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
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

Device
- isReady: boolean
+ reset(): void

Classes in an inheritance hierarchy, with encapsulated data

Receiver
- data: String
+ reset(): void
+ trigger(): void
+ value(): String

Transmitter
+ reset(): void
+ send(String): boolean

• 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

<table>
<thead>
<tr>
<th>Frammis</th>
<th>+Fum()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mumble</th>
<th>+Foo()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Method Foo() is defined only for subclass Mumble

• 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

• During maintenance

<table>
<thead>
<tr>
<th>Frammis</th>
<th>+Fum()</th>
<th>+Foo()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mumble</th>
<th>+Foo()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Dynamic binding will invoke overriding version
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 counterintuitive interaction with LSP on overriding an operation
  - Do not strengthen preconditions or weaken postconditions

  
  \[
  \begin{align*}
  \text{Super} & \quad \text{Precond}(\text{Op}) = \text{Pre}1 \\
  \quad +\text{Op}() & \quad \text{Postcond}(\text{Op}) = \text{Post}1 \\
  \end{align*}
  \]

  Caller may only know this, and not the subclasses

  
  \[
  \begin{align*}
  \text{Sub} & \quad \text{Precond}(\text{Op}) = \text{Pre}1 \text{ and } \text{Pre}2 \\
  \quad +\text{Op}() & \quad \text{Postcond}(\text{Op}) = \text{Post}1 \text{ or } \text{Post}2
  \end{align*}
  \]

  Problem case:

  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

- **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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Pooling</td>
<td>AC</td>
<td>AC</td>
<td>AC</td>
<td>AC</td>
<td>AC</td>
<td>N/A</td>
<td>N/A*</td>
</tr>
<tr>
<td>Stack Allocation</td>
<td>AC</td>
<td>N/A*</td>
<td>MMI</td>
<td>AC</td>
<td>MMI</td>
<td>N/A</td>
<td>MMI</td>
</tr>
<tr>
<td>Manual Heap Management</td>
<td>MMI**</td>
<td>AC** *</td>
<td>AC</td>
<td>AC</td>
<td>AC</td>
<td>N/A</td>
<td>MMI</td>
</tr>
<tr>
<td>Garbage Collection</td>
<td>MMI</td>
<td>MMI</td>
<td>MMI</td>
<td>AC</td>
<td>MMI</td>
<td>MMI</td>
<td>MMI</td>
</tr>
</tbody>
</table>

**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
      o CERT Secure Coding Standards (C++, Java)
      o 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

• Object-Oriented Programming
• Language-related vulnerabilities
  - AdaCore. High-Integrity Object-Oriented Programming in Ada; June 2013. extranet.eu.adacore.com/articles/HighIntegrityAda.pdf

• Other resources
# Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSP</td>
<td>Liskov Substitution Principle</td>
</tr>
<tr>
<td>OO</td>
<td>Object-Oriented</td>
</tr>
<tr>
<td>OOD</td>
<td>Object-Oriented Design</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-Oriented Programming</td>
</tr>
<tr>
<td>OOTiA</td>
<td>Object-Oriented Technology in Aviation</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst Case Execution Time</td>
</tr>
</tbody>
</table>

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