

FAA National Software Conference, May 2002

Object Oriented Guidelines



AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Gary Daugherty
Rockwell Collins
gwdaughe@rockwellcollins.com
319.295.4065



Aerospace Vehicle Systems Institute

What is AVS I?

- A consortium of Boeing and its suppliers
- A forum for collaborative research and related efforts
- Managed by Texas A&M
- Two technical panels
- This project is sponsored by the AVS I Common Tools & Processes Panel
- Participants: Boeing, B.F. Goodrich, Honeywell, Rockwell-Collins

FAA National Software Conference, May 2002

Object Oriented Guidelines



Project: AFE #7 Certification Issues for Embedded Object-Oriented Software

Situation

- Object-Oriented software is rapidly becoming commonplace since it reduces cost via reuse
- However, it has not been widely used in safety-critical avionics software
- DO-178B and FAA do not have *explicit* guidelines for the use of this technology
- As a result, individual programs have verification and certification risks

Page 3

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Project: AFE #7 Certification Issues for Embedded Object-Oriented Software

Idea

- Identify and resolve issues specific to Object-Oriented software with respect to DO-178B by pooling the resources of Boeing and its suppliers

Approach

- Document the guidelines: *A Guide to the Certification of Systems with Embedded Object-Oriented Software*
- Evaluate supporting tools, e.g. for coverage and enforcement
- Make FAA presentations and obtain their concurrence
- Cooperate with FAA funded effort by John Chilenski to define coverage requirements

Page 4

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

 Team	
Participating Company	Team Members
Boeing Co-Principal Investigator	John Chilenski Seattle, WA
Honeywell Co-Principal Investigator	Dennis Cornhill, Wayne Schultz Minneapolis, MN
BFGoodrich Co-Principal Investigator	Tom Rhoads Vergennes, VT
Rockwell Collins Prime Contractor	Gary Daugherty Cedar Rapids, IA
AVSI	David Lund College Station, TX

Page 5 AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

 Issues	
<p>The project addressed use of the following OO features :</p> <ul style="list-style-type: none">- Dynamic dispatch- Single inheritance of interfaces and the overriding of operations- Single inheritance of implementation and the overriding of methods- Multiple inheritance of interfaces and implementation- Inlining- Template classes and template operations- Dead and deactivated code in reusable components	

Page 6 AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

Result

A Guide to the Certification of Systems with Embedded Object-Oriented Software

- OO features evaluated with respect to their impact on the analysis and testing requirements of DO-178B
- Results summarized in tables similar to those appearing in *Guide for the Use of the Ada Programming Language in High Integrity Systems*, IS OI EC PDTR 15942
- Restrictions defined by a collection of design and process “patterns”
- Expressed first at a design level, then mapped to language specific rules
- Rationale for the guidelines
- Related tool support

Page 7
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

General guidelines

Issue	FA	RC	SU	TA	OMU	RT	SC
1. Dynamic dispatch	Rstr ¹	Inc	Inc	Rstr ²	Inc	Rstr ³	Rstr ^{3,4,5}
2. Inline	Rstr ⁷	Inc	Rstr ⁸	Rstr ⁸	Inc	Inc	Rstr ^{9, 10, 15}
3. Dead code	Exc	Exc	Exc	Exc	Exc	Exc	Exc
4. Deactivated code	Rstr ¹³	Rstr ¹³	Rstr ¹⁴	Rstr ¹⁴	Rstr ¹⁴	Rstr ¹³	Rstr ¹³
5. Single inheritance of interfaces and overriding	Inc	Inc	Inc	Inc	Inc	Rstr ³	Inc
6. Multiple inheritance of interfaces	Inc	Inc	Inc	Inc	Inc	Rstr ^{3, 6}	Inc
7. Single inheritance of implementations and overriding	Rstr	Inc	Inc	Rstr	Inc	Rstr ³	Rstr ³
8. Multiple inheritance of implementations	Rstr ¹¹	Inc	Inc	Rstr	Inc	Rstr ³	Rstr ¹¹
9. Template classes	Rstr ¹²	Inc	Rstr ¹²	Rstr ¹²	Inc	Inc	Rstr ¹²
10. Template operations	Rstr ¹²	Inc	Rstr ¹²	Rstr ¹²	Inc	Inc	Rstr ¹²

Page 8
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

DO-178B specific guidelines

Issue	<i>Level A</i>	<i>Level B</i>	<i>Level C</i>	<i>Level D</i>
1. Dynamic dispatch	Rstr ^{1,2,3}	Rstr ^{1,2}	Rstr ^{1,2}	Rstr ¹
2. Inline	Rstr ^{6,7,8}	Rstr ^{6,8}	Rstr ^{6,8}	Rstr
3. Dead Code	Exc ⁵	Exc ⁵	Exc ⁵	Rstr
4. Deactivated Code	Rstr ¹¹	Rstr ¹¹	Rstr ¹¹	Rstr ¹¹
5. Single inheritance of interfaces and overriding	Rstr ^{1,2}	Rstr ^{1,2}	Rstr ^{1,2}	Rstr ¹
6. Multiple inheritance of interfaces	Rstr ^{1,2,4}	Rstr ^{1,2,4}	Rstr ^{1,2,4}	Rstr ^{1,4}
7. Single inheritance of implementations and overriding	Rstr ^{1,2}	Rstr ^{1,2}	Rstr ^{1,2}	Rstr ¹
8. Multiple inheritance of implementation	Exc ⁹	Exc ⁹	Exc ⁹	Rstr ^{1,9}
9. Template classes	Rstr ¹⁰	Rstr ¹⁰	Rstr ¹⁰	Rstr ¹⁰
10. Template operations	Rstr ¹⁰	Rstr ¹⁰	Rstr ¹⁰	Rstr ¹⁰

Page 9
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

Pattern 4.1: Inheritance with overriding

3 rules + general guidance

Addresses issues related to inheritance, overriding and dispatch

1. Simple overriding rule
2. Simple dispatch rule
3. Initialization rule

Page 10
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Background: Inheritance

- Inheritance was originally viewed as a mechanism for sharing code and data definitions
- As understanding of OO modeling has matured, however, the focus has increasingly been on the specification of interfaces
- And the specification of interfaces as 'contracts' between clients and implementers

Page 11

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Background: Classes

A

Class ≈ Ada83 package spec that defines a record type & associated operations

Class instance = object

Page 12

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

 **Background: Attributes**

A

..... Attribute \approx Record field
public (+)
protected (#)
private (-)

Page 13 AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

 **Background: Operations, methods, signatures**

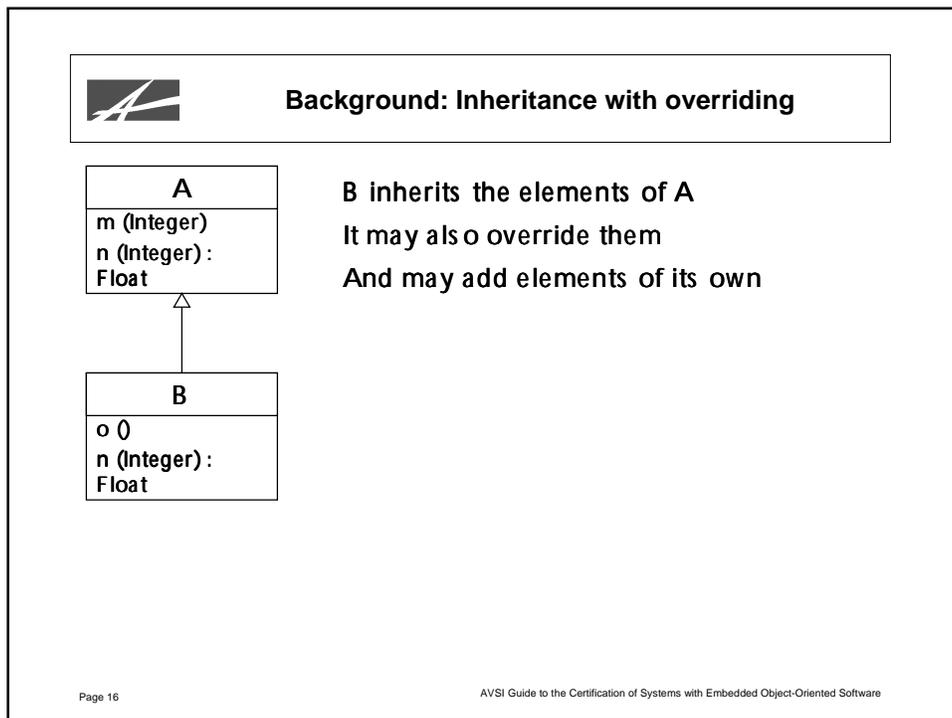
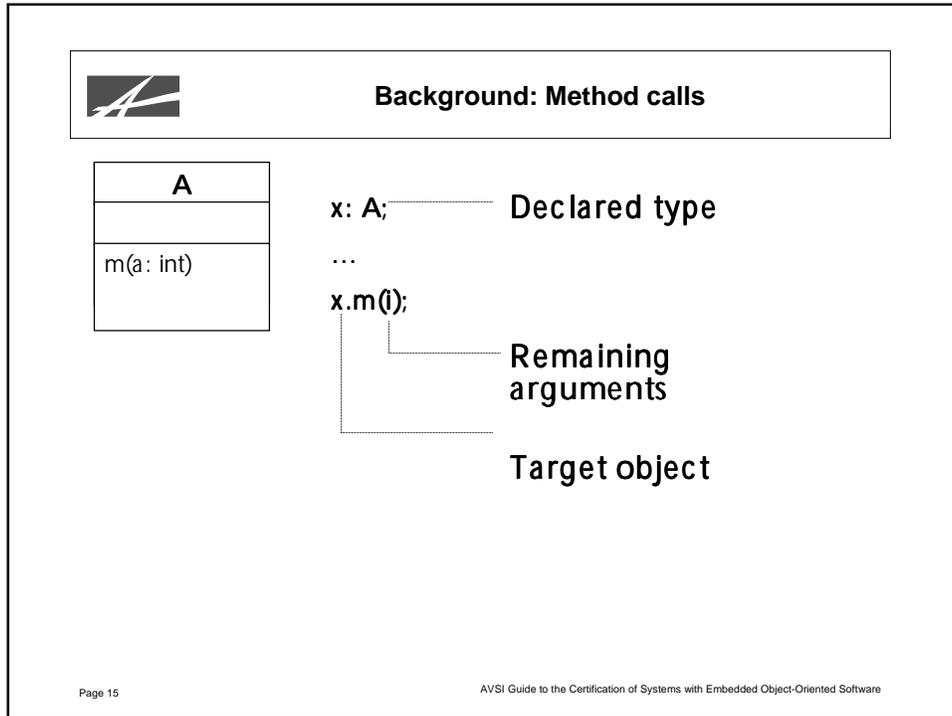
A

..... Operation \approx subprogram spec
Also public (+), protected (#), private (-)
Method \approx subprogram body
Signature, e.g. "m (p: Integer)"
Overloading
Constructor
Destructor

Page 14 AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



FAA National Software Conference, May 2002

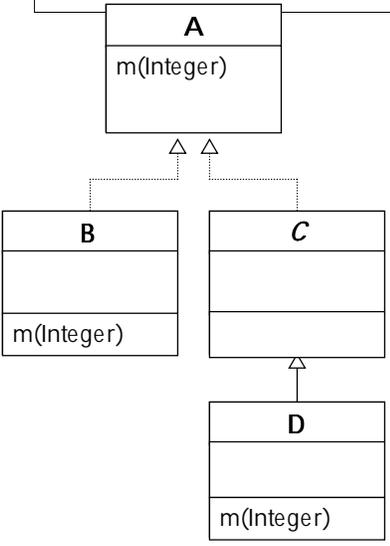
Object Oriented Guidelines

**Background: Polymorphism**

- **Polymorphism permits instances of a subclass to be assigned to variables declared to be an associated superclass**
target: DeclaredType = new RunTimeType ();
- **Dynamic dispatch ensures the method executed by a call is that associated with the object's run time type**
target.operation (arguments);

Page 17 AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

**Background: Polymorphism**



```
classDiagram
    class A {
        m(Integer)
    }
    class B {
        m(Integer)
    }
    class C {
        m(Integer)
    }
    class D {
        m(Integer)
    }
    A <|-- B
    A <|-- C
    C <|-- D
```

x: A = new B; Run-time type
 Declared type

...

x.m(i);

Page 18 AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Background: Dynamic dispatch

- Method selection is a function of the run time type and the method signature
- Conceptually there is a single dispatch routine for all calls containing a pair of nested case statements

```
case of run time type
case (< type> )
  case of method signature
  case (< method> )
    call < method> defined by < type>
  end
end
```
- In practice, calls to this universal dispatch method are 'inlined' at the point of call

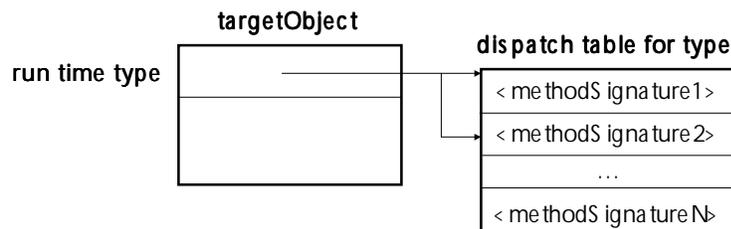
Page 19

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Background: Dynamic dispatch

- Typically implemented using "dispatch tables"
- Small, fixed overhead
- At the point of call: (1) get the dispatch table associated with the target object, (2) index it by a number associated with the method signature, and (3) invoke the method



Page 20

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.1: Inheritance with overriding

1. Simple overriding rule

An operation may override an inherited operation with the same signature by associating a method with it in the subclass definition, by making it more visible to clients, by subtyping its return type, or by being more restrictive regarding the types of errors it can report to clients (e.g., as exceptions or by setting error return codes). No other form of overriding should be allowed.

Ensures LSP is not violated at the language level, in terms of method declarations.

Page 21

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

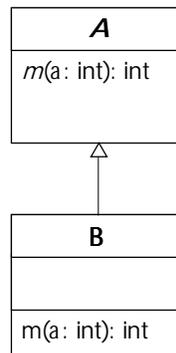


Simple overriding rule

Associating a method with an operation in a subclass:

```
abstract class A {  
    abstract int m(int a);  
}
```

```
class B extends A {  
    int m(int a) {  
        return a*a;  
    }  
}
```



Page 22

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

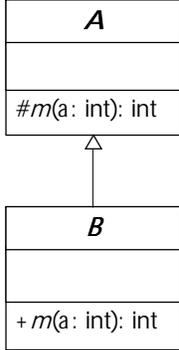


Simple overriding rule

Making an operation more visible to clients:

```
abstract class A {  
    abstract protected int m(int a);  
}
```

```
abstract class B {  
    abstract public int m(int a);  
}
```



```
classDiagram  
    class A {  
        #m(a: int): int  
    }  
    class B {  
        +m(a: int): int  
    }  
    A <|-- B
```

Page 23 AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

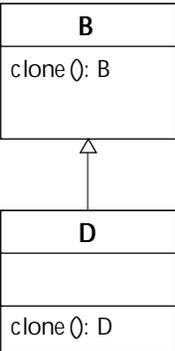


Simple overriding rule

Subtyping the return type:

```
class B {  
    public:  
    virtual B* clone () {  
        return new B(*this);  
    }  
}
```

```
class D: public B {  
    public:  
    virtual D* clone () {  
        return new D(*this);  
    }  
}
```



```
classDiagram  
    class B {  
        virtual clone(): B  
    }  
    class D {  
        virtual clone(): D  
    }  
    B <|-- D
```

Page 24 AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

Simple overriding rule

Restricting the types of errors reported:

```
interface A {
    abstract int m(int a) throws Exception;
}
```

```
interface B extends A {
    int m(int a) throws MyException;
}
```

A
<i>m</i> (a: int): int

↑

B
<i>m</i> (a: int): int

Page 25
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

Simple overriding rule

Overriding combinations of these properties:

```
abstract class A {
    abstract protected int m(int a);
}
```

```
class B extends A {
    public int m(int a) {
        return a*a;
    }
}
```

A
<i>m</i> (a: int): int

↑

B
+ <i>m</i> (a: int): int

Page 26
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Violation of simple overriding rule

Throwing unexpected exceptions [Binder, naughty children]

```
interface A {
    int m(int a);
}

interface B extends A {
    int m(int a) throws SomeException;
}
```

Page 27

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Violation of simple overriding rule

Redefinition of inherited default parameter values in C++ [Scott Meyers, Effective C++, p. 171]

```
enum ShapeColor {RED, GREEN, BLUE};

class Shape {
public:
    virtual void draw (ShapeColor color = RED) const = 0;
    ...
};
```

Page 28

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Violation of simple overriding rule

```
class Rectangle: public Shape {  
public:  
    virtual void draw (ShapeColor color = GREEN) const;  
    ...  
};
```

```
Shape *pr = new Rectangle;  
pr->draw(); //call to draw defined by Rectangle
```

This, however, does not draw a green rectangle as we would expect

Page 29

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Violation of simple overriding rule

Subtyping of parameters in Eiffel [Binder, naughty children]

```
class Skier  
feature  
    share (other: Skier) is  
        ...  
    end  
end
```

```
class Girl inherit Skier redefine share end  
feature  
    share (other: Girl) is  
        ...  
    end  
end
```

Page 30

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.1: Inheritance with overriding

2. Simple dispatch rule

When an operation is invoked on an object, the method associated with the operation in its run time class should be executed.

This rule should apply to all calls except explicit calls to superclass methods, which should be addressed as described in Pattern 4.3 (Method Extension).

Ensures method dispatch is semantically equivalent to case, without considering its underlying implementation.

Page 31

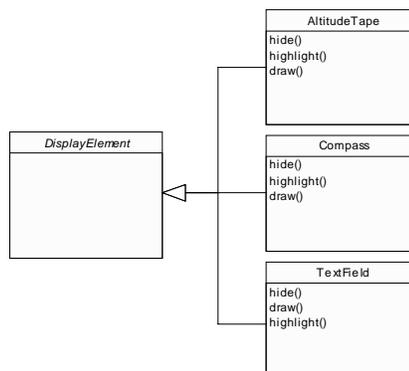
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Simple dispatch rule

When an operation is invoked on an object, the method associated with the operation in its run time class should be executed.

```
DisplayElement e;  
e = new AltitudeTape();  
e.draw();
```



Page 32

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Violation of the simple dispatch rule

Overriding of non-virtual function in C++ [Scott Meyers, Effective C++, p. 169] (issue: programmer specified optimizations)

```
class B {  
public:  
    void mf(); //mf, as defined by B  
    ...  
};
```

Page 33

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Violation of the simple dispatch rule

```
class D: public B {  
public:  
    void mf(); //includes action to maintain invariant for D  
    ...  
};  
  
B *pB = new D;  
pb->mf(); //calls B's mf on an object with run time class D
```

As a result, we fail to maintain the invariant for object x

Page 34

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.1: Inheritance with overriding

3. Initialization rule

No call to an externally visible operation of an object other than its constructors should be allowed until it has been fully initialized.

Prevents calls to subclass methods before subclass attributes have been initialized and subclass invariants have been established.



Initialization rule

No call to an externally visible operation of an object other than its constructors should be allowed until it has been fully initialized.

```
class Y extends X {  
    private int a;  
    private Vector b;  
    public X(int n) {super(); a = n; b = createVector (a);}  
    private Vector createVector (int size) { ... }  
}
```

FAA National Software Conference, May 2002

Object Oriented Guidelines



Violation of the initialization rule

Dispatch to subclass method during object construction (issue = initialization)

```
class AA {
    int next;

    public AA() {next = firStElement();}

    public int firStElement() {return 1;}
}
```

Page 37

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Violation of the initialization rule

```
class BB extends AA {
    int min;
    int max;

    public BB (int min, int max) {
        super();
        this.min = min;
        this.max = max;
    }

    public int firStElement() {return min;}
}
```

Page 38

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.1: Inheritance with overriding

Source code to object code traceability

Nearly all compilers implement [dynamic dispatch] by associating a **method table** with the target object's run time class that is **indexed by a method number at the point of call**.

Where concerns about source code to object code traceability and timing analysis dictate, the **compiler vendor** may be asked to provide **evidence of this mapping**, or evidence of a semantically equivalent mapping that guarantees that dispatch times are predictable and bounded.

Where concerns about source code to object code traceability lead to inspection of the object code produced by the compiler, it may also be necessary to **examine the structure of the method tables and the compiler generated code at the point of call**.

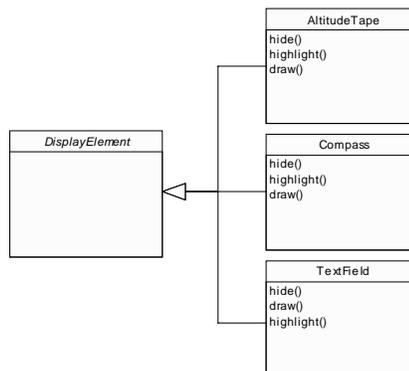
Page 39

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Source code to object code traceability

```
DisplayElement e;  
e = new AltitudeTape();  
e.draw();
```



Page 40

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

Source code to object code traceability

```

dispatch (DisplayElement e, Method m)
  case of e's run time type
  case (AltitudeTape)
    case of method m
    case (draw)
      invoke AltitudeTape.draw();
    case (highlight)
      invoke Compass.highlight();
    case (hide)
      invoke TextField.hide();
    end
  ...
end
end
            
```

AltitudeTape
hide()
highlight()
draw()

Compass
hide()
highlight()
draw()

TextField
hide()
draw()
highlight()

Page 41
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

Source code to object code traceability

RO →	e		AltitudeTape
		0	hide
		1	highlight
		2	draw

```

MOV @RO, R1 -- Get the method table address
ADD #2, R1 -- Add the offset for "draw()"
JSR PC,@R1 -- Invoke the method
            
```

AltitudeTape
hide()
highlight()
draw()

Compass
hide()
highlight()
draw()

TextField
hide()
draw()
highlight()

Page 42
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Source code to object code traceability

- Think of dispatching as the normal case
- Think of all calls as mapping to the previous instruction sequence
- Statically bound calls are then an optimization
- And correspond to the special case in which there is a single choice of which method to call

Page 43

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Pattern 4.2: Subtyping

2 rules + general guidance

Addresses issues related to superclass/subclass compatibility and the reuse of verification artifacts and results

1. Inherited test case rule
2. Separate context rule

Page 44

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.2: Subtyping

1. 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. Only test cases for private operations (like private operations themselves) are not inherited.

Enforces LSP without requiring formal specification.

Page 45

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Pattern 4.2: Subtyping

Corollary

If the subclass invariant is stronger than that of its superclasses, 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.

When we run the inherited test cases against a subclass instance we must use the subclass invariant to check the result.

Page 46

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.2: Subtyping

2. Separate context rule

Each method should be separately tested in the context of every class in which it appears, irrespective of whether it is defined by the class or inherited by it.

Test cases are inherited but test results are not.

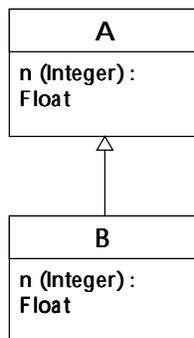
Each method must be re-tested in the context of each subclass.

Page 47

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Subtype compatibility



Consider the methods `n (Integer) : Float` and `n (Integer) : Float`

Each has a precondition = initial state for test cases

Each has a postcondition = expected result for test cases

Page 48

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

 **Liskov Substitution Principle**

A

n (Integer) :
Float

↑

B

n (Integer) :
Float

Precondition of n, must require the same or less than the precondition of n

Postcondition of n, must deliver the same or more than the postcondition of n

We test for this by "inheriting" all test cases

Page 49AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

 **Violation of subtyping rules: stronger precondition**

Stronger precondition in subclass [Binder, naughty children]

```
interface List {  
    ...  
    /*  
    * Adds an element to the list at a given position.  
    * postcondition: item appears in the list  
    */  
    void add (Object element, int index) throws OutOfMemory;  
}
```

Page 50AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Violation of subtyping rules: stronger precondition

```
interface BoundedList extends List {
    void setBounds (int min, int max);

    ...

    /*
     * Adds an element to the list at a given position.
     * precondition: index is within bounds
     * postcondition: item appears in the list
     */
    void add (Object element, int index) throws OutOfMemory;
}
```

Page 51

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Violation of subtyping rules: weaker postcondition

Weaker postcondition in subclass [Binder, naughty children]

```
interface Log {
    final int HIGH = 100;
    ...

    /*
     * Writes a message to the log.
     * postcondition: The message appears in the permanent log. If the
     priority
     * is HIGH or greater, the message is also displayed.
     */
    void writeMessage (String message, int priority);
}
```

Page 52

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Violation of subtyping rules: weaker postcondition

```
interface SimpleLog extends Log {
    ...
    /*
     * Writes a message to the log.
     * postcondition: The message appears in the permanent log.
     */
    void writeMessage (String message, int priority);
}
```

Page 53

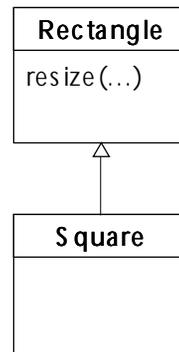
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Violation of subtyping rules: square peg

Square as a subtype of Rectangle, with inherited operation `resize (int width, int height)` [Binder, square peg in a round hole, faulty intuition]

```
interface Rectangle {
    ...
    void resize (int width, int height);
}
/*
 * invariant: width == height
 */
interface Square extends Rectangle {
    ...
}
```



Page 54

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

 **Violation of subtyping rules: accidental override**

Subclass defines a method with the signature of an unrelated superclass method [Binder, accidental override]

```
class SimpleDirectory {  
    ...  
    /*  
     * Add a file with a given name.  
     */  
    void add ($tring file) {  
        ...  
    }  
}
```

SimpleDirectory
add(\$tring file)

↑

HierarchicalDirectory
add(\$tring subdirectory)

Page 55 AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

 **Violation of subtyping rules: accidental override**

```
class HierarchicalDirectory  
    extends SimpleDirectory {  
    ...  
    /*  
     * Add a subdirectory with a given name.  
     */  
    void add ($tring subdirectory) {  
        ...  
    }  
}
```

SimpleDirectory
add(\$tring file)

↑

HierarchicalDirectory
add(\$tring subdirectory)

Page 56 AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

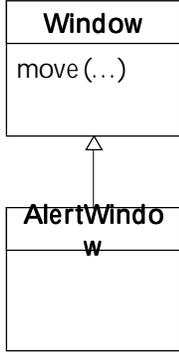
FAA National Software Conference, May 2002

Object Oriented Guidelines

**Violation of subtyping rules: missing override**

Inherited superclass method fails to maintain subclass invariant [Binder, missing override]

```
interface Window {  
    ...  
    /**  
     * Moves a window to a particular position  
     * on the screen.  
     */  
    void move (int x, int y);  
}
```



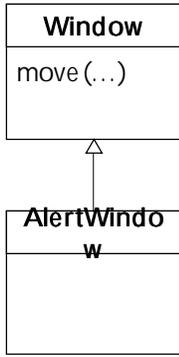
```
classDiagram  
    class Window {  
        move(...)  
    }  
    class AlertWindow {  
        w  
    }  
    Window <|-- AlertWindow
```

Page 57

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

**Violation of subtyping rules: missing override**

```
/**  
 * invariant: Window appears in front of all others  
 */  
interface AlertWindow extends Window {  
    ...  
}
```



```
classDiagram  
    class Window {  
        move(...)  
    }  
    class AlertWindow {  
        w  
    }  
    Window <|-- AlertWindow
```

Page 58

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.3: Method extension

The sole exception to the simple dispatch rule

Permits the implementation of a subclass method in terms of its *corresponding* superclass method

Commonly used to define subclass constructors in terms of superclass constructors, but can be used to extend the implementation of any method



Pattern 4.3: Method extension

Method extension rule

To extend the functionality of an inherited method, the subclass method should explicitly call the inherited version of the same operation, followed by additional code that extends the overall effect (postcondition).

The explicit call must be to the corresponding superclass version of the same method and must be statically bound.

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.3: Method extension

Extension of a method's implementation in a subclass

```
class A {
    void m (int a) {
        //do something
    }
}
class B extends A {
    void m (int a) {
        super.m(a);
        //do something more
    }
}
```

Page 61

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Pattern 4.3: Method extension

We do not allow a method to extend a superclass method with a *different* signature in this way because:

- This can lead to a confusing situation if the other method is later overridden
- There is no need to do so; we can just call the inherited version of the other method

Page 62

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.4: Class coupling

5 rules + general guidance

Addresses DO-178B concerns with coupling

Addresses coupling between client and classes

Addresses coupling between superclasses and subclasses

Encourages data hiding and hardware abstraction (which supports partitioning and reduces the cost of future likely changes)

Supports the enforcement of key class and system invariants (which may have a direct impact upon safety)

Page 63

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Pattern 4.4: Class coupling

1. 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.

Page 64

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.4: Class coupling

2. Client hardware abstraction rule

Clients should access any hardware abstracted by the class only through its 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.

Page 65

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Pattern 4.4: Class coupling

3. Invariant rule

The invariant for the class should be a part of the postcondition of every class constructor, a part of the precondition of the class destructor (if any), and a part of the precondition and postcondition of every other publicly accessible operation.

And clients should be able to influence the value of the invariant only through execution of these operations.

Page 66

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.4: Class coupling

4. Subclass data abstraction rule

Subclasses should access the data representation of the class only through its 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.



Pattern 4.4: Class coupling

5. Subclass hardware abstraction rule

Clients can access any hardware abstracted by the class only through its 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.

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.5: Multiple interface inheritance

3 rules + general guidance

Addresses issues related to ambiguity and intent.

1. Repeated interface inheritance rule
2. Interface redefinition rule
3. Independent interface definition rule

Page 69

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Background: Multiple inheritance

- Inheritance was originally viewed primarily as a mechanism for sharing code and data definitions
- In this context, multiple inheritance was viewed as a mechanism for constructing a subclass implementation from multiple superclass implementations

Page 70

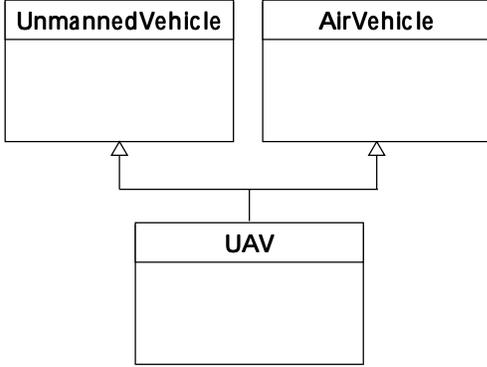
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

**Background: Multiple interface inheritance**

- With an increased emphasis on *interface inheritance* during analysis and design, multiple inheritance is now used primarily as a means of classifying entities that logically belong to more than a single category



```
classDiagram
    class UAV
    class UnmannedVehicle
    class AirVehicle
    UAV --|> UnmannedVehicle
    UAV --|> AirVehicle
```

Page 71

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

**Background: Multiple interface inheritance**

- As a result, languages such as Java only support multiple inheritance involving *interfaces*
- And rely on *delegation* to achieve the effects of multiple *implementation* inheritance

Page 72

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

Inheritance: Using symbols to represent operations

B inherits the operations of A
It may also override them Δ
And may add operations of its own

- \square m (a: int)
- Δ n (x: int) : float
- \heartsuit o ()
- Δ n (a: int): float

Page 73
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

1. Independently defined ...

Independently defined operations with the same signature

Page 74
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

Inheritance “math”

?

Page 75AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

LSP

- Signature is the same
- Pre, weaker than either or
- Post, stronger than both and

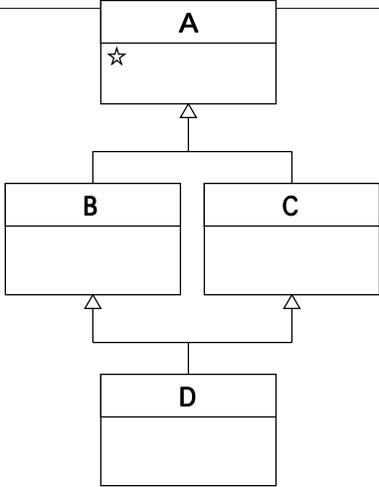
Page 76AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



2. Repeated inheritance



```
classDiagram
    class A {
        ☆
    }
    class B
    class C
    class D
    A <|-- B
    A <|-- C
    D <|-- B
    D <|-- C
```

Same element "☆" is inherited along more than one path

Page 77AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Inheritance "math"









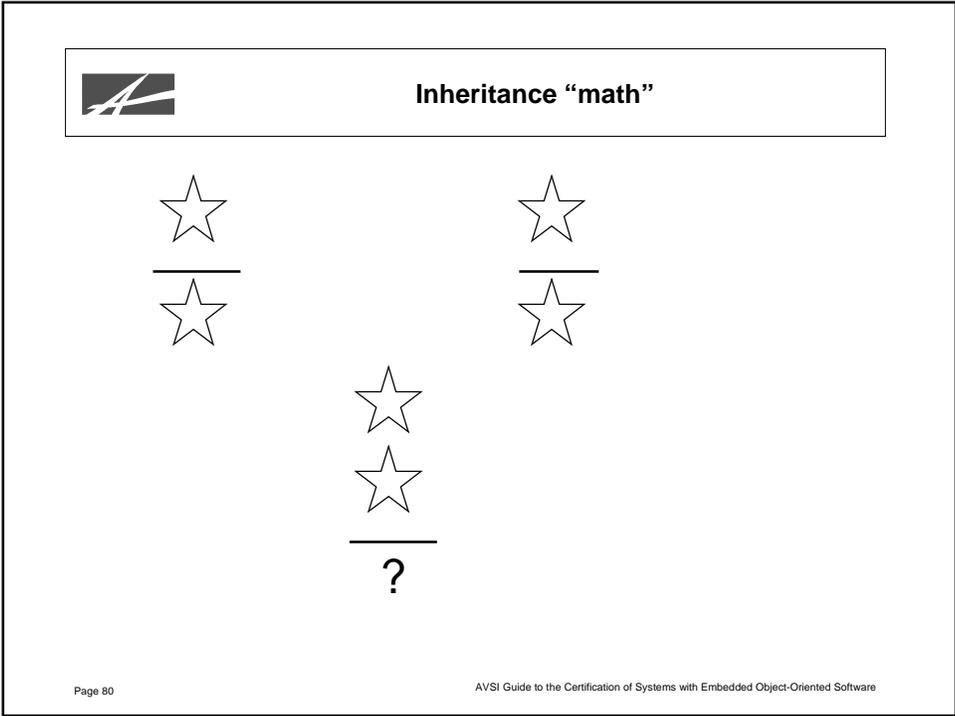
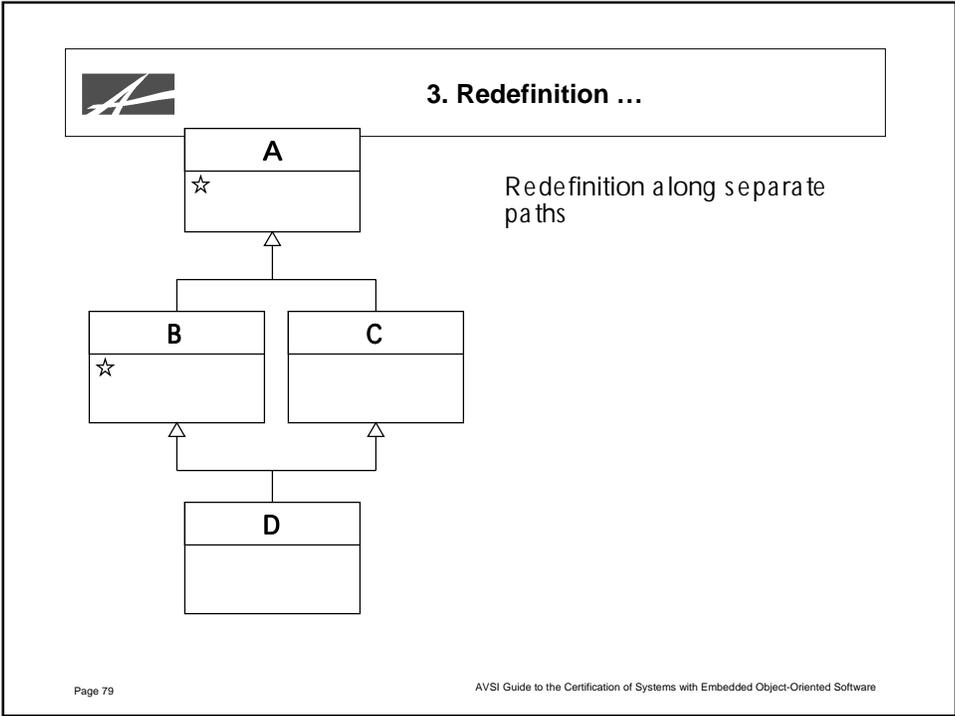





Page 78AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



FAA National Software Conference, May 2002

Object Oriented Guidelines

LSP

- Signature is the same
- Pre, weaker than either ☆ or ☆
- Post, stronger than both ☆ and ☆

Page 81
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

Pattern 4.5: Multiple interface inheritance

1. Repeated interface inheritance rule

When the same operation is inherited by an interface via more than one path through the interface hierarchy, this should result in a single operation in the subinterface.

Makes no sense to provide two identical operations to clients - so just one.

When the same operation is inherited by an interface via more than one path through the interface hierarchy, this should result in a single operation in the subinterface.

Makes no sense to provide two identical operations to clients - so just one.

```

classDiagram
    class A {
        ☆
    }
    class B
    class C
    class D
    A <|-- B
    A <|-- C
    B <|-- D
    C <|-- D
            
```

Page 82
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

Pattern 4.5: Multiple interface inheritance

Repeated interface inheritance

```

interface A {
    /*
     * pre: some precondition
     * post: some postcondition
     */
    int f(int p, int q) throws Exception;
}

interface B extends A {}
interface C extends A {}
interface D extends B, C {}
                
```

```

classDiagram
    class A {
        *
    }
    class B
    class C
    class D
    A <|-- B
    A <|-- C
    B <|-- D
    C <|-- D
                
```

Page 83
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

Pattern 4.5: Multiple interface inheritance

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* with respect to each parent interface.

To ensure that the client interface is always stated explicitly, and to double check intent.

```

classDiagram
    class A {
        *
    }
    class B {
        *
    }
    class C
    class D {
        *
    }
    A <|-- B
    A <|-- C
    B <|-- D
    C <|-- D
                
```

Page 84
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

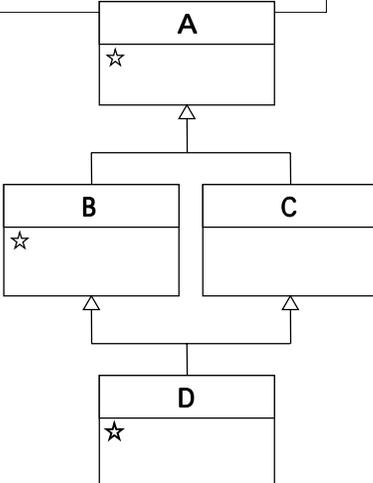

Pattern 4.5: Multiple interface inheritance

Interface redefinition and join

```

interface A {
  /**
   * pre: some precondition
   * post: some postcondition
   */
  int f(int p, int q) throws Exception;
}

interface B extends A {
  /**
   * @override
   * pre: weaker precondition, B.pre
   * post: stronger postcondition, B.post
   */
  int f(int p, int q) throws MyException;
}
        
```



Page 85
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

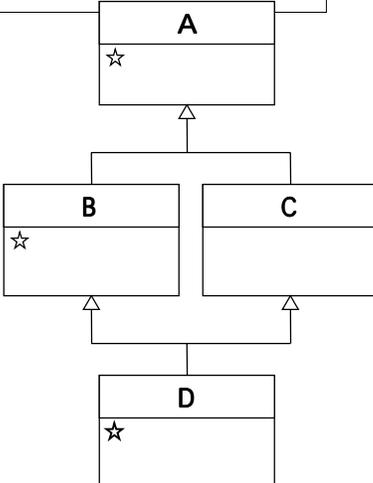

Pattern 4.5: Multiple interface inheritance

Interface redefinition and join

```

interface C extends A {}

interface D extends B, C {
  /**
   * @Join
   * pre: B.pre
   * post: B.post
   */
  int f(int p, int q) throws MyException;
}
        
```



Page 86
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.5: Multiple interface inheritance

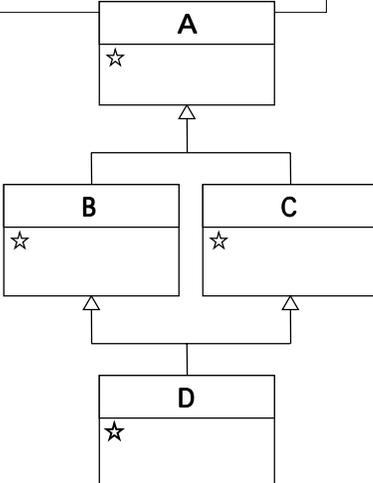
Interface redefinition and join, 2nd example

```

interface A {
    /**
     * pre: some precondition
     * post: some postcondition
     */
    int f(int p, int q) throws Exception;
}

interface B extends A {
    /**
     * @override
     * pre: weaker precondition, B.pre
     * post: stronger postcondition, B.post
     */
    int f(int p, int q) throws MyException;
}
                    
```

Page 87



```

classDiagram
    class A {
        *
    }
    class B {
        *
    }
    class C {
        *
    }
    class D {
        *
    }
    A <|-- B
    A <|-- C
    B <|-- D
    C <|-- D
                    
```

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Pattern 4.5: Multiple interface inheritance

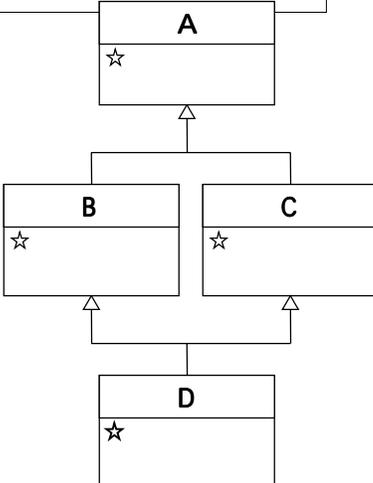
Interface redefinition and join, 2nd example

```

interface C extends A {
    /**
     * @override
     * pre: weaker precondition, C.pre
     * post: stronger postcondition, C.post
     */
    int f(int p, int q) throws MyException;
}

interface D extends B, C {
    /**
     * @join
     * pre: B.pre or C.pre
     * post: B.post and C.post
     */
    int f(int p, int q) throws MyException;
}
                    
```

Page 88



```

classDiagram
    class A {
        *
    }
    class B {
        *
    }
    class C {
        *
    }
    class D {
        *
    }
    A <|-- B
    A <|-- C
    B <|-- D
    C <|-- D
                    
```

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

Pattern 4.5: Multiple interface inheritance

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.

Errors should be caught by normal testing.

```

classDiagram
    class A {
        ☆
    }
    class B {
        ☆
    }
    class C
    C --|> A
    C --|> B
            
```

Page 89
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

Pattern 4.5: Multiple interface inheritance

Independent definition and join

```

interface A {
    /**
     * pre: some precondition
     * post: some postcondition
     */
    int f(int p, int q) throws MyException;
}

interface B {
    /**
     * pre: the same precondition
     * post: the same postcondition
     */
    int f(int p, int q) throws MyException;
}
            
```

```

classDiagram
    class A {
        ☆
    }
    class B {
        ☆
    }
    class C {
        ☆
    }
    C --|> A
    C --|> B
            
```

Page 90
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

Pattern 4.5: Multiple interface inheritance

Independent definition and join

```

interface C extends A, B {
    /**
     * @join
     * pre: the same precondition
     * post: the same postcondition
     */
    int f(int p, int q) throws MyException;
}
                    
```

```

classDiagram
    class A {
        *
    }
    class B {
        *
    }
    class C {
        *
    }
    C --|> A
    C --|> B
                    
```

Page 91
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

Pattern 4.5: Multiple interface inheritance

Independent definition and join, 2nd example

```

interface A {
    /**
     * pre: some precondition
     * post: some postcondition
     */
    int f(int p, int q) throws Exception;
}

interface B {
    /**
     * pre: some other precondition
     * post: some other postcondition
     */
    int f(int p, int q) throws MyException;
}
                    
```

```

classDiagram
    class A {
        *
    }
    class B {
        *
    }
    class C {
        *
    }
    C --|> A
    C --|> B
                    
```

Page 92
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

 **Pattern 4.5: Multiple interface inheritance**

Independent definition and join, 2nd example

```
interface C extends A, B {
  /*
   * @join
   * pre: A.pre or B.pre
   * post: A.post and B.post
   */
  int f(int p, int q) throws MyException;
}
```

A
☆

B
☆

C
☆

Page 93AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

 **Pattern 4.6: Multiple implementation inheritance**

3 rules + general guidance
Addresses issues related to ambiguity and intent.

1. Repeated implementation inheritance rule
2. Implementation redefinition rule
3. Independent implementation definition rule

The general guidance recommends use of multiple *implementation* inheritance only for level D software, and in accordance with the above rules.

Delegation is the recommended work around in Java and C++.

The Ada Rationale offers a standard work around for Ada95.

Page 94AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Background: Multiple implementation inheritance

- Involves the composition of *competing* implementations developed along separate paths (rather than the extension of a single superclass implementation along a single path)
- Involves the composition of executable elements (rather than interface specifications)
- Involves the composition of elements that reference one another (sometimes in subtle ways)

Page 95

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Background: Multiple implementation inheritance

- May introduce dead or deactivated code and data referenced only by overridden (unchosen) implementations
- Is difficult to implement well (C++ vs. Eiffel)
- Has an acceptable work around (delegation)

Page 96

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

Background: Delegation

Delegation of client calls to B::m(int) to A::m(int)

```

classDiagram
    class A {
        m(a: int)
    }
    class B {
        m(a: int)
    }
    A <|-- B
    B o-- A
        
```

```

class A {
    ...
    int m(int a) { ... }
}

class B {
    private A delegate;
    ...
    public int m(int a) {
        return delegate.m(a);
    }
}
        
```

Page 97
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

Background: Delegation as a work around for MI

```

classDiagram
    class A
    class B
    class C
    C <|-- A
    C o-- B
        
```

Single inheritance + delegation

Inheritance of the implementation of one superclass

Delegation to the implementation of another

Page 98
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

Pattern 4.6: Multiple implementation inheritance

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 (by default) result in a single feature in the subclass.

```

classDiagram
    class A {
        +square
        +star
        +triangle
    }
    class B {
        +triangle
    }
    class C {
        +triangle
    }
    class D {
        +triangle
    }
    A <|-- B
    A <|-- C
    B <|-- D
    C <|-- D
        
```

Page 99
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

Pattern 4.6: Multiple implementation inheritance

Repeated implementation inheritance with sharing

```

class A {
private:
    int a;
public:
    /**
     * pre: some precondition
     * post: some postcondition
     */
    int f (int p, int q); //references a, calls n
protected:
    /**
     * pre: some other precondition
     * post: some other postcondition
     */
    float n (int x);
}
        
```

```

classDiagram
    class A {
        +square
        +star
        +triangle
    }
    class B {
        +triangle
    }
    class C {
        +triangle
    }
    class D {
        +triangle
    }
    A <|-- B
    A <|-- C
    B <|-- D
    C <|-- D
        
```

Page 100
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

Pattern 4.6: Multiple implementation inheritance

```

class B: public virtual A {
protected:
  /**
   * @override
   */
  float n (int x);
}
class C: public virtual A {}
class D: public B, public C {
protected:
  /**
   * @join
   */
  float n (int x) { ... }
}

```

```

classDiagram
    class A {
        +
        *
        Δ
    }
    class B {
        Δ
    }
    class C {
    }
    class D {
        Δ
    }
    A <|-- B
    A <|-- C
    B <|-- D
    C <|-- D

```

Page 101
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

Pattern 4.6: Multiple implementation inheritance

Repeated implementation inheritance with replication

```

class A {
private:
  int a;
public:
  /**
   * pre: some precondition
   * post: some postcondition
   */
  int f (int p, int q); //references a, calls n
protected:
  /**
   * pre: some other precondition
   * post: some other postcondition
   */
  float n (int x);
}

```

```

classDiagram
    class A {
        +
        *
        Δ
    }
    class B {
        Δ
    }
    class C {
    }
    class D {
        Δ
    }
    A <|-- B
    A <|-- C
    B <|-- D
    C <|-- D

```

Page 102
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

Pattern 4.6: Multiple implementation inheritance

```

class B: public A {
protected:
    /**
     * @override
     */
    float n (int x);
}
class C: public A {}
class D: public B, public C { #@replicate A
public:
    /**
     * @join
     */
    int f (int p, int q) { ... }
protected:
    /**
     * @join
     */
    float n (int x) { ... }
}
                
```

Page 103
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

Pattern 4.6: Multiple implementation inheritance

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* with respect to each parent class.

Page 104
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

Pattern 4.6: Multiple implementation inheritance

Implementation redefinition

```

class A {
private:
    int a;
public:
    /**
     * pre: some precondition
     * pos t: some pos tcondition
     */
    int f (int p, int q); //references a, calls n
protected:
    /**
     * pre: some other precondition
     * pos t: some other pos tcondition
     */
    float n (int x);
}
                
```

Page 105
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

Pattern 4.6: Multiple implementation inheritance

```

class B: public virtual A {
public:
    /**
     * @ override
     */
    int f (int p, int q); s till references a, calls n
protected:
    /**
     * @ override
     */
    float n (int x);
}
                
```

Page 106
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

Pattern 4.6: Multiple implementation inheritance

```

class C: public virtual A {
private:
    int b;
public:
    /**
     * @override
     */
    int f (int p, int q); //references b
}
        
```

The diagram shows class A at the top with a square icon, a star, and a triangle. Below it are classes B and C. Class B has a star and a triangle. Class C has a diamond and a star. Class D is at the bottom, inheriting from both B and C. D has a star and a triangle.

Page 107
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

Pattern 4.6: Multiple implementation inheritance

```

class D: public B, public C {
public:
    /**
     * @join
     */
    int f (int p, int q) { ... }
protected:
    /**
     * @join
     */
    float n (int x) { ... }
}
        
```

The diagram shows class A at the top with a square icon, a star, and a triangle. Below it are classes B and C. Class B has a star and a triangle. Class C has a diamond and a star. Class D is at the bottom, inheriting from both B and C. D has a star and a triangle.

Page 108
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

Pattern 4.6: Multiple implementation inheritance

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 an overriding method should be explicitly defined in the subclass to combine them.

```

classDiagram
    class A {
        □
        ☆
        △
    }
    class B {
        ◇
        ☆
        ☆☆
    }
    class C {
        ☆
    }
    A <|-- C
    B <|-- C
        
```

Page 109
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

Pattern 4.6: Multiple implementation inheritance

Independent implementation inheritance

```

class A {
private:
    int a;
public:
    /**
     * pre: some precondition
     * pos t: some pos tcondition
     */
    int f (int p, int q); //references a, calls n
protected:
    /**
     * pre: some other precondition
     * pos t: some other pos tcondition
     */
    float n (int x);
}
        
```

```

classDiagram
    class A {
        □
        ☆
        △
    }
    class B {
        ◇
        ☆
        ☆☆
    }
    class C {
        ☆
    }
    A <|-- C
    B <|-- C
        
```

Page 110
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

Pattern 4.6: Multiple implementation inheritance

```

class B {
private:
    int b;
public:
    /**
     * pre: some precondition
     * pos t: some pos tcondition
     */
    int f (int p, int q); //references b, calls z
protected:
    /**
     * pre: some other precondition
     * pos t: some other pos tcondition
     */
    float z 0;
}
                
```

Page 111
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

Pattern 4.6: Multiple implementation inheritance

```

class C: public A, public B {
public:
    /**
     * @Join
     */
    int f (int p, int q) { ... }
}
                
```

Page 112
AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.7: Re-usable Components

A process pattern rather than a design pattern

5 guidelines

Addresses the reuse of certification artifacts in general

Addresses component requirements and system requirements traceability

Addresses the testing of re-usable components, e.g., in contexts where not all operations are used

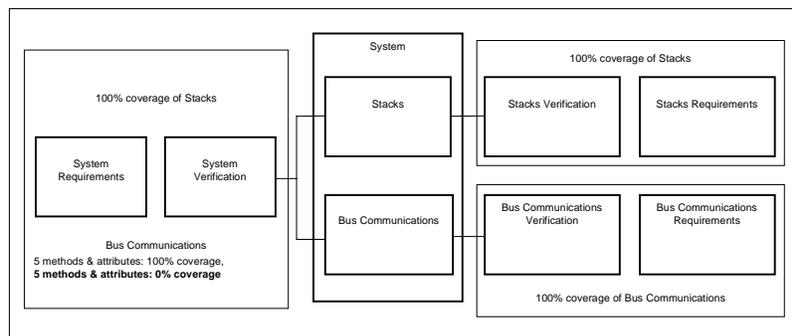
Page 113

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Reuse/retest issues

"The reuse of software across products or systems raises several issues, especially in the area of re-test of the software." [AVSI Guide, p. 29]



Page 114

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines

Reuse/retest issues

“At the application level, five of these are used and can be shown to be tested at the system level. The other five are deactivated for this application. However, these five requirements are still tested by re-executing the test cases that were previously developed specifically for the component.” [AVSI Guide, p. 29]

The diagram illustrates the relationship between system-level components and their verification and requirements artifacts. It is organized into three main sections:

- System Level (Left):** Contains 'System Requirements' and 'System Verification' boxes. Below them is a box for 'Bus Communications' with the text: '5 methods & attributes: 100% coverage, 5 methods & attributes: 0% coverage'.
- System Component (Center):** A box labeled 'System' containing 'Stacks' and 'Bus Communications' sub-components.
- Verification and Requirements (Right):** Two groups of boxes. The top group, labeled '100% coverage of Stacks', contains 'Stacks Verification' and 'Stacks Requirements'. The bottom group, labeled '100% coverage of Bus Communications', contains 'Bus Communications Verification' and 'Bus Communications Requirements'.

Lines connect the 'System' box to the 'Stacks' and 'Bus Communications' sub-components, and further to their respective verification and requirements artifacts.

Page 115 AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

Pattern 4.7: Re-usable Components

Component certification package

“Each component must have a complete, **reusable certification package**, just as the software itself is intended to be reusable.”

“The component must contain **all of the artifacts** needed to certify at or above the software level of the application.”

“This should include the certification **planning data** that were originally used to develop the component. This planning data may differ from the plans used to develop the application code in the final systems; however, these plans must be submitted to certification agencies each time a system application is made. The **PSAC** of the final system should reference this planning data.”

Page 116 AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.7: Re-usable Components

Component requirements specification

"Each component must have its own complete set of requirements. This should include a high-level SW requirement specification and low-level requirements, as these are defined in DO-178B, along with any architectural considerations."

"It is important that requirements **specify the required interfaces** unambiguously, along with any underlying assumptions in the architecture, so that it is clear to the application developers whether the component is fit for use in the intended application."

"A clear set of requirements **will help later application developers avoid thinking that some of the software in the component is dead code.**"

Page 117

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Pattern 4.7: Re-usable Components

System requirements traceability and the identification of deactivated code/data

"There must be some appropriate connection in the trace matrix between the application's high-level software requirements and the appropriate high-level requirements of the component."

"In some cases, this may be a direct trace ... However, other lower-level requirements may be derived."

"**The parts of the component not used in the applications should be indicated as unused somewhere in the application data.**"

Page 118

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.7: Re-usable Components

Component testing

"Each component must have tests written which verify the code against all of the component requirements."

"Application requirements that trace to component requirements may make use of these test cases to show coverage. Structural coverage data must also have been collected to show 100% coverage of the code or include an analysis if collection did not achieve 100%."



Pattern 4.7: Re-usable Components

Component testing

"DO-178B requires tests to be run on the final platform or equivalent simulation environment. Therefore, if the final object code of the component software has not been previously tested on the final or equivalent platform, its test cases must be re-executed on that platform."

"If the bit-pattern of the object code of the component has not changed, there is no need to re-collect the structural coverage data, as the paths have not changed. If the object code of the component is changed in some way (e.g., different compiler, different compiler version, different optimization options), structural coverage data should be re-collected as the compiler may have introduced new paths into the object code"

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.7: Re-usable Components

Structural coverage of unused (deactivated) code

"The Structural Coverage Analysis Resolution rules for **Deactivated code** in section 6.4.4.3 of DO-178B apply to the component functions that are not intended to be executed."

Page 121

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Pattern 4.8: Template classes and template operations

5 rules + general guidance

Addresses the testing of individual instantiations

Addresses situations where the use of templates is problematic

Addresses issues related to the manner in which templates are instantiated

Provides a consistent view of template instantiation and inline expansion

Page 122

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.8: Template classes and template operations

Templates should be instantiated and tested with each type argument to parameter binding in the system unless:

1. the types map to the same underlying representation
2. *and* the object code can be shown to be equivalent



Pattern 4.8: Template classes and template operations

Nested templates, templates with child packages (Ada), and templates with friend classes (C++) should be prohibited for levels A, B, and C.

Formal "inout" should be prohibited for levels A, B, and C.

Templates should be compiled using "macro-expansion" rather than "code sharing".

For macro-expanded templates, the guidelines for inlining should be followed inasmuch as they apply.

FAA National Software Conference, May 2002

Object Oriented Guidelines



Pattern 4.8: Template classes and template operations

Object code equivalence

Equivalence implies that no object code has been added or removed between the two versions of object code, although base addresses and references to constants may differ.

For example, if the op-codes (e.g., 32-bit instructions for copy/move/etc. in all versions) and the code sequence are the same, and the stack frames are the same size and have the same offsets (base addresses can differ) then equivalence can be shown.

Control variables and constants can be different, but should be shown to be of the same size and usage should be shown to be consistent. This definition of equivalence, however, is not intended to be complete.

Page 125

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Section 3.1 Notes On Inlining

5 notes

Address the impact of inlining on:

- data and control flow analysis (FA)
- stack usage and timing analysis (SU & TA)
- source code to object code traceability
- structural coverage analysis (SC)

Page 126

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Section 3.1 Notes On Inlining

Impact of inlining on data and control flow analysis

"Flow Analysis, recommended for levels A-C, is impacted by Inlining (just what are the data coupling and control coupling relationships in the executable code?)."

"The data coupling and control coupling relationships can transfer from the inlined component to the inlining component."

As a result, "data coupling and control coupling should take into account the inlining of code including call tree and data set/use analysis."

Page 127

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software



Section 3.1 Notes On Inlining

Impact of inlining on stack usage and timing analysis

"Since inline expansion can eliminate parameter passing, this can effect the amount of information pushed on the stack as well as the total amount of code generated."

"This, in turn, can effect the stack usage and the timing analysis."

As a result, "s tack usage and timing analysis should take into account the inlining of code."

Page 128

AVSI Guide to the Certification of Systems with Embedded Object-Oriented Software

FAA National Software Conference, May 2002

Object Oriented Guidelines



Section 3.1 Notes On Inlining

Impact of inlining on structural coverage analysis

"For inline expansion in level A software, source code should be traced to object code at each point of expansion."

"Inline expansion may not be handled identically at different points of expansion."

"This can be especially true when inlined code is optimized."

As a result, "if object code is removed or object code is added, as determined by the source to object code trace, then structural coverage must be verified separately for each expansion."

And, "structural coverage tools [may] need to know what will be inlined and what will not be inlined when inlining is requested."