Handbook for Object-Oriented Technology in Aviation (OOTiA)

Volume 1: Handbook Overview

Revision 0

October 26, 2004

This Handbook does not constitute Federal Aviation Administration (FAA) policy or guidance, nor is it intended to be an endorsement of object-oriented technology (OOT). This Handbook is not to be used as a standalone product but, rather, as input when considering issues in a project-specific context.

Contents

Figures

NOTE: This Handbook does not constitute Federal Aviation Administration (FAA) policy or guidance, nor is it intended to be an endorsement of object-oriented technology (OOT). This Handbook is not to be used as a standalone product but, rather, as input when considering issues in a project-specific context.

1.1 Handbook Introduction

1.1.1 Purpose

The purpose of this four-volume Handbook is to identify key issues and provide some potential approaches to address these issues when using OOT in airborne products. Although some of the issues identified in this Handbook are not unique to OOT, they are discussed in the Handbook because the way they are addressed is key to safe implementation of OOT. This Handbook also provides an approach for certification authorities and their designees to help ensure that OOT issues have been addressed in the projects they are reviewing or approving.

1.1.2 Background

Compliance with the objectives of RTCA/DO-178B, *Software Considerations in Airborne Systems and Equipment Certification* [1], is the primary means of obtaining approval of software used in civil aviation products. When DO-178B was published in 1992, procedure programming was the predominant technique for organizing and coding computer programs. Consequently, DO-178B provides guidelines for software developed using a functional technique and does not specifically consider software developed using OOT. OOT is a software development technique that is "expressed in terms of objects and connections between those objects" [9]. Since object-oriented technology differs from the traditional functional approach to software development, satisfying some of the DO-178B objectives when using OOT may be unclear and/or complicated.

To date, few airborne computer systems in civil aviation have been implemented using OOT. Although OOT is intended to promote productivity, increase reusability of software, and improve quality, uncertainty about how to comply with certification requirements has been a key obstacle to using OOT in airborne systems.

Although organizations such as the Object Management Group (OMG) work to develop specifications for OOT, no universal guidelines exist for using OOT in safety-critical systems. Certification authorities have been using issue papers on a project-by-project basis to address OOT concerns. These project-specific issue papers document high-level safety issues and concerns with OOT but do not suggest or provide detailed issues or acceptable solutions.

This Handbook extends the use of issue papers by identifying key issues and providing some guidelines to help the software community satisfy applicable DO-178B objectives when using OOT. It also provides an approach for certification authorities and their designees when evaluating OOT projects.

The FAA co-sponsored the Object-Oriented Technology in Aviation (OOTiA) project with the National Aeronautics and Space Administration (NASA) to address OOT challenges in aviation. The FAA, NASA, other government organizations, academia, international certification authorities, airborne systems manufacturers, and aircraft manufacturers collaborated through two OOTiA workshops and the OOTiA workshop committee to produce this Handbook [see Section 1.6 for additional information on the workshops].

It is anticipated that this Handbook and other documents may be used to impact future changes to the FAA's software guidance (e.g., to impact future revisions to DO-178B or supplementary guidance). It is also anticipated that this Handbook will be updated in the future as OOT in aviation matures and lessons are learned. If you have comments, suggested improvements to this Handbook, additional issues, or potential solutions to address an issue(s), please complete and submit the feedback form in Section 1.8 of this volume.

1.2 Handbook Organization

1.2.1 Scope

This Handbook documents key issues and some potential approaches to address these issues when using OOT in airborne systems. It is intended to be informational and educational – a compilation of what we know to date regarding the use of OOT in aviation systems. **This Handbook does not constitute Federal Aviation Administration (FAA) policy or guidance nor is it intended to be an endorsement of OOT. This Handbook is not to be used as a standalone product but, rather, as input when considering issues in a project-specific context.** In addition, this Handbook does not attempt to define the project criteria or circumstances under which this Handbook should or should not be used in software development. That determination is left to project planners, decision makers, or developers, as appropriate.

This Handbook addresses issues that were identified as having potential impact in safely applying OOT in airborne systems. Certification authorities, industry, and others submitted potential issues through a web site dedicated to the OOTiA project [7]. Some of the issues are not unique to OOT (e.g., inlining and templates); however, these issues are discussed in the Handbook because the way they are addressed is key to safe implementation of OOT. Note that this Handbook does not address all potential issues, nor are the guidelines in Volume 3 the only possible solutions to addressing the related issues. As technology advances and experience with OOT increases within the aviation community, this Handbook will likely be updated.

1.2.2 Approach

The Handbook follows a "tiered" approach as shown in Figure 1.2-1 in which each of its four volumes provides the foundation for all volumes above it. For example, Volume 3: *Best Practices* relies on contents in both Volume 1: *Handbook Overview* and Volume 2: *Considerations and Issues* in substantiating its guidelines.

The four volumes are:

- Volume 1: *Handbook Overview* (this volume)
- Volume 2: *Considerations and Issues*
- Volume 3: *Best Practices*
- Volume 4: *Certification Practices*

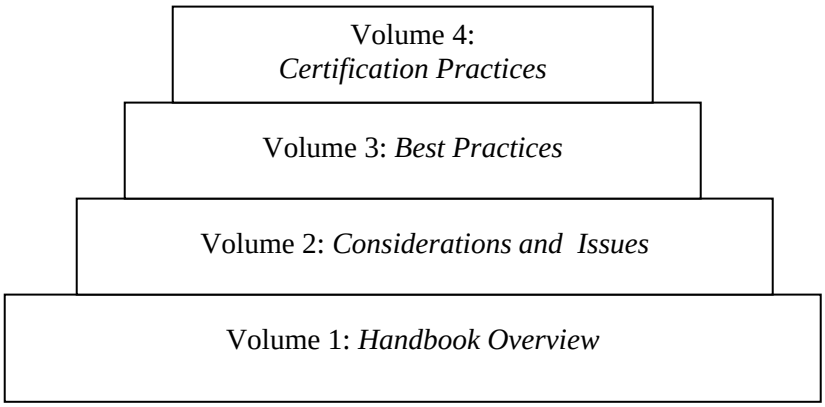

Figure 1.2-1 Handbook Approach

NOTE: This Handbook does not constitute Federal Aviation Administration (FAA) policy or guidance, nor is it intended to be an endorsement of object-oriented technology (OOT). This Handbook is not to be used as a standalone product but, rather, as input when considering issues in a project-specific context.

Each volume is written for a unique combination of target audience and purpose. Each volume is self-contained in that each has a separate list of references applicable to that volume alone. However, to provide a consistent basis among volumes, Volume 1: *Handbook Overview* contains the Acronym List (see section 1.4) and Glossary (see section 1.5) common to all volumes.

The following sections provide the title, target audience, purpose, and overview of the contents for each volume.

1.2.2.1 Volume 1: Handbook Overview

Target Audience: All Handbook users

Purpose: Provide background and foundational information needed to use all other volumes

Contents: Handbook introduction

Organizational overview of Handbook into volumes

OOT background

Handbook acronym list

Handbook glossary

OOTiA workshop committee and participants lists

References for Volume 1

Feedback form for suggested improvements to the Handbook

1.2.2.2 Volume 2: Considerations and Issues

Target Audience: Project planners, decision makers, certification authorities, designees

Purpose: Report and discuss the challenges collected throughout the OOTiA project

Contents: Considerations before making the decision to use OOT

Considerations after making the decision to use OOT

Open issues (summary of OOTiA Workshop #2 brainstorming session)

Summary

References for Volume 2

Results of the "Beyond the Handbook" session of OOTiA Workshop #2

Mapping of issue list to considerations

Additional considerations for project planning

1.2.2.3 Volume 3: Best Practices

Target Audience: Airborne systems and OOT software developers, certification authorities, designees

Purpose: Identify best practices to safely implement OOT in airborne systems by providing some known ways to address the issues documented in Volume 2

Contents: Mapping of Volume 2 issues to Volume 3 guidelines

Guidelines for demonstrating DO-178B compliance and safely addressing:

- Single inheritance and dynamic dispatch
- Multiple inheritance

- Templates
- Inlining
- Type conversion
- Overloading and method resolution
- Dead and deactivated code, and reuse
- Object-oriented tools
- Traceability
- Structural coverage

References for Volume 3

Index of terms

Frequently asked questions

Extended guidelines and examples

1.2.2.4 Volume 4: Certification Practices

Target Audience: Certification authorities and designees

Purpose: Provide an approach to ensure that OOT issues are addressed

Contents: Activities for Stages of Involvement 1- 4

References for Volume 4

1.3 OOT Background

1.3.1 OOT Basics

Object-oriented approaches date to the introduction of the programming language Simula in 1967. Most recently they have been standardized by the Object Management Group (OMG) through their definition of a Unified Modeling Language (UML) [16], and in other specifications related to model-driven architectures, distributed object communication, etc.

OOT is a software development technique that is centered on "objects." The Institute of Electrical and Electronics Engineers (IEEE) refers to OOT as "a software development technique in which a system or component is expressed in terms of objects and connections between those objects" [9]. An object can be compared to a "black box" at the software level – it sends and receives messages. The object contains both code (methods) and data (structures). The user does not need to have insight into the internal details of the object in order to use the object, hence the comparison to a black box. An object can model real world entities, such as a sensor or hardware controller, as separate software components with defined behaviors.

A major concept in OOT is the "class." A class is a set of objects that share the same attributes, methods, relationships, and semantics – they share a common structure and behavior [16]. A class describes the characteristics and behavior of a real world entity. Figure 1.3-1 illustrates a representation of a class definition for an object.

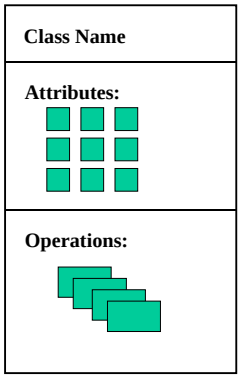

Figure 1.3-1 Object-Oriented Class Representation

1.3.2 Principles of OOT

There are seven principles that form the foundation for OOT: abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence [8]. Not all of these principles are unique to OOT, but the OOT development methodology embodies these seven principles. Abstraction, modularity, concurrency, and persistence are principles that are commonly used in other development methodologies. However, encapsulation (using a technique called *information hiding*), hierarchy (using a technique called *inheritance*), and typing (using a concept called *polymorphism*) are relatively unique to OOT. Each of the seven principles is described below.

Abstraction is one of the fundamental ways that complexity is addressed in software development. “An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer" [8].

Encapsulation is the process of hiding the design details in the object implementation. Encapsulation can be described as “the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse” [15]. Encapsulation is generally achieved through *information hiding*, which is the process of hiding the aspects of an object that are not essential for the user to see. Typically, both the structure and the implementation methods of the object are hidden.

Modularity is the process of dividing a program into logically separated and defined components that possess defined interactions and limited access to data. Booch writes that modularity is a “property of a system that has been decomposed into a set of cohesive and loosely coupled modules” [8].

Hierarchy is simply the ordering of abstractions. Examples of hierarchy are *single inheritance* and *multiple inheritance*. In OOT, when a sub-class is created, this new class “inherits” all of the existing attributes and operations of the original class, called the “parent” or “superclass” [13]. Inheritance is a relationship between classes where one class is the “parent” (also called “base,” “superclass,” or “ancestor”) class of another [6]. One author puts it this way, “Inheritance is a relationship among classes where a child class can share the structure and operations of a parent class and adapt it for its own use” [10].

Inheritance is one of the key differences between OOT and conventional software development. There are two types of inheritance: *single inheritance* and *multiple inheritance*. In *single inheritance*, the sub-class inherits the attributes and operations from a single superclass. In *multiple inheritance*, the sub-class inherits some attributes from one class and others from another class. *Multiple inheritance* is controversial, because it complicates the class hierarchy and configuration control [14].

Typing is a principle that is used in OOT that has many definitions. Booch presents one of the most clear and concise definitions by stating, "Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways" [8]. Examples of OOT typing are strong typing, weak typing, static typing, and dynamic typing. Each OOT programming language varies in its implementation of typing.

Another OOT concept closely related to typing is *polymorphism*. *Polymorphism* comes from the Greek meaning "many forms." It allows one name to be used for two or more related but different purposes [15]. It is the ability of an object to assume or become many different forms of an object. *Polymorphism* specifies slightly different or additional structure or behavior for an object, when assuming or becoming an object [11]. This allows different underlying implementations for the same command. For example, assume there exists a vehicle class that includes a steer-left command. If a boat object was created from the vehicle class, the steer-left command would be implemented by a push to the right on a tiller. However, if a car object was created from the same class, it might use a counter-clockwise rotation of the steering wheel to achieve the same command.

Concurrency is the process of carrying out several events simultaneously.

Persistence is "the property of an object through which its existence transcends time (i.e., the object continues to exist after its creator ceases to exist) and/or space (i.e., the object's location moves from the address space in which it was created)" [8].

1.3.3 OOT Methodology

OOT can typically be described in four processes: Object-Oriented Analysis (OOA), Object-Oriented Design (OOD), Object-Oriented Programming (OOP), and Object-Oriented Verification/Test (OOV/T). The implementation of these processes is typically iterative or evolutionary. An overview of each process is addressed below.

OOA is the process of defining all classes that are relevant to solve the problem and the relationships and behavior associated with them [14].

A number of tasks occur to carry out the OOA as shown in Figure 1.3-2. The tasks are reapplied until the model is completed. As shown in Figure 1.3-2, use cases, class-responsibility-collaborator (CRC) models, object-relationship (OR) models, and object-behavior (OB) models are methods typically used to carry out the OOA. The use case is a method utilized to identify the user's requirements. The CRC model is used to identify the class attributes, operations, and hierarchy. The OR model is used to illustrate the relationship between the numerous objects. And, the OB model is used to model the behavior of each object.

OOD transforms the OOA into a "blueprint" for software "construction." Four layers of design are usually defined: subsystem layer, class and object layer, message layer, and responsibilities layer. The *subsystem design layer* represents each subsystem that enables software to achieve the requirements. The *class and object design layer* contains class hierarchies and object designs. The *message design layer* contains the internal and external interfaces to communicate between objects. The *responsibilities design layer* contains the algorithm design and data structures for attributes and operations of each object.

OOP is the coding process of the software development, using an OO language (e.g., C++, Ada 95).

OOV/T is the process of detecting errors and verifying the correctness of the OOA, OOD, and OOP. OOV/T includes reviews, analyses, and tests of the software requirements, design, and implementation. OOV/T requires slightly different strategies and tactics than the traditional functional approach. The variance in the approach is driven by characteristics like inheritance, encapsulation, and polymorphism. Many developers use a *design for testability* approach to begin addressing any verification/test issues early in the program.

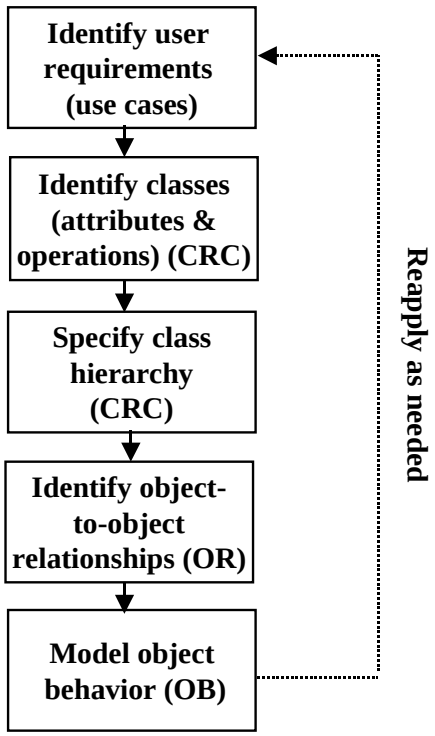

Figure 1.3-2 OOA Tasks

1.3.4 OOT Languages

Many OO languages exist. Some of the most well known are C++, Smalltalk, Ada 95, and Java. C++, Ada 95, and Java are of particular interest for designers of embedded software. C++'s tool support, Ada 95's extension of Ada 83, and Java's platform independence make these languages very appealing to the developers of airborne systems. But each language has its own set of challenges. Volume 3 of this Handbook attempts to address language-specific issues, where possible. Other volumes are intended to be language-independent. C# is another OO language being considered for embedded software. However, it is still maturing and is not yet addressed in this Handbook.

1.3.5 Additional Key OO Concepts

In addition to the OO concepts described above, the following concepts are important OO concepts mentioned in this Handbook.

Dynamic dispatch is the association of a method with a call based on the run-time type of the target object. Dynamic dispatch is not related to dynamic linking nor dynamic link libraries. Dynamic dispatch is sometimes referred to as "dynamic binding." There are two types of dynamic dispatch used in OO:

- ***Single dispatch*** is dynamic dispatch based on only the run-time type of the target object. Most OO languages, including Java, Ada 95 and C++ are single dispatching.
- ***Multiple dispatch*** is dynamic dispatch based on the run-time types of all the arguments to a call, rather than only the run-time type of the target object.

NOTE: This Handbook does not constitute Federal Aviation Administration (FAA) policy or guidance, nor is it intended to be an endorsement of object-oriented technology (OOT). This Handbook is not to be used as a standalone product but, rather, as input when considering issues in a project-specific context.

The ***Liskov substitution principle*** *(LSP)* is a set of subtyping rules that ensure that instances of a subclass are substitutable for instances of all parent classes in every context in which they may appear. These rules go beyond the simple checking of signatures, taking into account the behavior of operations (as defined by their pre- and post-conditions) and the invariants defined by classes. Even if classes are not defined formally, the principle can be upheld by requiring the inheritance of test cases. Reference [17] provides additional insight into the LSP concept.

1.3.6 Further OOT Reading

The section identifies some resources recommended for those who desire to study OOT in more depth. While this is not an exhaustive list of OO references, it is intended to provide a starting point for those interested in learning more about OOT. The Reference sections of this volume and the other Handbook volumes provide additional resources.

- *The Object-Oriented Thought Process* by Matt Weisfeld (SAMS Publishing, 2000): This book provides a simple introduction to the OO fundamentals. It is a good resource for those transitioning from the functional approach to OO.
- *Object-Oriented Analysis and Design* by Grady Booch (Addison-Wesley, 2nd edition, 1994): This book provides a practical introduction to OO concepts, methods, and applications.
- *Pitfalls of Object-Oriented Development* by Bruce Webster (M&T Books, 1995): Although somewhat dated, this book provides a sound overview of the potential problems in OO development. A summary of its main points are included in Appendix C of Volume 2 since the book is now out of print.
- *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (Addison-Wesley, 1995): Patterns are widely used by the OO community to address analysis and design problems. This book provides a guide for effective development and use of patterns.
- *Object-Oriented Software Construction* by Bertrand Meyer (Prentice Hall, 2nd edition, 1997): Although a large book, this one provides good fundamental information for OO developers.
- *Testing Object-Oriented Systems: Models, Patterns, and Tools* by Robert V. Binder (Addison-Wesley, Reading, MA, 2000): This book addresses one of the more difficult aspects of OOT testing.

1.4 Acronym List

The following acronym list applies to all volumes of this Handbook:

AC	Advisory Circular
ACB	Anomalous Construction Behavior
AMJ	Advisory Material Joint
API	Application Programming Interface
AVSI	Aerospace Vehicle Systems Institute
BIT	Built-in Test
CAST	Certification Authorities Software Team
CC	Control Category
CM	Configuration Management
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-The-Shelf
CRC	Class-Responsibility-Collaborator
DBC	Design By Contract
DER	Designated Engineering Representative
EASA	European Aviation Safety Agency
EUROCAE	European Organization for Civil Aviation Equipment
FAA	Federal Aviation Administration
HIT	Hierarchical Incremental Testing
IEEE	Institute of Electrical and Electronics Engineers
IL	Issue List
IP	Issue Paper
JAA	Joint Aviation Authorities
LRU	Line Replaceable Unit
LSP	Liskov Substitution Principle
MC/DC	Modified Condition / Decision Coverage
NASA	National Aeronautics and Space Administration
OB	Object Behavior
OMG	Object Management Group
OO	Object-Oriented
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OOP	Object-Oriented Programming
OOT	Object-Oriented Technology
OOTiA	Object-Oriented Technology in Aviation
OOV/T	Object-Oriented Verification/Test
OR	Object Relationship
PDS	Previously Developed Software
PSAC	Plan for Software Aspects of Certification
RBT	Requirements-Based Testing
R-D-C	Requirements-Design-Code
RSC	Reusable Software Component
RTCA	RTCA, Inc.

NOTE: This Handbook does not constitute Federal Aviation Administration (FAA) policy or guidance, nor is it intended to be an endorsement of object-oriented technology (OOT). This Handbook is not to be used as a standalone product but, rather, as input when considering issues in a project-specific context.

SAS	Software Accomplishment Summary
SCI	Software Configuration Index
SDA	State Definition Anomaly
SDI	State Defined Incorrectly
SDIH	State Definition Inconsistency
SOI	Stage of Involvement
SSA	System Safety Assessment
SVA	State Visibility Anomaly
UML	Unified Modeling Language

1.5 Glossary

This glossary provides definitions for terms used in this Handbook to discuss object-oriented technology issues. It is intended to provide terminology for consistent communication and discussion of issues. Many terms are taken directly from the glossary[1] of RTCA/DO-178B, *Software Considerations in Airborne Systems and Equipment Certification* [1]. Other terms are taken from references on object-oriented technology. References are noted, as appropriate, after each definition. Relationships between terms are denoted as *contrast*, *synonym*, and *see*, where specifically pertinent to the definition. This glossary is a resource to be used for all volumes of the Handbook.

Abstract class - A class that cannot be directly instantiated. Any class containing an abstract operation must itself be abstract. Contrast: *concrete clas*s. [2], [16]

Abstract operation - An operation that is declared but not implemented by an abstract class. Abstract operations do not have associated methods (bodies) in the class that defines them, but must have an associated implementation in concrete subclasses. See: *operation*, *method*. [2], [8] Contrast: *concrete operation*.

Abstract pattern - A pattern that does not prescribe a particular approach. Additional guidelines are defined by sub-patterns of the abstract pattern where these guidelines vary by the approach used.

Access mechanism - The manner in which a software component is called upon to perform its intended function. This includes invocation mechanisms and data flow to and from the component. This is typically part of the interface description data. [3]

Actual parameter - See: *argument.*

Aggregation - A composition technique for building a new object from one or more existing objects that support some or all of the new object's required interfaces. Synonym: *composition*. [18]

Algorithm - A finite set of well-defined rules that gives a sequence of operations for performing a specific task. [1]

Anomalous behavior - Behavior that is inconsistent with specified requirements. [1]

Applicant - A person or organization seeking approval from the certification authority. [1]

Approval - The act or instance of expressing a favorable opinion or giving formal or official sanction. [1]

Argument - A binding for a parameter that resolves to a run-time instance. Synonym: *actual parameter*. Contrast: *parameter*. [16]

Aspect-oriented programming - An approach used to encapsulate policies and strategies that cross-cut the core functionality of a system. Such system- or subsystem-wide policies are referred to aspects. They include policies for error handling, synchronization, resource allocation, fault-tolerance, performance, software monitoring, distributed data access, and other potentially safety related issues. Aspect-oriented programming is generally viewed as complementary to object-oriented development. [2]

Assertion - A Boolean expression whose value should always evaluate to true at a given point in a program's execution. For example, a pre-condition, whose value should evaluate to true whenever a given operation is called; a post-condition, whose value should evaluate to true upon the normal (non-exceptional) return from a given operation; or a class invariant, whose value should evaluate to true after the execution of a class constructor and before/after the execution of every externally visible operation on the class.

Association - The semantic relationship between two or more classifiers that specifies connections among their instances. Such connections may be represented as pointers or access types that reference other objects. They may also be computed rather than stored. [2], [16]

Assurance - The planned and systematic actions necessary to provide adequate confidence and evidence that a product or process satisfies given requirements. [1]

[1] The glossary definitions from RTCA/DO-178B are used with permission of RTCA.

Attribute - A feature within a class that describes a range of values those instances of the class may hold. Attributes are stored values or fields in Ada 95, C++, and Java. They may represent either data values or references to other objects (association ends). [2], [16]

Audit - An independent examination of the software life cycle processes and their outputs to confirm required attributes. [1]

Baseline - The approved, recorded configuration of one or more configuration items, that thereafter serves as the basis for further development, and that is changed only through change control procedures. [1]

Certification - Legal recognition by the certification authority that a product, service, organization, or person complies with the requirements. Such certification comprises the activity of technically checking the product, service, organization, or person and the formal recognition of compliance with the applicable requirements by issue of a certificate, license, approval, or other documents as required by national laws and procedures. In particular, certification of a product involves: (a) the process of assessing the design of a product to ensure that it complies with a set of standards applicable to that type of product so as to demonstrate an acceptable level of safety; (b) the process of assessing an individual product to ensure that it conforms with the certified type design; (c) the issuance of a certificate required by national laws to declare that compliance or conformity has been found with standards in accordance with items (a) or (b) above. [1]

Certification authority - The organization or person responsible within the state or country concerned with the certification of compliance with the requirements.

Note: *A matter concerned with aircraft, engine, or propeller type certification or with equipment approval would usually be addressed by the certification authority; matters concerned with continuing airworthiness might be addressed by what would be referred to as the airworthiness authority.* [1]

Certification credit - Acceptance by the certification authority that a process, product, or demonstration satisfies a certification requirement. [1] See: *credit*.

Change control - (1) The process of recording, evaluating, approving, or disapproving and coordinating changes to configuration items after formal establishment of their configuration identification or to baselines after their establishment. (2) The systematic evaluation, coordination, approval, or disapproval and implementation of approved changes in the configuration of a configuration item after formal establishment of its configuration identification or to baselines after their establishment.

Note: *This term may be called configuration control in other industry standards.* [1]

Checked type conversion - Types are checked if conversion from one type to the other includes a determination either by the compiler or at run-time as to whether they are normally convertible.

Child - In a generalization relationship, the specialization of another element, the parent. See: *subclass*, *subtype*. Contrast: *parent, superclass, supertype*. Child classes inherit from their parent classes. Similarly, subclasses inherit from their superclasses. [2], [16]

Class - Informally, any classifier. Formally, a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment. [2], [16]

Class hierarchy - A collection of classes connected by generalization relationships. The root of the hierarchy represents the most general of these classes. The leaves represent the most specific of these classes. Synonym: *inheritance hierarchy*.

Classifier - The Unified Modeling Language (UML) defines the term *classifier* to include interfaces, classes, datatypes, and components. In the *Aerospace Vehicle Systems Institute (AVSI) guide* (and elsewhere), the term “class” is often used informally as a synonym for classifier. Formally, however, classes describe only objects, which have an identity and state, and not datatypes, interfaces, or components. [2], [16]

Client class - A class that can reference the attributes of another class.

Client operation - An operation accessible to classes other than the defining class and its subclasses.

Code - The implementation of particular data or a particular computer program in a symbolic form, such as source code, object code, or machine code. [1]

Code-sharing - The sharing of code by more than one class or component (e.g., by means of implementation inheritance or delegation). See: *implementation inheritance, delegation.*

Note: *There are many ways to support the sharing of code. The risk is that inheritance can be misused to support only the sharing of code and data structure, without attempting to follow behavioral subtyping rules.*

Commercial off-the-shelf (COTS) software - Commercially available applications sold by vendors through public catalog listings. COTS software is not intended to be customized or enhanced. Contract-negotiated software developed for a specific application is not COTS software. [1]

Compiler - Program that translates source code statements of a high level language, such as FORTRAN or Pascal, into object code. [1]

Component - (1) A self-contained part, combination of parts, sub-assemblies or units, which performs a distinct function of a system. (2) A physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. [1], [16]

Composition - See: *aggregation.*

Concrete class - A class that can be directly instantiated. A concrete class has no abstract operations. Contrast: *abstract class.* [2], [16]

Concrete operation - An operation that has an associated method in the context of a given class. Contrast: *abstract operation.* [2]

Concurrency - (1) Process of carrying out several events simultaneously. (2) A technique used in an operating system for sharing a single processor between several independent jobs. [18]

Condition - A Boolean expression containing no Boolean operators. [1]

Condition/Decision Coverage - Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken on all possible outcomes at least once, and every decision in the program has taken on all possible outcomes at least once. [1]

Configuration identification - (1) The process of designating the configuration items in a system and recording their characteristics. (2) The approved documentation that defines a configuration item. [1]

Configuration item - (1) One or more hardware or software components treated as a unit for configuration management purposes. (2) Software life cycle data treated as a unit for configuration management purposes. [1]

Configuration management - (1) The process of identifying and defining the configuration items of a system, controlling the release and change of these items throughout the software life cycle, recording and reporting the status of configuration items and change requests and verifying the completeness and correctness of configuration items. (2) A discipline applying technical and administrative direction and surveillance to: (a) identify and record the functional and physical characteristics of a configuration item, (b) control changes to those characteristics, and (c) record and report change control processing and implementation status. [1]

Configuration status accounting - The recording and reporting of the information necessary to manage a configuration effectively, including a listing of the approved configuration identification, the status of proposed changes to the configuration and the implementation status of approved changes. [1]

Constraint - A semantic condition or restriction. Constraints include pre-conditions, post-conditions, and invariants. They may apply to a single class of objects, to relationships between classes of objects, to states, or to use cases. [16]

Constructor - An operation that creates an object and/or initializes its state. Formally, the constructor is responsible for establishing any class invariant.

Control coupling - The manner or degree by which one software component influences the execution of another software component. [1]

Control coupling analysis - Evaluation of the execution relationships and dependencies between software components and in component logic to ensure application execution is correctly designed and implemented.

Control flow analysis - (1) Analysis typically used in the identification and confirmation of control coupling. DO-178B does not explicitly define this term or the related term *control flow*. However, it references both (on pages 21, 28, 52, 61, and 57). [1] (2) Analysis whose objectives are: to ensure the code is executed in the right sequence, to ensure the code is well structured, to locate any syntactically unreachable code, and to highlight parts of the code where termination needs to be considered (i.e., loops and recursion). Call tree analysis is cited as an example of one of many control flow analysis techniques, and is offered as a means of confirming that design rules for the partitioning of critical and non-critical code have been followed. [5]

Control program - A computer program designed to schedule and to supervise the execution of programs in a computer system; e.g., operating system, executive, run-time system. [1]

Conversion - See: *type conversion.*

CORBA - An industry wide standard for communication between distributed objects, independent of their location and target language. The CORBA standard is defined by the Object Management Group (OMG). CORBA itself is an acronym for Common Object Request Broker Architecture. [2]

Coupling - A relationship between components or elements.

Coverage analysis - The process of determining the degree to which a proposed software verification process activity satisfies its objective. [1]

Credit - The compliance to one or more RTCA/DO-178B objectives supported by RTCA/DO-178B software life cycle data. This compliance is used to show that the certification basis has been met and the equipment may receive a certificate. Three types of credit are referred to throughout AC 20-148 [3]:

1. Full credit: fully meets the RTCA/DO-178B objective and requires no further activity by the user.
2. Partial credit: partially meets the RTCA/DO-178B objective and requires additional activity by the user to complete compliance.
3. No credit: does not meet the RTCA/DO-178B objective and must be completed by the user for compliance. [3]

Data abstraction - An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects, suppressing all non-essential details. In data abstraction the non-essential details deal with the underling data representation. [8] [16]

Database - A set of data, part or the whole of another set of data, consisting of at least one file that is sufficient for a given purpose or for a given data processing system. [1]

Data coupling - The dependence of a software component on data not exclusively under the control of that software component. [1]

Data coupling analysis - An evaluation of the data flow relationships and dependencies between software components to ensure they are correctly designed and implemented.

Data dictionary - The detailed description of data, parameters, variables, and constants used by the system. [1]

Data flow analysis - (1) Analysis typically used in the identification and confirmation of data coupling. DO-178B does not explicitly define this term or the related term *data flow*. However, it references both (on pages 21, 28, 52, 61, and 57). [1] (2) Analysis whose objective is to show that there is no execution path in the software that would access a variable that does not have a set value. Data flow analysis uses the results of control flow analysis in conjunction with the read or write access to variables to perform the analysis. Data flow analysis can also detect other code anomalies such as multiple writes without intervening reads. [5]

Data type, Datatype - (1) A class of data characterized by the members of the class and the operations that can be applied to them. Examples are character types and enumeration types. (2) A descriptor of a set of values that lack identity and whose operations do not have side effects. Datatypes include primitive pre-defined types and user-definable types. Pre-defined types include numbers, string and time. User-definable types include enumerations.

Note: *Instances of datatypes, unlike objects, do not have identity or state, but are immutable. As a result, operations on datatypes do not change the state of values they act upon, but compute new values based on existing ones. Some languages use the term immutable in combination with terms class and object to denote a datatype and its values.* [1], [2] See: *immutable.*

Deactivated code - Executable object code (or data) which by design is either (a) not intended to be executed (code) or used (data), for example, a part of a previously developed software component, or (b) is only executed (code) or used (data) in certain configurations of the target computer environment, for example, code that is enabled by a hardware pin selection or software programmed options. [1]

Dead code - Executable object code (or data) which, as a result of a design error cannot be executed (code) or used (data) in a operational configuration of the target computer environment and is not traceable to a system or software requirement. An exception is embedded identifiers. [1]

Decision - A Boolean expression composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition. If a condition appears more than once in a decision, each occurrence is a distinct condition. [1]

Decision Coverage - Every point of entry and exit in the program has been invoked at least once and every decision in the program has taken on all possible outcomes at least once. [1]

Declared type - The type associated with a name (such as a variable, constant or parameter) at its point of declaration. The run-time type of any associated object must be a subtype of its declared type. [2]

Delegation - The implementation of an operation by means of a call to an equivalent operation on a component object (the delegate). Delegation can be used as an alternative to implementation inheritance. Contrast: *inheritanc*e. [2]

Derived requirements - Additional requirements resulting from the software development processes, which may not be directly traceable to higher level requirements. [1]

Derived type - A type derived for specialization from another type. The derived type is a specialization from the conceptual point of view and may be an expansion from the structural point of view. [4]

Design pattern - A documented solution to a commonly encountered design problem. In general, a design pattern presents a problem, followed by a description of its solution in a given context and programming language. [2]

Designee - An industry representative that has been authorized by the FAA to make findings of compliance on behalf of the FAA.

Destructor - An operation that frees the state of an object and/or destroys the object itself. [8]

Dynamic binding - See: *dynamic dispatch*.

Dynamic classification - A semantic variation of generalization in which an object may change its classifier. Contrast: *static classificatio*n. Using dynamic classification, the class of an object may change during its lifetime. Using static classification, it may not. [2], [16]

Dynamic dispatch - The association of a method with a call based on the run-time type of the target object. Dynamic dispatch is not related to dynamic linking or dynamic link libraries. Synonym: *dynamic binding*. [2]

Dynamic loading (of classes) - The loading of classes dynamically (at run-time) when they are first referenced by an application. The desktop Java environment, for example, provides a class loader capable of finding and loading a named class appearing in any of a prescribed list of locations, which may be either local or remote. In real-time systems, class loading is generally not supported or permitted.

Emulator - A device, computer program, or system that accepts the same inputs and produces the same output as a given system using the same object code. [1]

Equivalence class - The partition of the input domain of a program such that a test of a representative value of the class is equivalent to a test of other values of the class. [1]

Error - With respect to software, a mistake in requirements, design or code. [1]

Executable Object Code - Consists of a form of Source/Object Code that is directly usable by the central processing unit of the target computer and is, therefore, the software that is loaded into the hardware or system. [1]

Explicit type conversion - Conversion of a value from its type to a designated type by use of a conversion routine.

Failure - The inability of a system or system component to perform a required function within specified limits. A failure may be produced when a fault is encountered. [1]

Failure condition - The effect on the aircraft and its occupants both direct and consequential, caused or contributed to by one or more failures, considering relevant adverse operational and environmental conditions. A failure condition is classified according to the severity of its effect as defined in FAA AC 25.1309-1A or JAA AMJ 25.1309. [1]

Fault - A manifestation of an error in software. A fault, if it occurs, may cause a failure. [1]

Fault tolerance - The built-in capability of a system to provide continued correct execution in the presence of a limited number of hardware or software faults. [1]

Feature - An attribute, operation, or method. This includes attributes that reference other objects (i.e., association ends). Features correspond to methods and fields in Java, methods and member functions in C++, and subprograms and record fields in Ada 95.

Flattened class - The *flattened* form of a class is a self contained module representing the composition of its elements with those inherited by it, taking into account the rules for inheritance associated with the language. Inherited elements appear in the flattened class if: 1) they are defined by a superclass and never overridden, or 2) they are defined by a superclass and referenced by some other element that is not overridden. Some languages (e.g., Eiffel) allow you to print the flattened form of a class interface. This is useful to clients because it specifies the full client interface, eliminating the need to refer to superclass definitions. [12]

Flow analysis - A term encompassing both control flow analysis and data flow analysis. [2]

Formal analysis - A type of analysis with a range of acceptable approaches, some of which are only minimally formal (assertions are specified in a reasonably precise manner), others of which are highly formal (involve use of formal specification languages and supporting tools). The former would involve writing assertions in some agreed upon, reasonably precise form, and some simple inspections (involving checking of signatures and pre-/post-conditions) where the basis for the checks is formal, but no training in formal methods is required and no formal methods tools need be used. The latter would involve formal methods to provide an automated approach with more precise, more completely specified assertions that serve as the basis for model checking and other forms of analysis.

Formal methods - Descriptive notations and analytical methods used to construct, develop, and reason about mathematical models of system behavior. [1]

Framework - A framework is a partially completed software application, which has a set of related classes that can be specialized and/or instantiated to implement the application. Since the UML is a formally defined language, some of the existing visual modeling tools use existing frameworks to help the coder/developer generate complete applications from UML models.

Generalization - A taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element may be used where the more general element is allowed. See: *inheritance.* [16]

Generic - In an Ada program, a generic is a unit that allows the same logical function on more than one type of data.

Hard real-time system - A real-time system in which lateness is not accepted under any circumstance. [6]

Hardware/software integration - The process of combining the software into the target computer. [1]

High-level requirements - Software requirements developed from analysis of system requirements, safety-related requirements, and system architecture. [1]

Host computer - The computer on which the software is developed. [1]

Immutable - Incapable of being changed. Immutable objects represent values whose state cannot be changed (e.g., the string literal “ABC” or the integer literal “4”). Immutable values, however, may be combined to produce new values. The string “ABC”, for example, may be concatenated with the string “DEF” to produce a new immutable string value “ABCDEF”. See: *data type, datatype.*

Implementation - A definition of how something is constructed or computed. For example, a class is an implementation of a type, a method is an implementation of an operation. [16]

Implementation inheritance - The inheritance of the implementation of a more specific element. Includes inheritance of the interface. Contrast: *interface inheritanc*e. Unlike interface inheritance, the inherited elements are more than specifications. They contribute to the executable object code. [2], [16]

Implicit type conversion - A type conversion generated by the compiler as the result of an association between variables of different types, resulting in a value being converted to an expected type based on context.

Independence - Separation of responsibilities which ensures the accomplishment of objective evaluation. (1) For software verification process activities, independence is achieved when the verification activity is performed by a person(s) other than the developer of the item being verified, and a tool(s) may be used to achieve an equivalence to the human verification activity. (2) For the software quality assurance process, independence also includes the authority to ensure corrective action. [1]

Inheritance - A mechanism by which more specific elements incorporate (inherit) the structure and behavior of more general elements. Inheritance can be used to support generalization, or misused to support only code sharing, without attempting to follow behavioral subtyping rules. See: *generalization*, *Liskov substitution principle*. [16]

Inheritance hierarchy - See: *class hierarchy.*

Inherited element - An element of a class inherited by its subclasses. In UML, inherited elements include operations, methods, associations, and constraints involving classes.

Inline - A command used in Java, Ada, and C++ to hint to the compiler that expansion of a method body within the code of a calling method is to be preferred to the usual call implementation. For all of these languages, the compiler can follow or ignore the recommendation to inline. [2]

Instance - An entity to which a set of operations can be applied and which has a state that stores the effects of the operations. See: *objec*t. [2]

Integral process - A process which assists the software development processes and other integral processes and, therefore, remains active throughout the software life cycle. The integral processes are the software verification process, the software quality assurance process, the software configuration management process, and the certification liaison process. [1]

Integrator - The manufacturer responsible for integrating a software component (e.g., a reusable software component) into the target computer and system with other software components. [3]

Interface – For object-oriented technology, an interface is a definition of the features accessible to clients of a class. Interfaces are distinct from classes, which may also contain methods, associations and modifiable attributes.

Note: *The UML definition of interface differs slightly from that defined by Java in that Java interfaces may contain constant fields, while UML interfaces may contain only operations.* [2]

Interface description data - Data that identifies the interface details of the reusable software component. It is provided by the reusable software component developer to the integrator and applicant. \ The interface description

data should explicitly define what activities are required by the integrator and/or applicant to ensure that the reusable software component will function in accordance with its approval basis. [3]

Interface inheritance - The inheritance of the interface of a more specific element. Does not include inheritance of the implementation. Contrast: *implementation inheritanc*e. Unlike implementation inheritance, the inherited elements are only specifications. They are not contained in the executable object code. [2], [16]

Interrupt - A suspension of a task, such as the execution of a computer program, caused by an event external to that task, and performed in such a way that the task can be resumed. [1]

Invariant - A condition associated with a class that is established when a new instance of the class is created and must be maintained by all its publicly accessible operations. As a result, the invariant is effectively a part of the pre-condition and the post-condition of every such operation. It may be violated in the intermediate states that represent the execution of a given method so long as the operations of the object are properly synchronized and such violations are not externally observable. [2]

Liskov substitution principle (LSP) - A set of subtyping rules that ensure that instances of a subclass are substitutable for instances of all parent classes in every context in which they may appear. These rules go beyond the simple checking of signatures, taking into account the behavior of operations (as defined by their pre and post conditions) and the invariants defined by classes. Even if classes are not defined formally, the principle can be upheld by requiring the inheritance of test cases. [2] See: *inheritance.*

Logically unrelated types - Types are logically unrelated when one does not define a set of operations that is a subset of the other.

Low-level requirements - Software requirements derived from high-level requirements, derived requirements, and design constraints from which source code can be directly implemented without further information. [1]

Macro-expansion - Full expansion of the code generated by the compiler for each instantiation of a template.

Maintenance code - Code residing in a airborne computer-based system that interfaces with an onboard maintenance computer or computer used by maintenance personnel. The function of this code is usually to report to the maintenance computer any problems detected during normal operations. [3]

Means of compliance - The intended method(s) to be used by the applicant to satisfy the requirements stated in the certification basis for an aircraft or engine. Examples include statements, drawings, analyses, calculations, testing, simulation, inspection, and environmental qualification. Advisory material issued by the certification authority is used if appropriate. [1]

Media - Devices or material which act as a means of transferal or storage of software, for example, programmable read-only memory, magnetic tapes or discs, and paper. [1]

Member Function - A method in C++. [17]

Memory device - An article of hardware capable of storing machine-readable computer programs and associated data. It may be an integrated circuit chip, a circuit card containing integrated circuit chips, a core memory, a disk, or a magnetic tape. [1]

Memory usage analysis - See: *other memory usage analysis* and *stack usage analysis*.

Message - In object-oriented programming, the invocation of an operation is referred to as a message, i.e., a message that identifies the operation being called and provides argument values for associated parameters.

Method - The implementation of an operation. A method specifies the algorithm or procedure associated with an operation. A method corresponds to a subprogram with a body in Ada 95, to a member function with a body in C++, and to a concrete method in Java. See: *operation*. [2], [16] Contrast: *multimethod.*

Modified Condition/Decision Coverage - Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to

independently affect that decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions. [1]

Monitoring - (1) [Safety] Functionality within a system which is designed to detect anomalous behavior of that system. (2) [Quality Assurance] The act of witnessing or inspecting selected instances of test, inspection, or other activity, or records of those activities, to assure that the activity is under control and that the reported results are representative of the expected results. Monitoring is usually associated with activities done over an extended period of time where 100% witnessing is considered impractical or unnecessary. Monitoring permits authentication that the claimed activity was performed as planned. [1]

Multimethod - A method invoked using multiple dispatch. Multimethods differ from ordinary methods in that the run-time classes of all parameters (rather than only the target object) are considered when selecting the method to be executed at the point of call. See: *method.*

Multiple dispatch - Dynamic dispatch based on the run-time types of all the arguments to a call, rather than only the run-time type of the target object. Contrast: *single dispatch*. [2]

Multiple inheritance - A semantic variation of generalization in which a type (a class) may have more than one supertype (superclass). Contrast: *single inheritanc*e. [16]

Multiple-version dissimilar software - A set of two or more programs developed separately to satisfy the same functional requirements. Errors specific to one of the versions are detected by comparison of the multiple outputs. [1]

Non-virtual - A C++ keyword that specifies that a given member function may be overridden in subclasses and calls to it are resolved at compile time (static binding).

Object - An entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships; behavior is represented by operations, methods, and state machines. An object is an instance of a class. See: *class, instance*. [16]

Object Code - A low-level representation of the computer program not usually in a form directly usable by the target computer but in a form which includes relocation information in addition to the processor instruction information. [1]

Object Management Group (OMG) - A standards body for the object-oriented development community. The membership includes all major object-oriented tool vendors, many companies offering OO training and consulting services, many companies offering COTS software, and many end users of OO technology, including several of the members of AVSI. The OMG defines interface standards for distributed object communication (e.g., CORBA) and for OO modeling tools (e.g., UML). [2]

Object-oriented (OO) - (1) the use of classes to support encapsulation, (2) the use of inheritance of interfaces to support subtyping, (3) the use of inheritance of implementation (state and code) to support subclassing, and (4) the use of dynamic dispatch (virtual method invocation) to support polymorphism and the inheritance of code. [2]

Operation - A service that can be requested of an object. An operation has a signature, which may restrict the actual parameters that are possible. An operation corresponds to a subprogram declaration in Ada 95, to a function member declaration in C++, and to an abstract method declaration in Java. It does not define an associated implementation. See *method*. [2], [16]

Other memory usage analysis - Analysis related to the sharing of resources between different software ‘partitions’. These forms of analysis include, but are not limited to, memory (heap), input/output (I/O) ports, and special purpose hardware, which perform specific computations or watchdog timer functions. [5]

Overloading - Use of the same name for different operators or behavioral features (operations or methods) visible within the same scope.

Overriding - The redefinition of an operation or method in a subclass. [2]

Parameter - The specification of a variable that can be changed, passed, or returned. A parameter may include a name, type, and direction. Parameters are used for operations, messages, and events. [16] See: *argument.*

Parent - In an inheritance relationship, the generalization of another element, producing the child. See: *superclass, supertype.* Contrast: *child, subclass, subtype*. Child classes inherit from their parent classes. Similarly, subclasses inherit from their superclasses. [2], [16]

Part number - A set of numbers, letters or other characters used to identify a configuration item. [1]

Partitioning - See: *software partitioning.*

Patch - A modification to an object program, in which one or more of the planned steps of re-compiling, re-assembling or re-linking is bypassed. This does not include identifiers embedded in the software product, for example, part numbers and checksums. [1]

Pattern - A documented solution to a commonly encountered analysis or design problem. Each pattern documents a single solution to the problem in a given context. [2] See: *process pattern*.

Polymorphism - A concept in type theory, according to which a name (such as a variable) may denote objects of many different classes that are related by some common superclass; thus, any object denoted by this name is able to respond to some common set of operations in different ways. [8]

Post-condition - A constraint that must be true at the completion of an operation. [16]

Pre-condition - A constraint that must be true when an operation is invoked. [16]

Process - A collection of activities performed in the software life cycle to produce a definable output or product. [1]

Process pattern - A documented solution to a problem with the software development process. A process pattern presents the problem, followed by a description of its solution in a given context. [2] See: *pattern.*

Product service history - A contiguous period of time during which the software is operated within a known environment, and during which successive failures are recorded. [1]

Proof of correctness - A logically sound argument that a program satisfies its requirements. [1]

Range checking - The verification that data values lie within specified ranges and maintain a specified accuracy. Range checking includes, but is not limited to, overflow and underflow analysis, the detection of rounding errors, range checking, and the checking of array bounds. [5]

Real-time system - A system that responds in a (timely) predictable way to unpredictable external stimuli arrivals. A real-time system has to fulfill under extreme load conditions including:

- timeliness: meet deadlines, it is required that the application has to finish certain tasks within the time boundaries it has to respect;
- simultaneity or simultaneous processing: more than one event may happen simultaneously, all deadlines should be met;
- predictability: the real-time system has to react to all possible events in a predictable way; and
- dependability or trustworthiness: it is necessary that the real-time system environment can rely on it. [6]

Relationship - A semantic connection among model elements. Examples of relationships include associations and generalizations. [16]

Release - The act of formally making available and authorizing the use of a retrievable configuration item. [1]

Repeated inheritance - The inheritance of an element via more than one path through the inheritance hierarchy.

Requirements-based testing - (1) Testing performed using test cases and procedures developed to confirm that the software performs its intended function as specified by its requirements. Requirements-based testing includes both normal range test cases, and robustness (abnormal range) test cases. The test cases are to be developed from the software requirements and the errors sources inherent in the software development process. [1] (2) Testing

performed with the objective of showing that the actual behavior of the program is in accordance with its requirements. Two common methods are cited for conducting requirements-based testing: equivalence class testing, and boundary value testing. [5] The use of the term in this Handbook is intended to encompass both definitions.

Requirements, Design, and Code Standards (R-D-C Standards) - Guidelines used to control, develop, and review software requirements, design, and code. [1]

Reusable software component (RSC) - The software, its supporting RTCA/DO-178B software life cycle data, and additional supporting documentation being considered for reuse. The component designated for reuse may be any collection of software, such as, libraries, operating systems, or specific system software functions. [3]

Reverse engineering - The method of extracting software design information from the source code. [1]

Robustness - The extent to which software can continue to operate correctly despite invalid inputs. [1]

Run-time type/class - The type/class associated with an object at run-time (e.g., when the object is first created). In Ada 95, this is the tag associated with objects of a tagged type. [2]

Scalar types - A type that defines a variable containing a single value at run-time. A scalar type is either a discrete type or a real type. The values of a scalar type are ordered.

Signature - The name and parameter types of an operation or method. A signature may include an optional returned parameter type (depending upon the target language). [16]

Simple dispatch - A restricted form of single dispatch, in which (a) all calls other than method extensions are dispatching, and (b) dispatch is semantically equivalent to invocation of a dispatch routine containing a case statement. [2]

Simulator - A device, computer program, or system used during software verification, that accepts the same inputs and produces the same output as a given system, using object code which is derived from the original object code. [1]

Single dispatch - Dynamic dispatch based on only the run-time type of the target object. Most OO languages, including Java, Ada 95 and C++ are single dispatching. Contrast: *multiple dispatch*. [16]

Single inheritance - A semantic variation of generalization in which a type (a class) may have only one supertype (superclass). Contrast: *multiple inheritanc*e. [16]

Software - Computer programs and, possibly, associated documentation and data pertaining to the operation of a computer system. [1]

Software architecture - The structure of the software selected to implement the software requirements. [1]

Software change - A modification in source code, object code, executable object code, or its related documentation from its baseline. [1]

Software integration - The process of combining code components. [1]

Software level - One of the software levels defined by DO-178B, section 2.2.2. Software level is based upon the contribution of software to potential failure conditions as determined by the safety assessment process. [1]

Software library - A controlled repository containing a collection of software and related data and documents designed to aid in software development, use or modification. Examples include software development library, master library, production library, program library and software repository. [1]

Software life cycle - (1) An ordered collection of processes determined by an organization to be sufficient and adequate to produce a software product. (2) The period of time that begins with the decision to produce or modify a software product and ends when the product is retired from service. [1]

Software partitioning - The process of separating, usually with the express purpose of isolating one or more attributes of the software, to prevent specific interactions and cross-coupling interference. [1]

Software product - The set of computer programs, and associated documentation and data, designated for delivery to a user. In the context of DO-178B, this term refers to software intended for use in airborne applications and the associated software life cycle data. [1]

Software requirement - A description of what is to be produced by the software given the inputs and constraints. Software requirements include both high-level requirements and low-level requirements. [1]

Software tool - A computer program used to help develop, test, analyze, produce or modify another program or its documentation. Examples are an automated design tool, a compiler, test tools and modification tools. [1]

Source code - Code written in source languages, such as assembly language and/or high level language, in a machine-readable form for input to an assembler or a compiler. [1]

Stack usage analysis - A form of shared resource analysis that establishes the maximum possible size of the stack required by the system and whether there is sufficient physical memory to support this stack size. Some compilers use multiple stacks, and this form of analysis is required for each stack. Potential stack-heap allocation collisions, when these forms of storage compete for the same space, are also included. [5]

Standard - A rule or basis of comparison used to provide both guidance in and assessment of the performance of a given activity or the content of a specified data item. [1]

State - A condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. [16]

Statement coverage - Every statement in the program has been invoked at least once. [1]

Static analyzer - A software tool that helps reveal certain properties of a program without executing the program. [1]

Static binding - With regard to calls to operations, the ability to associate a particular method (implementation of an operation) with a particular call to that operation at compile time.

Static classification - A semantic variation of generalization in which an object may not change [its] classifier. Contrast: *dynamic classification*. Using dynamic classification, the class of an object may change during its life time. Using static classification, it may not. [2], [16]

Strongly typed - A characteristic of a programming language, according to which all expressions are guaranteed to be type consistent. [8]

Strongly typed language - A strongly typed language associates a type with each data element (variable or expression), and ensures that only operations appropriate to that type are applied to the data element. Only meaningful conversions between logically related types are permitted. A subset of a language may be considered strongly typed, even if the full language is not.

Structural coverage - A software-program method of determining the adequacy of the extent of the verification accomplished in the composition/decomposition of the software program.

Structural coverage analysis - An analysis that (1) determines which software structures and code structures were not exercised by the requirements-based test procedures; and (2) provides traceability between the implementation of the software requirements in the code structure and the verification of those requirements via test cases. [1]

Structure - A specified arrangement or interrelation of parts to form a whole. [1]

Subclass - In a generalization relationship, the specialization of another class; the superclass (parent). See: *generalization, child*. Contrast: *superclass, parent.*

Note: *"subclass" and "child" are used interchangeably in object-oriented development.* [16]

Subinterface - A subclass/subtype that is an interface (defines no methods, associations, or modifiable attributes). See: *interface*.

Substitutability - With regard to subtyping, the ability to substitute an instance of a given subtype for an instance of one of its supertypes in any context in which the supertype instance may appear (see also Liskov substitution principle (LSP), which defines a set of typing rules intended to guarantee substitutability).

Substitution - With regard to subtyping, the replacement of an instance of a subtype with an instance of one of its supertypes.

Subtype - In a generalization relationship, the specialization of another type; the supertype. See: *generalizatio*n. Contrast: *supertyp*e. [2], [16]

Superclass - In a generalization relationship, the generalization of another class; the subclass. See: *generalization, parent.* Contrast: *subclas*s, *child.*

Note: *"superclass" and "parent" are used interchangeably in object-oriented development.* [16]

Superinterface - A superclass/supertype that is an interface (defines no methods, associations, or modifiable attributes). See: *interface*.

Supertype - In a generalization relationship, the generalization of another type; the subtype. See: *generalizatio*n. Contrast: *subtyp*e. [16]

System - A collection of hardware and software components organized to accomplish a specific function or set of functions. [1]

System architecture - The structure of the hardware and software selected to implement the system requirements. [1]

System safety assessment - An ongoing, systematic, comprehensive evaluation of the proposed system to show that relevant safety-related requirements are satisfied. [1]

System safety assessment process - Those activities which demonstrate compliance with airworthiness requirements and associated guidance material, such as, JAA AMJ/FAA AC 25.1309. The major activities within this process include: functional hazard assessment, preliminary system safety assessment, and system safety assessment. The rigor of the activities will depend on the criticality, complexity, novelty, and relevant service experience of the system concerned. [1]

Target computer - The physical processor that will execute the program while airborne. [3]

Target computer environment - The target computer and all its support hardware and systems needed to function in its actual airborne environment. [3]

Target environment - See: *target computer environment.* [3]

Target object - The object that is the *target* of a method call [most often written targetObject.methodName (argumentList);]. Dynamic dispatch typically involves selection of a method based on the declared types of the arguments and the run-time type of the target object. [2]

Task - The basic unit of work from the standpoint of a control program. [1]

Template - A parameterized model element with unbound (formal) parameters that must be bound to actual (type) parameters before it can be used. At a target language level, templates correspond to Ada generics and to C++ templates. [2]

Template class - A parameterized class. Template classes are implemented as generic packages in Ada, and to template classes in C++. Java does not currently support parameterized class definitions. [2]

Template operation - A parameterized operation or method. Template operations are referred to as generic subprograms in Ada, and as template member functions in C++. Java does not currently support parameterized class definitions. [2]

Test case - A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. [1]

Testing - The process of exercising a system or system component to verify that it satisfies specified requirements and to detect errors. [1]

Test procedure - Detailed instructions for the set-up and execution of a given set of test cases, and instructions for the evaluation of results of executing the test cases. [1]

Timing analysis - A form of analysis to establish the temporal properties of the input/output dependencies. A common and important aspect of this analysis is the worst-case execution time for the correct behavior of the overall system. Certain languages offer features that make timing analysis difficult, e.g., loops without static upper bounds and the manipulation of dynamic data structures. [5]

Tool qualification - The process necessary to obtain certification credit for a software tool within the context of a specific airborne system. [1]

Traceability - The evidence of an association between items, such as between process outputs, between an output and its originating process, or between a requirement and its implementation. [1]

Transition criteria - The minimum conditions, as defined by the software planning process, to be satisfied to enter a process. [1]

Type - The concepts of type and class are in general distinguished, with a type representing an abstraction implemented by one or more classes. In most of the classical object-oriented programming languages this distinction is not performed. [4] In UML and languages such as Java, however, a distinction is made between interface types (abstractions) and class types, which implement them.

Type conversion - The act of producing a representation of some value of a target type from a representation of some value of a source type. Type conversion is used to resolve mismatched types in assignments, expressions, or when passing parameters. Type conversions may be either implicit or explicit. With implicit type conversion the compiler is given the responsibility for determining that a conversion is required and how to perform the conversion. With explicit type conversion the programmer assumes the responsibility. Synonym: *conversion.*

Unchecked type conversion - Types are unchecked if conversion from one type to the other does not include a determination either at compiler time or run-time as to whether they are normally convertible.

Unified Modeling Language (UML) - A language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems. [16]

Use case - The specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system. [16]

Validation - The process of determining that the requirements are the correct requirements and that they are complete. The system life cycle process may use software requirements and derived requirements in system validation. [1]

Variables - Named memory locations that contain data that may change during software execution.

Verification - The evaluation of the results of a process to ensure correctness and consistency with respect to the inputs and standards provided to that process. [1]

Virtual - For C++, “virtual” is a keyword that specifies a given method (member function) may be overridden in subclasses, and that calls to it are dispatching.

Weakly typed - Strict enforcement of type rules but with well-defined exceptions or an explicit type-violation mechanic, e.g., operations on a data element are not restricted to those defined for its logical type (Booleans can be added to one another), or automatic conversions between logically unrelated types are permitted with no change in data representation (integer values are converted to floating point values). Contrast: *strongly typed.*

Weakly typed language - A programming language or modeling language that is weakly typed. Contrast: *strongly typed language*.

1.6 OOTiA Workshops

NASA and FAA sponsored two OOTiA workshops for the purposes of:

- Identifying safety and certification issues related to using OOT;
- Coordinating and communicating with industry, government, and academia on OOT; and
- Working together to establish positions on the key OOT issues.

The workshops were led and coordinated by a workshop committee and were attended by international representatives from government, industry, and academia (see section 1.6.1 for a list of workshop committee members and sections 1.6.2 and 1.6.3 for lists of workshop participants). OOTiA Workshop #1 was held in April 2002 and was followed by Workshop #2 in March 2003.

1.6.1 Workshop Committee

Participant	Affiliation
J. Chilenski	Boeing Commercial Airplanes
G. Daugherty	Rockwell Collins, Inc.
K. Hayhurst	NASA Langley Research Center
C. Kilgore	FAA Technical Center (AAR-470)
J. Knickerbocker	Sunrise Certification & Consulting, Inc.
J. Lewis	FAA Headquarters (AIR-120)
B. Lingberg	FAA Headquarters (AIR-120)
S. Obeid	embeddedPlus Engineering
B. Ocker	FAA Chicago Aircraft Certification Office (ACE-117C)
T. Rhoads	Goodrich
L. Rierson	FAA Chief Scientific and Technical Advisor (AIR-106N)
W. Schultz	Honeywell
W. Struck	FAA Transport Directorate (ANE-111)
D. Wallace	FAA Ft. Worth Aircraft Certification Office (ASW-170)

1.6.2 Participants in Workshop #1

Participant	Affiliation
K. Achenbach	Rolls-Royce Corporation
M. Almesåker	Saab AB Sweden
J. Angermayer	MITRE
J. Auld	NovAtel Inc.
I. Baxter	Semantic Designs
A. Bell	The Boeing Company
D. Bernier	Rockwell Collins

B. Bianchi	Ametek
B. Bogdan	Computer Science Corporation
M. Brennan	Applied Microsystems
D. Brown	Rolls-Royce plc
V. Brown	Honeywell, Inc.
R. Butler	NASA Langley Research Center
B. Calloni	Lockheed Martin Aeronautics Company
R. Calloway	NASA Langley Research Center
S. Chappell	Computer Science Corporation
J. Chelini	Verocel, Inc.
J. Chilenski	Boeing Commercial Airplanes
M. Christie	Universal Avionics Systems Corporation
J. Coleman	Hamilton Sundstrand
O. Collins	Raytheon - IATC
M. Consiglio	ICASE
M. Cors	Goodrich Avionics Systems
J. Daly	TRW (Aeronautical Systems)
G. Daugherty	Rockwell Collins, Inc.
R. Deal	Honeywell
D. DeHoff	Raytheon Technical Services Company
M. DeWalt	Certification Services, Inc.
V. Dovydaitis	Foliage Software Systems, Inc.
P. Dunn	Northrop Grumman Commercial Nav Systems
G. Edmands	The MITRE Corporation
E. Edora	Solers, Inc.
C. Erwin	FAA Wichita Aircraft Certification Office (ACE-115)
T. Ferrell	FAA Consulting
U. Ferrell	FAA Consulting
G. Finelli	NASA Langley Research Center
S. Fischer	LITEF GmbH, Germany
L. Framarini	BAE Systems
D. Geis	Goodrich Avionics Systems
G. Graessle	Honeywell, Intl.
S. Grainger	Marinvent Corporation
M. Gulick	Solers, Inc.
T. Hammer	Honeywell
K. Hayhurst	NASA Langley Research Center
M. Haynes	Marinvent Corporation
R. Hirt	Raytheon Aircraft Company

M. Holloway	NASA Langley Research Center
G. Horan	FAA Engine Directorate (ANE-110)
M. Isaacs	FAA Mike Monroney Aeronautical Center (AOS-240)
D. Johnson	Astronautics Corporation of America
R. Johnson	Bell Helicopter
M. Jones	NovAtel Inc.
G. Kelly	Honeywell
C. Kilgore	FAA Technical Center (AAR-421)
J. Klein	Lockheed Martin Air Traffic Management
J. Knickerbocker	Sunrise Certification & Consulting, Inc.
J. Knight	University of Virginia
T. Lambregts	FAA Chief Scientific and Technical Advisor (ANM-113N)
P. Lawrence	Boeing
J. Lee	Boeing
Y. Lee	Arizona State University
S. Leichtnam	Computer Science Corporation
J. Lewis	FAA Headquarters (AIR-120)
B. Lingberg	FAA Headquarters (AIR-120)
J. Masalskis	Boeing
J. Mason	Boeing
G. Millican	Honeywell
J. Monagan	Rockwell Collins
J. Monfret	BarcoView
B. Moody	USAF
B. Newman	Astronautics Corporation of America
S. Obeid	embeddedPlus Engineering
B. Ocker	FAA Chicago Aircraft Certification Office (ACE-117C)
A. Oswald	MITRE/CAASD
C. Paganoni	SAIC
M. Patel	WPAFB
G. Pavlin	Brightline Avionics GmbH
L. Peckham	NASA Langley Research Center
T. Petroski	Boeing
M. J. Peuser	Honeywell
C. Pohlman	Lockheed Martin Aeronautics Company
G. Putsche	Boeing
H. Quach	Lockheed Martin Corporation
R. Rader	Lockheed Martin
R. Randall	Boeing Wichita Modification & Maintenance Center

T. Reeve	Patmos Engineering Services
T. Rhoads	Goodrich FUS
W. Rieger	Boeing
L. Rierson	FAA Chief Scientific and Technical Advisor (AIR-106N)
K. Rigby	BAE Systems
B. Rivet	Hamilton Sundstrand
D. Robinson	FAA Headquarters (AIR-130)
C. Rosay	JAA-CEAT
T. Roth	Honeywell International Inc.
V. Santhanam	Boeing Wichita Development & Modification Center
T. Schavey	Smiths Aerospace
S. M. Schedra	Wind River Systems, Inc.
W. Schultz	Honeywell International
M.l Smith	Ametek
F. Sogandares	MITRE/CAASD
M. Sonnek	Honeywell
R. Souter	FAA Wichita Aircraft Certification Office (ACE-116W)
C. Spitzer	AvioniCon
E. Startzman	Boeing Wichita Development & Modification Center
D. Stephens	Boeing
W. Struck	FAA Transport Directorate (ANM-111)
D. Sungenis	Computer Science Corporation
A. Theodore	UNITECH
H. Thomas	Honeywell, Inc.
M. Valentin	Airbus France
J. Van Leeuwen	United Technologies - Sikorsky Aircraft
D. Wallace	FAA Ft. Worth Aircraft Certification Office (ASW-170)
D. Woodward	BAE Systems
P. Wright	Boeing

1.6.3 Participants in Workshop #2

Participant	**Affiliation**
G. Adams	Lockheed Martin Aero
J. Angermayer	The MITRE Corp.
J. Auld	NovAtel
F. Barber	Avidyne Corporation
B. Bianchi	Ametek

T. Bihari	AMT Systems Engineering, Inc.
R. Bogdan	Computer Sciences Corporation
F. Bortkiewicz	The Boeing Company
M. Brennan	Metrowerks Corporation
E. Brockway	Lockheed Martin
B. Brosgol	Ada Core Technologies, Inc.
D. Brown	Rolls-Royce plc
J. Burck	Smiths Aerospace - Electronic Systems
B. Cain	METI
J. Carlton	Escher Technologies
K. Carroll	Lockheed Martin Aero
P. Catlin	Goodrich Avionics Systems, Inc.
R. Chapman	Praxis Critical Systems Limited
R. Charley	The Boeing Company
J. Chelini	Verocel, Inc.
J. Chilenski	The Boeing Company
E. Chiuchiolo	FAA
K. Clegg	University of York
D. Coleman	MDHI
J. Coleman	Hamilton Sundstrand
M. Consiglio	NASA Langley Research Center
M. Cors	Goodrich Avionics Systems
D. Crocker	Escher Technologies
E. Danielson	Rockwell Collins
G. Daugherty	Rockwell Collins
T. Deaver	Northrop Grumman
L. Demeestere	BarcoView
M. DeWalt	Certification Services, Inc.
B. Dulic	Transport Canada
G. Edmands	The MITRE Corporation
C. Erwin	FAA
D. Fisher	Ada Core Technologies, Inc.
G. Frye	FAA/AIR-130
R. Fulton	Honeywell
E. Galiana	CMC Electronics
D. Geis	Goodrich Avionics Systems
C. Gibson	Honeywell
F. Guay	FWGC
M. Gulick	Solers, Inc.

D. Hatfield	FAA
R. Hawkins	University of York
M. Hawthornthwaite	Engenuity Technologies
K. Hayhurst	NASA Langley Research Center
M. Haynes	Marinvent Corporation
B. Hendrix	Lockheed Martin Aeronautics Company
R. Hirt	FAA
T. Hofmann	Diehl Avionik Systeme GmbH
M. Holloway	NASA Langley Research Center
S. Hutchesson	Rolls-Royce plc
M. Jones	NovAtel
R. Key	FAA
J. Kilchert	Diehl Avionik Systeme GmbH
J. Knickerbocker	Sunrise Certification and Consulting
J. Knight	University of Virginia
P. La Pietra	Honeywell
T. Lambregts	FAA
J. D. Lawrence	DRPM AAA
J. Lewis	FAA
B. Lingberg	FAA
J. Liscouski	BAE Systems
P. Maneely	Honeywell
E. Mannisto	Honeywell
S. Matthews	Avidyne Corporation
D. Mayerhoefer	Green Hills Software
M. Mehlich	Semantic Designs, Inc.
B. Mierow	Hamilton Sundstrand
G. Millican	Honeywell
S. Morton	Applied Dynamics International
S. Obeid	Embedded Plus Engineering
J. Offutt	George Mason University
R. Oracheff	Paragon Transportation LLC
L. Peckham	NASA Langley Research Center
M. Peuser	Honeywell
S. Ray	BAE Systems Controls
T. Reeve	Patmos Engineering. Services
B. Reynolds	Rockwell Collins
T. Rhoads	Goodrich
W. Richter	Gulfstream

L. Rierson	FAA
B. Rivet	Hamilton Sundstrand - UTC
D. Robinson	FAA
C. Rosay	JAA/CEAT
T. Roth	Honeywell
W. Ryan	FAA
L. Schad-Alford	The Boeing Company
K. Schlatter	Jeppesen
E. Schonberg	Ada Core Technologies, Inc.
V. Shapiro	AMTI
T. Smith	Air Traffic Software Architecture
C. Spitzer	AvioniCon
R. Stanley	Air Traffic Software Architecture
E. Startzman	The Boeing Company
J. Steidl	Astronautics Corporation of America
E. Strunk	University of Virginia
T. Swinehart	Goodrich Avionics Systems, Inc.
B. Thedford	Hanscom AFB
A. Theodore	Unitech
L. Thompson	Honeywell
M. Valentin	AIRBUS France
J. Van Leeuwen	Sikorsky Aircraft
D. Wallace	FAA
P. Whiston	High Integrity Solutions Ltd
M. Whitehurst	The Boeing Company
A. Wils	K.U. Leuven
M. Wittman	Honeywell
D. Woodward	BAE Systems
P. Wright	The Boeing Company

1.7 References

1. RTCA, Inc., *Software Considerations in Airborne Systems and Equipment Certification*, RTCA/DO-178B, December 1992, Washington, D.C.

2. Aerospace Vehicle Systems Institute. *Guide to the Certification of Systems with Embedded Object-Oriented Software*, version 1.5.

3. FAA AC 20-148, *Reusable Software Components*, final (unsigned) version dated August 3, 2004.

4. Laplante, Phillip A (editor). *Dictionary of Computer Science*, CRC Press LLC, 2001.

5. *Guidance for the Use of Ada in High Integrity Systems*, ISO/IEC TR 15942:2000 see http://isotc.iso.ch/livelink/livelink/fetch/2000/2489/Ittf_Home/ITTF.htm.

6. *Dedicated Systems Encyclopedia*, available from http://www.dedicated-systems.com/encyc/techno/terms/defini/def.htm.

7. Object-Oriented Technology in Aviation Program website: http://shemesh.larc.nasa.gov/foot/.

8. Booch, Grady. *Object-Oriented Analysis and Design*. Addison-Wesley, 2nd edition, 1994.

9. "Glossary of Software Engineering Terminology." ANSI/IEEE Standard, 1983.

10. Gomaa, Hassan. *Software Design Methods for Concurrent and Real-time Systems*. Addison-Wesley, 1993.

11. Hathaway, Bob. "Frequently Asked Questions on Object-Oriented." Web site: http://www.cs.cmu.edu/Groups/AI/html/faqs/lang/oop/faq-doc-0.html.

12. Meyer, Bertrand. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

13. Montlick, Terry. "What is Object-Oriented Software?" Web site: http://www.softwaredesign.com/.

14. Pressman, Roger. *Software Engineering: A Practitioner's Approach*. McGraw Hill, 4th edition, 1997.

15. Schildt, Herbert. *Teach Yourself C++*. McGraw Hill, 1998.

16. Object Management Group. *OMG Unified Modeling Language Specification*, version 1.3, June 1999, available from http://www.omg.org/technology/documents/vault.htm#modeling.

17. Liskov, Barbara and Jeanette Wing. "A Behavioral Notion of Subtyping," *ACM Transactions on Programming Languages and Systems*, 16(6): 1811-1841, November 1994.

18. FOLDOC: *Free Online Dictionary of Computing*, http://foldoc.doc.ic.ac.uk.

1.8 Feedback Form

U.S. Department
of Transportation

**Federal Aviation
Administration**

Feedback Information

Please submit any written comments or recommendations for improving this Handbook. You may also suggest new items or subjects to be added. And, if you find an error, please tell us about it. Send to: FAA, AIR-120, Room 815, 800 Independence Ave., S.W., Washington, DC 20591.

Subject: Handbook for Object-Oriented Technology in Aviation (OOTiA)

To: OOTiA Handbook POC, FAA/AIR-120, Software Program Manager

(*Please check all appropriate line items)*

☐ An error has been noted in Volume __, section _____, paragraph _____, on page _____ .

☐ Recommend Volume __, section _____, paragraph _____ on page _____ be changed as follows:
(attach separate sheet if necessary)

☐ In a future change to this Handbook, please include coverage on the following subject:
(briefly describe what you want added):

☐ Other comments:

☐ I would like to discuss the above. Please contact me.

Submitted by: _____ Date: _____

Phone: _____ Address: _____

Email: _____ Routing Symbol (if applicable): _____

NOTE: This Handbook does not constitute Federal Aviation Administration (FAA) policy or guidance, nor is it intended to be an endorsement of object-oriented technology (OOT). This Handbook is not to be used as a standalone product but, rather, as input when considering issues in a project-specific context.