Software Engineering III

OO Testing

Pressman 7/e Ch 19

Introduction

- Testing conventional structured software use strategies:
  - Top down
  - Bottom up
  - White box
  - Black Box
- Top down and bottom up do not apply to the architecture of OO.
  - The architecture of OO software is a **network** of collaborating classes.
  - Therefore it is necessary to test an OO system at a variety of different levels in an effort to uncover errors that may occur as classes collaborate with one another and the subsystems communicate across architectural layers.

Approach to Testing OO Systems

- While black box and white box strategies are applied to OO systems as to conventional structured systems the tactics are different.

- Because OO analysis and design models align more closely in structure and content to the resultant OO software, testing can begin with a review of these models.

Approach to Testing OO Systems

- As a class evolves through analysis and design to the coded stage, it becomes a target for test case design.
  - Once classes are coded, a series of tests are designed that exercise class operations and examine whether errors exist as objects of one class **collaborates** with those of other classes.

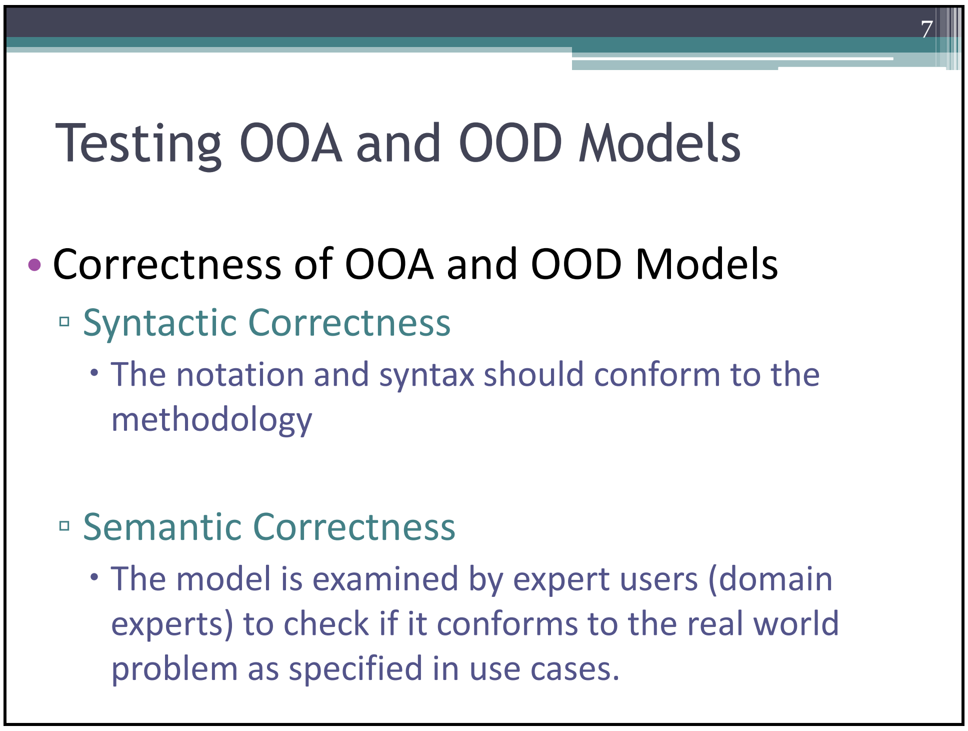
- Attributes and operations are encapsulated
  - Encapsulation produces test obstacles such as the reporting of the attribute values.

Approach to Testing OO Systems

- Inheritance leads to additional challenges
  - each level in the hierarchy produces a new context/environment of usage and requires retesting of the class at each level.
  - Each subclass in its own domain will require it's own specific test cases.
- White box testing can be applied to individual operations in isolation but the effort is often better applied to these tests at the class level.
- Black box testing is also appropriate to OO and the Use Case is a natural source of the test case.

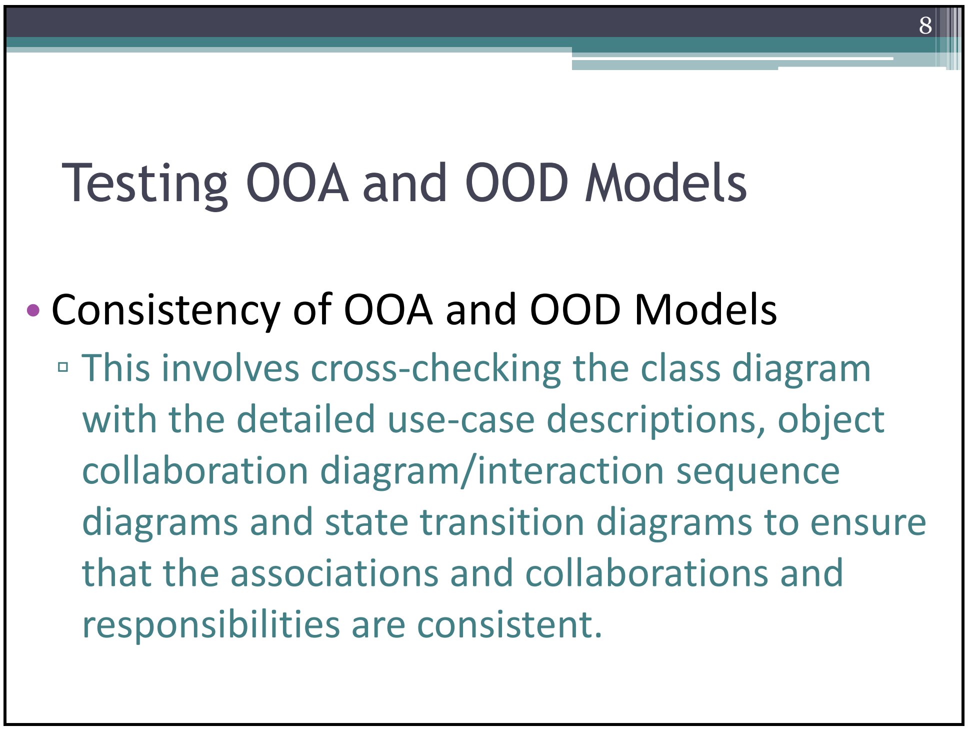
Testing OO Analysis and Design Models

- These models iteratively evolve from informal representations of system requirements to detailed models of classes, attributes, operations, associations, messages and sequencing, events ..
- Therefore problems that are detected in these properties at the early iteration stages of analysis and design stage will circumvent problems at the later stages of development.
- Technical reviews of these models are considered as important as final testing



Testing OOA and OOD Models

- Correctness of OOA and OOD Models
  - Syntactic Correctness
    - The notation and syntax should conform to the methodology
  - Semantic Correctness
    - The model is examined by expert users (domain experts) to check if it conforms to the real world problem as specified in use cases.

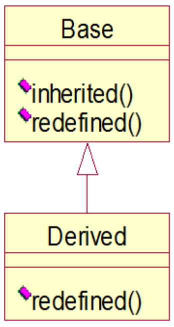


Testing OOA and OOD Models

- Consistency of OOA and OOD Models
  - This involves cross-checking the class diagram with the detailed use-case descriptions, object collaboration diagram/interaction sequence diagrams and state transition diagrams to ensure that the associations and collaborations and responsibilities are consistent.

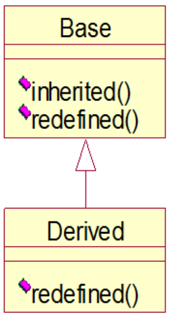
Test Cases and the Class Hierarchy

- Inheritance does not remove the need to test all derived classes.
- Example. Class *Derived* redefines *redefined()* to serve in a local context.
- *Derived::redefined()* must also be tested but does *Derived::inherited()* need to be tested?
  - If *Derived::inherited()* calls *Derived:: redefined()* then Derived::inherited() could misuse the new behaviour of *Derived:: redefined()* and therefore must be tested as part of *Derived*.



Test Cases and the Class Hierarchy

- *Base::redefined()* and *Derived::redefined()* are different as each implements different requirements and therefore require a separate set of test cases.
- The similarities in the operation will lead to similarities in the test cases.
- The better the OO design, the better the overlap in the inherited features.
  - New test will be required only for the **additional** *Derived::redefined()* requirements



Scenario-Based Test Design

- Scenario-based testing concentrates on what the user does (rather than what the built product does).
  - This means capturing the tasks (via use-cases) that the user has to perform, then applying them and their variants as tests. (essentially black box)

Final Steps of OO Testing

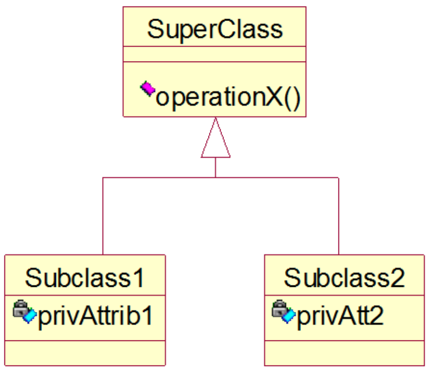
- As classes are integrated to form subsystems, thread-based, use-based and cluster-based (see next slides) testing along with fault-based approaches are applied to fully exercise collaborating classes.
  - Collaboration or Interaction errors are targeted by exercising multiple subsystems in a single test case.

OO Testing Strategies: Class/Unit Testing

- The classical strategy for testing structured software is unit testing (smallest indivisible module, subroutine etc.) leading progressively up to integration testing
  - these tests require regression testing to check for the effects of adding new components, major bug fixes, finishing with system testing and validation.
- In OO context unit testing is a different concept.
- Encapsulation of *attributes* and *operations* into a class dictates that a class should be tested as an indivisible unit in conjunction with other collaborating classes.

OO Testing Strategies: Class/unit Testing

- Testing the operationX() in isolation may not uncover the subtle differences that it encounters with the environment of each subclass and the private attributes of its environment.
- Class testing is driven by the operations and attributes encapsulated within the class along with the state behaviour.

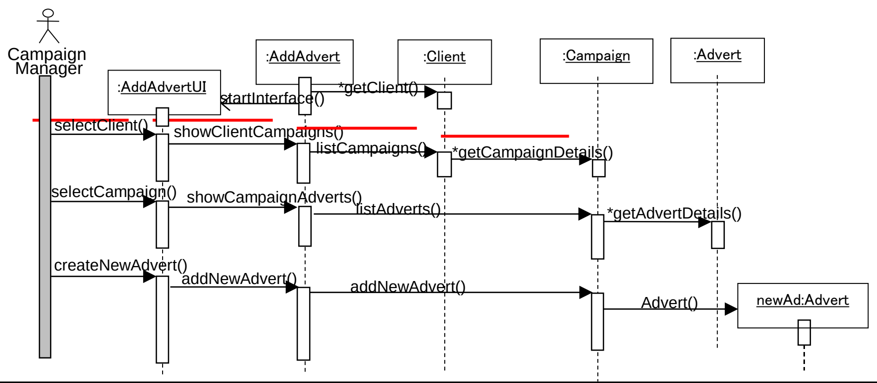


OO Testing Strategies: Integration Testing

- Because OO software does not consist of a hierarchy of program modules as in classical structured design, top down and bottom up testing is not applicable.
- Further, adding operations one at a time and testing the class after each addition (incremental testing) is often ruled out due to the complexity of the collaborations required.

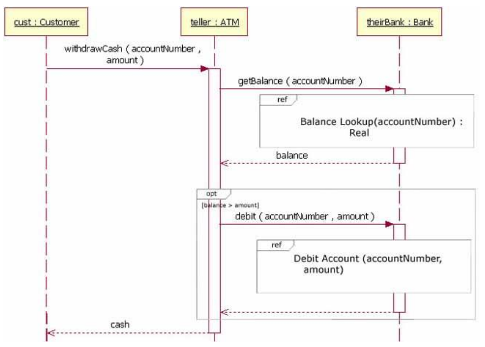
OO Testing Strategies: Thread Based Integration Testing

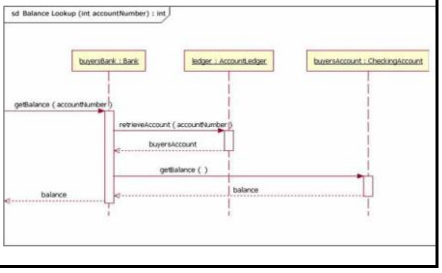
- The classes involved in one incoming event or input message (taken from the Interaction Sequence Diagram) for thread and are assembled and tested.
- As threads are integrated, regression tests are done to check for any side effects of the integration.



Cluster Testing

- Grouping together cooperating classes that have similar functions or work together to accomplish a particular goal.
  - Typically the functionality of these classes would have already been tested using dummy drivers or test harness software.
  - This cluster may be modelled as a referenced sequence diagram.


cust : Customer
teller : ATM
theirBank : Bank
withdrawCash (accountNumber , amount)
getBalance (accountNumber)
ref
Balance Lookup(accountNumber) : Real
balance
opt
[balance > amount]
debit (accountNumber , amount)
ref
Debit Account (accountNumber, amount)
cash


sd Balance Lookup (int accountNumber) : int
buyersBank : Bank
ledger : AccountLedger
buyersAccount : CheckingAccount
getBalance (accountNumber)
retrieveAccount (accountNumber)
buyersAccount
getBalance ( )
balance
balance

Test Cases for a Class

- The test should drive a selection of objects from the class through its states and for each test, the following characteristics should be tested for:
  - List of states for the class
  - List of messages and operations invoked
  - List of exceptions occurring

- The purpose of the test should be stated.

Test Driven Development

Firstly - Unit Testing

- Goal
  - Isolate individual parts of the program and show that they behave correctly
- Benefits
  - A unit test explicitly specifies conditions that the individual program parts must satisfy.
  - This facilitates uncovering issues at an early stage and also ease in testing future modifications to ensure existing tests are still successful.
  -
- Limitations
  - Testing cannot be expected to catch every error in the program

Traditional Approach to Unit Testing

- Write the code
  - Write your implementation – e.g. java methods
- Test the code
  - Write a class that instantiates an object of the class and tests the methods by executing them and checking results etc.
  - Fix any problems encountered…

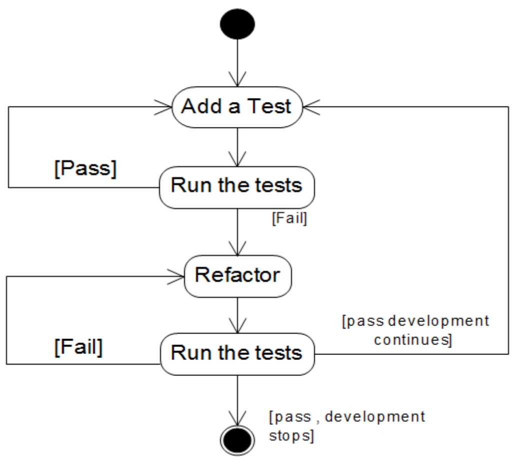
Test Driven Development (TDD)

from www.martinfowler.com

Test Driven Development (TDD) is a *design technique* that drives the development process through testing. In essence you follow three simple steps repeatedly:

1. Write a test for the next bit of functionality you want to add.
2. Write the functional code until the test passes.
3. Refactor both new and old code to make it well structured.

Test Driven Development (TDD)



Why TDD

- Code Modularization, Flexibility, Extensibility
- Clean code
- Leads to better design
- Better code documentation
- More productive

What to do…

- A design model will have defined our classes (attributes and *operations*).
- Tools can generally automatically create the skeleton code from the class diagram (e.g *MyClass.java*) – where methods have return values, one line of code is inserted which returns a null.
- Next we can write a test class (e.g. *MyClassTest.java*) within which we write methods that test instances of our actual class.
- Remember, none of our methods actually have any code – so the tests should all fail.
- Take one test at a time and go back to our actual class and write just enough code to pass that test.

Using Mock Objects in Testing

- When writing Unit tests, there can be a number of collaborating objects that are required to be available in order to test a given class.
  - Objects can be passed as parameters to the method under test.
  - The method under test may use objects which are attributes of its class.
- The test class needs to create the necessary instances of these objects so that the method under test can be called in an appropriate context
- This can be a challenging obstacle when testing non-trivial classes (object creation, attribute setting etc.).

Using Mock Objects in Testing

- Remember – when we are writing unit tests, we are only interested in testing the unit and *not* any other units that it requires to perform its function.
- What if we could ‘mock up’ any other objects that our object under test requires.
- This would free us up from being dependent on those other objects.

```java
public class IncomeCalculator {

   private ICalcMethod calcMethod;
   private Position position;

   public void setCalcMethod(ICalcMethod calcMethod) {
       this.calcMethod = calcMethod;
   }

   public void setPosition(Position position) {
       this.position = position;
   }

   public double calc() {
   if (calcMethod == null) {
       throw new RuntimeException("CalcMethod not yet maintained");
   }
   if (position == null) {
       throw new RuntimeException("Position not yet maintained");
   }
   return calcMethod.calc(position);
   }
}
```

Note, this attribute is an interface type (“coding to the interface”)

The attribute is used here

Example taken from: http://www.vogella.com/articles/EasyMock/article.html

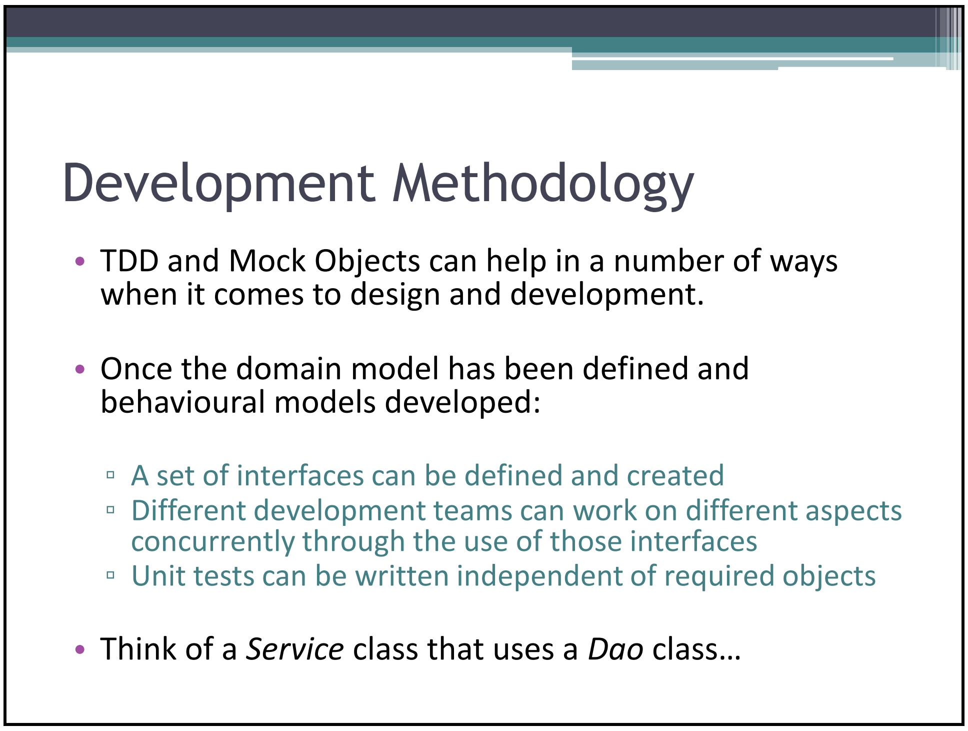
Test Code - Creating a Mock Object

```
@Before
public void setUp() throws Exception {
    calcMethod =
    EasyMock.createMock(ICalcMethod.class);

    calc = new IncomeCalculator();

    calc.setCalcMethod(calcMethod);
}
```

```
public void testCalc1() {

   // Setting up the expected value of the method call calc
   expect(calcMethod.calc(Position.BOSS)).andReturn(70000.0).times(2);
   expect(calcMethod.calc(Position.PROGRAMMER)).andReturn(50000.0);
   // Setup is finished need to activate the mock
   replay(calcMethod);

   calc.setCalcMethod(calcMethod);
   try {
       calc.calc();
       fail("Exception did not occur");
   } catch (RuntimeException e) {

   }

   calc.setPosition(Position.BOSS);
   assertEquals(70000.0, calc.calc(), 0);
   assertEquals(70000.0, calc.calc(), 0);

   calc.setPosition(Position.PROGRAMMER);
   assertEquals(50000.0, calc.calc(), 0);

   calc.setPosition(Position.SURFER);
   verify(calcMethod);
}
```

This sets up our mock object so that when we run the test the mocking framework will execute the mock object behaviour as specified

We can verify that the mock object was utilised as expected.



Development Methodology

- TDD and Mock Objects can help in a number of ways when it comes to design and development.
- Once the domain model has been defined and behavioural models developed:
  - A set of interfaces can be defined and created
  - Different development teams can work on different aspects concurrently through the use of those interfaces
  - Unit tests can be written independent of required objects
- Think of a *Service* class that uses a *Dao* class…