AdaCore
The GNAT Pro Company

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

Object-Oriented Programming for High-Integrity Systems: *Local Type Consistency Verification without Tears*

STC 2013
Salt Lake City, Utah

Track 1
Wednesday, April 10, 2013
9:00 – 9:45 am

Ben Brosgol • brosgol@adacore.com

Overview

- Introduction / basics
  - High-Integrity Software
  - Object-Oriented Programming (OOP) concepts
  - DO-178 essentials
- Inheritance
  - “Liskov Substitution Principle”
  - Contract-based programming
  - Is a Square a Rectangle?
- Dynamic binding
  - Coverage and substitutability issues
  - Local type consistency verification
- Conclusions
- References
- Acronyms

High-Integrity Software

- Software that affects whether/how a system meets safety and/or security requirements
  - Reliability (correctness): software meets its requirements
  - Analyzability: software has the relevant safety / security properties
- Generally subject to domain-specific standards
  - Safety: DO-178B for airborne systems in commercial aircraft:
    - Revised in December 2011: DO-178C and Supplements
    - Objectives and activities based on software life-cycle processes
  - Security: relevant Common Criteria "protection profile"
    - Catalog of "Security Functional Requirements" and "Security Assurance Requirements"

Object-Oriented Programming 1

- What is OOP?
  - Software development methodology with primary focus on data elements and their relationships
  - Secondary focus on the processing
- Language concepts / terminology
  - Class = module / data template with members
    - Operations (methods, functions) ⇨ behavior
    - Data fields (attributes) ⇨ state
  - Object = class instance
  - Encapsulation = control over visibility/accessibility of classes and their members)
  - Interface = restricted class
    - No data fields
    - All operations are "abstract" (only signatures, no implementation)

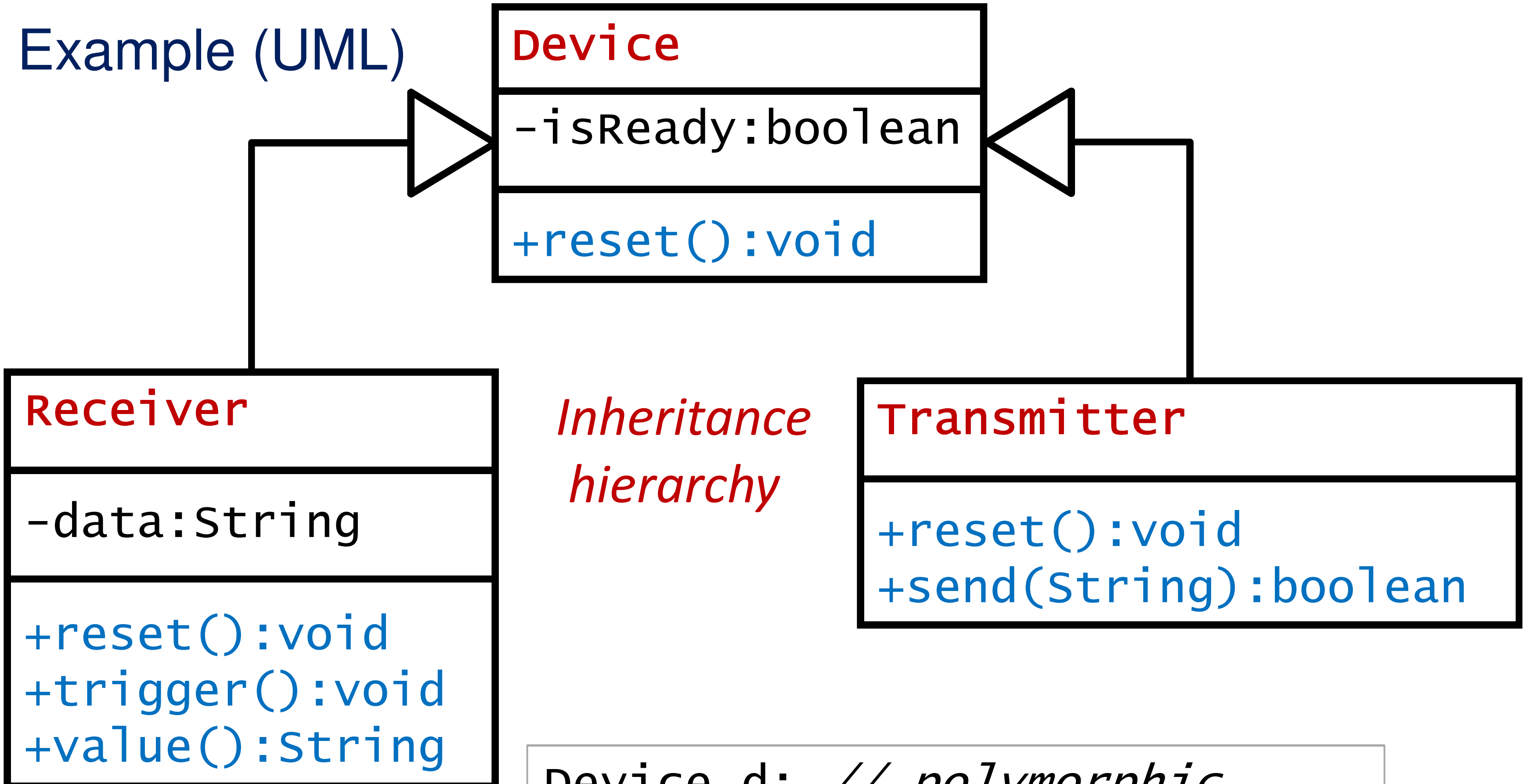
Object-Oriented Design ("OOD") [brace spanning Class through Encapsulation]

Object-Oriented Programming 2

- Language concepts / terminology (cont'd.)
  - Inheritance = "programming by extension"
    - Subclass can define new members and/or override the implementation of the superclass's operations
    - Subclass cannot remove any members
    - Interface inheritance: superclass is an interface
    - Implementation inheritance: superclass is not an interface
  - Inheritance hierarchy
    - A class together with all its direct and indirect subclasses
  - Polymorphism
    - The ability of a variable to reference objects from different classes (in the same class hierarchy) at different times
  - Dynamic binding ("dispatching")
    - The interpretation of an operation applied to a polymorphic variable based on the class of the object referenced by the variable, versus the type (class) of the variable itself

Object-Oriented Programming 3

- Example (UML)

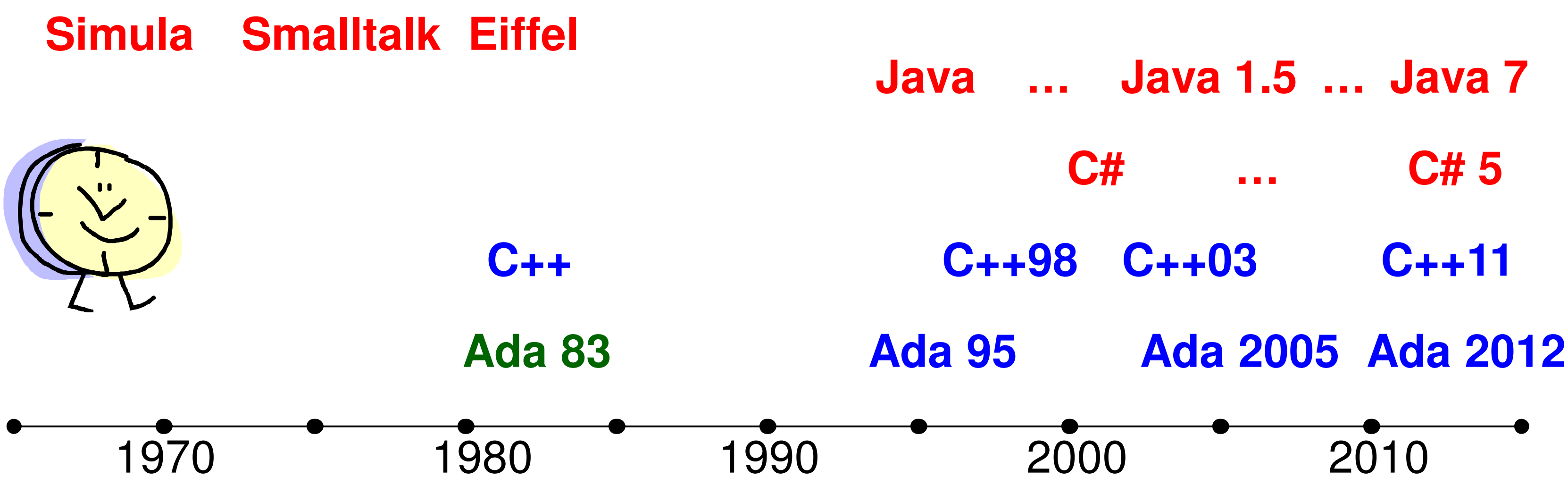


- Code fragment (Java)

```
Device d; // polymorphic
…
d = new Receiver();
…
d.reset(); // dynamic binding
```

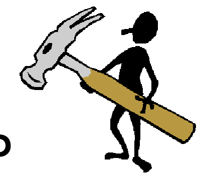
Object-Oriented Programming 4

- Language features / technology used with OOP
  - Overloading
  - Type conversion
  - Generic templates
  - Exceptions
  - Virtualization (e.g., JVM)
- Timeline of OO language evolution (sampler)

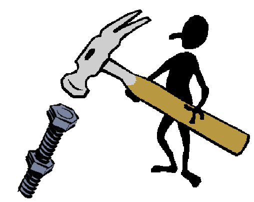


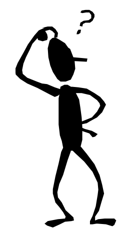
OOP in High-Integrity Systems?

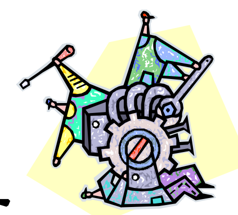
- Why consider OOP?
  - Ease of maintaining large systems
  - Tools may generate OO code that needs to be certified
  - Languages used for High-Integrity systems support OOP
  - Legacy OO code may need to be certified

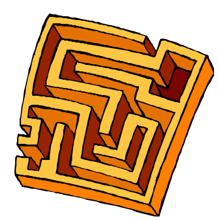


- What's the catch?
  - Paradigm clash
    - OOP decentralization of processing conflicts with standards' emphasis on traceability of functions
  - Culture clash
    - Certification authority evaluation personnel are domain experts, not "language lawyers"
  - Technical challenges
    - Dynamic flexibility that is heart of OOP conflicts with need to statically understand / analyze the source text



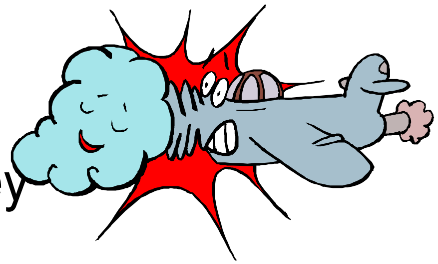


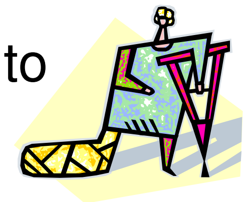




Addressing the Technical Issues

- Safety
  - Major effort has been in the context of DO-178B
  - Series of workshops organized by NASA Langley in conjunction with the FAA
    - *Object-Oriented Technology in Aviation* (OOTiA) handbook
  - Subgroup of Working Group that produced DO-178C
    - *Object Oriented Technology and Related Techniques Supplement (DO-332)*
  - DO-332 guidance can be adapted to safety standards in other domains





- Security
  - Nothing specific to OOP in Common Criteria
  - But the reliability and analyzability issues that arise in safety also occur at the higher Evaluation Assurance Levels

Remainder of the talk will be based on the discussion in DO-332

Some DO-178 Essentials

- DO-178B and DO-178C provide "guidance" for software in airborne systems
  - Objectives and activities related to "life cycle processes"
  - Major emphasis on verification
  - Objectives and activities depend on Software Level
    - Most demanding is Level A (failure could lead to loss of aircraft)
- Verification process aims to provide confidence (in correctness) proportional to Software Level
  - Coverage analysis
    - All software requirements are met (requirements-based tests)
    - Requirements-based tests cover the source code
    - No "extraneous" code (code not traceable to requirements)
    - "Deactivated" code (not intended to be executed) must be justified
  - Rushby paper discusses relationship between correctness and safety

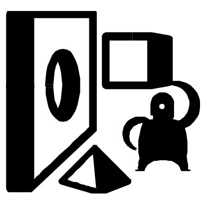
Summary of OOP Issues for High-Integrity

- Class structure
  - Unused operations
  - Encapsulation issues
- Inheritance issues
  - Unintended inheritance
  - "Fragile base class" problems
  - Improper usage of inheritance
  - Interaction with contract-based programming
  - Multiple inheritance
- Polymorphism
  - Reference semantics
  - Dynamic memory management
- Dynamic binding
  - Distinction from static binding
  - Coverage
  - Substitutability
- Other OOP issues
  - Constructors
  - Destructors
- Related features / technology
  - Overloading
  - Type conversions
  - Generic templates
  - Exception handling
  - Virtualization

Major Focus

Misuse of Inheritance

- Classes in an OO design exhibit various relationships
  - "Uses" (client)
  - "Has a" (aggregation)
  - "Is a" (specialization)
- Inheritance should only be used for specialization
  - Every superclass operation should apply (perhaps overridden) in the subclass
  - Sometimes known as the "Liskov substitution principle" (LSP)
    - "Let *q(x)* be a property provable about objects of type *T*. Then *q(y)* should be true for objects of type *S* where *S* is a subtype of *T*."
  - If LSP is violated then problems may arise
    - Operations that are inherited from the superclass may be inappropriate for the subclass, causing run-time errors if invoked



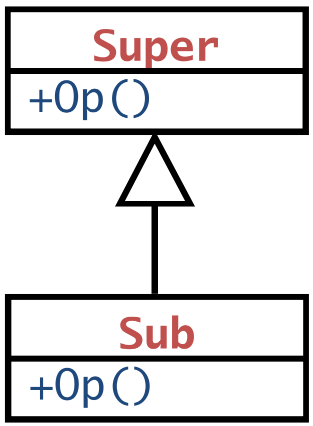
  - Use other language features for client, aggregation relationships

Contract-Based Programming

- Contract is assertion associated with operation or class
  - Operation precondition
    - Boolean condition that must be obeyed by caller, at the call
    - Can reference formal parameters, global data
  - Operation postcondition
    - Boolean condition that can be assumed by caller, on return
    - Can reference formal parameters (old and new), returned value
  - Class invariant
    - Postcondition of every public operation
- Use of contracts
  - Comments to human reader
  - Run-time check that can be enabled
  - Input to static analysis tool that can verify whether source code is consistent with contracts
- Supported to various degrees by current OO languages

Contracts and LSP

- Contract-based programming has important but counter-intuitive interaction with LSP on overriding an operation
  - Do not strengthen preconditions or weaken postconditions



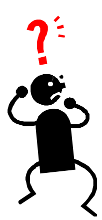

```
Precond(Op)  = Pre1
Postcond(Op) = Post1
```

Caller may only know this, and not the subclasses

Problem case:

```
Precond(Op)  = Pre1 and Pre2  // stronger
Postcond(Op) = Post1 or Post2 // weaker
```

```
Super ref;
… // may end up referencing an object from class Sub
ref.Op(); // Only knows to satisfy Pre1 (call may fail)
// Expects at least Post1 (further execution may fail)
```

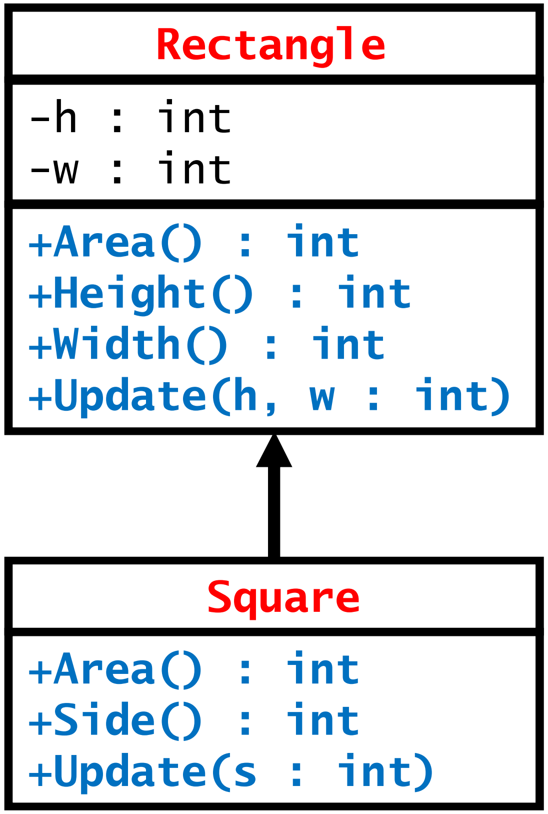


- Counterintuitive since specialization is more restrictive
  - Stronger precondition might be expected for subclass operations

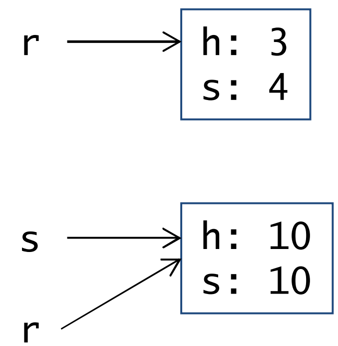
Implications of LSP

- A subclass should satisfy two properties
  - Behavioral consistency
    - Each operation (whether inherited or overriding) meets the requirements of the superclass's operation
  - Contract consistency
    - No operation (whether inherited or overriding) strengthens the precondition or weakens the postcondition of the superclass's operation
- Otherwise verification requires extra effort
  - If either property not met, then some method invocations may fail
  - Need to demonstrate that this will not occur

Example: Is a Square a Rectangle?



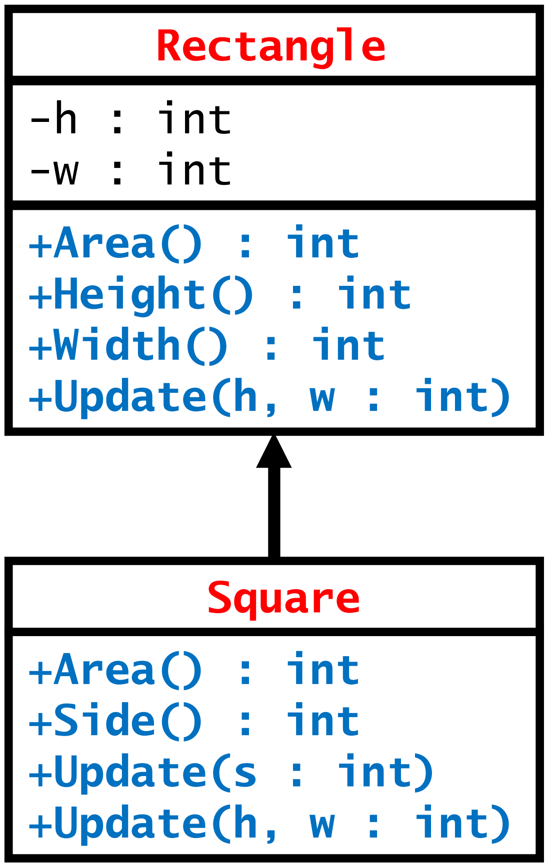

```
Rectangle r = new Rectangle();
Square s = new Square();
int j;
…
j = r.Area();
j = r.Height();
r.Update(3, 4);

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 `Square` is stronger than the *true* precondition in `Rectangle.Update()`

Solution 1: Is a Square a Rectangle?

- Override `Update(h, w)`, add explicit precondition h==w
- Precondition may be expressed with special syntax or explicit test
- Issues
  - The stronger precondition violates LSP
  - Invoking `Square.update(h, w)` when `h != w` is an error and should throw an exception

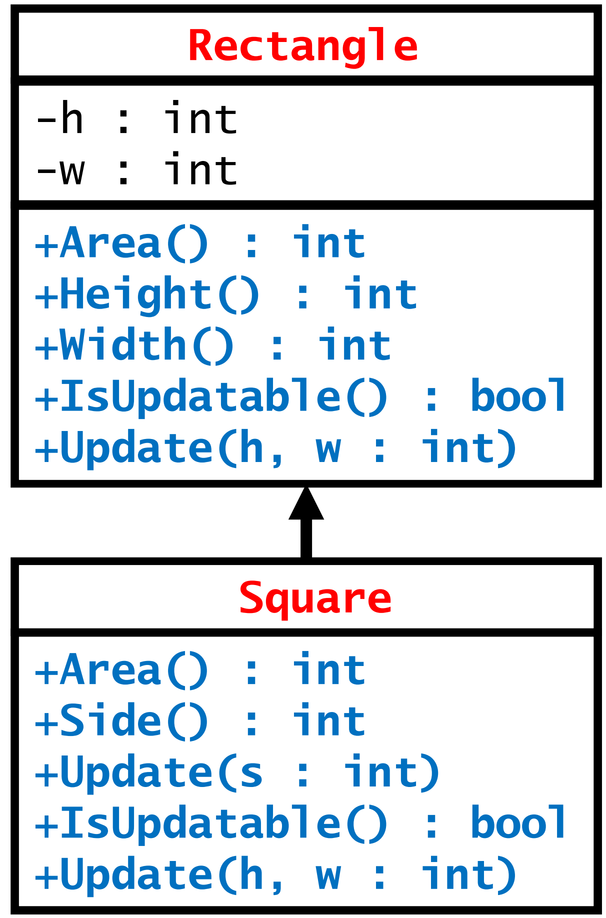


Solution 2: Is a Square a Rectangle?

- Override `Update(h,w)` as in Solution 1
- Add a dynamically bound precondition `IsUpdatable()`
  - Returns `True` for `Rectangle`, h==w for `Square`
- Stylistic convention
  - Call `r.IsUpdatable()` before each invocation of `r.Update(h, w)`

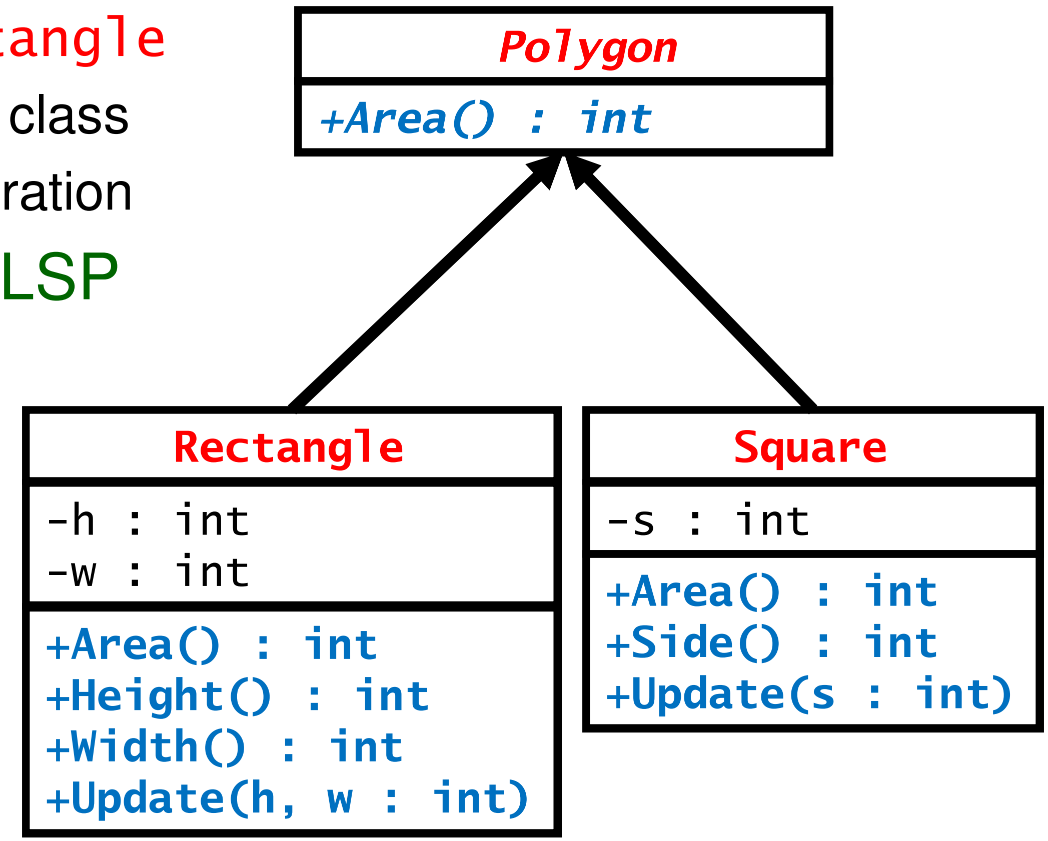
```
if (r.IsUpdatable()){
   r.Update(h, w);
}
```

- Notes
  - LSP is preserved (arguably)
  - OOP is compromised
  - Further discussion: B. Meyer's book, p. 576ff



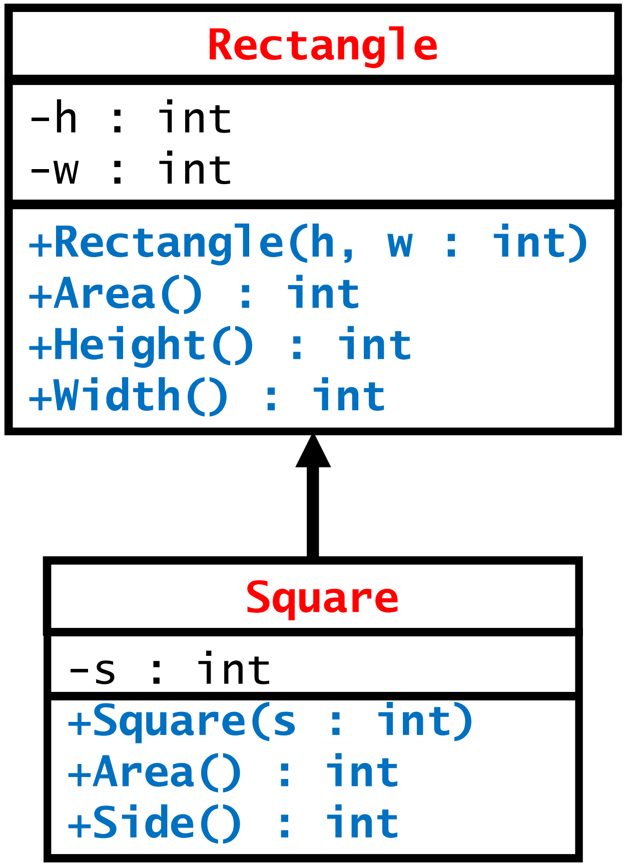
Solution 3: Is a Square a Rectangle?

- Redesign the class hierarchy
  - A Square is not a Rectangle
  - Polygon is an abstract class
  - Area is an abstract operation
- This design preserves LSP



Solution 4: Is a Square a Rectangle?

- Make `Rectangle` and `Square` *immutable*
  - Remove the `Update()` operation
  - Include constructors
- LSP is preserved
- A `Square` is a `Rectangle`
- Immutability effects
  - Avoids aliasing issues
  - Entails extra object construction

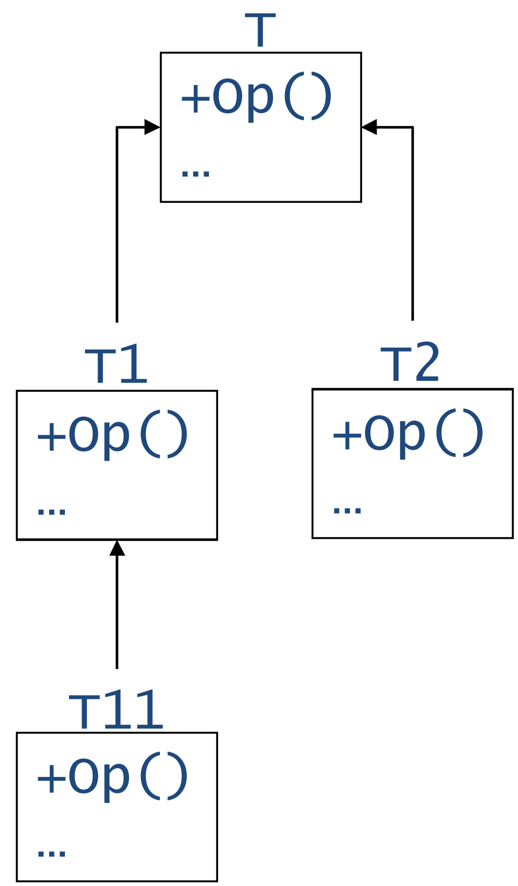


Dynamic Binding

- Verification / code coverage issue
  - Application Code

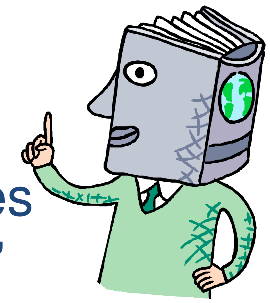
```
T p;  // polymorphic reference
p = …;
p.Op(); //dynamic dispatch
…
p.Op(); //dynamic dispatch
```

  - What is needed for full statement coverage?
    - Test *some* subclass at each invocation
      - Untested subclass may have error
    - Test *each* subclass at each invocation
      - May be redundant effort for some subclasses
  - Rather than adapt definition of statement coverage, DO-332 identifies the required verification based on the class hierarchy's "substitutability" (compliance with LSP)



Class hierarchy

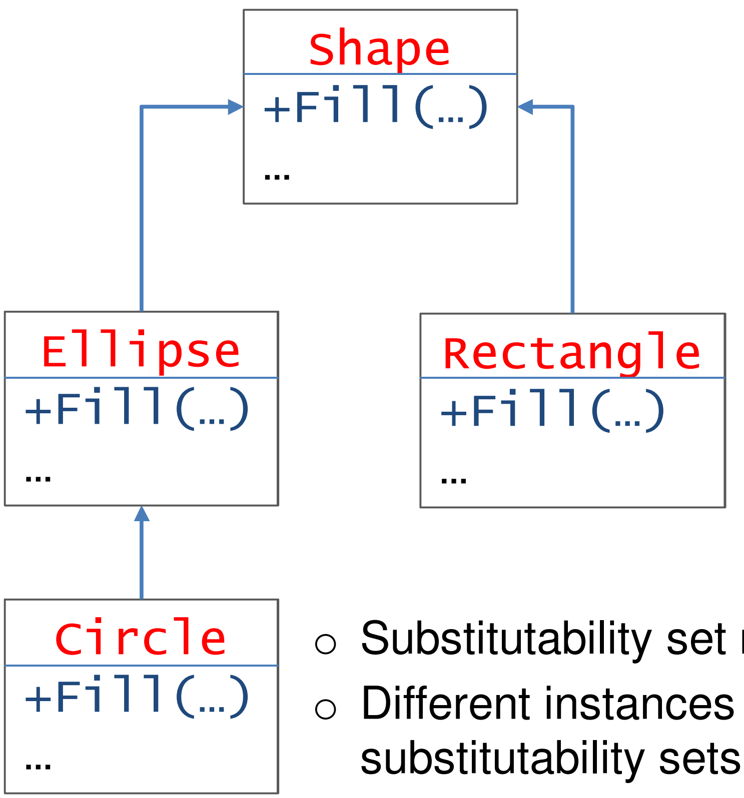
Local Type Consistency Verification 1



- OO.6.7 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.6.7.1 Local Type Consistency Verification Objective
  - "Verify that all type substitutions are safe by testing or formal analysis."
- OO.6.7.2 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)."

Local Type Consistency Verification 2

- Translation into English
  - "For each subtype where substitution is used"
    - For each occurrence of dynamic binding `p.Op(…)`, where p is of type `T`, identify all subclasses for objects that p could reference there
    - This set is the *substitutability set for p*, or *SS*(p)




```
Shape ref;
ref = (x>0) ? new Circle()
            : new Rectangle();
ref.Fill(…);
// SS(ref) = {Circle, Rectangle}
…
Ref = new Ellipse();
Ref.Fill(…);
// SS(ref) = {Ellipse}
```

- Substitutability set may be a subset of the full inheritance hierarchy
- Different instances of dynamic binding may have different substitutability sets

Local Type Consistency Verification 3

- "Optimistic" approach (LSP obeyed)
  - "Formally verify substitutability"
    - Through formal methods, demonstrate for each class `C` in *SS*(p)
      - `C.Op()` satisfies the requirements of `T.Op()` (behavioral consistency)
      - `C.Op()` does not strengthen the precondition or weaken the postcondition of `T.Op()` (contract consistency)
  - "Ensure that each class passes all the tests of all its parent types which the class can replace"
    - For each class `C` in *SS*(p), show that `C.Op()` passes the requirements-based tests of `T.Op()`
    - Implies verifying that preconditions not strengthened, postconditions not weakened
  - For either of these cases, only need to test some class `C` in *SS*(p) at each occurrence of dynamic binding of `p.Op()`

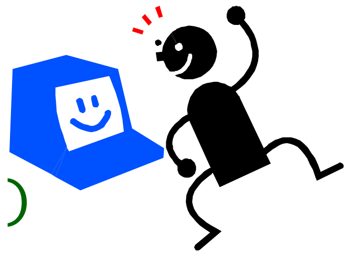
Local Type Consistency Verification 4

- "Pessimistic" approach (LSP not obeyed)
  - "For each call point, test every method that can be invoked at that call point (pessimistic testing)"
    - For each class `C` in *SS*(p), show that `p.Op()` executes correctly when p references an object of class `C`
  - Pessimistic testing may be practical in some situations
    - Few instances of dynamic binding
    - Shallow or narrow inheritance hierarchy
- Subclass verification
  - Local type consistency verification is in addition to the requirements-based tests (and other verification) for the subclass
- Local type consistency does not require LSP
  - But verification is simpler if class structure complies with LSP
  - Only need to consider local context

Conclusions

- OOP is "double-edged sword" for High-Integrity software
  - Some elements help; e.g., encapsulation
  - But analyzability and reliability problems arise from some of OOP's essential features including inheritance and dynamic binding

- In brief
  - Design inheritance hierarchies to adhere to LSP
    - Maintain contract consistency between subclass and superclass



  - Ensure that each dynamic binding occurrence `p.Op()` behaves correctly for the subclass of any object that could be referenced by p
    - "Optimistic" or "pessimistic" approaches, depending on whether LSP is obeyed
  - See DO-332 for comprehensive discussion of vulnerabilities

References 1

- High-Integrity standards

  - RTCA /EUROCAE **DO-178B/ED-12B**. *Software Considerations in Airborne Systems and Equipment Certification*, December 1992
  - RTCA /EUROCAE **DO-178C/ED-12C**. *Software Considerations in Airborne Systems and Equipment Certification*, December 2011
  - *Common Criteria: Common Methodology for Information Technology Security Evaluation*. www.commoncriteriaportal.org

- Object-Oriented Programming

  - RTCA /EUROCAE **DO-332/ED-217**. *Object-Oriented Technology 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
  - AdaCore. *High-Integrity Object-Oriented Programming in Ada*; March 2013. extranet.eu.adacore.com/articles/HighIntegrityAda.pdf

References 2

- Other resources

  - B. Liskov and J. Wing. “A behavioral notion of subtyping”, *ACM Transactions on Programming Languages and Systems* (TOPLAS), Vol. 16, Issue 6 (November 1994), pp 1811-1841
  - L. Rierson. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2013
  - J. Rushby. *New Challenges in Certification for Aircraft Software* www.csl.sri.com/users/rushby/papers/emsoft11.pdf
  - B. Meyer. *Object-Oriented Software Construction* (Second Edition), Prentice Hall; 1997

Acronyms

• JVM	Java Virtual Machine
• LSP	Liskov Substitution Principle
• OO	Object-Oriented
• OOD	Object-Oriented Design
• OOP	Object-Oriented Programming
• OOTiA	Object-Oriented Technology in Aviation
• UML	Unified Modeling Language