informative) Classification of software defect attributes	46
Annex E (informative) Examples of software data metrics obtained from data collection	50
Annex F (informative) Example of combined hardware/software reliability functions	53
Annex G (informative) Summary of software reliability model metrics	55
Annex H (informative) Software reliability models selection and application	56
Bibliography	59
Figure 1 – Software life cycle activities	14
Figure F.1 – Block diagram for a monitoring control system	53
Table C.1 – Comparison of capability and maturity levels	43
Table D.1 – Classification of software defect attributes when a fault is found	46
Table D.2 – Classification of software defect attributes when a fault is fixed	47
Table D.3 – Design review/code inspection activity to triggers mapping	47
Table D.4 – Unit test activity to triggers mapping	48
Table D.5 – Function test activity to triggers mapping	48
Table D.6 – System test activity to triggers mapping	49
Table H.1 – Examples of software reliability models	57

INTRODUCTION

Software has widespread applications in today's products and systems. Examples include software applications in programmable control equipment, computer systems and communication networks. Over the years, many standards have been developed for software engineering, software process management, software quality and reliability assurance, but only a few standards have addressed the software issues from a dependability perspective.

Dependability is the ability of a system to perform as and when required to meet specific objectives under given conditions of use. The dependability of a system infers that the system is trustworthy and capable of performing the desired service upon demand to satisfy user needs. The increasing trends in software applications in the service industry have permeated in the rapid growth of Internet services and Web development. Standardized interfaces and protocols have enabled the use of third-party software functionality over the Internet to permit cross-platform, cross-provider, and cross-domain applications. Software has become a driving mechanism to realize complex system operations and enable the achievement of viable e-businesses for seamless integration and enterprise process management. Software design has assumed the primary function in data processing, safety monitoring, security protection and communication links in network services. This paradigm shift has put the global business communities in trust of a situation relying heavily on the software systems to sustain business operations. Software dependability plays a dominant role to influence the success in system performance and data integrity.

This International Standard provides current industry best practices and presents relevant methodology to facilitate the achievement of software dependability. It identifies the influence of management on software aspects of dependability and provides relevant technical processes to engineer software dependability into systems. The evolution of software technology and rapid adaptation of software applications in industry practices have created the need for practical software dependability standard for the global business environment. A structured approach is provided for guidance on the use of this standard.

The generic software dependability requirements and processes are presented in this standard. They form the basis for dependability applications for most software product development and software system implementation. Additional requirements are needed for mission critical, safety and security applications. Industry specific software qualification issues for reliability and quality conformance are not addressed in this standard.

This standard can also serve as guidance for dependability design of firmware. It does not however, address the implementation aspects of firmware with software contained or embedded in the hardware chips to realize their dedicated functions. Examples include application specific integrated circuit (ASIC) chips and microprocessor driven controller devices. These products are often designed and integrated as part of the physical hardware features to minimize their size and weight and facilitate real time applications such as those used in cell phones. Although the general dependability principles and practices described in this standard can be used to guide design and application of firmware, specific requirements are needed for their physical construction, device fabrication and embedded software product implementation. The physics of failure of application specific devices behaves differently as compared to software system failures.

This International Standard is not intended for conformity assessment or certification purposes.

NOTES

AUSTRALIAN STANDARD

Guidance on software aspects of dependability**1 Scope**

This International Standard addresses the issues concerning software aspects of dependability and gives guidance on achievement of dependability in software performance influenced by management disciplines, design processes and application environments. It establishes a generic framework on software dependability requirements, provides a software dependability process for system life cycle applications, presents assurance criteria and methodology for software dependability design and implementation and provides practical approaches for performance evaluation and measurement of dependability characteristics in software systems.

This standard is applicable for guidance to software system developers and suppliers, system integrators, operators and maintainers and users of software systems who are concerned with practical approaches and application engineering to achieve dependability of software products and systems.

2 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

IEC 60050-191, *International Electrotechnical Vocabulary – Chapter 191: Dependability and quality of service*

IEC 60300-3-15, *Dependability management – Part 3-15: Application guide – Engineering of system dependability*

3 Terms, definitions and abbreviations

For the purposes of this document, the terms and definitions given in IEC 60050-191, as well as the following apply.

3.1 Terms and definitions**3.1.1****software**

programs, procedures, rules, documentation and data of an information processing system

Note 1 to entry: Software is an intellectual creation that is independent of the medium upon which it is recorded.

Note 2 to entry: Software requires hardware devices to execute programs and to store and transmit data.

Note 3 to entry: Types of software include firmware, system software and application software.

Note 4 to entry: Documentation includes: requirements specifications, design specifications, source code listings, comments in source code, “help” text and messages for display at the computer/human interface, installation instructions, operating instructions, user manuals and support guides used in software maintenance.

3.1.2**firmware**

software contained in a read-only memory device, and not intended for modification

EXAMPLE Basic input/output system (BIOS) of a personal computer.

Note 1 to entry: Software modification requires the hardware device containing it to be replaced or re-programmed.

3.1.3

embedded software

software within a system whose primary purpose is not computational

EXAMPLES Software used in the engine management system or brake control systems of motor vehicles.

3.1.4

software unit

software module

software element that can be separately compiled in programming codes to perform a task or activity to achieve a desired outcome of a software function or functions

Note 1 to entry: The terms "module" and "unit" are often used interchangeably or defined to be sub-elements of one another in different ways depending upon the context. The relationship of these terms is not yet standardized.

Note 2 to entry: In an ideal situation, a software unit can be designed and programmed to perform exactly a specific function. In some applications, it may require two or more software units combined to achieve the specified software function. In such cases, these software units are tested as a single software function.

3.1.5

software configuration item

software item that has been configured and treated as a single item in the configuration management process

Note 1 to entry: A software configuration item can consist of one or more software units to perform a software function.

3.1.6

software function

elementary operation performed by the software module or unit as specified or defined as per stated requirements

3.1.7

software system

defined set of software items that, when integrated, behave collectively to satisfy a requirement

EXAMPLES Application software (software for accounting and information management); programming software (software for performance analysis and CASE tools) and system software (software for control and management of computer hardware system such as operating systems).

3.1.8

software dependability

ability of the software item to perform as and when required when integrated in system operation

3.1.9

software fault

bug

state of a software item that may prevent it from performing as required

Note 1 to entry: Software faults are either specification faults, design faults, programming faults, compiler-inserted faults or faults introduced during software maintenance.

Note 2 to entry: A software fault is dormant until activated by a specific trigger, and usually reverts to being dormant when the trigger is removed.

Note 3 to entry: In the context of this standard, a bug is a special case of software fault also known as latent software fault.

3.1.10**software failure**

failure that is a manifestation of a software fault

Note 1 to entry: A single software fault will continue to manifest itself as a failure until it is removed.

3.1.11**code**

character or bit pattern that is assigned a particular meaning to express a computer program in a programming language

Note 1 to entry: Source codes are coded instructions and data definitions expressed in a form suitable for input to an assembler, compiler, or other translator.

Note 2 to entry: Coding is the process of transforming of logic and data from design specifications or descriptions into a programming language.

Note 3 to entry: A programming language is a language used to express computer programs.

3.1.12**(computer) program**

set of coded instructions executed to perform specified logical and mathematical operations on data

Note 1 to entry: Programming is the general activity of software development in which the programmer or computer user states a specific set of instructions that the computer must perform.

Note 2 to entry: A program consists of a combination of coded instructions and data definitions that enable computer hardware to perform computational or control functions.

3.2 Abbreviations

ASIC	Application specific integrated circuit
CASE	Computer-aided software engineering
CMM	Capability maturity model
CMMI	Capability maturity model integration
COTS	Commercial-off-the-shelf
FMEA	Failure mode and effects analysis
FTA	Fault tree analysis
IP	Internet protocol
IT	Information technology
KSLOC	Kilo-(thousand) source lines of code
ODC	Orthogonal defect classification
RBD	Reliability block diagram
USB	Universal serial bus

4 Overview of software aspects of dependability**4.1 Software and software systems**

Software is a virtual entity. In the context of this standard, software refers to procedures, programs, codes, data and instructions for system control and information processing. A software system consists of an integrated collection of software items such as computer programs, procedures, and executable codes, and incorporated into physical host of the processing and control hardware to realize system operation and deliver performance functions. The hierarchy of the software system can be viewed as a structure representing the system architecture and consisting of subsystem software programs and lower-level software units. A software unit can be tested as specified in the design of a program. In some cases

two or more software units are required to construct a software function. The system encompasses both hardware and software elements interacting to provide useful functions in rendering the required performance services.

In a combined hardware/software system, the software elements of the system contribute in two major roles: a) operating software to run continuously to sustain hardware elements in system operation; and b) application software to run as and when required upon user demands for provision of specific customer services. Dependability analysis of the software sub-systems has to consider the software application time factors in the system operational profile and those software elements required for full-time system operation. Software modelling is needed for reliability allocation and dependability assessment of software-based systems.

Human aspects of dependability [1]¹ play a pivotal role in guiding effective software design and implementation. The human-machine interface and operating environment influence the outcome of software and hardware interaction and affect the dependability of system performance. This leads to a strategic need for software dependability design and perfective maintenance efforts in the software life cycle process [2].

4.2 Software dependability and software organizations

Software dependability is achieved by proper design and appropriate incorporation into system operation. This standard presents an approach where existing dependability techniques and established industry best practices can be identified and used for software dependability design and implementation. The dependability management systems [3, 4] describe where relevant dependability activities can be effectively implemented in the life cycle process. The achievement of software dependability is influenced by

- management policy and technical direction;
- design and implementation processes;
- project specific needs and application environments.

Software organizations are organized and managed groups that have people and facilities with responsibilities, authorities and relationships involving software as part of their routine activities. They exist in governments, public and private corporations, companies, associations and institutions. Software organizations are structured according to specific business needs and application environments for various combinations of development, operation and service provision.

Typical software organizations include those that

- a) develop software as their primary product,
- b) develop hardware products with embedded software,
- c) provide software service support to clients,
- d) operate and maintain software networks and systems.

Annex A describes the categorization of software and software applications provided by typical software organizations.

4.3 Relationship between software and hardware dependability

Software behaviour and performance characteristics are different than those experienced in hardware from a dependability perspective. Software codes are created by humans. They are susceptible to human errors, which are influenced by the design environment and organizational culture. Whereas most hardware component failure data are well documented and experienced in use environment, the nature of software faults and their traceability of

¹ Reference in square brackets refers to the bibliography.

cause and effects are not easy to determine in system operation. In most cases the software faults leading to system failures cannot be consistently duplicated. Corrective actions on system failures due to software faults do not guarantee total elimination of the root causes of the software problem.

A bug, after being triggered, results in a software failure (event) and exhibits as a software fault (state). All software faults that cause the inability of the software to accomplish its intended functions are noticed by the software user. Faults and bugs cause problems in the software to perform as designed. Software containing bugs could still accomplish its intended function that is not noticeable to the user. Bugs could cause failures, but could also create nuisance issues that are not affecting a certain function. A software fault can cause system failure, which may exhibit systematic failure symptom.

Software systems and hardware products also have many similarities. They both are managed throughout their design and development stages, and followed by integration and test and production. The discovery of failures and latent faults occur through rigorous analysis, test and verification process with high-levels of test or fault coverage. The high-levels of coverage of the verification process are determined by the assessment of its percentage of fault detection, or fault detection probability. While the management techniques are similar, there are also differences [5, 6]. The following are some examples:

- Software has no physical properties, while hardware does. Software does not wear-out. Failures attributable to software faults appear without advance warning and often provide no indication that they have occurred. Hardware often provides a period of gradual wear-out and possibly graceful degradation until reaching a failed condition.
- Changes to software are flexible and much less time consuming or costly as compared to hardware design changes. Changes to hardware designs require a series of time-consuming adjustments to capital equipment, material procurement, fabrication, assembly, and documentation. However, regression testing of large and complex software programs could be constrained by time and cost limitations.
- Hardware verification and testing is simplified since it is possible to conduct limited testing through knowledge of the physics of the device to analyse and predict behaviour. Software testing can also become simplified through regression testing and analysis to verify minor changes to software due to an identified failure cause. However, minor changes to correct probabilistic failure causes of software, such as race conditions, could lead to very elaborate test and verification cycles to demonstrate adequate correction of the problem.
- Repair and maintenance actions would restore hardware to its operational state generally without design changes. Software repair and maintenance would involve design changes with new service packs or software releases to correct or rectify software faults.

4.4 Software and hardware interaction

Software and hardware interaction occurs in system operation. Dependability issues exist in the interface between the hardware and the operating system. The issues are generally resolved by incorporation of error detection and correction techniques, and exception handling of the hardware and the operating system to mitigate physical faults, and information and timing errors that exist in the interaction. The advent of multi-core processors has enabled redundant multi-threading to enhance dependability in system performance. This enables the user, programmer, or system architect to influence and exploit the redundancy inherent in the multi-core processors to enhance detection and recovery from errors. This also provides opportunity for recovery from soft errors or transient errors that affect either hardware or software or both. The exploitation of increased complexity in multi-core redundancy should be taken into consideration in such applications.

In any control system, the system is controlling some physical processes of actual hardware devices such as sensors and actuators that can fail in system operation. Many of these devices contain embedded software not accessible to the system designer or architect. Examples include smart sensors that contain error detection, redundancy and some error correction features, which are driven by the embedded software. It is important to review the software control algorithms. This is to ensure that the control algorithms are resilient to bad

sensor data and missing sensor values, and that they can detect failed actuation and are capable to compensate or revert to fail-safe condition. Sensor feedback is essential to confirm successful actuation. The feedback mechanism should contain some independent checking of the effects of the commanded actuation. The control system behaviour, assumptions and failure modes should be considered in the design of the software control system.

Intentional and malicious injection of hardware faults to thwart or foil the software algorithms could happen when the system is exposed to deliberate cyber attack. For example, one can inject hardware faults into a cryptographic system to extract the key, or inject a virus into the USB device that is used to initialize a voting machine. The software and hardware interaction could create serious problems to the system operation and affect the dependability in system performance.

Interoperability problems associated with software and hardware interaction could also exist when the software is inappropriately reused in a different environment or for a different application.

The solution to dependability problems related to software and hardware interaction is to increase better understanding of how the new technological system works, and to exercise caution in conducting dependability assessment and testing to fully consider the effects of hardware failures on the software system.

5 Software dependability engineering and application

5.1 System life cycle framework

A system life cycle framework should be established to guide product development and system implementation. The framework is used for defining the system life cycle and governing the performance of the system life cycle processes. IEC 60300-3-15 describes the engineering of system dependability and life cycle implementation, which is based on the technical processes of ISO/IEC 15288 [7]. This applies to any system, whether composed of hardware, software or both.

5.2 Software dependability project implementation

Software engineering activities during the design cycle and useful life period of the system life cycle should be planned, coordinated and managed accordingly along with their hardware counterparts. Engineering activities during the useful life period would involve design changes that could be caused by high failure rates in the customer application, or hardware obsolescence while supplying spares for sustainment of operations. As the hardware changes over the product life cycle, the software would need to change as well. Changes to the software are necessary, as the system design requires forward and backward compatibility between different versions and configurations of the system design.

Dependability activities should be integrated in the respective project plans and incorporated in the system engineering tasks for effective system design, realization, implementation, operation and maintenance. The guidance to engineering dependability into systems per IEC 60300-3-15 applies to this standard. The guidance on software aspects of dependability consists of the following recommended procedures for software dependability achievement in software project implementation:

- a) identify the software application objectives and requirements relevant to the software life cycle (see 5.3) and application environment (see Clause A.2);
- b) identify the applicable software dependability attributes (see 5.4) relevant to the software project;
- c) review the adequacy of dependability management processes and available resources to support software project development and implementation (see 5.5);
- d) establish software requirements and dependability objectives (see 5.6, Annex B);

- e) classify software faults (see 5.7) and identify relevant software metrics (see 6.2, Annex E) for software dependability strategy implementation (see 5.8);
- f) apply relevant dependability methodology for software design and realization (see 6.1, 6.3);
- g) initiate dependability improvement where needed taking into consideration of various constraints and limitations for project tailoring (see 6.4, 7.2);
- h) monitor development and implementation process for control and feedback to sustain software operability and assure dependability in system operation (see Clause 7).

5.3 Software life cycle activities

The software life cycle encompasses the following activities:

- *requirements definition* identifies the system requirements for combined hardware and software elements in response to the users' needs and constraints of system applications;
- *requirements analysis* determines the feasible design options and transforms the system requirements for service applications into a technical view for hardware and software subsystem design and system development;
- *architectural design* provides a solution to meet system requirements by allocation of system elements into subsystem building blocks to establish a baseline structure for software subsystem decomposition and identify relevant software functions to meet the specified requirements;
- *detailed design* provides a design for each identified function in the system architecture and creates the needed software units and interfaces for the function which can be apportioned to software, hardware, or both. The functions apportioned to software are defined with sufficient details to permit coding and testing. The software function can be labelled as software subsystem and identified as a software configuration item for design control;
- *realization* produces the executable software units that meet verification criteria and design requirements including lower level activities in
 - *coding* of the software units;
 - *unit test* for verification of software unit to meet design requirements;
 - *subsystem test* for verification of software program functions to meet design requirements;
- *integration* assembles the software units and subsystems consistent with the architectural design configuration and installs the complete software system in the host hardware system for testing;
- *acceptance* establishes the system capability and validates the software applications to provide the required performance service for specified system operations in the target environment; software acceptance tests include lower level activities in
 - *reliability growth testing* to increase the reliability of the software system; the testing is conducted after the software system is fully integrated and executed in simulated field operational conditions representing the target environment;
 - *qualification testing* to validate acceptance of the software system for customer release;
- *software operation and maintenance* engages the software in system operation, sustains the system operational capability and responds to application service demands to deliver specific operational services;
- *software update/enhancement* improves the software performance with added features;
- *software disposal* terminates the support of specific software service.

Annex B presents typical software system requirements and related dependability activities for the software life cycle stages.

Figure 1 shows the key dependability activities important to the software life cycle identified for project implementation.

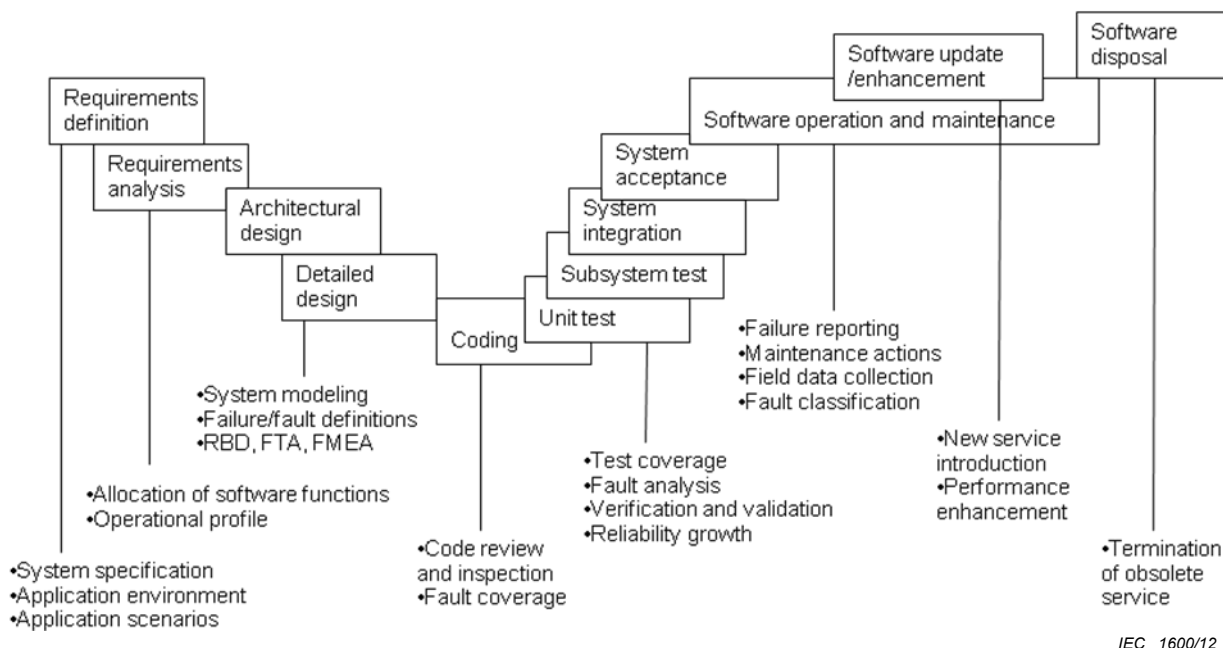


Figure 1 – Software life cycle activities

5.4 Software dependability attributes

Software dependability attributes are those characteristics inherent in the software by design. Specific application related performance attributes should be taken into consideration for incorporation in system design and construction to achieve combined hardware/software system dependability objectives.

The main software dependability attributes or inherent software dependability characteristics contributing to system dependability objectives include:

- *availability*: for readiness of software operation;
- *reliability*: for continuity of software service;
- *maintainability*: for ease of software modification, upgrade and enhancement;
- *recoverability*: for software restoration, following a failure, with or without external actions;
- *integrity*: for correctness of software data.

The specific application related performance attributes contributing to system dependability objectives include, but not limited to the following:

- *security*: for protection from intrusion in software application and use;
- *safety*: for prevention of harm in software application and use;
- *operability*: for robust, fault tolerant, and non-disruptive operation;
- *reusability*: for using an existing software for other applications;
- *supportability*: for sustaining system performance with logistic and maintenance resources;
- *portability*: for cross platform applications.

These inherent software dependability characteristics and specific application related performance attributes form the basis for software system design and application.

5.5 Software design environment

The dependability management objective is to provide a well-balanced design environment for creativity within project budget resources, time schedule and delivery targets. Organizations associated with software development and provision of software services are user application oriented. Tailoring for software projects is needed to manage the allocation of available resources and seek out appropriate design options for effective implementation. The selection and adoption of applicable processes for engineering dependability into a specific software system is accomplished through the project tailoring process for effective dependability management. The recommended tasks for implementation of tailoring process are provided in 7.2. The opportunities for outsourcing design construction, software reuse, and application of commercial-off-the-shelf (COTS) software products for system integration should be explored.

The software design environment relies on an organized process to promote good design practices for error-free code generation, minimize mistakes in requirements definition, and assure test validation for software release. The cultural aspects in software management approach often adopt a capability maturity model concept for infrastructure development [8]. This is similar to the formal implementation of *Capability maturity model integration* [9] described in Annex C for software process management. Software development is a technical process following established software engineering disciplines and application guidelines. The software design environment and practice principles should be included in the organization's policy to establish mission and goals for dependability achievement.

Software design often engages the applications of CASE (Computer-aided software engineering) tools. An effective automated system provides the computational accuracy, traceability of data, configuration management, and a means for collecting the required measurements or input metrics to the models automated. Most data collection systems for field failure reporting, analysis and corrective actions are automated for the same reasons. Historical experience data on software products and services is an indispensable and valuable asset.

5.6 Establishing software requirements and dependability objectives

Software requirements should be established for the software life cycle stages. Applicable dependability activities should be identified for implementation relevant to each stage. Timing for implementation of relevant dependability activities is important. Dependability applications are time dependent and have extensive impact on system life cycle cost [10]. Project tailoring is essential for design trade-offs and constraints resolution. The software requirements and dependability objectives should form part of the overall software product specifications. The strategy for software dependability implementation is described in 5.8. The methodology for engineering dependability into software modules or units and for building software system architecture is addressed in Clause 6. Tailoring process is described in 7.2.

The dependability activities associated with the software requirements are application specific. They reflect the software design and implementation needs to deliver the required system functions for service performance applications. Systematic approaches for implementing relevant dependability activities throughout the software life cycle would ensure the achievement of dependability objectives. Specific dependability objectives are derived from the selection of key dependability attributes and the relevant quantitative metrics. Software dependability requirements can be formulated for specific projects by using the baseline information contained in Annex B.

The influencing conditions on combined hardware/software system dependability specifications are described in the system dependability specifications [11]. The following influencing factors affecting dependability achievement in software development should be considered:

- the organization's design culture, the capability maturity process, and the experience in software design, development and implementation (see Annex C);

- understanding the application environments, user needs, and changing market dynamics for new platform or feature development for practical implementation;
- documentation processes such as failure reporting, data collection, software configuration management for control of software versions and maintenance of experience data records;
- application of software design rules for fault avoidance by controlling the design processes to optimize software performance in software complexity, program complexity, and functional complexity;
- effective use of applicable software methods and tools such as structured design, fault tolerance, design review [12], and software fault management to enhance reliability growth;
- selection of appropriate higher order of programming languages more suitable for specific software structured development;
- established requirements for qualification and measurement of software dependability characteristics.

5.7 Classification of software faults

Software faults could be classified as specification faults, design faults, programming faults, compiler-inserted faults, or faults introduced during software maintenance.

Classification of software faults provides a means for capturing and grouping relevant software fault information. The classification process helps software designers to discover unusual fault patterns for corrective actions. The objective is to eliminate the recurrence of the class of similar faults.

The *orthogonal defect classification* (ODC) [13] is a method used in software engineering for analysis of software fault (defect) data. The ODC addresses the causal effects of quality issues concerning software design and code in a procedural language environment. A defect is a non-fulfilment of a requirement related to an intended or specified use of the software. In this context, a fault due to the inability of the software to perform its required functions exhibits the characteristics of defect attributes in the ODC scheme. Defect attributes are the signature of a defect containing relevant information related to the software fault. The ODC method captures the software fault information of the defect attributes for analysis and modelling. The analysis of ODC data provides a valuable diagnostic method for evaluating the maturity of the software product at various stages of the software life cycle. The ODC can also be used to evaluate the process by analysing the types of triggers to identify specific technical needs to stimulate the missing triggers. The causal analysis of fault (defect) data presents a means for software fault reduction and reliability improvement.

The ODC defect attributes are classified as *Activity*, *Trigger*, *Target*, *Defect Type*, *Defect Qualifier*, *Source*, *Impact*, and *Age*. They are normally collected and analysed during software development to benefit design improvement. The information on defects is available at two specific points in time. When a fault is found, the circumstances leading to the fault exposure and the likely impact to the user are generally known. When a fault is closed after the fix is made, the exact nature of the fault and the scope of the fix are known. ODC categories capture the semantics of a fault (defect) from these two perspectives. By defining the *Activities* during the development process and their mapping to the ODC *Triggers*, the ODC provides valuable insights and customized the fault (defect) information for the software development organization.

The set of defect attributes is summarized: a) when a fault is found which is known as *opener section*, and b) when a fault is fixed which is known as *closer section*. ODC is most useful in mature software organizations where extensive data are normally collected and analysed for software product improvement.

Annex D presents a summary of classification of software defect attributes.

5.8 Strategy for software dependability implementation

5.8.1 Software fault avoidance

Software codes are generated to produce a software product. A mistake made during software design and coding could manifest itself when triggered, to become a software fault leading to a system failure. Since faults are the main cause of system failures, preventing faults from being introduced during design, such as code review and removing the residual faults that had escaped detection by testing, are common approaches to lessen the existence of fault problems for the software life cycle. The recommended software fault avoidance strategy includes fault prevention and fault removal.

a) Fault prevention

- Establish fault prevention objectives in software engineering disciplines.
- Initiate requirement specifications review.
- Conduct early user interaction and refinement of the software requirements.
- Introduce formal methods where applicable and practicable.
- Implement systematic techniques for software reuse and assurance for application.

b) Fault removal

- Detect and eliminate the existence of software faults by testing.
- Conduct formal inspection on finding faults, correcting faults, and verifying the corrections.
- Perform corrective and perfective maintenance actions during software in-service operation.

5.8.2 Software fault control

Software faults are difficult to detect and fault removal can be achieved by various means including rigorous software testing and inspection. Exhaustive testing of software is often limited by time and cost constraints in project management. Dependability assurance based on testing alone does not guarantee complete fault elimination. Software fault control employs fault tolerance and forecasting methods to minimize the manifestation of latent software faults or bugs that can still exist after the software release for use. The recommended software fault control strategy includes fault tolerance and fault/failure forecasting.

a) Fault tolerance

- Establish methodology for fault confinement, fault detection and fault recovery.
- Implement software design diversity and fall-back schemes.
- Introduce multi-version programming techniques.
- Implement self-checking programming techniques.

b) Fault/failure forecasting

- Establish fault/failure relationships in operational environment.
- Establish data collection system to capture relevant data.
- Conduct reliability growth testing where applicable.
- Develop and implement relevant reliability models for fault/failure estimation.
- Refine forecasting techniques for time projection of software version release.

6 Methodology for software dependability applications

6.1 Software development practices for dependability achievement

The capability maturity in software development reflects an organization's ability to develop software with consistency and dependable products for intended applications. The following fault avoidance and fault control techniques are recommended for incorporation where applicable in software development:

- a) standardize methods for high-level architectural design, detailed design, coding and testing, and documentation to facilitate communications and fault avoidance;
- b) develop modular designs for software units and subsystems with well-defined software functions and interfaces by building simple, separate and independent software units to facilitate design interaction, maintenance, error traceability, fault mitigation and bug removal;
- c) use design patterns, which are general reusable solutions of well-tested software, as templates for solving software design problems to speed up the development process;
- d) institute formal design methods where appropriate for control and documentation of software design and development process;
- e) utilize *software reliability engineering* [14] techniques for software reliability assessment and enhancement [15];
- f) reuse software available from software library on well-tested software units and subsystems for similar application and operational profile to reduce development cost and time, and minimize new design fault introduction;
- g) develop regression testing methods to ensure functionality of existing software as new functionality is introduced or fault removal is performed;
- h) testing software units and subsystems to verify low-level design functions and validate integrated high-level design architectural system performance for progressive bug removal to prevent fault propagation;
- i) conduct inspections and reviews of software design requirements, software codes, user manuals, training materials, and test documents to detect and eliminate as much as possible mistakes; different review teams for comparison of results should be considered and employed where practicable;
- j) control change to reduce fault occurrences such as version and change control process in software configuration management;
- k) analyse root cause problems and implement appropriate corrective actions for continuous software improvement;
- l) establish data collection system for knowledge base capture of software faults and performance data history.

6.2 Software dependability metrics and data collection

Software dependability metrics are measures of dependability characteristics of a software system. The measurement of metrics provides a quantitative scale and method to determine the value of a specific characteristic associated with the software system. These industry standard metrics are obtained either by direct measurement or by deduction. They are used for software system performance measurements. The following software metrics are de facto industry standards for application. They should be considered where appropriate for software system dependability assessment.

- a) *Availability*: provides a measure of up time over the duration of system operation.
- b) *Failure frequency*: provides a measure of the number of failure over the duration of system operation.
- c) *Time-to-failure*: provides a measure of the failure-free time period.

- d) *Restoration time*: provides a measure of the time for restoration of a system from a failed condition (down state) back to normal operation (up state).
- e) *Fault density*: provides a measure of the number of faults contained per kilo source lines of code (KSLOC) or per function point and is used for software reliability assessment.
- f) *Function point*: provides a measure of the functional size of application software for software project planning by means of *function point analysis* method [16].
- g) *Code coverage*: provides a measure of the degree to which the source code and the logical branches of a software program that has been systematically tested; *code coverage* is an indicator on the thoroughness of software testing, it is used to represent *fault coverage* that indicates the percentage of faults detected during the test in code execution.
- h) *Fault removal rate*: provides a measure of the number of faults detected and corrected in a software product for a defined period of time or software execution duration; *fault removal rate* is used in reliability growth to establish reliability improvement trend.
- i) *Residual faults in software*: provide a measure of the estimated number of bugs still remaining in the software product after testing for bug removal.
- j) *Time for software release*: provides a measure of the estimated time for software product release schedule based on established criteria on an acceptable level of bugs still remaining in the software product for software project management.
- k) *Software complexity*: provides a measure of the degree of difficulty for design and implementation of a software function or a software system; other complexity measures based on the complexity concept include program complexity, functional complexity, operational complexity; complexity-related metrics are used as inputs for reliability assessments and prediction models.

There are numerous metrics used for various reasons during the software life cycle. Software metrics can be grouped into three general categories to facilitate data collection.

- 1) *Fault data metrics*: capture the software problem reporting data for measuring the impact of the faults and the efficiency of the reporting process to improve software maintenance.
- 2) *Product data metrics*: capture the software product information by categorizing the size, functionality, complexity, location of use, and other characteristics to facilitate the experienced data as inputs to benefit new product development. The metrics provide performance history and data, and information of various software product groups.
- 3) *Process data metrics*: capture the software restoration process information and conditions at the time of fault detection and removal for reliability model inputs in reliability prediction.

The data collection process is critical for measuring software dependability attributes and performance characteristics. An effective data collection system should be practical for implementation. The amount and types of data should be relatively simple to collect, easy to interpret for data analysis, and useful for software dependability assessment, improvement and enhancement. The data collected is used to determine system reliability trends, frequency and time duration needed for software maintenance, response time for service calls, degraded performance restoration and maintenance support requirements.

Annex E presents examples of software data metrics obtained from data collection.

6.3 Software dependability assessment

6.3.1 Software dependability assessment process

The objective of software dependability process implementation is to ensure software system maturity in development and dependability achievement. The assessment process is the enabling mechanism to ensure verification of software requirements and validation of software dependability in system performance results. The dependability assessment process incorporates crucial software engineering activities adopted from established industry