Testing and Object-Oriented Software

João Pereira ©

Introduction

- Testing is a search/hunt of bugs
- Identifying bug hazards is essencial for **effective** testing
- Must examine the ways in which OOP languages can go wrong

Object-Oriented Programming Languages

- OOPL would solve all problems, including bugs

  → No more Testing!

- However, programmers make errors independently of the language

- Some errors are language specific

  – Create a subclass that is inconsistent with its superclass

Fault model for object-oriented programming

- OO programming reduces some kinds of errors
  - Methods often consist of a few lines
  - Encapsulation prevents bugs that result from global data scoping

- However, it increases (or creates) the chance of others
  - Encapsulation, polymorphism, and method sequence pose new bug hazards

Encapsulation

- Access control mechanism
- Can be an **obstacle** to testing

Real example commercial C++ library

```
class IntSet {
  public:
     IntSet();
     ~IntSet();
     IntSet& add(int);  // add a member
     IntSet& remove(int);  // remove a member
     IntSet& clear();  // remove all members
     int is_member(int);  // is arg a member?
     int extent();  // number of elements
     int is_empty();  // empty or not?
     …
}
```

Encapsulation - 2

- Problem was when adding the same number twice
  - *add()* should throw an exception in this case
- Test case?

```
try {
   set.add(1);
   set.add(1);
   fail("Error. Same number added twice");
} catch (DuplicateException de) {
}
```

- What is missing?
- Check content of set inside catch
  - Encapsulation prevents that

```
try {
   set.add(1);
   set.add(1);
   fail("Error. Same number added twice");
} catch (DuplicateException de) {
   set.remove(1);
   assertTrue(set.is_member(1) == 0);
}
```

But

Inheritance

- Crucial in OO paradigm
  - Supports reusability
  - Allows efficient extensibility (type & subtype)
- Unfortunately, it can be abused in many ways
- Can make difficult to understand source code
- How to test Generic Types and Abstract Classes?

Inheritance: What can go wrong?

- Incorrect initialization
  - Can easily go wrong
  - Superclass initialization code not executed
- Forgotten methods
  - Proper use of upper-level features may become obscure when we have many levels of inheritance
- Multiple inheritance
  - a class inherits directly from 2 or more classes which may contain features with the same name
  - Presents many bug hazards
- …

Polymorphism

- Is the ability to bind a reference to more than one class of objects
- Dynamic binding: the specific method is determined in runtime
- Produces compact, elegant and extensible code
- Each possible binding of a polymorphic method is a distinct computation
- It may be difficult to identify and exercise all such bindings

Polymorphism: What can go wrong?

- Polymorphic methods can result in hard-to-understand error-prone code: yo-yo problem
- Class hierarchy must be carefully designed, otherwise it can produce strange results for some messages
  - Overriding a method that is incompatible with the original one – client code will not work in some cases
- Messages can be bound to the wrong class
  - Only for untyped languages
  - Many classes may use the same method names

Extra information

Polymorphism: What can go wrong – 2?

- The yo-yo problem
- As classes grow deeper the likelihood of misusing polymorphic methods increases
- Combination of
  - Template method design pattern
  - Invocation of super version in a subclass method
- Make it very difficult to understand the behavior of lower level classes

The yo-yo problem

```
abstract class One {
  public void A() {  print( "A, 1" );    B();   C(); }
  public void C() { print( "C, 1" ); }
  abstract public void B();
  abstract public void D();
}
class Two extends One {
  public void B() { print( "B, 2" );  D();}
  public void D() { print( "D, 2" ); }
}
class Three extends Two {
  public void A() { print( "A, 3" ); super.A(); }
  public void B() { print( "B, 3" ); super.B(); }
}
```

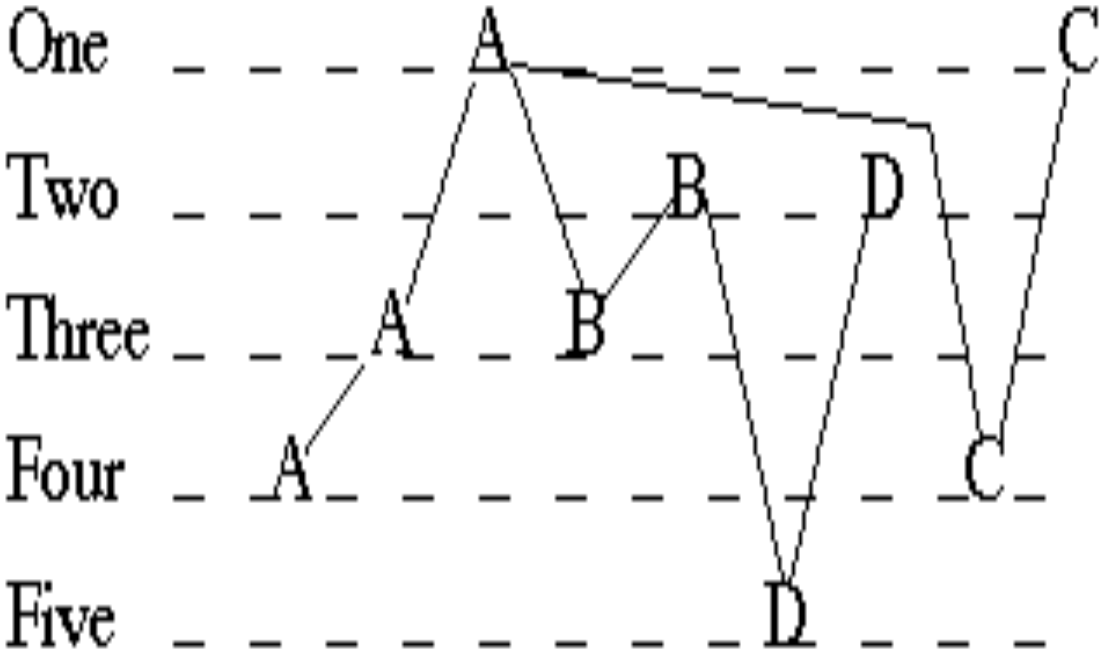
```
class Four extends Three {
  public void A() {
    print( "A, 4" ); super.A(); }
  public void C() {
    print( "C, 4" ); super.C(); }
}

class Five extends Four {
  public void D() {
    print( "D, 5" );
    super.D(); }
}
```

- What is the result of

```
public static void main( String args[] ){
   Five yoyo = new Five();
   yoyo.A();
 }
```

The yo-yo problem - 2



- Invocation of a single method on class Five triggered the invocation of 9 methods on the same object
  - One of these interactions maybe be wrong

Message sequence

- The packing of methods and states into classes is fundamental to the OO paradigm. As a result, messages are sent in some sequence.
- Account object :
  - *must have sufficient funds (be in an open state) before a withdrawal message can occur.*
  - *If withdrawal causes the balance to become negative then we transition from open to overdrawn state*
- State may be corrupted under certain message sequence patterns
  - add, *withdrawal (balance negative), withdrawal (refused), withdrawal (accepted)*
- Testing Goal:
  - Select a set of sequences to test (including correct and incorrected)
- We can use a state-based test model to model this.
  - State based testing derives test cases by modeling a class as a state machine.