OO Testing

Overview

- OO Software Testing
- OO Unit Testing: class testing
- OO Integration Testing: multiple class testing
**OO Software Testing**

- Some of the older testing techniques are still useful
  - Class testing is similar to unit testing
  - Multiple class testing is similar to integration testing
- New testing techniques are specially designed for OO software
  - State-based testing

**OO Unit Testing: Class Testing**

- Traditional view of “unit”: a procedure
- In OO: a method is similar to a procedure
- But a method is a part of a class, and is tightly coupled with other methods and fields in the class
- The smallest testable unit is a class
Class Testing

- DU pairs can still be used to design data flow based testing
  - However, test cases should cover both DU pairs inside a method and crossing method boundaries
    - i.e., intra-method and inter-method
- Testing method ordering
- Testing polymorphism

DU-Pair Testing Example

class A {
  private int index;
  public void m1() {
    index = ...;
    ...
    m2();
  }
  private void m2() { ... x = index; ... }
  public void m3() { ... z = index; ... }
}

test 1: call m1, which writes index and then call m2 which reads the value of index

test 2: call m1, and then call m3
Testing Method Ordering

• Random testing
  – Conduct random test to exercise different call sequences and different class instance life histories

• Partition testing
  – Similar to equivalence partition to reduce test cases

Random Testing Example

<table>
<thead>
<tr>
<th>Account</th>
<th>open()</th>
<th>setup()</th>
<th>deposit()</th>
<th>withdraw()</th>
<th>balance()</th>
<th>summarize()</th>
<th>creditLimit()</th>
<th>close()</th>
</tr>
</thead>
</table>

• Test case 1
  – open•setup•deposit•deposit•balance•summarize•withdraw•close

• Test case 2
  – open•setup•deposit•withdraw•deposit•balance•creditLimit•withdraw•close

• Limitation
  – too random to be effective
  – may test some infeasible cases
Partition Testing

• State-based partitioning
  – To categorize class operations based on their ability to change the state of the class
  – To design different test cases to
    • cover every set of operations
    • cover every state of the class

Finite State Machine Diagram

• Two types of operations
  – State operations
    • open(), setup(), deposit(), withdraw(), close()
  – Nonstate operations
    • balance(), summarize(), creditLimit()
Test Cases

- Test case 1
  - open • setup • deposit(initial) • withdraw(final)
  - close
- Test case 2
  - open • setup • deposit(initial) • deposit • balance •
    credit • withdraw(final) • close
- Test case 3
  - open • setup • deposit(initial) • deposit •
    withdraw • accountInfo • withdraw(final) • close

Polymorphism

- Suppose class X has a method calling a.foo(), where variable a is of type A
  - The function call may invoke
    A.foo(), B.foo(), C.foo(), D.foo()
- How to “drive” call site a.foo() through all possible bindings?
Testing Polymorphism

- **All-receiver-classes**: execute every possible receiver of type A
  - A.foo(), B.foo(), C.foo(), D.foo()
- **All-invoked-methods**: execute with receivers whose classes define foo()
  - A.foo() (or B.foo() or D.foo()), C.foo()

How to Find All Possible Method Targets?

- **Class Hierarchy Analysis (CHA)**
  - Conduct compile-time analysis to get type hierarchy info and find all possible method targets at call site a.foo()
    - Know all subclasses of class A
    - Know all methods defined in those classes and A with method signature foo()
    - Every found method is a possible method target
Refinement: Rapid Type Analysis

• Limitation of CHA
  – Not all “possible” method targets are actually invoked
• Rapid Type Analysis (RTA)
  – Also collect info on which classes are actually instantiated

Example

static void main(){
  B b1 = new B();
  B b2 = new C();
  f(b1);
  g(b2);
}
static void f(A a2){
  a2.foo();
}
static void g(B b2){
  b2.foo();
}
class A {
  foo(){..}
}
class B extends A{
  foo() {...}
}
class C extends B{
  foo() {...}
}
class D extends B{
  foo(){...}
}
**CHA Result**

```java
static void main(){
    B b1 = new B();
    B b2 = new C();
    f(b1);
    g(b2);
}

static void f(A a2){
a2.foo();
}

static void g(B b2){
b2.foo();
}

class A {
    foo(){..}
}

class B extends A{
    foo() {...}
}

class C extends B{
    foo() {...}
}

class D extends B{
    foo() {...}
}
```

**RTA Result**

```java
static void main(){
    B b1 = new B();
    B b2 = new C();
    f(b1);
    g(b2);
}

static void f(A a2){
a2.foo();
}

static void g(B b2){
b2.foo();
}

class A {
    foo(){..}
}

class B extends A{
    foo() {...}
}

class C extends B{
    foo() {...}
}

class D extends B{
    foo() {...}
}
```
Myths about Inheritance

• “If we have a well-tested superclass, we can reuse its code (in subclasses, through inheritance) without retesting inherited code”
• “A good-quality test suite used for a superclass will also be good for a subclass”

Problems with Inheritance

• P1: Incorrect initialization of superclass attributes by the subclass
• P2: Missing overriding methods
  – Typical example: `equals()` and `clone()`
• P3: Subclass may cause side effects and violate an invariant from the superclass
Example 1

class A {
    protected int x; // invariant: x > 100
    void m() { // correctness depends on
        // the invariant ... } ...
}
class B extends A {
    void m1() { x = 1; ... } ...
}

• If m1 has a bug and breaks the invariant, m is incorrect in the context
  of B, even though it is correct in A
  – P1, P3

Example 2

class A {
    void m() { ... m2(); ... }
    void m2 { ... } ...
}
class B extends A {
    void m2() { ... } ...
}

• If m2() is buggy, so is m() called on B instance
  – P3
Testing of Inheritance

- **Principle:** inherited method should be retested in the context of a subclass
  - Example 1: if we change some method \( m() \) in a superclass, we need to retest \( m() \) inside all subclasses that inherit it
  - Example 2: if we add or change a subclass, we need to retest all methods inherited from a superclass in the context of the new/changed subclass
  - Goal: check **behavioral conformance** of the subclass w.r.t. to the superclass (LSP)

Multiple Class Testing

- UML interaction diagrams: sequences of messages among a set of objects
- Basic idea: devise tests that cover all diagrams, all messages, and all conditions inside each diagram
  - If a diagram does not have conditions and iteration, it contains only one path
Communication Diagram

:Customer

1:cardInserted
2:password
3:withdraw
4:amount
5:log off

:System

2.1:verifyAccountPwd
4.1:validateWithdrawC

Test case:
cardInserted•password•verifyAccountPwd•withdraw•amount•validateWithdrawC•validateWithdrawA•debit•logoff

ATMOwner:Bank

4.2:validateWithdrawA
4.3:debit

cardIssuer:Bank

Alternative Scenario 1

• If the password is not correct
  – ATM prompts the customer to try again
  – Customer enters a password
  – ATM requests the card issuer bank to verify again
  – Repeat the above steps until verification succeeds or trialNumber == limit
Alternative Scenario 2

• If the verification finally fails and no retry is allowed
  – ATM reports the failure and returns the card

Alternative Scenario 3

• If the amount to withdraw is greater than the cash amount in ATM
  – ATM reports “not enough money”
  – ATM prompts the customer to retry
  – If the customer wants to cancel the transaction, logoff; Otherwise, the customer enters an amount
  – Repeat the above steps until the amount meets the requirement
Homework 3: Multiple Class Testing

• **Withdraw money from ATM**
  – Draw a CFG to cover all scenarios shown by the communication diagram and alternative descriptions
  – Devise test cases based on that
  – Feel free to define new operations if necessary

Requirements of Test Cases

• **Cover all scenarios (successful + failing)**
  – basis path testing (assume limit = 3)
  – loop testing
    • for an n-iteration loop, test scenarios: 0, 1, n-1, n
    • for an infinite loop, test scenarios: 0, 1, m (m > 1)
• **List test cases for each technique**
  – Briefly explain why these test cases are selected