

**DRAFT**

# **Handbook for Object-Oriented Technology in Aviation (OOTiA)**

**Volume 3: Best Practices**

**vPC.0**

**January 30, 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

3.1	INTRODUCTION.....	1
3.1.1	<i>Purpose</i> .....	1
3.1.2	<i>Organization</i> .....	1
3.2	MAPPING OF VOLUME 2 ISSUES TO VOLUME 3 GUIDELINES.....	2
3.2.1	<i>Key Concerns/Issues Addressed by the Guidelines</i> .....	2
3.3	SINGLE INHERITANCE AND DYNAMIC DISPATCH.....	11
3.3.1	<i>Purpose</i> .....	11
3.3.2	<i>Background</i> .....	11
3.3.3	<i>Overall Approach</i> .....	13
3.3.4	<i>Inheritance with Overriding</i> .....	13
3.3.5	<i>Method Extension</i> .....	16
3.3.6	<i>Subtyping</i> .....	17
3.3.7	<i>Formal Subtyping</i> .....	18
3.3.8	<i>Unit Level Testing of Substitutability</i> .....	19
3.3.9	<i>System Level Testing of Substitutability Using Assertions</i> .....	21
3.3.10	<i>System Level Testing of Substitutability Using Specialized Test Cases</i> .....	23
3.3.11	<i>Class Coupling</i> .....	24
3.3.12	<i>Deep Hierarchy</i> .....	26
3.4	MULTIPLE INHERITANCE.....	28
3.4.1	<i>Purpose</i> .....	28
3.4.2	<i>Background</i> .....	28
3.4.3	<i>Overall approach</i> .....	28
3.4.4	<i>Multiple Interface Inheritance</i> .....	29
3.4.5	<i>Multiple Implementation Inheritance</i> .....	30
3.4.6	<i>Mixed Multiple Inheritance</i> .....	31
3.4.7	<i>Combination of Distinct Abstractions</i> .....	32
3.4.8	<i>Top Heavy Hierarchy</i> .....	33
3.5	TEMPLATES.....	35
3.5.1	<i>Purpose</i> .....	35
3.5.2	<i>Background</i> .....	35
3.5.3	<i>Source Code Review</i> .....	35
3.5.4	<i>Requirements-based Test Development, Review, and Coverage</i> .....	36
3.5.5	<i>Structural Coverage for Templates</i> .....	36
3.6	INLINING.....	38
3.6.1	<i>Purpose</i> .....	38
3.6.2	<i>Background</i> .....	38
3.6.3	<i>Inlining and Structural Coverage</i> .....	38
3.6.4	<i>Source Code Review of Inlined Code</i> .....	39
3.7	TYPE CONVERSION.....	40
3.7.1	<i>Purpose</i> .....	40
3.7.2	<i>Background</i> .....	40
3.7.3	<i>Overall approach</i> .....	40
3.7.4	<i>Source Code Review, Checklist, and Coding Standards</i> .....	40
3.7.5	<i>Loss of Precision in Type Conversions</i> .....	41
3.7.6	<i>Type Conversions of References and Pointers</i> .....	41
3.7.7	<i>Language specific guidelines</i> .....	42
3.8	OVERLOADING AND METHOD RESOLUTION.....	43
3.8.1	<i>Purpose</i> .....	43
3.8.2	<i>Background</i> .....	43
3.8.3	<i>Code Review Method</i> .....	43
3.8.4	<i>Implicit Conversion</i> .....	44
3.9	DEAD AND DEACTIVATED CODE, AND REUSE.....	45
3.9.1	<i>Purpose</i> .....	45

3.9.2	<i>Background</i> .....	45
3.9.3	<i>Reuse of Software Components</i> .....	45
3.9.4	<i>Requirements Traceability</i> .....	47
3.9.5	<i>Certification Credit for Reused but Modified Class Hierarchy</i> .....	48
3.9.6	<i>Changes in the Status of Deactivated Code Versus Actively Used Code</i> .....	48
3.9.7	<i>Service History Credit and Deactivated Code</i> .....	49
3.10	<b>OBJECT-ORIENTED TOOLS</b> .....	50
3.10.1	<i>Purpose</i> .....	50
3.10.2	<i>Background</i> .....	50
3.10.3	<i>Traceability When Using OO Tools</i> .....	50
3.10.4	<i>Configuration Management When Using Visual Modeling Tools</i> .....	50
3.10.5	<i>Visual Modeling Tools Frameworks</i> .....	51
3.10.6	<i>Automatic Code Generators</i> .....	51
3.10.7	<i>Structural Coverage Analysis Tools</i> .....	53
3.10.8	<i>Structural Coverage Analysis for Inheritance</i> .....	53
3.10.9	<i>Structural Coverage Analysis for Dynamic Dispatch</i> .....	53
3.11	<b>TRACEABILITY</b> .....	55
3.11.1	<i>Purpose</i> .....	55
3.11.2	<i>Scope/Background</i> .....	55
3.11.3	<i>Overall approach</i> .....	55
3.11.4	<i>Tracing to Functional Requirements</i> .....	56
3.11.5	<i>Complex Class Hierarchies and Relationships</i> .....	57
3.11.6	<i>OO Design Notation and Traceability Ambiguity</i> .....	58
3.11.7	<i>Traceability and Dynamic Binding/Overriding</i> .....	58
3.11.8	<i>Dead and Deactivated Code</i> .....	58
3.11.9	<i>Many to Many Mapping of Requirements to Methods</i> .....	59
3.11.10	<i>Iterative Development</i> .....	59
3.11.11	<i>Change Management for Reusable Components</i> .....	59
3.12	<b>STRUCTURAL COVERAGE</b> .....	60
3.12.1	<i>Purpose</i> .....	60
3.12.2	<i>Background</i> .....	60
3.12.3	<i>Overall approach</i> .....	60
3.12.4	<i>Structural Coverage of Inheritance</i> .....	60
3.12.5	<i>Polymorphism with Dynamic Dispatch</i> .....	63
3.12.6	<i>Data Coupling and Control Coupling</i> .....	65
3.13	<b>REFERENCES</b> .....	66
3.14	<b>INDEX OF TERMS</b> .....	68
<b>APPENDIX A FREQUENTLY ASKED QUESTIONS (FAQS)</b> .....		<b>70</b>
<b>APPENDIX B EXTENDED GUIDELINES AND EXAMPLES</b> .....		<b>72</b>
B.1	<b>SINGLE INHERITANCE</b> .....	72
B.1.1	<i>Extension of the Inheritance with Overriding Guidelines</i> .....	72
B.1.2	<i>Extension of the Method Extension Guidelines</i> .....	76
B.2	<b>MULTIPLE INHERITANCE</b> .....	78
B.2.1	<i>Composition involving multiple inheritance</i> .....	78
B.2.2	<i>Extended guidelines</i> .....	84
B.3	<b>DEAD AND DEACTIVATED CODE, AND REUSE</b> .....	86
B.3.1	<i>Deactivated Code Examples</i> .....	86
B.3.2	<i>Hierarchy Changes and Method Overriding</i> .....	87

## Figures

<i>Figure 3.4-1 Combination of Distinct Abstractions .....</i>	<i>32</i>
<i>Figure 3.10-1 Code Generation using Visual Modeling Tools.....</i>	<i>52</i>
<i>Figure 3.11-1 Overview of Traceability .....</i>	<i>57</i>
<i>Figure 3.12-1 Inheritance.....</i>	<i>61</i>
<i>Figure 3.12-2 Concrete Coverage.....</i>	<i>62</i>
<i>Figure 3.12-3 Context Coverage.....</i>	<i>62</i>
<i>Figure 3.12-4 Flattened Inheritance .....</i>	<i>63</i>
<i>Figure 3.12-5 Dynamic Dispatch .....</i>	<i>64</i>

## Tables

<i>Table 3.2-1 Mapping of Key Concerns and Guidelines.....</i>	<i>10</i>
<i>Table 3.12-1 Hierarchical Incremental Testing Summary .....</i>	<i>61</i>

## 3.1 Introduction

This volume has been written to help developers and certification authorities identify current best practices for the use of object-oriented technology (OOT) in aviation. As OOT in embedded and safety critical systems is still an evolving discipline, additional information that was not available when this material was compiled may be available in the future. In any case, the OOT standards and methods the developer intends to use should be documented in the planning process documents and presented to the certification authorities as early as possible in the program to reduce risk.

### 3.1.1 Purpose

The purpose of this volume is to identify best practices to safely implement OOT in aviation by providing some known ways to address the issues documented in Volume 2. The guidelines presented in this volume are not necessarily the only way to address these issues. There may also be other means that are effective, and the handbook may itself define alternative ways to resolve a given issue. In some areas, this volume does not provide best practices but, rather, cites areas of ongoing research that should be monitored by prospective OOT developers. Such areas of research occur most notably in the areas control and data coupling, and structural coverage analysis as applied to OOT in aviation. In all cases, it should be noted that it is still the developer's responsibility to demonstrate that the OOT methods and processes they have selected to utilize can, and will, provide the appropriate integrity for safe software implementation.

### 3.1.2 Organization

This volume is organized in sections as follows:

Introduction

Mapping of Volume 2 Issues to Volume 3 Guidelines

Guidelines for:

- 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

Appendix A Frequently Asked Questions

Appendix B Extended Guidelines And Examples

## 3.2 Mapping of Volume 2 Issues to Volume 3 Guidelines

### 3.2.1 Key Concerns/Issues Addressed by the Guidelines

The following table provides a mapping between the key concerns with associated issues in volume 2 and the volume 3 guidelines that address them. Footnotes are provided when an explanation of the mapping (the manner in which the guidelines address a particular concern or a specific issue) is required.

When volume 3 provides alternative ways to address a key concern, more than one row appears in the Guidelines column opposite the key concern in the table. The key concern can then be addressed by following the guidelines listed in any one of these rows. That is, only one guideline of those listed (separated by OR) needs to be followed to address the concern. The exception is where a section identifies an area of active research where no guidelines currently exist.

Key concern	IL #'s	Guidelines
Volume 2, section 2.3.1.1 How does the life cycle data from an OO development process map to the life cycle data specified in DO-178B?	77, 87	Section 3.10.3 Section 3.11.4.1 Section 3.11.6.1
Volume 2, section 2.3.1.2 Are OO approaches adequate to define all types of requirements? Specifically, can we capture all nonfunctional requirements of interest? And can we avoid problems associated with graphical grouping?	63, 75, 78, 79, 80	Section 3.10.3 Section 3.11.5.1 Section 3.11.6.1 Section 3.11.9.1 IL 78 and IL 80 may not be adequately addressed in Volume 3.
Volume 2, section 2.3.1.2 A well-define means to map formal specifications to natural language and/or other less formal notations (e.g. UML) is needed to make formal specifications generally accessible.	73	Not addressed in Volume 3.
Volume 2, section 2.3.1.3 Have language features such as multiple inheritance been evaluated carefully in the planning process? And have appropriate restrictions been established, documented, and followed?	38, 58	Other issue list entries on multiple interface and multiple implementation inheritance may elaborate on the underlying issues that motivate IL 38. IL 38 and IL 58 may not be adequately addressed in Volume 3.
Volume 2, section 2.3.2.1.1 Is subtyping used to improperly define types that are not substitutable for their parent types?	17, 22, 23, 42, 90, 95	Section 3.3.4.3, <i>Simple overriding rule</i> :, Section 3.3.4.3, <i>Complete initialization rule</i> :, Section 3.3.4.3, <i>Initialization dispatch rule</i> :, Section 3.3.4.3, <i>Accidental override rule</i> :, Section 3.3.4.3, <i>Simple dispatch rule</i> :, Section 3.3.6.3, Section 3.3.7.3 (optional), and Section 3.3.8.3

		<p>OR</p> <p>Section 3.3.4.3, <i>Simple overriding rule</i>:,                  Section 3.3.4.3,  <i>Complete initialization rule</i>:,                  Section 3.3.4.3, <i>Initialization dispatch rule</i>:,                  Section 3.3.4.3, <i>Accidental override rule</i>:,                  Section 3.3.4.3, <i>Simple dispatch rule</i>:,                  Section 3.3.6.3,                  Section 3.3.7.3, and                  Section 3.3.9.3</p>
		<p>OR</p> <p>Section 3.3.4.3, <i>Simple overriding rule</i>:,                  Section 3.3.4.3,  <i>Complete initialization rule</i>:,                  Section 3.3.4.3, <i>Initialization dispatch rule</i>:,                  Section 3.3.4.3, <i>Accidental override rule</i>:,                  Section 3.3.4.3, <i>Simple dispatch rule</i>:,                  Section 3.3.6.3,                  Section 3.3.7.3 (optional), and                  Section 3.3.10.3</p>
<p>Volume 2, section 2.3.2.1.2</p> <p>How do we ensure that new methods defined by a subclass do not introduce anomalous behavior by producing a state inconsistent with that defined by the superclass?</p>	<p>91</p>	<p>Same as Volume 2, section 2.3.2.1.1 (above)</p>
<p>Volume 2, section 2.3.2.2.1</p> <p>How do we ensure that the developer's intent is always clear when using OO features such as multiple inheritance, and when creating very deep inheritance hierarchies?</p>	<p>7, 10, 15, 21, 24, 25, 27, 28, 29, 30, 33, 37</p>	<p>Section 3.3.12.3,                  Section 3.4.4.3,                  Section 3.4.5.3,                  Section 3.4.6.3, and                  Section 3.4.8.3</p> <p>OR</p> <p>Section 3.3.12.3,                  Section 3.4.4.3,                  Section 3.4.6.3,                  Section 3.4.7.3, and                  Section 3.4.8.3</p> <p>In addition, the following section identifies an area of active research related to the developer's understanding of data and control coupling between inherited methods: Section 3.12.6.1</p>

<p>Volume 2, section 2.3.2.2.2</p> <p>How do we avoid errors associated with the unintentional/accidental overriding of methods?</p>	<p>20, 31, 92, 93, 94, 96, 97, 99</p>	<p>Section 3.3.4.3, <i>Simple overriding rule</i>:,                  Section 3.3.4.3, <i>Accidental override rule</i>:,                  Section 3.3.4.3, <i>Simple dispatch rule</i>:,                  Section 3.3.4.3,  <i>Complete initialization rule</i>:,                  Section 3.3.4.3, <i>Initialization dispatch rule</i>:,                  Section 3.3.5.3,                  Section 3.3.6.3,                  Section 3.3.7.3 (optional), and                  Section 3.3.8.3</p> <p>OR</p> <p>Section 3.3.4.3, <i>Simple overriding rule</i>:,                  Section 3.3.4.3, <i>Accidental override rule</i>:,                  Section 3.3.4.3, <i>Simple dispatch rule</i>:,                  Section 3.3.4.3,  <i>Complete initialization rule</i>:,                  Section 3.3.4.3, <i>Initialization dispatch rule</i>:,                  Section 3.3.5.3,                  Section 3.3.6.3,                  Section 3.3.7.3, and                  Section 3.3.9.3</p> <p>OR</p> <p>Section 3.3.4.3, <i>Simple overriding rule</i>:,                  Section 3.3.4.3, <i>Accidental override rule</i>:,                  Section 3.3.4.3, <i>Simple dispatch rule</i>:,                  Section 3.3.4.3,  <i>Complete initialization rule</i>:,                  Section 3.3.4.3, <i>Initialization dispatch rule</i>:,                  Section 3.3.5.3,                  Section 3.3.6.3,                  Section 3.3.7.3 (optional), and                  Section 3.3.10.3</p>
<p>Volume 2, section 2.3.2.2.2</p> <p>How do we avoid the confusion and human error associated with the definition of overloaded operations that have the same name but different semantics?</p>	<p>60</p>	<p>Sections 3.8.3.2 and 3.8.4.2</p>
<p>Volume 2, section 2.3.2.3.1</p> <p>Traditional allocation and deallocation</p>	<p>66</p>	<p>Issue is not specific to OOT.</p>

algorithms may be unpredictable in terms of their worst-case memory use and execution times, resulting in indeterminate execution profiles.		Not addressed in Volume 3.
Volume 2, section 2.3.2.3.2 How do we ensure that subclass methods are not called by superclass constructors before all the attributes of the subclass have been initialized?	19, 98	Section 3.3.4.3, <i>Complete initialization rule</i> :, Section 3.3.4.3, <i>Initialization dispatch rule</i> :, Section 3.3.5.3, and Section 3.3.12.3
Volume 2, section 2.3.2.4.1 How do we identify dead and deactivated code in OOT programs and reusable components?	70, 71, 106	Section 3.11.8.1, Section 3.9.4.2, and Section 3.11.11.1
Volume 2, section 2.3.2.4.2 How do we ensure that deactivated code is properly addressed when working with general purpose libraries and frameworks?	1, 57	Sections 3.9.3.2, 3.11.8.1, and 3.11.11.1
Volume 2, section 2.3.3.1.1 Dynamic dispatch, polymorphism, multiple implementation inheritance, and inlining may complicate data and control flow analysis.	2, 9, 16, 43, 56, 89	Section 3.3.4.2, assumption 3, Section 3.3.4.3, <i>Simple dispatch rule</i> : , Section 3.4.5.3, Section 3.6.3.2, and Section 3.10.9.2
		OR Section 3.3.4.2, assumption 3, Section 3.3.4.3, <i>Simple dispatch rule</i> :, Section 3.4.7.3, Section 3.6.3.2, and Section 3.10.9.2
		<i>Note</i> : In addition, the following section identifies an area of active research: Section 3.12.6.1
Volume 2, section 2.3.3.1.2 How do we account for dynamic dispatch and the run time classes of objects when measuring the structural coverage of object-oriented program?	5, 11, 48, 49, 55	Section 3.3.4.2, assumption 3. Section 3.3.4.3, <i>Simple dispatch rule</i> : Section 3.3.6.3, <i>Substitutability compliance rule</i> : Section 3.3.8
		OR Section 3.3.4.2, assumption 3. Section 3.3.4.3, <i>Simple dispatch rule</i> : Section 3.3.6.3, <i>Substitutability compliance rule</i> : Section 3.3.9

		OR Section 3.3.4.2, assumption 3. Section 3.3.4.3, <i>Simple dispatch rule</i> : Section 3.3.6.3, <i>Substitutability compliance rule</i> : Section 3.3.10
Volume 2, section 2.3.3.1.2 How do we measure structural coverage when inlining and templates are used?	45, 47, 52	Sections 3.5.5.1.2, 3.5.5.3.2, and 3.6.3.2
Volume 2, section 2.3.3.1.3 How do we avoid problems related to timing analysis when using dynamic dispatch?	3, 107	Section 3.3.4.3, <i>Dispatch time rule</i> :
Volume 2, section 2.3.3.1.3 How do we avoid problems related to timing analysis when using inlining, templates, and macro expansion?	44, 50, 53	Sections 3.5.3.3, 3.5.4.3, and 3.6.3.2
Volume 2, section 2.3.3.1.4 How do we provide source to object code traceability when using dynamic dispatch?	6, 8, 12	Section 3.3.4.3, <i>Object code traceability rule</i> :
Volume 2, section 2.3.3.1.4 How do we provide source to object code traceability in non-level A systems?	81	Source to object code traceability is not required by DO-178B for non-level A systems. Not addressed on Volume 3.
Volume 2, section 2.3.3.1.4 How do we provide source to object code traceability when using inlining?	46	Section 3.6.3.2
Volume 2, section 2.3.3.1.4 How do we provide source to object code traceability when using implicit type conversion?	59	Sections 3.7.4.2, 3.7.5.2, and 3.7.6.2 <i>Note</i> : Above sections do not specifically address performance and timing issues (IL59) nor source to object code traceability (vol. 2, section 2.3.3.1.4).
Volume 2, section 2.3.3.2.1 How is functional coverage of low level requirements determined?	62	Sections 3.5.4.3, 3.9.4.2, and 3.11.4.1
Volume 2, section 2.3.3.2.1 How do we ensure adequate requirements coverage at all levels of integration when the number of test cases may be excessively large?	64	Section 3.3.10.3

<p>Volume 2, section 2.3.3.2.2</p> <p>To what extent can/should test cases developed for a class be reused to test its subclasses?</p>	<p>4, 18</p>	<p>Section 3.3.4.3, <i>Simple overriding rule</i>:,                  Section 3.3.4.3,  <i>Complete initialization rule</i>:,                  Section 3.3.4.3, <i>Initialization dispatch rule</i>:,                  Section 3.3.4.3, <i>Accidental override rule</i>:,                  Section 3.3.4.3, <i>Simple dispatch rule</i>:,                  Section 3.3.6.3,                  Section 3.3.7.3 (optional),                  Section 3.3.8.3, and                  Section 3.11.7.1</p> <p>OR</p> <p>Section 3.3.4.3, <i>Simple overriding rule</i>:,                  Section 3.3.4.3,  <i>Complete initialization rule</i>:,                  Section 3.3.4.3, <i>Initialization dispatch rule</i>:,                  Section 3.3.4.3, <i>Accidental override rule</i>:,                  Section 3.3.4.3, <i>Simple dispatch rule</i>:,                  Section 3.3.6.3,                  Section 3.3.7.3                  Section 3.3.9.3, and                  Section 3.11.7.1</p> <p>OR</p> <p>Section 3.3.4.3, <i>Simple overriding rule</i>:,                  Section 3.3.4.3,  <i>Complete initialization rule</i>:,                  Section 3.3.4.3, <i>Initialization dispatch rule</i>:,                  Section 3.3.4.3, <i>Accidental override rule</i>:,                  Section 3.3.4.3, <i>Simple dispatch rule</i>:,                  Section 3.3.6.3,                  Section 3.3.7.3 (optional)                  Section 3.3.10.3, and                  Section 3.11.7.1</p>
<p>Volume 2, section 2.3.3.3.1</p> <p>How do we define unique configuration items in OOT systems?</p>	<p>76</p>	<p>Sections 3.10.4.2 and 3.11.11.1</p>
<p>Volume 2, section 2.3.3.3.2</p>	<p>34, 74, 88</p>	<p>Sections 3.4.4.3, 3.10.4.2, 3.11.4.1, 3.11.10.1, and 3.11.11.1</p>

<p>How do OO tools and modeling languages affect the way configuration items are managed and changed?</p>		<p>OR</p> <p>Sections 3.4.5.3, 3.10.4.2, 3.11.4.1, 3.11.10.1, and 3.11.11.1</p> <p>OR</p> <p>Sections 3.4.6.3, 3.10.4.2, 3.11.4.1, 3.11.10.1, and 3.11.11.1</p> <p>OR</p> <p>Sections 3.4.7.3, 3.10.4.2, 3.11.4.1, 3.11.10.1, and 3.11.11.1</p>
<p>Volume 2, section 2.3.3.4.1</p> <p>How do we ensure traceability between functional requirements and object-oriented implementations?</p>	<p>61, 69</p>	<p>Section 3.3.4.3, <i>Simple overriding rule</i>:,</p> <p>Section 3.3.4.3,</p> <p><i>Complete initialization rule</i>:,</p> <p>Section 3.3.4.3, <i>Initialization dispatch rule</i>:,</p> <p>Section 3.3.4.3, <i>Accidental override rule</i>:,</p> <p>Section 3.3.4.3, <i>Simple dispatch rule</i>:,</p> <p>Section 3.8.4.2,</p> <p>Section 3.9.4.2,</p> <p>Section 3.11.4.1, and</p> <p>Section 3.11.9.1</p>
<p>Volume 2, section 2.3.3.4.2</p> <p>How do we ensure traceability when constructing inheritance hierarchies?</p>	<p>13, 35, 104</p>	<p>Section 3.3.4.2, assumption 3,</p> <p>Section 3.3.4.3, <i>Simple dispatch rule</i>:,</p> <p>Section 3.11.5.1,</p> <p>Section 3.11.6.1, and</p> <p>Section 3.11.7.1</p> <p>Section 3.3.4.2, assumption 3,</p> <p>Section 3.3.4.3, <i>Simple dispatch rule</i>:,</p> <p>Section 3.4.4.3,</p> <p>Section 3.4.5.3,</p> <p>Section 3.4.6.3</p> <p>Section 3.11.5.1,</p> <p>Section 3.11.6.1, and</p> <p>Section 3.11.7.1</p> <p>Section 3.3.4.2, assumption 3,</p> <p>Section 3.3.4.3, <i>Simple dispatch rule</i>:,</p> <p>Section 3.4.4.3,</p> <p>Section 3.4.6.3,</p> <p>Section 3.4.7.3,</p> <p>Section 3.11.5.1,</p> <p>Section 3.11.6.1, and</p> <p>Section 3.11.7.1</p>

Volume 2, section 2.3.3.4.3 How do we deal with behavioral requirements that map to multiple graphical views in OOT models?	72	Sections 3.10.3 and 3.11.6.1
Volume 2, section 2.3.3.4.4 How do we maintain traceability when using an iterative development process that leads to a large number of changes to a large number of artifacts?	105	Section 3.11.10.1
Volume 2, section 2.3.4.1 How can we ensure that visual modeling tools support compliance with DO-178B without introducing additional verification burden?	101, 102	Sections 3.10.4.2, 3.10.5.2 and 3.10.6.3
Volume 2, section 2.3.4.1 How can we ensure that structural coverage tools provide a reliable measurement of the structural coverage achieved?	103	Sections 3.10.7, 3.10.8.2, and 3.10.9.2
Volume 2, section 2.3.4.2 Considering the rapid rate of tool evolution and new tool types, how will tools be identified and maintained to meet long-term needs for development and maintenance?	86	The known type of tools for which best practices have been identified were addressed in the Handbook.
Volume 2, section 2.3.4.2 Considering the rapid rate of tool evolution and new tool types, how can we ensure that tools are properly controlled and retrievable?	84, 85	Issues are not specific to OOT. Not addressed in Volume 3.
Volume 2, section 2.3.4.3 How is tool qualification assured to address, for example, tool validation, independence, configuration management?	83	Issue is not specific to OOT. Not addressed in Volume 3.
Volume 2, section 2.3.4.3 How can we ensure that tool qualification criteria are appropriately identified and applied to OO tools?	82, 100	Sections 3.10.5.2 and 3.10.4.2
Not categorized in Volume 2	26	Section 3.3.3
Not categorized in Volume 2	40	Not addressed in Volume 3.
Not categorized in Volume 2	51	Section 3.5.5.1.2
Not categorized in Volume 2	54	Not addressed in Volume 3.
Not categorized in Volume 2	65	Not addressed in Volume 3.
Not categorized in Volume 2	67	Not addressed in Volume 3.
Not categorized in Volume 2	68	Due to lack of mainstream languages that support multiple dispatch, guidelines developed to address this issue were dropped from the

		handbook after the OOTiA Workshop #2. Not addressed in Volume 3.
--	--	---

Table 3.2-1 Mapping of Key Concerns and Guidelines

## 3.3 Single Inheritance and Dynamic Dispatch

### 3.3.1 Purpose

This section provides guidelines for the safe implementation and use of single inheritance and dynamic dispatch (also known as dynamic binding) in projects that use object-oriented (OO) technology (OOT).

### 3.3.2 Background

*Inheritance.* Inheritance supports the organization of object-oriented systems in terms of classes and class hierarchies. This is a fundamental concept that permits OO systems to directly represent and classify objects representing real-world entities from the problem domain without introducing redundancy.

*Classes.* Classes may define a variety of elements, including operations (which specify the services provided by the class), methods (which provide the code to implement operations), attributes (which represent stored data values), and associations (representing references to other objects).

*Visibility.* Class elements may be restricted in terms of their visibility. Unified Modeling Language (UML) [4], for instance, distinguishes between elements that are visible to all clients that have access to the class itself (public access), elements that are visible to clients within the same package (package access), and elements that are accessible only to subclasses (protected access). Elements may also be accessible to both classes in the same package and to subclasses (e.g., in Java), or to a named set of classes (e.g., in Eiffel).

*Operations.* Operations accessible to classes other than the defining class and its subclasses are sometimes referred to as client operations. All operations are identified by their signatures. The signature of an operation consists of its name and a list of the types of its parameters – the information needed to match a call to the operation being called. Consider the UML definition of an operation “m (p: Integer, q: Float)”. The signature of this operation consists of its name “m” and its parameter types “Integer” and “Float”. In some languages, the return parameter (if any) is considered a part of the signature, while in others (such as C++) it is not.

Most OO languages support constructors and destructors. A constructor is an operation called by the run time environment when a new object is allocated to ensure it is properly initialized. Conversely, a destructor is an operation called by the run time environment when an object is deallocated to ensure any resources held by the object are released. In most OO languages, a class may define more than one constructor, each with its own signature. The constructor called by the run-time environment is the one that matches the arguments supplied by the program at the point it requests the allocation of a new object. Destructors typically have no parameters and, as a result, at most one destructor is associated with a class.

*Constraints.* Class definitions may also include constraints in the form of preconditions, postconditions, and invariants. Preconditions represent constraints that must hold at the time a given method is called. Postconditions represent constraints that are guaranteed to hold once execution of the method completes, provided its preconditions were first met. Invariants represent constraints that are established by the class constructor and are considered to be a part of the precondition and postcondition of every client operation. Additional constraints may also apply to the relationships between classes. Constraints may be used to specify safety predicates as well as conditions for correctness, and acceptable use.

*Class hierarchies.* Class hierarchies consist of classes connected via generalization relationships. In such a relationship, the more general of the classes is termed the superclass, while the more specialized class is termed the subclass. The relationship itself is also referred to as subclassing or subtyping.

The class hierarchy may be extended to any depth, although very deep class hierarchies may cause difficulties. Subclasses inherit the elements of their superclasses. Subclasses may also extend these superclass definitions to include additional elements they define themselves, or redefine elements by overriding their inherited definitions.

Single inheritance allows each class to have at most one immediate superclass, while multiple inheritance permits a class to have more than one immediate superclass. Interface inheritance involves the inheritance of only interface elements (such as operation specifications and constraints), while implementation inheritance involves the inheritance of implementation elements (such as methods, attributes, and references to other objects).

*Polymorphism.* In most object-oriented languages, an object is permanently assigned a run-time class at the point at which it is allocated and initialized. Although the run-time class of the object never changes, the object can be treated not only as a member of its run-time class, but also as a member of any superclass of this class. This ability to treat an object as a member of any of its superclasses is referred to as polymorphism. Polymorphism supports the replacement of general implementations with more specialized ones. It, however, requires strict adherence to subtyping rules that guarantee that instances of subclasses behave like instances of their superclasses.

*Substitutability.* The basic subtyping rules are those given by Liskov and Wing [7]. Because they guarantee that we can substitute an instance of a subclass for an instance of a superclass, they are often collectively referred to as the Liskov Substitution Principle (LSP). Although the term *LSP* was not used by the authors [7], it has become a convenient way to refer to the principles required to guarantee substitutability and will be used in such context to discuss inheritance issues. The subtyping rules have also been popularized by Bertrand Meyer [17][18] in terms of a contracting metaphor between the clients of a class and its implementation.

In contracting terms, the client is responsible for establishing the precondition of an operation before calling it. Given this precondition, the method that implements the operation is then responsible for either delivering on the postcondition, or reporting an error to the client. The class invariant is established by the constructor when the object is first created, and must be maintained by all client operations. As a result, it is considered to be a part of the precondition and the postcondition of every client operation. The class invariant, however, need not hold at all points during the execution of a client operation, only at the beginning and at the end. This is sufficient to ensure that temporary violations of the invariant are not observable by clients if data is encapsulated and calls to client operations are properly synchronized.

Substitutability requires, quite simply, that subclasses not break the contract between client and implementation established by their superclasses. This applies both to the redefinition of one operation by another operation and the implementation of an operation by a method. As a result, the precondition of an operation in the subclass must be weaker (demand less) or the same as the precondition of the same operation in the superclass. Conversely, the postcondition must either be stronger (deliver more) or the same. Viewed in this way, substitutability requires that we not demand more of clients, i.e., the types of input parameters must be either be made more general or left unchanged, and that we deliver at least as much as promised, i.e., the types of output parameters must either be made more specific or left unchanged.

With regard to errors, the subclass version of an operation can only report the same types of errors as its superclass version. Otherwise clients would be expected to handle error cases that were not part of the original contract.

Substitutability also applies to changes to the signatures of operations introduced in subclasses. In this regard, the types of an operation's input parameters are logically a part of its precondition. Similarly the types of an operation's output parameters (and any return parameter type) are logically a part of its postcondition. In most OO languages, dynamic dispatch is used to associate 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, nor is it any more dynamic than the use of a case statement to explicitly select a method based on the run-time type of the target object.

*Dynamic dispatch.* Method selection based only on the type of the target object is referred to as single dispatch (since it involves only consideration of the run-time class of the object, i.e., a single parameter). In a few OO languages, method selection also includes the run-time classes of the remaining parameters. This is referred to as multiple dispatch [14]. The methods considered for selection are referred to as multi-methods. Languages that support multiple dispatch are more flexible in terms of the overriding of methods than single dispatch languages (and able to deal more elegantly with issues such as the binary methods problem [12][13] Analogous to single dispatch, multiple dispatch is logically equivalent to the use of a series of nested case statements for method selection.

*Issues and guidelines.* A number of issues arise when using single inheritance and dynamic dispatch that may make compliance with DO-178B difficult. Volume 2 documents the issues, related DO-178B sections and objectives, and applicable guidelines. Guidelines assume that all source code is available for software developed to meet DO-178B levels A, B, and C. In general, these "guidelines" do not represent new "guidance", but an interpretation of existing guidance (DO-178B) with respect to the use of particular OO features. The "rules" associated with these guidelines are also rules only in the sense that they must be followed in order to adopt the given approach. Often there are also alternative approaches that can be followed in order to address the same issues and still comply with DO-178B.

### 3.3.3 *Overall Approach*

This section is intended to provide an approach for addressing DO-178B objectives when using OO features related to single inheritance and dynamic dispatch. The issues list appearing in volume 2 specifies potential obstacles to DO-178B compliance. This list is not intended to address only OO unique issues, but also related issues that are of particular importance to the use of single inheritance and dynamic dispatch.

Where it appears possible to use a feature or combination of features in a way that complies with DO-178B, we have provided guidelines that describe an associated approach. The existence of these guidelines, however, does not constitute a recommendation that the feature(s) be used, only acceptance that the given guidelines resolve the issues in a manner consistent with DO-178B.

The overall collection of guidelines is also intended to be open-ended. As a result, new approaches and new guidelines may be added that address the same issues as existing approaches, under different circumstances.

The guidelines on *Inheritance with Overriding* address the core issues related to inheritance, overriding and dynamic dispatch. Because all these features are closely related, they are addressed together, rather than separately. In addition, the rules apply both to redefinition of one operation by another and the implementation of an operation by a method where operation and method are defined using UML<sup>1</sup> [4]. The emphasis is on simplicity through the strict enforcement of a small set of basic principles, an approach similar to that taken by Meyer with respect to Eiffel [18].

Note that, there may be some situations in which it is entirely reasonable to use inheritance to support code sharing without intending to achieve type substitutability. The aim is usually to achieve aggregation with export of methods (and possibly attributes) from the aggregated class(es). The problem is that most OO programming languages do not distinguish this form of inheritance from subtyping inheritance (or provide a mechanism for aggregation with export of features). Inheritance should be used for this purpose only when it is documented and the use of polymorphism and dynamic dispatch in respect of these classes is avoided.

The guidelines on *Subtyping* and *Formal Subtyping* complement those on *Inheritance with Overriding* by specifying how to test for superclass/subclass compatibility. Several different approaches are possible. The simplest involves unit level testing and the inheritance of unit level test cases as in the guidelines for *Unit Level Testing of Substitutability*. For organizations that want to do all testing at a system level, two approaches are provided. The guidelines for *System Level Testing of Substitutability Using Assertions* involve instrumentation of the code with assertion checks. The guidelines for *System Level Testing of Substitutability Using Specialized Test Cases* involve the development of specialized versions of system level test cases for this purpose.

Note that, although the subtyping guidelines address compliance from a behavioral perspective, inheritance involves both classification and implementation, and the valid use of inheritance need not be limited to subtyping (although it often is, e.g., by UML).

The remaining guidelines address special cases and individual issues. The guidelines on *Method Extension* deal with the definition of a subclass method as an extension of an inherited version of the same method. It applies most often to constructors, but can be used to extend the implementation of any method.

The guidelines on *Class Coupling* address concerns related to flow analysis between superclass and subclass definitions. They recommend the definition of an abstract interface between a class and its subclasses analogous to the client interface for the class.

The guidelines on Deep Hierarchy provide a rule to help identify class hierarchies that are “too deep”. Unlike the rules associated with the other patterns, this is intended only as a rule of thumb. Engineering judgment is required to balance the tradeoffs associated with any proposed changes.

### 3.3.4 *Inheritance with Overriding*

This section provides a set of guidelines on the use of inheritance, overriding, and dynamic dispatch which are equivalent to the use of hand-coded dispatch using case statements or compound if statements. When these

---

<sup>1</sup> Use of UML is not a requirement. Users are free to choose their own approach to modeling and OO development.

guidelines are extended to include behavioral subtyping, they provide explicit criteria to verify substitutability. These guidelines assume the use of languages that support single dispatch on the target object.

### 3.3.4.1 Motivation

The unrestricted use of dynamic dispatch raises a number of issues with respect to certification, especially with regard to weakly typed languages, and systems that permit the run-time loading of new classes (that are not a part of a previously verified system configuration). With suitable language restrictions (i.e., a precisely defined language subset that permits use of static analysis techniques), dynamic dispatch is semantically equivalent to the use of hand-coded dispatch methods containing nested case statements or compound if statements. The automation of dynamic dispatch by the compiler is then equivalent to the auto-generation of these dispatch routines and inlined calls to them. This treatment as an inlined call, combined with compliance with structural coverage criteria identified in Sections 3.10 and 3.12, provides a means to ensure compliance with DO-178B.

### 3.3.4.2 Applicability

These guidelines assume:

1. a strongly typed language,
2. single dispatch,
3. the set of classes associated with the system is statically known,
4. no dynamic classification (i.e. the run-time class of an object never changes)
5. use of polymorphism.

### 3.3.4.3 Guidelines

The following rules define a form of inheritance, overriding, and dynamic dispatch which is equivalent to hand-coded dispatch using case statements or compound if statements:

#### 1. Simple overriding rule:

*An operation may redefine an inherited operation, and a method may implement an operation so long as changes to its signature guarantee substitutability.*

Specifically, a redefined operation may be made more visible to clients. And a redefining operation or implementing method may be made more restrictive regarding the types of errors it can report to clients (e.g., as exceptions or by setting error return codes).

With regard to parameter types, an operation may override an inherited operation or a method may implement an operation by supertyping its input parameters, or subtyping its return type or the types of output parameters. The types of parameters that represent both inputs and outputs must remain unchanged (invariant). No other form of overriding should be allowed for languages supporting only single dispatch.

#### 2. Accidental override rule:

To ensure that overriding is always intentional rather than accidental, design and code inspections should consider whether locally defined features are intended to override inherited features with a matching signature<sup>2,3</sup>.

#### 3. Simple dispatch rule:

When an operation is invoked on an object, a method associated with the operation in its run time class should be executed. This rule applies to all calls except explicit calls to superclass methods, which should be addressed as described by the *Method Extension* guidelines.

<sup>2</sup> When the language itself does not allow the user to make the intent to override an inherited operation/method explicit.

<sup>3</sup> As defined in the Glossary, a feature is an attribute, operation, or method. This includes attributes that reference other objects (i.e., association ends).

4. *Complete initialization rule:*

Every attribute must be initialized to a value consistent with the class invariant by the class constructor.

5. *Initialization dispatch rule:*

No overridden method should be called during the initialization (construction) of an object.

6. *Dispatch time rule:*

All dispatch times should be bounded and deterministic

7. *Object code traceability rule:*

Everywhere concerns about source code to object code traceability and timing analysis dictate, the compiler vendor may be asked to provide evidence of deterministic, bounded mapping of the dispatched call. If the evidence is not available from the compiler vendor, it may be necessary to examine the structure of the compiler-generated code and data structures (e.g., method tables) at the point of call.

A dispatching method call is considered semantically equivalent to the invocation of a dispatching routine containing a case statement of the form:

**case of** <target-object-run-time-class>

**case** <class>:

        <statically-resolved-call-to-method-implemented-by-class>;

    ...

**end**

Each case of this case statement handles dispatch to an implementation of the method by the target object's declared type or one of its subclasses, i.e., the class corresponding to the object's run time type.

With regard to the *Simple overriding rule*, the inability to subtype the types of input parameters does not preclude the use of overloading for this purpose. The developer, however, must clearly understand that the selection of an overloaded method is based on the declared types (rather than the run time types) of the arguments at the point of call.

In accordance with the *Simple dispatch rule*, method calls are expected to be dispatching.<sup>4</sup> Dispatch must account for the run time type of the target. Static resolution is regarded as an optimization in those cases where only one resolution is possible.

*Note:* The *initialization rule* is not intended to be an obstacle to the creation of "reset" operations that can be called by clients to reinitialize on object after it has been constructed, or to the sharing of initialization code with class constructors. It suggests only that the client reset operation (which has the class invariant as a part of its precondition) and the constructors (which do not) call a non-overridden, internal operation that performs the initialization steps common to them all.

The *Simple overriding rule* ensures substitutability is not violated at the language level, in terms of method declarations. The *Subtyping* extends this to include testing for substitutability at the behavioral level. Compliance with substitutability is necessary if instances of subclasses are to be treated as instances of their superclasses. This is not only required by the UML definitions of generalization and inheritance, but a fundamental assumption underlying the use of polymorphism and dynamic dispatch. Verification of substitutability for the most critical software can be shown in one of two ways: 1) by testing each case at each call site (when dynamic dispatch is only rarely used) or 2) by conforming to the Subtyping guidelines (more practical for software where dynamic dispatch is more widely used).

---

<sup>4</sup> In OOT languages such as Ada95, C++, and Java, binding occurs when the application is built, not at execution time. Only in languages such as Smalltalk and Common Lisp is dynamic binding truly dynamic (execution time). In the other languages mentioned dynamic dispatch is no more dynamic than a case statement, i.e., all alternatives are statically determined. As a result, the developer should avoid languages such as Smalltalk and Lisp for avionics applications.

The *accidental override rule* is intended to guard against errors that could occur in languages that assume subclass operations and methods override superclass operations and methods with a matching signature. This rule is unnecessary if the language forces the developer to explicitly state that overriding is intended (as in C#).

The *simple dispatch rule* is intended to support a model of object-oriented behavior in which (1) each class can be completely understood by looking at it in flattened form, and (2) the behavior of any object can be completely understood by looking at the flattened definition of its run-time class. The *simple dispatch rule* guarantees this even when the declared type of the object is a superclass of its run-time class (i.e., when polymorphism is used).

The *initialization rule* is intended to avoid errors that may arise during the construction of an object when a subclass version of a method is called before associated subclass attributes have been initialized and the subclass invariant (if any) has been established. In particular, the class invariant is implicitly a part of the precondition and postcondition of every client operation, and the class invariant is not guaranteed to be true until the constructor completes. As a result, we should not call overridden client operations during object construction. For similar reasons, special care should also be taken with respect to calls to client operations in destructors, at other points where the class invariant may no longer hold.

#### 3.3.4.4 *Related guidelines*

The guidelines on *Overloading and Method Resolution* are closely related to those given in this section because both overriding and overloading define families of operations whose specifications should be related by principles guaranteeing substitutability.

The guidelines in the section *Top Heavy Hierarchy* provide metrics to limit the complexity of inheritance hierarchies.

The *Subtyping* guidelines extend those appearing in this section to include the testing of substitutability at the behavioral level.

### 3.3.5 *Method Extension*

*Method Extension* supports the implementation of an operation as an extension of an inherited method without introducing redundancy, and with the assurance that the resulting postcondition is stronger than or the same as that in the superclass. It helps to address issues related to initialization by allowing a subclass constructor to be defined as an extension of its parent class constructor, without introducing problems related to the use of dynamic dispatch during initialization.

#### 3.3.5.1 *Motivation*

Often we want to provide a subclass version of an operation that extends the functionality provided by the operation in its superclass. This extension in functionality must be consistent with substitutability. As a result, the precondition for the subclass operation must be weaker than or the same as its precondition in the superclass, and the postcondition for the subclass operation must be stronger than or the same as its postcondition in the superclass.

By implementing an extended operation in terms of an explicit call to the superclass method, preceded or followed by additional code, we are able to:

- avoid repeating the code appearing in the superclass method,
- provide code before the call to handle the additional cases implied by a weaker precondition<sup>5</sup>,
- provide code after the call that adds to its effect, as implied by a stronger postcondition.

Such explicit calls to superclass methods are by their nature statically bound, and do not involve dynamic dispatch. The code that follows the call to the superclass method must not undo its effects in order for the overall postcondition to be an extension of that for the superclass.

---

<sup>5</sup> Weakening the precondition makes it valid to call an operation with additional inputs or input combinations. Consider the operation  $f(p: \text{Integer})$  **pre**  $p > 0$ . If we weaken the precondition to give us  $f(p: \text{Integer})$  **pre**  $p \geq 0$ ., then the implementation must handle the additional case in which  $p$  is zero.

### 3.3.5.2 *Applicability*

These guidelines are commonly applied to constructors, which begin by calling the constructor for the superclass, then initialize all the attributes defined by the class itself. It may also be used to select between competing inherited versions of a method (multiple implementation inheritance).

### 3.3.5.3 *Guidelines*

The following rule for method extension represents the sole exception to the *Simple dispatch rule* given in 3.3.4.3:

*Method extension rule:* When extending the functionality of an inherited method, the subclass method should explicitly call the superclass version of the same method.

Note that the terms *method* and *operation* are used in accordance with the UML definitions (see Glossary) since some languages blur the distinction. Implementation of method extension in different target languages is described in section B.1.2.

### 3.3.5.4 *Related guidelines*

The guidelines on *Method Extension* represent the sole exception to the *Simple dispatch rule* given in section 3.3.4.3.

## 3.3.6 *Subtyping*

The *Subtyping* guidelines extend the *Inheritance with Overriding* guidelines (which addresses substitutability at a language level in terms of operation signatures) in order to verify compliance with substitutability at a behavioral level.

### 3.3.6.1 *Motivation*

DO-178B verification activities may involve testing (at either a unit or system level), the use of formal or informal proofs, or other techniques. This section does not prescribe a particular approach. That is left to those sections that extend the basic guidelines given in this section.

Intuition can be misleading when it comes to subtyping relationships. We might, for instance, think (intuitively and mathematically) that all squares are rectangles, so Square should be a subclass of Rectangle. Whether Square should be a subclass of Rectangle, however, should not be based on our intuition, or any mathematical definition, but on the interfaces we define for these classes. If the interface for Square specializes the interface for Rectangle in accordance with substitutability, then it is appropriate for it to be a subclass of Rectangle. Otherwise, it is not [11].

By assuming that instances of subclasses must always be substitutable for instances of their superclasses, these guidelines restrict the preconditions and postconditions of redefined operations in addition to their signatures. Formally, the precondition for a redefined operation must be weaker than or the same as the precondition of the operation it redefines. Conversely, the postcondition for a redefined operation must be stronger than or the same as the postcondition of the operation it redefines.

In terms of the client interface, this means that a subclass is compatible with a superclass if we:

- (1) expect no more of clients than we do in the superclass (the preconditions of overridden operations are weaker or the same), and
- (2) deliver at least as much (the postconditions of overridden operations are stronger or the same).

The same rules apply to the relationship between methods and the operations they implement: the precondition for an implementing method must be weaker than or the same as the precondition of the operation its implements, and the postcondition for an implementing method must be stronger than or the same as the postcondition of the operation it implements.

Note that in real time systems, the deadline by which a method must complete may be treated as part of the post-conditions. If it is important to know up-front in system development that timing constraints are met, then every implementation must be required to meet its respective timing bound. Otherwise, when complete closure exists, the system as a whole must be tested to insure it meets its required bounds. In a broader sense, post-conditions can address other quality of service (QOS) issues in addition to timing constraints.

### 3.3.6.2 *Applicability*

These guidelines apply when instances of different subclasses may be assigned (polymorphically) to a given variable or parameter.

### 3.3.6.3 *Guidelines*

The guidelines in this section extend those on *Inheritance with Overriding* in section 3.3.4 to include the following additional rules.

1. *Minimum compatibility rule:*

At a minimum, superclass/subclass compatibility should be verified with respect to all classes involved in the polymorphic assignment of different subclass instances to the same variable or parameter during the execution of the system.

2. *Substitutability compliance rule:*

Any approach used to verify superclass/subclass compatibility should be consistent with the principles of behavioral subtyping defined by Liskov and Wing [7].

As specified by UML, the semantics of subclassing implies superclass/subclass compatibility in accordance with substitutability. In practice we must, at a minimum, ensure that we verify this in all cases where instances of different subclasses may be associated with the same variable or parameter during the execution of the system under test.

Additional guidelines are defined by a number of sections that extend these basic guidelines. These guidelines vary with the approach used to verify superclass/subclass compatibility. The standard for superclass/subclass compatibility, however, is the same: compliance with the principles of behavioral subtyping defined by [7].

### 3.3.6.4 *Related guidelines*

Related guidelines include those for *Formal Subtyping*.

## 3.3.7 *Formal Subtyping*

The *Formal Subtyping* guidelines apply the principles of Design by Contract [17] with formally defined pre/postconditions and invariants. By requiring formal specification of the classifiers to be checked in a precondition/postcondition/invariant style, these assertions can then be used to either generate the needed test cases, or as the basis for analysis and formal proofs. Providing a complete and precise specification of the interface also helps prevent errors by making the contract between the clients and implementers of a class explicit, makes it easier to enforce rules for substitutability, and supports the traceability of high level requirements to low level requirements.

### 3.3.7.1 *Motivation*

The signature of an operation, which includes its name, parameter types, result types (if any), and errors (if any), provides clues to the operation's behavior but, by itself is insufficient. Comments that describe the purpose of the operation and the relationships between inputs, outputs, and errors are also helpful. But most comments are informal, cannot be processed by tools, and lack the precision to serve as a basis for analysis, proofs, or the development of test cases.

As a result, it is generally recommended that class interfaces be specified in a precondition/postcondition/invariant style (an approach referred to by Meyer as ‘Design by Contract’). Expressing low level requirements in this way helps prevent errors by making the semantics of the interface clear to developers before client code is written. Such specifications can also be processed by tools and used as a basis for analysis, formal proofs, and the generation of test cases. This, in turn, supports reverification and regression testing in response to changes to the class introduced in order to comply with other guidelines.

It is also useful to specify global data access and information flow relations as part of an operation’s contract. While these are not as strong as the recommended precondition/postcondition/invariant style, they can be efficiently checked and also used as the basis for analysis, formal proofs, and the generation of test cases.

### 3.3.7.2 *Applicability*

These guidelines apply when instances of different subclasses may be assigned (polymorphically) to a given variable or parameter.

### 3.3.7.3 *Guidelines*

The guidelines in this section extend those on *Subtyping* (section 3.3.6) to include the following rules:

*Explicit pre/postcondition/invariant rule:* To ensure that all classes define their interfaces as contracts, all pre/postconditions and invariants for operations and methods must be explicitly stated and all errors returned by them must be specified. Unless the program is to be subjected to automated format analysis, this includes pre/postconditions, invariants, and error lists that are considered to be trivial (e.g., conditions whose value is *true*, and error lists that are empty).

*Frame condition rule:* Unless the language provides a separate mechanism for indicating which variables may and may not change, ideally each postcondition should also include a ‘frame condition’ which indicates which variables are guaranteed not to change as a result of executing the operation/method.

Software developers tend to rely on testing to verify substitutability. While associated test cases may be developed at either the system or the unit level, the development of test cases alone, however, has its limitations. Test cases are no substitute for a complete, precise specification of behavior, which is needed by the clients of a class, and by developers seeking to subclass an existing class.

A complete and precise specification of behavior is also needed in order to develop the test cases for a class and to provide traceability from high level to low level requirements. As a result, it is generally recommended that class interfaces be specified in a precondition/ postcondition/invariant style (an approach referred to by Meyer as ‘Design by Contract’). This style provides the basis to make compliance with substitutability easier and helps avoid errors by making contracts between clients and classes explicit. In addition, explicit statement of pre/postcondition and invariants can be used to generate unit level test cases, or can be used as the basis for formal analysis and proofs.

Realistic implementation of these guidelines requires the use of an unambiguous programming language and application of static analysis techniques.

### 3.3.7.4 *Related guidelines*

Related guidelines include those for *Unit Level Testing of Substitutability*, *System Level Testing of Substitutability Using Assertions*, and *System Level Testing of Substitutability Using Specialized Test Cases*.

## 3.3.8 *Unit Level Testing of Substitutability*

The guidelines in this section check for superclass/subclass compatibility by requiring that all unit level test cases associated with a class are inherited by its subclasses. These guidelines help address the same issues as those defined for *Subtyping*.

### 3.3.8.1 Motivation

The verification of superclass/subclass compatibility is straightforward if we develop a set of unit level test cases for all classes identified by the *minimum compatibility rule* (*Subtyping*).

Subtype compatibility then means that all superclass test cases should run successfully against all subclass instances (superclass test cases are inherited by subclasses) provided that all such tests satisfy the method precondition(s) involved, taking account of any dynamic dispatch involved in evaluating the precondition. Subclasses also often extend this set of superclass test cases to include their own more specialized tests (the subclass test set is a superset of the superclass test set).

### 3.3.8.2 Applicability

These guidelines apply when concerns exist about compatibility between specific classes, and test cases are written to directly test these classes. Typically, this applies to low-level requirements. These guidelines should be applied to situations where instances of different subclasses may be assigned at run-time to a variable or parameter whose declared type is an associated superclass (polymorphic assignment).

In contrast to other *Subtyping* guidelines, the guidelines in this section are most effective when the development organization relies on a combination of system level and class level testing, rather than on system level testing alone. Separate sections address System Level Testing for Substitutability (see the guidelines for *System Level Testing of Substitutability Using Assertions* and *System Level Testing of Substitutability Using Specialized Test Cases*).

### 3.3.8.3 Guidelines

The guidelines in this section extend those on *Subtyping* (section 3.3.6) to include the following rules which apply to all classes identified by the *minimum compatibility rule*:

*Inherited test case rule:* Every test case appearing in the set of test cases associated with a class should appear in the set of test cases associated with each of its subclasses.

*Separate context rule:* If dynamic dispatch is involved in the execution of a method, the method should be separately tested in the context of every concrete class in which it appears, irrespective of whether it is defined by the class or inherited by it, provided that all such tests take account of the method precondition(s) involved, taking account of any dynamic binding involved in evaluating the precondition. An exception is made for methods that are guaranteed not to directly or indirectly invoke a method that is dynamically bound with respect to the current object, for example, simple *get* and *set* methods that only assign a value to, or return the value of an attribute or association. Such methods need only be tested once, in the context of the defining class.

These rules are intended to imply that all inherited test cases (other than those for simple gets and sets) should be run against instances of all concrete subclasses. As a result, changes to the code inherited by a class that affect its flattened form should result in its retest precisely as if the class itself had been edited.

The *inherited test case rule* is intended to apply to all test cases, including those introduced solely to meet structural coverage criteria. It could be argued that such tests should only be inherited when the tested code is also inherited. It, however, seems simpler and safer to recommend that they be inherited in all cases, since they should pass when run against the subclass.

When applying the *inherited test case rule*, if the subclass invariant is stronger than that of its superclass, then a check of this invariant (rather than the weaker superclass invariant) should be a part of the pass/fail check of each inherited test case. In this way, the “missing override” issue is resolved.

The *separate context rule* is intended to ensure that superclass methods are separately tested in the context of each subclass. This recommendation addresses the fact that even when a given method is inherited without change, the methods called by it may be overridden in the subclass, leading to a different behavior. An exception is made for methods that are guaranteed not to directly or indirectly invoke a method that is dynamically bound with respect to the current object, for example simple *get* and *set* methods that reference only data, and do not call other methods. A more complete impact analysis could be used to determine whether other inherited methods need to be retested in

the context of each subclass. The guidelines in this section, however, assume it is simpler and easier to rerun such tests than to perform such an analysis.

Note that testing in accordance with these guidelines will ensure that all dispatch table entries are exercised at some call site, equivalent to providing MC/DC of the case statement assumed to be associated with the dispatch routine.

Although not required to use these guidelines, it is recommended that class interfaces be specified in a pre/post-condition style prior to writing test cases (an approach referred to by Meyer [17] as ‘Design by Contract’). Expressing low-level requirements in this way helps prevent errors by making the semantics of the interface clear to developers before client code is written. Pre- and post- conditions may be specified in a variety of ways, e.g. as informal comments, as formal annotations, in table form, in terms of a state diagram, or in terms of executable run time checks used by a test driver. Such pre- and post- conditions may also be useful as input to test case generation tools.

Although these guidelines typically apply to the testing of low-level classes and requirements, they can be used at a high level if the classes and subclasses to be tested for compatibility represent a software system or a subsystem. For example, given a class System with subclasses SystemA and SystemB, SystemA and SystemB should inherit the test cases defined for System (which are based on the high level requirements common to both of them).

#### 3.3.8.4 *Related guidelines*

Related guidelines include those for *System Level Testing of Substitutability Using Assertions*, *System Level Testing of Substitutability Using Specialized Test Cases*, and *Percolation* [11, pp. 882-896].

### 3.3.9 ***System Level Testing of Substitutability Using Assertions***

The guidelines for *System Level Testing of Substitutability Using Assertions* check for superclass/subclass compatibility by instrumenting a version of the software with assertion checks related to substitutability. Existing system level test cases are then run (without change) against this version (to test for substitutability violations), then run a second time against the uninstrumented target version of the software. These guidelines help address the same issues as those for *Subtyping*.

#### 3.3.9.1 *Motivation*

Some projects prefer to focus exclusively on system level, requirements based testing, without developing any unit level/class level test cases. Use of the *Unit Level Testing of Substitutability* is clearly in conflict with this approach. A number of programming languages and tools, however, support the selective use of pre- and post- condition and invariant declarations as run time checks. One particularly simple way to test for substitutability at a system level is to take advantage of the use of these checks to verify superclass/subclass compatibility for classes identified by the *minimum compatibility rule (Subtyping)*. This does not involve any substantial changes to existing system level, requirements based tests. It does, however, require that these tests be run against both an instrumented and uninstrumented version of the software.

#### 3.3.9.2 *Applicability*

These guidelines apply when concerns exist about the substitutability of various subclasses for one another at run-time and requirements-based test cases are written to test these configurations. Typically, the test cases are based on high-level requirements.

Although these guidelines typically apply to the testing of high-level requirements at a system level, they can also be applied at a subsystem level, in terms of low-level or derived requirements. These guidelines differ from those for the *Unit Level Testing of Substitutability* in that test cases are written against some entity (e.g., system or subsystem) that contains instances of the classes we wish to test for compatibility, and not directly against these classes.

Use of these guidelines is straightforward if a project already instruments the code to measure structural coverage, or is already defining interfaces as contracts [17][18, Design by Contract]. These, though, are not prerequisites.

The guidelines, however, do assume that system level test cases have been developed (or will be developed) to test for all system configurations in which instances of various subclasses may be substituted for one another at run time. (Development of these test cases should be driven by high-level requirements related to substitutability). Substitutability related assertions require language or tool support.

As with the instrumentation of code for any reason (e.g. measurement of structural coverage), care should be taken to account for the overhead associated with the run time checks involved, e.g., timing may be affected. It is also necessary to consider what should be done if any of the conditions (pre-, post-, and invariant) are violated, i.e., handling of exceptions needs to be accounted for.

### 3.3.9.3 Guidelines

The guidelines in this section extend those for *Subtyping* (section 3.3.6). The following rules describe the type of assertion checks required to test for superclass/subclass compatibility in accordance with substitutability. Such checks should be performed on all classes identified by the *minimum compatibility rule*.

*Precondition assertion rule:* An assertion to check the operation's precondition should appear before the body of all methods that implement a public operation. In accordance with substitutability, this precondition may only be weakened or the same in overridden versions of the operation.

*Postcondition assertion rule:* An assertion to check the operation's postcondition should appear after the body of all methods that implement a public operation. In accordance with substitutability, this postcondition may only be strengthened or the same in overridden versions of the operation.

*Invariant assertion rule:* An assertion to check the operation's invariant should be a part of the precondition check and the postcondition check of all public operations. In accordance with substitutability, the invariant may only be strengthened or the same in all subclasses of a class.

*Instrumented/uninstrumented testing rule:* A test case run against an instrumented version of the code should be considered to pass only if all assertion checks associated with substitutability hold during its execution. A test case run against an uninstrumented version of the code should be considered to pass only if it produces the same result that it did when run against an instrumented version of the same code.

The first three rules echo the basic principles of substitutability. Some languages (such as Eiffel [17][18]) enforce these rules directly and provide facilities for enabling and disabling associated run time checks, as required for instrumented/ uninstrumented testing. In other languages (such as C++ and Java), it is possible to use language level assertions to achieve the same effect, although the substitutability relations between preconditions, postconditions and invariants must be enforced by code reviews. A subset of the language that is amenable to use of substitutability and that will help static analysis should strongly be considered.

A simple way in which to ensure that these relations hold is to require that:

- new preconditions be of the form 'overridden\_pre **or** some\_condition'
- new postconditions be of the form 'overridden\_post **and** some\_condition'
- new subclass invariants be of the form 'superclass\_invariant **and** some\_condition'.

Otherwise the precondition, postcondition or invariant must be the same.

Tool support is also available from a number of sources. Usually this includes enforcement of the first three rules, and a facility for enabling and disabling the use of assertions as run time checks (analogous to Eiffel). Assertions may be written in a number of notations, ranging from simple boolean expressions in the target language to first order logic [20], with quantification.

Assertions may also be introduced at an analysis or design level and mapped down to run time checks in the target language. Many tools that favor this approach, however, rely on proofs for verification, rather than the introduction of run time checks.

Binder discusses the instrumentation of the code with substitutability run time assertion checks in detail (with examples and sample code) in his *Percolation* pattern [11, pp. 882-896].

Not all assertion checks should necessarily be removed in the uninstrumented version of the code. In accordance with the advice of Liskov and Guttag, “it is usually worthwhile to retain at least the inexpensive checks” [8, p. 251].

### 3.3.9.4 *Related guidelines*

Related guidelines include those defined for *Percolation* [11, pp. 882-896], *Unit Level Testing of Substitutability*, and *System Level Testing of Substitutability Using Specialized Test Cases*.

## 3.3.10 *System Level Testing of Substitutability Using Specialized Test Cases*

The guidelines in this section check for superclass/subclass compatibility by developing system level test cases in a manner that first ignores compliance with substitutability and then introduces specialized versions of existing system level test cases to explicitly test for substitutability compliance. These guidelines help address the same issues as the those for *Subtyping*.

### 3.3.10.1 *Motivation*

Some projects prefer to focus exclusively on system level, requirements-based testing, without developing any unit level /class level test cases. Use of the *Unit Level Testing of Substitutability* is clearly in conflict with this approach. There are, however, ways to test for substitutability at a system level. One approach involves the development of specialized system level tests for this purpose. This has the advantage of avoiding *instrumentation (guidelines for System Level Testing of Substitutability Using Assertions)* although it typically requires that test cases be developed with this approach in mind.

### 3.3.10.2 *Applicability*

These guidelines apply when concerns exist about substitutability of various subclasses for one another at run-time and requirements-based test cases are written to test these configurations. Typically, the test cases are based on high-level requirements.

Although these guidelines typically apply to the testing of high-level requirements at a system level, they can also be applied at a subsystem level, in terms of low-level and derived requirements. They differs from the guidelines for *Unit Level Testing of Substitutability* in that test cases are written against some entity (e.g., system or subsystem) that contains instances of the classes we wish to test for compatibility, and not directly against these classes.

Use of these guidelines is straightforward if test cases have not yet been developed, or if the current set of test cases ignores substitutability.

### 3.3.10.3 *Guidelines*

The guidelines in this section extend those for *Subtyping* (section 3.3.6). The following rules relate to the process used to develop system level test cases to test for substitutability [3, section 5]. They assume that, with suitable language restrictions, dynamic dispatch is semantically equivalent to the use of hand-coded dispatch methods containing case statements or compound if statements. The rules begin with conformance to coverage criteria based on inlined calls to a case statement with MC/DC testing for Level A systems. They culminate in conformance to coverage criteria that guarantees substitutability between subtypes and supertypes.

*Generalized test case rule:* First construct a set of system level test cases to meet the required DO-178B coverage criteria while considering only the declared classes of objects and object references. The run time classes of objects and dynamic dispatch should be ignored other than to mark test cases that include dispatching calls as polymorphic.

*Specialized test case rule:* Next create a set of specialized test cases for each polymorphic test case that are explicitly designed to test for substitutability. The set of test cases generated from a given polymorphic test case should be designed to drive dynamic dispatch down different paths with regard to the selection of subclass methods. The initial state and resulting state associated with each specialized test case should be compatible (in terms of substitutability) with the more general test case from which it was derived.

Although the steps to comply with the above-listed rules are the same, it is also possible by complying with the following rules to develop a set of test cases that meet any of a number of different substitutability-related coverage criteria.

*Min Substitutability coverage rule:* By this test case coverage criteria, the full set of specialized test cases must exercise dynamic dispatch to subclass methods to the extent required to meet the structural coverage criteria of DO-178B.

*Max Substitutability coverage rule:* By this test case coverage criteria, the set of specialized test cases derived from each polymorphic test case must exercise all reachable subclass methods at each point of call involving dynamic dispatch.

*Mid Substitutability coverage rule:* By this test case coverage criteria, the full set of specialized test cases must meet the criteria set by the Min Substitutability coverage rule. Additional specialized test cases, however, are introduced to test specifically for the types of problems raised by issues 20, 21, 22, 26, and 40 identified in Volume 2, Appendix B of this Handbook.

Compliance with DO-178B objectives requires conformance to the *Min Substitutability coverage rule*. This rule is designed to rely on DO-178B to set the criteria for test case coverage. Rather than require additional test cases to test for substitutability, it takes advantage of the process for test case generation to add substitutability-related compatibility checks to the test cases created (“The initial state and resulting state associated with each specialized test case should be compatible (in terms of substitutability) with the more general test case from which it was derived.” [7]). This is not as rigorous as the process of the *Unit Level Testing of Substitutability*, but it does make the most of the test cases already required by DO-178B.

The *Max Substitutability coverage rule* matches the degree of rigor offered by the *Unit Level Testing of Substitutability*, and exceeds it. It is most appropriate when the DO-178B software level is high (e.g. level A) and the number of calls involving dynamic dispatch is small. If run-time substitution of different subclass instances is commonplace (*Subtyping, minimum compatibility rule*), exhaustive testing of all subclass methods in the context of every system level test case is impractical. Typically, however, the most safety critical applications are also the most static, making this level of substitutability test coverage acceptable in many cases.

The *Mid Substitutability coverage rule* attempts to strike a balance between the previous coverage criteria by requiring compliance with the *Min Substitutability coverage rule*, but adding rigor through the introduction of additional test cases targeted specifically to the types of problems raised 20, 21, 22, 26, and 40 identified in Volume 2, Appendix B of this Handbook. Additionally, explicit robustness test cases may need to be developed but this is not unique to substitutability-related testing [19].

Tools are often used to verify structural coverage criteria. Structural coverage analysis tools for OO languages should measure coverage for each polymorphic reference and each resolution for each set of identical polymorphic references. When a tool does not have the capability to measure coverage in this way, then a process will need to be performed to augment the tools analysis capabilities to satisfy structural coverage objectives. The sets of rules in this section may be used to augment a tool’s capability to meet DO-178B structural coverage objectives. A clarification of the coverage requirements for class structure versus the coverage of method internals within the class structure is found in sections 3.10.8 and 3.10.9.

#### 3.3.10.4 Related guidelines

Related guidelines include those for *Unit Level Testing of Substitutability*, *System Level Testing of Substitutability Using Assertions*, and *Percolation* [11, pp. 882-896].

#### 3.3.11 Class Coupling

The guidelines in this section limit control flow and data flow between clients and classes and between classes and subclasses to facilitate future changes and to simplify analysis.

### 3.3.11.1 Motivation

One of the fundamental principles of object-oriented development is data abstraction. The goal is to hide the details of the data representation behind an abstract class interface. This permits the data representation to change without affecting other classes. It also simplifies the enforcement of class invariants, and permits control over concurrent access to shared data. Extending this principle, we can use abstract class interfaces to control access to hardware resources as well as data.

For example, consider the implementation of a class for a set. The selection of an optimum data representation (list, tree, hash table, etc.) will vary depending on the mix of operations required by the application. Typical strategies involve a choice between a sorted and a hashed representation, and a choice between fast insert/delete and fast lookup. The use of data abstraction makes it easy to change this representation with no significant change to client code.

The same situation arises with respect to the abstraction of other resources. For example, a number of different hardware devices may perform a given function, or a number of different caching policies may be associated with access to a given file system, or a number of different ways may exist to represent the elements of a given display, with a number of different strategies for drawing/redrawing them. In all these cases, we want to publish a single interface but support a number of different underlying implementations and data representations.

In terms of DO-178B, data abstraction supports partitioning [1, p. 9, section 2.3.1] by permitting the developer to restrict access to the resources controlled by the class. By limiting the degree of control and data coupling between software components, we also simplify analysis [1, p. 74, objective 8] and make it easier to verify key system invariants maintained by a given class [1, p. 70, objectives 1 and 4].

To be effective, the interface should provide a true abstraction of the data and other resources controlled by the class, rather than simple accessors (*get* and *set* operations) for each attribute or hardware register. Any invariants associated with the class should be established by the class constructor, and maintained by every publicly accessible operation. Access restrictions associated with the class interface permit a proof of the invariant to be local to the class, rather than global to the system and different for each system in which the class appears.

These principles apply not only to interfaces provided by the class to clients but to interfaces between superclasses and subclasses. In particular, it should be possible for the developer to define the interface between a class and its subclasses in the same manner as the interface between the class and its clients. Both can be considered ‘contracts’, only with different parties. The interface between client and class is concerned with how the class will be used. The interface between superclass and subclass is concerned with how the class definition and implementation can be extended. Both can be defined formally (in terms of pre- and post- conditions) when desired.

### 3.3.11.2 Applicability

These guidelines apply when control and data coupling between classes is a concern in an object-oriented system. Access to the attributes of a class is provided by public and protected operations, which can be inlined to avoid the overhead associated with a call, producing code comparable to that for direct access. When performance is critical, but the target compiler does not support inlining or does not perform inlining efficiently, use of these guidelines may not be appropriate.

### 3.3.11.3 Guidelines

The following rules address the basic issues associated with *Class Coupling*:

*Client data abstraction rule:*

- Clients should access the data representation of the class only through its public operations.
- All attributes should be hidden (private or protected), and all strategies associated with the choice of data representation should be abstracted by its set of public operations.
- All hardware registers should be hidden (private or protected), and all strategies associated with the use of a particular hardware device should be abstracted by its set of public operations.

*Invariant rule:* The invariant for the class should be:

- an implicit or explicit part of the postcondition of every class constructor,
- an implicit or explicit part of the precondition of the class destructor (if any),
- an implicit or explicit part of the precondition and postcondition of every other publicly accessible operation.

As a result, clients should be able to influence the value of the invariant only through execution of these operations. Private and protected operations are exempted in the *invariant rule* since the invariant need not hold at all times, but only at points where it is externally observable.

These guidelines may be extended to deal with the coupling between classes and subclasses by also adopting the following rule:

*Subclass data abstraction rule:*

- A subclass should access the data representation of its superclass only through the superclass' public and protected operations.
- All attributes should be hidden (private), and all strategies associated with the choice of data representation should be abstracted by its set of public and protected operations.
- All hardware registers should be hidden (private), and all strategies associated with the use of a particular hardware device should be abstracted by its set of public and protected operations.
- The class invariant should also be an implicit or explicit part of the precondition and postcondition of each protected method of a class, and part of the postcondition of every protected constructor.

To be most effective, both the client and subclass interfaces should provide a true abstraction of the data and other resources controlled by the class, rather than simple accessors (*get* and *set* operations) for each attribute or hardware register.

### **3.3.12 Deep Hierarchy**

The guidelines in this section address issues that can occur when the complexity of class hierarchies is very deep.

#### *3.3.12.1 Motivation*

Most class hierarchies have a characteristic depth of between three and six, irrespective of the application. Class hierarchies that are either too deep or too shallow can cause problems. Dynamic dispatch can introduce problems related to initialization, especially with regard to deep class hierarchies. Top-heavy multiple inheritance and deep hierarchies also tend to be error-prone, even when they conform to good design practice. The wrong variable type, variable, or method may be inherited, for example, due to confusion about a multiple inheritance structure. Binder refers to this as “spaghetti inheritance” [11].

#### *3.3.12.2 Applicability*

These guidelines apply when the complexity of the class hierarchy is a concern.

### 3.3.12.3 Guidelines

The following rule addresses the issues associated with *Deep Hierarchies*:

*Six deep rule:* Any class hierarchy with a depth greater than six warrants a careful review that specifically addresses the above issues and weighs this against the need to isolate various proposed changes. When extending an existing framework, depth should be measured from the point at which the framework is first subclassed. When developing an application specific class hierarchy, depth should be measured from the root. In languages in which all classes implicitly inherit from a common root class, this class should not be included in the count.

A class hierarchy that successfully passes an inspection should be marked so as to avoid repeated review with respect to the same issue.

A certain amount of variation in the threshold (e.g., plus or minus two) may also be expected based on the results of reviews using the six deep threshold. Deep hierarchies may also be more of a problem with respect to implementation inheritance than interface inheritance. The use of multiple inheritance can also be an important factor in adjusting the threshold.

This rule is based on the metrics for *Class hierarchy nesting level* appearing in [9, pp. 61-64]. Note that this rule does not, in anyway, imply that class hierarchies with a depth of six or less do not require careful review.

### 3.3.12.4 Related Guidelines

Related guidelines include those for *Inheritance with Overriding*, *Multiple Interface Inheritance*, and *Multiple Implementation Inheritance*.

## 3.4 Multiple Inheritance

### 3.4.1 *Purpose*

This section provides guidelines regarding safe implementation and use of multiple inheritance in projects that use object-oriented (OO) technology.

### 3.4.2 *Background*

Single inheritance, overriding, subtyping, and dynamic dispatch are described in Section 3.3. Multiple inheritance permits a class to have more than one superclass. It may involve either interface inheritance or implementation inheritance, or some combination of these. Interface inheritance involves the inheritance of only interface elements (such as operation specifications and constraints), while implementation inheritance involves the inheritance of implementation elements (such as methods, attributes, and references to other objects).

Multiple inheritance may lead to name clashes involving elements inherited from different superclasses that have the same signature. Some languages support renaming as a means of resolving such name clashes. Eiffel is particularly elegant in dealing with this [18]. Most other languages either require more complicated workarounds [22, section 12.8] or the editing of the superclass definitions to rename inherited elements.

Most issues arise with respect to multiple implementation inheritance because it is difficult to implement well, because associated errors have run-time consequences, and because the inherited elements reference one another and may interact in subtle ways, increasing overall complexity and the potential for error.

Because delegation is considered an effective substitute for multiple implementation inheritance, many more recent languages (such as Java and C#) only support multiple inheritance involving interface specifications. The Aerospace Vehicle Systems Institute (AVSI) Guide [21] also recommends the use of delegation rather than multiple implementation inheritance for systems certified to levels A, B, and C.

A number of issues arise when using multiple inheritance that may make compliance with DO-178B difficult. Section 3.4 documents the issues, related DO-178B sections and objectives, and applicable guidelines.

In general, these “guidelines” do not represent new “guidance”, but an interpretation of existing guidance (DO-178B) with respect to the use of particular OO features. The “rules” associated with these guidelines are also rules only in the sense that they must be followed in order to adopt the given approach. Often there are also alternative approaches that can be followed in order to address the same issues and still comply with DO-178B.

### 3.4.3 *Overall approach*

This section is intended to provide an approach for addressing DO-178B objectives when using OO features related to multiple inheritance. In this regard, multiple inheritance is treated as an extension of single inheritance and, as a result, all guidelines related to the use of single inheritance and dynamic dispatch also apply here. Issues related to multiple inheritance are listed in volume 2.

Guidelines that attempt to resolve these issues appear in sections 3.4.5 through 3.4.8. Each of these sections should be understood to provide one (of possibly many) approaches that assist in compliance to DO-178B objectives.

The overall collection of guidelines is open-ended. As a result, new approaches and new guidelines may be added that address the same issues as existing approaches, under different circumstances.

A sharp distinction is drawn between the use of interface and implementation inheritance. The guidelines for *Multiple Interface Inheritance* address the simpler case, in which we are concerned only with inherited operation specifications (that do not reference one another).

The guidelines for *Multiple Implementation Inheritance* deal with the more difficult case involving the inheritance of code and data. Although use of these guidelines helps us deal with the issues raised with respect to ambiguity and complexity, delegation is still considered preferable to the use of multiple implementation inheritance for most systems (as recommended by the AVSI Guide [21]).

The guidelines for the *Combination of Distinct Abstractions* provide an alternative to those for *Multiple Implementation Inheritance*. They forbid the use of repeated inheritance in order to eliminate related sources of ambiguity.

The guidelines for *Mixed Multiple Inheritance* address the case in which we have a mix of interface and implementation inheritance.

### 3.4.4 *Multiple Interface Inheritance*

Multiple interface inheritance permits the categorization of entities in terms of their interfaces, where each entity may appear in more than one category. These guidelines extend those for single inheritance and dynamic dispatch to address multiple interface inheritance. When applying the *Subtyping* guidelines, this means that a subclass with more than one superclass inherits the test cases defined by all its superclasses.

#### 3.4.4.1 *Motivation*

In the real world, objects are often classified in more than one way. Multiple interface inheritance allows us to model this without introducing redundancy (and without the complications associated with multiple implementation inheritance).

#### 3.4.4.2 *Applicability*

Multiple interface inheritance involves two or more superinterfaces, each of which contributes features (compile-time constants and operations) to a single subinterface. Each super-interface may, in turn, itself inherit from other interfaces, either singly or multiply. The resulting inheritance hierarchy forms a directed acyclic graph that permits the definition of common ancestors, and the inheritance of the same feature along more than one path (repeated inheritance). Features may also be redefined, potentially resulting in different definitions of the same feature along different paths. Appendix B.2 illustrates the following issues with multiple interface inheritance:

- Repeated inheritance,
- Redefinition along separate paths,
- Independently defined operations with same signature

#### 3.4.4.3 *Guidelines*

The guidelines in this section extend those for *Inheritance with Overriding* (section 3.3.4) to include the following rules that addresses the three issues listed above:

1. *Repeated interface inheritance rule:*

When the same operation declaration is inherited by an interface via more than one path through the interface hierarchy without redeclaration or renaming, this should result in a single operation in the subinterface.

2. *Interface redefinition rule:*

When a subinterface inherits different definitions of the same operation (as a result of redefinition along separate paths), the definitions must be combined by explicitly defining an operation in the subinterface that follows the *Simple overriding rule* (section 3.3.4.3) with respect to each parent interface.

3. *Independent interface definition rule:*

When more than one parent *independently* defines an operation with the same signature, the user must explicitly decide whether they represent the same operation or whether this represents an error. Such decisions should be recorded as explicit annotations to the source code. If the operations are not intended to be the same, one of them should be renamed. If the operations are intended to be the same, any preconditions and postconditions should also be the same.

#### 4. *Compile time constant rule:*

All of the above rules apply to compile time constants as well as operations. Constants whose value involves run-time computation should not be permitted in interfaces.

The rationale for the *Repeated interface inheritance rule*: is that cases involving the sharing of operations are common while cases that demand replication are not (Appendix B.2). Sharing is also supported by many languages, whereas replication is not. Therefore sharing is defined to be the normal, expected behavior and additional work is needed to support replication in those rare cases in which it is required.

The *Interface redefinition rule*: is derived from the guidelines for behavioral subtyping [7] that inspired the Simple overriding rule: (section 3.3.4.3). The user is required to define the operation representing the combination of the inherited definitions in order to make its specification explicit even when the language does not require it. The intent here is that clients of the sub-interface be able to *directly see* the result of combining the inherited definitions.

The *Independent interface definition rule*: requires the user to always explicitly decide when two independently defined operations with the same signature are intended to represent the same operation and when they are not. The intent here is to avoid errors resulting from the accidental matching of operation signatures.

The *Compile time constant rule*: specifies that compile time constants be treated in the same manner as operations with respect to the previous cases. It applies only to constants with an initial value that can be computed at compilation time. Constants whose value is computed at run-time require the generation of code to perform the computation and assignment. This, in turn, conflicts with the fundamental definition of an interface, which is not permitted to define either the code or the data.

Languages specific guidelines are provided in Appendix B.2. In general, it is only necessary to enforce (e.g., by means of design and code inspections) those guidelines that the language does not enforce itself.

#### 3.4.4.4 *Related guidelines*

The guidelines in this section are related to those for *Subtyping*.

### 3.4.5 *Multiple Implementation Inheritance*

Multiple implementation inheritance supports the construction of a class implementation in terms of the implementations of other existing classes. These guidelines extend those for single inheritance and dynamic dispatch to address multiple implementation inheritance. When applying the guidelines for *Subtyping*, this means that a subclass with more than one superclass inherits the test cases defined by all its superclasses.

#### 3.4.5.1 *Motivation*

Multiple implementation inheritance supports maximum reuse of code.

#### 3.4.5.2 *Applicability*

A given class C is implemented by inheriting the methods and attributes of two or more superclasses S1, S2.

#### 3.4.5.3 *Guidelines*

The guidelines in this section extend those for *Inheritance with Overriding* (section 3.3.4) to address the basic issues raised by the multiple inheritance of code and data in a manner consistent with the multiple inheritance of interface specifications:

1. *Repeated implementation inheritance rule:*

When the same feature (method or attribute) is inherited by a class via more than one path through the interface hierarchy, this should result in a single feature in the subclass.

2. *Implementation redefinition rule:*

When a subclass inherits different definitions of the same method (as a result of redefinition along separate paths), the definitions must be combined by explicitly defining a method in the subclass that follows the *Simple overriding rule:* (section 3.3.4.3) with respect to each parent class.

3. *Independent implementation definition rule:*

When more than one parent *independently* defines a method with the same signature, the user must explicitly decide whether they represent the same method or whether this represents an error. If they are intended to be different, renaming should be used to distinguish them. Otherwise, the definitions must be combined by explicitly defining a method in the subclass that follows the *Simple overriding rule:* (section 3.3.4.3) with respect to each parent class.

As in the guidelines on *Inheritance with Overriding*, it is recommended that decisions related to the *Independent implementation definition rule:* be recorded as explicit annotations to the source code.

Languages specific guidelines for C++ are provided in Appendix B.2. In general, it is only necessary to enforce (e.g. by means of design and code inspections) those guidelines that the language does not enforce itself.

#### 3.4.5.4 *Related guidelines*

The guidelines in this section are related to those for *Subtyping*.

### 3.4.6 *Mixed Multiple Inheritance*

Multiple inheritance may involve only interface specifications, may involve only implementations or may involve some combination of these. The guidelines in this section address the issues associated with the combination of interface and implementation inheritance (mixed multiple inheritance).

#### 3.4.6.1 *Motivation*

Often we want to define a class that implements one of more interfaces while building on the implementation provided by a second class. This can be accomplished in a number of ways. In situations where the resulting class is logically a subtype of the other classes, inheritance is a natural choice. This typically works well as long as all the superclasses are interfaces, save one.

In the more general case, involving an arbitrary combination of interfaces, abstract classes, and concrete classes, the problems associated with multiple implementation inheritance may also arise.

#### 3.4.6.2 *Applicability*

These guidelines apply when a class has at least one parent class that is an interface and at least one parent class that is not.

#### 3.4.6.3 *Guidelines*

Both the guidelines for *Inheritance with Overriding* and *Multiple Interface Inheritance* apply here. If more than one superclass provides an implementation, the guidelines on *Multiple Implementation Inheritance* also apply.

Verification in accordance with the guidelines on *Subtyping* (section 3.3.6) is also recommended.

### 3.4.6.4 Related guidelines

The *Top Heavy Hierarchy* and *Deep Hierarchy* sections provide metrics to limit the complexity of the inheritance hierarchy.

### 3.4.7 Combination of Distinct Abstractions

The guidelines in this section extend those for *Multiple Implementation Inheritance* by restricting the use of multiple implementation inheritance to cases that do not involve repeated inheritance or the redefinition of competing implementations along separate paths.

#### 3.4.7.1 Motivation

The guidelines in this section are intended to maximize the reuse of code while complying with the associated guidelines for *Subtyping*. Any use of multiple inheritance, however, can lead to ambiguities within the class hierarchy. Three potential sources of ambiguity are identified in Appendix A: (1) repeated inheritance, (2) redefinition along separate paths, and (3) independently defined operations with the same signature. The first two are eliminated by the guidelines in this section, which require that superclasses always be *distinct* rather than subclasses of a common superclass. As Meyer suggests, “This is the form that you will need most often in building inheritance structures, ...” [18, page 521].

#### 3.4.7.2 Applicability

A given class is implemented by inheriting the features of two or more distinct superclasses. The superclasses are considered *distinct* because they are not variants of a single abstraction [18, page 521] (have no common ancestors). The following example illustrates the basic structure.

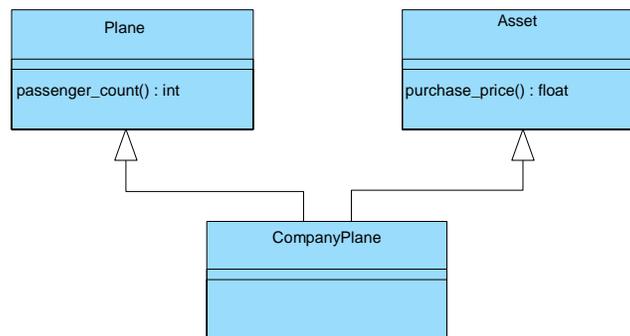


Figure 3.4-1 Combination of Distinct Abstractions

As described by Meyer [18, page 521], a class *Plane* describes the abstraction suggested by its name. Operations are provided to query the *passenger\_count*, *altitude*, *position* and *speed* of the airplane. Additional operations include commands to *take\_off* and *set\_speed*. In a completely different domain, we have a class *Asset* that represents something that a company owns. Our concerns here are related to accounting, the manner in which the asset is paid for, its depreciation and sale. Operations associated with an asset include queries to determine its *purchase\_price* and *resale\_value*, and the actions *depreciate*, *resell*, and *pay\_installment*.

These classes are then combined by means of inheritance to create a new class *CompanyPlane*. Because each superclass is taken from a different domain and because they have no common ancestors, the odds of inheriting two independently defined operations with the same signature is small.

### 3.4.7.3 Guidelines

Although the guidelines in this section extend those for *Multiple Implementation Inheritance*, the restrictions on inheritance structure eliminate ambiguities arising from repeated inheritance and redefinition along separate paths. As a result, we do not need the *repeated implementation inheritance rule* or the *implementation redefinition rule* defined by the guidelines for *Multiple Implementation Inheritance*.

Only the *independent implementation definition rule* from the guidelines on *Multiple Implementation Inheritance* is required, to handle cases in which superclasses independently define operations with the same signature. Because the superclasses are distinct, any ambiguity usually represents an error, and should result in a renaming of one of the inherited operations in order to make them distinct.

We must also follow the rules associated with the guidelines on *Inheritance with Overriding* and *Subtyping*.

*No diamond rule:* Repeated inheritance is not permitted, i.e. no subclass may inherit from the same superclass via more than one path.

*Independent implementation definition rule:* When more than one parent *independently* defines a method with the same signature, the user must explicitly decide whether they represent the same method or whether this represents an error. If they are intended to be different, renaming should be used to distinguish them. Otherwise, the definitions must be combined by explicitly defining a method in the subclass that follows the *Simple overriding rule* with respect to each parent class. (Identical to rule of same name in the section on *Multiple Implementation Inheritance*)

### 3.4.7.4 Related guidelines

The guidelines in this section are related to those for *Inheritance with Overriding* and *Subtyping*.

## 3.4.8 Top Heavy Hierarchy

The guidelines in this section address issues of complex class hierarchies that contain many classes and inherited features near the top (root) of the hierarchy. The intent is to reduce the number of opportunities for errors related to composition of competing parent implementations.

### 3.4.8.1 Motivation

Most class hierarchies have a characteristic shape. They are generally narrow near their top (root) and broad near their base, with a depth of between three and six. As a result the number of classes increases as a function of their distance from the root and the number of inherited elements increases in small steps.

Problems can arise when class hierarchies fail to exhibit this shape. A class hierarchy with many classes near the root, and with many features associated with these top-level classes can be difficult to understand and change. “Top-heavy multiple inheritance and deep hierarchies are error-prone, even when they conform to good design practice. The wrong variable type, variable, or method may be inherited, for example, due to confusion about a multiple inheritance structure.” [11, p. 503, spaghetti inheritance]

### 3.4.8.2 Applicability

These guidelines apply when the complexity of the class hierarchy is a concern and there are too many classes near its root. This is a particular concern when multiple inheritance is used and the number of features inherited from each of these upper level classes is large.

### 3.4.8.3 Guidelines

The following rules define the basic approach:

*Three parents rule:* Any class near the top of the hierarchy with three or more parents warrants careful review.

*Top heavy composition rule:* Any class near the top of the hierarchy that inherits more than 20 features from each of two or more parent classes warrants careful review.

*Top to bottom rule:* Any class hierarchy that contains more classes near the top of the hierarchy than near the bottom warrants careful review.

A class is considered “near the top of the hierarchy” if it appears in one of the top two levels. A class is considered “near the bottom of the hierarchy” if it appears in one of the bottom two levels. A class hierarchy that successfully passes an inspection should be marked so as to avoid repeated review with respect to the same issue.

#### 3.4.8.4 *Related guidelines*

Related guidelines include those for *Inheritance with Overriding*, *Multiple Interface Inheritance*, and *Multiple Implementation Inheritance*.

## 3.5 Templates

### 3.5.1 Purpose

Templates and the guidelines on Templates in this section are not unique to OOT. This section addresses issues raised and considered regarding DO-178B guidelines for development and verification when using templates in an object-oriented technology (OOT) environment.

### 3.5.2 Background

Templates provide a means of abstracting common structural and behavioral aspects of a family of classes or operations in a domain independent way. Template is the UML term for a parameterized model element with unbound (formal) parameters that must be bound to actual parameters before it can be instantiated.

At a target language level, templates correspond to Ada generics and to C++ templates. A template is a parameterized code replication feature that provides stronger typing than macros. Templates provide for reusability in programming languages. Consider a Stack with a generically parameterized base type. This allows a single Stack class to provide many class instantiations such as a Stack of integers, a Stack of any fundamental or user defined type, or even a Stack of Stacks.

A template's behavior results from its implementation, the values of the arguments used to instantiate the template, and the behavior of any types specified to it as arguments. The use of templates directly impacts: source code reviews, coding standards, requirements-based test case and procedure development and review, timing analysis, memory usage, requirements-based test coverage, source code to object code traceability, and structural coverage, including data coupling analysis and control coupling analysis. Dead code and deactivated code may also be a concern because unused functionality of a template may be considered either dead or deactivated code.

### 3.5.3 Source Code Review

#### 3.5.3.1 Motivation

The use of templates can affect source code reviews. Depending on the parameter types and the scope of the call, a different instantiation of the template may be invoked by the compiler. Each instantiation may use different sub-components and features of the template. The template may contain features that are not used at all by a specific application.

#### 3.5.3.2 Related DO-178B Sections and Objectives

The following DO-178B objectives for verification and coding standards are relevant to the recommendations in section 3.5.3.3 for using templates: Table A-1 objective 5, Table A-5 objectives 1 and 3, and Table A-9 objective 1.

#### 3.5.3.3 Guidelines

Standards need to address the issues identified in this section. The source code developer and code reviewer should be aware of the implications and potential effects of using templates. A template must be reviewed with respect to the actual parameters to determine if the source code is verifiable. Consequently, the following practices are recommended:

- It may be necessary for types to be defined in low level requirements to facilitate test coverage.
- Coding standards should require templates to document all the assumptions about types to be used with that template.
- Coding standards should be established to ensure the specific features of each template are understood, to ensure that they are the correct features for the instance, and that they comply with low level requirements.

- Coding standards should be established to determine which features of a template, if any, are not used by the application. Unused features may be considered dead or deactivated code.
- Each template should be reviewed with respect to the actual parameters.

### **3.5.4 Requirements-based Test Development, Review, and Coverage**

#### *3.5.4.1 Motivation*

Templates are instantiated by substituting zero or more specific arguments for each formal parameter defined in the template class or operation. Test cases and procedures are developed based on the software high-level, low-level, and derived requirements. Consequently, test developers and reviewers may not be aware whether a template contains additional functionality, and may not be aware of or have visibility into all the functionality contained within a specific function that is instantiated by a template call. In general, requirements-based test cases and procedures may not test all functionality of the template, especially those functions which are not instantiated for a particular template. All instantiations should be tested to guarantee that the template functions as intended [23][11].

#### *3.5.4.2 Related DO-178B Sections and Objectives*

See DO-178B section 6.4.4.3d, Table A-6: objectives 1-4, and Table A-7: objectives 1-4 regarding verification and integration processes.

#### *3.5.4.3 Guidelines*

Each instance of a template with a unique set of arguments should be tested for software at Levels A, B, and C. It is theoretically possible to test the template for all known instances *if the types map to the same underlying representation and object code can be shown to be equivalent*. In practice, the complexity of the instantiation process makes it difficult to verify all instances of a template without testing each instance individually [23]. It also complicates the requirements to test coverage traceability, as many tests may need to be executed to cover all possible instantiations that do not trace to specific requirement of the application. Also, the developer may need to provide protection that ensures that the unused functionality of the template (deactivated code) cannot be inadvertently activated. Therefore, while templates may lead to coding efficiencies, the use of templates may actually substantially increase the amount of requirements-based test development, review, and coverage needed.

### **3.5.5 Structural Coverage for Templates**

#### *3.5.5.1 Nested Templates*

Nested templates and using templates with other language constructs increases the complexity of the code. For example, child packages in Ada and friend classes in C++ can result in complex code. Although complex code is not prohibited by DO-178B, complexity can make structural coverage analysis more difficult.

##### *3.5.5.1.1 Related DO-178B Sections and Objectives*

The following DO-178B sections and objectives for integration and test coverage are relevant to recommendations in section 3.3.1.2 for using templates: DO-178B section 6.4.4.2, Table A-6: objectives 1-4 and Table A-7: objectives 5-7.

##### *3.5.5.1.2 Guidelines*

In general, Templates should be analyzed for complexity and complex Templates should be avoided. Nested Templates and Templates used with other language constructs should be analyzed for complexity and justified.

### *3.5.5.2 Templates and Object Code Traceability*

Templates can be compiled using "code sharing" or "macro-expansion". Code sharing is highly parametric, with small changes in actual parameters resulting in dramatic differences in object code. Object code coverage is difficult and mappings from a template to object code can be complex when the compiler uses the "code sharing" approach.

#### *3.5.5.2.1 Related DO-178B Sections and Objectives*

The following DO-178B sections and objectives for structural coverage are relevant to the recommendation in section 3.3.2.2 for using templates: DO-178B section 6.4.4.2 and Table A-7: objectives 5-7.

#### *3.5.5.2.2 Guidelines*

Code sharing is not widely used. In general, code sharing should be avoided.

### *3.5.5.3 Data and Control Coupling Analysis*

Use of templates can complicate data coupling analysis and control coupling analysis by not allowing visibility into the template for the analyst to verify that the correct template functionality is invoked for each instantiation (control coupling) based on the parameters (data coupling).

#### *3.5.5.3.1 Related DO-178B Sections and Objectives*

DO-178B section 6.4.4.3 c and Table A-7 objective 8 on data and control coupling are relevant.

#### *3.5.5.3.2 Guidelines*

Data and control coupling associated with templates should be evaluated with respect to the actual parameters.

## 3.6 Inlining

### 3.6.1 Purpose

Inlining and the guidelines on Inlining in this section are not unique to OOT. This section addresses issues raised and considered regarding DO-178B guidelines for development and verification when Inlining in an object-oriented technology (OOT) environment.

### 3.6.2 Background

When a compiler chooses to Inline, a method body is compiled without the call overhead (i.e., the Inlined method body's object code is physically placed in the calling method's object code). This is in contrast to the "usual" implementation of making a call (and potential context switch) during execution to a separate method with the associated parameters, if any being passed to the called method. If the compiler chooses not to Inline, a call to the method is inserted in the caller's object code, and the called method's object code remains separate. The Inlining of methods eliminates the overhead associated with a call, and is thus useful for optimization of performance. This performance optimization results in a space penalty, unless the Inlined method is shorter than the sequence of instructions used to make the call. Simple get and set methods, for example, are commonly used in object-oriented software and can sometimes be smaller than the calling sequence.

When Inlining, it is important to know if the compiler will honor or ignore the inline request, whether the code has been Inlined or not, and what the impact is to the code.

The following analyses are directly impacted by Inlining: memory and stack usage analyses, timing (performance) analysis, structural coverage analysis, and source code to object code traceability.

Inlining may affect a number of verification methods as noted below. The use of Inlining, however, is not an obstacle to certification so long as its effects are understood and documented, and each effect upon each of these verification methods is addressed

### 3.6.3 Inlining and Structural Coverage

Some language constructs in combination with Inlining can impact structural coverage analyses, including data and control coupling and source to object code traceability analysis.

#### 3.6.3.1 Related DO-178B Sections and Objectives

The following DO-178B sections and objectives for structural coverage and data and control coupling are relevant to the recommendations in section 3.6.3.2 for Inlining: Section 6.4.4.2 and Table A-7 objectives 5 through 8.

#### 3.6.3.2 Guidelines

Table A-7, objectives 5-8: In general, Inlining should consist of a simple expression only, which is almost always one line of code. Virtual methods and methods that access other class methods should not be Inlined.

Table A-7, objectives 5-7: Inline expansion may be handled differently at different points of expansion in order to optimize the code for the caller's context. The structural coverage of the Inlined method should be evaluated at the point of each expansion. If object code is removed or object code is added, as determined by the source to object code trace for Level A software, then structural coverage must be verified separately for each expansion.

Table A-7, objective 8: Inline expansion can eliminate parameter passing, which can affect the amount of information pushed on the stack as well as the total amount of code generated. This, in turn, can affect the stack usage and timing analysis. In addition, data coupling and control coupling relationships can transfer from the Inlined component to the Inlining component. For data coupling and control coupling, the verification approach should address the Inlining of code including worst-case memory usage analysis, stack usage analysis, timing analysis, call tree analysis, and data set/use analysis.

### **3.6.4**      *Source Code Review of Inlined Code*

Inlining can complicate source code reviews. Both the code developer and the source code reviewers must be aware of the implications and potential effects of Inlining. When an Inlined method is expanded in the context of the caller it may be possible for the compiler to simplify it in a number of ways, involving both space and speed. Compiler optimizations include directly referencing arguments and unrolling loops with known bounds as common examples. This is generally only a problem when an Inlined method is relatively complex and is optimized based on the context of the caller.

#### *3.6.4.1 Related DO-178B Sections and Objectives*

The following DO-178B objectives for verification are relevant to the recommendations in section 3.6.4.2 for Inlining: Table A-5 objectives 1 and 3.

#### *3.6.4.2 Guidelines*

Table A-5, objective 1: Review of the Inlined method against the low-level requirements is sufficient to verify the behavior of all expansions.

Table A-5, objective 3: Review of the Inlined method against the low-level requirements is sufficient to verify the behavior of all expansions.

## 3.7 Type Conversion

### 3.7.1 Purpose

Type Conversion and the guidelines on Type Conversion in this section are not unique to OOT. This section addresses issues raised and considered regarding DO-178B guidelines for verification and coding standards when types are converted in strongly typed languages, including when it is appropriate to convert types implicitly and when type conversion should be explicit. These “guidelines” do not represent new “guidance”, but an interpretation of existing guidance (DO-178B) with respect to Type Conversion for object-oriented (OO) languages that provide for strong typing with abstraction. Guidelines are in the form of recommended practices which support compliance with DO-178B objectives. An analysis should be conducted to examine the effects of type conversion for all languages as a part of satisfying the DO-178B verification objectives. Languages that are not strongly typed are not within the scope of this document. Examples of strongly typed languages include Ada, C++ and Java. Dynamic dispatch, which uses a form of implicit type conversion, is discussed in a separate chapter in this document.

### 3.7.2 Background

Strongly typed languages are an improvement over languages that are not strongly typed because they provide additional control of type conversion. User-defined type conversions are easier to identify and understand. Type Conversion may be implicit or explicit and may be checked (to determine if the type conversion results are valid and correct) or unchecked. 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 responsibilities. Checked types can be checked at compile time (producing a compilation error for an invalid conversion) or at run time (usually resulting in a run-time error). Conversion can result in loss of data. Unchecked type conversions need to be verified by test to ensure conversion was correct.

The following are directly impacted by implicit type conversion: potential loss of data or precision, performance and timing analysis, requirements-based test development, review and execution results, structural coverage analysis, data flow and control flow analyses, and source code to object code traceability.

### 3.7.3 Overall approach

Implicit type conversion raises certification issues related to the ability to perform various forms of analyses and to satisfy the verification objectives of DO-178B, including requirements-based testing and structural coverage analysis. Explicit type conversions can cause implicit loops and implicit conditionals. The use of explicitly checked type conversions are regarded as acceptable so long as they are properly verified and do not inhibit other verification methods, such as guaranteeing no loss of information and no unacceptable loss of data accuracy or precision.

### 3.7.4 Source Code Review, Checklist, and Coding Standards

Type conversions that are not checked by the language, either at compile time or at run time, can potentially result in code that has unintended behavior.

#### 3.7.4.1 Related DO-178B Sections and Objectives

For object code traceability, source code review, checklist, and coding standards, see DO-178B Table A-1: objective 5, Table A-5: objectives 1,3,4, and 6, Table A-6: objectives 1-4, Table A-7: objectives 3 and 4, and Table A-9: objective 1.

#### 3.7.4.2 Guidelines

Unchecked type conversions are error prone and should be addressed specifically in coding standards and during verification. Coding standards may address unchecked type conversions, as in the following recommended practice:

*Conversion rule:* To help ensure intended function and verification, all checked and unchecked conversions should be justified or should be explicit, use the most restrictive conversion available, be conspicuously marked (identified) in the program source code, and be permitted only after thorough review and analysis of potential adverse effects.

Verification of unchecked type conversions may be accomplished through code reviews, checklists, or analysis (e.g., static code checking), and testing.

### **3.7.5**      *Loss of Precision in Type Conversions*

Conversions, both implicit and explicit, that result in loss of data, data accuracy, or precision result in code that may be incorrect. The following type conversions, for example, may result in loss of data or precision in some languages:

- from integer types to the floating point types
- from a floating point type to an integer type
- from a more precise numeric type to a less precise version of the same numeric type; e.g. long to short, double to float, etc.

Loss of data, data accuracy, or precision can be especially difficult to analyze and detect for implicit conversions and can be language dependent.

#### **3.7.5.1**    *Related DO-178B Sections and Objectives*

For source code review, checklist, coding standards, and data coupling analysis, see DO-178B Table A-5: objectives 1, 3, 4 and 6; Table A-6: objectives 1-4; and Table A-7: objective 8.

#### **3.7.5.2**    *Guidelines*

Type conversions should be addressed for loss of data, data accuracy, or precision. This may be accomplished through coding standards and through verification including code reviews, checklists, or analysis. Coding standards may address type conversions that can result in loss of data, data accuracy, or precision, as in the following recommended practice:

*Loss of information rule:* To help ensure correctness, any conversions that may result in loss of data or data accuracy and precision should be justified or should:

- be explicit,
- use the most restrictive conversion available,
- be conspicuously marked (identified) in the program source code, and only be permitted after thorough review and analysis of potential adverse effects.

A clear understanding of the programming language being used is needed to identify and analyze implicit type conversions for potential loss of information.

### **3.7.6**      *Type Conversions of References and Pointers*

Some references and pointers can be implicitly converted. Converting a reference from one type to a dissimilar type can result in code that has unintended behavior and is difficult to verify.

#### **3.7.6.1**    *Related DO-178B Sections and Objectives*

For source code review, checklist, and coding standards, see DO-178B Table A-5: objectives 1, 3, 4 and 6; and Table A-6: objectives 1-4.

### 3.7.6.2 *Guidelines*

The following recommended practice may address implicitly converted references and pointers that result in code with unintended behavior:

*Supertype rule:* To help ensure intended function and verification, all implicit type conversions involving references/pointers to class instances should be justified or should only represent a conversion from a subtype to one of its supertypes.

### 3.7.7 *Language specific guidelines*

As a result of the proposed solutions, the following language best practices should be taken into consideration [Source code review, checklist, coding standards]:

1. All implicit conversions should be checked for potential loss of precision or loss of data. Specifically, the following should be justified:
  - From integer types to the floating point types (potential loss of precision) (JAVA and C++)
  - From a floating point type to an integer type (potential loss of data) (C++)
  - From a more precise numeric type to a less precise version of the same numeric type; e.g. long to short, double to float, etc. (potential loss of data or precision) (C++)
2. Implicit conversions between logically unrelated types should be justified. Types are logically unrelated when one does not define a set of operations that is a subset of the other. For example, in C++ single argument constructors, such as a stack class taking a single integer argument, would allow implicit conversion. It is a good programming practice to use the keyword “explicit” to avoid implicit conversion between logically unrelated types. Unless the single argument constructor was created with the expressed purpose of permitting implicit conversion between the argument type and the class type, the constructor should be declared with the explicit keyword. Not using the explicit keyword should be justified.

## 3.8 Overloading and Method Resolution

### 3.8.1 *Purpose*

Overloading and the guidelines on Overloading in this section are not unique to OOT. This section addresses issues raised and considered regarding DO-178B guidelines when overloading is used. Guidelines are in the form of recommended practices which support compliance with DO-178B objectives. These “guidelines” do not represent new “guidance”, but an interpretation of existing guidance (DO-178B) with respect to overloading and method resolution.

### 3.8.2 *Background*

Overloading means: using the same name for different operators or behavioral features (operations or methods) visible within the same scope. Overloading is a simple but useful form of static polymorphism when used consistently. Conversely, too much overloading and improper use of overloading can make source code readability more difficult, and can thus contribute to human error.

Overloading can enhance readability when the overloaded operators, operations or methods are semantically consistent. However, overloaded operators, operations, and methods could coincidentally have the same name and potentially have very different semantic behaviors. Specifically, overloading can be confusing when it introduces methods that have the same name but different semantics. This may be further complicated when overloading is combined with other language features (e.g., overriding, templates, etc.). Overloading can also complicate matters for tool use (e.g., structural coverage and control flow analysis tools) if the overloading rules for the language are overly complex.

Overloading may affect a number of verification methods as noted below. The use of overloading is not an obstacle to certification so long as it is verified that the intended operation is, in fact, the operation called.

### 3.8.3 *Code Review Method*

Overloaded methods and operators can introduce unintended functionality. Overloading special language constructs, such as indexing or dereferencing, can make verification difficult. Overloading methods and operators inconsistently can result in ambiguous code.

#### 3.8.3.1 *Related DO-178B Sections and Objectives*

Overloading can potentially result in code that is overly complex and difficult to verify. See DO-178B [1] Table A-5: objectives 3-4 related to verifiability of source code and conformance to standards.

#### 3.8.3.2 *Guidelines*

Overloaded methods and operators should be addressed for inconsistencies which can lead to code complexity and ambiguities. Overloading special language features should be discouraged or justified. This may be accomplished through code reviews, checklists, or coding standards. Software code standards should include complexity restrictions that address overloading. For example, the following practices are recommended:

*Overloaded method rule:* Overloaded operations or methods should form "families" that use the same semantics, share the same name, have the same purpose, and that are differentiated by the types of their formal parameters.

*Overloaded operator rule:* Overloaded operators should obey the "natural" meaning and follow conventions of the language. For example, a C++ operator "+=" should have the same meaning as "+" and "=". Arithmetic operators should be overloaded using conventional notation whenever possible.

*Overloading in general:* When calls are overloaded, reviewers should know exactly what is being called. Overloading of special language constructs must be justified.

In performing code reviews, the pre- and post- conditions for overloaded methods should be examined for consistent behavior in the context of the code under review.

### **3.8.4**      *Implicit Conversion*

Object-oriented languages support varying levels of implicit conversion of arguments. That is, arguments can be implicitly converted to match the arguments of a method with the appropriate signature. The use of overloaded operators and methods with arguments that are implicitly convertible can potentially result in problems associating calls with methods that do not share the same structure of preconditions and postconditions.

#### *3.8.4.1 Related DO-178B Sections and Objectives*

Overloading can impact structural coverage analysis and data and control flow analysis. Overloading can also inhibit source to object code traceability. See DO-178B [1] Table A-7: objectives 5-8 related to test coverage.

#### *3.8.4.2 Guidelines*

To avoid potential problems involving the association of calls with methods, it is recommended that any family of overloaded operators and methods whose arguments (signatures) are implicitly convertible to one another or from one to another be required to have the same semantics. Formally, this means they must have the same structure of preconditions and postconditions. Informally, this means that they may share the same structure of test cases.

The software user of any family of overloaded operators and methods whose arguments (signatures) are implicitly convertible should: (1) perform the call using arguments that do not need to be converted and (2) perform analysis appropriate to the level of the software to ensure the proper method is being called.

## 3.9 Dead and Deactivated Code, and Reuse

### 3.9.1 Purpose

This section identifies issues, perspectives, and recommendations for addressing concerns associated with software reuse, deactivated code, and dead code contained in aviation software applications as a result of using object-oriented technology (OOT) development processes, environments, and tool libraries. This section provides clarification for certain sections of DO-178B and contains no new or additional guidance material.

### 3.9.2 Background

A major objective of OOT is to provide developers with the ability to create new software systems utilizing reusable software components where a component may be comprised of classes, methods, procedures, packages, modules, and so on. A reusable component often contains more software functionality than required by the system being certified. The requirements and design of the reusable component should be more generic and cover more situations if the component is truly reusable. If this extra functionality results in extra code in the system itself, then there may be deactivated code with which to deal. Deactivated code will likely be present in any application that uses general purpose software components and libraries, such as commercial off-the-shelf (COTS) software libraries provided with a compiler, operating system or run-time environment, for object-oriented development frameworks.

Particular areas of concern for deactivated code when utilizing object-oriented (OO) techniques include:

- Compiler generated default methods (e.g., default constructors, destructors, and assignment operators)
- Completely unused classes
- Unused methods within classes
- Unused operations within overloaded methods
- Overridden methods due to use of sub-classes
- Existence of unexecuted paths because the entry conditions for those paths are never satisfied for a given system
- Unused class attributes
- Unintended functionality or anomalous behavior

RTCA documents DO-178B [1] and DO-248B [2] provide guidance for reuse and modification of Previously Developed Software (PDS), deactivated code, and dead code. FAA Order 8110.49 [24] and AC 20-RSC [34] also provide additional guidance for the use of reusable software components including life cycle data. PDS is included in this discussion as it is one form of a reusable component. DO-178B guidance for upgrading software from a previous development (such as DO-178A) still applies.

OO components taken from the non-avionics or non-commercial avionics industries and reused may have been designed to be highly reusable, but quite often not all of the necessary software life cycle data is available. Again, normal guidance from DO-178B applies in these situations. Generally, the product and process assurance artifacts must be produced as part of a first software approval.

While DO-178B already provides guidance for deactivated and dead code, significantly more software component reuse is expected through OOT. The following subsections provide elaboration of issues which may become more pervasive as more and more software components are reused through the application of OOT. The issues for reuse, dead code, and deactivated code are addressed in the following subsections. References to existing DO-178B guidance are provided, as appropriate.

### 3.9.3 Reuse of Software Components

Reuse of software components may occur in three general ways. One way is through the use of “generic” or non-application specific software components. Another approach, perhaps within the same system, is through the creation of multiple, option-selectable, “custom” or application specific components that implement variants of the

same algorithms. Finally, application specific components may be reused from one system in another similar but not identical system.

Common examples of non-application specific reusable components are operating systems, standard libraries, math libraries and so on. Often these reusable components are developed by a third party to be highly reusable. The components may or may not be written using OOT; this detail may be hidden to the application developer. These non-application specific components may be provided as object libraries, which are linked into the system or alternatively may be provided as source libraries in the form of an OO framework.

An example of reusable, option-selectable, application specific software components would be two slightly different flight control algorithms which correspond to two different airframes. In this case, some of the classes are active in one installation and deactivated in another.

Application specific reuse occurs frequently and is generally desirable. However, unintended functionality may result. In one well-publicized case an inertial reference system was reused from one series of launch vehicle to another. Unfortunately the reused software was not modified to consider a different launch profile used by the latter series of launch vehicles. This oversight resulted in the destruction of a launch vehicle early in the launch process.

Two of the previously noted approaches will likely result in some unused functionality as well as deactivated code. All of the approaches could potentially lead to *unintended* functionality. In all cases the software and the supporting software life cycle data must satisfy the objectives of DO-178B.

### 3.9.3.1 Related DO-178B Sections and Objectives

See DO-178B sections 2.4e, 4.2h, 5.4.3a&b, 11.1g, 11.10k, 12.1, 12.1.1, 12.1.2, 12.1.3, 12.1.4, 12.1.5, and 12.1.6. Additional clarification is contained in DO-248B sections 3.8 and 3.70. Also refer to AC 20-RSC [34] and FAA Order 8110.49 [24].

### 3.9.3.2 Guidelines

For standard libraries, the applicant needs to identify which Application Programming Interfaces (API) of the library are used and unused. For option-selectable software, the applicant needs to identify which classes and methods within classes are used and unused for particular installations.

The use of individual subclasses, each of which represents a different option, is a good way to implement option selectable software. Only the appropriate sub-class is actually created in the system. The caller of the desired algorithm need not know which algorithm is actually in the system – it is hidden from the other parts of the program.

Deactivated code for both forms of reusable components should conform to the *Developer's intent rule* and the *All code verified rule*. In both cases unused interfaces, classes, and methods will be tied to either derived requirements or explicit, option-selectable requirements specified by the integrator. The derived requirements must be verified and driven into the System Safety Assessment (SSA) for consideration by the integrator. One acceptable means to identify unused functionality to the integrator would be through the use of traceability matrices. Verification data for derived requirements must satisfy the objectives of DO-178B. In addition, if the deactivated classes and methods are built into the executable object code for the various installation options, analysis should be performed to show that the deactivated code in a particular installation cannot be activated.

*Developer's intent rule:* All code must be exercised by requirements-based tests (the requirements *may* be derived). Code not associated with requirements should be carefully evaluated and either removed (if it is dead code) or requirements should be developed for the code. Code which exists due to derived requirements needs to be explicitly identified by the developer and the associated requirements must be noted as "derived" for inclusion in the SSA process by the integrator.

*All code verified rule:* All code executed within any aircraft or engine configuration must be verified per applicable DO-178B objectives, even if it can be demonstrated that a particular piece of code can never be activated in a specific system. Note that requirements will be needed for all deactivated code associated with the various aircraft or engine configurations (either derived requirements or explicit, option-selectable requirements specified by the integrator).

Reused components must be associated with requirements for the new system. The level of reuse may vary and include reused requirements data, reused code, and some reused of verification data. However the reused data must still satisfy the *Developer's intent rule* and be consistent with the SSA. Objective evidence must exist to demonstrate all reused data satisfies the objectives of DO-178B.

### 3.9.4 *Requirements Traceability*

Traceability can be challenging when utilizing OO techniques. Of particular issue is how dead code, deactivated code and active code can be differentiated and verified in OO software. Given a method that is not used, or a method of an abstract class that is overridden in all subclasses, or an attribute that is never referenced, it may not be clear if this was intentional for possible future use, an accident, or an error. That is, in terms of source code the developer's original intent may not be clear. It should be noted that additional traceability concerns are documented in Section 3.11 of this handbook. This subsection just focuses on traceability as it relates to reuse, dead code, and deactivated code.

#### 3.9.4.1 *Related DO-178B Sections and Objectives*

See DO-178B sections 5.4.3a&b, 5.5, 6.3.1a, 6.3.2a, 6.3.4e, 6.4.3, 6.4.4.1, 6.4.4.2, and 6.4.4.3. Also refer to DO-248B sections 3.8, 3.28 and 3.70.

#### 3.9.4.2 *Guidelines*

Requirements should be developed to a sufficiently low level of detail so that the traceability between requirements and corresponding classes, methods and operators is clear. Tools to support OO software design and assist with traceability relationships are highly recommended.

An applicant must demonstrate that an adequate process is in place to resolve dead code issues. A careful evaluation of apparently dead code is needed to ensure that code that appears dead is actually dead and that it is not just a documentation omission that makes it appear to be dead code. The *Developer's intent rule*, as discussed in section 3.9.3.2, should provide the documentation to make this task much easier. The process also needs to cover overriding of class methods including compiler generated default methods, operators, virtual functions or other OO-specific instances where confusion might arise due to the OO structure of the code.

The following points should be observed when dealing with deactivated code:

- Deactivated code which is intended to operate in any configuration used within an aircraft or engine requires associated explicit option-selectable requirements specified by the integrator or derived requirements created by the developer.
- The design and architecture life cycle data will need to account for deactivated methods and attributes.
- The documented traceability relationships should exist from the source code to either derived or explicit requirements (deactivated code will have associated requirements – dead code will exist due to a design error and there will be no associated requirements).
- The completeness of requirements based tests against structural coverage objectives (derived requirements must be verified and will have associated source code – dead code will exist due to a design error and there will be no associated requirements).
- The effect of derived requirements on the SSA process and an examination of unintended functionality which could be introduced by the derived requirements (derived requirements must be provided to the SSA process).
- Runtime examination of methods invoked when the software component is integrated into the final target environment (control and data coupling/flow coverage are part of the DO-178B objectives – reusable software component interfaces will need to be examined by the integrator).

Note that many OO languages provide features that if fully utilized will make the above activities much more difficult – e.g., multiple inheritance. Developers may find it useful to develop standards which create a deterministic, verifiable subset of a given OO language.

In the case of Level A software, object code not directly traceable to the source code will also be of concern as noted in DO-178B in 6.4.4.2, item b. Examples of non-traceable, compiler generated object code could include default constructors, default destructors, default copy methods, and default assignment methods. The developer should provide explicit guidelines in terms of utilization of default methods and the appropriate verification techniques in the planning and standards documents.

See Appendix B.3.1 for an example of deactivated code with both deactivated methods and attributes.

### **3.9.5 Certification Credit for Reused but Modified Class Hierarchy**

When a previously approved class hierarchy is updated, it can be unclear how much re-verification must be performed. Current guidance requires that a software change impact analysis must be performed to determine the extent of required re-verification activities. The situation is no different for OOT. See Appendix B.3.2 for an example of how this class hierarchy change can have a subtle effect without any obvious changes to code.

#### **3.9.5.1 Related DO-178B Sections and Objectives**

See DO-178B [1], sections 11.3h and 12.1. Also, refer to FAA Order 8110.49[24].

#### **3.9.5.2 Guidelines**

This issue applies to a certification when class hierarchies are not being completely re-tested. In this case, applicants must provide a regression analysis of all changes to a class hierarchy in the form of flattened class hierarchy. More succinctly, the applicant should adhere to the following rule:

*Flattened class re-verification rule:* When a change to an element of a class occurs, re-verification of all subclasses whose flattened form contains the changed element is recommended.

A clear trace of the subclasses that are affected by changes in base classes could be created either manually or with tools.

If full re-testing is performed, an alternative (and recommended) approach is to apply the guidelines described for the *Inherited test case rule* (section 3.3.8.3). This states that “every test case appearing in the set of test cases associated with a class should appear in the set of test cases associated with each of its subclasses.” This conforms to the Liskov Substitution Principle [7] and will ensure the behaviors of the inherited methods are appropriate for each subclass.

### **3.9.6 Changes in the Status of Deactivated Code Versus Actively Used Code**

#### **3.9.6.1 Related DO-178B Sections and Objectives**

See DO-178B [1, sections 5.4.3a, 6.4.4.2c, 11.1g, 11.10k, 12.1.2a, and 12.1.3e]. Also, refer to AC 20-RSC [34].

#### **3.9.6.2 Guidelines**

When previously developed software is submitted by an applicant for a new certification, changes in the status of deactivated code versus actively used code must be documented. Since the reusable components should follow the *All code verified rule* as previously discussed, the components should have already have been verified to obtain regulatory approval. However, in order to gain regulatory approval for the previously deactivated code, the integrator will need to document and verify the interfaces associated with the newly activated code. This may drive requirements, design, code, and verification changes from the previously approved baseline.

### 3.9.7 *Service History Credit and Deactivated Code*

Software verification data has the potential to be valid for a particular certification basis and installation but not valid for different certification basis and installation. Often, applicants may apply for service history credit rather than re-doing the artifacts to comply with the certification basis.

For example, assume an entire class hierarchy is developed and approved under DO-178B. A portion of this class hierarchy is approved in a certified system and has 10 years of service history. Ten years later, a new system is developed and potentially the regulatory guidance or certification basis has changed and it is now desired to use portions of the previously unused class hierarchy. It is not clear that one can use service history at this point or the old certification as a baseline, because of the activation of previously inactive code.

#### 3.9.7.1 *Related DO-178B Sections and Objectives*

See DO-178B [1, section 12.3.5]. Also, refer to DO-248B [2, 3.19 and 4.4].

#### 3.9.7.2 *Guidelines*

Service history credit may be granted for deactivation mechanisms with appropriate service history data.

The applicant should adhere to the following rule:

*Service history rule:* Service history credit may only be given for activated code and deactivation mechanisms that have been actually executed. The target environment, certification basis, and SSA will need to be considered in this process.

Integrators and regulators need to be aware that deactivated code in a previous certification basis could easily become active in a newer certification effort (previously identified derived requirements for deactivated code that were provided for the earlier SSA process can assist this activity). Integrators and regulators need to review the documentation for code that has now become active and ensure that at least for that particular code, proper certification artifacts and life cycle data for the appropriate (new) standard as well as potentially new hazards within the system have been addressed. Code that becomes deactivated in a later certification is fine to leave in the code, as long as it is documented per volume 2 of this document. Note that DO-248B [2, section 4.4] addresses this issue.

## 3.10 Object-Oriented Tools

### 3.10.1 *Purpose*

This section addresses issues impacting compliance with DO-178B objectives for traceability, configuration management, development, verification, structural coverage analysis (including data coupling and control coupling analyses), dead code, and deactivated code when object-oriented development and verification tools are used. Additionally, tool issues related to dynamic dispatch, polymorphism, inheritance, frameworks and automatic code generation will be discussed. Tool qualification with regard to object-oriented technology (OOT) will also be discussed. As with most of the sections in this handbook, UML terminology and examples are used, since UML is the predominant OO modeling language currently being considered by the aviation community, and other modeling languages present similar issues.

### 3.10.2 *Background*

Current object-oriented (OO) tools (either internally developed or commercially procured) may not fully satisfy DO-178B objectives with regard to configuration management, development, and verification. This section provides guidelines for object-oriented tools to assist applicants in satisfying DO-178B objectives. OO may present new challenges to OO tool vendors and applicants that have not been prevalent with structure-based development. The specific OO tool issues and guidelines are presented in the following sections; however, the more general concerns are:

1. Addressing verification coverage for OO software,
2. Using OO frameworks, automatic code generators, dynamic dispatch, polymorphism, and inheritance, and
3. Addressing requirements management and traceability during OO development.

### 3.10.3 *Traceability When Using OO Tools*

Specific traceability issues and guidelines are addressed in sections 3.11.4 and 3.11.6 of this handbook. OO tools should implement the best practices as defined in those traceability sections and recommended guidelines for addressing those issues.

### 3.10.4 *Configuration Management When Using Visual Modeling Tools*

When using OO tools to develop software requirements, design and implementation, it is beneficial to work at the visual model level, especially when using UML. When working with OO tools, configuration management might be done at the modeling level (i.e., diagrams). This may cause a concern when the OO tools can introduce subtle errors into the diagrams. Since the model contains the software requirements and design, there must be some checks or assurance that the raw data files (model files) are not changed by the tool when saving, opening, checking files into the Configuration Management (CM) system or checking files out of the CM system. Some of the OO tools may introduce errors at the modeling level when opening, saving, or closing files. If the output of the tools is used by a subsequent process or configuration item, they may require special attention to establish determinism.

#### 3.10.4.1 *Related DO-178B Sections and Objectives*

See DO-178B [1, Section 7] on the software configuration management process and objectives. Files that are part of the software “type design” (See [1], Section 9.4) should be controlled to Control Category 1 (CC1) criteria ([1], Section 7.3 and Table 7-1, and Annex A, Table A-2). It is likely that the visual models and charts and their corresponding files should be controlled to CC1, as they represent the Software Requirements Data ([1, Section 11.9]) and the software Design Description ([1], Section 11.10). Guidance for tool qualification is in [1, Section 12.2].

### *3.10.4.2 Guidelines*

The developer should control the visual models, charts, any intermediate translations of the models and the code generated from the model. It is important to understand that there are two parts to UML: (1) a graphical notation, and (2) an underlying model representation with a well-defined semantics. The diagrams are simply views of the underlying model, and are physically separate from it. Each model element may appear in any number of diagrams or none at all. In the extreme, a UML model need not have any graphical representation at all. If the tools used to develop the models will automatically track the different components of the software requirements and design, then the tool may need to be qualified, unless the tool can be shown to be deterministic or the output of the tool is verified.

Visual modeling tools should be shown to be deterministic and preserve software life cycle data integrity. Software visual modeling tools may need to add integrity checks to their raw data files to ensure that when the tool opens or closes a file, the file's integrity is maintained. For example, checksums or printed copies of the models can be compared to the electronic visual models to ensure integrity or electronic copies could have integrity checks (cyclic redundancy checks) of the files computed and compared to ensure the tool does not introduce errors. The visual tools can also be qualified as software development tools.

As with any software life cycle data, DO-178B configuration management concepts apply to the artifacts of OO tools (i.e. frameworks, model files, manual or automatic generated code, and the software environment)

## **3.10.5 Visual Modeling Tools Frameworks**

Current visual modeling tools that are used for OO development make use of frameworks for automatic code generation, replacing tedious programming tasks. Frameworks may include patterns, templates, generics, and classes in ways requiring new verification approaches. The tool's framework may or may not enforce requirements, design and coding standards. The tool's framework may or may not ensure that the relevant objectives for software requirements, design and source code are achieved for all software to be included in the final software to be embedded in the airborne system.

### *3.10.5.1 Relevant DO-178B Sections and Objectives*

DO-178B [1], Sections 4.4, 4.4.1 and 4.4.2 contain guidance for planning the software development environment to be used to develop the airborne software, including some language and compiler considerations. The environment and tools to be used are specified in [1], Section 11.15, the Software Life Cycle Configuration Index. Tool qualification criteria for software development tools are in [1], Section 12.2. Since components of the "framework" may end up embedded in the airborne software, guidance related to Previously Developed Software should also be addressed [24].

### *3.10.5.2 Guidelines*

Some of the OO tools provide a framework to automate and generate the source code from the UML model and framework "libraries" of patterns, templates, generics and classes; replacing the tedious coding tasks for the user. This will add code or objects (i.e., some tools may generate dynamic constructors, destructors, queues, stubs, skeletons, and other features). Engineers should understand the tools that are used to develop the airborne application and what components of the framework are ending up embedded in the airborne software. The framework and the tool may need to be qualified, and components of the framework embedded in the airborne application should be assured to the same level as the application.

If components of the framework are going to be part of the airborne application, they must be assured to the same level as other components of the airborne application.

## **3.10.6 Automatic Code Generators**

Current visual modeling tools that are used for OO development provide a capability to generate source code directly from UML models. Most of the existing UML tools today can use visual modeling diagrams to construct models and generate source code from these models. The level of source code generation depends on the tool and on

the user of the tool. Some of the tools support “full” automatic code generation and some generate only the skeleton of the code. For “full” automatic code generation, the user may also need to produce additional information (e.g., state charts, activity diagrams, and/or UML action semantics) to specify additional detail of their implementations.

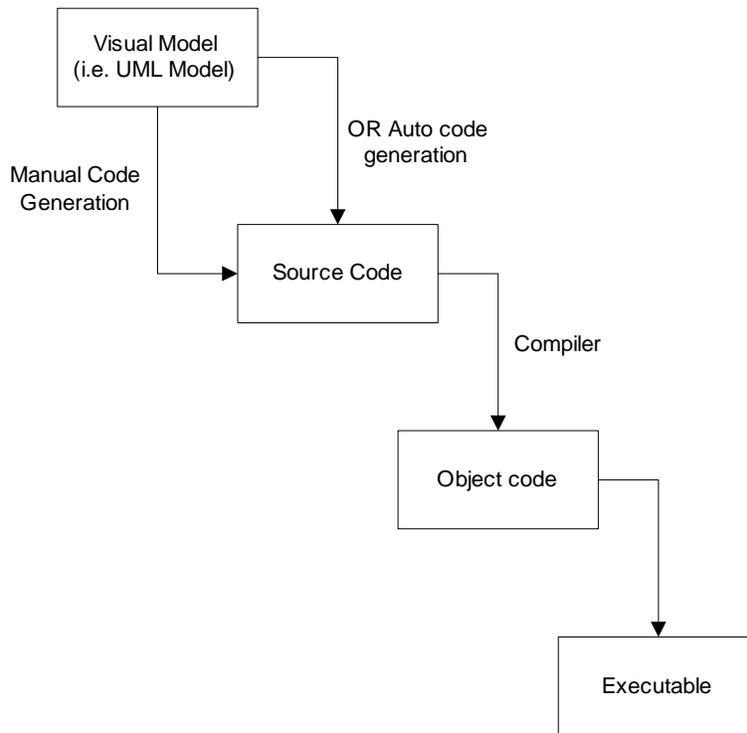


Figure 3.10-1 Code Generation using Visual Modeling Tools

### 3.10.6.1 Applicability

When discussing automatic code generation, this section assumes that the tool is an automatic code generator that generates source code. Then the source code is provided to a compiler that generates the object code. The object code can then be linked to generate the executable object code, and then loaded (or burned) into computer memory for execution. The tool may automate this process by providing one make file to do all of the steps. The code generated has to comply with the coding standards; the user may or may not have control over the code generator. This may be a problem is the code generated does not comply with the coding standard.

### 3.10.6.2 Related DO-178B Sections and Objectives

See DO-178B [1] sections 5.5 and 6.4.4, and see section 12.2 for software development tool qualification.

### 3.10.6.3 Guidelines

Most of the OO UML-based tools use some type of automatic code generation. Since the UML is a formally defined language, the tools can take the class diagrams, object model diagrams, state charts, and activity diagrams and generate the source code, or at least the structure. The code generator is a software development tool that may need to be qualified to the same level as the airborne software application (see section 12.2.2 of DO-178B [1] and FAA Order 8110.49 [24]). Qualification will be of a defined configuration. If the automatically generated code will be manually modified, the applicant should define and implement a process for controlling the automatically generated code, manually modified code and keeping track of all changes.

### **3.10.7 Structural Coverage Analysis Tools**

Because of the large manual effort required to perform and measure structural coverage, developers have become increasingly reliant on tools that measure the structural coverage and identify “gaps” in that coverage. The current structural coverage tools available may not “be aware” or have visibility to the internals of inherited methods and attributes and polymorphic references supported with dynamic binding such that they can provide a reliable measurement per section 3.12 of the structural coverage achieved by the requirements-based testing. Sections 3.12.4 and 3.12.5 provide the details for inheritance and dynamic binding respectively.

### **3.10.8 Structural Coverage Analysis for Inheritance**

Current tools performing structural coverage analysis of inheritance fall into two classes as identified in chapter 13:

- Those that perform concrete coverage analysis,
- Those that perform context coverage analysis.

If a context sensitive structural coverage analysis tool is used, then where unnecessary re-verification was not performed, an explanation will need to be given for the shortfall in the structural coverage analysis coverage report. The HIT analysis can be used to document that re-verification was not necessary.

If a context-insensitive (i.e., concrete) structural coverage analysis tool is used, the process will need to be augmented to ensure the appropriate re-verification of inherited features occurs, with the corresponding structural coverage of the inherited feature in the inherited context. The HIT analysis can be used to identify the necessary re-verification.

#### **3.10.8.1 Related DO-178B Sections and Objectives**

See DO-178B [1] Table A-7 objectives 5, 6, 7 & 8, and sections 6.4.4.2 and 6.4.4.3 regarding structural coverage analysis and related sections 6.3.6, 11.3c(2), d and g, 11.13, and 11.14.

#### **3.10.8.2 Guidelines**

Both applicants and the FAA need to understand which form of analysis for inheritance their structural coverage analysis tools perform. If the tool performs analysis on concrete methods only, then the verification process will need to identify where additional verification is needed, and that the appropriate tests are run and the appropriate structural coverage is obtained. If the tool performs analysis on both concrete and inherited methods, then the verification process will need to identify where tool reported non-coverage was not needed. Additionally, the structural coverage tool may need to be qualified as a software verification tool.

### **3.10.9 Structural Coverage Analysis for Dynamic Dispatch**

Two specific issues of concern for structural coverage are related to dynamic dispatch. The first issue exists because many current Structural Coverage Analysis tools do not “understand” dynamic dispatch, i. e. do not treat it as equivalent to a call to a dispatch routine containing a case statement that selects between alternative methods based on the run- time type of the object. As well, control and data flow analysis requirements of DO- 178B with respect to dynamic dispatch are more complex with respect with dynamic dispatch.

As identified in Section 3.11, there are multiple approaches to the structural coverage analysis of dynamic dispatch (polymorphic reference). As the discussion in Section 3.11 makes clear, the proper verification of polymorphism and dynamic dispatch is still an active research area. Until a final answer is in, a minimal requirement should be the execution of all polymorphic references, and the execution of all possible dispatches collectively (i.e., coverage of all entries in the dispatch table) [26]. Unfortunately, current tools performing structural coverage of polymorphism with dynamic dispatch only measure the execution of the polymorphic reference.

#### **3.10.9.1 Related DO-178B Sections and Objectives**

See DO-178B [1] sections 6.4.4.2 and 6.4.4.3; and also related sections 6.3.6, 11.3c(2), d and g, 11.13, and 11.14.

### 3.10.9.2 Guidelines

Structural coverage analysis tools for OO languages should measure coverage for each polymorphic reference and each entry in the dispatch tables. Unfortunately, current tools performing structural coverage of polymorphism with dynamic dispatch only measure the execution of the polymorphic reference, and thereby support only the first part of the recommended coverage analysis. These tools will need to be augmented with other analyses or tools in order to support the second part of the recommendation, coverage of every entry in the dispatch tables. For example, this can be achieved as a result of using the substitutability test guidelines described in Section 3.3 of this handbook (covering every method table entry):

- *Subtyping,*
- *Formal Subtyping,*
- *Unit Level Testing of Substitutability,*
- *System Level Testing of Substitutability Using Assertions,*
- *System Level Testing of Substitutability Using Specialized Test Cases,*

Compliance with these guidelines should be combined with statement coverage (executing every polymorphic call). Additionally, the structural coverage tool may need to be qualified as a software verification tool.

## 3.11 Traceability

### 3.11.1 Purpose

This section addresses issues influencing compliance with DO-178B objectives regarding traceability, when object-oriented development and verification methods and tools are used. As with most of the sections in this handbook, UML terminology is used, since UML is the predominant OO modeling language currently being considered by the aviation community, and other modeling languages present similar issues. In this section, we assume that the software development process starts with a set of system level requirements allocated to software. These requirements can be functional requirements written in text, a requirement model, a set of Use Cases, or even a combination of these approaches.

### 3.11.2 Scope/Background

Current object-oriented (OO) methods and tools (either internally developed or commercially procured) may not satisfy the DO-178B objectives related to traceability. This section of the handbook documents the issues related to traceability when OO methods and tools are used and defines some guidelines for addressing those issues.

Traceability is an important aspect of meeting DO-178B objectives. Traceability is used to:

1. Enable verification of implemented system requirements, high-level requirements, and low-level requirements;
2. Verify the absence of unintended function and/or undocumented source code;
3. Provide visibility to the derived requirements. Traceability applies to both the verification and configuration management processes [2, FAQ #71].

In general, traceability is complicated by:

- Functional requirements specified by Use Cases,
- Dynamic dispatch, polymorphism, and inheritance,
- Overloading and overriding functionality.

### 3.11.3 Overall approach

This section focuses on traceability that provides the evidence of a link between a requirement and its implementation, including the identification of derived requirements. Additionally, the link between UML artifacts and source code is discussed. DO-178B [1] defines traceability as: *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.*

The verification process provides traceability between the implementation of the software requirements and verification of those software requirements:

- The traceability between the software requirements and the test cases is accomplished by the requirements-based coverage analysis.
- The traceability between the code structure and the test cases is accomplished by the structural coverage analysis.

DO-178B guidelines require traceability between system requirements and software requirements to enable verification of the complete implementation of the system requirements. The low-level requirements should be traced to the high-level requirements ensure full implementation of the high-level requirements and to verify the architectural design decisions made during the software design process. In addition, traceability between source code and low-level requirements should be provided to enable verification of the absence of undocumented source code and verification of the complete implementation of the low-level requirements. Traceability from system

requirements to high-level requirements to low-level requirements to code also helps identify derived requirements and ensure they are passed up to the system safety assessment process.

### *3.11.3.1 Related DO-178B Sections and Objectives*

The following DO-178B objectives are directly related to traceability:

- Objective 6 of Table A-3;
- Objective 6 of Table A-4;
- Objective 5 of Table A-5;
- Objectives 3 through 8 of Table A-7;
- Objective 2 of Table A-8.

Other objectives within DO-178B are indirectly related to traceability. The following OOT issues specifically make it difficult to comply with DO-178B's objectives. Each issue is discussed with recommended guidelines to address the issues.

## **3.11.4 Tracing to Functional Requirements**

Traceability of functional requirements through implementation may be lost or difficult with an object-oriented design or life cycle that supports OOT. A mismatch between function-oriented requirements and an object-oriented implementation may cause traceability problems. For example, providing traceability from a code sequence to a specific requirement may be difficult. Tracing to a "logical view" may not be sufficient.

### *3.11.4.1 Guidelines*

The software development process usually starts with a set of system level requirements allocated to software. These requirements can be documented using text, requirement model or Use Cases. In UML, Use Cases represent the functional requirements of the system. These functional requirements can be traced to the system level requirements that can be documented in text. The software functional requirements will be decomposed to create the low level requirements and any derived requirements necessary for the implementation of the software. Every method or a message reception in every object in the software will have to be traced to its requirement.

Figure 3.11-1 shows a traceability model that can be used for OO based system. There are few ways to specify requirements, the user may chose to specify requirements in text, in use cases, or create a requirements model. Such a requirement model may be represented by a combination of requirements level (customer validated) Use Cases in addition to a class model of the system. The user may specify the requirement for a class Statecharts. In this case, the transition and the states of each Statechart could become the low-level requirements.

Whatever method the user use to specify the requirements they need to be clearly tagged. The diagram above shows that design, implementation, and testing elements are traced to the requirements.

In the design stage, instances are traced to classes. Classes may have relations and will have to be traced to each other. If class interfaces are specified as contracts (with pre/post conditions for operations and class invariants), then tools can be used to help trace the relationship between classes and between operations and check them for consistency and correctness. Each operation or a message provided by the instance will be traced to the instance. The implementation files will be traced to the instances.

The test model will be traced to the requirements. This paper assumed that test classes will not have relations and each test class will have one instance.

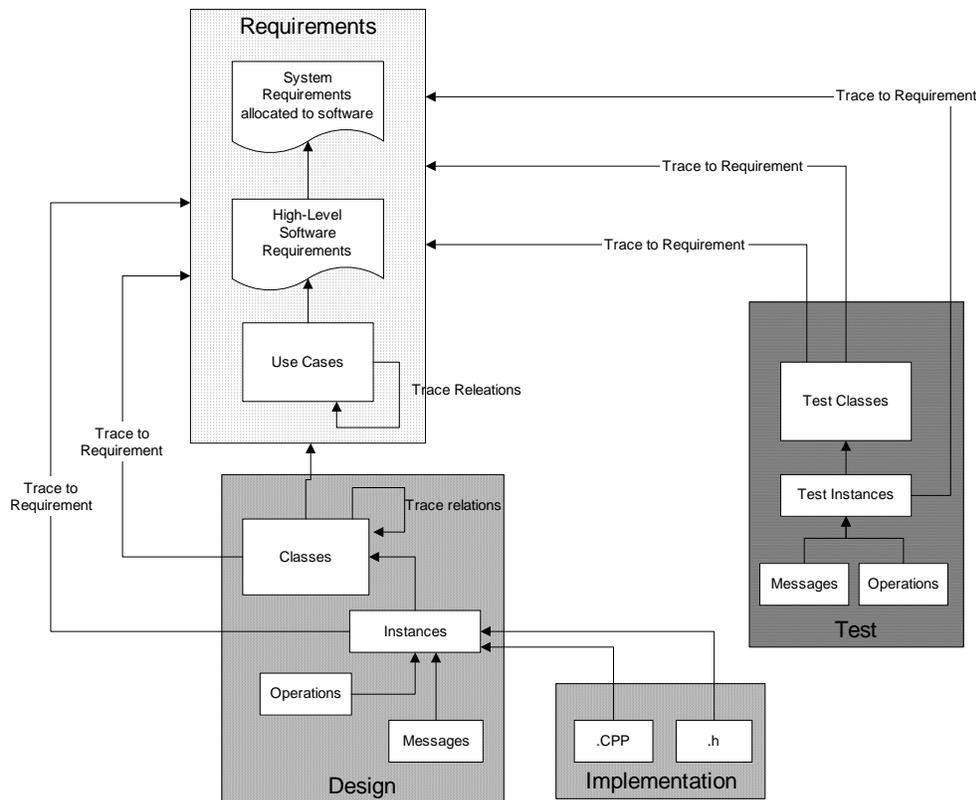


Figure 3.11-1 Overview of Traceability

### 3.11.5 Complex Class Hierarchies and Relationships

Class hierarchies can become overly complex, which complicates traceability. Generalization, weak aggregation, strong aggregation, association and composition are some of the relations that can be used to create the class diagrams.

#### 3.11.5.1 Guidelines

The realization of a Use Case may be specified by a set of collaborations. The collaborations define how instances in the software interact to perform the sequences of the Use Case. Traceability should be done at the instance level. When the user creates an instance, they must know what that instance is traced to and where its requirements are coming from. The instances themselves must be traced to their classes. The class and all of its contents and relations should be traced to a higher-level class or to a Use Case. Since the high level classes are the realization of Use Cases, which is the functional requirements of the software, the high level classes should be traced to the Use Cases that they implement. The Use Case must be traced to a system level requirement or a higher-level Use Case. Not all of the UML diagrams must be traced; tracing all of the objects in the software should be sufficient. Only the diagrams and UML modeling elements that add some requirements or affect the generation of the executable should be traced. Class hierarchies and their relations need to be flattened and every class should have clear traceability to its high level class, Use Case, or text requirements.

The following relations affect traceability in the following ways:

- **Association** is the semantic relationship between two or more classes that specifies connections among their instances. In this case, there is no need to trace associations, since they only indicate that one class can talk to the

other. Each class should have its own traceability to its requirements. Association classes are treated as regular classes and they will have to be traced to the requirements that caused their existence.

- **Aggregation** is a special form of association that specifies the whole-part relationship between the aggregate (whole) and a component (part). Aggregation will affect traceability, because the instance of the aggregate (whole) will have to be traced to both the whole and the part requirements. The part will only have to be traced to its own requirements
- **Generalization** is the relationship between a more general element and a more specific element. The specific element is fully consistent with the more general element. An instance of the more specific element may be used where the more general element is allowed. Generalization will affect traceability, because any instance of the specific element should be traced to its own requirements. Additionally, it should be traced to the requirements of the general element from which it is inherited.

Normally traceability is limited to the UML elements and does not include diagrams.

### **3.11.6 OO Design Notation and Traceability Ambiguity**

When working with UML or OOT in general, the requirements, design, and implementation may have multiple views. Each view may add or show different information. Unfortunately, many of the UML tools do not currently provide a traceability mechanism. Additionally, UML is a language that was written to provide the user with maximum flexibility, which in the safety-critical world might reduce controllability.

#### **3.11.6.1 Guidelines**

Each UML element (e.g., class, method, object, Use Case) should have traceability, but not every diagram containing those elements needs to be traced. For example, an object model diagram may not need to be traced, but every object in the diagram should be traced to its own requirements. The developer should have a process that enforces the guidelines for traceability, some of the basic guidelines include:

- Every object in the software should be traced to its class.
- Every class in the software should be traced to its super class.
- Every function call or an message should be traced to its class.
- Each overridden or overloaded operation should be traced to some requirement(s) or Use Case (s).
- Every class or super class should be traced to the Use Case (s) that they realize.

### **3.11.7 Traceability and Dynamic Binding/Overriding**

Establishing functional requirements coverage of a class is difficult to assess given dynamic binding and overriding. Specifically, it may be difficult to know if a class has been fully exercised.

#### **3.11.7.1 Guidelines**

Traceability must be performed to the object level. Each function in an object needs to be traced. This will provide traceability to the implementation of any virtual functions or functions that have been overridden. Component based design or Design-by-contract approaches may be used.

### **3.11.8 Dead and Deactivated Code**

The difference between dead and deactivated code is not always clear when using OOT. Without good traceability, identifying dead versus deactivated code may be difficult or impossible.

#### **3.11.8.1 Guidelines**

If the OOT traceability is done to the function or event (message) level, then it will be possible to identify dead code at the function level. This will not cover code within a function. OO concepts encourage building reusable classes.

The idea of reuse means that classes are built with generic functional requirements that can be used in multiple systems. In this case, the reusable library may include some deactivated or dead code in a specific application. Traceability analysis should be performed on the reusable library in order to identify dead and deactivated code. Dead and deactivated code is further addressed in section 3.9 of this handbook.

### **3.11.9 *Many to Many Mapping of Requirements to Methods***

The isolation of functions into classes may result in a mapping of requirements to OOT models in which:

- a) A given requirement may map to a number of functions spread over several classes;
- b) The same function, in a given class, may contribute to more than one requirement.

This issue applies to the mapping of requirements to methods at all levels of OO modeling (during both analysis and design).

#### **3.11.9.1 *Guidelines***

Each function within an instance should be traced to its requirements. The function can have more than one requirement, but it should not have more than one Use Case. This is the normal case for a Use Case driven development approach but it is a good practice to reduce the complexity of traceability even if another approach is used. In this case, we may have one Use Case traced to many functions. The traceability data for each Use Case should contain the references to all of the functions within the objects that implement that Uses Case.

If it is necessary to trace one function to multiple Use Cases, then Use Case relationships may be used to help eliminate redundancy and support traceability.

### **3.11.10 *Iterative Development***

Iterative development is often desired in OO implementation. Each iterative cycle has its own requirements (normally a set of Use Cases), design, implementation, and test. There is a risk of losing traceability when using iterative development. This can be caused by adding or changing requirements, design, or implementations.

#### **3.11.10.1 *Guidelines***

Iterative development is most effective when each iterative cycle is completed prior to starting the next cycle. The iteration should have its own requirements, design, implementation, and test. Completeing traceability for each iterative cycle is recommended, and an impact analysis should be done on the current iteration, whenever the requirements from the previous iterations are changed.

### **3.11.11 *Change Management for Reusable Components***

Reusability is one of the objectives of OO development, but reusable components may be hard to trace because they are designed to support multiple usages of the same component. Reusable components may also have functionality that may not be used in every application.

#### **3.11.11.1 *Guidelines***

Traceability must be done for each application regardless of its usage of reusable components. When reusable components are used, the traceability should show implementation and verification mapping for all requirements, or as a minimum, provide verification and justification for the unused functionality. See section 3.9 of this handbook for further guidelines on reuse.

## 3.12 Structural Coverage

### 3.12.1 Purpose

This section addresses issues concerning compliance with the structural coverage objectives given in DO-178B [1] and clarified in DO-248B [2] for certain features of object-oriented technology.

### 3.12.2 Background

Structural coverage as one of the adequacy measures of requirements-based testing will not go away with OOT. Most of what has been learned in the testing of traditional (functional) systems still applies within OOT. However, traditional functional testing with traditional structural coverage metrics based on source code may not be adequate for object-oriented software [25][33][34]. In particular, inheritance and polymorphism with dynamic dispatch are two OOT mechanisms that present problems with verification in general, and testing and structural coverage in particular [25][34][19].

The FAA has already sponsored some research in this area [26]. Not only were issues identified for inheritance and polymorphism with dynamic dispatch, but the broader issue of data coupling and control coupling was also raised.

### 3.12.3 Overall approach

This section is intended to provide an approach for addressing DO-178B [2, FAQ #71] objectives for structural coverage when using the OO features of inheritance and polymorphism with dynamic dispatch. The broader issue of data coupling and control coupling is also mentioned as a place-holder. The section is related to DO-178B [1] sections 6.4.4.2 and 6.4.4.3; and also related to sections 6.3.6, 11.3c(2), d and g, 11.13, and 11.14, Table A-7, objectives 5, 6, 7 and 8.

### 3.12.4 Structural Coverage of Inheritance

The issue concerning the adequate verification of inheritance is whether fully verified inherited features, particularly methods, need re-verification in the subclass. It turns out that some, but not all, inherited features require re-testing (and coverage) within the subclass [27][28, ch. 7, pp. 249-267]][29, ch. 11, pp. 164-213]. The Hierarchical Incremental Testing (HIT) of class structures approach was developed to identify where retesting of inherited features was necessary, and where it was not [27]. Table 3.12-1 presents a summary of HIT. The first column of Table 3.12-1 identifies the incremental change made in the subclass. There are three changes made in subclasses that impact the testing performed on the superclass:

- A method can be inherited. If there are no direct or indirect polymorphic references affecting this method, then it remains unchanged in the purest sense, and no retesting is needed. If there are polymorphic references affecting the method, then the method can be considered to have a change in integration, and some retesting will be necessary.
- A method can be overridden. This is the case when a new method is provided in the subclass for an existing method in the superclass.
- A new method can be added. This includes either the addition of a new operation and method in the subclass, or the addition of a method for an abstract operation in the superclass.

In addition, new attributes can be added in the subclass. The impact of these new attributes is through the new or overridden methods that access those attributes. The second column of Table 3.12-1 identifies the testing impact in the subclass for the incremental change.

Incremental Change	Testing Impact
Inherited (unchanged) Method	No retesting needed if the method interacts, directly or indirectly, only with inherited methods and attributes. Limited retesting needed when the method interacts, directly or indirectly, with new or overridden methods or attributes. Existing tests that deal only with inherited methods and attributes are still applicable, along with the coverage analysis for those tests. Some existing tests can be reused, though new coverage analysis will be needed for those tests. Some new tests will be needed, along with the coverage analysis for those tests.
Overridden (changed) Method	Extensive retesting and coverage analysis is needed. Many existing tests can be reused, though new coverage analysis will be needed for those tests. For most of these tests, the requirements coverage should still be applicable since (only) the implementation has changed. Some new tests will be needed, along with the coverage analysis for those tests.
New Method	Complete testing and coverage analysis is needed. All new tests will be needed, along with the coverage analysis for those tests.

Table 3.12-1 Hierarchical Incremental Testing Summary

One of the techniques introduced for the understanding of inheritance is that of the flattened class. In a flattened class, all inherited features are represented along with the features explicitly defined within the subclass. Figure 3.12-1 illustrates a normal inheritance hierarchy on the left, and a flattened inheritance hierarchy on the right. In the flattened form, all inherited attributes and methods appear in italics.

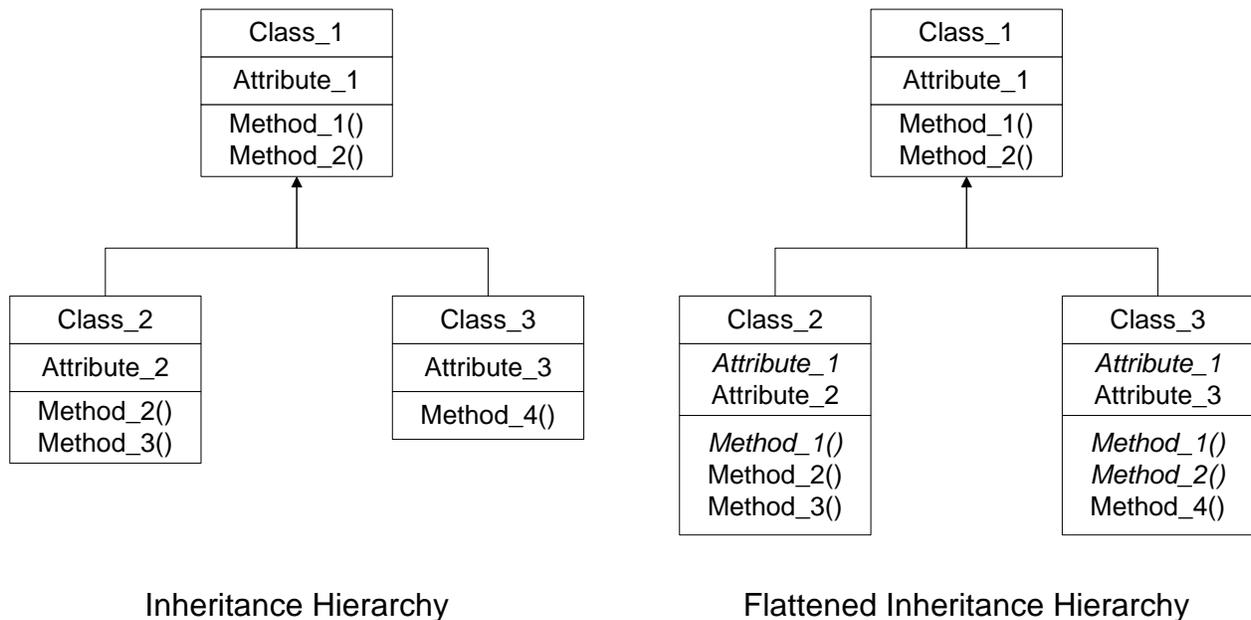


Figure 3.12-1 Inheritance

### 3.12.4.1 Guidelines

Requirements-based testing and requirements and structural coverage of the flattened class is a recommended practice for OO [11, ch. 10.5]. A number of researchers recommend class flattening as they believe the savings from

trying to save tests will be negated by the effort of the analysis, plus the circumstances where savings are possible will be very rare in real systems. Binder states “Retesting can be safely skipped only when (1) there is no possible data flow or control flow from or to the superclass and subclass methods, or (2) the subclass is null and simply renames the superclass.” [11, p. 510]. The difference between covering the class, and covering the flattened class for Method\_1 from Figure 3.12-1 is demonstrated in Figure 3.12-2 and Figure 3.12-3.

In Figure 3.12-2, complete coverage is shown for Method\_1 as a composite of the coverages obtained within Class\_1, Class\_2 and Class\_3. This is referred to as coverage of concrete features.

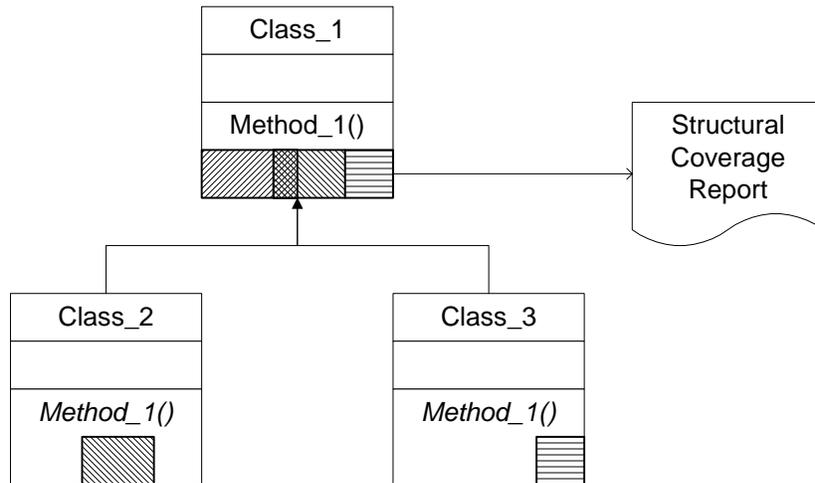


Figure 3.12-2 Concrete Coverage

In Figure 3.12-3, only partial coverage is shown for Method\_1 in the three different classes. This is referred to as context coverage.

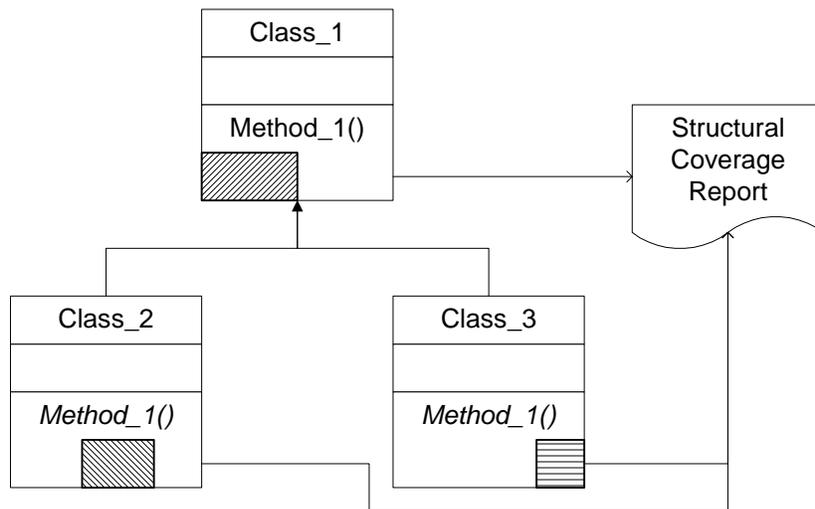


Figure 3.12-3 Context Coverage

DO-178B does not require the context sensitive verification required by flattened classes. Always flattening the class hierarchy for verification, recommended in [26], will lead to some over-verification of certain inherited features (e.g., simple get and set methods for attributes). Fortunately, HIT can be used to prevent the over-

verification by identifying those methods that do not need any re-verification. The HIT analysis could be automated, incorporated into structural coverage analysis tools and qualified.

### 3.12.5 Polymorphism with Dynamic Dispatch

Polymorphism with dynamic dispatch is a mechanism within OOT whereby a name can refer to objects of different classes. The issue concerning the adequate verification of polymorphism with dynamic dispatch is whether the method with the polymorphic reference has been adequately integrated with all of the methods the polymorphic reference can dispatch to. Polymorphism with dynamic dispatch is illustrated in Figure 3.12-4 and Figure 3.12-5. Figure 3.12-4 presents the flattened class from Figure 3.12-1. Recall that in a flattened class, all inherited features are italicized.

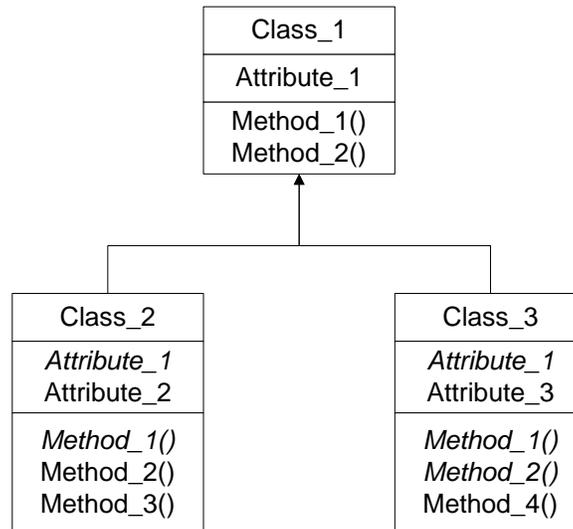


Figure 3.12-4 Flattened Inheritance

Assume that in our system we have an “Object\_X” that can refer to objects of either Class\_1, Class\_2 or Class\_3. When we see the call to “Object\_X.Method\_2()” in the source code for our system, different Method\_2’s will be called depending on the run-time class of the object that “Object\_X” refers to. This is depicted graphically in Figure 3.12-5.

Figure 3.12-5 shows that the reference “Object\_X.Method\_2()” can dispatch to either Class\_1.Method\_2() or Class\_2.Method\_2(). Class\_1.Method\_2() will be called if “Object\_X” refers to an object of either Class\_1 or Class\_3. This is because Class\_1.Method\_2() was defined in Class\_1 and inherited in Class\_3. Class\_2.Method\_2() will be called if “Object\_X” refers to an object of Class\_2. This is because Class\_2.Method\_2() was defined in Class\_2, where it overrode the definition of Class\_1.Method\_2(). Polymorphism with dynamic dispatch has made some of the control flow, and thereby the associated data flow, implicit in the source code rather than explicit.

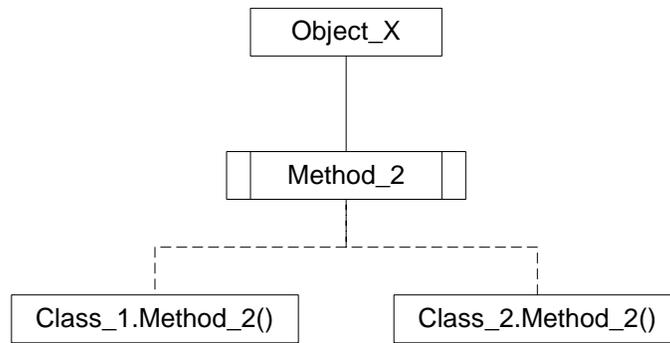


Figure 3.12-5 Dynamic Dispatch

### 3.12.5.1 Guidelines

Numerous approaches have been proposed for the adequate testing of polymorphic references:

1. The first approach says that execution of the polymorphic reference is sufficient. This approach confines testing to the level of abstraction of the source code (i.e., every statement has been executed). This approach assumes that the underlying implementation hides no details requiring verification below the abstraction level of the source code. This approach treats dynamic dispatch as being equivalent to static dispatch. Watson and McCabe refer to this as the “optimistic approach” [30, pp. 62-63]. DO-178B currently does not require verification beyond the level of the source code for software at Levels C and B. DO-178B does require verification beyond the source code for software at Level A.
2. The second approach says to treat a polymorphic reference as a case/switch statement [31]. This approach assumes that the underlying implementation is important. This approach can be implemented in two major ways. The first way is to recognize that the “case/switch statement” is repeated in multiple places. Testing and coverage could be obtained from the collective executions of all the references (inlined call to the case/switch statement). The second way is to exhaustively test and cover the feasible “branches” at each reference, which makes this approach equivalent to the next one.
3. The third approach says that execution of every possible dispatch is required. Binder states “Although a polymorphic message is a single statement, it is an interface to many different methods. Just as we would not have high confidence in code for which a only a small fraction of the statements or branches had been exercised, high confidence is not warranted for a client of a polymorphic server unless all the message dispatches generated by the client are exercised” [29, p. 438]. Watson and McCabe refer to this as the “pessimistic approach” [30, p. 63]. This approach can also be implemented in two ways. The first way is to look at polymorphic messages only and ignore polymorphic parameters. The second way is to consider the polymorphic parameters. This brings in a level of complexity that the next approach attempts to deal with.
4. The fourth approach says that execution of a “mathematically significant” subset of all possible dispatches is required[32]. This approach attempts to mitigate the enormous number of tests and required effort that can result from the exhaustive execution of all possible dispatches.
5. The fifth and final approach says that we need to execute every polymorphic reference, and for one reference site that forms an equivalence class with all others dispatching on the same base class, require execution of every possible dispatch. Note that the exhaustive dispatching can be done in a test driver instead of the application. This is another attempt to mitigate the exhaustive approach. Watson and McCabe refer to this as the “balanced approach” [30, pp. 63-64].

As the previous discussion has made clear, the proper verification of polymorphism and dynamic dispatch is still an active research area. Until a final answer is in, a minimal requirement should be the execution of all polymorphic references, and the execution of all possible dispatches collectively (i.e., coverage of all entries in the dispatch table) [21][26].

Current tools performing structural coverage of polymorphism with dynamic dispatch only measure the execution of the polymorphic reference, and thereby support the first approach discussed previously. These tools support the first part of the recommended approach. These tools will need to be augmented with other analyses or tools in order to support the second part of the recommendation.

### ***3.12.6 Data Coupling and Control Coupling***

Data coupling and control coupling relationships can be far more complicated and obscure in OOT than they are in traditional (functional) systems/software.

One impact on data coupling and control coupling is in the nature of OOT. OOT encourages the development of many small, simple methods to perform the services provided by a class. Often the control flow is moved out of the source code through the use of polymorphism and dynamic dispatch. In essence, the control flow, and thereby the control coupling, will become implicit in the source code, as opposed to being explicit. There is a similar effect on the data flow, and thereby the data coupling.

OOT also encourages hiding the details of the data representation (i.e., attributes) behind an abstract class interface. Suggested “best practice” is that attributes of an object should be private, and access to them only provided through the methods appropriate to the class of the object. Being able to access attributes only through methods makes the interaction between two or more objects implicit in the code.

Some simple examples demonstrating how complex things can get are given in[35].

#### ***3.12.6.1 Guidelines***

This is an active area of research, both within academia and the FAA. At this point in time, this handbook can give no guidance on dealing with data coupling and control coupling within OOT.

### 3.13 References

1. Software Considerations in Airborne Systems and Equipment Certification, Document No. RTCA/DO-178B, RTCA Inc., 1828 L Street, Northwest, Suite 805, Washington, DC 20036.
2. DO-248B, Final Report For Clarification of DO-178B, Software Considerations in Airborne Systems and Equipment Certification, 10-12-01.
3. Daugherty, Gary. Application of the Subtyping Pattern to System Level Test, Rockwell Collins technical report, April 2002, available from the author (gwdaughe@rockwellcollins.com).
4. Object Management Group. OMG Unified Modeling Language Specification, version 1.3, June 1999, available from <http://uml.shl.com/artifacts.htm>
5. Gosling, James et al. The Java Language Specification, Addison-Wesley, 1996.
6. Lindholm, Tim and Frank Yellin. The Java Virtual Machine Specification: Second Edition, Addison-Wesley, 1999.
7. Liskov, Barbara and Jeanette Wing. "A Behavioral Notion of Subtyping", *ACM Transactions on Programming Languages and Systems*, 16(6): 1811-1841, November 1994.
8. Liskov, Barbara with John Guttag. Program Development in Java: Abstraction, Specification, and Object-Oriented Design, Addison-Wesley, ISBN: 0201657686, 2001.
9. Lorentz, Mark and Jeff Kidd. Object-Oriented Software Metrics, Prentice-Hall, Englewood Cliffs, NJ, ISBN: 0-13-179292-X, 1994.
10. Barnes, John. Programming in Ada95, 2nd edition, Addison-Wesley, 1998.
11. Binder, Robert V. Testing Object-Oriented Systems: Models, Patterns, and Tools, Addison-Wesley, Reading, MA, 2000.
12. Bruce, Kim, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens and Benjamin Pierce. On Binary Methods, Iowa State University, technical report #95-08a, December 1995.
13. Castagna, Giuseppe. Object-Oriented Programming: A Unified Foundation, Birkauer, Boston, ISBN: 0-8176-3905-5, 1997.
14. Clifton, Curtis, Gary T. Leavens, Craig Chambers, and Todd Millstein. "MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java", *OOPSLA 2000 Conference Proceedings: ACM SIGPLAN Notices*, vol. 35, no. 10, October 2000, pp. 130-145.
15. Johnston, Simon. Ada95 for C and C++ Programmers, Addison-Wesley, 1997.
16. Meyers, Scott. Effective C++, 2nd edition, Addison-Wesley, Reading, MA, 1998.
17. Meyer, Bertrand. "Applying design by contract." *IEEE Computer* 25(10):40-51, October 1992.
18. Meyer, Bertrand. Object-oriented Software Construction, 2nd edition, Prentice-Hall PTR, Upper Saddle River, NJ, 1997.
19. Jeff Offutt, Roger Alexander, Ye Wu, Quansheng Xiao, and Chuck Hutchinson. "A Fault Model for Subtype Inheritance and Polymorphism", *Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE '01)*, IEEE Computer Society Press, November 2001, pp. 84-93.
20. FOLDOC: Free Online Dictionary of Computing, <http://foldoc.doc.ic.ac.uk>.

21. AVSI. Guide to the Certification of Systems with Embedded Object-Oriented Software, available from Boeing, Rockwell Collins, Honeywell, and Goodrich, or directly from the Aerospace Vehicle Systems Institute (AVSI).
22. Stroustrup, Bjarne. The Design and Evolution of C++, Addison-Wesley, 1994.
23. Guide for the Use of the Ada Programming Language in High Integrity Systems, ISO/IEC PDTR 15942, July 1, 1999, see <http://anubis.dkuug.dk/JTC1/SC22/WG9/documents.htm>
24. FAA Order 8110.49, Software Approval Guidelines, Chapter 4, June 3, 2003.
25. Perry, Dewayne E. and Gail E. Kaiser. “Adequate Testing and Object-Oriented Programming”, *Journal of Object-Oriented Programming*, January/February 1990
26. Chilenski, John Joseph, Thomas C. Timberlake and John M. Masalskis, Issues Concerning the Structural Coverage of Object-Oriented Software, DOT/FAA/AR-02/113, November 2002
27. Harrold, Mary Jean, John D. McGregor and Kevin J. Fitzpatrick. “Incremental Testing of Object-Oriented Class Structures”, *Proceedings of the 14<sup>th</sup> International Conference on Software Engineering*, 1992
28. McGregor, John D. and David A. Sykes. A Practical Guide to Testing Object-Oriented Software, Addison-Wesley, 2001
29. Siegel, Shel. Object Oriented Software Testing : A Hierarchical Approach, John Wiley & Sons, 1996
30. Watson, Arthur H., Thomas J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric, NIST Special Publication 500-235, 1996
31. Jacobson, Ivar, Magnus Christenerson, Patrik Jonsson and Gunnar Overgaard. Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, 1992
32. McDaniel, Robert and John D. McGregor. Testing the Polymorphic Interactions between Classes, TR94-103, Clemson University, 1994
33. Booch, Grady. *Object-Oriented Analysis and Design with Applications*, 2<sup>nd</sup> edition, Benjamin/ Cummings, Redwood City, CA, 1994.
34. FAA AC 20-RSC, Reusable Software Components, Draft (\*\*\*) will update when AC is signed)
35. *Dictionary of Computer Science, Engineering, and Technology*, Editor-in-Chief Phillip A. Laplante, CRC Press LLC, Boca Raton, Florida, 2001.

### 3.14 Index of Terms

abstraction	26, 27, 33	IEEE	67
Ada	68, 75	implementation inheritance	29
Aerospace Vehicle Systems Institute	29	inlining	26
<u>Aggregation</u>	59	Inlining	39
<u>Association</u>	58	instrumented	22, 23
AVSI	29, 68	interface inheritance	28, 29, 30, 31, 79, 82, 84
C#	17, 29	Java	12, 23, 29, 67, 72, 74, 75, 76, 79, 82, 84
C++	12, 23, 32, 67, 68, 72, 75, 76, 79, 82, 84, 85	LSP6	7, 13, 14, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 55
class	12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 26, 27, 28, 29, 31, 32, 33, 34, 35, 46, 47, 48, 49, 50, 53, 72, 73, 74, 75, 76, 78, 84, 86, 87	MC/DC	22
concrete class	73	MultiJava	67
constructors	12, 14, 16, 18, 46, 49, 52, 74, 75, 76	multiple inheritance	12, 27, 28, 29, 31, 33, 34, 48, 76, 78
control coupling	25, 38, 51	Multiple inheritance	29, 32
coupling	14, 27	Multiple Inheritance	29, 32, 78
data coupling	25, 26, 48, 51	Object Management Group	67
deactivated code	46, 47, 48, 49, 50, 51	OMG	67
Dead and Deactivated Code, and Reuse	46, 86	OOT	46, 47, 49, 51
dead code	46, 47, 48, 51, 86	option-selectable	46, 47, 48
derived requirements	22, 24, 47, 48, 50	Overloading	44
destructors	12, 17, 46, 49, 52, 76	pattern	15, 19, 22, 23, 24, 33, 76, 84
DO-178B	13, 14, 18, 24, 25, 26, 29, 36, 37, 38, 39, 40, 41, 42, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 57, 67	<i>polymorphism</i>	13, 16, 17, 51
dynamic binding	12, 54	postcondition	12, 13, 17, 18, 19, 20, 23, 27, 82
dynamic dispatch	12, 13, 14, 15, 16, 17, 21, 24, 25, 29, 30, 31, 51, 54, 74, 75, 76	precondition	12, 13, 16, 17, 18, 19, 20, 23, 27, 82
<i>Dynamic dispatch</i>	13, 27, 75, 76	<i>rule3</i>	4, 5, 6, 7, 8, 9, 10, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 30, 31, 32, 34, 35, 47, 48, 49, 50, 74, 75, 76, 82, 84, 85
Dynamic Dispatch	12, 54	run-time class	13, 17
dynamic linking	13	service history	50
Eiffel	12, 14, 23, 29	single inheritance	12, 13, 14, 29, 30, 31, 74, 75, 76
extension	14, 17, 18, 29, 75	Single inheritance	12, 29
flattened form	17, 21, 49	Single Inheritance	12, 72
frameworks	46, 51, 52	software change impact analysis	49
<u>Generalization</u>	58	<i>Software Considerations in Airborne Systems and Equipment Certification</i>	67
hierarchy	12, 27, 28, 30, 32, 33, 34, 35, 49, 50, 75, 78, 82, 87	spaghetti inheritance	27, 34
		structural coverage	21, 22, 23, 25, 48, 51, 54

subclass	12, 13, 14, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 30, 31, 32, 34, 49, 76, 85	traceability	19, 20, 41, 47, 48, 51, 56, 57, 58, 59, 60
subinterface	30, 78, 79, 82, 83	Traceability	48, 51, 56
subtyping	12, 13, 18, 19, 29, 31, 84	Type Conversion	41
superclass	12, 13, 14, 17, 18, 19, 20, 21, 22, 23, 24, 26, 27, 29, 30, 31, 32, 33, 34, 74, 76, 85	UML	12, 16, 19, 51, 52, 53, 84
System Safety Assessment	47	Unified Modeling Language	12, 67
Templates	36	uninstrumented	22, 23, 24
Tools	48, 51, 52, 54, 67	Use Cases	56, 57, 58, 60
		visual modeling	52
		visual models	51, 52. <i>See</i> visual modeling

## Appendix A Frequently asked questions (FAQs)

### **Does DO-178B require compliance with the subtyping rules for substitutability (LSP)?**

No. However, the structural coverage criteria must still be met, and the issues raised in volume 2 must still be dealt with in some manner. Compliance with the subtyping rules is just one way to do so. And is compatible with the definition of the Generalization relationship by UML (below).

### **Does UML require compliance with the subtyping rules for substitutability (LSP)?**

UML does not directly mention the Liskov Substitution Principle (LSP) or the rules given in this handbook. UML, however, does say that the *Generalization* (subtyping) relationship implies substitutability, which indirectly implies compliance with LSP.

### **When is it necessary to verify subtyping relationships?**

At a minimum, we want to ensure substitutability when instances of different subclasses may be assigned to a variable at run time in a given system (i.e., when polymorphic assignment is actually used).

Technically, we need not be worried about cases in which a given variable is assigned instances of different subclasses, only in different instances of the system (e.g., as a means of parameterizing the system's behavior).

Still, given that the UML definition of subtyping implies substitutability, it is also technically an error to use this relationship in UML models where this is not the case.

And it is, in general, a "good idea" to verify subtyping relationships from the outset, not just when we discover substitution actually occurs within a given system.

### **Is it necessary to use unit level testing to verify subtyping relationships?**

No. System level testing, static analysis, and proofs may also be used to ensure (or help ensure) that the subtyping rules are followed.

### **Is it necessary to use UML in order to follow the guidelines?**

No. The use of UML terminology and UML examples is only a convenience. The principles underlying the guidelines apply to other OO methodologies and modeling notations as well.

### **Is it necessary to use special OO coverage tools to measure structural coverage of dynamic dispatch?**

No. By following an appropriate test process, it is possible to ensure that all entries of all dispatch tables are exercised. The use of tools to measure this type of coverage, however, is still encouraged.

### **What role should static analysis play in the verification of OO systems?**

The guidelines have been developed to restrict the use of OO in such a way as to encourage the use of various forms of static analysis. The AVSI Guide [21], which was the basis for the first version of the Handbook, specifically addresses how various guidelines affect the forms of static analysis recommended by [23], and required by DO-178B.

### **How do the guidelines map to the DO-178B software levels?**

The original AVSI Guide provided a mapping of guidelines to DO-178B software levels. This mapping was based primarily on a consideration of which restrictions on OO features were necessary to enable the use of the types of analysis techniques required by DO-178B at each software level, and secondarily on the error prone nature of certain features (such as multiple implementation inheritance). This table does not appear in the publically released version of the Guide. A similar mapping based on similar criteria, however, may appear in a future version of this handbook.

**How about the use of “work arounds” as a substitute for the direct use of OO features?**

The guidelines are not intended to give a “free pass” to the use of OO-like work arounds as a substitute for the direct use of OO features. For example, the intent is not to favor the hand coding of dispatch routines (containing explicit case statements) over the use of dynamic dispatch. By drawing an analogy between the two, the guidelines impose the same coverage criteria for both.

Similarly, issues related to substitutability must be addressed by any system that permits the “plugging in” (substitution) of one piece of software in place of another (either at run time or in different versions of the system intended for different customers), whether the system uses OO or not. The OO subtyping guidelines are only a plus in this regard. If you intend to allow substitution (of either form) in an OO system, following the guidelines provides a well trod path for doing so. Something comparable, and not addressed by this Handbook, would still be required if a different approach were taken.

## Appendix B Extended guidelines and examples

### B.1 Single Inheritance

#### B.1.1 Extension of the Inheritance with Overriding Guidelines

The following sections extend the definition of the *Inheritance with Overriding* to include:

- an avionics related example,
- a description of the general structure of the problem and the roles of the participants,
- a mapping of the general guidelines it offers to language specific guidelines for Java, Ada95, and C++.

##### B.1.1.1 Examples

Consider an avionics display system that defines a class DisplayElement and its subclasses (Figure B.1-1).

A given display is composed by drawing its associated elements. In order to draw the pilot's attention to critical information, we highlight specific elements while hiding others. The specifics of drawing an element, hiding it and highlighting it vary according to the type of element. To allow variations on the display for specific customers and aircraft, and to minimize the impact of future changes, the overall application should deal with elements only abstractly (e.g., in terms of the kind of information they display and not how they display it).

The code to draw the display is then:

```
for each display element
    call the draw method associated with its run time class
end
```

The code to highlight a selected set of elements is then:

```
for each display element
    call the highlight method associated with its run time class
end
```

The code to declutter the display is then:

```
for each display element
    compare the importance of the element to a cutoff value for the importance of elements to be displayed
    if the element is not important enough
        call the hide method associated with its run time class
    end
end
```

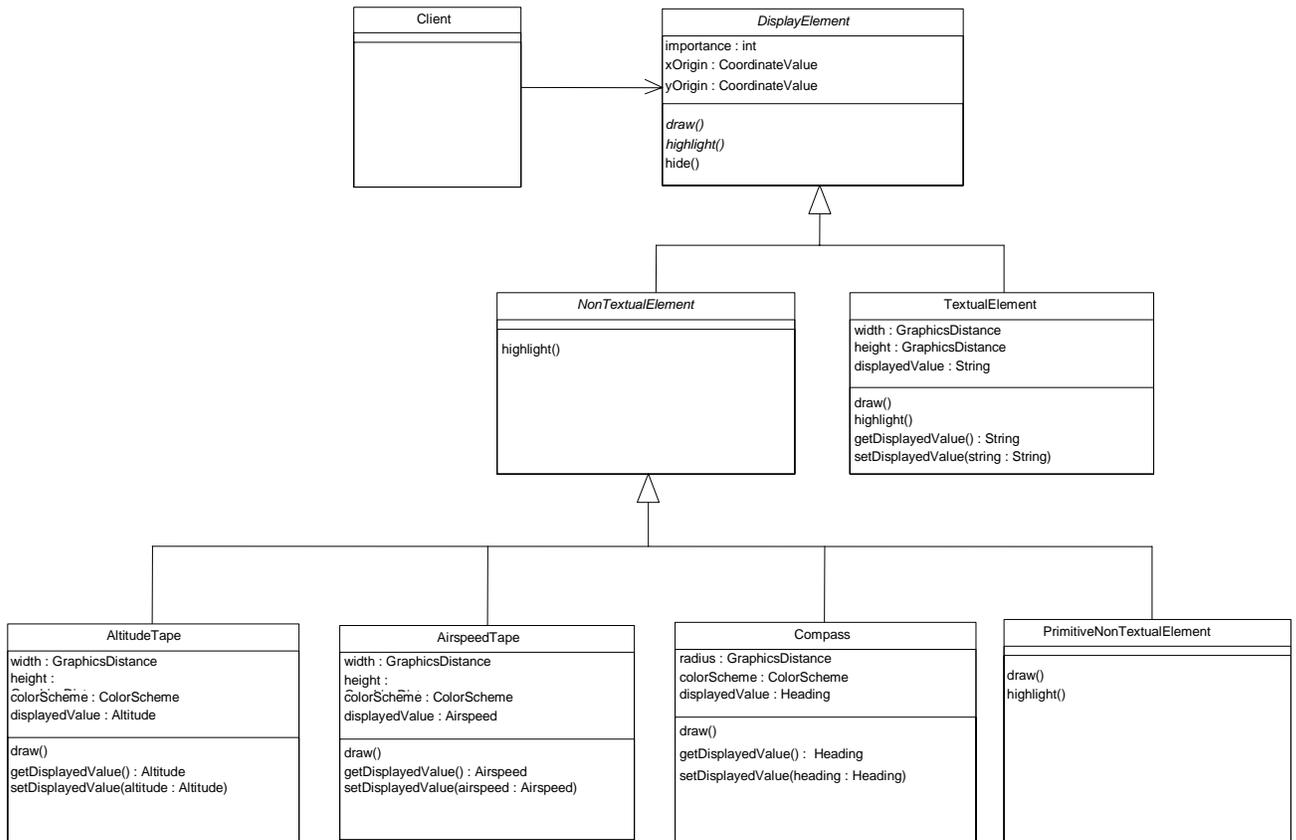


Figure B.1-1 Class Hierarchy

**B.1.1.2 Structure**

In general, a client method calls an operation associated with a target object. The client method may be associated with any object, including the target object itself. The run time class of the target object is a concrete class which may have superclasses and subclasses. The method associated with the run time class of the object is determined at compile time using the simple guidelines for specialization and overriding.

Polymorphically the client method may view the target object at run time as an instance of its run time class or, more abstractly, as an instance of any of its superclasses. In all circumstances, however, it is the method associated with the run time class of the target object that is executed when the operation is called.

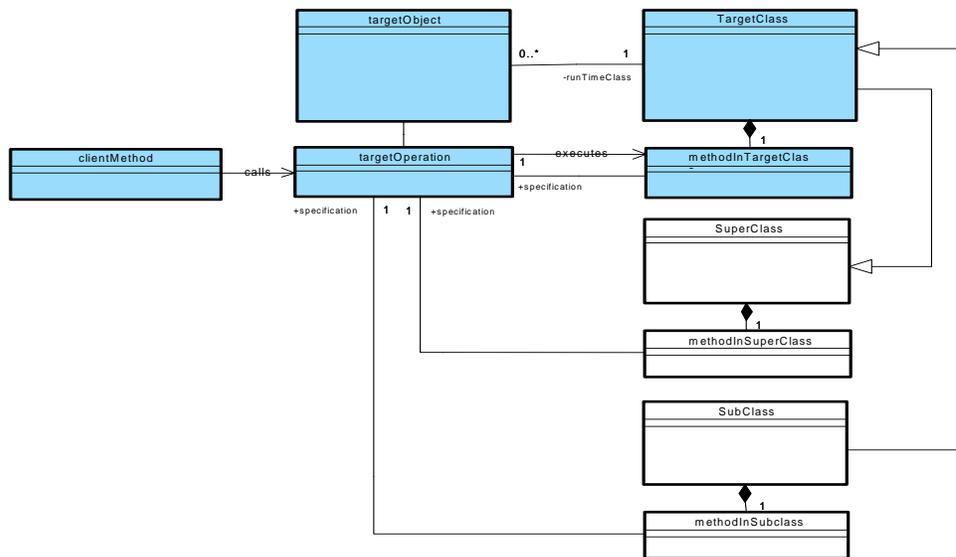


Figure B.1-2 Class Relationships

### B.1.1.3 Participants

The client method calls the target operation on the target object, which executes the method associated with it by the run time class of the object.

### B.1.1.4 Java guidelines

Java is a strongly typed language. It provides dynamic dispatch based on the target object of a method call. The dynamic loading of classes is supported but can be restricted (by eliminating the class loader from the run time environment). Only single inheritance of implementation is permitted. The run time type of an object is assigned at the point at which it is created and cannot be changed during the object's life time.

The *Simple overriding rule*: is enforced by the language if the overriding of concrete methods by abstract methods [5, p. 159, section 8.4.3] is disallowed, and if all explicitly thrown exceptions are checked exceptions (in the Java sense). When exception handling is not used, code reviews should be used to enforce the more general guideline that an overridden version of a method can only report either the same errors, or a more restricted set of errors than its parent version.

The use of the keyword `super` in a call expression can be used to violate the *Simple dispatch rule*:. As a result, the use of the keyword `super` should only be permitted as a means of extending a superclass method (in accordance with the guidelines for *Method Extension*).

In accordance with the *Initialization dispatch rule*:. the body of a constructor should not be permitted to call any operations on the object under construction except other constructors or private operations.

The bounded and deterministic nature of dynamic dispatch must be demonstrated based on the actual implementation. Typically dispatch tables are constructed by a static linker, or by the Java Virtual Machine (JVM) or a Java processor as classes are pre-loaded. This makes dispatch times for `invokevirtual` [6] both small and fixed. The dispatch time for `invokeinterface` [6] potentially involves a search and may introduce a higher overhead. Dispatch times, however, should still be both bounded and deterministic. `invokeinterface` can also be

implemented using dispatch tables if the implementation takes advantage of the fact that new classes cannot be loaded dynamically, making it equivalent to `invokevirtual`.

### **B.1.1.5 Ada95 guidelines**

Ada is a strongly typed language. To introduce basic object-oriented features, Ada95 provides tagged types as an extension to the existing concept of a record. Class wide types provide a means to declare objects that may (polymorphically) hold any of a number of related tagged type values, or corresponding access type values. The Ada95 language requires primitive operations on a tagged type to appear in the same declaration list as the type declaration, and to have at least one parameter or a return type that is of the tagged type. Operations can also be provided that take a corresponding class wide parameter. An Ada package that defines a single tagged type and primitive and class wide operations on that type corresponds to the concept of a class in C++ and Java [15, p. 169, section 6.2.1]<sup>6</sup>. The dynamic loading of classes is generally not supported, and only single inheritance of either interface or implementation is permitted. For tagged types, the run time type (tag) of an object is assigned at the point at which it is created and cannot be changed during the object's life time.

With regard to the *Simple overriding rule*., code reviews should be used to enforce the rule that an overridden version of a method can only report either the same errors, or a more restricted set of errors than its parent version. The other restrictions are enforced by the language.

Tagged types provide the run-time type information (tag) required to make dispatching calls to primitive operations associated with a type. Dynamic dispatch occurs when the argument corresponding to the tagged type parameter is of a class wide type (polymorphic). With regard to the *Simple dispatch rule*., the risk is that an overridden operation might be called with an argument declared to be of a specific tagged type, when the argument itself has the run-time tag of some derived type<sup>7</sup>. This can occur because Ada95 permits *view conversions* between specific tagged types so long as this conversion is toward the root of the hierarchy [10, pp. 278]. In such conversions the underlying object (and its tag) are not changed, only the program's view of it [10, pp. 288]. The easiest way to enforce the *Simple dispatch rule*., is to forbid view conversions between specific tagged types, ensuring that all arguments are either of a class wide type or of a specific tagged type with a matching tag. Conversions between a specific tagged type and a class wide type of an ancestor type, however, are still allowed.

In accordance with the *Initialization dispatch rule*., the body of a constructor should not be permitted to call any operations on the object under construction except other constructors or private operations. Ada95 does not provide implicitly called constructors. By convention, however, we can provide initialization procedures that can be called explicitly.

Such an initialization procedure should call the parent type's initialization procedure to initialize all inherited fields, then initialize the fields defined by the type extension. In accordance with *Method Extension*, this call to the parent initialization procedure *should* involve an explicit view conversion of the argument to the specific parent type, intentionally avoiding the use of dynamic dispatch. To help ensure the initialization procedure is called when an object is created, we can also provide a create function [15, p. 194] that allocates the object, calls the initialization procedure, and returns the initialized result.

The bounded and deterministic nature of dynamic dispatch must be demonstrated based on the actual implementation. Typically, however, dispatch tables are constructed by the compiler or linker, making dispatch times both small and fixed.

### **B.1.1.6 C++ guidelines**

C++ is a strongly typed language if conversions between logically unrelated types are avoided. Since such avoidance is not possible within the constructs of the language itself, a tool that specifically checks for such conversions must be used. C++ supports single dispatch based on the target object of the method call. The dynamic

---

<sup>6</sup> Although, unlike C++ and Java, we cannot control the visibility of individual attributes (record fields). This, however, is of little consequence if all data is hidden, as in normal practice. The Ada95 tagged type can be designated as private to ensure this.

<sup>7</sup> This would not be a problem if the argument were declared to be of a class wide type because dynamic dispatch would then occur.

loading of classes is generally not supported. The run time type of an object is assigned at the point at which it is created and cannot be changed during the object's life time.

With regard to multiple inheritance, code reviews should be used to ensure that only single inheritance is permitted with respect to implementation.

With regard to the *Simple overriding rule*., code reviews should be used to ensure that the overridden version of a method can only report either the same errors, or a more restricted set of errors than its parent version. Overriding methods should not declare default parameter values [16, p. 171]. All other restrictions are enforced by the language.

Dynamic dispatch occurs in C++ when the called method is declared to be virtual and the target object is specified as a pointer or reference. With regard to the *Simple dispatch rule*., , the risk is that an overridden operation might be called with respect to a target object whose declared type is a superclass of its actual run-time type, and dynamic dispatch might not occur.

To avoid problems with the declaration of overridden methods, a subclass should never be allowed to redefine an inherited non-virtual function [16, p. 169]. This requires all public and protected operations to be declared using the keyword "virtual" if subclasses are allowed to redefine them.

To avoid problems with the specification of the target object, all calls to virtual functions should involve a target object specified as a pointer or reference.

Since the normal rules for dynamic dispatch do not apply during the execution of constructors and destructors, direct and indirect calls to overridden methods during their execution should also be avoided.

Doing so is also consistent with the *Initialization dispatch rule*., , which forbids calls to overridden methods during object construction.

The bounded and deterministic nature of dynamic dispatch must be demonstrated based on the actual implementation. Typically, however, dispatch tables are constructed by the compiler or linker, making dispatch times both small and fixed.

## B.1.2 Extension of the Method Extension Guidelines

The following sections extend the definition of *Method Extension* to describe its implementation in Java, Ada95, and C++.

### B.1.2.1 Java guidelines

In Java, the implementation of these guidelines involves a call to the superclass version of the same method using the keyword `super`. Only calls to the superclass version of the same method should be allowed.

### B.1.2.2 Ada95 guidelines

In Ada95, the implementation of these guidelines involves a view conversion from the derived type to the parent (superclass) type and then a call to the parent operation that supports this implementation. This has the same effect as the use of the keyword `super` in Java. Only calls to superclass versions of the same method should be allowed.

### B.1.2.3 C++ guidelines

In C++, the implementation of these guidelines involves qualification of the "::" operator. Only qualified calls to the immediate base class version of the same member function should be allowed. In addition, the "::" operator may be used under the following circumstances:

- If class A defines method `f()` and then class B inherits A and defines method `f(int)`, within class B the method `f()` is hidden by the declaration of `f(int)`. The only way to get `f()` from within B is to use `A::f()`. This is considered to be safe if `f()` cannot be overridden (i.e. it is not declared 'virtual'). Similarly, the "::" operator is needed to access global functions that are hidden by method declarations (although global methods should generally be avoided), and to access methods declared in a namespace.

- When code is generated by a tool and a method call is made using an object whose exact type is known, it is reasonable for the tool to use the '::' qualification to avoid the overhead of dynamic dispatch.

## B.2 Multiple Inheritance

### B.2.1 Composition involving multiple inheritance

The following cases illustrate the primary issues to be resolved with respect to composition involving multiple inheritance. They are based on examples appearing in Meyer [18].

#### B.2.1.1 Case 1: Repeated inheritance

“As soon as multiple inheritance is allowed into a language, it becomes possible for a class (e.g. FrenchUSDriver) to inherit from two classes (e.g. FrenchDriver and USDriver), both of which are subclasses of the same class (e.g. Driver). This situation is called *repeated inheritance*.” [18, p. 543]. It is characterized by the diamond shape of the inheritance hierarchy (Figure B.2-1 ).

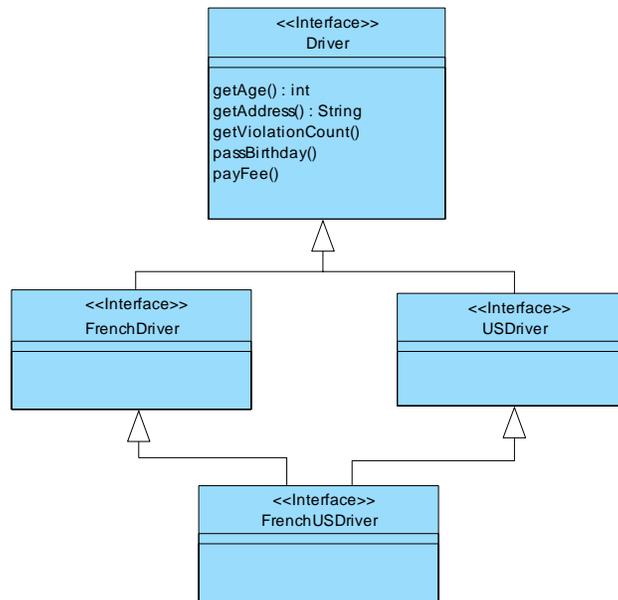


Figure B.2-1 Repeated inheritance: sharing and replication, based on [18, p. 547]

The fundamental question with respect to repeated inheritance is whether inheritance of the same operation along more than one path should result in a single operation in the subinterface or in multiple operations.

Since we are dealing with this issue with respect to interfaces (and not implementation), we must view this question from the client’s perspective. In general, a client will be satisfied if all subinterfaces of a given interface inherit a definition of the expected operation. Since this is guaranteed (we will have at least one definition of the operation), clients will always be happy in this regard.

The remaining question is whether repeated inheritance should ever result in more than one definition of the operation in the subinterface. A case for this can be made by the example appearing in Figure B.2-2. Each driver [of a motor vehicle] has an age, and a primary residence (and associated address). We are also interested in tracking the number of traffic violations committed by the driver, leading to a potential revocation of the person’s license.

Subinterfaces of `Driver` represent French drivers and US drivers. Drivers who have licenses in both countries are categorized as both French and US drivers<sup>8</sup>.

In terms of this example, it is clear that there should be a single operation to get the driver's age, which will be the same in both countries. Address and number of traffic violations, however, are potentially a different matter. The driver may have different addresses in each country and traffic violations committed in one country may not count against his/her driving record in the other. Similarly license fees may have to be paid at different times in each country and paying the fee in one country will not necessarily satisfy the other (although it may be possible to obtain an international driving license that can be used in both).

The need to be able to define both shared operations (such as `getAge` and `passBirthday`) and replicated operations (such as `getAddress`, `getViolationCount` and `payFee`) can be satisfied in a number of ways.

We could require that all replicated operations be specified redundantly. Doing so, we would require that `FrenchDriver` define the operation `getFrenchAddress`, while `UsDriver` defines an otherwise identical operation `getUsAddress`.

Alternately, the language could simply permit the renaming of those operations to be replicated in the subinterface and assume that repeated inheritance otherwise implies sharing. This is appealing because the situations in which replication is the right choice are relatively rare. In most cases (especially those involving interfaces), sharing is the desired result. As Meyer notes, "Cases of repeated inheritance similar to the transcontinental drivers (Figure B.2-2), with duplicated operations as well as shared ones, do occur in practice, but not frequently." The case involving only shared operations is far more common, especially with regard to interface inheritance. For this reason, it is most important that sharing be supported well at the language level, or in any guidelines we prescribe. Additional work (or the use of work arounds) is probably acceptable in the less common case involving replication.

Sharing is also appropriate when we view multiple interface inheritance as a means of breaking up a large interface specification into smaller interface specifications (superinterfaces) intended for particular categories of clients. In Figure B.2-3, for instance, we begin with the definition of a single large interface `AvionicsDataServiceInterface`. This interface is large because it contains the operations needed by all clients. This, unfortunately, makes it unwieldy for them all.

No particular type of client, however, may need the full set of operations. Rather clients of type `Producer` may need a given subset of the operations, while clients of type `Consumer` may need a different (but overlapping) subset, and so on. To simplify each client's view, we define a separate interface containing only the operations that it needs (Figure B.2-4). Because operations that appear in more than one of these client specific superinterfaces have the same source (`AvionicsDataServiceInterface`), it is clear that they are intended to represent the same operation. (The operation `getDataChannel` in the `Producer` interface is the same as the operation `getDataChannel` in the `Consumer` interface because both are taken from the definition of `getDataChannel` provided by `AvionicsDataServiceInterface`.) As a result, definitions of such operations should always be shared. This view is also consistent with the policies of Java and C++.

---

<sup>8</sup> Bertrand Meyer, whose wife is French, is a case in point.

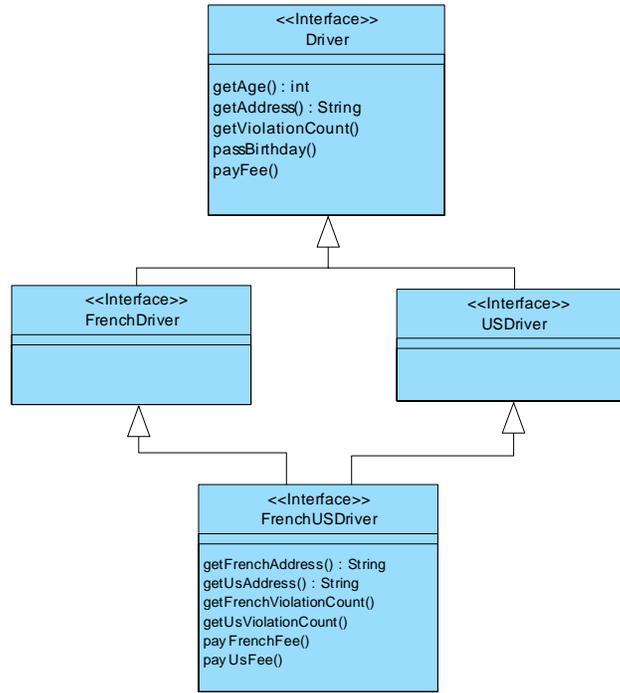


Figure B.2-2 Shared and replicated operations

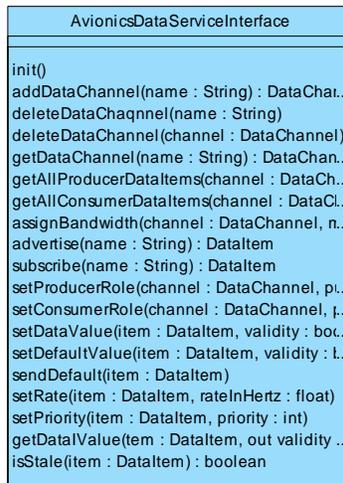


Figure B.2-3 A Single large interface to an avionics data source



Figure B.2-4 Separate interfaces for different types of clients

### B.2.1.2 Case 2: Redefinition along separate paths

The ability to specialize the definition of an operation in a subinterface is fundamental to object-oriented development. The same operation, however, may be specialized (redefined) in different ways along different paths in the classification hierarchy. The question then arises as to what the result should be when we inherit more than one definition/redefinition of *the same* operation in a given subinterface.

The answer hinges on whether sharing or replication is intended, and (if sharing is intended) whether the specializations are compatible.

A simple way to guarantee this result is to require the user to define a version of the operation in the subinterface that obeys the *Simple overriding rule*: with respect to each of its parent interfaces. This leads us to the result in Figure B.2-5.

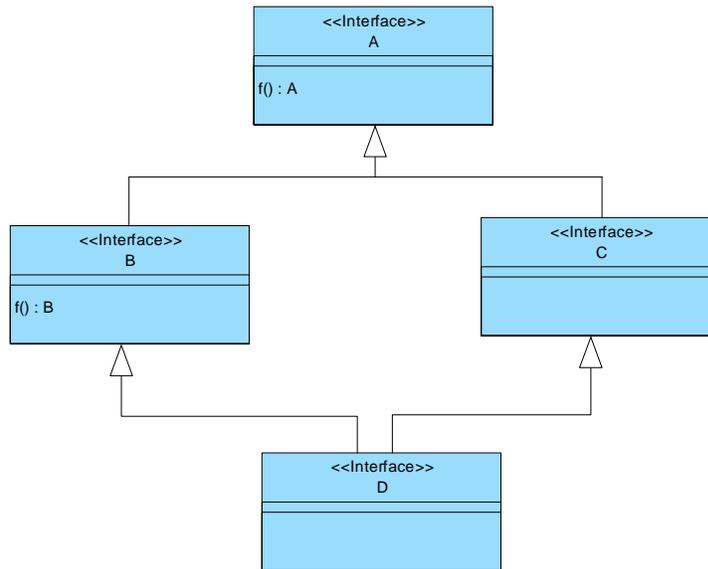


Figure B.2-5 Redefinition along separate paths, based on [18, p. 551]

In general, the combined operation has a precondition that represents an *or'ing* of the preconditions of all inherited definitions of the operation, and a postcondition that represents an *and'ing* of all inherited postconditions. Type constraints on *in* parameters are considered part of the precondition. Type constraints on *out* parameters and the result, and any restrictions on errors reported/exceptions thrown are considered part of the postcondition.

These guidelines are simply intended to help the user write the correct signature for the combined operation. The target language compiler should catch all errors associated with the result of doing so, including errors resulting from attempts to combine conflicting definitions.

Adopting the simple view of interface inheritance as a factoring of a large interface into smaller ones targeted to specific categories of clients, we could instead forbid refinement of operations along separate paths. This is certainly consistent with the idea that all refinements of an operation be compatible (in this case they would have to be identical). However, forbidding refinement of operations may be less flexible than we would like in situations such as that given above, and would certainly be more restrictive than is required to type safe<sup>9</sup>.

<sup>9</sup> With respect to the languages of primary interest, the ways in which operations may be refined are limited (C++ permits the return type to be made more specific, Java permits the elimination of exceptions from the exception list). The ability to make the return type more specific, however, has been shown to have a large effect upon the number of run time casts required [22].

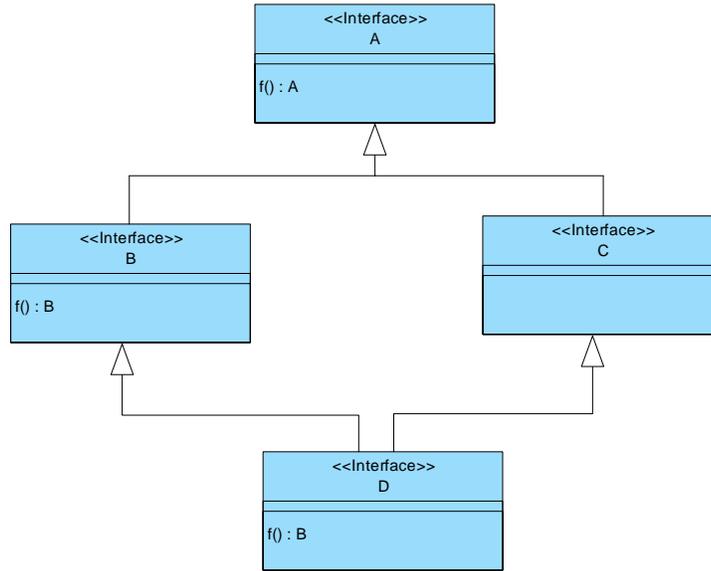


Figure B.2-6 Explicit definition of combined operation in subinterface

**B.2.1.3 Case 3: Independently defined operations with same signature**

A different situation arises when two parent interfaces *independently* define operations with the same signature. This is not repeated inheritance since we are not talking about inheriting *the same* operation via more than one path, but *different* operations, independently defined, that have the same signature. The key question is whether the matching of the signatures is intentional or accidental.

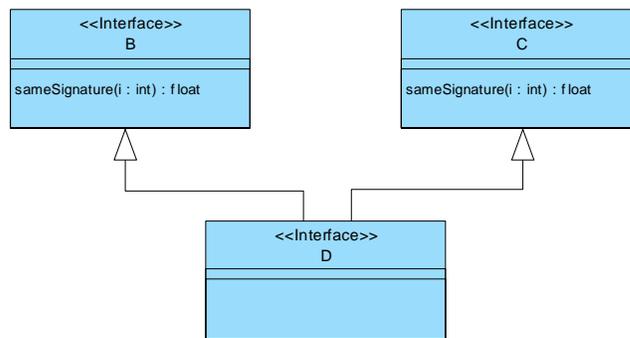


Figure B.2-7 Independently defined operations with same signature, based on [18, p. 550]

If the operations were completely and formally specified, we could compare preconditions and postconditions to see if the semantics are the same. If they are, then a single operation that does what they both promise to do should be sufficient in all cases.

Alternately we could adopt the view that interface inheritance represents only a factoring of a large interface into smaller ones targeted to specific categories of clients. If we use interface inheritance in only this way, then it is clear that we intend the separately inherited operations to be the same (i.e. sharing is always the right answer). This view is also consistent with the policies of Java and C++.

## B.2.2 Extended guidelines

### B.2.2.1 Extension of the Multiple Interface Inheritance Guidelines

The following sections extend the guidelines for *Multiple Interface Inheritance* to include language specific guidelines for Java and C++. In general, it is only necessary to enforce (e.g., by means of design and code inspections) those guidelines that the language does not enforce itself.

#### B.2.2.1.1 Java guidelines

In Java, a UML interface is represented by a Java interface defining only abstract methods and compile time constants. Constants whose value is computed at run-time should not be permitted, even when this value is computed once and never again changed.

The Java language enforces the *Repeated interface inheritance rule*:. Where operations should be replicated rather than shared, they must be given distinct names.

Java *implicitly* combines redefined methods inherited along different paths, enforcing the subtyping guidelines with respect to method signatures and the use of checked exceptions. It also permits the *explicit* combination of redefined methods in the sub-interface as recommended by the *Interface redefinition rule*:. Code reviews must be used to enforce this.

When more than one super-interface *independently* defines a method with the same signature, Java considers them to represent the same method. Code reviews must be used to ensure this is the real intent, i.e. that the matching of signatures is not simply accidental. As suggested by the guidelines for *Multiple Interface Inheritance*, a comment annotation should be used to document this intent, ensuring it is properly maintained.

#### B.2.2.1.2 C++ guidelines

In C++, a UML interface is represented by an abstract class defining only pure virtual member functions and compile time constants. Constants whose value is computed at run-time should not be permitted, even when this value is computed once and never again changed.

In accordance with the *Repeated interface inheritance rule*:. all base classes of a C++ interface class must be virtual base classes. Where operations should be replicated rather than shared, they must be given distinct names.

C++ *implicitly* combines redefined methods inherited along different paths, enforcing the subtyping guidelines with respect to method signatures. It also permits the *explicit* combination of redefined methods in the sub-interface as recommended by the *Independent interface definition rule*:. Code reviews must be used to enforce this.

When more than one super-interface *independently* defines a method with the same signature, C++ considers them to represent the same method. Code reviews must be used to ensure this is the real intent, i.e. that the matching of signatures is not simply accidental. As suggested by the guidelines for *Multiple Interface Inheritance*, a comment annotation should be used to document this intent, ensuring it is properly maintained.

### B.2.2.2 Extension of the Multiple Implementation Inheritance Guidelines

The following section extends the guidelines for *Multiple Implementation Inheritance* to include language specific guidelines for C++. In general, it is only necessary to enforce (e.g., by means of design and code inspections) those guidelines that the language does not enforce itself.

### **B.2.2.2.1 C++ guidelines**

In accordance with the *Repeated implementation inheritance rule*:, virtual inheritance should be used by default. Performance considerations should be taken into account only in response to a demonstrated need, and in accordance with the 80-20 rule (which suggests that some 20% of the code is executed 80% of the time).

Renaming [22, pp. 273..275] should always be used to distinguish inherited methods that are intended to be different in the subclass. Otherwise, an overriding method should be defined in the subclass that either selects between the competing implementations or otherwise combines them<sup>10</sup>.

The overridden methods must be compatible with one another (in terms of their preconditions and postconditions) for their overriding by a single overriding subclass method to be valid. This is true both when the competing implementations have a common definition in a superclass (in accordance with the *Implementation redefinition rule*:) or when they do not (in accordance with the *Independent implementation definition rule*:).

---

<sup>10</sup> This explicit form of selection is preferred even though C++ provides for implicit selection in some cases in accordance with its own dominance rule [22, p. 263].

### B.3 Dead and Deactivated Code, and Reuse

#### B.3.1 Deactivated Code Examples

Figure B.3-1 presents a class (*C<sub>x</sub>*) being used by a client (our system). In this diagram, the methods (*M<sub>x</sub>*) are annotated with the attributes (*A<sub>x</sub>*) and methods they access in italics, as is the client. From the point of view of the client, class (*C<sub>3</sub>*), methods (*M<sub>3</sub>*, *M<sub>4</sub>*, *M<sub>6</sub>*) and attributes (*A<sub>2</sub>*, *A<sub>4</sub>*) appear to be dead code (i.e., not used by this system).

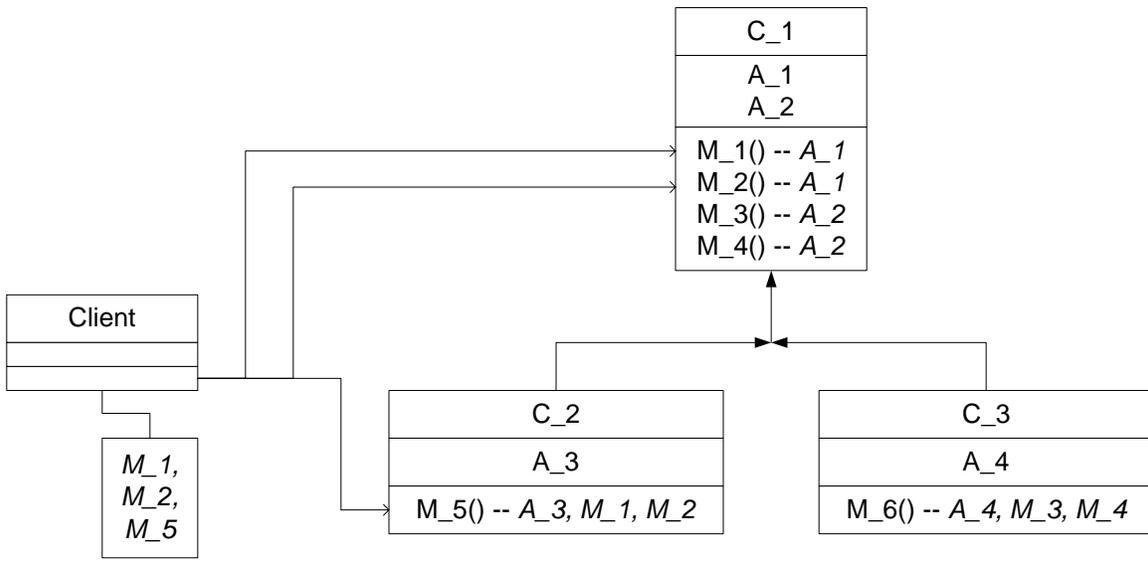


Figure B.3-1 Deactivated Code

### B.3.2 Hierarchy Changes and Method Overriding

For an example of a subtle effect in object-oriented (OO) software, consider the classes, shown in Figure B.3-2, displayed in both a normal and flattened hierarchy. Here, class C\_1, which contains methods M\_1() and M\_2(). M\_1() calls M\_2(). Now consider a sub-class, C\_2 that inherits C\_1, but overrides M\_2(). M\_1() in class C\_2 is effectively also overridden as it makes a call to a different M\_2() than the M\_1() in C\_1. There are other situations where changes in the class hierarchy can be subtle and difficult to discover.

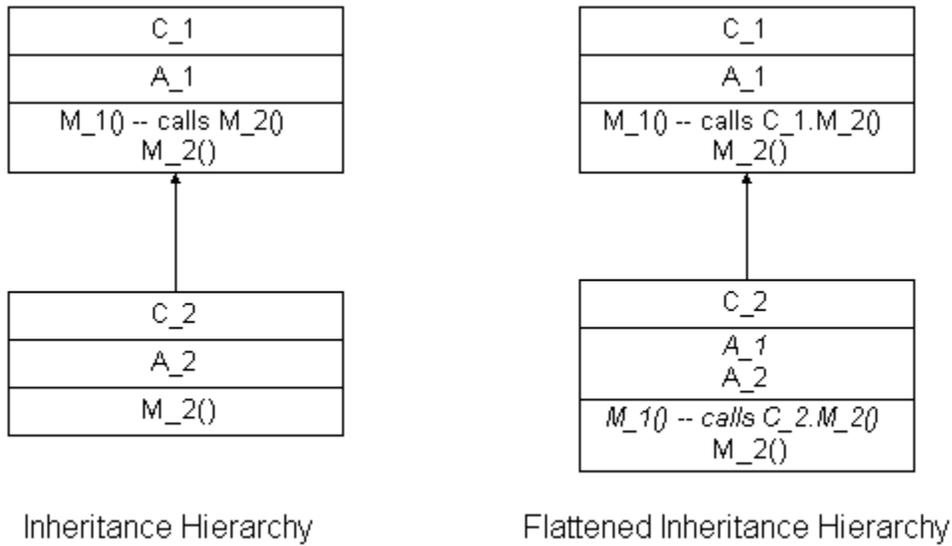


Figure B.3-2 Method Overriding