Object-Oriented Programming for High-Integrity Systems: *Pitfalls and How to Avoid Them*

SSTC 2012
Salt Lake City, Utah

Track 1
Monday, April 23, 2012
3:30 – 4:15 pm

Ben Brosgol • brosgol@adacore.com

North American Headquarters:
104 Fifth Avenue, 15th Floor
New York, NY 10011
USA
+1-212-620-7300 (voice)
+1-212-807-0162 (FAX)

European Headquarters:
46 rue d'Amsterdam
75009 Paris
France
+33-1-4970-6716 (voice)
+33-1-4970-0552 (FAX)

www.adacore.com

Outline

Summary of basic concepts

- High-Integrity system ⇨ stringent safety and/or security requirements
  - Software must be *reliable*, and also *analyzable* to demonstrate relevant safety and/or security properties
  - Compliance with domain-specific certification standard may be required
    - Examples: DO-178C/ED-12C (commercial avionics), Common Criteria (security)
  - At the highest levels, need very high degree of assurance
    - Use of formal methods may be relevant or mandatory
- Object Orientation
  - Development methodology that can ease the maintenance of large complex systems
  - “Programming by extension”
    - Object-Oriented Programming (OOP)
    - Object-Oriented Technology (OOT)

Challenges presented by OOT

- The flexibility and dynamic tailorability of OOT present a variety of vulnerabilities and conflict with the need to statically demonstrate a program’s assurance properties
- Virtual machine execution platform (e.g., JVM) blurs distinction between code and data

Addressing the challenges

- DO-332/ED-217: “OOT and Related Techniques” Supplement to DO-178C/ED-12C

Object-Oriented Technology ("OOT")

What is OOT?

- Software development methodology supported by language features
  - Primary focus is on data elements and their relationships
  - Secondary focus is on the processing that is performed
- Applicable during entire software life cycle

Language concepts ("Object-Oriented Programming", or "OOP")

- *Object* = state ("attributes") + operations ("methods")
- *Class* = module + object creation template
- *Encapsulation* = separation of interface (spec for methods) from implementation (state, algorithms)

Object-Oriented Design ("OOD") also known as *Object-Based Programming*

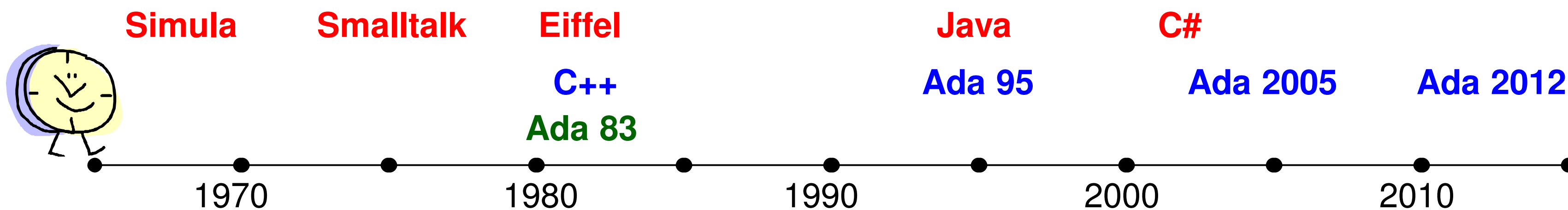
- *Inheritance* has two perspectives
  - *Type theoretic/ data modeling:* specialization ("is-a") relationship between classes
  - *Programming language feature:* Extend a class, adding new state and adding/ overriding operations
- *Polymorphism* = ability of a variable to reference objects from different classes at different times
- *Dynamic binding ("dispatching")* = interpretation of operation applied to polymorphic variable based on current class of referenced object

Object-Oriented Technology (“OOT”), cont’d.

Additional OOP elements

- Single versus multiple inheritance
  - “Interface” for a simple form of multiple inheritance
  - Implementation inheritance versus interface inheritance
- Use of constructors / finalizers

Timeline of OO language history (sampler)



Related language features

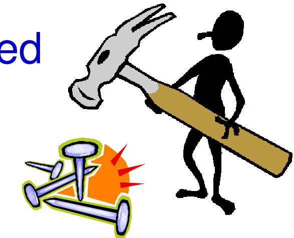
- Method overloading
- Type conversion
- Inline expansion
- “Contract-based programming”: preconditions, postconditions, type invariants, assertions
  - May be verified statically or checked dynamically
- Generic templates (“parametric polymorphism”)
- Exceptions

AdaCore

Object-Oriented Technology & High-Integrity Systems

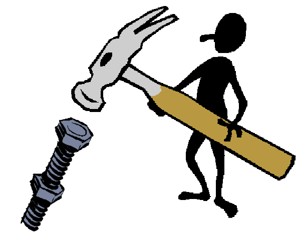
Why consider OOT for High-Integrity software?

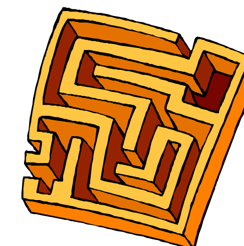
- Data-centric approach, encapsulation ease maintenance of many large systems
- Model-driven architecture / UML tools may generate OO code to be certified
- Many programmers know OO languages such as C++, Java, or Ada 95
- Languages (Ada, C++) used for safety-critical systems have OO features
- May want to take OO legacy code and certify *à posteriori*

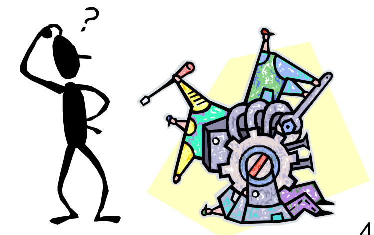


What's the catch?

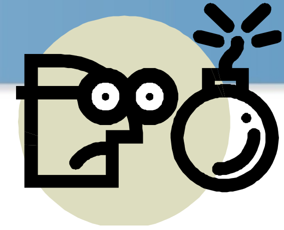
- Paradigm clash
  - OOT's distribution of functionality across classes, vs. certification's focus on tracing between requirements and implemented functions
- Technical issues (both reliability and analyzability)
  - Explored at NASA/FAA *Object-Oriented Technology in Aviation* (OOTiA) workshops
  - Addressed by *OOT and Related Techniques* supplement to DO-178C, and described below
- Cultural issues
  - Evaluation personnel are generally not language experts and are (rightfully) concerned about how to deal with unfamiliar technology







Class Concept: Issues and Solutions



Unused code

- Override method, don’t instantiate superclass ⇨
  - Explain deactivated code (superclass method)
- Only invoke some operations from a given class ⇨
  - Use “smart linker” to avoid loading unused code
  - Explain deactivated code

Encapsulation

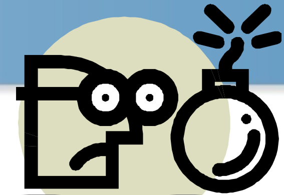
- Overhead from invoking accessor operations (“getter” / “setter”) versus directly accessing the fields ⇨
  - Inline expansion of simple operations

Overencapsulation

- Robustness tests can’t access / assign to encapsulated data ⇨
  - Use appropriate language mechanisms (C++ “friends”, Ada child/private packages)

AdaCore

Inheritance: Issues and Solutions



Design issue

- Incompatible subclass (not a specialization relationship) ⇨
  - Redesign class hierarchy
  - Program defensively

Discussed below
(“Liskov Substitution Principle”)

Programming / maintenance errors

- Unintended inheritance, unintended overriding ⇨
  - Language syntax / rules to make overriding intent explicit
- Unintended field name interpretation from ancestor class ⇨
  - Language syntax / rules to make overriding intent explicit
- Failure to override when subclass adds new state (i.e., state not properly initialized) ⇨
  - Detect during source code review/analysis or testing

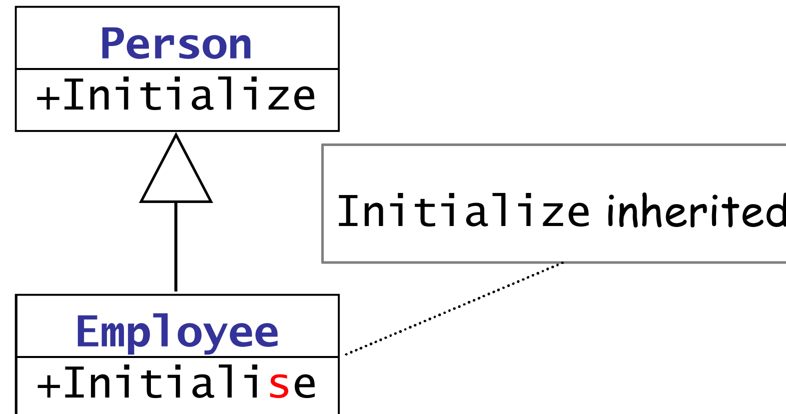
Multiple inheritance

- Multiple interface inheritance: may have incompatible signatures, or signatures for operations with different “contracts” ⇨
  - Apply language-specific solution, or revise design
- Multiple implementation inheritance: complexity of “diamond” inheritance ⇨
  - Prevent via appropriate language features or coding standard
  - Detect during source code review/analyses

AdaCore

Example: Unintended Inheritance

Misspelling of method name in subclass

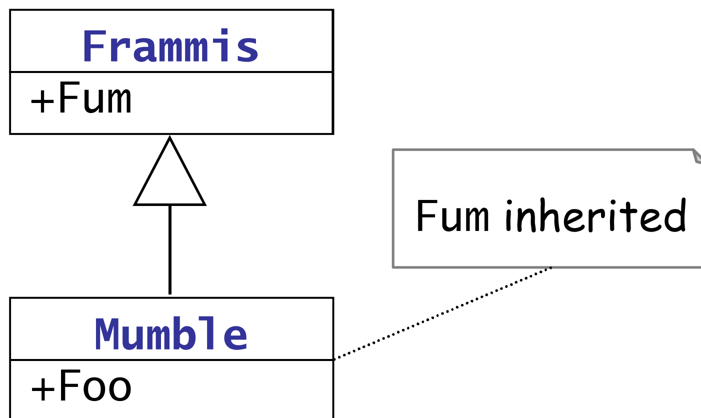


- Programmer meant to override `Initialize` but misspelled the name
- Superclass's `Initialize` is inherited rather than overridden
- Dynamic binding for `p.Initialize()`, when p (a polymorphic `Person`) references an `Employee`, invokes the `Initialize` method defined for `Person`
  - Serious program bug, since some new fields in `Employee` will not be initialized

AdaCore

Example: Unintended Overriding

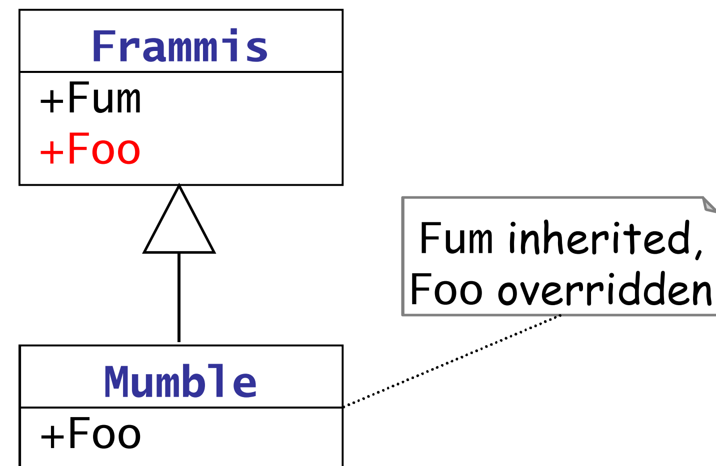
Initially:



- Method Foo is defined only for subclass `Mumble`

During maintenance:

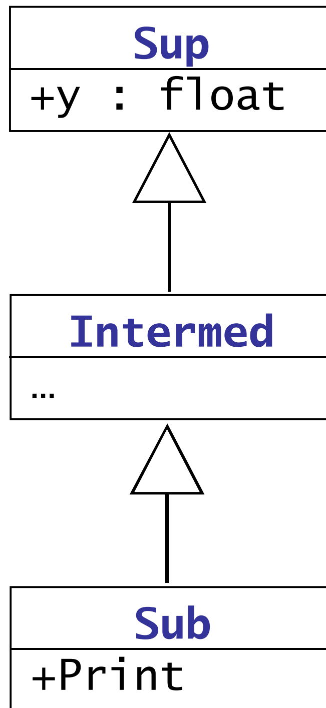
- Foo is added to superclass `Frammis`



- Dynamic binding for `p.Foo()`, where p is a polymorphic `Frammis` and references a `Mumble`, invokes the Foo method previously defined for `Mumble`
- This is not the intended effect

Example: Incorrect Interpretation of Field Name

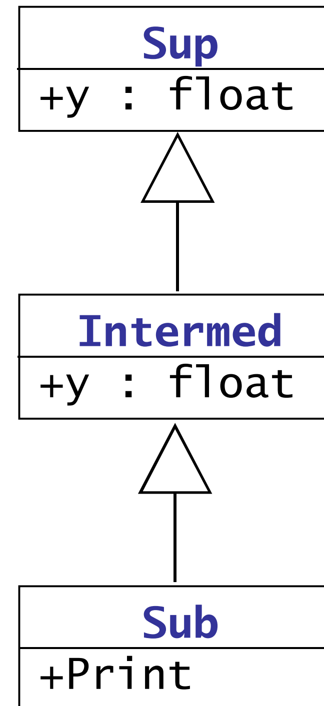
Initially:



- Method `Print` displays the value of field y, from class Sup

During maintenance:

- Field y is added to class `Intermed`

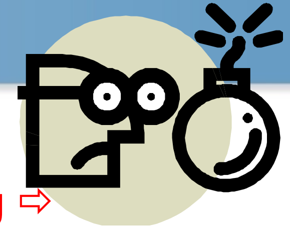


Some languages allow same name to be used for fields in different classes in same hierarchy

- Method `Print` now displays the value of y from class `Intermed`
- This is not the intended effect

AdaCore

Polymorphism: Issues and Solutions

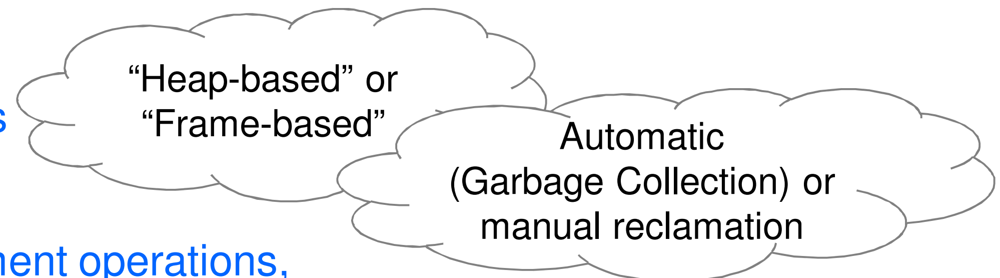


Reference semantics

- Assignment, parameter passing involve "pointer copy" and thus object sharing ⇨
  - Conduct code analysis to ensure no problems from aliasing

Memory management

- Use of dynamic allocation raises a variety of issues with respect to program correctness and resource predictability ⇨
  - Correctness of allocation routine (returns reference to new object)
  - No fragmentation
  - Sufficient space for allocated objects
  - No dangling references
  - Bounded time for memory management operations, non interference with application timing

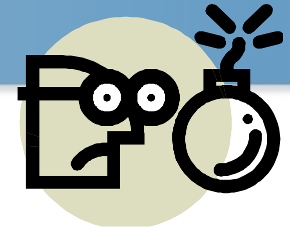


or

  - Avoid dynamic allocation by using "pool" of objects
    - Either statically reserved ("global") or dynamically allocated during system initialization

Dynamic Binding: Issues and Solutions

AdaCore



Programming errors

- Obtaining static binding when dynamic binding was intended, or vice versa ⇨
  - Prevent via language rules or detect during code review / testing

Software verification issues

- Resource usage – e.g., stack/heap space, WCET (Worst Case Execution Time) – complicated by dynamic binding ⇨
  - Need to consider all possible invocations
- Incompatible subclass (not a specialization relationship) can lead to invocation of operation that fails ⇨
  - Redesign class structure or add appropriate precondition checks
- Code coverage requirement complicated by dynamic binding ⇨
  - Perform "local type consistency verification"

Implementation insecurity

- Vtable uninitialized or corrupted ⇨
  - Implementation needs to protect vtable, e.g. store in ROM

Type Consistency in OOT

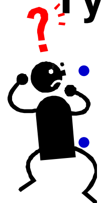
General principle: type consistency (subclass substitutability)

- Since inheritance is a specialization relationship, an instance of a subclass should be able to substitute for an instance of any of its superclasses, in any context
- If this principle is violated (by operation *Op* in subclass *S* of class *C*), then applying *Op* to a polymorphic variable of type *C* may fail when the variable references an instance of *S*
- Type consistency is an OO program's manifestation of *Liskov Substitution Principle*

"Liskov Substitution Principle" (LSP)

- "Let *q(x)* be a property provable about objects *x* of type *T*. Then *q(y)* should be true for objects of type *S* where *S* is a subtype of *T*"

Type consistency / LSP requirements on an overriding operation



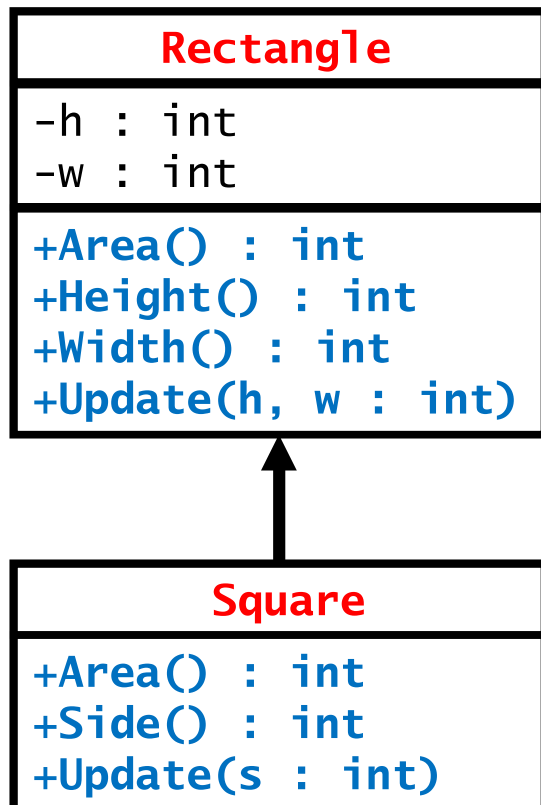
- Preconditions (of superclass's operation) must not be strengthened
- Postconditions (of superclass's operation) must not be weakened
  - Subclass operation must not throw exceptions that wouldn't be thrown by superclass
- Invariants (of superclass) must not be weakened

Paradox?

Two forms of type consistency

- **Global**: in each of the system's type hierarchies
- **Local**: in a context (variable, field, parameter) where substitution can occur
  - If subclass S of class C violates LSP for operation Op, but Op is never called on a C reference that denotes an S instance, then the violation doesn't matter

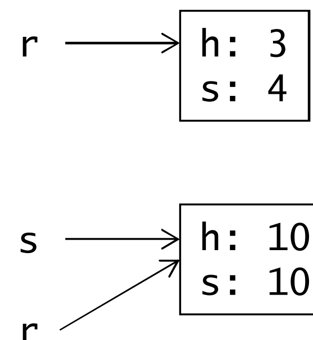
LSP Example: Is a Square a Rectangle? (1)




```
Rectangle r = new Rectangle();
Square s = new Square();
int j;
…
j = r.Area();
j = r.Height();
r.Update(3, 4);

j = s.Area();
j = s.Side();
s.Update(10);

r = s;
j = r.Area();        // Overriding version - OK
j = r.Height();      // Inherited version - OK
r.Update(10, 20);    // Inherited version - Oops
```



Problem: Inherited operation `Update(h, w : int)` violates LSP and corrupts the object

- Its implicit precondition h==w in class `Square` is stronger than the *true* precondition for the operation in class `Rectangle`

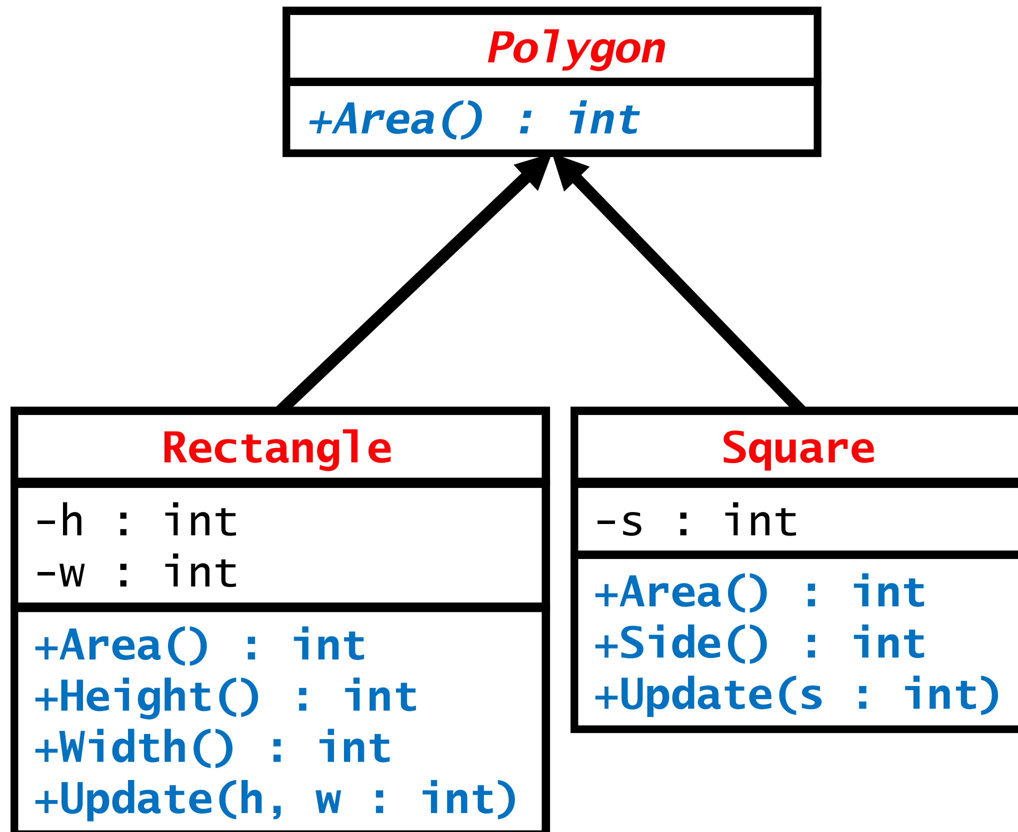
Solutions:

- Modify classes to comply with LSP
    - Make Rectangle and Square "immutable" (i.e, remove the Update operations)
    - Redesign the class hierarchy (Rectangle and Square inherit from a common base class)
- Override `Update(h,w:int)` in `Square` so that it throws an exception if `h!=`w

AdaCore

LSP Example: Is a Square a Rectangle? (2)

Solution 1: redesign class hierarchy

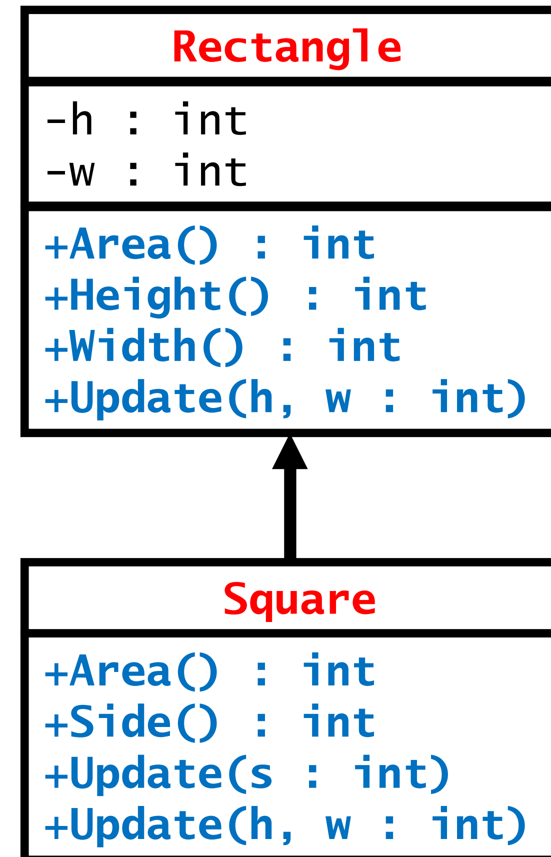


Notes:

- `Polygon` is an abstract class, `Area` is an abstract operation (indicated by italics)
- LSP is preserved by this hierarchy

Solution 2: override Update(h, w)

- Precondition: h==w



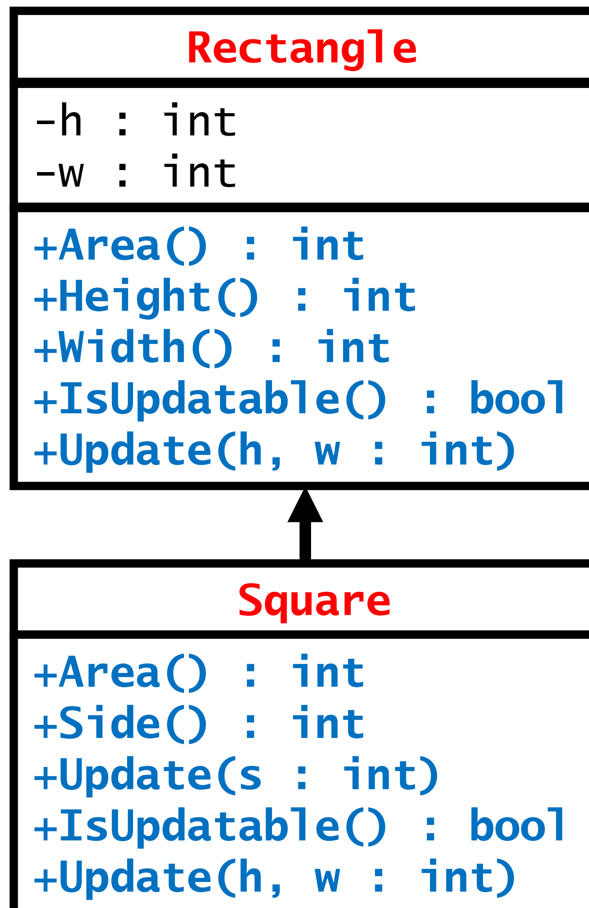
Notes:

- Precondition may be expressed with special syntax or explicit test
- Invoking `Square.Update(h, w)` when precondition not met violates LSP and should throw an exception

LSP Example: Is a Square a Rectangle? (3)

Solution 2':

- As in 2, override Update(h, w)
- Supply a dispatching precondition IsUpdatable()



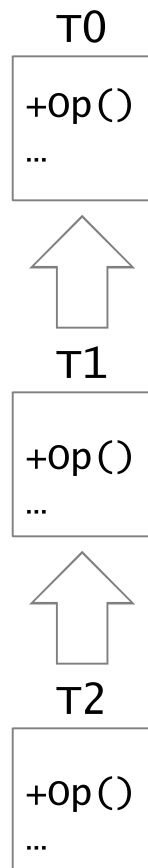
Notes

- r.IsUpdatable() returns true for Rectangle and r.h==r.w for Square
- Stylistic convention: invoke r.IsUpdatable() before each invocation of r.Update(h, w)
  - ```
    if (r.Updatable)
        {r.Update(h, w);}
    ```
- This style preserves LSP (or may be claimed to do so) but compromises OOP
- See Meyer's book (p. 576 ff) for further discussion
    ```

Dynamic Dispatch and Structural Coverage in DO-178C

Class hierarchy

- Op() overridden at each level



Application code

```
T0 p;  // polymorphic reference
p = …;
p.Op(); //dynamic dispatch
…
p.Op(); //dynamic dispatch
```

What is needed for full statement coverage?

- "Pessimistic" testing
  - Treat p.Op() as equivalent to a conditional statement that invokes the appropriate version of Op based on the type of the object referenced by p
  - Test every possible dispatch at each dispatching call
- "Optimistic" testing
  - Treat p.Op() as an indirect call through a Vtable
  - Make sure that each version of Op() is invoked in some dispatching call
  - This may fail to detect some errors if an overridden version of Op() does not satisfy its requirements in some context (specific dispatching call) that was not exercised
    - I.e., where subclass cannot substitute for the superclass

Rather than adapt definition of statement coverage, OOT Supplement adds a new objective based on subclass substitutability

AdaCore

New Guidance on Dynamic Dispatch

General scenario

- Each operation in a class is associated with a set of tests that verify its requirements
- An overridden method has an extended set of requirements (thus additional tests)

OO.6.5 Local Type Consistency Verification

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

OO.6.5.1 Local Type Consistency Verification Objective

- *Verify that all type substitutions are safe by testing or formal analysis.*

OO.6.5.2 Local Type Consistency Verification Activity

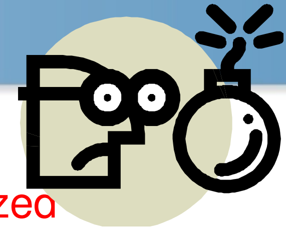
- *For each subtype where substitution is used, perform one of the following:*
  - *formally verify substitutability,*
  - *ensure that each class passes all the tests of all its parent types which the class can replace, or*
  - *for each call point, test every method that can be invoked at that call point (pessimistic testing).*

Implications of formal verification of substitutability

- For each subclass, run the extended tests that apply to that specific subclass
- No need to run the superclass tests

AdaCore

Other OOT Issues



Class initialization (constructors)

- Initialization order may lead to references to uninitialized or default-initialized fields ⇨
  - Prevent via coding conventions or detect during analysis/testing

Class finalization (destructors)

- Unspecified semantics if associated with garbage collection ⇨
  - Don’t use finalizers, or make no assumptions about finalizers in connection with garbage collection
- Unpredictability of execution time ⇨
  - Don’t use finalizers, or conduct thorough analysis to compute WCET in the presence of finalizers

Virtualization

- Code may be treated as data, resulting in compromised verification ⇨
  - Ensure that data treated as code is verified as executable code

Conclusions

OOT is "double-edged sword" for High-Integrity software

- Some elements help; e.g., encapsulation
- But analyzability and reliability problems arise from some of OOT's essential features
  - Inheritance
  - Polymorphism
  - Dynamic binding

Some issues can be addressed by language syntax/semantics

- Explicit syntax for specifying overriding versus non-overriding
- Clear rules for distinguishing static from dynamic binding
- Explicit support for "contract-based programming"

Don't forget related issues, including:

- Generics ("Parametric Polymorphism")
- Exceptions

In brief

- OOT will be seeing increasing usage in High-Integrity systems
- Addressing the pitfalls may depend on language features, run-time library implementation, static analysis tools, manual analysis, coding standard restrictions, and application style
- Read the *OOT and Related Techniques* supplement to DO-178C

References

B. M. Brosgol, *High-integrity object-oriented programming in Ada*, 3-part series published in EE Times, July-August 2011:

- http://www.eetimes.com/design/military-aerospace-design/4218039/High-Integrity-Object-Oriented-Programming-with-Ada
- http://www.eetimes.com/design/military-aerospace-design/4218257/High-integrity-object-oriented-programming-with-Ada---Part-2
- http://www.eetimes.com/design/military-aerospace-design/4218480/High-integrity-object-oriented-programming-with-Ada---Part-3

RTCA DO-178C/EUROCAE ED-12C, *Software Considerations in Airborne Systems and Equipment Certification*, December 2011

RTCA DO-332/EUROCAE ED-217, Object-Oriented Technologies and Related Techniques Supplement to DO-178C and DO-278A, December 2011.

FAA, *Handbook for Object-Oriented Technology in Aviation* (OOTiA), October 2004. www.faa.gov/aircraft/air_cert/design_approvals/air_software/oot

B. Webster, *Pitfalls of Object-Oriented Development*, M&T Books, 1995

B. Meyer, *Object-Oriented Software Construction (Second Edition)*, Prentice Hall; 1997

AdaCore

Acronyms and Abbreviations

JVM	Java Virtual Machine
OO	Object-Oriented
OOD	Object-Oriented Design
OOP	Object-Oriented Programming
OOT	Object-Oriented Technology
OOTiA	Object-Oriented Technology in Aviation
UML	Unified Modeling Language
WCET	Worst Case Execution Time

DO-178 (Not an acronym, “DO” is simply an abbreviation for “Document”)