Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 24 — March 18, 2019

Patrick Lam *version 2*

We'll continue discussing how to engineer your test suites today. In particular, we'll see more details on how to make behaviour verification actually happen (using mock objects); we'll discuss the bane of flaky tests; and we'll talk about continuous integration.

Test Doubles

Mock objects are a particular kind of test double. We need test doubles because objects collaborate with other objects, but we only want to test one object at a time. Meszaros categorizes test doubles as follows:

- dummy objects: these are not actually test doubles; they don't do anything, but just take up space in parameter lists. Are like `null`, but get past nullness checks in code.
- fake objects: have actual behaviour (which is correct), but somehow unsuitable for use in production; typical example is an in-memory database.
- stubs: produce canned answers in response to interactions from the class under test.
- mocks: like stubs, also produce canned answers. Difference: mock objects also check that the class under test makes the appropriate calls.
- spies: usually wraps the real object (instead of the mock, which stubs it), and records interactions for later verification.

Shorter reference about test doubles: `martinfowler.com/articles/mocksArentStubs.html`

Mock Objects

Before we talk about mock objects, let's look at a stub. Imagine that you have a service that sends out emails. You don't actually want to send out emails while you're testing. So here's a class that pretends to send out emails.

```
public class MailServiceStub implements MailService {
  private List<Message> messages = new ArrayList<Message>();
  public void send (Message msg) {
    messages.add(msg);
  }
  public int numberSent() {
    return messages.size();
  }
}
```

This stub permits *state verification*, as seen in the following assert in a test:

```
  assertEquals(1, mailer.numberSent());
```

This is state verification because it's checking the contents of memory (which should reflect interactions that have happened in the past). One could also check the recipients, contents of messages, etc.

jMock example. Instead of state verification, we can also do behaviour verification. This is jMock syntax.

```
class OrderInteractionTester... {
  public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    Mock warehouse = mock(Warehouse.class);
    Mock mailer = mock(MailService.class);
    order.setMailer((MailService) mailer.proxy());

    mailer.expects(once()).method("send");
    warehouse.expects(once()).method("hasInventory")
      .withAnyArguments()
      .will(returnValue(false));

    order.fill((Warehouse) warehouse.proxy());
  }
}
```

The calls to `mock()` create mock objects which have the appropriate type. If you are using the objects as simple dummy objects, calling `mock()` and `proxy()` is enough. Note that we have a real `Order` object but we're giving it the fake proxy objects, as created by the `Mock`'s `proxy()` methods.

We also specify the expected behaviour of the `mailer` and the `warehouse`. The test case is saying that the mailer ought to have `send()` called on it once, and that the warehouse ought to have `hasInventory()` called; that method should return `false()`.

EasyMock example. Different mock object libraries have different syntax. Here's another example, this time for EasyMock.

```
@RunWith(EasyMockRunner.class)
public class ExampleTest {

  @TestSubject
  private ClassUnderTest classUnderTest = new ClassUnderTest();

  @Mock // creates a mock object
  private Collaborator mock;

  @Test
```

```
  public void testRemoveNonExistingDocument() {
    replay(mock);
    classUnderTest.removeDocument("Does not exist");
  }
}
```

Here we are testing the `ClassUnderTest` and creating a mock object of `Collaborator` type. EasyMock 2.3 reads the `@Mock` annotation and automatically fills in a mock object of the appropriate type. In our test case, we call `replay(mock)` to indicate that we are no longer recording expectations, but are instead starting the test case itself. In the above code, there are currently no expectations.

Let's add some expectations.

```
@Test
public void testAddDocument() {
  // ** recording phase **
  // expect document addition
  mock.documentAdded("Document");
  // expect to be asked to vote for document removal, and vote for it
  expect(mock.voteForRemoval("Document"))
            .andReturn((byte) 42);
  // expect document removal
  mock.documentRemoved("Document");
  replay(mock);
  // ** replaying phase ** we expect the recorded actions to happen
  classUnderTest.addDocument("New Document", new byte[0]);
  // check that the behaviour actually happened:
  verify(mock);
}
```

Here we record the fact that the mock should be called with `documentAdded` and a parameter "New Document". We also record that the mock's `voteForRemoval` method should be called, and when that happens, it should return value 42. Finally, we add a call `verify()` to let EasyMock know that we're done and that it can go ahead and check that the expected behaviour actually happened.

Flaky Tests

The second test engineering topic I want to talk about today is flaky tests. Flaky tests are those that sometimes fail (nondeterministically). Flakiness is not something you want in your test cases. (I have heard one defense of a flaky test: it lets you know that the system has the potential to actually work.) In general, flaky tests don't play well with the expectation that your test suite passes 100%.

Reference:
Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, Darko Marinov. "An Empirical Analysis of Flaky Tests". In Proceedings of Foundations of Software Engineering '14.

Dealing with flaky tests. Companies with large test suites have found mitigations for the flaky test suite problem. One can label known-flaky tests as flaky and automatically re-run them to see if they eventually pass. One can also ignore or remove flaky tests. But this is unsatisfactory: it takes a long time to re-run failing tests.

Causes of flakiness. Luo et al studied 201 fixes to flaky tests in open-source projects. They found that the three most common causes of fixable flaky tests were:

1. improper waits for asynchronous responses;
2. concurrency; and
3. test order dependency.

The problem that caused flakiness for asynchronous waits was that there was typically a `sleep()` call which didn't wait long enough for the action (perhaps a network call) to finish. The best practice is to use some sort of `wait()` call to wait for the result instead of hardcoding a sleep time.

Concurrency problems were what one might expect. The problem could either be in the system under test or in the test itself. Problems included data races, atomicity violations, and deadlocks; the solutions were the proper use of concurrency primitives (e.g. locks) as seen in your Operating Systems course.

Test order dependency problems arose when some tests expected other tests to have already executed (and left a side effect like a file in the filesystem). They came up especially in the transition from Java 6 to Java 7 because that transition changed the (not-guaranteed) test execution order. The solution is to remove the dependency.