CS1632, Lecture 10: 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 doDuckStuff(Duck d, boolean q){
    if (q) {
       d.quack();
    } else {
       d.eat();
    }
  }
}
```

1. How can we test Example.doDuckStuff() without Duck?
2. What is there to test to begin with? There are no values to test!

Advance Unit Testing Techniques

- Doubles
- Stubs
- Mocks and Behavior Verification

Advance Unit Testing Techniques

- Doubles
- Stubs
- Mocks and Behavior Verification

Dependency on other classes == bad

- Why?
  - If a failure occurs in a test, where is the problem?
    - This method?
    - Other method?
    - Yet another method that another method called?
    - Cannot be *localized*
  - What if quack() hasn't been completed yet?

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

examples

1. A doubled database connection, so you don't need to actually connect to the database
2. A doubled File object, so you can test read/write failures without actually making a file on disk
3. A doubled RandomNumberGenerator, so you can always produce the same number when testing

Doubles help keep tests localized

- They let you test only the item under test, not the whole application, allowing you to focus on the current item.
- Remember, double objects of classes that the current class depends on; don't double the current class!
  - That would mean you are making a “fake” version of what you are testing

example

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

Advance Unit Testing Techniques

- Doubles
- Stubs
- Mocks and Behavior Verification

stubs

- Doubles are “fake objects”.
- Stubs are “fake methods”.

stubs

- Stubbing a method says "hey, instead of actually calling that method, just do whatever I tell you.“
- "Whatever I tell you" is usually just return a value.

Example

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

Create a test double, stub the method

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

We have made the test independent

- 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

Advance Unit Testing Techniques

- Doubles
- Stubs
- Mocks and Behavior Verification

OK, how about this?

```
public class Example {
  public void doDuckStuff(Duck d, boolean q){
    if (q) {
       d.quack();
    } else {
       d.eat();
    }
  }
}
```

1. How can we test Example.doDuckStuff() without Duck?
2. What is there to test to begin with? There are no values to test!

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.

Example - A Slightly Modified quackAlot()

```
public void quackAlot(Duck d, int num) {
   int throwaway = 0;
   for (int j=0; j < num; j++) {
      throwaway = d.quack();
   }
} // What can we test here?
```

Example test

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

Structuring unit tests

- Two philosophies:
  - Test only public methods.
    - This is the true interface to an object. We should be allowed to change the implementation details at will.
    - Private methods will be tested as a side effect of any public method calls.
    - Private methods may be difficult to test due to language/framework.
  - Test every method – public and private.
    - Public/private distinction is arbitrary – you still want it all to be correct.
    - Unit testing means testing the lowest level; we should test as close to the actual methods as possible.

example

```
class Bird {
    public int chirpify(int n) {
      return nirpify(n) + noogiefy(n + 1);
    }
    private int nirpify(int n) { … }
    private int noogiefy(int n) { … }
    // Dead code, can never be called!
    private void catify(double f) { … }
}
```

Another example

```
// Assume all the called methods are complex
public boolean foo(boolean n) {
  if (bar(n) && baz(n) && quux(n)) {
    return true;
  } else if (baz(n) ^ (thud(n) || baa(n)) {
    return false;
  } else if (meow(n) || chew(n) || chirp(n)) {
    return true;
  } else {
    return false;
  }
}
```

We have to come up with a decision!

- Like most software engineering decisions - it depends. I don't think that there is a right answer.
- That being said, for this class, we are going to follow the philosophy of testing all public methods, not private methods.
- If you are interested in testing private methods in Java, you need to use something called "reflection" – see Chapter 24 in AFIST

What inputs should I test on those methods?

- Ideally...
  - Each equivalence class
  - Boundary values
  - Any other edge cases
- And also failure modes
  - Failure modes: inputs where method is expected to fail
  - Failing where it should is also part of its requirements

What are The equivalence classes, boundary values, and failure modes we should test?

```
public int quack(int n) throws Exception {
  if (n < 10) {
    return 1;
  } else if (n < 20) {
    return 2;
  } else {
    throw new Exception("too many quacks");
  }
}
```

What if it is difficult to test things?

- It happens
- Especially when working with legacy code.
- Such is life.
- Don’t give up!

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.

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!

Now Please Read Textbook Chapter 14

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