Object-Oriented Programming for High-Integrity Systems: *Avoiding the Vulnerabilities*

STC 2014
Long Beach, California

Session 32 / Track 4
Wednesday, April 2, 2014
9:00 – 9:45 am

Ben Brosgol • brosgol@adacore.com

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)

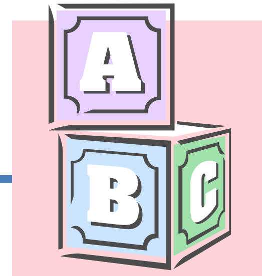
European Headquarters:
46 rue d'Amsterdam
75009 Paris
France
+33-1-4970-6716 (voice)
+33-1-4970-0552 (FAX)

www.adacore.com

Overview

- Introduction / basics
- Class structure issues
- Inheritance issues
- Polymorphism issues
- Dynamic binding issues
- Other issues
- Conclusions
- References
- Acronyms

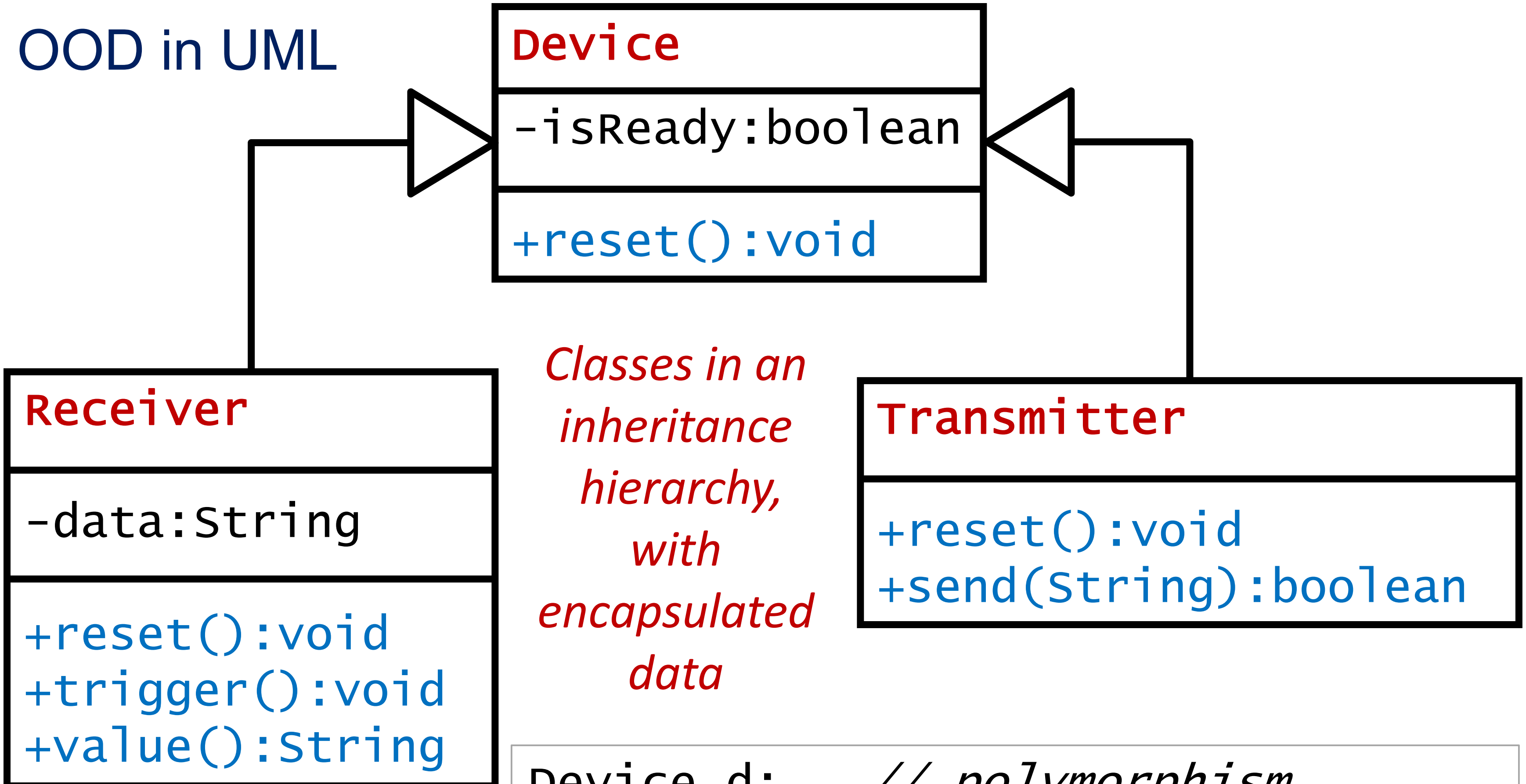
Basic Concepts



- "Object Orientation"
  - Development methodology based on classes and their relationships
  - Supported by many languages (C++, Java, Ada, C#, …)
- "High-Integrity" system
  - Stringent safety and/or security requirements
  - Compliance with certification standard may be required
    - Examples: DO-178C/ED-12C (commercial avionics), Common Criteria protection profile (security)
  - Software must be *reliable*
    - Correctly implements its specification
  - Software must be *analyzable*
    - Demonstrate that it meets relevant properties
- "Vulnerability"
  - An error that can result in a violation of a safety or security requirement

Object-Oriented Design & Programming Example

- OOD in UML



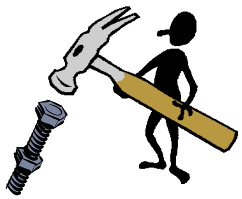
- OOP in Java

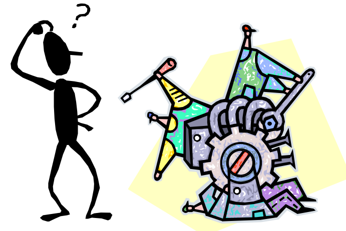
```
Device d;    // polymorphism
…
d = new Receiver(); // constructor
…
d.reset();   // dynamic binding
```

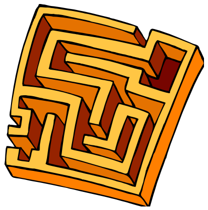
OOP in High-Integrity Systems?

- 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

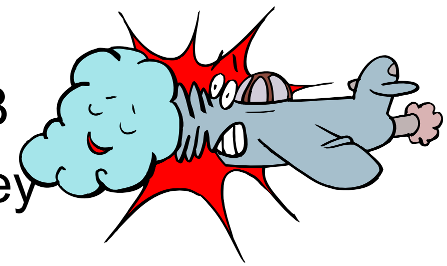
- What's the catch?
  - Paradigm clash
    - OOP decentralization of processing conflicts with standards' emphasis on traceability of functions
  - Culture clash
    - Certification authority evaluation personnel are domain experts, not "language lawyers"
  - Technical challenges
    - Dynamic flexibility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



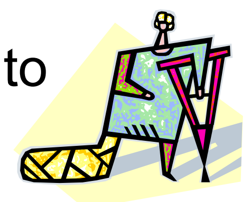


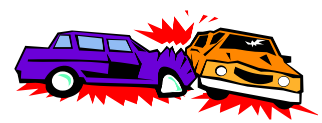


Addressing the Technical Issues

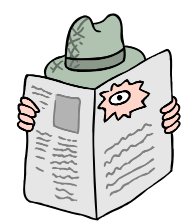


- Safety
  - Major effort has been in the context of DO-178B
  - Series of workshops organized by NASA Langley in conjunction with the FAA
    - *Object-Oriented Technology in Aviation* (OOTiA) handbook
  - Subgroup of Working Group that produced DO-178C
    - *Object Oriented Technology and Related 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

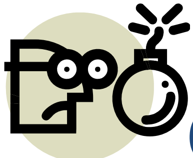


Remainder of the talk will be based on the guidance in DO-332

Summary of OOP Issues for High-Integrity

- Class structure
  - Unused operations
  - Encapsulation issues
- Inheritance
  - 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
  - "Vtable" corruption
- Other OOP issues
  - Constructors
  - Destructors

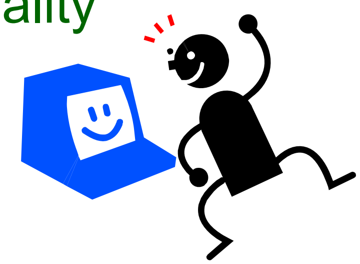
- Most of these issues relate to reliability problems in application code
  - Error may lead to system failure that presents a safety hazard or an exploitable security vulnerability
- Some relate to reliability problems in implementation code/libraries
  - Dynamic memory management
  - Vtable corruption



Class Structure: Unused Code

- Issue
  - Only invoke some of a class's operations, or override an operation but don't instantiate the superclass
  - Uninvoked operations will not be covered by tests that can be traced back to system requirements or high-level software requirements

- Solutions
  - Smart linker removes unused code
  - Specialized libraries have restricted functionality
  - Treat as deactivated code
    - Show they can't be invoked
    - Define "derived" requirements
    - Unused code still needs to be tested

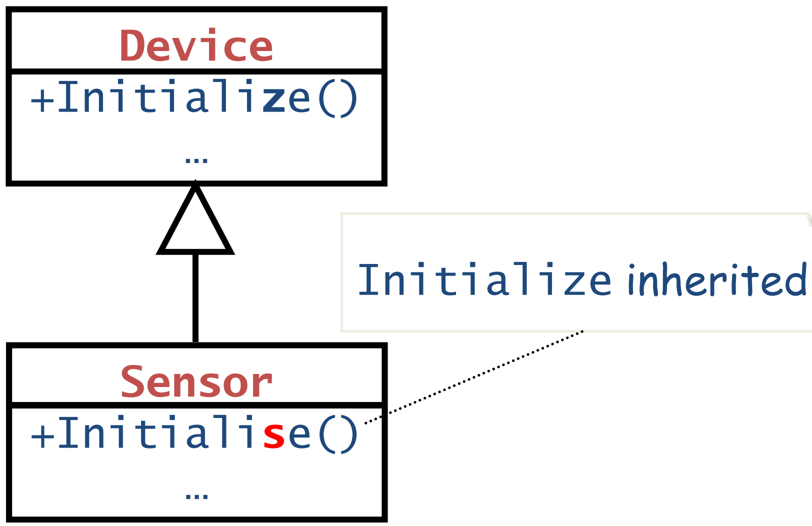


Class Structure: Encapsulation

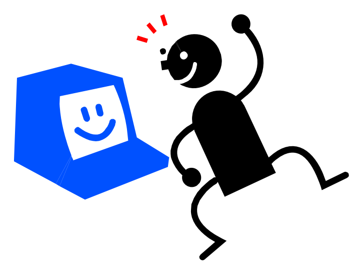
- Accessor operations
  - Issue
    - “Getter”,”Setter” operations for referencing fields ⇨ inline expansion
    - Affects traceability, coverage analysis, stack size analysis
  - Solution
    - Account for effects of inline expansion
- Robustness tests
  - Issue
    - Such tests require manipulation of encapsulated data
    - Encapsulation prevents such accesses
  - Solution
    - Use language-specific feature (C++ friends, Ada child package)

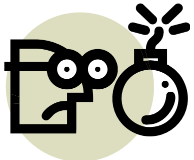
Unintended Inheritance

- Issue: misspell operation name in subclass
  - Dynamic binding will invoke the implicitly inherited version



- Solution
  - Language-specific syntax to explicitly override
    - If an overriding indication appears, the operation must be an overriding
    - Supported by Ada, Java, C++11
  - Will detect error when applied to a non-overriding operation

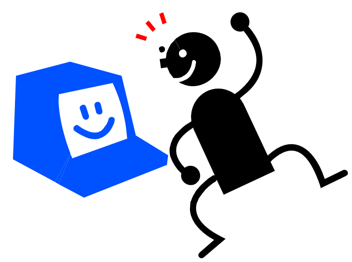




“Fragile Base Class” Problem

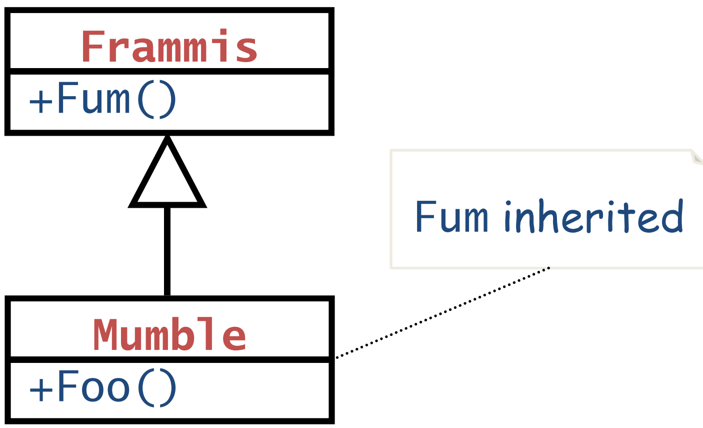
- Issue
  - Adding declarations to a superclass (e.g., during maintenance) can cause compilation errors or silent bugs in subclasses
  - Examples
    - Unintended overriding when new operation added to superclass
    - Access to incorrect data when field added to an intermediate level class

- Approaches
  - Language syntax to detect errors
  - Careful review / analysis on recompilation of subclasses



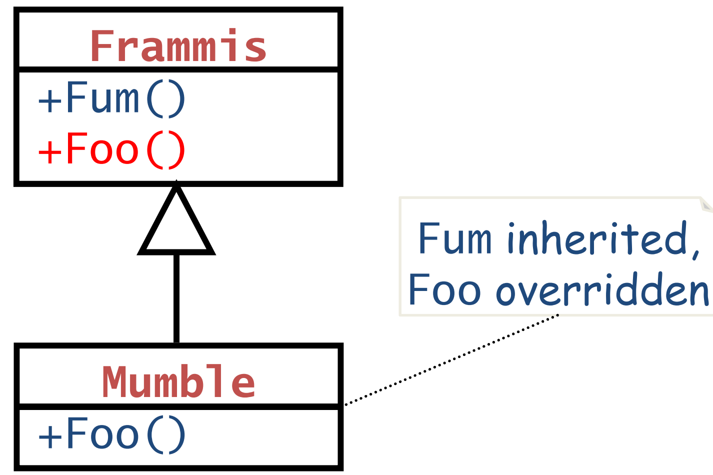
Unintended Overriding

- Initially



- Method Foo() is defined only for subclass Mumble

- During maintenance

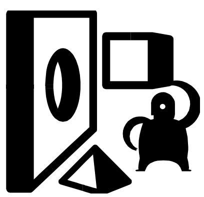


- Dynamic binding will invoke overriding version

- Solution
  - Language-specific syntax
    - If an overriding indication is not supplied, operation must not be an overriding
    - For compatibility reasons, this solution is not part of current standards

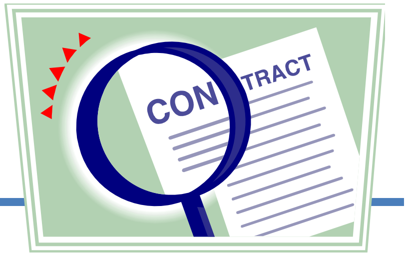
Misuse of Inheritance

- 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



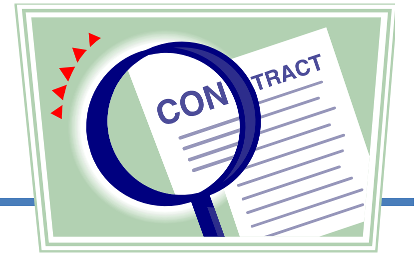
  - Use other language features for client, aggregation relationships

Contract-Based Programming

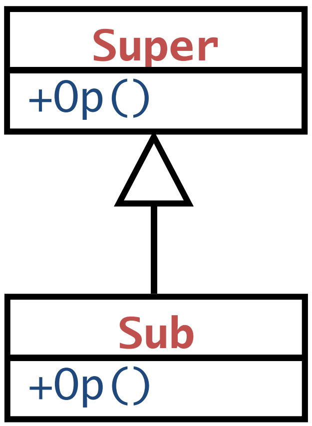


- Contract is assertion associated with operation or class
  - Operation precondition
    - Boolean condition that must be obeyed by caller, at the call
    - Can reference formal parameters, global data
  - Operation postcondition
    - Boolean condition that can be assumed by caller, on return
    - 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



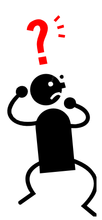

```
Precond(Op)  = Pre1
Postcond(Op) = Post1
```

Caller may only know this, and not the subclasses

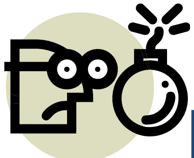
Problem case:

```
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



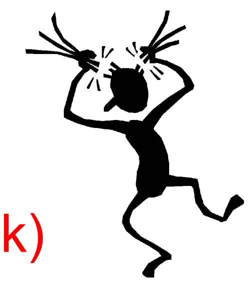
Multiple Inheritance

- Issues with multiple implementation inheritance
  - Semantic complexity
  - Easy to misuse (i.e., to violate LSP)

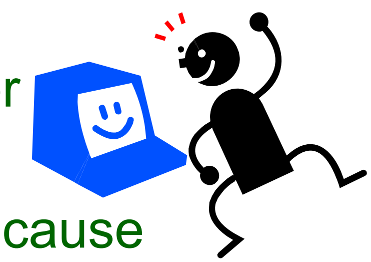
- Issues with multiple interface inheritance
  - Inconsistent signatures from different interfaces
  - Identical signatures from different interfaces
- Solution
  - Use multiple interface inheritance,
    not multiple implementation inheritance
  - Resolve signature clashes from multiple interfaces
    using language-specific techniques

Polymorphism: Reference Semantics

- Polymorphic variable of type (class) T is represented as a reference
  - Either null, or a pointer to an object from some class in T hierarchy
- Issues
  - Dereferencing a null pointer is a run-time error
  - Aliasing complicates demonstration of data consistency
  - Possibility of dangling reference (to an object on the stack)



- Solutions
  - Static analysis to show absence of null pointer dereferencing
  - Static analysis to show that aliasing does not cause unwanted side effects
  - Language checks or static analysis to show absence of dangling references

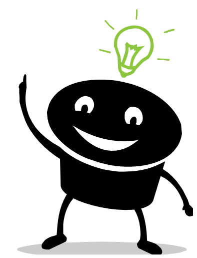


Polymorphism: Dynamic Memory

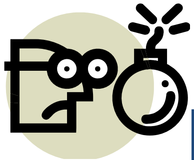
- Vulnerabilities*
  - Ambiguous references
  - Fragmentation starvation
  - Deallocation starvation
  - Heap memory exhaustion
    - Underestimation of heap size
    - Memory leak
  - Premature deallocation
  - Unsynchronized object movement
  - Timing overrun

Dynamic memory management is the single most critical factor that affects certifying Object-Oriented code

- Avoiding the problem
  - No dynamic allocation
  - Dynamic allocation only at application startup, no deallocation
- Solving the problem*
  - Object pooling
  - Stack allocation
  - Manual heap management
  - Automatic heap management ("garbage collection")
- Responsibilities*
  - May be either the application or the implementation



* DO-332, §OO.D.1.6



Polymorphism: Dynamic Memory

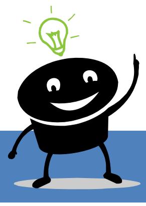


Table OO.D.1.6.3

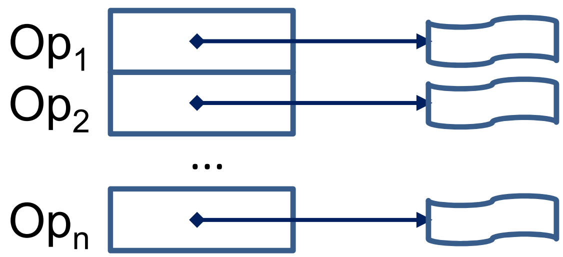
	Objective						
Technique	**Unambig. Reference**	**Fragment. Avoidance**	**Dealloc. Starvation Avoidance**	**Sufficient Memory**	**Premature Dealloc. Avoidance**	**Atomic Move**	**No Timing Overrun**
Object Pooling	AC	AC	AC	AC	AC	N/A	N/A*
Stack Allocation	AC	N/A*	MMI	AC	MMI	N/A	MMI
Manual Heap Management	MMI**	AC** *	AC	AC	AC	N/A	MMI
Garbage Collection	MMI	MMI	MMI	AC	MMI	MMI	MMI

Key
AC = Application Code
MMI = Memory Management Infrastructure
N/A = Not Applicable by either AC or MMI

Entries in Table OO.D.1.6.3:
*** MMI**
**** AC**
***** "Difficult to ensure by either AC or MMI"**

Dynamic Binding

- Integrity of "Vtable"
  - Dynamic binding generally implemented by per-class data structure (table of addresses of dispatching operations)
  - Issue
    - Vulnerability if Vtable not correctly initialized, or if corrupted during execution
  - Solution
    - Verify initial contents, store in ROM



- Distinguishing dynamic and static binding
  - Major semantic difference between dynamic and static binding of operation invocation `ref.Op(…)`
  - Issue
    - Program bug if incorrectly use one rather than the other
  - Solution
    - Language-specific semantics

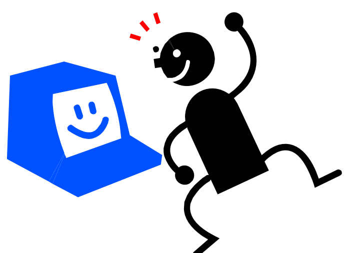
Other OOP Features

- 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 (Worst Case Execution Time) in the presence of finalizers

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

- In brief
  - OOP will be seeing increasing usage in High-Integrity systems
    - Will become a more attractive target for malware
  - Addressing the vulnerabilities involves many factors
    - Language features
    - Run-time library implementation
    - Static analysis tools
    - Coding standard restrictions
      - CERT Secure Coding Standards (C++, Java)
      - ISO/IEC JTC 1/SC 22/WG 23 (Programming Language Vulnerabilities)
    - Effective class hierarchy design



  - 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
  - B. Meyer. *Object-Oriented Software Construction* (Second Edition), Prentice Hall; 1997.
  - FAA. *Handbook for Object-Oriented Technology in Aviation (OOTiA)*, October 2004.
    www.faa.gov/aircraft/air_cert/design_approvals/air_software/oot
  - RTCA /EUROCAE **DO-332/ED-217**. *Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A*, December 2011.

References 2

- Language-related vulnerabilities
  - ISO/IEC JTC 1/SC 22/WG 23. *Technical Report ISO/IEC TR 24772, Second edition 2013-03-01*; “Guidance to avoiding vulnerabilities in programming languages through language selection and use”
  - AdaCore. *High-Integrity Object-Oriented Programming in Ada*; June 2013. extranet.eu.adacore.com/articles/HighIntegrityAda.pdf
  - *The CERT Oracle Secure Coding Standard for Java*, Object Orientation. https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=6029486
  - *CERT C++ Secure Coding Standard*, Object Oriented Programming. https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=2708
- 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

Acronyms

• 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
• WCET	Worst Case Execution Time

Note: the "DO" in DO-178B, DO-178C, and DO-332 is not an acronym; it is simply an abbreviation of "Document"