



**CANDU<sup>®</sup> COMPUTER  
SYSTEMS  
ENGINEERING  
CENTRE OF  
EXCELLENCE**

**STANDARD**

**CE-1001-STD  
Revision 2**

**Standard for  
Software Engineering of  
Safety Critical Software**

**December 1999**



**AVAILABLE**


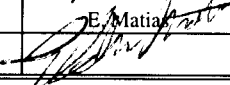
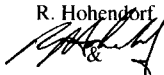

**Copyright © Atomic Energy of Canada Ltd. and  
Ontario Power Generation, Inc., 1999.**

All rights reserved by ATOMIC ENERGY OF CANADA LIMITED and  
ONTARIO POWER GENERATION, INC.

**TRADEMARKS**

CANDU<sup>®</sup> is a registered trademark of Atomic Energy of Canada Limited (AECL).

RELEASE SHEET

REV. NO.	ISSUE DATE	PREPARED BY AND DATE	REVIEWED BY AND DATE	APPROVED BY AND DATE	ISSUED FOR
0	December 1990	P.K. Joannou J. Harauz D.R. Tremaine L.T. Fong M.E. Saari A.B. Clark		P.K. Joannou	Issued for trial use for a period of 12 months.
1	January 1995	P.K. Joannou J. Harauz M. Viola R. Cirjanic D. Chan R. Whittall D.R. Tremaine		R. Hohendorf N. Ichiyen	Use
2D0	September 1999	G. Moum			Comment
2D1	November 1999	G. Moum	M. Viola		Comment
2	December 1999	G. Moum 	M. Viola 2114 Nelson & E. Matia 	R. Hohendorf  & A. Hepburn 	Use

## REVISION HISTORY SHEET

REVISION NUMBER	SECTION / PARAGRAPH	DESCRIPTION OF REVISION
0		Initial Issue.
1		Incorporated: <ul style="list-style-type: none"> <li>• Outstanding Rev 0 comments,</li> <li>• Pickering NGS B Trip Meter team comments,</li> <li>• AECL Wolsong 2 team comments,</li> <li>• AECL CANDU 3 team comments, and</li> <li>• Darlington Freeze 3 software assessment team comments.</li> </ul>
2	General	This document has been revised to reflect comments and to more closely match the organization of the <i>Standard for Software Engineering of Category II Software and Category III Software</i> [1].  This document has been revised to reflect comments from: <ul style="list-style-type: none"> <li>• a review of current industry standards, and</li> <li>• experience on the Darlington NGD Shutdown System Trip Computer Software Redesign Project.</li> </ul> This document has been reorganized so that the requirements on outputs now immediately follow the description of the process. Consequently, the contents of appendices B, C and D of revision 01 are now in Sections 3, 4 and 5 of this revision.  Minor changes that do not affect the substance of this standard are not noted in this revision history.  This document is now classified as available rather than controlled.
	1.1	The redundant last sentence regarding acceptable quality has been removed.
	1.2	This section has been revised to more closely define the scope of the standard.
	1.3	This section now reflects section 2.2 of the previous revision.
	2	This section now reflects section 2.1 of the previous revision, the <i>Standard for Computer System Engineering</i> and the relationship between this standard and the <i>Standard for Computer System Engineering</i> . The figure has been replaced with a pair of figures that have a slightly different focus and more detail.
	3	This section now documents the requirements on the development outputs previously documented in appendix B.
	3.1	This section has been revised to more closely match the organization of the <i>Standard for Software Engineering of Category II Software and Category III Software</i> .
	3.1.2	This section has been reorganized.  Item 3.1.2.c (describe the context) has been added.
	3.2	This section has been revised to more closely match the organization of the <i>Standard for Software Engineering of Category II Software and Category III Software</i> .
	3.2.2	This section has been reorganized.  Item 3.2.2.1.c (explain how performance requirements are met) has been added.  Item 3.2.2.1.d (describe the use of resources) has been added.  Item 3.2.2.2.e (strategy for managing interactions with input variables and output variables) has been added.  Item 3.2.2.2.g (describe essential elements of the software build process) has been added.
	3.3	The redundant previous item B.2.7.b (remain in states no longer than required) has been dropped.  This section has been revised to more closely match the organization of the <i>Standard for Software Engineering of Category II Software and Category III Software</i> .

REVISION NUMBER	SECTION / PARAGRAPH	DESCRIPTION OF REVISION
2 (continued)	3.3.2	This section has been reorganized. Items 3.3.2.c.1 and 3.3.2.c.3 have been clarified. Item 3.3.2.e (take advantage of compiler checks) has been added. Item 3.3.2.v (specify how the software is built) has been added.
	4	This section now documents the requirements on the verification outputs previously documented in Appendix C. This section has been revised to more closely match the organization of the <i>Standard for Software Engineering of Category II Software and Category III Software</i> .
	4.2.2.1	Item 4.2.2.1.b (verify that there are no functions not in SDD) has been added.
	4.3.1.1	Item 4.3.1.1.a (to find faults in the translation from source code) has been added. An item (to ensure by exercising each part of every code component) has been removed since it was a requirement rather than an objective.
	4.3.2.1	The objective of finding faults in the translation from source code has been added. The SDD has been added to the list of inputs.
	4.3.2.2	The previous item C.8.i (random testing) has been dropped.
	4.4.2.2	Item 4.4.2.2.b (define the reliability hypothesis) has been added. The previous item C.10.i (Ensure that the requirement regarding initial values is met) has been dropped. The previous item C.10.j (Ensure that the requirement regarding final values is met) has been dropped.
	4.5	This section has been added to generalize common requirements on verification outputs. This results in additional requirements since the requirements now uniformly apply. Item 4.5.1.j.3 (verify compliance with the SPH) has been added. Item 4.5.3.i (identify corrective action lists) has been added. Item 4.5.3.j (identify actual items used) has been added.
	5	This section now documents the requirements on the support outputs previously documented in Appendix D. Item 5.2.1.2.c.1 (estimate based on preliminary design breakdown) has been added. Item 5.2.1.2.c.10 (installation, commissioning and documentation) has been added. Item 5.2.1.3.g (identify qualification requirements for predeveloped software and software tools) has been added. Item 5.2.3.g (define a mechanism for identifying baselines) has been added.
	6	The glossary has been revised to reflect a wider scope that includes the other standards in this family of standards.
	7	The bibliography has been updated to reflect current significant developments in relevant software engineering topics.
	A	Item A.d (Describe the problem domain) has been added. Item A.e (Define human interface requirements) has been added. Item A.h (Define quality assurance requirements to be met) has been added. Item A.i (Define the quality objectives, if different) has been added. The previous item A.d (identify and describe each computer) has been removed.
	B	This section ( <i>Rationale for the Requirements of this Standard</i> ) was previously Section 2.4.
	B.1	The <i>quality objectives</i> and their definitions have been revised to more closely match those in the <i>Standard for Software Engineering of Category II Software and Category III Software</i> .

REVISION NUMBER	SECTION / PARAGRAPH	DESCRIPTION OF REVISION
2 (continued)	B.3  B.4	Item B.3.i (Behaviour over the full range of inputs shall be specified) has been added. Item B.3.j (All errors reported by other components shall be checked and handled) has been added. Item B.3.k (The design shall be conservative and simple) has been added. This section documents the coverage of quality attributes which was previously documented by the organization of the sections within Appendix B.

## FOREWORD

This standard has been jointly prepared by Ontario Power Generation, Nuclear and Atomic Energy of Canada Limited to specify the requirements for the engineering of *safety critical software* used in *real-time* protective, control and monitoring systems in CANDU<sup>®</sup> nuclear generating stations.

This standard is part of a family of standards defining the engineering requirements for different classes and categories of real-time *software*, which are distinguished by criticality of application, complexity of system and source of supply.

This standard differs from the Standard for Software Engineering of Category II Software and Category III Software by requiring systematic (mathematical) *verification*, *hazards analysis* and *reliability qualification* and by imposing more rigorous requirements on the other processes identified.

This standard is not intended to specified all quality assurance program requirements, rather it provides specific requirements for *software engineering*. The quality assurance program requirements not addressed in this standard are addressed in the AECL quality assurance program and the Ontario Power Generation, Nuclear set of Governing Documents.

Although the standard is meant to be *methodology* independent, it is written assuming the use of *procedural languages*. The principles apply equally well to other methodologies, such as Functional Graphical Languages, but in such cases, some specific details require tailoring.

A number of national and international standards have been consulted in the preparation of this software standard. The largest contribution has come from the two standards listed below. This standard also owes much of its content to the experience gained in developing, licensing, and maintaining software for the Darlington Nuclear Generating Station Shutdown System Trip Computers, as well as other safety critical software.

- IEC 60880, “Software for Computers in the Safety Systems of Nuclear Power Stations”
- CAN/CSA-Q396.1.1-89, “Quality Assurance Program for the Development of Software Used in Critical Applications”

Other standards, guides and papers used are listed in the bibliography. Documents directly referred to are listed in the references section.

Italicized words are used in this standard to denote words defined in the Glossary (Section 6). In general, such a word is italicized the first time that it appears in a section of this standard.





## TABLE OF CONTENTS

<b>1</b>	<b>INTRODUCTION .....</b>	<b>1</b>
1.1	Purpose .....	1
1.2	Scope .....	2
1.3	Structure of this Document .....	2
<b>2</b>	<b>SOFTWARE ENGINEERING PROCESS .....</b>	<b>3</b>
<b>3</b>	<b>DEVELOPMENT PROCESSES.....</b>	<b>7</b>
3.1	Requirements Definition.....	7
3.2	Design.....	9
3.3	Code Implementation.....	12
<b>4</b>	<b>VERIFICATION AND VALIDATION PROCESSES.....</b>	<b>15</b>
4.1	Review .....	15
4.1.1	Requirements Review .....	15
4.1.2	Design Review .....	16
4.1.3	Code Review .....	17
4.2	Systematic Verification.....	17
4.2.1	Systematic Design Verification.....	17
4.2.2	Systematic Code Verification.....	18
4.3	Testing .....	19
4.3.1	Unit Testing.....	19
4.3.2	Integration Testing .....	21
4.3.3	Validation Testing.....	22
4.4	Other .....	23
4.4.1	Hazards Analysis.....	23
4.4.2	Reliability Qualification.....	24
4.5	Common Verification and Validation Output Requirements.....	25
4.5.1	Test Procedures .....	25
4.5.2	Test Reports .....	26
4.5.3	Reports .....	27
<b>5</b>	<b>SUPPORT PROCESSES .....</b>	<b>28</b>
5.1	Objectives .....	28
5.1.1	Planning .....	28
5.1.2	Configuration Management.....	29
5.1.3	Training .....	29
5.2	Requirements on the SDP .....	30
5.2.1	Project Plan .....	30
5.2.2	Documentation Plan .....	33
5.2.3	Configuration Management Plan.....	34
5.2.4	Training Plan.....	35
5.3	Requirements on the SPH .....	35
5.3.1	Process Standards and Procedures .....	35
5.3.2	Documentation Standards and Procedures.....	36
5.3.3	Configuration Management Standards and Procedures.....	37
5.3.4	Training Standards and Procedures.....	38
<b>6</b>	<b>GLOSSARY .....</b>	<b>39</b>

---

<b>7</b>	<b>REFERENCES .....</b>	<b>51</b>
<b>8</b>	<b>BIBLIOGRAPHY .....</b>	<b>52</b>
	<b>APPENDIX A PREREQUISITES FOR THE SOFTWARE ENGINEERING PROCESS.....</b>	<b>57</b>
	<b>APPENDIX B RATIONALE FOR THE REQUIREMENTS OF THIS STANDARD.....</b>	<b>59</b>
	B.1 Quality Objectives .....	60
	B.2 Quality Attributes .....	61
	B.3 Software Engineering Principles.....	62
	B.4 Coverage of the Quality Attributes .....	64
	B.4.1 Coverage of the Quality Attributes by the SRS.....	64
	B.4.2 Coverage of the Quality Attributes by the SDD.....	65
	B.4.3 Coverage of the Quality Attributes by the Code .....	66
	<b>INDEX.....</b>	<b>67</b>

## LIST OF FIGURES AND TABLES

Figure 2-1 - Software Engineering Process: Development, Review and Testing.....	4
Figure 2-2 - Software Engineering Process: Systematic Verification, Hazards Analysis and Reliability Qualification....	5
Table 2-1 - Software Engineering Processes and Outputs.....	6
Table 5.2.1.3 - Independence Requirements .....	32

# 1 INTRODUCTION

## 1.1 Purpose

This standard applies to the *software engineering* of *safety critical software* used in *real-time* protective, control and monitoring systems. Software engineering includes requirements definition, design, code implementation, review, systematic verification, testing, hazards analysis, reliability qualification, planning, configuration management and training processes.

This standard applies to safety critical software (also known as *category I software*) which is part of a *special safety system* and which is directly required for the *special safety system* to meet its minimum allowable performance requirements as defined for the specific project. In a CANDU nuclear generating station, the four special safety systems are the two shutdown systems, the emergency core cooling system and the containment system.

This standard defines:

- A minimum set of software engineering processes to be followed in creating and revising the software,
- The minimum set of outputs to be produced by the processes, and
- Requirements for the content of the outputs.

The requirements in this standard are not intended to unnecessarily constrain the *methodologies* and work practices used for producing the outputs. Thus, requirements are imposed on the outputs produced, not on the methodology used to create the outputs. As well, flexibility in the processes followed and documents produced is allowed, within the constraint of meeting all requirements presented in this standard. For example, two documents may be combined into a single document, as long as the cumulative requirements presented are met by that new document. Document names may be changed to suit the project conventions. Levels of testing may be combined or split, if the project organization warrants it, providing that the coverage and other requirements of this standard are met.

This standard requires that the software engineering project using it adopt and/or define and document the *standards*, *procedures*, *methodologies*, and *guidelines* for the work practices that will be employed (including this standard). These standards, procedures, methodologies, and guidelines shall be followed throughout the system life to provide confidence that the software within the system is of acceptable quality from inception until retirement.

## 1.2 Scope

This standard applies to software engineering of safety critical software used in real-time protective, control and monitoring systems used in CANDU<sup>®</sup> nuclear generating stations.

The standard is not intended to apply directly to systems programmed in non-*procedural languages* such as function-block languages, although a tailored version of this standard could be used for such applications.

This standard does not explicitly impose requirements on the acquisition or qualification of *predeveloped software*. However, there may be requirements imposed on predeveloped software coming from the engineering of software according to this standard.

This standard is not intended to apply directly to software not originally developed to this standard. This standard is not intended to apply directly to the modification of existing software not developed to this standard.

## 1.3 Structure of this Document

Section 2 describes the software engineering process.

Section 3 describes the processes of, inputs to, and outputs from the *development* process. Section 4 describes the processes of, inputs to, and outputs from the *verification* and *validation* process. Section 5 describes the processes of, inputs to, and outputs from the *support* processes. These sections also list the requirements to be met by the outputs of these processes.

The glossary (Section 6) contains terminology necessary to understand this document.

Appendix A specifies minimum requirements for the *Design Input Documentation*.

Appendix B documents the rationale for the requirements specified in this standard.

Throughout this standard,

- Terms shown in *Italics* are defined in the glossary (Section 6),
- Numbers within square brackets “[ ]” indicate references listed in Section 7 and
- Section numbers, where referred to, pertain to this standard.

## 2 SOFTWARE ENGINEERING PROCESS

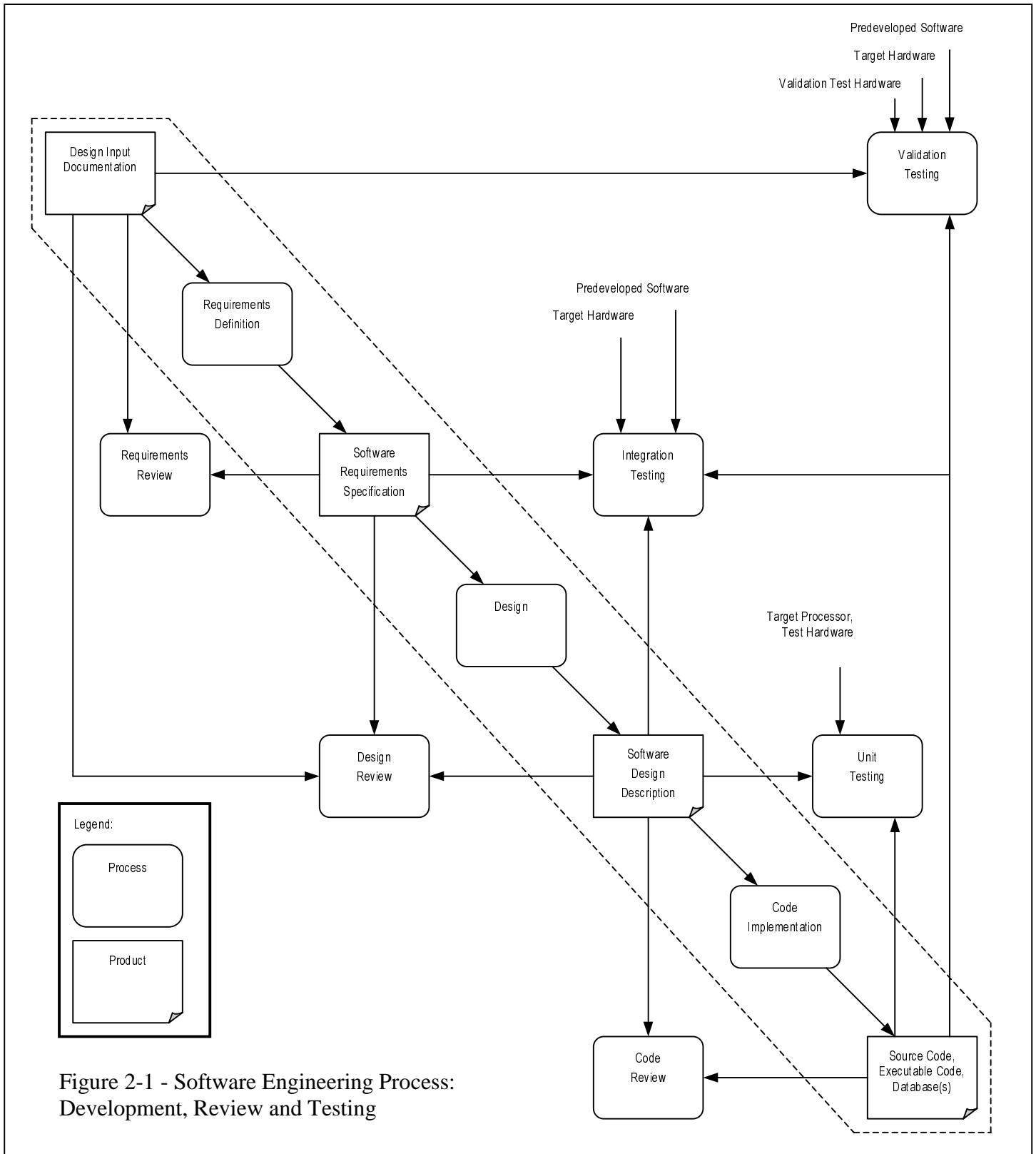
This standard defines a minimum set of processes that must be part of the *software engineering* effort for *category I software*. These processes are *development, verification, validation* or *support* processes. For each process defined, this standard identifies the outputs that must be produced and the requirements that those outputs must satisfy.

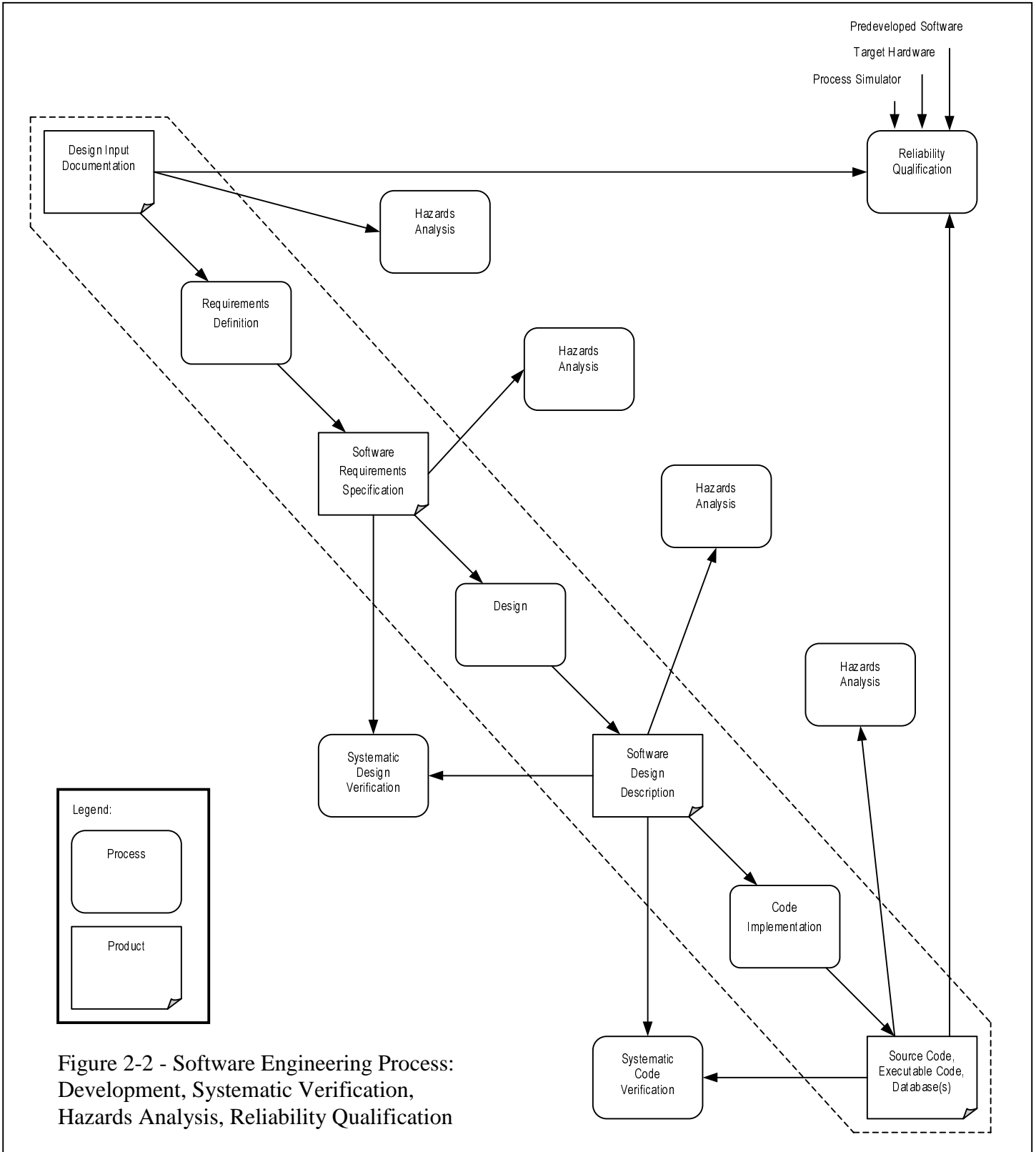
The software engineering process is a distinct set of activities that forms an integral part of the engineering of a computer system such as described in the Standard for Computer System Engineering [2]. The prerequisite information for the software engineering process is collectively referred to as the *Design Input Documentation (DID)* and is described in Appendix A. If the *Standard for Computer System Engineering* has been applied, this information will be contained as part of its *development outputs*.

The intent of the software engineering process is to achieve the required outputs through a comprehensive sequence of development, verification and validation steps. These steps are shown in Figures 2-1 and 2-2. Development involves requirements definition followed by design followed by *code implementation*. This is shown in the main diagonal in the two figures. Verification and validation of the development process outputs are performed after the outputs are produced. Figure 2-1 shows development with the *review* and *testing* portions of verification and validation. Figure 2-2 shows development with the systematic verification, *hazards analysis*, and *reliability qualification* portions of verification. For clarity, these figures do not show the verification or validation products. The outputs of all processes are listed in Table 2-1.

In practice, this sequential model is not followed exactly since there is feedback to preceding processes and iterations are required. However, when iteration occurs, it must be performed in conformance with the established procedures. Any release of the software intended for use in actual operation, including maintenance releases, must satisfy all requirements as if the sequential model had been followed.

The software engineering process for a specific *project* is completely defined in the Software Development Plan (SDP) and the Standards and Procedures Handbook (SPH), which are produced to meet the requirements of this standard. The requirements specified in the SDP and SPH shall be followed by the project.





**Figure 2-2 - Software Engineering Process: Development, Systematic Verification, Hazards Analysis, Reliability Qualification**

Table 2-1 lists all processes required for the software engineering of *safety critical software*. For each process, this table lists the name of the process, the name(s) and acronyms of the required output(s) and the section of this standard describing the process (objectives, lists of inputs and outputs) and the requirements on the output(s).

<b>Table 2-1 - Software Engineering Processes and Outputs</b>		
<b>Process</b>	<b>Output(s) *</b>	<b>Described In</b>
<b>Development</b>		
Requirements Definition	Software Requirements Specification (SRS)	3.1
Design	Software Design Description (SDD)	3.2
Code Implementation	Source Code, Executable Code, Database(s)	3.3
<b>Verification - Review</b>		
Requirements Review	Requirements Review Report (RRR)	4.1.1
Design Review	Design Review Report (DRR)	4.1.2
Code Review	Code Review Report (CRR)	4.1.3
<b>Verification - Systematic Verification</b>		
Systematic Design Verification	Design Verification Report (DVR)	4.2.1
Systematic Code Verification	Code Verification Report (CVR)	4.2.2
<b>Verification - Testing</b>		
Unit Testing	Unit Test Procedures (UTP)	4.3.1
	Unit Test Report (UTR)	
Integration Testing	Integration Test Procedures (ITP)	4.3.2
	Integration Test Report (ITR)	
Validation Testing	Validation Test Procedures (VTP)	4.3.3
	Validation Test Report (VTR)	
<b>Verification - Hazards Analysis</b>	Hazards Analysis Report (HAR)	4.4.1
<b>Verification - Reliability Qualification</b>	Reliability Qualification Report (RQR)	4.4.2
<b>Support</b>		
Planning	Software Development Plan (SDP)	5.2
	Project Plan	5.2.1
	Documentation Plan	5.2.2
	Configuration Management Plan	5.2.3
	Training Plan	5.2.4
	Standards and Procedures Handbook (SPH)	5.3
Configuration Management	Releases	5.3.3
	Approved Change Requests	
	Configuration Management Records	
Training	Training Records	5.3.4

\* Design notes are outputs that can come from any process. Generic requirements on design notes are in Section 5.2.2.



### 3 DEVELOPMENT PROCESSES

This section describes the objectives, inputs and outputs of the *development* processes. These processes are:

- Requirements Definition,
- Design, and
- *Code Implementation.*

#### 3.1 Requirements Definition

##### 3.1.1 Objectives

The objectives of the Requirements Definition process are:

- (a) To analyze and document the requirements for the software.
- (b) To identify and document any implementation constraints on the software.

The inputs to this process are the DID and SPH. The output of this process is the Software Requirements Specification (SRS).

##### 3.1.2 Requirements on the SRS

The Software Requirements Specification (SRS) contains all requirements from the DID which are relevant to the software as well as all other software specific requirements which arise due to the environment in which the software will operate. The SRS presents an external view of the software by providing a “black box” description of its behaviour.

Internal design details are not incorporated into the SRS so that software *design decisions* are not artificially constrained. The SRS does, however, document all necessary software design constraints and these may limit the solutions from which the developer has to choose.

The SRS shall:

- (a) Specify all requirements derived from the DID which are relevant to the software. This includes all functional, performance, safety, reliability, security, operability, and maintainability requirements.
- (b) Specify all requirements derived from sources other than the DID. Justification shall be provided for each of these requirements, along with identification of the source.
- (c) Describe the context of the software *subsystem*.

- (d) Identify those properties of the physical environment (that is, temperatures, pressures, display readings, etc.) that the software must monitor and/or control and represent them with mathematical variables (*monitored variables* and *controlled variables*).
- (e) Specify the characteristics of variables the software has access to (*input variables* and *output variables*), addressing such issues as types, formats, units, and valid ranges.
- (f) Specify the relationship between the monitored variables and input variables and between the output variables and controlled variables. This may be achieved by referencing appropriate hardware and/or *predeveloped software* documents.
- (g) Define, with the use of mathematical functions, the required behaviour of the controlled variables in terms of the monitored variables. The entire domain of the monitored variables shall be covered by these definitions. The mathematical functions shall use a notation that has consistent, unambiguous, precisely defined *syntax* and *semantics*.
- (h) Specify the response to all expected types of errors and *failure modes* identified by the *hazards analysis* of the system described in the DID. Indicate all cases for which error recovery must be attempted or for which fail-safe action must be taken.
- (i) Specify requirements for *fault tolerance* and *graceful degradation*.
- (j) Specify the timing tolerances and the accuracy requirements as the allowable deviation from the required behaviour of the controlled variables.
- (k) Identify all those requirements for which future changes are anticipated. This will provide a basis for the use of *information hiding* concepts in the design.
- (l) Identify any software implementation *design constraints*. This might include the processors on which the software is required to execute, and the predeveloped software required to be used.
- (m) Present requirements and design constraints only. It shall limit the range of valid solutions, but shall not specify a particular design.
- (n) Specify no requirements that are in conflict with each other.
- (o) Specify each unique requirement once to prevent inconsistent updates.
- (p) Identify each requirement uniquely so that it can be readily referenced by the SDD.
- (q) Reference *design notes* that document design decisions relevant to the software requirements.
- (r) Demonstrate explicitly the mapping and complete coverage of all relevant requirements and design constraints in the DID by such means as a *coverage matrix* or cross-reference.
- (s) Conform to the applicable *standards* and *procedures* in the SPH.

- (t) Use consistent terminology and definitions throughout the document.
- (u) Enable its intended audience to effectively determine the role each SRS component plays in the *subsystem*.

## 3.2 Design

### 3.2.1 Objectives

The objectives of the Design process are:

- (a) To produce a *software architecture* and detailed design that satisfies the requirements in the SRS.
- (b) To identify self-checks not derived from the SRS that enhance the *robustness* of the design to hardware *failures* or other system level *hazards*.

The inputs to this process are the SRS and SPH. The output is the Software Design Description (SDD).

### 3.2.2 Requirements on the SDD

The Software Design Description (SDD) is a representation of the software design.

#### 3.2.2.1 Architecture

The SDD shall:

- (a) Describe a design that meets all functional, performance, safety, reliability, security, operability, and maintainability requirements of the *subsystem* and all *design constraints* as described in the SRS.
- (b) Define a design that meets the requirements of the SRS to a level of detail that requires no further refinement of the *module* structure, module program functions, module *interfaces*, *data structures*, or *databases* in the code.
- (c) Explain how *performance requirements* are met by the design, using *worst case analysis*.
- (d) Describe the use of resources (for example, memory, processing time) by the design.
- (e) Explain how the *fault tolerance* and *graceful degradation* requirements are met by the design.
- (f) Define a method of scheduling computer resources that is primarily deterministic and predictable rather than dynamic.

- (g) Minimize the use of interrupts and *event driven software*.
- (h) Provide justification for all uses of interrupts and event driven software.
- (i) Use a logical control structure. Control must pass from the highest to successively more detailed levels. Control must always be returned to the calling *program*, with the exception of error/*exception handlers*.
- (j) Describe a design using *information hiding* concepts, as follows:
  1. Organize modules so that anticipated changes in the requirements can be implemented by only requiring changes to one or a small number of modules.
  2. Design those functions and data structures likely to change with interfaces to be insensitive to changes in individual functions.
  3. Describe a design which partitions data structure access, database access and I/O access from the application software by the use of access programs (globally accessible data must not be used).
  4. Partition functionality into programs to maximize the internal *cohesion* of programs and to minimize program *coupling*.
- (k) Describe a design in which each program has a single purpose.
- (l) Use specific criteria to limit program size.

### 3.2.2.2 Detailed Design

The SDD shall:

- (a) Provide the following information for each module:
  1. A unique identifying name.
  2. The purpose of the module.
  3. The data structure(s) maintained by the module.
  4. The list of programs contained in the module.
- (b) Provide the following information for each program:
  1. Unique identifying name.
  2. Program type (e.g., main program, subroutine, function, macro, interrupt handler).
  3. The purpose of the program (what requirement it satisfies).
  4. The behaviour of each program covering the entire domain of each program input.
  5. The subordinates which compose the program (to identify hierarchical structure to assist in tracing requirements).

6. Program dependencies (necessary relationships with other programs, such as: interactions, data flow, timing, order of execution, data sharing, etc.).
  7. Program interfaces (method of invocation or interruption, communication via parameters, database access, message passing, protocols, data formats, ranges of validity).
  8. Required resources (elements external to the design; for example, devices, software services, libraries, *operating system* services, processing resource requirements).
  9. *Program* processing details, if necessary, to ensure that a particular algorithm is used (refinement of function, algorithm, contingency responses to error cases, timing, sequence of events, priorities, processing steps).
  10. *Program* local data definitions (data type, initial value, use, valid range).
  11. The programming language (or subset of the programming language) to be used.
- (c) Identify processing steps within programs so that they can be uniquely referenced by the *source code*.
  - (d) Include or reference *design notes* that document *design decisions* relevant to the software design.
  - (e) Describe the strategy for managing interactions (e.g., serial input queues, analog input delays) between the software and the *input variables* and *output variables*.
  - (f) Define the types and handling of errors that are not specified in the SRS.
  - (g) Describe all elements of the software build process essential to the design.
  - (h) Define plausibility checking on the execution of programs to uncover errors associated with the frequency and/or order of program execution and, where applicable, the permissiveness of program execution.
  - (i) Describe the behaviour of each program using a notation that has precisely defined *syntax* and *semantics* so that the SDD can be systematically verified against the SRS and so that the code can be systematically verified against the SDD.

### 3.2.2.3 General

The SDD shall:

- (a) Factor in relevant experience from the previous systems that are identified in the SDP.
- (b) Identify and provide rationale for all behaviour specified in the SDD that is outside the scope of the SRS.
- (c) Identify and provide rationale for any requirement that cannot be met.

- (d) Demonstrate the mapping and complete coverage of all requirements and design constraints in the SRS by means such as a *coverage matrix* or a cross reference.
- (e) Conform to the applicable *standards, procedures* and *guidelines* in the SPH.
- (f) Use consistent terminology and definitions through out the document.
- (g) Avoid unnecessarily complex designs and design representations.
- (h) Enable its intended audience to effectively determine the role each component plays in the overall design.
- (i) Specify only programming languages (or subsets of each programming language) that have a defined syntax and semantics so that the results of any program constructs are unambiguous.

### **3.3 Code Implementation**

#### **3.3.1 Objectives**

The objectives of the *Code Implementation* process are:

- (a) To translate the SDD into *source code*.
- (b) To translate the source code into *executable code*.
- (c) To integrate the executable code.
- (d) To debug the executable code.
- (e) To create all required *databases* complete with appropriate initial data.

The inputs to this process are the SDD and SPH. The outputs are the source code, executable code, databases and build by-products such as memory maps and build logs.

#### **3.3.2 Requirements on the Code**

The source code is a complete and accurate translation of the design described in the SDD.

The source code shall:

- (a) Precisely implement the design as described in the SDD.
- (b) Compile without error.

- (c) Use *structured programming*. Specifically, the code shall:
  - 1. Not contain variable performance constructs (e.g., recursion).
  - 2. Not contain any *self-modifying code*.
  - 3. Not contain any potential infinite loops (with the possible exceptions of the mainline and fatal error handlers).
  - 4. Not contain *programs* with multiple *entry points* or multiple *exit points* (with the exception of fatal error handling).
  - 5. Not contain loops with multiple entrances.
  - 6. Be written so that each function of the program is a recognizable block of code.
- (d) Defend against detectable run-time errors. These errors include out-of-range array index values, division by zero, out-of-range variables, and stack overflow.
- (e) Take advantage of compiler-implemented type checking where this is feasible.
- (f) Contain or reference a revision history of all code modifications and the reason for them.
- (g) Have consistent and useful layout and format. Specifically, the code shall:
  - 1. Use consistent indentation to show nesting of structures.
  - 2. Use adequate white space to make structures stand out clearly.
  - 3. Use comment blocks to highlight the overall flow of program logic.
- (h) Have useful and clear comments using English language phrases. The comments shall:
  - 1. Accurately describe the functionality of the code in terms of the parameters being monitored and/or controlled; or alternatively, provide precise cross-reference to the SDD where the functionality is specified,
  - 2. Accurately describe the function of complex or non-obvious algorithms,
  - 3. Not just mimic the code, and
  - 4. Not make assumptions on the functionality of other modules such that the degree of *information hiding* is reduced.
- (i) Use consistently descriptive predefined *naming conventions*. The conventions must describe the criteria they attempt to meet and the rationale for these criteria.
- (j) Conform to the applicable *standards, procedures* and *guidelines* in the SPH.
- (k) Avoid implementation practices and techniques that are more difficult to verify than necessary.
- (l) Not rely on any defaults provided by the *programming language* used, unless they are part of the standard language definition.

- (m) Ensure that the accuracy requirements of all variables (as described in the SDD) are met by the implemented data types and algorithms.
- (n) Define all required databases complete with appropriate initial data.
- (o) Have data declarations that are consistent with their use.
- (p) Not contain any unreferenced or undefined symbols.
- (q) Refer to constants symbolically to facilitate change.
- (r) Include cross-references or *data dictionaries* showing variable and constant access by program.
- (s) Provide a cross-reference framework through which the code can be easily traced to the SDD.
- (t) Reference the *design notes* that document *design decisions* relevant to the code implementation.
- (u) Be written following guidelines that promote static logic. Examples of this include the use of constant maximum ranges on loops and the use of constants in branch tables.
- (v) Specify how the software is built. This shall include all script files that build the software.



## 4 VERIFICATION AND VALIDATION PROCESSES

This section describes the objectives, inputs and outputs of the *verification* and *validation* processes. These processes are:

- *Review*,
- *Systematic Verification*,
- *Testing*,
- *Hazards Analysis*, and
- *Reliability Qualification*.

### 4.1 Review

This section describes the objectives, inputs and outputs of the review processes. These processes are:

- *Requirements Review*,
- *Design Review*, and
- *Code Review*.

#### 4.1.1 Requirements Review

##### 4.1.1.1 Objectives

The objectives of the Requirements Review process are:

- (a) To identify ambiguities and incompleteness in the requirements specified in the DID.
- (b) To verify that the SRS meets the requirements of the DID.
- (c) To verify the justification for including any requirements and *design constraints* in the SRS which were not derived from the DID.
- (d) To verify that the SRS meets the requirements in the SPH.

The inputs to this process are the DID, SRS, and SPH. The output is the Requirements Review Report (RRR).

#### 4.1.1.2 Requirements on the RRR

The Requirements Review Report (RRR) shall:

- (a) Provide evidence that the review has covered all requirements and design constraints in the DID and SRS.
- (b) Provide evidence that the review has covered all requirements and design constraints appearing in the SRS which are not derived from the DID.
- (c) Provide evidence that the SRS has been reviewed against all applicable *standards* and *procedures* in the SPH.
- (d) Conform to the common requirements on reports (Section 4.5.3).

#### 4.1.2 Design Review

##### 4.1.2.1 Objectives

The objectives of the Design Review process are:

- (a) To verify that the *design decisions* are consistent with good *software engineering* practice (i.e., that the defined *software architecture* is appropriate to meet all requirements and constraints),
- (b) To verify that the SDD meets the intent of the requirements specified in the SRS and DID,
- (c) To check the justification for inclusion of any functionality outside the scope of the requirements to confirm that the resulting design is consistent with the intent of the requirements,
- (d) To verify that the SDD meets the requirements in the SPH.

The inputs to this process are the DID, SRS, SDD, and SPH. The output is the Design Review Report (DRR).

##### 4.1.2.2 Requirements on the DRR

The Design Review Report (DRR) shall:

- (a) Provide evidence that the review has covered all sections in the SDD, including all *modules, programs, data structures, and databases*.
- (b) Provide evidence that the SDD has been reviewed against the all applicable *standards, procedures* and *guidelines* in the SPH.
- (c) Conform to the common requirements on reports (Section 4.5.3).

### 4.1.3 Code Review

#### 4.1.3.1 Objectives

The objectives of the Code Review process are:

- (a) To verify that the *code implementation* decisions are consistent with good *software engineering* practice based on expert opinion.
- (b) To verify that the *code* meets the requirements in the SPH.

The inputs to this process are the SDD, *source code*, and SPH. The output is the Code Review Report (CRR).

#### 4.1.3.2 Requirements on the CRR

The Code Review Report (CRR) shall:

- (a) Provide evidence that the review has covered all *programs, data structures, and databases* in the source code.
- (b) Provide evidence that the source code has been reviewed against all applicable *standards, procedures* and *guidelines* in the SPH.
- (c) Conform to the common requirements on reports (Section 4.5.3).

## 4.2 Systematic Verification

This section describes the objectives, inputs and outputs of the systematic verification processes. These processes are:

- Systematic Design Verification, and
- Systematic Code Verification.

### 4.2.1 Systematic Design Verification

#### 4.2.1.1 Objectives

The objectives of the Systematic Design Verification process are:

- (a) To verify, using mathematical verification techniques or rigorous arguments, that for every output, the behaviour for that output as defined in the SDD is in compliance with the requirements for the behaviour imposed by the SRS.

- (b) To identify any functions outside the scope of the requirements specified in the SRS and to check that justification has been provided for their existence.

One of the side benefits of the Systematic Design Verification process is that it identifies ambiguities and incompleteness in the SRS.

The inputs to this process are the SRS, SDD and SPH. The output is the Design Verification Report (DVR).

#### **4.2.1.2 Requirements on the DVR**

The Design Verification Report (DVR) shall:

- (a) Provide evidence that the verification has covered all requirements in the SRS and all *programs, data structures* and *databases* in the SDD.
- (b) Provide evidence that the verification has covered justification for inclusion in the SDD of any functionality outside the scope of the requirements in the SRS.
- (c) Conform to the common requirements on reports (Section 4.5.3).

### **4.2.2 Systematic Code Verification**

#### **4.2.2.1 Objectives**

The objectives of the Systematic Code Verification process are:

- (a) To verify, using mathematical verification techniques or rigorous arguments, that the behaviour of outputs with respect to inputs in the code is the same as that specified by the SDD for the entire domain of the inputs.
- (b) To verify that there are no functions in the code which are not contained in the SDD.

One of the side benefits of Systematic Code Verification process is that it identifies ambiguities and incompleteness in the SDD.

The inputs to the process are the *source code*, SDD, and SPH. The output is the Code Verification Report (CVR).

#### **4.2.2.2 Requirements on the CVR**

The Code Verification Report (CVR) shall:

- (a) Provide evidence that the verification has covered all *programs, data structures*, and *databases* in the SDD and all of the *source code*.

- (b) Conform to the common requirements on reports (Section 4.5.3).

## 4.3 Testing

This section describes the objectives, inputs and outputs of the testing processes. These processes are:

- Unit Testing,
- Integration Testing, and
- Validation Testing.

The overall objectives of testing are to find *faults* in the *software* and problems in the requirements.

Each testing process includes:

- Preparing the Test Procedures,
- Verifying the Test Procedures,
- Carrying out the Test Procedures,
- Preparing the Test Report, and
- Verifying the Test Report.

### 4.3.1 Unit Testing

#### 4.3.1.1 Objectives

The objectives of the Unit Testing process are:

- (a) To find *faults* in the translation from *source code* to *executable code*.
- (b) To test that the executable code of each *program* and *module* behaves as specified in the SDD.
- (c) To test that the executable code of each program and module does not perform unintended functions.
- (d) To find faults in the program *interfaces*.

The inputs to this process are the SDD, source code, executable code, *databases*, test hardware, and SPH. The outputs are the Unit Test Procedures (UTP), including test stubs and test drivers, and the Unit Test Report (UTR).

#### 4.3.1.2 Requirements on the UTP

The Unit Test Procedures (UTP) shall:

- (a) Define tests that are executed on the *target* processor but possibly using test drivers and stubs to simulate other parts of the system.
- (b) Define a set of *test cases*, derived from the analysis of the SDD, to test that the executable code for each program behaves as specified in the SDD. This set of test cases shall be considered sufficient when it includes:
  1. All possible decision outcomes.
  2. Tests on each boundary and values on each side of each boundary for each input. The test values shall be determined by using *boundary value analysis* and *equivalence partitioning*.
  3. Tests based on postulated coding implementation errors.
- (c) Define a set of test cases, derived from the analysis of the code, to test that the executable code for each program behaves as specified in the SDD. This set of test cases shall be considered sufficient when it causes:
  1. Execution of every program statement.
  2. Execution of all possible decision outcomes.
  3. Execution of each loop with minimum, maximum, and at least one intermediate number of repetitions.
  4. A read and write to every memory location used for variable data.
  5. A read of every memory location used for constant data.
- (d) Define a sufficient number of tests to cause each interface to be exercised.
- (e) Require that the testing use the same *compiler* used to generate the final executable code.
- (f) Conform to the common requirements on test procedures (Section 4.5.1).

#### 4.3.1.3 Requirements on the UTR

The Unit Test Report (UTR) shall:

- (a) Conform to the common requirements on test reports (Section 4.5.2).

## 4.3.2 Integration Testing

### 4.3.2.1 Objectives

The objectives of the Integration Testing process are:

- (a) To find *faults* in the translation from *source code* to integrated system.
- (b) To test that the *executable code* meets the requirements specified in the SRS.
- (c) To find faults in the interfaces between the software, *target hardware* and *predeveloped software*.
- (d) To find *faults* in handling stress conditions, timing, fail-safe features, error conditions, and error recovery.

The inputs to this process are the SRS, SDD, source code, executable code, *databases*, target hardware, predeveloped software, and SPH. The outputs are the Integration Test Procedures (ITP), including test software, and the Integration Test Report (ITR).

### 4.3.2.2 Requirements on the ITP

The Integration Test Procedures (ITP) shall:

- (a) Require all tests to be done with *predeveloped software* and target hardware.
- (b) Define test cases to test each *functional requirement* in the SRS.
- (c) Define test cases to test each *performance requirement* in the SRS.
- (d) Identify all resources used by the software *subsystem* and define test cases to test the software *subsystem* under conditions that attempt to overload the identified resources to determine if the functional and *performance requirements* defined in the SRS are met.
- (e) Define test cases to find *faults* in the *interfaces* between:
  - 1. The modules of the *custom developed software*, and
  - 2. The custom developed software and its environment (including hardware and *predeveloped software*).
- (f) Define test cases to find *faults* in each *hardware configuration* and with each operational option.
- (g) Define test cases to find *faults* in the response to software, hardware, and data errors.
- (h) Define test cases that attempt to subvert any existing safety or *security* mechanisms.

- (i) Conform to the common requirements on test procedures (Section 4.5.1).

### 4.3.2.3 Requirements on the ITR

The Integration Test Report (ITR) shall:

- (a) Conform to the common requirements on test reports (Section 4.5.2).

## 4.3.3 Validation Testing

### 4.3.3.1 Objective

The objective of the Validation Testing process is:

- (a) To test that the *custom developed software*, as built, correctly addresses the capabilities needed by *users*.

The inputs to this process are the DID, ITP, custom developed software, *target hardware*, validation test hardware, *predeveloped software*, and SPH. Note that the definition of *test cases*, and the associated pass/fail criteria, are to be based on the involvement of the user or other person knowledgeable about the problem or objective to be addressed by the software, and the DID. The use of other documents is permitted, when necessary to set up the test scenarios, if clearly noted in the Validation Test Procedures (VTP). The outputs are the VTP and the Validation Test Report (VTR).

### 4.3.3.2 Requirements on the VTP

The Validation Test Procedures (VTP) shall:

- (a) Require all *tests* to be done on the entire software with the target hardware and predeveloped software.
- (b) Define test cases that exercise the capabilities needed by *users*.
- (c) Define test cases, using *dynamic simulation* of input signals, to demonstrate the *subsystem* capabilities under steady state conditions, changing input conditions, abnormal input conditions, and accident conditions requiring software subsystem action as indicated in the DID. The test cases shall cover all modes of operation.
- (d) Document coverage of all *functional* and *performance requirements* in the DID that are applicable to the software subsystem.
- (e) Conform to the common requirements on test procedures (Section 4.5.1).



### 4.3.3.3 Requirements on the VTR

The Validation Test Report (VTR) shall:

- (a) Conform to the common requirements on test reports (Section 4.5.2).

## 4.4 Other

### 4.4.1 Hazards Analysis

This section describes the objectives, inputs and outputs of the *hazards analysis* process.

#### 4.4.1.1 Objectives

The objectives of the Hazards Analysis process are:

- (a) To undertake a review of the *software* from the *safety* and *reliability* perspective (as orthogonal to the functional perspective) and thus identify any *failure modes* that can lead to an unsafe state and make recommendations for changes.
- (b) To determine sequences of inputs which could lead to the software causing an unsafe state and to make recommendations for changes.
- (c) To verify that the software required to handle system failure modes does so effectively.

Hazards analyses are performed at several points in the software design process. The inputs to the process are the DID, SRS, SDD, *source code*, *databases*, and software safety design principles contained in the SPH. The output is the Hazards Analysis Report (HAR).

#### 4.4.1.2 Requirements on the HAR

The Hazards Analysis Report (HAR) shall:

- (a) Identify software safety design principles that are proactively applied to mitigate potential *hazards*.
- (b) Identify the input conditions and subsystem *failures* that could lead to the software causing an unsafe state. For example:
  - 1. RAM variables whose corruption could lead to an unsafe subsystem failure.
  - 2. ROM constants whose corruption could lead to an unsafe subsystem failure.
  - 3. Code sequences that could lead to an unsafe subsystem failure.
- (c) Identify all self-checking software and its ability to eliminate the identified failure modes or reduce their likelihood of occurring.

- (d) Identify and recommend modifications to the software and the safety design principles that would eliminate the identified failure modes or reduce their likelihood of occurring.
- (e) Conform to the common requirements on reports (Section 4.5.3).

#### 4.4.2 Reliability Qualification

This section describes the objectives, inputs and outputs of the *reliability qualification* process.

##### 4.4.2.1 Objectives

The objective of Reliability Qualification process is:

- (a) To demonstrate that the reliability requirements are achieved for the *executable code* (integrated with the *target* hardware and any *predeveloped software*) with the degree of confidence necessary (as specified in the DID or SRS).

The inputs to Reliability Qualification are the DID, *code*, *target* hardware, and predeveloped software. The SRS may also be used to supplement the DID if it does not provide sufficiently detailed information. The outputs are the Reliability Qualification Procedure (RQP) and the Reliability Qualification Report (RQR).

##### 4.4.2.2 Requirements on the RQP

The Reliability Qualification Procedures (RQP) shall:

- (a) Require all tests to be done with the target hardware and predeveloped software.
- (b) Define the *reliability hypothesis*.
- (c) Explicitly identify the basis for the reliability hypothesis.
- (d) Define a sufficient number of tests representative of the software's usage profile, to provide *statistically valid* evidence showing that the probability of failure of the software is small enough that the *computer system* will meet its *reliability requirements* as described in the DID.
- (e) Define each test in terms of:
  1. The initial value of each input.
  2. The final value of each input.
  3. The length of time of the test or period.
  4. The time-related function describing how each input will vary over the period of the test.

- (f) Randomly select the initial input values from a distribution that is representative of the values seen when the system is not required to act.
- (g) Randomly select the final input values from a distribution that is representative of the values seen when the system is required to act.
- (h) For each input, randomly select the time-related function from a representative distribution that represents intermediate values assumed by the input as it progresses from the initial value to the final value. The function will include the effects of instrument response times, signal noise, and any other characteristics that are known about the input.
- (i) Randomly select the time period of the tests and ensure that all retained memory is initialized prior to running each test, so that the effects of the retained memory of the software do not invalidate the independence requirement between tests to ensure statistical validity. Justification shall be provided for any exceptions made to this requirement (for example, excessively long or infinite time periods not covered).
- (j) Provide justification supporting the chosen distributions showing that they are representative of the software's usage profile.
- (k) Identify all equipment (and its required calibration), tools, and support software required to produce the input and expected result data for the test cases.
- (l) Conform to the common requirements on test procedures (Section 4.5.1).

#### **4.4.2.3 Requirements on the RQR**

The Reliability Qualification Report (RQR) shall:

- (a) Conform to the common requirements on test reports (Section 4.5.2).

### **4.5 Common Verification and Validation Output Requirements**

This section defines the requirements common to all *verification* and *validation* outputs. Verification and validation outputs include review reports, systematic verification reports, test procedures and test reports.

#### **4.5.1 Test Procedures**

All test procedures shall:

- (a) Describe expected results of each *test case* from information provided in the input document(s) so that a pass/fail determination can be made as to the outcome of the *test*.
- (b) Define the acceptance criteria for comparing the expected behaviour and actual behaviour.

- (c) Describe *test* coverage by providing a cross-reference between the input document(s) and the *test procedures*.
- (d) Identify all equipment (and its required calibration), tools, and support software required to perform the test and provide adequate setup and test execution instructions so that the test could be performed by personnel who did not prepare the test procedure.
- (e) Describe or reference how the *executable code* to be tested is built.
- (f) Identify the items being tested.
- (g) Provide rationale for qualification of any support software used.
- (h) Identify the versions of relevant documents.
- (i) Identify and comply with the applicable SPH requirements.
- (j) Be verified to ensure that, for example, the:
  1. Required test coverage is provided.
  2. Procedures are specified such that the tests can be performed by personnel who did not prepare the test procedures.
  3. Tests are prepared in compliance with the SPH.
  4. Software referenced in the test procedures is qualified for use.

#### **4.5.2 Test Reports**

All test reports shall:

- (a) Identify the *test procedures*.
- (b) Include the comparison of actual *test* results with expected test results as defined in the referenced test procedures.
- (c) Include or reference the detailed test results.
- (d) Be verified to ensure that, for example, the:
  1. Tests were executed as specified in the test procedures.
  2. Test results were recorded and properly analyzed.
  3. Test report complies with the SPH.
- (e) Conform to the common requirements on reports (Section 4.5.3).

### 4.5.3 Reports

All reports shall:

- (a) Summarize the activities performed and the methods and tools used.
- (b) Identify and summarize the discrepancies.
- (c) Summarize the positive findings.
- (d) Describe the conclusions and recommendations.
- (e) Identify the versions of relevant documents.
- (f) Identify all participants.
- (g) Identify the start and end dates of the *verification* or *validation* activity.
- (h) Identify and comply with the applicable SPH requirements.
- (i) Identify or reference corrective actions resulting from the verification or validation process.
- (j) Identify the actual equipment, tools and support software used.

## 5 SUPPORT PROCESSES

This section describes the objectives, inputs and outputs of the *support* processes. These processes are:

- Planning,
- Configuration Management, and
- Training.

### 5.1 Objectives

#### 5.1.1 Planning

The objectives of the Planning process are:

- (a) To define all processes in the software *life cycle* for the *software engineering project* relating to how and when they are to be done and who is to do them. A specific model for the software life cycle is adopted as a focus for planning and for the activities mapped to it.
- (b) To define all *development, verification, and support* processes at the start of the project.
- (c) To identify and document all project-specific *standards and procedures*.
- (d) To define an organization that meets the independence requirements for personnel participating in the various software engineering processes to ensure that work is carried out objectively and effectively.
- (e) To establish estimates, schedules, budgets, and resource requirements. These include the effort required to prepare the infrastructure including the development of procedures, support tools, and facilities.

The inputs to this process are the DID and existing standards and procedures. The outputs are the Software Development Plan (SDP) and the Standards and Procedures Handbook (SPH). The SDP should not duplicate information contained in the overall project management documentation (i.e., the Project Execution Plan (PEP) which should follow adopted company standards), but rather reference it.

### 5.1.2 Configuration Management

The objectives of the *Configuration Management* process are:

- (a) To ensure that the correct version of each *configuration item* is being used at any point in time, by identifying the *configuration* of the software and *target* systems at discrete points in time.
- (b) To control all changes made to the software.
- (c) To ensure that changed configuration items are developed and verified with the same rigour as applied to the original items.
- (d) To provide an on-going analysis of encountered errors to be used as input for continuous improvements to the *standards* and *procedures*.

The inputs to this process include all items whose *configuration* is managed (this includes, as a minimum, all *project* documents listed in Table 2-1, including the SDP and SPH) and all software *change requests*. The outputs of this process include the controlled version of all project documents, software *releases*, approved change requests, and software configuration status reports (i.e., Baseline Release Package Indices (BRPIs)).

### 5.1.3 Training

The objectives of the Training process are to ensure that the personnel deployed on the *software engineering project*:

- (a) Have the necessary skills and knowledge to perform their tasks.
- (b) Are completely conversant with all required *procedures* and *standards* that affect their work.

The inputs to this process are the SDP, SPH, and *skills inventories*. The outputs are training initiatives and individual training *records* that are used to update the skills inventories and to improve training effectiveness.

## 5.2 Requirements on the SDP

The Software Development Plan (SDP) provides the comprehensive plan for the management of the software engineering process. The SDP consists of the:

- Project Plan,
- Documentation Plan,
- Configuration Management Plan, and
- Training Plan.

The SDP shall be revised when there are major changes to either the software scope of work or to the software team organizational structure.

### 5.2.1 Project Plan

The project plan portion of the SDP describes the scope of the software engineering effort (including *development*, *verification* and *support*), the organization and responsibilities, and the *software engineering life cycle* model. It identifies key *milestones* and dates based on other detailed budget and schedule documentation used for monitoring the work effort.

#### 5.2.1.1 Planning

The project plan portion of the SDP shall:

- (a) Identify the DID.
- (b) Adopt a specific software life cycle model as a focus for planning to ensure a systematic software engineering process is followed over the entire life of the *software*. It shall treat each of development and maintenance as an integral, continuous, interactive and iterative process (as described in Section 2).
- (c) Describe unambiguously the scope of work for the software *project*. It shall describe the product and process goals in terms of the safety, functionality, reliability, maintainability, and reviewability requirements specified in the DID.
- (d) Identify any key design and implementation issues and preliminary studies, *simulation modelling*, and/or the *prototyping* required to resolve them.
- (e) Mandate the project-specific *standards* and *procedures* to be followed and contained in the SPH.
- (f) Require that the SDP be revised when there are major changes to either the software scope of work or to the organizational structure.



- (g) Identify any previously developed systems from which developers may obtain relevant design and operating experience that can be factored into their current design.

### 5.2.1.2 Scheduling

The project plan portion of the SDP shall:

- (a) Partition the software effort into uniquely identifiable *development*, *verification*, *validation* and *support* processes with well-defined inputs and outputs.
- (b) Identify appropriate verification activities for each output. These activities may consist of one or more of the following: *animation*, supervisory review, *walkthrough*, *review* and comment, *testing*, and *prototyping*. Also, mandate that a *formal design review* be carried out for requirements verification and that a formal design review be carried out for design verification.
- (c) Define the overall software development effort and cost. Document this estimate and the rationale used to determine the estimate for future use and review. As a minimum, the cost estimate shall be based on:
  1. An estimate of software size derived from a preliminary design breakdown of software functionality into manageable software *subsystems* or modules.
  2. A refined breakdown of process activities.
  3. The effort required to achieve software *reliability requirements*.
  4. Anticipated software complexity.
  5. Hardware constraints (memory, CPU loading and I/O capacity).
  6. Personnel and training constraints (availability, experience, and skills of personnel, training required, duration, and availability).
  7. Project constraints (system budget and schedule).
  8. Contingency (delivery delays, requirements volatility, staff turnover, design infeasibility, environment constraints).
  9. Available historical productivity data.
  10. Installation, commissioning and documentation close-out processes.
- (d) Provide scheduling information for managing the software project. The following scheduling information is required:
  1. Identification of key milestones and *hold points* beyond which no further activities can be started until the necessary completion criteria are successfully met for the prerequisite activities.
  2. Identification of integration milestones on the schedule and their relationships with the delivery of *predeveloped software*, hardware, and documentation.

- (e) Require that, for each release of the software intended for use in actual operation, verification activities for a particular software development process shall not start until all verification activities for the preceding development process have been completed.

**5.2.1.3 Resources**

The project plan portion of the SDP shall:

- (a) Describe the software project organizational structure and *interfaces* to ensure that the various distinct software engineering processes are carried out objectively and effectively, and reduce the chance of making common mistakes in development and verification. To achieve this, the following independence rules shall be met:
  1. Define the following independent roles: Developer, Verifier, and Validator.
  2. Responsibility for the software engineering processes shall be assigned to the defined roles as described in Table 5.2.1.3.
  3. The supervision of the developers shall be different from the supervision of the verifiers and validators.

<b>Table 5.2.1.3 - Independence Requirements</b>	
<b>Process</b>	<b>Role</b>
Requirements Definition	Any
Design	Developer
Code Implementation	Developer
Requirements Review	Any <sup>**</sup>
Design Review	Verifier <sup>*</sup>
Systematic Design Verification	Verifier <sup>*</sup>
Code Review	Verifier <sup>*</sup>
Systematic Code Verification	Verifier <sup>*</sup>
Hazards Analysis	Verifier <sup>*</sup> or Validator
Unit Testing	Verifier <sup>*</sup>
Integration Testing	Verifier <sup>*</sup>
Validation Testing	Validator
Reliability Qualification	Verifier <sup>*</sup> or Validator
Planning	Any
Configuration Management	Any
Training	Any

\* The verifier has primary responsibility for the process but developer personnel may also be involved provided they do not verify their own work.

\*\* Exclusive of those who participated in preparing the Software Requirements Specification (that is, if a developer prepares the SRS, a verifier or validator should perform the requirements review).

- (b) Specify the approach and *methodologies* to be used for the development, verification and support processes.

- (c) Specify facilities, tools, and aids to be used during development, verification and support.
- (d) Identify personnel required to maintain and manage the facilities, tools and aids.
- (e) Include a policy statement that the SDP shall be followed by all personnel who are responsible for the production of an output required by the SDP.
- (f) Include a policy statement that the personnel who produce each output required by the SDP have primary responsibility for the output's quality.
- (g) Identify the qualification requirements for facilities, *software tools* and predeveloped software for the *target* system.

### 5.2.2 Documentation Plan

The documentation plan portion of the SDP provides an index and summary descriptions of all project documentation related to software. It also identifies responsibility for producing, reviewing, approving, and issuing documents. At the very least, the plan covers all outputs identified in Section 2.

The documentation plan portion of the SDP shall:

- (a) List and briefly describe all project documents and *records* that are a result of the software project covered by the SDP.
- (b) Distinguish between *deliverable* end-user documentation and *non-deliverable* documentation.
- (c) Contain or reference an index of all deliverable software, test software, and support software.
- (d) List all deliverable documents obtained from external suppliers as they become known.
- (e) Require the use of *design notes* to document the experience, options, trade-offs, decisions, rationale, design preferences and other issues that have been incorporated into the design.
- (f) Identify responsibility for producing, reviewing, approving, filing, and issuing documents, including design notes.
- (g) Specify or reference document distribution lists.
- (h) Identify facilities, tools and aids used to produce, manage, and issue documentation.

### 5.2.3 Configuration Management Plan

The *configuration management* plan portion of the SDP identifies the configuration management activities and the associated support tools and facilities.

The configuration management plan portion of the SDP shall:

- (a) Uniquely identify all categories of *configuration items* that form a developmental baseline for software configuration management. These categories include application software (source, object, listing, load images), software media, all *deliverable* and *non-deliverable* documents, support tools (*compilers*, linkers, *operating systems*), data, test software, command files or scripts, and *predeveloped software*.
- (b) Specify when and under what conditions all configuration items come under *change control*. As a minimum, every document and piece of *source code* shall go under change control before it is given to an independent party for *verification* purposes.
- (c) Require that each change be implemented starting at the highest affected output and proceeding to subsequent outputs (for example, DID to SRS to SDD to source code).
- (d) Require that each change undergo the same degree of verification as provided for the original outputs.
- (e) Require that the extent of the verification of each change be determined by an analysis of the scope of the change.
- (f) Define a mechanism to periodically review, classify, and adjudicate submitted *change requests*. The mechanism must allow for participation by members representing each organizational unit that is responsible for one of the project outputs identified in Section 2.
- (g) Define a mechanism for identifying *baselines*.
- (h) Identify a centralized support library to provide storage for, and controlled access to, software and documentation in both human-readable and machine-readable forms as well as *records* to support audit and review of the software *configuration*.
- (i) Identify a mechanism for analysis of the errors detected during *verification* as an aid to on-going improvement of *standards* and *procedures* in the SPH and the various *test procedures*.
- (j) Identify how to integrate predeveloped software and documentation into the adopted configuration management system.

#### 5.2.4 Training Plan

The training plan portion of the SDP identifies the types of training, the training organization, training tools, resources, and documents.

The training plan portion of the SDP shall:

- (a) Establish the minimum *qualification* criteria for the positions performing each set of tasks consistent with accepted engineering practice for systems of this nature.
- (b) Specifically tailor training to job positions.
- (c) Provide for the evaluation of training effectiveness.
- (d) Identify orientation training for new personnel with specific emphasis on project *standards* and *procedures* in the SPH.
- (e) Identify training required when changes occur (for example, changes to requirements, standards and procedures).
- (f) Identify requirements for training courses and materials with respect to the use of *software tools* and *methodologies*.
- (g) Identify training facilities, tools, responsibility for training, and the contents of the training *records*.

### 5.3 Requirements on the SPH

The Standards and Procedures Handbook (SPH) defines the *standards* and *procedures* that must be defined and used. The SPH consists of:

- Process Standards and Procedures.
- Documentation Standards and Procedures.
- Configuration Management Standards and Procedures.
- Training Standards and Procedures.

#### 5.3.1 Process Standards and Procedures

The process standards and procedures shall:

- (a) Identify all pertinent software safety design principles.

- (b) Specify standards, procedures, *methodologies*, and *guidelines* to ensure that all *development* processes and *development outputs* specified in this standard meet the objectives and requirements specified in this standard.
- (c) Identify all pertinent design and coding guidelines.
- (d) Specify standards, procedures, methodologies, and guidelines to ensure that all *verification* processes and verification outputs specified in this standard meet the objectives and requirements specified in this standard. (Note that, for testing processes, these guidelines and methods are not to be confused with the detailed *test procedures* that describe specifically how to test a particular product. Rather, these guidelines and methods define the overall test method used and the content and format of the test procedures).
- (e) Define checklists and guidelines for requirements *review*, design review, and code review to achieve a known and consistent level of coverage.
- (f) Define procedures for a *formal design review* of the SRS.
- (g) Define procedures for a formal design review of the SDD.
- (h) For systematic design verification, define transformation rules that convert either the SRS or the SDD to a notation that is directly comparable with the other.
- (i) For systematic code verification, define transformation rules that convert either the SDD or the code to a notation that is directly comparable with the other.
- (j) Specify criteria and guidelines for the generation of *test cases* to achieve a known and consistent level of coverage.
- (k) Identify procedures for using tools and facilities in the development, verification and *support* processes.
- (l) Identify or reference pertinent documents that identify all verification tools, techniques, checklists, procedures, and *databases*, their proper use and current versions.
- (m) Define procedures for ensuring that the part of the software not affected by change will still work correctly following a change implementation (*regression testing*).

### 5.3.2 Documentation Standards and Procedures

The documentation standards and procedures shall:

- (a) Identify a standard document style and format to be used for all documents. The style and format *guidelines* shall be chosen to make the documents consistent, understandable, reviewable, and maintainable. The guidelines shall address:
  1. Fonts and attributes.

2. Page and paragraph layout (indentation, margins, spacing, etc).
  3. Headers and footers.
  4. Cover page.
  5. Revision history (detailing all changes to the document).
  6. Table of contents.
  7. Labelling of figures and tables.
  8. Page identification.
  9. Section identification.
  10. Cross-referencing between documents.
  11. Glossary.
  12. Index.
  13. References.
  14. Date, signature, and a (controlled) document number.
- (b) Define a glossary to ensure common understanding and consistent usage of project-specific and industry accepted terminology. The use of novel terminology and definitions shall be minimized.

### 5.3.3 Configuration Management Standards and Procedures

The configuration management standards and procedures shall:

- (a) Specify *guidelines* to handle and document exception cases or problems arising from errors or exceptions which are detected during software *development* and *verification* (for example, non-conformances to the SDP or SPH, errors found during testing activities, changes arising during installation and operation, tests deferred to a later testing phase).
- (b) Identify a *procedure* for ensuring that all personnel are working from the appropriate version of the documentation.
- (c) Identify procedures for unique identification of:
  1. Software changes.
  2. Software and *software components*.
  3. *Release*, version, and update designations.
  4. Documentation.
  5. Media.

- (d) Define procedures for issuing, analyzing, estimating the impact of, classifying, approving or disapproving, distributing, scheduling implementation of and tracking software *change requests* to completion. These procedures shall require that the activities necessary to verify the change are identified and justified.
- (e) Define procedures for identifying all implemented software change requests and all *configuration items* and revision numbers for all software *releases*.
- (f) Identify procedures for support library operation (i.e., *security*, disaster recovery, archive maintenance, protection, access control, retention period, change history, withdrawal, submission, issue, and approval).
- (g) Define procedures for change request documentation, error analysis summary reports and *configuration management* status reports in order to identify the most current *configuration* and to provide a traceable history of software changes.
- (h) Define procedures for approving and issuing software releases to the *user* and for confirming and documenting that the software delivered is identical to that approved for release.

#### **5.3.4 Training Standards and Procedures**

Training standards and procedures shall:

- (a) Specify how to maintain training *records* and *skills inventories*.



## 6 GLOSSARY

This glossary defines terminology that is not explicitly defined within this standard.

The definitions used in this glossary come from a variety of sources. Those definitions which are either taken directly or paraphrased from a source are referenced. The use of a source does not imply an exact quote from it, but rather, acknowledges the originating definition.

**animation** A software requirement analysis technique that involves walking through a specific portion of the specification by using certain inputs to determine its behaviour under different circumstances. [3]

**assembler** A tool for translating *assembly language* code into a form which is executable on a *computer*.

**assembly language** A representation of *computer* instructions and data that usually has a one-to-one correspondence with *machine language*. Assembly language is more representative of the application program than machine language. [3]

**baseline** A set of *configuration items* which serves as the basis for further *development*, and can be changed only through *change control procedures*. [3]

**Baseline Release Package Index** A single source which identifies every *configuration item* in one or more *baselines*.

**boundary value analysis** A technique in which test data are chosen to lie along "boundaries" or extremes of input domain (or output range) partitions, *data structures*, program parameters, and so forth. [3]

**BRPI** Abbreviation for *Baseline Release Package Index*.

**category I software** See *safety critical software*.

**change control** See *configuration control*. [5]

**change request** A formal written request that a modification or correction be made to *configuration items*.

**code** 1. Computer instructions and data definitions expressed in a programming language or in a form output by an assembler, *compiler*, or other *translator*. See *source code*, *executable code*. 2. To express a computer program in a programming language. 3. A character or bit pattern that is assigned a particular meaning; for example, a status code. [5]

**code implementation** The phase of *software engineering* that involves the conversion of a software design into *source code* and *executable code*. [3]

**cohesion** The degree to which the tasks performed by a single *program* are functionally related. Cohesion is a measure of the strength of association of the elements within a program. [3] Contrast with *coupling*.

**compiler** A tool for translating computer language *source code* into a form that is executable on a *computer system*.

**completeness** A *quality attribute* that refers to the extent to which all of the required behaviour and *design constraints* are present and fully developed in the *development outputs*.

**computer** A functional programmable unit that consists of one or more associated processing units and peripheral equipment that is controlled by internally stored *executable code* and that can perform substantial computation, including numerous arithmetic operations or logic operations, without human intervention. [3]

**computer system** A system composed of computer(s), peripheral equipment such as disks, printers and terminals, and the software necessary to make them operate together. [3]

**configuration** 1. The arrangement of a *computer system* or component as defined by the number, nature, and interconnections of its constituent parts. 2. In *configuration management*, the functional and physical characteristics of hardware or software as set forth in technical documentation or achieved in a product. [5]

**configuration control** An element of *configuration management*, consisting of the evaluation, coordination, approval or disapproval, and implementation of changes to *configuration items* after formal establishment of the configuration identification. [5]

**configuration item** An aggregation of hardware, *software*, or both, that is designated for *configuration management* and treated as a single entity in the configuration management process. [5]

**configuration management** The process of identifying *configuration items* and *baselines*, controlling changes, and maintaining the integrity and *traceability* of the *configuration*. [6]

**consistency** A *quality attribute* that refers to the extent to which the *development outputs* contain uniform notations, terminology, comments, symbology, and implementation techniques.

**controlled variable** A mathematical variable used to model a property of the environment or *computer system* that the *system* is intended to control. [7] See *four variable model*.

**control flow** 1. The sequence of execution of statements in a *program*. [5] Contrast with *data flow*. 2. The mechanism for one portion of a system to trigger activity in another portion of a system.

**correctness** A *quality attribute* that refers to the ability of the *development outputs* to describe or produce the specified outputs when given the specified inputs, and the extent to which they match or satisfy the requirements in the development inputs.

**coupling** A measure of the interdependence among *modules*. Coupling is the amount of information shared between two modules. [3] Contrast with *cohesion*.

**coverage matrix** A method of indicating which portion of a *development output* satisfies the requirements of a development input to assist *traceability* and *completeness* checks.

**custom developed software** *Software* developed to fulfil the requirements of a specific application from a user requirements specification.

**CRR** Abbreviation for Code Review Report.

**CVR** Abbreviation for Code Verification Report.

**data dictionary** A collection of names of all data items used in the *software*, together with relevant properties of those items; for example, length of data item and representation. [3]

**data flow** The sequence in which data transfer, use, and transformation are performed during the execution of a computer *program*. [3] Contrast with *control flow*.

**data structure** A representation of the logical relationship among individual data elements. [8]

**database** An organized collection of information that is accessed via *software*. [8]

**debugging** The activity of detecting errors, then determining the exact nature and location of each error and either fixing or repairing the error. [9]

**deliverable** Contracted *software item* to be delivered. [6]

**design constraint** A requirement that impacts or constrains the design. Examples of design constraints are physical requirements, software development *standards*, and software quality assurance standards. [3]

**design decision** The selection from a collection of alternate design solutions based on analysis.

**Design Input Documentation** The documentation required as input to the *software engineering* process.

**design notes** A means of recording the rationale, experience, options, and trade-offs behind design decisions.

**development** The process by which user needs are translated into a software product. The process involves translating user needs into software requirements, transforming the software requirements into design and implementing the code. Sometimes includes the *verification*, *validation* and *support* processes. [5]

**development output** A product of a *development* process. Examples of development outputs include requirements, design documents and code.

**DID** Abbreviation for *Design Input Documentation*.

**DRR** Abbreviation for Design Review Report.

**DVR** Abbreviation for Design Verification Report.

**dynamic simulation** Imitation of the changing behaviour of one or more inputs to a subsystem.

**efficiency** A *quality objective* that requires that the *system* achieve an acceptable level of performance within the constraints of hardware resources (e.g., memory, CPU speed).

**entry point** The location in a *program* at which it will begin execution.

**equivalence partitioning** A *testing* technique which relies on looking at the set of valid inputs specified for a *program* - its domain - and dividing it up into classes of data that should, according to the specification, be treated identically. These equivalence classes will not overlap and one set of *test data* is then chosen to represent each equivalence class. [10]

**error** 1. The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. For example, a difference of 30 metres between a computed result and a correct result. 2. An incorrect step, process, or data definition. For example, an incorrect instruction in a computer *program*. 3. An incorrect result. For example, a computed result of 12 when the correct result is 10. 4. A human action that produces an incorrect result. For example, an incorrect action on the part of a programmer or operator. Note: While all four definitions are commonly used, one distinction assigns definition 1 to the word “error,” definition 2 to the word “fault,” definition 3 to the word “failure,” definition 4 to the word “mistake.” [5] See *fault, failure*.

**event driven software** *Software* which is activated and responds to events which occur at unpredictable instances of time.

**exception handler** A *program* which is responsible for handling abnormal conditions. This may include reporting and recording the occurrence of the condition and performing the necessary recovery, mitigating or halting activities.

**executable code** A *program* in a language that can be directly executed by a *computer*. [3]

**exit point** The location in a *program* at which it can terminate execution.

**failure** The inability of a system or component to perform its required functions within specified performance requirements. [5] See *error, fault*.

**failure mode** A way in which a *system* can fail to meet its requirements.

**fault** 1. A defect in a hardware device or component; for example, a short circuit or broken wire. 2. An incorrect step, process, or data definition. For example, an incorrect instruction in a computer program. [5] See *error, failure*.

**fault tolerance** The ability of a system or component to continue normal operation despite the presence of hardware or software *faults*. [5]

**formal design review** A *review*, at specific *development* stages, of the technical aspects of *development outputs* to ensure compliance with requirements, functional specifications, *standards*, and other requirements appropriate to that stage. [6]

**four variable model** A mathematical model of a *computer system* and its requirements clearly defining the computer system and its boundaries. The four sets of variables in this model that define the boundary are the *monitored variables*, *input variables*, *output variables* and *controlled variables*.

**function** An association between two sets in which each element of one set has one assigned element in the other set. [11]

**functional requirement** A requirement that specifies the behaviour of a *system* or system component in a given circumstance.

**functionality** A *quality objective* that requires that the *system* implement all required behaviour and meet the *performance requirements*.

**graceful degradation** Stepwise reduction of capabilities in response to detected *failures* while essential behaviour is maintained. [12]

**guideline** A documented set of rules, heuristics or steps that are used to guide an activity or task. Guidelines usually usually allow judgment and flexibility. [3]

**HAR** Abbreviation for Hazards Analysis Report.

**hardware configuration** An arrangement of hardware components.

**hazard** A condition that can lead to an accident. [13]

**hazards analysis** An iterative process composed of identification and evaluation of *hazards* associated with a *computer system*, to enable them to be eliminated or, if that is not practical, to assist in the reduction of the associated risks to an acceptable level. [13]

**hold points** Those points in a project beyond which the work can not proceed before meeting specific conditions.

**information hiding** A software design technique in which each *module*'s interface reveals as little as possible about the module's inner workings. Other modules are forbidden to use information about the module that is not in its interface specification. [3]

**input variable** A mathematical variable that corresponds to a *monitored variable* that the *software* can directly access. [7] See *four variable model*.

**inspection** A semiformal to formal evaluation technique in which software requirements, design, or code are examined in detail by a person or group other than the originator to detect *faults*, violations of *development* standards, and other problems. The review members are peers (equals) of the developer. Traditional error data is collected during inspections for later analysis and to assist in future inspections. Sometimes called a *walkthrough* or peer review. [3]

**interface** 1. A shared boundary. An interface might be a hardware component to link two devices or it might be a portion of storage or registers accessed by two or more *programs*. [3] 2. To interact or communicate with another *subsystem* component or organizational entity. [3]

**ITP** Abbreviation for Integration Test Procedures.

**ITR** Abbreviation for Integration Test Report.

**life cycle** All the steps or phases an item passes through during its useful life. [3]

**machine language** The instructions that are directly executable by a *computer*. [3]

**maintainability** A *quality objective* that requires that the *system* be structured so that those items most likely to require modification can be changed reliably and efficiently. This quality objective also requires the rationale for *design decisions* be evident to a third party.

**methodology** A general approach to solving an engineering problem. [3]

**milestone** A scheduled event that is used to measure progress. [3]

**modifiability** A *quality attribute* that refers to the characteristics of the *development outputs* which facilitate the incorporation of changes.

**modularity** A *quality attribute* that refers to the extent to which the *development output* is composed of discrete components such that a change to one component has minimal impact on the others.

**module** A collection of *data structures* and *programs* that can act on those data structures. The data structures can only be accessed externally through access programs provided by the module.

**monitored variable** A mathematical variable used to model a property of the environment or *computer system* that the *user* wants the computer system to measure. [7] See *four variable model*.

**naming convention** A *guideline* for the naming of various entities within a software system.

**non-deliverable** A document or item which, although useful, is not required to be delivered.

**notation** A physical, graphical, or textual means of describing a requirement, design, or code. [3]

**operability** A metric indicating the ease of operation of a program. [8]

**operating system** *Software* which provides application processes with controlled access to computer resources. As a minimum, it provides orderly system startup and termination functions and *exception handling* capability.

**output variable** A mathematical variable that corresponds to a *controlled variable*, that the *software* can directly access. [7] See *four variable model*.

**performance requirement** A requirement specifying a performance characteristic; for example, speed, accuracy, or frequency. [3]

**PEP** Abbreviation for *Project Execution Plan*.

**portability** A *quality objective* that requires that the *software* be capable of being transferred to different environments to the degree required by users.

**predeveloped software** *Software* which has been produced prior to the issuing of a contract or purchase order, or to satisfy a general market need. [3]

**predictability** A *quality attribute* that refers to the extent to which the behaviour and performance of a *system* are deterministic for a specified set of inputs.

**procedure** A written set of rules to be followed in the performance of a task.

**procedural language** A textual computer language in which the *user* specifies a set of executable operations and the sequence in which they are to be performed (for example, FORTRAN, *assembler*, Pascal).

**program** A uniquely identifiable sequence of instructions and data which is part of a *module* (e.g., main program, subroutine, or macro).

**project** An activity characterized by a start date, specific objectives and constraints, established responsibilities, a budget and schedule, and a completion date. (If the objective of the project is to develop *software*, then it is sometimes called a software development or *software engineering* project). [3]

**Project Execution Plan** A strategy document that provides a comprehensive presentation of the issues involved in the planning, design, procurement, construction, and start-up of a project.

**prototyping** A hardware or software *development* technique in which a preliminary version of part or all of the hardware or software is developed to permit user feedback, determine feasibility, or investigate timing or other issues in support of the development process. [5]

**qualification** The process of determining whether a product meets its stated quality requirements.

**quality** The totality of features and characteristics of a product or service that bears on its ability to satisfy given needs. [3]

**quality attributes** The features and characteristics of a *development output* that determine its ability to satisfy requirements. [6]

**quality objective** A general statement about the relative importance of the *quality attributes*.

**real-time** Pertaining to a system or mode of operation in which computation is performed during the actual time that an external process occurs, in order that the computation results can be used to control, monitor, or respond in a timely manner to the external processes. [3]

**record** A set of related items treated as a unit. For example, in stock control, the data for each invoice could constitute one record. [3]

**regression testing** The rerunning of *test cases* that a *program* has previously executed correctly in order to detect errors created during software correction or modification activities. [3]

**release** A planned output consisting of a verified and validated version of a set of *configuration items*.

**reliability** A *quality objective* that requires that the *system* perform its required behaviour such that the probability of it successfully performing that behaviour is consistent with the *reliability requirements* identified.

**reliability hypothesis** The collection of assumptions used to develop the statistical (e.g., first order Markov) model which allows an upper limit failure probability with associated confidence level to be assigned to a software system based on a number of tests. The major assumptions include: 1) there are only successful test trials, 2) all tests are independent, 3) each test has an equal failure probability, 4) all failures are detected during testing, and 5) the test usage profile is an accurate representation of the expected in-situ demand usage profile.

**reliability qualification** A *testing methodology* in which a reliability hypothesis is demonstrated to the degree of confidence required, using randomized input values.

**reliability requirements** A *requirement* specifying the probability of not encountering a sequence of inputs which lead to a *failure*. [14]

**requirement** 1. A capability needed by a *user* to solve a problem or to achieve an objective.  
2. A capability that must be met or possessed by a *system* or system component to satisfy a contract, *standard*, specification, or other formally imposed document. 3. The set of all requirements that form the basis for subsequent *development* of the system or system component. [3]

**review** 1. The examination of an output for the purpose of detection and remedy of deficiencies and for identification of potential improvements in performance, safety or economy. 2. A formal meeting at which a product or document is presented to interested parties for comment and approval. It can also be a review of the management and technical progress of a project. [3]

**reviewability** A *quality objective* that requires that the *system* be developed and documented so that it can be systematically inspected by a third party for conformance to requirements.

**robustness** A *quality attribute* that refers to the extent to which the *development outputs* require and implement the ability to continue to perform despite some *subsystem failure*.



**RRR** Abbreviation for Requirements Review Report.

**safety** A *quality objective* that requires that the *system* function in accordance with its requirements, in a consistent and predictable manner, under all conditions. When the system can no longer perform its required role, this quality objective requires that it act to maintain the equipment and processes it controls in a safe state in all situations.

**safety critical software** Safety critical software (category I) is *software* that is critical to nuclear safety. Failure of software in this category can result in either a system with a high safety related *reliability requirement* (such as a *special safety system*) not meeting its minimum *performance requirements*, or a serious initiating event in a process system (i.e., an event for which the initiating-event frequency limit is very low). [4]

**safety requirements** The requirements which are concerned with the prevention or mitigation of the effects of *failures* which could lead to an unsafe system state.

**SCV** Abbreviation for Systematic Code Verification.

**SDD** Abbreviation for Software Design Description.

**SDP** Abbreviation for Software Development Plan.

**SDV** Abbreviation for Systematic Design Verification.

**security** The establishment and application of safeguards to protect data, software and computer hardware from accidental or malicious modification, destruction, or disclosure. [3]

**self-modifying code** *Code* which, by design, achieves its function by overwriting itself.

**semantics** The meaning of a sentence. [3] Contrast with *syntax*.

**simulation modelling** 1. Physical or mathematical representation of a system to expedite the evaluation of selected system parameters. Simulation is used to explore the effects that alternative system characteristics will have on system performance without actually producing and testing each alternative system. 2. Use of an executable model to represent the behaviour of an object. During *testing*, the computational hardware, the external environment, and even code segments may be simulated. [3]

**skills inventories** A list of personnel available for a project with a list of each person's skills which are pertinent to the project.

**software** A set of *programs*, associated data, *procedures*, rules, documentation, and materials concerned with the *development*, use, operation, and maintenance of a *computer system*. [6]

**software architecture** The set of major components of the *software* and their interrelationships, the main algorithms that these components employ, and the major data structures. [11]

**software component** A *software item* that is placed under *configuration control*. [6]

**software engineering** 1. The practical application of computer science, management, and other sciences to the analysis, design, construction, and maintenance of *software* and associated documentation. 2. An engineering science that applies the concept of analysis, design, coding, testing, documentation, and management to successful completion of large custom developed software. 3. The systematic application of methods, tools, and techniques to achieve a stated requirement or objective for an effective and efficient software system. 4. The application of scientific principles to:

- the early transformation of a problem into a working software solution, and
- subsequent maintenance of that software until the end of its useful life. [3]

**software item** A functionally or logically distinct part of the *software*. [6]

**software tools** Software tools are computer programs used in the *development*, testing, analysis, or maintenance of a program or its documentation. Examples include compilers, editors, generators, drivers, analyzers. [5]

**source code** Computer instructions and data definitions expressed in a form suitable for input to an *assembler*, *compiler*, or other *translator*. [5]

**special safety system** These are the shutdown systems, the emergency coolant injection system, and the containment system of a CANDU<sup>®</sup> nuclear generating station.

**SPH** Abbreviation for Standards and Procedures Handbook.

**SRS** Abbreviation for Software Requirements Specification.

**standard** 1. A standard is an approved, documented, and available set of criteria used to determine the adequacy of an action or object. 2. A document that sets forth the standards and *procedures* to be followed on a given project or by a given organization. 3. A *software engineering* standard

- defines the requirements on the procedures that define the processes for and
- provides descriptions that define the quantity and quality of a product from a software engineering project. [3]

**statistically valid** In *reliability qualification*, the characteristic that the distribution of the *test cases* matches that found in practice and that the sequence of test cases is longer than the longest sequence that would occur in actual use. [14]

**structured programming** A programming technique in which *programs* are constructed from a basic set of control structures, each having one entry and one exit. The set of control structures typically includes: sequence of two or more instructions, conditional selection of one of two or more sequences of instructions, and repetition of a sequence of instructions. [3]

**structuredness** A *quality attribute* that refers to the extent to which the *development outputs* possess a definite pattern in their interdependent parts. This implies that the design has proceeded in an orderly and systematic manner (e.g., *top-down design*), has minimized *coupling* between *modules* and *subsystems*, and that standard control structures have been used to produce a well structured *system*.

**subsystem** A secondary or subordinate *system* within a larger system. [5]

**support** A set of activities ancillary to *development*, *verification* and *validation* and comprises planning, *configuration management* and training. [5]

**syntax** The relationships of word groups, phrases, clauses, and sentences, that is, sentence structure. [3] Contrast with *semantics*.

**system** A collection of hardware, *software*, people, facilities, and *procedures* organized to accomplish some common objectives. [3]

**target** The version of a *configuration item* that is intended to be used in actual operation of the *computer system*.

**task** 1. A sequence of instructions treated as a basic unit of work by the supervisory program of an operating system. 2. In software design, a *software component* that can operate in parallel with other software components. [3]

**test** 1. An activity in which a system or component is executed under specified conditions, the results are observed and recorded, and an evaluation is made of some aspect of the system or component. 2. A set of one or more *test cases* and *test procedures*. [5]

**test case** A specific set of test data and associated *procedures* developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. [3]

**test procedure** 1. Detailed instructions for the set-up, execution, and evaluation of results for a given *test case*. 2. A document containing a set of associated instructions as in definition 1. 3. Documentation specifying a sequence of actions for the execution of a *test*. [5]

**testing** The process of detecting errors in order to produce a *system* that meets requirements under all credible operating conditions.

**top-down design** The process of designing a product by identifying its major components, decomposing them into their low-level components, and iterating this process until the desired level of detail is achieved. [3]

**traceability** A *quality attribute* that refers to those characteristics of the *development outputs* that provide a thread to their antecedent and subsequent documents. It also refers to the ability to trace the design decision history and reasons for changes.

**translator** A tool for translating from a higher level language to a lower level language (for example, a *compiler* or *assembler*).

**usability** A *quality objective* that requires that the *system* be easy and efficient to use, easy to learn, easy to remember, acceptable to *users*, and one which leads to few errors.

**understandability** A *quality attribute* that refers to the extent to which the meaning of the *development outputs* are clear to the third party.

**unit** a separately compilable component of the source code. Usually corresponds to a *module*.

**user** Someone who operates, maintains, tests or inspects a *system*.

**UTP** Abbreviation for Unit Test Procedures.

**UTR** Abbreviation for Unit Test Report.

**validation** The process of determining the correctness of the final product with respect to the initial requirements. [3] Contrast with *verification*.

**verifiability** A *quality attribute* that refers to the extent to which the *development outputs* have been created to facilitate *verification* using both static methods and *testing*.

**verification** The process of determining whether the products of a given phase fulfil the requirements established at the previous phase. [3] Contrast with *validation*.

**VTP** Abbreviation for Validation Test Procedures.

**VTR** Abbreviation for Validation Test Report.

**walkthrough** A software inspection process conducted by the peers of the developer to evaluate a *software item*. Although usually associated with code examination, this process is also applicable to the software requirements and software design. The major objectives of the walkthrough are to find defects (e.g., omissions, unwanted additions, and contradictions) in a specification or other product and to consider alternative functionality, performance objectives, or representations. [3]

**worst case analysis** A study and analysis of behaviour when all circumstances are as unfavourable as possible. [11]

## 7 REFERENCES

1. CANDU Computer Systems Engineering Centre of Excellence, "Standard for Software Engineering of Category II Software and Category III Software", CE-0101-STD, Revision 0, 2000.
2. CANDU Computer Systems Engineering Centre of Excellence, "Standard for Computer System Engineering", CE-0100-STD, Revision 0, December 1999.
3. Dorfman, M., Thayer, R.H., "Standards, Guidelines, and Examples on System and Software Requirements Engineering", Glossary, IEEE Computer Society Press, 1990.
4. Archinoff, G.A., Lau, D.K., de GrosBois, J., Bowman, W.C., "Guideline for Categorization of Software in Nuclear Power Plant Safety, Control, Monitoring and Testing Systems", September 1995, COG-95-264-I, Revision 1.0.
5. IEEE Std 610.12-1990, "IEEE Standard Glossary for Software Engineering Terminology", February 1991.
6. CAN/CSA-Q396.1.1-89, "Quality Assurance Program for the Development of Software Used in Critical Applications".
7. Parnas, D.L. and Madey, J., "Functional Documentation for Computer Systems Engineering", September 1990, Technical Report 90-287.
8. Pressman, R.S., "Software Engineering", New York, McGraw-Hill Book Company, 1987.
9. Myers, G.J., "The Art of Software Testing", New York, N.Y., Wiley-Interscience, July, 1979.
10. Ould, M.A., Unwin, C. (ed), "Testing in Software Development", British Computer Society Monograph, Cambridge University Press, 1988.
11. Oxford University Press, "Dictionary of Computing", Third Edition, 1990, ISBN 0 19 853825 1.
12. IEC 65A(Secretariat)94, "Software for Computers in the Application of Industrial Safety Related Systems".
13. MOD Defence Standard 00-56, "Safety Management Requirements for Defence Systems", DEF STAN 00-56, December 13, 1996.
14. Parnas, D.L., Schouwen, J., and Kwan, S.P., "Evaluation of Safety Critical Software", Communication of the ACM, Vol. 33, Number 6, June 1990, pp 636-648.

## 8 BIBLIOGRAPHY

The following documents were considered in the preparation of this standard.

AECL, “Standard for Software Engineering of Safety Critical Software – Supplement for Function Block Diagram Languages”, 00-00902-STD-002, Revision 0, November 1995.

AECL CANDU, “CANDU 3 Software Quality Assurance Manual”, 74-01913-MAN-003, Rev. 0, April 1991.

AECL CANDU, “Safety Software Design Principles”, 74-66300-DG-001.

Archinoff, G.A., Lau, D.K., de GrosBois, J., Bowman, W.C., “Guideline for Categorization of Software in Nuclear Power Plant Safety, Control, Monitoring and Testing Systems”, September 1995, COG-95-264-I, Revision 1.0.

Babich, W.A., “Software Configuration Management”, Addison-Wesley Publishing Company, 1986.

Bishop, P.G (ed), “Dependability of Critical Computer Systems 3”, Guidelines produced by EWICS TC7, 1, Elsevier Applied Science, London, 1990.

Booch, G., “Software Engineering with Ada”, Benjamin /Cummings Publishing Company, Menlo Park CA, 1987.

Boehm, B.W., “Software Engineering Economics”, Prentice-Hall Inc., 1981.

Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., Macleod, G.J., Merritt, M.J., “Characteristics of Software Quality”, TRW Series of Software Technology, Volume 1, TRW and North-Holland Publishing Company, 1978.

Bowen, J., Stavridou, V., “Safety Critical Systems, Formal Methods and Standards”, Submitted to IEEE Transactions on Software Engineering, January 30, 1992.

Buhr, R.J.A, “System Design with Ada”, Prentice-Hall, Englewood Cliffs, NJ, 1984.

CAN/CSA-Q396.1.1-89 “Quality Assurance Program for the Development of Software Used in Critical Applications”.

CAN3-N286.2-86 “Design Quality Assurance for Nuclear Power Plants”.

Daughtrey, H.T., “Measuring the Full Range of Software Qualities”, Babcock & Wilcox, ASQC Quality Congress, Dallas, 1988.

DOD-STD-2167a, “Defense System Software Development”, February 1988, and the following data item descriptions:

DI-CMAN-80534, “System/Segment Design Document (SSDD)”,

DI-MCCR-80030, “Software Development Plan (SDP)”,

DI-MCCR-80025, “Software Requirements Specification (SRS)”,  
DI-MCCR-80026, “Interface Requirements Specification (IRS)”,  
DI-MCCR-80027, “Interface Design Document (IDD)”,  
DI-MCCR-80012, “Software Design Document (SDD)”,  
DI-MCCR-80029, “Software Product Specification (SPS)”,  
DI-MCCR-80013, “Version Description Document (VDD)”,  
DI-MCCR-80014, “Software Test Plan (STP)”,  
DI-MCCR-80015, “Software Test Description (STD)”.

Dorfman, M., Thayer, R.H., “Standards, Guidelines, and Examples on System and Software Requirements Engineering”, Glossary, IEEE Computer Society Press, 1990.

EIA/IEEE J-STD-016:1995, “Standard for Information Technology – Software Life Cycle Processes – Software Development: Acquirer-Supplier Agreement”.

EPRI, “Software Development Roadmap – Requirements & Guidelines”, Version 1.2, March 22, 1999.

European Space Agency, “ESA Software Engineering Standards”, ESA PSS-05-0 Issue 2, 1991 February.

Fenton, N.E., Neil, M., “A Strategy for Improving Safety Related Software Engineering Standards”, IEEE Transactions on Software Engineering, November 1998, Volume 24, Number 11, ISSN 0098-5589.

Ferguson, J., Sheard, S., “Leveraging your CMM Efforts for IEEE/EIA 12207”, IEEE Software, September/October 1998, Volume 15, Number 5, 0740-7459.

Glass, R.L., “Inspections – Some Surprising Findings”, Communications of the ACM, Volume 42, No. 4, April 1999.

Gerhart, S., Craigen, D., Ralston, R.-P., “Experience with Formal Methods in Critical Systems”, IEEE Software, January 1994, pp. 21-39.

Harauz, J., “International Trends in Software Engineering and Quality System Standards from the Perspective of Ontario Hydro”, VIII International Conference on Software Technology: Software Quality, June 11-13, 1997, Curitiba, Brazil.

Herrmann, D.S., “Software Safety and Reliability”, IEEE Computer Society, 1999, ISBN 0-7695-0299-7.

Hicks, D.B., Harauz, J. and Hohendorf, R.J., “Software Quality Assessment Process”, CANDU Owner's Group Conference, Toronto, November 1990.

IEC SC45A – WG3, “Nuclear Power Plants, Instrumentation and Control, Computer-based systems important for safety, Software aspects for I&C systems of class 2 and 3”, March 1999.

IEC 60880, “Software for Computers in the Safety Systems of Nuclear Power Stations”, previously known as IEC 880.

IEC 60880 Amd.1 Ed. 1.0, “Amendment to IEC 60880 – Software for computers important to safety for nuclear power plants – First supplement to IEC 60880”, draft.

IEC 61508, “Functional safety of electrical/electronic/programmable electronic safety-related systems”, Version 4.0, May 12, 1997

IEEE Std 730-1989, “IEEE Standard for Software Quality Assurance Plans (ANSI)”.

IEEE Std 730.1-1995, “IEEE Guide for Software Quality Assurance Plans (ANSI)”.

IEEE Std 828-1990, “IEEE Standard for Software Configuration Management Plans (ANSI)”.

IEEE Std 829-1983, (Reaff 1991), “IEEE Standard for Software Test Documentation (ANSI)”.

IEEE Std 830-1993, “IEEE Recommended Practice for Software Requirements Specifications (ANSI)”.

IEEE Std 982.2-1988, “IEEE Guide for the Use of IEEE Standard Dictionary”.

IEEE Std 1008-1987 (Reaff 1993), “IEEE Standard for Software Unit Testing (ANSI)”.

IEEE Std 1012-1986 (Reaff 1992), “IEEE Standard for Software Verification and Validation Plans (ANSI)”.

IEEE Std 1028-1988 (Reaff 1993), “IEEE Standard for Software Reviews and Audits (ANSI)”.

IEEE Std 1042-1987 (Reaff 1993), “IEEE Guide for Software Configuration Management (ANSI)”.

IEEE Std 1061-1992, “IEEE Standard for a Software Quality Metrics Methodology (ANSI)”.

IEEE Std 1074-1995, “IEEE Standard for Developing Software Life Cycle Processes (ANSI)”.

IEEE Std 1074.1-1995, “IEEE Guide for Developing Software Life Cycle Processes (ANSI)”.

IEEE/EIA Std 12207.0-1996, “Software life cycle processes”.

IEEE/EIA Std 12207.1-1997, “Software life cycle processes – Life cycle data”.

IEEE/EIA Std 12207.2-1997, “Software life cycle processes – Implementation considerations”.

IEEE Transactions on Software Engineering, “Special Section on Software Engineering Project Management”, Vol SE-10, No. 1, January 1984.



- ISO/IEC 9126:1991E, “Information technology – Software product evaluation – Quality characteristics and guidelines for their use”.
- ISO/IEC 12207:1995(E), “Information technology – Software life cycle processes”.
- ISO/IEC JTC1/SC7/WG3, N-136, “Evaluation of Methods and Tools”, 1989.
- ISO/IEC JTC1/SC7/WG3/SG2, “Foundations of Software Quality Assessment”, R.E. Nance and J.D. Arthur, Systems research Center and The Department of Computer Science, Virginia Tech, 1990.
- Jensen, R.W., Tonies, C.C., “Software Engineering”, Prentice-Hall Inc., 1979.
- Joint Services Computer Resources Management Group, “Software System Safety Handbook”, Draft, September 30, 1997.
- Juran, J.M., Gryna, F.M., (ed), “Juran's Quality Control Handbook”, 4th edition, McGraw Hill Book Company, 1988.
- Kopetz, H., “Software Reliability”, Macmillan Press Ltd., 1979.
- Lamb, D.A., “Software Engineering, Planning for Change”, Englewood Cliffs, NJ: Prentice-Hall, 1988.
- Le Metayer, D., Nicolas, V., Ridoux, O., “Exploring the Software Development Trilogy”, IEEE Software, November/December 1998, Volume 15, Number 6, 0740-7459.
- Leveson, N.J., Safeware: “System Safety and Computers”, Addison-Wesley, September 1995, ISBN 0-201-11972-2.
- Martin, J., McClure, C., “Software Maintenance”, Prentice-Hall Inc., 1983.
- MOD Defence Standard 00-55, “Requirements for Safety Related Software in Defence Equipment”, August 1, 1997.
- Myers, G.J., “The Art of Software Testing”, New York, N.Y. Wiley-Interscience, July, 1979.
- NUREG/CR-4640, PNL-5787, “Handbook of Software Quality Assurance Techniques Application to the Nuclear Industry”.
- NASA, “Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems – Volume I: Planning and Technology Insertion”, December 1998, NASA/TP-98-208193, Release 2.0.
- NASA, “Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems – Volume II: A Practitioner’s Companion”, May 1997, NASA-GB-002.

- Ontario Hydro, Darlington Engineering Department, "Darlington Nuclear Generating Station A Control Computers Software Quality Assurance Manual", NK38-69000, Ontario Hydro, December 15, 1986.
- Ontario Hydro, "Quality Engineering Manual", 968-01913.1T1, Ontario Hydro, March 11, 1988.
- Ould, M.A., Unwin, C. (ed), "Testing in Software Development", British Computer Society Monograph, Cambridge, University Press, 1988.
- Parnas, D.L., "Proposed Standard for Software for Computers in the Safety Systems of Nuclear Power Stations (based on IEC Standard 880)", (DLP-880), Queen's University, March 1991.
- Parnas, D.L., Madey, J., "Functional Documentation for Computer Systems Engineering", Kingston, Ontario: Telecommunications Research Institute of Ontario, Queen's University, Technical Report 90-287, ISSN O836-0227, September 1990.
- Pressman, R.S., "Software Engineering", New York, McGraw-Hill Book Company, 1987.
- Rannem, S., and E. Hung, "On Factors Contributing to Quality of Nuclear Control Computer Software", Ontario Hydro.
- Redmill, F.J. (ed), "Dependability of Critical Computer Systems 1", Guidelines produced by EWICS TC7, 1, Elsevier Applied Science, London, 1988.
- Redmill, F.J. (ed), "Dependability of Critical Computer Systems 2", Guidelines produced by EWICS TC7, 1, Elsevier Applied Science, London, 1989.
- Rook, P. (ed) "Software Reliability Handbook", Elsevier Applied Science, London, 1990.
- Sheard, S.A., "The Frameworks Quagmire, A Brief Look", 1997 INCOSE Conference, Los Angeles, August 3-7, 1997.
- Smith, D.J., Wood, K.B., "Engineering Quality Software", 2nd edition, London: Elsevier Applied Science, 1989.
- Storey, N., "Safety Critical Computer Systems", Addison Wesley Longman, 1996, ISBN 0-201-42787-7.
- Tausworthe, R.C., "Standardized Development of Computer Software", Prentice-Hall Inc., 1977.

## **APPENDIX A PREREQUISITES FOR THE SOFTWARE ENGINEERING PROCESS**

This appendix identifies the information that must be available in the Design Input Documentation (DID) to enable the software engineering process to begin. Note that if the Standard for Computer System Engineering [2] has been applied, this information will be contained in its development outputs.

The DID shall:

- (a) Partition the *computer system* so that the safety critical software subsystem is isolated from other subsystems and the software functionality not associated with meeting the required performance specification of the special safety system is minimized.
- (b) Define the functional, performance, safety, reliability, security, operability, and maintainability requirements of the subsystem, clearly identifying the safety requirements.
- (c) Define the scope and boundary of the subsystem and define all details of the interfaces to other subsystems.
- (d) Contain or reference a description of the problem domain including natural restrictions on the required behaviour.
- (e) Define human interface requirements.
- (f) Define all accuracy requirements and tolerances.
- (g) Define all design constraints on the software.
- (h) Define any Quality Assurance and Software Quality Assurance (SQA) requirements to be met by the processes used to implement the computer system.
- (i) Define the quality objectives, if different to those defined in this standard.
- (j) Define anticipated changes and enhancements to the system.
- (k) Provide a clear definition of terms.
- (l) Contain no requirements that are in conflict with each other.
- (m) Identify each requirement uniquely and define each requirement completely in one location to prevent inconsistent updates and to facilitate easy referencing by subsequent documents and verification processes.
- (n) Be complete, consistent and unambiguous.

- (o) Define the reliability requirements of the software in a form useful for reliability qualification.
- (p) Define all failure modes (identified by a hazards analysis of the computer system), and the appropriate response to them and identify any degraded operation modes required to be handled.
- (q) Limit and localize the use of complex calculations upon which safety critical decisions depend.
- (r) Contain or reference a revision history.

## **APPENDIX B      RATIONALE FOR THE REQUIREMENTS OF THIS STANDARD**

In this standard, a *software* product is considered to include the full set of *development outputs*. The overall *quality* of a software product is determined by the extent to which the product achieves a set of desired *quality objectives*.

These quality objectives can be built into a software product by adhering to a process governed by some *software engineering* principles, and can be measured in a software product by a set of more readily assessable *quality attributes*.

This standard defines a set of activities and specific requirements that are derived from the quality attributes and software engineering principles defined in the following sections. These attributes are used to describe development outputs only, because the purpose of the *verification*, *validation* and *support* processes is to provide assurance that the quality attributes of the development outputs are achieved.

Coverage of the quality objectives by the requirements on the development outputs is documented in the last section of this appendix.

## B.1 Quality Objectives

The *quality objectives* listed below consist of primary and secondary objectives. The secondary quality objectives shall only be considered if they do not compromise the primary quality objectives.

The primary quality objectives are:

**Safety:** A quality objective that requires that the system function in accordance with its *requirements*, in a consistent and predictable manner, under all conditions. When the system can no longer perform its required role, this quality objective requires that it act to maintain the equipment and processes it controls in a safe state in all situations. This objective implies the quality attributes *completeness, correctness, predictability, robustness, understandability, and verifiability*.

**Functionality:** A quality objective that requires that the system implement all required behaviour and meet the *performance requirements*. This objective implies the quality attributes *completeness, correctness, understandability, and verifiability*.

**Reliability:** A quality objective that requires that the system perform its required behaviour such that the probability of it successfully performing that behaviour is consistent with the *reliability requirements* identified. This objective implies the quality attributes *predictability and robustness*.

**Maintainability:** A quality objective that requires that the system be structured so that those items most likely to require modification can be changed reliably and efficiently. This quality objective also requires the rationale for *design decisions* be evident to a third party. This objective implies the quality attributes *consistency, modifiability, modularity, structuredness, traceability, understandability and verifiability*.

**Reviewability:** A quality objective that requires that the system be developed and documented so that it can be systematically inspected by a third party for conformance to *requirements*. This objective implies the quality attributes *consistency, modularity, structuredness, traceability, understandability and verifiability*.

The secondary quality objectives are:

**Portability:** A quality objective that requires that the software be capable of being transferred to different environments to the degree required by *users*. This objective does not directly imply any specific quality attributes.

**Usability:** A quality objective that requires that the *system* be easy and efficient to use, easy to learn, easy to remember, acceptable to users, and one which leads to few *errors*. This objective implies the quality attributes *predictability and understandability*.

**Efficiency:** A quality objective that requires that the system achieve an acceptable level of performance within the identified constraints. This objective does not directly imply any specific quality attributes.

## B.2 Quality Attributes

The *quality attributes* are:

**Completeness:** A quality attribute that refers to the extent to which all of the required behaviour and *design constraints* are present and fully developed in the *development outputs*.

**Consistency:** A quality attribute that refers to the extent to which the development outputs contain uniform notations, terminology, comments, symbology, and implementation techniques.

**Correctness:** A quality attribute that refers to the ability of the development outputs to describe or produce the specified outputs when given the specified inputs, and the extent to which they match or satisfy the requirements in the development inputs.

**Modifiability:** A quality attribute that refers to the characteristics of the development outputs which facilitate the incorporation of changes.

**Modularity:** A quality attribute that refers to the extent to which the development output is composed of discrete components such that a change to one component has minimal impact on the others.

**Predictability:** A quality attribute that refers to the extent to which the behaviour and performance of the *system* are deterministic for a specified set of inputs.

**Robustness:** A quality attribute that refers to the extent to which the development outputs require and implement the ability to continue to perform despite some *subsystem* failure.

**Structuredness:** A quality attribute that refers to the extent to which the development outputs possess a definite pattern in their interdependent parts. This implies that the design has proceeded in an orderly and systematic manner (e.g., *top-down design*), has minimized *coupling* between *modules* and *subsystems*, and that standard control structures have been used to produce a well structured system.

**Traceability:** A quality attribute that refers to those characteristics of the development outputs that provide a thread to their antecedent and subsequent documents. It also refers to the ability to trace the *design decision* history and reasons for changes.

**Understandability:** A quality attribute that refers to the extent to which the meaning of the development outputs are clear to a third party.

**Verifiability:** A quality attribute that refers to the extent to which the development outputs have been created to facilitate *verification* using both static methods and testing.

### B.3 Software Engineering Principles

The *software engineering* principles, together with the *quality attributes* they address, are:

- (a) The required behaviour of the *software* shall be documented. Mathematical functions in a notation that has well defined *syntax* and *semantics* shall be used. [*completeness, consistency, correctness, predictability, understandability, verifiability*]
- (b) The structure of the software shall be based on *information hiding* and the use of recognized *software engineering* practices (for example: Structured Analysis and Design, Object-Oriented Design, Top-Down Decomposition, *Structured Programming*, etc.) [*modifiability, modularity, predictability, structuredness, understandability, verifiability*]
- (c) The outputs from each development process shall be reviewed to verify that they comply with the *requirements* specified in the inputs to that process. Mathematical *verification* techniques or rigorous arguments of correctness shall be used to verify that the code meets the required behaviour specified in the SDD, which meets the required behaviour specified in the SRS. [*completeness, correctness, traceability, understandability*]
- (d) Verification of the software shall be carried out throughout its entire life. Any changes shall be verified to the same or better degree of rigour as the original development. The extent of verification necessary is determined by an analysis of the scope of the change. [*correctness, verifiability*]
- (e) Independence of design and verification or *validation* personnel to the extent required shall be maintained to help ensure an unbiased verification or validation process. [*correctness*]
- (f) Engineering of the software shall follow a planned and systematic process over the entire life of the software. All activities in the software *life cycle* shall be carried out using approved *procedures* and *guidelines* that conform to the intent of this standard. [*correctness, modifiability, verifiability, structuredness*]
- (g) *Configuration management* shall be maintained throughout the entire life of the software to ensure up-to-date and consistent software and documentation. [*consistency, correctness, verifiability, traceability*]
- (h) Training shall be undertaken to ensure that personnel have the knowledge, skills and attitudes required to perform their jobs. [*correctness, understandability*]
- (i) Behaviour over the full range of possible values of *monitored variables* shall be specified and implemented. [*completeness, predictability*]
- (j) All errors reported from interactions with hardware or *predeveloped software* shall be checked and handled. [*completeness, predictability, robustness*]



- (k) The software design shall be as conservative and simple as possible, while meeting all requirements and retaining spare capacity for anticipated changes. [*modifiability, understandability, verifiability*]
- (l) Analyses shall be performed to identify and evaluate safety *hazards* associated with the computer system with the aim of either eliminating them or assisting in the reduction of any associated risks. [*robustness*]
- (m) Reliability of the *safety critical software* shall be demonstrated using a *statistically valid, trajectory-based methodology* in which a *reliability hypothesis* is proven to the degree of confidence required. [*robustness*]
- (n) Adequate testing shall be performed to meet predefined test coverage criteria. [*correctness, robustness*]

## B.4 Coverage of the Quality Attributes

### B.4.1 Coverage of the Quality Attributes by the SRS

To ensure that the output of the design and coding activities fulfil their acceptance criteria for the quality objectives *safety, functionality, reliability, reviewability* and *maintainability*, the SRS shall have the following quality attributes: *completeness, consistency, correctness, modifiability, robustness, traceability, understandability* and *verifiability*. Coverage of these quality attributes is documented in the following table.

Requirements on the SRS	completeness	consistency	correctness	modifiability	modularity	predictability	robustness	structuredness	traceability	understandability	verifiability
3.1.2.a - Specify all requirements from the DID	√										
3.1.2.b - Specify additional requirements	√		√								
3.1.2.c - Describe context										√	
3.1.2.d - Identify external properties	√									√	
3.1.2.e - Describe I/O characteristics	√										
3.1.2.f - Describe I/O relationships	√										
3.1.2.g - Define behaviour	√		√							√	√
3.1.2.h - Specify exception response	√						√				
3.1.2.i - Specify fault tolerance							√				
3.1.2.j - Specify timing tolerances	√										
3.1.2.k - Identify anticipated changes	√			√							
3.1.2.l - Identify design constraints	√										
3.1.2.m - Requirements and constraints only	√			√							
3.1.2.n - No conflicts		√	√							√	√
3.1.2.o - Unique requirements		√		√							
3.1.2.p - Identify requirements uniquely									√		
3.1.2.q - Reference design notes									√		
3.1.2.r - Demonstrate mapping									√		
3.1.2.s - Conform to standards	√	√	√								
3.1.2.t - Use consistent terminology		√								√	√
3.1.2.u - Enable role determination			√	√						√	√





## INDEX

- accuracy ..... 8, 14, 45, 57
- activities 3, 27, 28, 31, 32, 34, 37, 38, 42, 46, 49, 59, 62, 64
- animation ..... 31, 39
- baseline ..... 34, 39
- Baseline Release Package Index (BRPI) ..... 29, 39
- boundary ..... 20, 39, 43, 44, 57
- change control ..... 34, 39
- change request ..... 29, 34, 38, 39
- code iii, 11, 12, 17, 18, 19, 20, 21, 23, 24, 26, 34, 39, 40, 42, 48, 50, 66
- code implementation ..... 1, 3, 14, 17, 39
- code review ..... vii, 6, 15, 17, 32, 36, 41
- Code Review Report (CRR) ..... 6, 17, 41
- code verification ..... 36
- Code Verification Report (CVR) ..... 6, 18, 41
- cohesion ..... 10, 39, 40
- commissioning ..... iii, 31
- completeness ..... 40, 41, 60, 61, 62, 64, 65, 66
- complexity ..... v, 31, 65
- configuration 1, 21, 29, 34, 37, 38, 39, 40, 42, 43, 46, 47, 49, 50
- configuration control ..... 39, 40, 42, 47, 50
- configuration management ..... 1, 34, 37, 38, 40, 49
- consistency ..... 40, 60, 61, 62, 64, 65, 66
- controlled variable ..... 8, 40, 43, 45
- convention ..... 1, 13, 44
- correctness ..... 40, 50, 60, 61, 62, 63, 64, 65, 66
- coupling ..... 10, 39, 40, 49, 61
- deliverable ..... 33, 34, 41
- design iii, iv, 1, 3, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16, 23, 24, 30, 31, 33, 35, 36, 39, 40, 41, 42, 43, 44, 45, 47, 48, 49, 50, 57, 60, 61, 62, 63, 64, 65, 66
- design constraint 7, 8, 9, 12, 15, 16, 40, 41, 57, 61, 64
- Design Input Documentation (DID) 2, 3, 7, 8, 15, 16, 22, 23, 24, 28, 30, 34, 41, 57, 64
- design notes ..... 6, 8, 11, 14, 33, 41, 64, 65, 66
- design review ..... vii, 6, 15, 16, 31, 32, 36, 41, 42
- Design Review Report (DRR) ..... 6, 16, 41
- design verification ..... 31, 36
- Design Verification Report (DVR) ..... 6, 18, 41
- development ii, 3, 7, 28, 30, 31, 32, 33, 36, 37, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 57, 59, 61, 62
- development output ii, 3, 36, 40, 41, 42, 44, 45, 46, 49, 50, 57, 59, 61
- development process ..... 2, 3, 7, 32, 36, 41, 45, 62
- efficiency ..... 42, 60
- failure ..... 8, 23, 24, 42, 46, 58, 61
- fault ..... 8, 9, 42, 64, 65
- fault tolerance ..... 8, 9, 42, 64, 65
- four variable model ..... 40, 43, 44, 45
- functionality 10, 13, 16, 18, 30, 31, 43, 50, 57, 60, 64
- graceful degradation ..... 8, 9, 43
- guideline 1, 12, 13, 14, 16, 17, 36, 37, 43, 44, 51, 52, 62, 66
- hazards analysis v, vii, viii, 1, 3, 6, 8, 15, 23, 32, 43, 58
- hold point ..... 31, 43
- independence ..... v, viii, 25, 28, 32, 34, 46, 62
- input variable ..... ii, 8, 11, 43
- inspection ..... 43, 50
- Integration Test Procedures (ITP) ..... 6, 21, 22, 44
- Integration Test Report (ITR) ..... 6, 21, 22, 44
- life cycle ..... 28, 30, 44, 54, 55, 62
- maintainability ..... 7, 9, 30, 44, 57, 60, 64
- methodology ..... v, 1, 32, 35, 36, 44, 46, 54, 63
- modifiability ..... 44, 60, 61, 62, 63, 64, 65, 66
- modularity ..... 44, 60, 61, 62, 64, 65, 66
- module ..... 9, 10, 19, 43, 44, 45, 50, 65
- monitored variable ..... 8, 43, 44, 62
- operability ..... 7, 9, 44, 57
- output variable ..... ii, 8, 11, 43, 45
- performance ii, 1, 7, 9, 13, 21, 22, 42, 43, 45, 46, 47, 50, 57, 60, 61, 65
- personnel ..... 26, 28, 29, 31, 32, 33, 35, 37, 47, 62
- planning ..... 1, 28, 30, 45, 49
- policy statement ..... 33
- portability ..... 45, 60
- predictability ..... 45, 60, 61, 62, 64, 65, 66
- procedure ..... 26, 37, 45, 49
- program v, 9, 10, 11, 12, 13, 14, 19, 20, 39, 40, 41, 42, 44, 45, 46, 48, 49, 65
- Project Execution Plan (PEP) ..... 28, 45
- prototyping ..... 30, 31, 45

qualification	iii, v, viii, 1, 2, 3, 6, 15, 24, 25, 26, 32, 33, 35, 45, 46, 48, 58	Software Quality Assurance (SQA)	57
quality attribute	iv, 40, 44, 45, 46, 49, 50, 59, 60, 61, 62, 64, 65, 66	Software Requirements Specification (SRS)	viii, 6, 7, 9, 11, 12, 15, 16, 17, 18, 21, 23, 24, 32, 34, 36, 48, 53, 54, 62, 64, 65
quality objective	iii, 42, 43, 44, 45, 46, 47, 50, 57, 59, 60, 64	Standards and Procedures Handbook (SPH)	iii, vii, 3, 6, 7, 8, 9, 12, 13, 15, 16, 17, 18, 20, 21, 22, 23, 26, 27, 28, 29, 30, 34, 35, 37, 48
real-time	v, 1, 2, 46	structured programming	13, 48, 66
record	29, 33, 34, 35, 38, 46	structuredness	49, 60, 61, 62, 64, 65, 66
regression testing	36, 46	support	iii, 2, 3, 25, 26, 27, 28, 30, 31, 32, 33, 34, 36, 38, 41, 45, 49, 59
release	3, 32, 38, 46	support output	iii
reliability	iii, v, vii, viii, 1, 3, 6, 7, 9, 15, 23, 24, 25, 30, 31, 32, 46, 47, 48, 53, 55, 56, 57, 58, 60, 63, 64	support process	2, 3, 28, 31, 32, 36, 41, 59
reliability hypothesis	iii, 24, 46, 63	syntax	8, 11, 12, 47, 49, 62
reliability qualification	v, vii, viii, 1, 3, 6, 15, 24, 25, 32, 46, 48, 58	systematic code verification	vii, 6, 17, 18, 32, 36, 47
requirements	ii, iii, v, 1, 2, 3, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 30, 31, 32, 33, 35, 36, 40, 41, 42, 43, 45, 46, 47, 48, 49, 50, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66	systematic design verification	vii, 6, 17, 18, 32, 36, 47
requirements definition	1, 3	systematic verification	vii, viii, 1, 3, 6, 15, 17, 25
requirements review	vii, 6, 15, 16, 32, 36, 47	terminology	2, 9, 12, 37, 39, 40, 61, 64, 65
Requirements Review Report (RRR)	6, 15, 16, 47	testing	iii, vii, viii, 1, 3, 6, 15, 19, 20, 21, 22, 31, 32, 36, 37, 42, 46, 47, 48, 49, 50, 51, 52, 54, 55, 56, 61, 63
review	ii, vii, viii, 1, 3, 6, 15, 16, 17, 23, 25, 31, 32, 34, 36, 41, 42, 43, 46, 47	tolerance	8, 9, 42, 57, 64, 65
review reports	25	traceability	40, 41, 49, 60, 61, 62, 64, 65, 66
reviewability	30, 46, 60, 64	training	vii, 1, 6, 28, 29, 30, 31, 32, 35, 38, 49, 62
revision	ii, 13, 38, 58	training record	29, 35, 38
robustness	9, 46, 60, 61, 62, 63, 64, 65, 66	understandability	50, 60, 61, 62, 63, 64, 65, 66
safety	1, v, 1, 2, 6, 7, 9, 21, 23, 24, 30, 35, 39, 46, 47, 48, 51, 52, 53, 54, 55, 56, 57, 58, 60, 63, 64	Unit Test Procedures (UTP)	6, 20, 50
safety critical	v, 1, 2, 6, 39, 47, 57, 58, 63	Unit Test Report (UTR)	6, 20, 50
safety design principle	23, 24, 35	usability	50, 60
safety related	47	validation	vii, 2, 3, 6, 15, 19, 22, 23, 25, 27, 31, 32, 41, 49, 50, 54, 59, 62
safety requirement	47, 57	validation test	22
safety-related	54	Validation Test Procedures (VTP)	6, 22, 50
security	7, 9, 21, 38, 47, 57	Validation Test Report (VTR)	6, 22, 23, 50
semantics	8, 11, 12, 47, 49, 62	verifiability	50, 60, 61, 62, 63, 64, 65, 66
Software Design Description (SDD)	iii, viii, 6, 8, 9, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21, 23, 34, 36, 47, 53, 62, 65	verification	iii, v, vii, viii, 1, 2, 3, 6, 15, 17, 18, 25, 27, 28, 30, 31, 32, 33, 34, 36, 37, 41, 47, 49, 50, 54, 55, 57, 59, 61, 62
Software Development Plan (SDP)	vii, 3, 6, 11, 28, 29, 30, 31, 32, 33, 34, 35, 37, 47, 52	verification output	iii, 16, 17, 18, 19, 24, 25, 26, 27, 36
software engineering	1, ii, iii, v, viii, 1, 2, 3, 6, 16, 17, 28, 29, 30, 32, 39, 41, 45, 48, 51, 52, 53, 54, 55, 56, 57, 59, 62	verification process	15, 17, 18, 27, 36, 57, 62
		walkthrough	31, 43, 50