



The European Organisation for Civil Aviation Equipment
L'Organisation Européenne pour l'Équipement de l'Aviation Civile

OBJECT-ORIENTED TECHNOLOGY AND RELATED TECHNIQUES SUPPLEMENT TO ED-12C AND ED-109A

This document is the exclusive intellectual and commercial property of EUROCAE.
It is presently commercialised by EUROCAE.

This electronic copy is delivered to your company/organisation for internal use exclusively.

In no case it may be re-sold, or hired, lent or exchanged outside your company.

ED-217

January 2012

OBJECT-ORIENTED TECHNOLOGY AND RELATED TECHNIQUES SUPPLEMENT TO ED-12C AND ED-109A

This document is the exclusive intellectual and commercial property of EUROCAE.

It is presently commercialised by EUROCAE.

This electronic copy is delivered to your company/organisation for internal use exclusively.

In no case it may be re-sold, or hired, lent or exchanged outside your company.

ED-217

January 2012

FOREWORD

1. This document was prepared jointly by EUROCAE Working Group 71 (WG-71) and RTCA Special Committee 205 (SC-205), was approved by the Council of EUROCAE on 9 January 2012.
2. EUROCAE is an international non-profit making organisation in Europe. Membership is open to manufacturers and users of equipment for aeronautics, trade associations, national civil aviation administrations, and, under certain conditions, non-European organisations. Its work programme is principally directed to the preparation of performance specifications and guidance documents for civil aviation equipment, for adoption and use at European and world-wide levels.
3. The findings of EUROCAE are resolved after discussion amongst Members of EUROCAE and in collaboration with RTCA Inc, Washington D.C, through their appropriate committees.
4. EUROCAE performance specifications and other documents are recommendations only. EUROCAE is not an official body of the European Governments. Its recommendations are valid as statements of official policy only when adopted by a particular government or conference of governments.
5. Copies of this document may be obtained from:

EUROCAE
102, rue Etienne Dolet
92240 MALAKOFF
France

Telephone: 33 1 40 92 79 30

Fax: 33 1 46 55 62 65

Email: eurocae@eurocae.net

Website: www.eurocae.net

TABLE OF CONTENTS

OO.1.0	INTRODUCTION	1
	OO.1.1 Purpose.....	1
	OO.1.2 Scope.....	1
	OO.1.3 Relationship to Other Documents.....	2
	OO.1.4 How to Use This Supplement	2
	OO.1.5 Document Overview	2
	OO.1.6 Characteristics of Object-oriented Technology and Related Techniques	2
	OO.1.6.1 Basic Concepts.....	3
	OO.1.6.1.1 Classes and Objects.....	3
	OO.1.6.1.2 Types and Type Safety.....	3
	OO.1.6.1.3 Hierarchical Encapsulation	4
	OO.1.6.1.4 Polymorphism	5
	OO.1.6.1.5 Function Passing and Closures.....	5
	OO.1.6.1.6 Method Dispatch.....	6
	OO.1.6.2 Key Features and Related Techniques	6
	OO.1.6.2.1 Inheritance	7
	OO.1.6.2.2 Parametric Polymorphism	7
	OO.1.6.2.3 Overloading (ad hoc polymorphism).....	7
	OO.1.6.2.4 Type Conversion.....	7
	OO.1.6.2.5 Software Exceptions and Exception Handling.	8
	OO.1.6.2.6 Dynamic Memory Management.....	8
	OO.1.6.2.7 Virtualization Techniques	8
OO.2.0	SYSTEM ASPECTS RELATING TO SOFTWARE DEVELOPMENT	10
OO.3.0	SOFTWARE LIFE CYCLE	11
OO.4.0	SOFTWARE PLANNING PROCESS	12
	OO.4.1 Software Planning Process Objectives.....	12
	OO.4.2 Software Planning Process Activities	13
	OO.4.3 Software Plans.....	14
	OO.4.4 Software Life Cycle Environment Planning	14
	OO.4.4.1 Software Development Environment.....	15
	OO.4.4.2 Language and Compiler Considerations	15
	OO.4.4.3 Software Test Environment	15
	OO.4.5 Software Development Standards.....	15
	OO.4.6 Review of the Software Planning Process.....	16
OO.5.0	SOFTWARE DEVELOPMENT PROCESSES	17
	OO.5.1 Software Requirements Process	17
	OO.5.1.1 Software Requirements Process Objectives.....	17
	OO.5.1.2 Software Requirements Process Activities	17
	OO.5.2 Software Design Process	17
	OO.5.2.1 Software Design Process Objectives	17
	OO.5.2.2 Software Design Process Activities.....	18
	OO.5.2.3 Designing for User-Modifiable Software.....	18
	OO.5.2.4 Designing for Deactivated Code.....	19
	OO.5.3 Software Coding Process	19
	OO.5.3.1 Software Coding Process Objectives.....	19
	OO.5.3.2 Software Coding Process Activities.....	19
	OO.5.4 Integration Process.....	19

	OO.5.4.1 Integration Process Objectives.....	19
	OO.5.4.2 Integration Process Activities	19
	OO.5.5 Software Development Process Traceability	19
OO.6.0	SOFTWARE VERIFICATION PROCESS	20
	OO.6.1 Purpose of Software Verification	20
	OO.6.2 Overview of Software Verification Process Activities	20
	OO.6.3 Software Reviews and Analyses	21
	OO.6.3.1 Reviews and Analyses of High-Level Requirements	21
	OO.6.3.2 Reviews and Analyses of Low-Level Requirements	21
	OO.6.3.3 Reviews and Analyses of Software Architecture.....	21
	OO.6.3.4 Reviews and Analyses of Source Code	22
	OO.6.3.5 Reviews and Analyses of the Outputs of the Integration Process	22
	OO.6.4 Software Testing.....	22
	OO.6.4.1 Test Environment	22
	OO.6.4.2 Requirements-Based Test Selection.....	22
	OO.6.4.2.1 Normal Range Test Cases	23
	OO.6.4.2.2 Robustness Test Cases.....	23
	OO.6.4.3 Requirements-Based Testing Methods.....	23
	OO.6.4.4 Test Coverage Analysis	23
	OO.6.4.4.1 Requirements-Based Test Coverage Analysis...	23
	OO.6.4.4.2 Structural Coverage Analysis	23
	OO.6.4.4.3 Structural Coverage Analysis Resolution	23
	OO.6.4.5 Reviews and Analyses of Test Cases, Procedures, and Results	23
	OO.6.5 Software Verification Process Traceability	23
	OO.6.6 Verification of Parameter Data Items.....	23
	OO.6.7 Local Type Consistency Verification.....	23
	OO.6.7.1 Local Type Consistency Verification Objective	23
	OO.6.7.2 Local Type Consistency Verification Activity	24
	OO.6.8 Dynamic Memory Management Verification.....	24
	OO.6.8.1 Dynamic Memory Management Verification Objective	24
	OO.6.8.2 Dynamic Memory Management Verification Activities	24
OO.7.0	SOFTWARE CONFIGURATION MANAGEMENT PROCESS	25
OO.8.0	SOFTWARE QUALITY ASSURANCE PROCESS	26
	OO.8.1 Software Quality Assurance Process Objectives	26
	OO.8.2 Software Quality Assurance Process Activities	26
	OO.8.3 Software Conformity Review	27
OO.9.0	CERTIFICATION LIAISON PROCESS.....	28
	OO.9.1 Means of Compliance and Planning.....	28
	OO.9.2 Compliance Substantiation	28
	OO.9.3 Minimum Software Life Cycle Data Submitted to Certification Authority	28
	OO.9.4 Software Life Cycle Data Related to Type Design	28
OO.10.0	OVERVIEW OF CERTIFICATION PROCESS.....	29
OO.11.0	SOFTWARE LIFE CYCLE DATA.....	30
	OO.11.1 Plan for Software Aspects of Certification.....	30
	OO.11.2 Software Development Plan.....	31
	OO.11.3 Software Verification Plan	31
	OO.11.4 Software Configuration Management Plan.....	31
	OO.11.5 Software Quality Assurance Plan.....	31

OO.11.6	Software Requirements Standards	31
OO.11.7	Software Design Standards.....	31
OO.11.8	Software Code Standards	31
OO.11.9	Software Requirements Data	32
OO.11.10	Design Description	32
OO.11.11	Source Code.....	32
OO.11.12	Executable Object Code.....	32
OO.11.13	Software Verification Cases and Procedures.....	32
OO.11.14	Software Verification Results	32
OO.11.15	Software Life Cycle Environment Configuration Index.....	32
OO.11.16	Software Configuration Index	32
OO.11.17	Problem Reports.....	32
OO.11.18	Software Configuration Management Records	33
OO.11.19	Software Quality Assurance Records.....	33
OO.11.20	Software Accomplishment Summary	33
OO.11.21	Trace Data.....	34
OO.11.22	Parameter Data Item File	34
OO.12.0	ADDITIONAL CONSIDERATIONS	35
OO.12.1	Use of Previously Developed Software	35
OO.12.1.1	Modifications to Previously Developed Software	35
OO.12.1.2	Change of Aircraft Installation.....	35
OO.12.1.3	Change of Application or Development Environment.....	35
OO.12.1.4	Upgrading a Development Baseline	35
OO.12.1.5	Software Configuration Management Considerations.....	35
OO.12.1.6	Software Quality Assurance Considerations	35
OO.12.2	Tool Qualification	35
OO.12.3	Alternative Methods	35
OO.12.3.4.2	Sufficiency of Accumulated Service History...	36
ANNEX OO.A	– PROCESS OBJECTIVES AND OUTPUTS BY SOFTWARE LEVEL IN ED-12C	37
ANNEX OO.B	– ACRONYMS AND GLOSSARY OF TERMS	48
ANNEX OO.C	– PROCESS OBJECTIVES AND OUTPUTS BY ASSURANCE LEVEL IN ED-109A	53
ANNEX OO.D	–VULNERABILITY ANALYSIS	64
OO.D.1	Key Features and Related Techniques	64
OO.D.1.1	Inheritance	64
OO.D.1.1.1	Vulnerabilities.....	64
OO.D.1.1.2	Related Guidance	65
OO.D.1.1.3	Supporting Information for Activities	65
OO.D.1.2	Parametric Polymorphism	65
OO.D.1.2.1	Vulnerabilities.....	65
OO.D.1.2.2	Related Guidance	66
OO.D.1.2.3	Supporting Information for Activities	66
OO.D.1.3	Overloading	66
OO.D.1.3.1	Vulnerabilities.....	66
OO.D.1.3.2	Related Guidance	66
OO.D.1.3.3	Supporting Information for Activities	66
OO.D.1.4	Type Conversion	66
OO.D.1.4.1	Vulnerabilities.....	67
OO.D.1.4.2	Related Guidance	67

OO.D.1.4.3 Supporting Information for Activities	67
OO.D.1.5 Exception Management.....	67
OO.D.1.5.1 Vulnerabilities.....	67
OO.D.1.5.2 Related Guidance	67
OO.D.1.5.3 Supporting Information for Activities	67
OO.D.1.6 Dynamic Memory Management (DMM)	67
OO.D.1.6.1 Vulnerabilities.....	68
OO.D.1.6.2 Related Guidance	68
OO.D.1.6.3 Supporting Information for Activities	68
OO.D.1.7 Virtualization	70
OO.D.1.7.1 Vulnerabilities.....	70
OO.D.1.7.2 Related Guidance	70
OO.D.1.7.3 Supporting Information for Activities	70
OO.D.2 General Issues.....	70
OO.D.2.1 Traceability	71
OO.D.2.1.1 OOT&RT Specific Considerations	71
OO.D.2.1.2 Related Guidance	71
OO.D.2.1.3 Supporting Information for Activities	71
OO.D.2.2 Structural Coverage	71
OO.D.2.2.1 OOT&RT Specific Considerations	72
OO.D.2.2.2 Related Guidance	72
OO.D.2.2.3 Supporting Information for Activities	72
OO.D.2.3 Component-Based Development	73
OO.D.2.3.1 Vulnerabilities.....	73
OO.D.2.3.2 Related Guidance	74
OO.D.2.3.3 Supporting Information for Activities	74
OO.D.2.4 Resource Analysis.....	75
OO.D.2.4.1 Annotations and Static Analysis	76
OO.D.2.4.2 Memory Usage Analysis	76
OO.D.2.4.3 Timing Analysis.....	81
APPENDIX OO.A – COMMITTEE MEMBERSHIP	83
APPENDIX OO.B –FREQUENTLY ASKED QUESTIONS	94
OO.B.1 General Questions	94
FAQ #1: What is covered by this supplement?	94
FAQ #2: What is the relationship between this supplement and the OOTiA Handbook?	95
FAQ #3: What additional information is required for the software planning process in an OO related system?.....	95
FAQ #4: What additional measures may be taken when reusing OO components?.....	95
FAQ #5: What additional objectives and activities have been added to support OOT&RT in the software development processes? ...	95
FAQ #6: What additional objectives may be met and activities may be performed when reviewing and analyzing software using OO components?.....	96
FAQ #7: What are the vulnerabilities required to be addressed in section OO.11.1?.....	96
FAQ #8: Are there any special concerns for using closures?	96
OO.B.2 Requirements Considerations	97

FAQ #9: What is the relationship between low-level requirements and classes?.....	97
FAQ #10: What role do classes play in the design process?.....	97
FAQ #11: When should a requirement be expressed as invariants as opposed to preconditions and postconditions?	97
FAQ #12: Do all methods, including constructor, accessor, mutator, and destructor, need requirements?	98
OO.B.3 Design Considerations.....	98
FAQ #13: What is typing?	98
FAQ #14: Why is the Liskov Substitution Principle (LSP) important?.....	98
FAQ #15: Must all classes in an application be type consistent?	99
FAQ #16: What impact does the Liskov Substitution Principle (LSP) have on architecture and low-level requirements?	99
FAQ #17: Are there garbage collection techniques that should be avoided?.....	100
FAQ #18: Should inheritance be used for code sharing?	100
FAQ #19: What techniques can be used to minimize potential problems with the use of multiple inheritance?.....	100
FAQ #20: Is static dispatch safer than dynamic dispatch?	101
FAQ #21: How can Liskov Substitution Principle (LSP) be violated in a strongly-typed language, such as Ada or Java?	101
FAQ #22: What would need to be done to make the examples in FAQ #21 conform to the Liskov Substitution Principle (LSP)?	104
FAQ #23: Does the discussion on preconditions, postconditions, and invariants imply the use of a particular design methodology?	105
OO.B.4 Programming Language Considerations	105
FAQ #24: What is an example of static dispatch with method overriding?	105
FAQ #25: Is it acceptable to use a COTS library, such as the C++ Standard Template Library, without explicitly specifying high-level and low-level software requirements, tests, and documents for this library?.....	106
FAQ #26: What considerations are important when using garbage collection?.....	107
FAQ #27: What considerations are important when using a virtual machine?.....	107
OO.B.5 Verification Considerations	108
FAQ #28: What additional objectives and activities have been added to support OOT&RT in the verification process?.....	108
FAQ #29: In section OO.5.5.d, what is meant by "a requirement which traces to a method"?	108
FAQ #30: How does the formal verification activity for ensuring local type consistency differ from a low-level requirements review?.....	108
FAQ #31: Does the guidance related to dynamic dispatch also apply to the use of function pointers (for example, function pointer tables and callback functions)?	109
FAQ #32: What is the difference between fulfilling type consistency locally and globally?.....	109
FAQ #33: How can local type consistency be verified?	109
FAQ #34: How can local type consistency with an abstract superclass be verified by testing without using pessimistic testing?	110
FAQ #35: How can classes that include unencapsulated data be verified for local type consistency?.....	110
FAQ #36: Does this supplement require tracing of low-level requirements to preconditions, postconditions, and invariants?.....	111
FAQ #37: Is "flattened class testing" a way of verifying local type consistency?	111

FAQ #38: When preconditions, post conditions, and invariants are checked at run time, what verification must be performed on the additional Source Code?	112
FAQ #39: Is there a problem with covariant collections?	112

LIST OF TABLES

TABLE OO.A-1 SOFTWARE PLANNING PROCESS	38
TABLE OO.A-2 SOFTWARE DEVELOPMENT PROCESSES.....	39
TABLE OO.A-3 VERIFICATION OF OUTPUTS OF SOFTWARE REQUIREMENTS PROCESS	40
TABLE OO.A-4 VERIFICATION OF OUTPUTS OF SOFTWARE DESIGN PROCESS	41
TABLE OO.A-5 VERIFICATION OF OUTPUTS OF SOFTWARE CODING & INTEGRATION PROCESSES	42
TABLE OO.A-6 TESTING OF OUTPUTS OF INTEGRATION PROCESS	43
TABLE OO.A-7 VERIFICATION OF VERIFICATION PROCESS RESULTS.....	44
TABLE OO.A-8 SOFTWARE CONFIGURATION MANAGEMENT PROCESS	45
TABLE OO.A-9 SOFTWARE QUALITY ASSURANCE PROCESS.....	46
TABLE OO.A-10 CERTIFICATION LIAISON PROCESS	47
TABLE OO.C-1 SOFTWARE PLANNING PROCESS	54
TABLE OO.C-2 SOFTWARE DEVELOPMENT PROCESSES	55
TABLE OO.C-3 VERIFICATION OF OUTPUTS OF SOFTWARE REQUIREMENTS PROCESS.	56
TABLE OO.C-4 VERIFICATION OF OUTPUTS OF SOFTWARE DESIGN PROCESS	57
TABLE OO.C-5 VERIFICATION OF OUTPUTS OF SOFTWARE CODING & INTEGRATION PROCESSES	58
TABLE OO.C-6 TESTING OF OUTPUTS OF INTEGRATION PROCESS	59
TABLE OO.C-7 VERIFICATION OF VERIFICATION PROCESS RESULTS.....	60
TABLE OO.C-8 SOFTWARE CONFIGURATION MANAGEMENT PROCESS	61
TABLE OO.C-9 SOFTWARE QUALITY ASSURANCE PROCESS.....	62
TABLE OO.C-10 CERTIFICATION LIAISON PROCESS	63
TABLE OO.D.1.6.3 WHERE ACTIVITIES ARE ADDRESSED FOR DMM	69

OO.1.0

INTRODUCTION

Object-oriented technology (OOT) has been widely adopted in non-critical software development projects. The use of this technology for critical software applications in avionics has increased, but there are a number of issues that need to be considered to ensure the safety and integrity goals are met. These issues are both directly related to language features and to complications encountered with meeting well-established safety objectives. In fact, there are a number of related techniques that are used with OOT that need to be considered as well. Clarifying each of these issues will ease the application of object-oriented technology and related techniques (OOT&RT); therefore, this supplement addresses both object-oriented technology and related techniques.

OO.1.1

PURPOSE

The purpose of this supplement is to provide guidance for the production of software using OOT&RT for systems and equipment that performs its intended function with a level of confidence in safety that complies with airworthiness requirements. This supplement contains modifications and additions to ED-12C objectives, activities, explanatory text, and software life cycle data that should be addressed when OOT&RT are used as part of the software life cycle.

Compliance with the objectives of ED-12C, Software Considerations in Airborne Systems and Equipment Certification, is the primary means of obtaining approval of software used in civil aviation products. Object-oriented technology is a software development methodology that uses objects and the connections between those objects to express the software design. There are a number of related techniques commonly associated with, but not limited to, object-oriented technology that are considered in this supplement. These related techniques include parametric polymorphism, overloading, type conversion, exception management, dynamic memory management, and virtualization. Since OOT&RT differs from the traditional functional approach to software development, satisfying some of the ED-12C objectives when using OOT&RT may be unclear and/or complicated.

This supplement identifies the additions, modifications, and deletions to ED-12C objectives when object-oriented technology or related techniques are used as part of the software development life cycle and additional guidance is required. This supplement, in conjunction with ED-12C, is intended to provide a common framework for the evaluation and acceptance of OOT&RT based systems.

This supplement provides the following data:

- Objectives for OOT&RT software life cycle processes.
- Descriptions of activities and design considerations for achieving those objectives.
- Descriptions of the evidence that indicate that the objectives have been satisfied.

OO.1.2

SCOPE

This supplement discusses the use of OOT&RT in the software life cycle for software that is produced in accordance with ED-12C. If the applicant is planning to use OOT or a related technique, then the applicant should comply with the relevant parts of this supplement. The related techniques described in section OO.1.6.2 for which this document applies are parametric polymorphism, overloading, type conversion, exception management, dynamic memory management, and virtualization.

The related techniques discussed in this supplement may be used outside of OOT. Although the primary purpose of their inclusion in this supplement is in their relation to OOT, the guidance for related techniques should be used even when OOT is not used.

OO.1.3 RELATIONSHIP TO OTHER DOCUMENTS

This document supplements the guidance given in ED-12C. This supplement may be used in conjunction with other ED-12C supplements.

OO.1.4 HOW TO USE THIS SUPPLEMENT

This supplement should be used with, and in the same way as, ED-12C; however, the following should be noted:

- a. Section OO.1 contains explanatory text to aid the reader in understanding OOT&RT, and therefore is not to be taken as guidance.
- b. Annex OO.A of this supplement describes how the ED-12C objectives are revised in line with this OOT&RT guidance.
- c. Annex OO.D of this supplement describes vulnerabilities associated with OOT&RT, as well as supporting information for activities of sections OO.4 through OO.12.
- d. Guidance from ED-12C should be used for all aspects of the software life cycle, apart from where OOT&RT are used. All guidance specific to OOT&RT are contained within this supplement.
- e. This supplement can be used with ED-109A in the same manner as ED-12C. Applicants complying with ED-109A should be aware that the text within this supplement is based on ED-12C; thus, the ED-109A applicant should ensure that the proper ED-109A guidance is being addressed. Annex OO.C of this supplement describes how the ED-109A, Annex A objectives are revised in line with the guidance in this supplement.
- f. In addition to the guidance contained in this supplement, this supplement also provides clarification on the guidance provided in the form of answers to Frequently Asked Questions (FAQs) (see Appendix OO.B). FAQs contain no new or additional guidance.
- g. Major sections are numbered as OO.X.0 throughout this supplement. It should be noted that references to an entire section are identified as “section X”; whereas, references to the content between section headers OO.X.0 and OO.X.1 are referenced as “section OO.X.0”.

OO.1.5 DOCUMENT OVERVIEW

This supplement is intended to be used with ED-12C. Sections OO.4 through OO.12 and Annex OO.A provide additional and modified text to the referenced sections of ED-12C. The supplement minimizes any redundancy between ED-12C and the supplement itself. Each affected section of ED-12C is included in the supplement. Where guidance from ED-12C applies unchanged, this is stated in this supplement. Text taken directly from ED-12C is shown in italics. For those sections from ED-12C included in this supplement that are unchanged, except for the references, the ED-12C text is included with the updated OO reference.

In Annex OO.A, only those tables with changes to corresponding tables in ED-12C are included. This includes new objectives and changes to referenced text for existing objectives.

OO.1.6 CHARACTERISTICS OF OBJECT-ORIENTED TECHNOLOGY AND RELATED TECHNIQUES

Object-oriented technology is a paradigm for system analysis, design, modeling, and programming centered on objects. There are concepts and techniques commonly found in object-oriented languages that need to be taken into account when considering software safety. Most do not require new guidance per se, but all require consistent interpretation of guidance to ensure safety. Many of these concepts and techniques are also found in other high-level languages; therefore, the discussion here is also germane to any development methodology that uses these concepts and techniques.

OO.1.6.1 Basic Concepts

When considering OOT&RT, there should be understanding of both the technology itself and the techniques with which it is frequently combined. A thorough treatment of OOT&RT features and the underlying concepts would exceed the scope of this supplement.

OO.1.6.1.1 Classes and Objects

The feature that distinguishes OOT is the use of classes to define objects along with the ability to create new classes via subclassing. In procedural programming, a program's behavior is defined by functions, and the state of a running program is defined by the contents of the data variables. In object-oriented programming, functions and data that are closely related are tightly coupled to form a coherent abstraction known as a class. A superclass is a class from which other classes are derived. A superclass is also called a parent class. The classes that are derived from a superclass are known as child classes, derived classes, or subclasses.

A class is a blueprint from which multiple concrete realizations can be created. These concrete realizations are known as objects. Although the terms object and class are often used interchangeably, the distinction should be clear; classes define the types of data within an object and the subprograms that may act upon that data, whereas objects are the concrete manifestation of the class definition within the system. This process of object creation is known as instantiation, and objects are synonymously referred to as class instances or more succinctly as instances.

The subprograms defined for a class are referred to as its methods, or member functions. Methods that operate on or use the data contained within an object instance are referred to as instance methods. This is in contrast to class methods that are associated only with a class, and do not require an associated instance to be invoked. Some languages provide for methods and procedures. Throughout this supplement, the term subprogram is used to refer generically to instance methods, class methods, member functions, and procedures.

Data variables associated with a class are referred to as attributes, fields, or data members. An attribute may be classified as an instance attribute, in which case a separate copy of the attribute exists for each object, or it may be a class attribute, in which case a single, shared copy of the attribute exists for all objects of the class.

OO.1.6.1.2 Types and Type Safety

Given that each object has an associated class, it is natural to consider the type of an object to be its associated class; where type is a designation for a set of objects with well-defined common semantics. A set of types and the rules for their use constitute a type system. These types may be primitive or composite. Primitive types are atomic with regard to the underlying implementation, and usually constitute a predefined, finite set. Composite types, on the other hand, are assembled from primitive types and are usually extendable through new definitions. In object-oriented systems, the set of types is open; that is, the programmer may define new types by creating new classes. Being able to create new types increases the power of object-oriented languages, but it also means that the programmer should ensure that new types do not introduce type inconsistencies.

If types were disjoint and conversion between types were not allowed, maintaining type consistency, and hence type safety, would be trivial; however, developing a system using only disjoint types and preventing type conversions would eliminate much of the advantage of OOT. In OOT, there can be definition of subclasses of already existing classes. Since a class is also a type, a subclass is also a subtype. This means that the subclass should be type consistent with all its superclasses.

For a type to be a proper subtype of some other type, it should exhibit the same behavior as each of its supertypes. In other words, any subtype of a given type should be usable wherever the given type is required. If a subclass is defined that does not adhere to this principle of substitutability, then the type system becomes unsafe. Type checking would fail to detect a misuse of that subtype, since type systems generally allow the substitution of a subtype for any of the type that it refines.

Type consistency can be considered globally or locally. Global type consistency is type consistency throughout the entire class hierarchy of the system. Local type consistency is a subset of global type consistency. Local type consistency is type consistency in a local context where substitution can occur. A local context may be the context of a variable, an attribute, or a parameter. It is generally sufficient to consider local type consistency for type safety.

OO.1.6.1.2.1 Liskov Substitution Principle

The Liskov Substitution Principle (LSP) formally defines what constitutes a proper, that is, type safe, subclass. The principle is, "Let $q(x)$ be a property provable about objects x of type T . Then $q(y)$ should be true for objects y of type S where S is a subtype of T ." The LSP limits how a subclass may behave. This means that each subprogram redefined in the subclass should meet the following requirements of the same subprogram in any of its superclasses:

- Preconditions may not be strengthened,
- Postconditions may not be weakened, and
- Invariants may not be weakened.

In addition, the principle implies that no new exceptions should be thrown by subprograms of the subclass, except where those exceptions are themselves subtypes of exceptions thrown by the subprograms of the superclass. In terms of requirements-based verification, LSP means that any subtype of a given type should fulfill all the requirements of the given type. LSP may be demonstrated through the application of formal methods or thorough testing.

OO.1.6.1.3 Hierarchical Encapsulation

There are many problems where a hierarchical system of classification of component parts can aid system understanding. For instance, in tracking helicopters, planes, dirigibles, and gliders, it is helpful to have a classification "aircraft" that has a common set of attributes and behaviors. On the other hand, there may also be interest in those aircraft that need to use a runway and the attributes and behavior that goes with horizontal takeoff and landing aircraft. Classes and subclasses are defined to organize the objects in a system into a hierarchy of object types or classes.

Classes become a much more powerful tool in managing complexity when they are related to each other through a class hierarchy formed through a set of superclass-to-subclass relationships. A class hierarchy is a directed acyclic graph of superclass-to-subclass relationships between classes. A class hierarchy enables the development and manipulation of a potentially large set of subclasses based on the abstraction of a single superclass. Class hierarchies can reduce verification effort and improve comprehension, maintainability, and reuse.

Class hierarchies are formed through both hierarchic composition and hierarchic decomposition. Hierarchic composition is a categorization process involving the identification of common behavior between disparate classes, and the formation of more general superclasses that standardize and generalize those behaviors. Hierarchic decomposition is a categorization process involving the definition of superclasses with behavior that is then supported by subclasses, which add behavior while maintaining substitutability with the superclass. When developing class hierarchies, it is important to recognize that a class hierarchy implies more than software reuse. A class, as a type, is more than a collection of interfaces, properties, and subprograms. The preconditions, postconditions, and invariants for a superclass should be upheld by all of its subclasses. This is particularly important when modifying the class hierarchy.

Class hierarchies are generally implemented via the use of language-supported features such as inheritance, polymorphism, overloading, and varying degrees of type checking. These language features are necessary, but not sufficient, to ensure type safety of a program. It is still necessary to ensure functional consistency.

OO.1.6.1.4 Polymorphism

Polymorphism, which literally means “many forms”, is a way of applying a single abstraction to multiple situations, so that each abstraction can be more broadly applied. Polymorphism extends abstractions into implementation. As such, one of the primary goals of polymorphism is to maintain the simplicity of the underlying abstraction.

Traditionally, subprograms are monomorphic. A monomorphic subprogram has operands of unique types. Polymorphism generalizes this by enabling operands (actual parameters) to have more than one type. Polymorphic types may be defined as types whose operations are applicable to operands of more than one type.

Polymorphism has several variants. There are two broad categories: ad-hoc polymorphism and universal polymorphism. Ad-hoc polymorphism can be divided further into implicit coercion and overloading. Universal polymorphism can also be divided into parametric polymorphism and inclusion polymorphism. Each of these forms is provided by widely-used programming languages.

Implicit coercion is the automatic conversion of an operand of one type into another in order to match the required types of a function, in a situation that would otherwise result in a type error. A very common example of this is the transformation of integer values to floating-point before a floating-point arithmetic operation is performed. Another is the extending the range of an integral type; for example, converting a short into an integer value.

Overloading is the use of the same name or symbol for functions that take different arguments; for instance, using the plus operator for both adding numbers and concatenating strings. Another example is an add function with two arguments and an add function with three arguments.

A parametric polymorphic or generic function is a function that takes one or more types as arguments, treating the types uniformly. It provides the means by which the abstraction of common structural and behavioral aspects of a family of classes or operations can be encapsulated in a domain-independent manner. For example, the length function maps a list object of arbitrary type to an integer. It should not matter what type the list is or the types of its members: all lists have the generic abstraction of length. A generic function does the same kind of work independently of the types of its arguments.

Inclusion polymorphism is the ability to substitute any subtype for its supertype. This also means a subprogram of the supertype can be applied to any instances of its subtypes. Since a subclass is also a subtype, subclassing is an example of inclusion polymorphism.

A primary advantage of all forms of polymorphism is the sharing of operands of many types and raising the level of abstraction of the program.

OO.1.6.1.5 Function Passing and Closures

Function passing is a language feature that enables a function to be passed as an argument to another function. In essence, it allows for passing of behavior. A common function-passing example is sorting. Different sorting algorithms, such as quick sort, shell sort, and bubble sort, have been devised. Function passing provides an ability to specify the kind of sort at run-time.

In object-oriented languages, an object's method typically operates on, or uses, the associated object data. This tight binding between an object's method and data makes it impossible to execute a method without the companion object data. Consequently, using function passing to pass a method should also be accompanied by an additional parameter to include the associated object data.

A closure is a subprogram that is evaluated in an environment containing bound variables. During execution, the subprogram can access these variables. In some languages, a closure may also occur when a function is defined within another function, and the inner function refers to local variables of the outer function. At run-time, when the outer function executes, a closure is formed, consisting of the inner function's code and references to any variables of the outer function required by the closure. These local variables become a set of "private" variables, which persist over several invocations of the function. The scope of the variable encompasses only the closed-over function, so it cannot be accessed from other program code. As with the attributes of an object, the variables have indefinite extent, so a value established in one invocation remains available in the next. A closure is a means for implementing the method equivalent to function passing

Closures are useful for increasing the power of function passing. Languages that do not support closures can emulate function passing and closures through object passing. This usually involves creating some interface that includes a method to be called, and then passing an object that implements the given interface. The recipient can then call the predetermined method as a passed function would be called. Because of their flexibility and conciseness, closures are available in some object-oriented languages.

OO.1.6.1.6 Method Dispatch

Method dispatch is the mapping of a method call to its associated implementation. This mapping can be done statically based on the declared type of the object, or it can be done dynamically based on the actual type of the object. In addition, dynamic dispatch can be single, that is, on the implicit or a single argument, or multiple, that is, on multiple arguments to the method.

Static dispatch is normally implemented as a procedure call. It can be done safely only when the method in question is not redefined in a subclass of the static type. When a subtype of the static type redefines the method being called, the method of the static type (an ancestor type of the actual type) is still called. Since this can lead to unintended or erroneous behavior, some languages provide a means to ensure that statically dispatched methods cannot be overridden.

When a method call is dynamic dispatched, the mapping of a specific implementation is performed at run-time. Dynamic dispatching ensures that the method of the actual class is called even though the actual type cannot be determined statically (during compilation). Though this dispatching is analogous to a switch statement over all the subtypes of the static type, it is generally not implemented this way, since to do so would require recompilation every time a new subclass of the static type is added to the system. Rather, each method is associated with an offset in the method table for the class of the object, so dispatching is just a simple indirect procedure call. Dynamic dispatch is only used for code invocation and not for other binding processes such as accessing object attributes directly.

When the decision as to which method to call is made only on the first or implicit argument, for example, the *x* in *x.write(stream)*, the technique is known as single dispatch. Single dispatch is the most common form of dynamic dispatch because it is both conceptually simpler and easier to implement efficiently. There are some languages that dispatch based on the types of all arguments, for example, both *x* and *stream*. This is known as multiple dispatching. There may be expanded verification activities as a result of the decision to use multiple dispatch.

OO.1.6.2 Key Features and Related Techniques

Besides these basic concepts, there are a number of features commonly provided in object-oriented languages. The division between concepts and features is somewhat arbitrary. In this supplement, features denote object-oriented or related technique elements that have direct safety considerations.

OO.1.6.2.1 Inheritance

Inheritance is a language feature in which a class shares properties defined in another class, where the defining class is considered to be the parent. The parent's properties that the inheriting class shares depend on the kind of inheritance used. With interface inheritance, the inheriting class shares only the complete signatures of the methods of its parent without the underlying implementation. On the other hand, with implementation inheritance, the inheriting class shares the implementation as well as the complete method signatures of its parent.

With both interface and implementation inheritance, an inheriting class may have either a single parent or more than one parent. The former case is referred to as single inheritance, the latter as multiple inheritance. Depending on the specific language, several combinations may be allowed, such as using single implementation inheritance while using multiple interface inheritance.

Inheritance enables substitution through subtyping, which means that where an object of a specific class is expected, an object of any of its subclasses can be provided. In the same way, polymorphism occurs when one object is used in the context of multiple declared classes. For example, a superclass may be substituted by one of several subclasses which all inherit the interface of the superclass.

OO.1.6.2.2 Parametric Polymorphism

Parametric polymorphism is a feature commonly used for enhancing the reusability of software components. For example, consider a stack with a generically parameterized type. Such a stack can be instantiated in any number of different forms such as a stack of integers, a stack of a user-defined type, or even a stack of stacks. Many languages support parametric polymorphism: as templates in C++ and Unified Modeling Language (UML), and as generics in Ada, Java, and Eiffel.

OO.1.6.2.3 Overloading (Ad Hoc Polymorphism)

Languages that support overloading enable the reuse of subprogram names with different implementations. The intent behind overloading is to assign the same name or operator symbols to subprograms that are similar semantically. A compiler will select the appropriate subprogram by matching its signature with the types of arguments at the call site.

OO.1.6.2.4 Type Conversion

Type conversion is the act of changing the representation of some value of a source type to the representation of the same value of a target type. Type conversion can be made explicitly by the programmer or implicitly by the compiler according to some predefined type conversion rules. For checked types, the conversion is verified at compile time or run-time if type conversion is possible. Unchecked types are not verified.

In general, there are two categories of type conversion when considering object-oriented languages: a type conversion that changes the underlying structure of the data that represents the type, and a type conversion that changes the view.

A structural change in which the destination data element contains sufficient precision and magnitude to represent the source data is a widening conversion. Conversely, a structural change in which the destination data element does not contain sufficient precision and magnitude to represent the source data is a narrowing conversion.

Type conversions that require changing the view of the underlying representation are more prevalent in OOT. The change of view can either be from a subtype to a supertype (upcast), or from a supertype to a subtype (downcast). The change of view from a subtype to a supertype is allowed by the type system and usually implicit. The change of view from a supertype to a subtype should be performed with care and is generally unsafe without sufficient precautions.

Type conversions may also be categorized as implicit or explicit. An implicit type conversion is an automatic type conversion performed by the compiler, whereas an explicit type conversion is performed based on information supplied by the programmer. Although languages and their compiler implementations generally have strict implicit conversion rules, a lack of familiarity of these rules, or ignorance of the conversion taking place are often the cause of coding problems.

OO.1.6.2.5 Software Exceptions And Exception Handling

Exceptions signal abnormal (error) conditions which are detected within a method. The ability to throw exceptions within a method and to handle exceptions in the calling method is a common feature of most object-oriented languages.

Exceptions may be raised implicitly through compiler-generated checks, explicit checks such as failure of preconditions and postconditions, or explicitly via user actions. Exception handling is used for conditions that deviate from the normal flow of execution. When an exception is raised the execution will transfer to the nearest enclosing handler.

OO.1.6.2.6 Dynamic Memory Management

In many languages, objects can be created on demand during run-time. The deletion of such objects after their use is either performed explicitly using a program statement or automatically by the run-time system.

During the creation of an object, memory is allocated from a memory pool. This memory is freed when the object is deleted. The management of the memory that can be dynamically allocated and deallocated is called dynamic memory management.

Four main techniques for managing dynamic memory are described in the following sections.

OO.1.6.2.6.1 Object Pooling

An initial set of unused objects of a particular type is provided. Each time a new object of the given type is needed, one is taken from the pool. Once an object is no longer needed, it is returned to the pool.

OO.1.6.2.6.2 Activation Frame Management

For objects that are needed only in certain execution contexts (for example, only in a given method and its called methods) allocation within the activation frame can be used. This method may be implemented directly on the stack or indirectly through secondary structures such as scoped memory or a secondary stack.

OO.1.6.2.6.3 Manual Heap Management

Manual heap-based object management requires the user to deallocate each object when and only when the object is no longer referenced. The application developer has to ensure that fragmentation does not cause allocations to fail.

OO.1.6.2.6.4 Automatic Heap Management

Garbage collection is the standard form of automatic heap management. A precise garbage collector ensures that objects that are no longer referenced within the program are deleted.

OO.1.6.2.7 Virtualization Techniques

Virtualization is the general technique of using software (or hardware) to abstract a set of resources at some higher level. Examples include microcode, operating systems, math libraries (abstract access to a floating-point unit), device drivers (abstract access to a physical device), virtual machine managers, and interpreters.

Some virtualization techniques, such as interpreters, are programs that emulate the execution of instructions of a given language. Interpreters can be as simple as an Extensible Markup Language (XML) or a state machine interpreter within an application, or as complex as a programming language interpreter.

The term interpreter should not be taken to mean all software that processes data. For example, making decisions based on the contents of a data packet header or use of parameter data items are not programs processed by such interpreters.

OO.2.0

SYSTEM ASPECTS RELATING TO SOFTWARE DEVELOPMENT

Section 2 of ED-12C is unchanged.

OO.3.0

SOFTWARE LIFE CYCLE

Section 3 of ED-12C is unchanged.

OO.4.0

SOFTWARE PLANNING PROCESS

This section discusses the objectives and activities of the software planning process. This process produces the software plans and standards that direct the software development processes and the integral processes. Table OO.A-1 of Annex OO.A is a summary of the objectives and outputs of the software planning process by software level.

The applicant should ensure that the intended use of this supplement is acceptable to the appropriate certification authority. Several supplements may be applied to an applicant's software life cycle. The applicant should plan an approach that will comply with the objectives of ED-12C and those applicable objectives from the supplements. As such, all relevant supplements and the supplements' associated objectives should be considered. If the applicant's particular approach identifies a conflict between the objectives, then the applicant should propose a solution to the conflict in the Plan for Software Aspects of Certification (PSAC). The applicant may use a single set of software plans or multiple sets of software plans addressing the different supplements. Using a single set of software plans containing sections addressing each supplement, will provide an integrated approach with a clear and comprehensive plan.

As supplements only extend guidance from ED-12C for a specific technology, the applicant should first consider all of the ED-12C objectives, and then the supplement's objectives, and lastly any other additional considerations. One approach is to consolidate all objectives of ED-12C and supplements, and for each objective provide a statement of how compliance will be achieved, along with identifying any applicable life cycle data items. Because the objective numbering scheme is unique between the Annex A Tables of ED-12C and the Annex A Tables of the supplements, the identification of the objectives and their document sources will be clear and unambiguous.

For the purpose of this supplement, the term target computer can be interpreted to mean target environment. The target environment is either a target computer or a combination of virtualization software and a target computer. Virtualization software also should comply with ED-12C and applicable supplements.

OO.4.1

SOFTWARE PLANNING PROCESS OBJECTIVES

The purpose of the software planning process is to define the means of producing software that will satisfy its requirements and provide the level of confidence that is consistent with the software level. There are seven objectives of the software planning process.

- a. *The activities of the software development processes and integral processes of the software life cycle that will address the system requirements and software level(s) are defined (see OO.4.2).*
- b. *The software life cycle(s), including the inter-relationships between the processes, their sequencing, feedback mechanisms, and transition criteria are determined (see ED-12C 3).*
- c. *The software life cycle environment, including the methods and tools to be used for the activities of each software life cycle process has been selected and defined (see OO.4.4).*
- d. *Additional considerations, such as those discussed in ED-12C section 12, have been addressed, if necessary.*
- e. *Software development standards consistent with the system safety objectives for the software to be produced are defined (see ED-12C 4.5).*
- f. *Software plans that comply with sections ED-12C 4.3 and OO.11 have been produced.*
- g. *Development and revision of the software plans are coordinated (see ED-12C 4.3).*

OO.4.2 SOFTWARE PLANNING PROCESS ACTIVITIES

Effective planning is a determining factor in producing software that satisfies the guidance of this document. Activities for the software planning process include:

- a. *The software plans should be developed that provide direction to the personnel performing the software life cycle processes. See also ED-12C section 9.1.*
- b. *The software development standards to be used for the project should be defined or selected.*
- c. *Methods and tools should be chosen that aid error prevention and provide defect detection in the software development processes.*
- d. *The software planning process should provide coordination between the software development and integral processes to provide consistency among strategies in the software plans.*
- e. *The means should be specified to revise the software plans as a project progresses.*
- f. *When multiple-version dissimilar software is used in a system, the software planning process should choose the methods and tools to achieve dissimilarity necessary to satisfy the system safety objectives.*
- g. *For the software planning process to be complete, the software plans and software development standards should be under change control and reviews of them completed (see OO.4.6).*
- h. *If deactivated code is planned, the software planning process should describe how the deactivation mechanism and deactivated code will be defined and verified to satisfy system safety objectives.*
- i. *If user-modifiable software is planned, related processes, tools, environment, and data items substantiating the design (see ED-12C 5.2.3) should be specified in the software plans and standards.*
- j. *When parameter data items are planned, the following should be addressed:*
 1. *The way parameter data items are used.*
 2. *The software level of the parameter data items.*
 3. *The processes to develop, verify, and modify parameter data items, and any associated tool qualification.*
 4. *Software load control and compatibility.*
- k. *The software planning process should address any additional considerations that are applicable.*
- l. *If software development activities will be performed by a supplier, planning should address supplier oversight.*
- m. *Describe any planned use of virtualization and its associated executable program(s). Any time that data is interpreted as a set of instructions providing flow control, rather than as a set of values to be processed, virtualization is being used. This data should be treated as executable code and all applicable objectives should be satisfied. See Annex OO.D.1.7.1, for vulnerabilities associated with some virtualization techniques.*
- n. *If the reuse of components is planned, describe how the component will be integrated with the new development. This includes the maintenance of type consistency, requirements mapping, and exception management strategy between the components and the using system. See Annex OO.D.1.1.1, OO.D.1.5.1, and OO.D.2.3.1 for inheritance, exception management, and component-based development vulnerabilities respectively.*

Other software life cycle processes may begin before completion of the software planning process if transition criteria for the specific process activity are satisfied.

OO.4.3 SOFTWARE PLANS

The software plans define the means of satisfying the objectives of ED-12C and other applicable documents. They specify the organizations that will perform those activities. The software plans are as follows:

- *The Plan for Software Aspects of Certification (section OO.11.1) serves as the primary means for communicating the proposed development methods to the certification authority for agreement, and defines the means of compliance. The Plan for Software Aspects of Certification gives an overview of how and where OOT&RT will be used and the means that will be used to address vulnerabilities and OOT&RT specific considerations.*
- *The Software Development Plan (ED-12C section 11.2) defines the software life cycle(s) and software development environment, and the means by which the software development process objectives will be satisfied. The Software Development Plan details the precise development steps that will include the use of OOT&RT. Software life cycle data of section OO.11.2 is unchanged from ED-12C section 11.2.*
- *The Software Verification Plan (ED-12C section 11.3) defines the means by which the software verification process objectives will be satisfied. The Software Verification Plan details the methods used to gather the verification evidence. Software life cycle data of section OO.11.3 is unchanged from ED-12C section 11.3.*

The remainder of ED-12C section 4.3 is unchanged.

OO.4.4 SOFTWARE LIFE CYCLE ENVIRONMENT PLANNING

Planning for the software life cycle environment defines the methods, tools, procedures, programming languages, and target computer that will be used to develop, verify, control, and produce the software life cycle data (see OO.11) and software product. Examples of how the software environment chosen can have a beneficial effect on the software include enforcing standards, detecting errors, and implementing error prevention and fault tolerance methods. The software life cycle environment is a potential error source that can contribute to failure conditions. Composition of this software life cycle environment may be influenced by the safety-related requirements determined by the system safety assessment process, for example, the use of dissimilar, redundant components.

The goal of error prevention methods is to avoid errors during the software development processes that might contribute to a failure condition. The basic principle is to choose requirements development and design methods, tools, and programming languages that limit the opportunity for introducing errors, and verification methods that ensure that errors introduced are detected. The goal of fault tolerance methods is to include safety features in the software design or Source Code to ensure that the software will respond correctly to input data errors and prevent output and control errors. The need for error prevention or fault tolerance methods is determined by the system requirements and the system safety assessment process.

The considerations presented above may affect:

- a. *The methods and notations used in the software requirements process and software design process.*
- b. *The programming language(s) and methods used in the software coding process.*
- c. *The software development environment tools.*
- d. *The software verification and software configuration management tools.*
- e. *The need for tool qualification (see ED-12C 12.2).*
- f. *The assessment of software component reuse and the availability of associated life cycle data.*
- g. *The inclusion of virtualization software within the target environment.*

OO.4.4.1 Software Development Environment

Section 4.4.1 of ED-12C is unchanged.

OO.4.4.2 Language and Compiler Considerations

Upon successful completion of verification of the software product, the compiler is considered acceptable for that product. For this to be valid, the software verification process needs to consider particular features of the programming language and compiler. The software planning process considers these features when choosing a programming language and planning for verification. Activities include:

- a. *Some compilers have features intended to optimize performance of the object code. If the test cases give coverage consistent with the software level, the correctness of the optimization need not be verified. Otherwise, the impact of these features on structural coverage analysis should be determined. Additional information can be found in section 6.4.4.2 of ED-12C.*
- b. *To implement certain features, compilers for some languages may produce object code that is not directly traceable to the Source Code, for example, initialization, built-in error detection, or exception handling (see ED-12C 6.4.4.2.b). The software planning process should provide a means to detect this object code and to ensure verification coverage, and should define the means in the appropriate plan.*
- c. *If a new compiler, linkage editor, or loader version is introduced, or compiler options are changed during the software life cycle, previous tests and coverage analyses may no longer be valid. The verification planning should provide a means of reverification that is consistent with sections OO.6 and ED-12C 12.1.3.*

NOTE: *Although the compiler is considered acceptable once all of the verification objectives are satisfied, the compiler is only considered acceptable for that product and not necessarily for other products.*

OO.4.4.3 Software Test Environment

Section 4.4.3 of ED-12C is unchanged.

OO.4.5 SOFTWARE DEVELOPMENT STANDARDS

Software development standards define the rules and constraints for the software development processes. The software development standards include the Software Requirements Standards, the Software Design Standards, and the Software Code Standards. The software verification process uses these standards as a basis for evaluating the compliance of actual outputs of a process with intended outputs. Activities for development of the software standards include:

- a. *The software development standards should comply with section OO.11.*
- b. *The software development standards should enable software components of a given software product or related set of products to be uniformly designed and implemented.*
- c. *The software development standards should disallow the use of constructs or methods that produce outputs that cannot be verified or that are not compatible with safety-related requirements.*
- d. *Robustness should be considered in the software development standards.*

NOTE 1: *In developing standards, consideration can be given to previous experience. Constraints and rules on development, design, and coding methods can be included to control complexity. Defensive programming practices may be considered to improve robustness.*

NOTE 2: *If allocated to software by system requirements, practices to detect and control errors in stored data, and refresh and monitor hardware status and configuration may be used to mitigate single event upsets.*

OO.4.6 REVIEW OF THE SOFTWARE PLANNING PROCESS

Reviews of the software planning process are conducted to ensure that the software plans and software development standards comply with the guidance of this document and means are provided to execute them. Activities include:

- a. Methods are chosen that enable the objectives of this document to be satisfied.*
- b. Software life cycle processes can be applied consistently.*
- c. Each process produces evidence that its outputs can be traced to their activity and inputs, showing the degree of independence of the activity, the environment, and the methods to be used.*
- d. The outputs of the software planning process are consistent and comply with section OO.11.*

OO.5.0

SOFTWARE DEVELOPMENT PROCESSES

This section discusses the objectives and activities of the software development processes. The software development processes are applied as defined by the software planning process (see OO.4) and the Software Development Plan (see ED-12C 11.2). Table OO.A-2 of Annex OO.A is a summary of the objectives and outputs of the software development processes by software level. The software development processes are:

- *Software requirements process.*
- *Software design process.*
- *Software coding process.*
- *Integration process.*

Software development processes produce one or more levels of software requirements. High-level requirements are produced directly through analysis of system requirements and system architecture. Usually, these high-level requirements are further developed during the software design process, thus producing one or more successive, lower levels of requirements. However, if Source Code is generated directly from high-level requirements, then the high-level requirements are also considered low-level requirements and the guidance for low-level requirements also apply.

NOTE: *The applicant may be required to justify software development processes that produce a single level of requirements.*

The development of software architecture involves decisions made about the structure of the software. During the software design process, the software architecture is defined and low-level requirements are developed. Low-level requirements are software requirements from which Source Code can be directly implemented without further information.

Each software development process may produce derived requirements. Some examples of requirements that might be determined to be derived requirements are:

- *The need for interrupt handling software to be developed for the chosen target computer.*
- *The specification of a periodic monitor's iteration rate when not specified by the system requirements allocated to software.*
- *The addition of scaling limits when using fixed point arithmetic.*

High-level requirements may include derived requirements, and low-level requirements may include derived requirements. In order to determine the effects of derived requirements on the system safety assessment and system requirements, all derived requirements should be made available to the system processes including the system safety assessment process.

OO.5.1 SOFTWARE REQUIREMENTS PROCESS

Section 5.1 of ED-12C is unchanged.

OO.5.1.1 Software Requirements Process Objectives

Section 5.1.1 of ED-12C is unchanged.

OO.5.1.2 Software Requirements Process Activities

Section 5.1.2 of ED-12C is unchanged.

OO.5.2 SOFTWARE DESIGN PROCESS

Section 5.2 of ED-12C is unchanged.

OO.5.2.1 Software Design Process Objectives

Section 5.2.1 of ED-12C is unchanged.

OO.5.2.2 Software Design Process Activities

The software design process inputs are the Software Requirements Data, the Software Development Plan, and the Software Design Standards. When the planned transition criteria have been satisfied, the high-level requirements are used in the design process to develop software architecture and low-level requirements. This may involve one or more lower levels of requirements.

The primary output of the process is the Design Description (see ED-12C 11.10) which includes the software architecture and the low-level requirements.

The software design process is complete when its objectives and the objectives of the integral processes associated with it are satisfied. Activities for this process include:

- a. *Low-level requirements and software architecture developed during the software design process should conform to the Software Design Standards and be traceable, verifiable, and consistent.*
- b. *Derived low-level requirements and the reason for their existence should be defined and analyzed to ensure that the higher level requirements are not compromised.*
- c. *Software design process activities could introduce possible modes of failure into the software or, conversely, preclude others. The use of partitioning or other architectural means in the software design may alter the software level assignment for some components of the software. In such cases, additional data should be defined as derived requirements and provided to the system processes, including the system safety assessment process.*
- d. *Interfaces between software components, in the form of data flow and control flow, should be defined to be consistent between the components.*
- e. *Control flow and data flow should be monitored when safety-related requirements dictate, for example, watchdog timers, reasonableness-checks, and cross-channel comparisons.*
- f. *Responses to failure conditions should be consistent with the safety-related requirements.*
- g. *Inadequate or incorrect inputs detected during the software design process should be provided to the system life cycle processes, the software requirements process, or the software planning process as feedback for clarification or correction.*
- h. *Class hierarchy should be developed based on high-level requirements.*
- i. *A locally type-consistent class hierarchy should be developed with associated low-level requirements wherever substitution is relied upon.*
- j. *As part of the software architecture, strategy for memory management should be developed. See Annex OO.D.1.6.1 for vulnerabilities.*
- k. *As part of the software architecture, strategy for exception management should be developed. See Annex OO.D.1.5.1 for vulnerabilities.*
- l. *When reusing components, the resulting derived requirements and any functionality to be deactivated should be identified. See Annex OO.D.2.3.1 for component-based development vulnerabilities.*

See Annex OO.D.1.1.1 for inheritance vulnerabilities.

NOTE: *The current state of software engineering does not permit a quantitative correlation between complexity and the attainment of system safety objectives. While no objective guidance can be provided, the software design process should avoid introducing complexity because as the complexity of software increases, it becomes more difficult to verify the design and to show that the safety-related requirements are satisfied.*

OO.5.2.3 Designing For User-Modifiable Software

Section 5.2.3 of ED-12C is unchanged.

OO.5.2.4 Designing For Deactivated Code

Section 5.2.4 of ED-12C is unchanged.

OO.5.3 SOFTWARE CODING PROCESS

Section 5.3 of ED-12C is unchanged.

OO.5.3.1 Software Coding Process Objectives

Section 5.3.1 of ED-12C is unchanged.

OO.5.3.2 Software Coding Process Activities

Section 5.3.2 of ED-12C is unchanged.

OO.5.4 INTEGRATION PROCESS

Section 5.4 of ED-12C is unchanged.

OO.5.4.1 Integration Process Objectives

Section 5.4.1 of ED-12C is unchanged.

OO.5.4.2 Integration Process Activities

Section 5.4.2 of ED-12C is unchanged.

OO.5.5 SOFTWARE DEVELOPMENT PROCESS TRACEABILITY

Software development process traceability activities include:

- a. *Trace Data, showing the bi-directional association between system requirements allocated to software and high-level requirements, is developed. The purpose of this Trace Data is to:*
 1. *Enable verification of the complete implementation of the system requirements allocated to software.*
 2. *Give visibility to those derived high-level requirements that are not directly traceable to system requirements.*
- b. *Trace Data, showing the bi-directional association between the high-level requirements and low-level requirements is developed. The purpose of this Trace Data is to:*
 1. *Enable verification of the complete implementation of the high-level requirements.*
 2. *Give visibility to those derived low-level requirements that are not directly traceable to high-level requirements and to the architectural design decisions made during the software design process.*
- c. *Trace Data, showing the bi-directional association between low-level requirements and Source Code, is developed. The purpose of this Trace Data is to:*
 1. *Enable verification that no Source Code implements an undocumented function.*
 2. *Enable verification of the complete implementation of the low-level requirements*
- d. *In an object-oriented design, all functionality is implemented in methods; therefore, traceability is from requirements to the methods and attributes that implement the requirements. Classes are an artifact of the architecture for organizing the requirements. Due to subclassing, a requirement, which traces to a method implemented in a class, should also trace to the method in its subclasses when the method is overridden in a subclass. This is in addition to tracing requirements that are specific to the subclass. See Annex OO.D.2.1.1 for supporting information on traceability.*

OO.6.0

SOFTWARE VERIFICATION PROCESS

This section discusses the objectives and activities of the software verification process. Verification is a technical assessment of the outputs of the software planning process, software development processes, and the software verification process. The software verification process is applied as defined by the software planning process (see OO.4) and the Software Verification Plan (see ED-12C 11.3). See OO.4.6 for the verification of the outputs of the planning process.

Verification is not simply testing. Testing, in general, cannot show the absence of errors. As a result, the following sections use the term "verify" instead of "test" to discuss the software verification process activities, which are typically a combination of reviews, analyses, and tests.

Tables OO.A-3 through OO.A-7 of Annex OO.A contain a summary of the objectives and outputs of the software verification process, by software level.

NOTE: *For lower software levels, less emphasis is on:*

- *Verification of Source Code.*
- *Verification of low-level requirements.*
- *Verification of the software architecture.*
- *Degree of test coverage.*
- *Control of verification procedures.*
- *Independence of software verification process activities.*
- *Overlapping software verification process activities, that is, multiple verification activities, each of which may be capable of detecting the same class of error.*
- *Robustness testing.*
- *Verification activities with an indirect effect on error prevention or detection, for example, conformance to software development standards.*

OO.6.1 PURPOSE OF SOFTWARE VERIFICATION

Section 6.1 of ED-12C is unchanged.

OO.6.2 OVERVIEW OF SOFTWARE VERIFICATION PROCESS ACTIVITIES

Software verification process objectives are satisfied through a combination of reviews, analyses, the development of test cases and procedures, and the subsequent execution of those test procedures. Reviews and analyses provide an assessment of the accuracy, completeness, and verifiability of the software requirements, software architecture, and Source Code. The development of test cases and procedures may provide further assessment of the internal consistency and completeness of the requirements. The execution of the test procedures provides a demonstration of compliance with the requirements.

The inputs to the software verification process include the system requirements, the software requirements, software architecture, Trace Data, Source Code, Executable Object Code, and the Software Verification Plan.

The outputs of the software verification process are recorded in the Software Verification Cases and Procedures (see ED-12C 11.13), the Software Verification Results (see OO.11.14), and the associated Trace Data (see ED-12C 11.21).

The need for the requirements to be verifiable once they have been implemented in the software may itself impose additional requirements or constraints on the software development processes.

Software verification considerations include:

- a. *If the code tested is not identical to the airborne software, those differences should be specified and justified.*
- b. *When it is not possible to verify specific software requirements by exercising the software in a realistic test environment, other means should be provided and their justification for satisfying the software verification process objectives defined in the Software Verification Plan or Software Verification Results.*
- c. *Deficiencies and errors discovered during the software verification process should be reported to other software life cycle processes for clarification and correction as applicable.*
- d. *Reverification should be conducted following corrective actions and/or changes that could impact the previously verified functionality. Reverification should ensure that the modification has been correctly implemented.*
- e. *Verification independence is achieved when the verification activity is performed by a person(s) other than the developer of the item being verified. A tool may be used to achieve equivalence to the human verification activity. For independence, the person who created a set of low-level requirements-based test cases should not be the same person who developed the associated Source Code from those low-level requirements.*
- f. *The class hierarchy should be verified for consistency with requirements.*
- g. *Local type consistency should be verified.*
- h. *The memory management system should be verified for consistency with the strategy defined in the software architecture (see OO.5.2.2.j).*
- i. *The exception management system should be verified for consistency with the strategy defined in the software architecture (see OO.5.2.2.k and Annex OO.D.1.5.1 for vulnerabilities).*

OO.6.3 SOFTWARE REVIEWS AND ANALYSES

Section 6.3 of ED-12C is unchanged.

OO.6.3.1 Reviews and Analyses of High-Level Requirements

Section 6.3.1 of ED-12C is unchanged.

OO.6.3.2 Reviews and Analyses of Low-Level Requirements

Section 6.3.2 of ED-12C is unchanged.

OO.6.3.3 Reviews and Analyses of Software Architecture

These review and analysis activities detect and report errors that may have been introduced during the development of the software architecture. These review and analysis activities confirm that the software architecture satisfies these objectives:

- a. *Compatibility with the high-level requirements:* *The objective is to ensure that the software architecture does not conflict with the high-level requirements, especially functions that ensure system integrity, for example, partitioning schemes, exception management, and memory management. See Annex OO.D.1.5.1 and OO.D.1.6.1 for exception and memory management vulnerabilities respectively.*
- b. *Consistency:* *The objective is to ensure that a correct relationship exists between the components of the software architecture. This relationship exists via data flow, control flow, and inheritance within the class hierarchy of the software architecture. If the interface is to a component of a lower software level, it should also be confirmed that the higher software level component has appropriate protection mechanisms in place to protect itself from potential erroneous inputs from the lower software level component.*
- c. *Compatibility with the target computer:* *The objective is to ensure that no conflicts exist, especially initialization, asynchronous operation, synchronization, and interrupts, between the software architecture and the hardware/software features of the target computer.*

- d. Verifiability: The objective is to ensure that the software architecture can be verified, for example, there are no unbounded recursive algorithms.
- e. Conformance to standards: The objective is to ensure that the Software Design Standards were followed during the software design process and that deviations to the standards are justified, for example, deviations to complexity restriction and design construct rules.
- f. Partitioning integrity: The objective is to ensure that partitioning breaches are prevented.

OO.6.3.4 Reviews and Analyses of Source Code

These review and analysis activities detect and report errors that may have been introduced during the software coding process. Primary concerns include correctness of the code with respect to the software requirements and the software architecture, and conformance to the Software Code Standards. These review and analysis activities are usually confined to the Source Code and confirm that the Source Code satisfies these objectives:

- a. Compliance with the low-level requirements: The objective is to ensure that the Source Code is accurate and complete with respect to the low-level requirements and that no Source Code implements an undocumented function.
- b. Compliance with the software architecture: The objective is to ensure that the Source Code matches the data flow, control flow, and class hierarchy defined in the software architecture.
- c. Verifiability: The objective is to ensure the Source Code does not contain statements and structures that cannot be verified and that the code does not have to be altered to test it.
- d. Conformance to standards: The objective is to ensure that the Software Code Standards were followed during the development of the code, for example, complexity restrictions and code constraints. Complexity includes the degree of coupling between software components, the nesting levels for control structures, and the complexity of logical or numeric expressions. This analysis also ensures that deviations to the standards are justified.
- e. Traceability: The objective is to ensure that the low-level requirements were developed into Source Code.
- f. Accuracy and consistency: The objective is to determine the correctness and consistency of the Source Code, including type consistency, type conversion, stack usage, memory usage, fixed point arithmetic overflow and resolution, floating-point arithmetic, resource contention and limitations, worst-case execution timing, exception handling, use of uninitialized variables, cache management, unused variables, and data corruption due to task or interrupt conflicts. The compiler (including its options), the linker (including its options), and some hardware features may have an impact on the worst-case execution timing and this impact should be assessed.

See Annex OO.D.1.3.1 and OO.D.1.4.1 for overloading and type conversion vulnerabilities respectively.

OO.6.3.5 Reviews and Analyses of The Outputs of The Integration Process

Section 6.3.5 of ED-12C is unchanged.

OO.6.4 SOFTWARE TESTING

Section 6.4 of ED-12C is unchanged.

OO.6.4.1 Test Environment

Section 6.4.1 of ED-12C is unchanged.

OO.6.4.2 Requirements-Based Test Selection

Section 6.4.2 of ED-12C is unchanged.

OO.6.4.2.1 Normal Range Test Cases

Normal range test cases demonstrate the ability of the software to respond to normal inputs and conditions. Activities include:

- a. Real and integer input variables should be exercised using valid equivalence classes and boundary values.*
- b. For time-related functions, such as filters, integrators, and delays, multiple iterations of the code should be performed to check the characteristics of the function in context.*
- c. For state transitions, test cases should be developed to exercise the transitions possible during normal operation.*
- d. For software requirements expressed by logic equations, the normal range test cases should verify the variable usage and the Boolean operators.*
- e. For object-oriented software, the normal range test cases should ensure that class constructors properly initialize the state of their objects and that the initial state is consistent with the requirements for the class.*

OO.6.4.2.2 Robustness Test Cases

Section 6.4.2.2 of ED-12C is unchanged.

OO.6.4.3 Requirements-Based Testing Methods

Section 6.4.3 of ED-12C is unchanged.

See Annex OO.D.2.4 for resource analysis supporting information.

OO.6.4.4 Test Coverage Analysis

Section 6.4.4 of ED-12C is unchanged.

OO.6.4.4.1 Requirements-Based Test Coverage Analysis

Section 6.4.4.1 of ED-12C is unchanged.

OO.6.4.4.2 Structural Coverage Analysis

Section 6.4.4.2 of ED-12C is unchanged.

See Annex OO.D.2.2.3 for structural coverage supporting information.

OO.6.4.4.3 Structural Coverage Analysis Resolution

Section 6.4.4.3 of ED-12C is unchanged.

OO.6.4.5 Reviews and Analyses of Test Cases, Procedures, and Results

Section 6.4.5 of ED-12C is unchanged.

OO.6.5 SOFTWARE VERIFICATION PROCESS TRACEABILITY

Section 6.5 of ED-12C is unchanged.

OO.6.6 VERIFICATION OF PARAMETER DATA ITEMS

Section 6.6 of ED-12C is unchanged.

OO.6.7 LOCAL TYPE CONSISTENCY VERIFICATION

The use of inheritance with method overriding and dynamic dispatch requires additional verification activities that should be done either by testing or by analysis using formal methods.

OO.6.7.1 Local Type Consistency Verification Objective

Verify that all type substitutions are safe.

OO.6.7.2 Local Type Consistency Verification Activity

For each subtype where substitution is used, perform one of the following activities (see Annex OO.D.1.1.1 for inheritance vulnerabilities):

- Verify substitutability using formal methods.
- Ensure that each class passes all the tests of all its parent types which the class can replace.
- For each call point, test every method that can be invoked at that call point (pessimistic testing).

OO.6.8 DYNAMIC MEMORY MANAGEMENT VERIFICATION

Object-oriented programming tends to rely on dynamic memory management. There are several vulnerabilities, outlined in Annex OO.D.1.6.1, that should be considered whenever any dynamic memory management technique is used. This leads to the following guidance.

OO.6.8.1 Dynamic Memory Management Verification Objective

Verify the use of dynamic memory management is robust with regards to reference ambiguity, fragmentation starvation, deallocation starvation, memory exhaustion, premature deallocation, lost updates and stale references, and unbound allocation or deallocation time. See Annex OO.D.1.6.1 for vulnerabilities.

OO.6.8.2 Dynamic Memory Management Verification Activities

When using dynamic memory, the memory management system should be complete and consistent.

For each of the vulnerabilities, identified in Annex OO.D.1.6.1, there should be a corresponding activity to ensure that a given vulnerability does not occur in the final system, including the following:

- a. Verify that the allocator returns a memory reference for which no other reference exists (exclusivity).
- b. Verify that memory is organized such that any allocation request will succeed when there is sufficient free memory available.
- c. Verify that allocated memory which is no longer referenced is reclaimed before that memory is needed for reuse.
- d. Verify that there is sufficient memory to accommodate the maximum storage required at any time by the program.
- e. Verify that reference consistency is maintained, that is, each object is unique, and is only viewed as that object.
- f. Verify that object moves are atomic with respect to all references to the said object.
- g. Verify that memory management operations (for example, allocation, deallocation, and the coalescence of free memory) complete within a specified time bound.

OO.7.0

SOFTWARE CONFIGURATION MANAGEMENT PROCESS

Section 7 of ED-12C is unchanged.

OO.8.0

SOFTWARE QUALITY ASSURANCE PROCESS

Section 8.0 of ED-12C is unchanged.

OO.8.1 SOFTWARE QUALITY ASSURANCE PROCESS OBJECTIVES

Section 8.1 of ED-12C is unchanged.

OO.8.2 SOFTWARE QUALITY ASSURANCE PROCESS ACTIVITIES

Activities for satisfying the SQA process objectives include:

- a. *The SQA process should take an active role in the activities of the software life cycle processes, and have those performing the SQA process enabled with the authority, responsibility, and independence to ensure that the SQA process objectives are satisfied.*
- b. *The SQA process should provide assurance that software plans and standards are developed and reviewed for compliance with ED-12C, this Supplement, and for consistency.*
- c. *The SQA process should provide assurance that the software life cycle processes comply with the approved software plans and standards.*
- d. *The SQA process should include audits of the software life cycle processes during the software life cycle to obtain assurance that:*
 1. *Software plans are available as specified in section OO.4.2.*
 2. *Deviations from the software plans and standards are detected, recorded, evaluated, tracked, and resolved.*

NOTE: *It is generally accepted that early detection of process deviations assists efficient achievement of software life cycle process objectives.*

3. *Approved deviations are recorded.*
4. *The software development environment has been provided as specified in the software plans.*
5. *The problem reporting, tracking, and corrective action process activities comply with the Software Configuration Management Plan.*
6. *Inputs provided to the software life cycle processes by the system processes, including the system safety assessment process, have been addressed.*

NOTE: *Monitoring of the activities of software life cycle processes may be performed to provide assurance that the activities are under control.*

- e. *The SQA process should provide assurance that the transition criteria for the software life cycle processes have been satisfied in compliance with the approved software plans.*
- f. *The SQA process should provide assurance that software life cycle data is controlled in accordance with the control categories as defined in ED-12C section 7.3 and the tables of Annex OO.A and ED-12C Annex A.*
- g. *Prior to the delivery of software products submitted as part of a certification application, a software conformity review should be conducted.*
- h. *The SQA process should produce records of the SQA process activities (see ED-12C 11.19), including audit results and evidence of completion of the software conformity review for each software product submitted as part of certification application.*
- i. *The SQA process should provide assurance that supplier processes and outputs comply with approved software plans and standards.*

OO.8.3 SOFTWARE CONFORMITY REVIEW
Section 8.3 of ED-12C is unchanged.

OO.9.0

CERTIFICATION LIAISON PROCESS

The objectives of the certification liaison process are to:

- a. *Establish communication and understanding between the applicant and the certification authority throughout the software life cycle to assist the certification process.*
- b. *Gain agreement on the means of compliance through approval of the Plan for Software Aspects of Certification.*
- c. *Provide compliance substantiation.*
- d. *The certification liaison process is applied as defined by the software planning process (see OO.4) and the Plan for Software Aspects of Certification (see OO.11.1). Table OO.A-10 of Annex OO.A is a summary of the objectives and outputs of this process.*

OO.9.1 MEANS OF COMPLIANCE AND PLANNING

The applicant proposes a means of compliance that defines how the development of the airborne system or equipment will satisfy the certification basis. The Plan for Software Aspects of Certification (see OO.11.1) defines the software aspects of the airborne system or equipment within the context of the proposed means of compliance. This plan also states the software level(s) as determined by the system safety assessment process.

Activities include:

- a. *Submitting the Plan for Software Aspects of Certification and other requested data to the certification authority for review.*
- b. *Resolving issues identified by the certification authority concerning the planning for the software aspects of certification.*
- c. *Obtaining agreement with the certification authority on the Plan for Software Aspects of Certification.*

OO.9.2 COMPLIANCE SUBSTANTIATION

The applicant provides evidence that the software life cycle processes satisfy the software plans, by making software life cycle data available to the certification authority for review. Certification authority reviews may take place at various facilities. For example, the reviews may take place at the applicant's facilities, the applicant's supplier's facilities, or at the certification authority's facilities. This may involve discussions with the applicant or its suppliers. The applicant arranges these reviews of the activities of the software life cycle processes and makes software life cycle data available as needed.

Activities include:

- a. *Resolving issues raised by the certification authority as a result of its reviews.*
- b. *Submitting the Software Accomplishment Summary (see OO.11.20) and Software Configuration Index (see ED-12C 11.16) to the certification authority.*
- c. *Submitting or making available other data or evidence of compliance requested by the certification authority.*

OO.9.3 MINIMUM SOFTWARE LIFE CYCLE DATA SUBMITTED TO CERTIFICATION AUTHORITY

Section 9.3 of ED-12C is unchanged.

OO.9.4 SOFTWARE LIFE CYCLE DATA RELATED TO TYPE DESIGN

Section 9.4 of ED-12C is unchanged.