CS1632, Lecture 9: Unit Testing, part 2

Wonsun Ahn

How would you Unit Test this Method?

```
public class Example {
    public static int doubleMe(int x) {
        return x * 2;
    }
}
```

Perhaps something like this…

```
@Test
public void zeroTest() {
   assertEquals(0, Example.doubleMe(0));
}
@Test
public void positiveTest() {
   assertEquals(20, Example.doubleMe(10));
}
@Test
public void negativeTest() {
   assertEquals(-8, Example.doubleMe(-4));
}
```

OK, how about this?

```
public class DoALot {
  public void quackALot(Duck duck, int num) {
    for (int j=0; j < num; j++) {
       duck.quack();
    }
  }
}
```

1. There is no return value! What should we test then?!
   - ☛ Test the behavior: somehow test `quack()` is called `num` times
2. We are testing `Duck` class along with `DoALot` class. How do we avoid this?
   - – `Duck` may not even be implemented yet
   - – Even if it were, we don't want to test `Duck` code --- we want tests *localized*
   - ☛ Use a "body double" for `Duck` that fakes a real duck

Advance Unit Testing Techniques

- Removing Class Dependencies
  - Test Doubles
  - Stubs
- Behavior Verification
  - Mocks

Advance Unit Testing Techniques

- Removing Class Dependencies
  - Test Doubles
  - Stubs
- Behavior Verification
  - Mocks

Test Doubles

- "Fake" objects used in place of real objects to test the target class method
  - These are objects of external classes that the target class method references (e.g. Method parameters, member variables, etc.)

- Goal: To *not execute code* in the real object as part of the test
  - Means target class can be tested without external classes having been implemented
  - Means if a defect is found, it is *localized* to within the target class
  - Means any changes in external classes will not impact the test

- Caveat: Double should *appear* like the real thing to target class
  - Even if double does not execute code in the external class
  - Double should emulate the real object in some state according to the test scenario

Test Double Examples

1. Doubled database object: for testing without a DB installation
   - Double doesn’t actually connect to a database
   - Double returns pre-determined database entries for testing
2. Doubled file object: for emulating failures hard to do with a real file
   - Double doesn’t actually read a file from the hard disk
   - Double emulates file read failures that are hard to trigger in a real hard disk
3. Doubled RandomNumberGenerator: for reproducible testing
   - Double doesn’t actually generate random numbers
   - Double returns pre-determined numbers to make SW deterministic

Double External Classes (NOT the Tested Class)

- Double objects of external classes that the tested class depends on

- Don't double the tested class!
  - If you double it, you would be testing the faked object and not the real one
  - Defeats the entire purpose of doing the test

JUnit Example without Test Double

```
public class LinkedListUnitTest {
    @Test
    public void testDeleteFrontOneItem() {
        LinkedList<Integer> ll = new LinkedList<Integer>();
        ll.addToFront(new Node<Integer>);
        ll.deleteFront();
        assertNull(ll.getFront());
    }
}
```

- We want to test `LinkedList`; we don't want to test `Node`
- But a defect in `Node` may cause test to fail → not a true unit test!

JUnit Example with Test Double

```
public class LinkedListUnitTest {
    @Test
    public void testDeleteFrontOneItem() {
        LinkedList<Integer> ll = new LinkedList<Integer>();
        ll.addToFront(Mockito.mock(Node.class));
        ll.deleteFront();
        assertNull(ll.getFront());
    }
}
```

- Test double `Node` with `Mockito.mock` API
- Do not double `LinkedList` because that is the target class for testing

What does Mockito.mock(Node.class) create?

new Node

```
// Member variables
Node nextNode;
int data;

// Member methods
void setData(int d) {
    data = d;
}
int getData() {
    return data;
}
…
```

Mockito.mock(Node.class)

```
// Member variables
// NONE!

// Empty methods (stubs)
void setData(int d) {

}
int getData() {
    return 0;
}
…
```

Advance Unit Testing Techniques

- Removing Class Dependencies
  - Test Doubles
  - Stubs
- Behavior Verification
  - Mocks

Stubs

- Test doubles are “fake objects”
- Stubs are “fake methods” for the “fake objects”

Test Doubles are Born with Default Stubs

```
Mockito.mock(Node.class)

// Member variables
// NONE!

// Default stubs
void setData(int d) {

}
int getData() {
   return 0;
}
…
```

- Default stubs are empty methods
  - Return type `void`: Does nothing
  - Return type `int`: returns `0`
  - Return type `boolean`: returns `false`
  - Return type `Object`: returns `null`
  - Code of original class is never executed
- You can modify these default stubs
  - Modified stubs will still be empty
  - But can have it return a different value (E.g. have it return `1` instead of `0`)
  - Make double behave the way you want it

Stub Example

```
public class DoALot {
    public int quackALot(Duck duck, int num) {
        int numQuacks = 0;
        for (int j=0; j < num; j++) {
            numQuacks += duck.quack();
        }
        return numQuacks;
    }
}
```

- First, mock `Duck` to localize test to `DoALot`
- Then, stub `duck.quack()` to make it return the desired value

Stubbing `quack()` to Return 1

```
public class TestDoALot {
    @Test
    public void testQuackALot100() {
        DoALot doALot = new DoALot();
        Duck mockDuck = mock(Duck.class);
        when(mockDuck.quack()).thenReturn(1);
        int val = doALot.quackALot(mockDuck, 100);
        assertEquals(100, val);
    }
}
```

- Made the mocked `Duck` quack once per `quack()` by stubbing it

Stubbing `quack()` to Return 2

```
public class TestDoALot {
    @Test
    public void testQuackALot200() {
        DoALot doALot = new DoALot();
        Duck mockDuck = mock(Duck.class);
        when(mockDuck.quack()).thenReturn(2);
        int val = doALot.quackALot(mockDuck, 100);
        assertEquals(200, val);
    }
}
```

- See if `quackALot` still works well when `quack()` returns 2

Stubs Enables Control of Mocked Objects

- Remember, mocked objects have no state; trick is to *appear* to have state
- Given the following class:

```
public class Node {
  private int data;
  public void setData(int d) { data = d; }
  public int getData() { return data; }
  public int getDouble() { return data*2; }
}
```

  - Which methods would you stub to have `Node` appear to have 10 as `data`?
  - Hint: Not `setData(int d)`. Since it doesn't have return value, cannot stub!

Stub “get” Methods to Emulate Real Objects

Real Object

```
// Member variables
int data;
// Member methods
void setData(int d) {
   data = d;
}
int getData() {
   return data;
}
int getDouble() {
   return data*2;
}
```

Mocked Object (for when data == 10)

```
// Member variables
// NONE!
// Member methods
void setData(T d) {
   // Cannot stub
}
int getData() {
   return 10; // Stubbed
}
int getDouble() {
   return 20; // Stubbed
}
```

Stub “get” Methods to Emulate Real Objects

Real Object

```
// Member variables
int data;
// Member methods
void setData(int d) {
   data = d;
}
int getData() {
   return data;
}
int getDouble() {
   return data*2;
}
```

Mocked Object (for when data == 20)

```
// Member variables
// NONE!
// Member methods
void setData(T d) {
   // Cannot stub
}
int getData() {
   return 20; // Stubbed
}
int getDouble() {
   return 40; // Stubbed
}
```

Incorrect Stubbing Results in Incorrect Object

Real Object

```
// Member variables
int data;
// Member methods
void setData(int d) {
   data = d;
}
int getData() {
   return data;
}
int getDouble() {
   return data*2;
}
```

Mocked Object (stubbed incorrectly)

```
// Member variables
// NONE!
// Member methods
void setData(T d) {
   // Cannot stub
}
int getData() {
   return 2; // data==2?
}
int getDouble() {
   return 2; // data==1?
}
```

What if Behavior is Too Complex to Stub?

```
public class Duck {
  private boolean alive = true;
  public void shoot(int distance) {
    boolean hit = …; // complex trajectory calculation
    if( hit ) alive = false;
  }
  public String toString() {
    return alive ? “alive!” : “dead!”;
  }
}
public HuntingTrip {
  public String hunt(Duck duck, int distance) { // Tested method
    String ret = duck.toString());  // duck.toString() returns “alive!”
    duck.shoot(distance);           // duck.alive == false in test case
    return ret + duck.toString());  // duck.toString() returns “dead!”
  }
}
```

- We want to mock duck. But impossible to stub `duck.toString()` to emulate real duck.

Attempt to Stub with “alive!” (Fail)

```
@Test
public void testShootDuck10Yards() {
   HuntingTrip trip = new HuntingTrip();
   // Create a mock Duck and stub
   Duck mockDuck = Mockito.mock(Duck.class);
   when(mockDuck.toString()).thenReturn(“alive!”);
   // Duck should be hit since it’s only 10 yards away
   String ret = trip.hunt(mockDuck, 10);
   // Expected behavior is “alive!dead!”
   assertEquals(“alive!dead!”, ret);
}
```

- Fail because `ret == “alive!alive!”` at the end of execution

Attempt to Stub with “dead!” (Fail)

```
@Test
public void testShootDuck10Yards() {
   HuntingTrip trip = new HuntingTrip();
   // Create a mock Duck and stub
   Duck mockDuck = Mockito.mock(Duck.class);
   when(mockDuck.toString()).thenReturn(“dead!”);
   // Duck should be hit since it’s only 10 yards away
   String ret = trip.hunt(mockDuck, 10);
   // Expected behavior is “alive!dead!”
   assertEquals(“alive!dead!”, ret);
}
```

- Fail because `ret == “dead!dead!”` at the end of execution
- Unable to stub because object state must change in the middle of the test

Create a “Fake” Instead

```
@Test
public void testShootDuck10Yards() {
    HuntingTrip trip = new HuntingTrip();
    // Create a “fake” Duck that is near us
    Duck fakeDuck = new FakeDuckNear();
    // Duck should be hit since it’s only 10 yards away
    String ret = trip.hunt(fakeDuck, 10);
    assertEquals(“alive!dead!”, ret);
}
// A simplified Duck with no complex trajectory calculation
public class FakeDuckNear extends Duck {
   // private boolean alive; inherited from Duck
   // Always hits, emulating a Duck that is close by
   public boolean shoot(int distance) { alive = false; }
}
```

- *Fake*: a test double that is a simplified version of the original object

Create a Different Fake for a Different Scenario

```
@Test
public void testShootDuck1000Yards() {
    HuntingTrip trip = new HuntingTrip();
    // Create a "fake" Duck that is far away from us
    Duck fakeDuck = new FakeDuckFar();
    // Duck should not be hit since it's a 1000 yards away
    String ret = trip.hunt(fakeDuck, 1000);
    assertEquals("alive!alive!", ret);
}
// A simplified Duck with no complex trajectory calculation
public class FakeDuckFar extends Duck {
   // private boolean alive; inherited from Duck
   // Always misses, emulating a Duck that is far away
   public boolean shoot(int distance) { alive = true; }
}
```

- Yes, we could have used a mocked Duck for this test case. But you get the point.

Advance Unit Testing Techniques

- Removing Class Dependencies
  - Test Doubles
  - Stubs
- Behavior Verification
  - Mocks

Original Problematic Example

```
public class DoALot {
  public void quackAlot(Duck duck, int num) {
    for (int j=0; j < num; j++) {
       duck.quack();
    }
  }
}
```

1. There is no return value! What should we test then?!
   - ☛ Test the behavior: somehow test `quack()` is called `num` times
2. We are testing Duck along with DoALot. How do we avoid this?

Behavior Verification

- No relation to "verification" in "verification and validation"
- *State Verification* vs. *Behavior Verification*
  - *State Verification*: Tests the state of the program
    - Whether state changes correctly as a result of method call(s)
    - Done through assertions on postconditions (what we've done so far)
  - *Behavior Verification*: Tests the behavior of code
    - Whether certain methods have been called a certain number of times
    - Whether methods have been called with the correct parameters
    - Done through verify in Mockito

Why do Behavior Verification?

- In the end, we are interested in *state* not *behavior*
  - Whether method returns correct value → state verification
  - Whether method updates heap object correctly → state verification
  - Whether method outputs correct message → state verification
  - We are less interested in what methods were called internally
- Given a choice, always do *state* over *behavior* verification
- But what if you need to verify state within a mocked object?
  - Remember, mocked objects don't have any state to begin with!
  - This is when you need to do behavior verification

When State Verification does not Work

```
public class Duck {
  private int quacks = 0;
  public void quack() { quacks++; }
  public int getQuacks() { return quacks; }
}
public class DoALot {
  public void quackAlot(Duck duck, int num) {
    for (int j=0; j < num; j++) {
       duck.quack(); // doesn't do duck.quacks++ (it's a mocked object)
    }
  }
}
@Test public void testQuackAlot() {
  DoALot doALot = new DoALot();
  Duck mockDuck = Mockito.mock(Duck.class);
  int quack = doALot.quackAlot(mockDuck, 5);
  assertEquals(5, mockDuck.getQuacks();  // returns 0, stub for getQuacks()
}
```

Behavior Verification to the Rescue!

```
@Test public void testQuackAlot() {
    DoALot doALot = new DoALot();
    // Make a mocked Duck, stub quack()
    Duck mockDuck = mock(Duck.class);
    // Call quackAlot, which calls mockDuck.quack() 5 times
    doALot.quackAlot(mockDuck, 5);
    // Verifies quack called 5 times on mockDuck
    Mockito.verify(mockDuck, times(5)).quack();
    // Note no assertions!  Assertions built in to verify.
}
```

- `Mockito.verify` does not directly check that `mockDuck` has the correct state
- But, had a real duck with correct code been used, it would have the correct state

Mock

- *Mock*: A test double which uses behavior verification

- Many frameworks uses the same API for doubles and mocks
  - `Mockito.mock` is used to create both doubles and mocks
  - If mocked object later uses behavior verification, it's a mock
  - If object happens to not use behavior verification, it's a double

- But technically, a mock is a specific kind of test double.

More JUnit / Mockito Examples

- sample_code/ junit_example/LinkedListUnitTest.java
  - Replica of LinkedListTest.java we saw in Part 1 but using Mockito
  - LinkedListTest: not a true unit test due to testing Node alongside LinkedList
  - LinkedListUnitTest: true unit test that mocks all Node instances

- Above uses Mockito.verify to check method call parameters
  - That Node methods are called properly from LinkedList
  - ```
    ll.addToFront(existingNode);
    ll.addToFront(testNode);
    Mockito.verify(testNode).setNext(existingNode);
    ```
    - ☛ Checks `testNode.setNext(existingNode)` is called in `addToFront`
    ```

Unit Testing Summary

What does a Good Unit Test Look Like?

- Reproducible on every run
- Independent of other tests
- Is localized (tests only the unit)
- Tests one test case at a time

Good Unit Test: Reproducible on Every Run

- Tests should either always pass or always fail. Otherwise:
  - When a test fails, it may be hard to reproduce the defect for debugging
  - When a test passes, there is no guarantee defect will not resurface later
- That means …
  - All preconditions must be precise and complete
  - There can be no random factor in the execution steps
    - No randomness in the test itself (e.g. testing a random input value)
    - No randomness internal to the program (e.g. game with a die roll)
- How do we remove randomness internal to the program?!
  - Don't worry, we will learn when we talk about Writing Testable Code ☺

Good Unit Test:
Independent of Other Tests

- Tests should not depend on other tests to run. Why?
  - We may choose to run a subset of tests in a test suite
  - We may choose to run tests in a different order (e.g. in parallel)
  - That other test may fail (which will cause this test to fail also)

- We learned how to create a test fixture using @Before, @After

Good Unit Test:
Localized (Tests only the unit)

- In other words, mock external classes and stub the methods

- We talked enough about this so no need to belabor

Good Unit Test:
Tests one thing at a time

- Do not test different test cases in a single test. Why?
  - If a test case fails (assertion fires), remaining test cases aren't tested
  - On test failure, hard to tell which test case failed

- Means you should try to call only one method from test
  - The one that you are testing
  - Except when you need to call other methods to set up preconditions
  - Except when you need to call other methods to check postconditions

JUnit is not the only unit test framework out there!

- Not even for Java!
- But xUnit frameworks are common and easy to understand
  - C++: CPPunit
  - JavaScript: JSUnit
  - PHP: PHPUnit
  - Python: PyUnit
- Ideas should apply to other testing frameworks easily

Unit Testing != System Testing

- The manual tests that you've done for Exercise 1 is a system test
  - Checks that the whole system works
- The automated tests that you will do for Exercise 2 are unit tests
  - Checks that each unit of functionality individually works
- A proper testing process includes both
  - Unit tests to detect local errors within units of code
  - System tests to check that all pieces of code work together correctly

My advice

- Try to add tests as soon as possible.
  - Ideally, write tests before coding
  - Will cover in our next chapter “Test Driven Development”.

- Develop in a way to make it easy for others to test.
  - E.g. if you create an external object inside a method, much harder to mock

```
public class RentACat {
  public void addCat() {
    Cat cat = new Cat(1, “cat”); // How can we mock this?
    list.add(cat);
  }
}
```

  - Will cover in our next, next chapter “Writing Testable Code”

Now Please Read Textbook Chapter 14

- In addition, look at code using Mockito in our JUnit example: sample_code/junit_example/LinkedListUnitTest.java

- Mockito User Manual:

https://javadoc.io/static/org.mockito/mockito-core/3.2.4/org/mockito/Mockito.html