CS1632, LECTURE 10: UNIT TESTING, PART 2

BILL LABOON

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();
        }
}
```

WE NEED MORE ADVANCED TECHNIQUES

- Doubles
- Stubs
- Mocks
- Verification

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);
}
```

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; }
}
```

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?

MOCK THE CLASS, 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 don't care about how quack() works, or Duck works, only our quackAlot() method.

If something goes wrong in Duck.quack, tests on THAT method will fail, not here.

Tests that break easily are called BRITTLE. If your tests depend on lots of other code working correctly, they are very brittle, and also make it difficult to know where the error actually is.

UNIT TESTS != SYSTEM TESTS

- 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.

VERIFICATION

- Note that this is different from the "verification" in "verification and validation". It's also different than the "verification" used when checking that a developer actually fixed a defect. So this is the third definition of the term "verification" in this course, and it shan't be the last.
- In this case, it means "verifying that a method has been called 0, 1, or n times."
- A test double which uses verification is called a Mock. However, many frameworks (such as Mockito, the one we are using) don't have a strong differentiation between doubles and mocks. Technically, though, a mock is a specific kind of test double.

WHAT IS VERIFICATION?

- I like to think of it as “an assertion on the execution of the code”

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);
      quackAlot(mockDuck, 5); // Execution

      // Note no assertions!  Assertions built in to verify

      // Make a true mock by verifying quack called 5 times
      Mockito.verify(mockDuck, times(5)).quack(); // make a true mock


      }
```

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.
    - Code is code. The 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 KINDS OF THINGS SHOULD I TEST ON THOSE METHODS?

- Ideally...
  - Each equivalence class
  - Boundary values
  - Failure modes
  - Any other edge cases

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!

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!