CS1632, Lecture 9: Unit Testing, part 2

Wonsun Ahn

How to test this method?

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

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

OK, how about this?

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

1. What is there to test to begin with? There are no values to test!
   - ☛ Test the behavior: somehow test `quack()` is called `num` times
2. How can we test `Example` class without `Duck` class?
   - – `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 you can use in your tests
- They can act in any way you want – they do not have to act exactly as their “real” counterparts

Test Double Examples

1. A doubled database connection
   - Double doesn’t actually connect to a database
   - Double returns pre-determined database entries for testing
2. A doubled File object
   - Double doesn’t actually open a file
   - Double emulates file read failures for the purposes of testing
3. A doubled RandomNumberGenerator
   - Double doesn’t actually generate random numbers
   - Double returns pre-determined numbers for reproducible testing

Double Dependent Class (NOT the Tested Class)

- Remember, double objects of classes that the tested class depends on
  - That is because we don’t want to test dependent classes
- Don't double the tested class!
- If you double the tested class, what are you testing? ☺

Test Double Example

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

- We want to test `LinkedList`; we don't want to test `Node`
- Test double `Node` with JUnit `mock` API

Advance Unit Testing Techniques

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

Stubs

- Doubles are “fake objects”

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

Stubs

- Stubbing a method says:
  - “Instead of calling that method, just do whatever I tell you.“
- "Whatever I tell you" is just return a value
  - Actual method is never executed

Stub Example

```
public int quackAlot(Duck d, int num) {
  int numQuacks = 0;
  for (int j=0; j < num; j++) {
    numQuacks += d.quack();
  }
  return numQuacks;
}
```

- We want to stub `quack()` to remove dependency on `Duck`

Create a test double, stub the method

```
@Test
public void testQuackAlot() {
    Duck mockDuck = Mockito.mock(Duck.class);
    Mockito.when(mockDuck.quack()).thenReturn(1);
    int val = quackAlot(mockDuck, 100);
    assertEquals(val, 100);
}
```

We have Localized the Test

- We only care about testing our quackAlot() method
  - We don't care about whether Duck.quack() works, or Duck works
  - Duck.quack() is tested separately in the unit tests for Duck class

- Unit tests should only test the unit being tested
  - Otherwise, test becomes BRITTLE (breaks easily due to external changes)
  - Otherwise, on failure, hard to pinpoint where defect occurred

What if you don’t stub a method?

- What if you don’t stub a method for a test double?
  - Still, original method is NOT executed
  - Returns a default value
    - E.g. If return type is boolean: false
    - E.g. if return type is int: 0
    - E.g. if return type is reference: null
- What if the method has return type `void`?
  - No need to stub since there is no return value

Advance Unit Testing Techniques

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

Original Problematic Example

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

1. What is there to test to begin with? There are no values to test!
   - ☛ Test the behavior: somehow test `quack()` is called `num` times
2. How can we test Example class without Duck class?

Behavior Verification

- No relation to "verification" in "verification and validation"
- Behavior Verification vs. State 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

Mock

- Mock: A test double which uses behavior verification

- Many frameworks (such as Mockito, the one we are using) don't differentiate between doubles and mocks
- Technically, a mock is a specific kind of test double.

Mock Example

```
@Test
public void testQuackAlot() {
   // Make a double of Duck, stub quack()
   Duck mockDuck = Mockito.mock(Duck.class);
   Mockito.when(mockDuck.quack()).thenReturn(1);
   // Call quackAlot, which calls mockDuck.quack() 5 times
   quackAlot(mockDuck, 5);
   // Make a true mock by verifying quack called 5 times
   Mockito.verify(mockDuck, Mockito.times(5)).quack();
   // Note no assertions!  Assertions built in to verify.
}
```

What if Method is Too Complex to Stub?

```
public class Duck {
  boolean alive = true;
  public void shoot() {
    boolean hit = …; /* complex trajectory calculation */
    if( hit ) alive = false;
  }
  public String toString() {
    return alive ? “alive” : “dead”;
  }
}
public DuckHunt {
  public void shootDuck(Duck d) {
    System.out.println(d.toString());  // should return “alive”
    d.shoot();
    System.out.println(d.toString());  // should return “dead”
  }
}
```

- Why is stubbing `d.toString()` hard to do?

Create a Fake

```
public class FakeDuck extends Duck {
  // Doesn’t do complex trajectory calculation
  public boolean shoot() { alive = false; }
}
@Test
public void testShootDuck() {
   // Make a fake Duck
   Duck fakeDuck = new FakeDuck();
   // Call shootDuck with fake Duck
   shootDuck(fakeDuck);
}
```

- Fake: a particular kind of double that is a simplified version of original object

What does a Good Unit Test Look Like?

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

Good Unit Test: Reproducible on Every Run

- Test should either always pass or always fail. Why?
  - Otherwise, impossible to tell which build or version caused defect
  - The defect may have crept in long time ago but we were just lucky
- That means …
  - All preconditions must be set up correctly before running each test
  - There can be no random factor while running test
    - No randomness in the test itself (e.g. passing 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 be impacted by other tests. Why?
- We may choose to run a subset of tests in a test suite
  - If a test that this test depends on is not chosen …
- We may choose to run tests in a different order
  - We may even choose to run them in parallel!
  - Most unit testing frameworks allow parallel execution for faster completion
- Allows completion of test suite even in the event of failure
  - Even if a test fails, it does not impact any other tests

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 call only one application method from test
  - The one that you are testing (unless it is unavoidable)
- If you use an “if..else” in a test, this is a code smell!
  - Value returned from a call used for something other than an assertion (for example, deciding which app method to call next, a no-no)

Good Unit Test:
Localized (Tests only the unit)

- Only test the code within the unit and nothing else. Why?
  - If test fails, you are sure the defect lies in the unit code
  - Otherwise, the defect may be in that “other” code you included
    ☛ Defeats the entire purpose of unit testing!

- What if the unit (method) depends upon other methods?
  - Test double the object and stub the method

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

My advice

- Try to add tests as soon as possible. DO NOT WRITE ALL OF YOUR CODE AND THEN TRY TO ADD TESTS.
- Ideally, write tests before coding (TDD).
- Develop in a way to make it easy for others to test.
- In legacy systems, add tests as you go. Don't fall into the morass!

Unit Testing != System Testing

- The manual testing that you’ve already done is a system test – it checks that the whole system works.
- This is not the goal of unit tests! Unit tests check that very small pieces of functionality work, not that the system as a whole works together.
- A proper testing process will include both –unit tests to pin down errors in particular pieces of code, system tests to check that all those supposedly-correct pieces of code work together.

Now Please Read Textbook Chapter 14

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

- For reference, the Mockito Javadoc:

https://javadoc.io/doc/org.mockito/mockito-core/latest/index.html