Software Testing, Quality Assurance and Maintenance Winter 2017

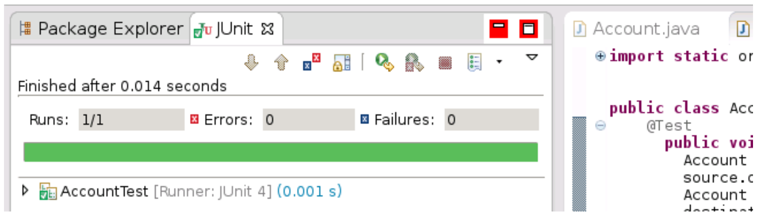
Lecture 25 — March 8, 2017

Patrick Lam version 1

We will see techniques for improving test design today, particularly with respect to verifying results.

Reference: Gerard Meszaros. xUnit Test Patterns: Refactoring Test Code. [Highly recommended. Available in the DC library.]

Goal. Well-designed tests are self-checking. That means that if the test runs with no errors and no failures (and hence produces a green bar in your IDE), we know that the test was successful.



```
Finished after 0.014 seconds
Runs: 1/1    Errors: 0    Failures: 0
AccountTest [Runner: JUnit 4] (0.001 s)
```

```
import static or

public class Acc
    @Test
    public voi
        Account
        source.c
        Account
```

Writing self-checking tests means that the tests automatically report the status of the code. This enables a “keep the bar green” coding style. Implications: 1) you can worry less about introducing bugs (but still take ordinary care); and 2) the tests help document your system’s specs.

How to write self-checking tests. One might think:

> “Isn’t it just calling asserts?”

Sadly, no. That’s not enough.

Two questions about actually deploying asserts:

- Q: what for?
  A: check method call results
- Q: where?
  A: usually after calling SUT (System Under Test)

Counter example. Here's some example code.

```java
public class Counter {
    int count;

    public int getCount() { return count; }
    public void addToCount(int n) { count += n; }
}
```

We can test it with the following JUnit test.

```java
// java -cp /usr/share/java/junit4.jar:. org.junit.runner.JUnitCore CounterTest
import static org.junit.Assert.*;
import org.junit.Test;

public class CounterTest {
  @org.junit.Test
  public void add10() {
      Counter c = new Counter();
      c.addToCount(10);
      // after calling SUT, read off results
      assertEquals("value", 10, c.getCount());
  }
}
```

What kind of test is this? Let's consider two kinds of tests: state-based tests vs. behaviour-based tests.

- **State:** e.g. object field values. Verify by calling accessor methods.
- **Behaviour:** which calls SUT makes. Verify by inserting observation points, monitoring interactions.

Does Counter Test verify state or behaviour?

Flight example. Here's more example code.

```java
// Meszaros, p. 471
// not self-checking
public void testRemoveFlightLogging_NSC() {
 // setup:
 FlightDto expectedFlightDto=createARegisteredFlight();
 FlightManagementFacade=new FlightManagementFacadeImpl();
 // exercise:
 facade.removeFlight(expectedFlightDto.getFlightNo());
 // verify:
 // have not found a way to verify the outcome yet
 //  Log contains record of Flight removal
}
```

Implementing State Verification

We can verify state:

```
// Meszaros, p. 471
// extended state specification
public void testRemoveFlightLogging_NSC() {
 // setup:
 FlightDto expectedFlightDto=createARegisteredFlight();
 FlightManagementFacade=new FlightManagementFacadeImpl();
 // exercise:
 facade.removeFlight(expectedFlightDto.getFlightNo());
 // verify:
 assertFalse("flight␣still␣exists␣after␣being␣removed",
             facade.flightExists(expectedFlightDto, getFlightNo()));
}
```

Note that we are exercising the SUT, verifying state, and checking return values.

In state-based tests, we inspect only outputs, and only call methods from SUT. We do not instrument the SUT. We do not check interactions.

You have two options for verifying state:

1. procedural (bunch of asserts); or,
2. via expected objects (stay tuned).

Returning to the flight example:

- We do check that the flight got removed.
- We don't check that the removal got logged.
- Hard to check state and observe logging.
- Solution: Spy on SUT behaviour.

Implementing Procedural Behaviour Verification

Or, we can implement behaviour verification. This is one way to do so, behaviourally:

```java
// Meszaros, p. 472
// procedural behaviour verification
public void testRemoveFlightLogging_PBV() {
 // fixture setup:
 FlightDto expectedFlightDto=createARegisteredFlight();
 FlightManagementFacade=new FlightManagementFacadeImpl();
 // test double setup:
 AuditLogSpy logSpy = new AuditLogSpy();
 facade.setAuditLog(logSpy);
 // exercise:
 facade.removeFlight(expectedFlightDto.getFlightNo());
 // verify:
 assertEquals("number␣of␣calls",
              1, logSpy.getNumberOfCalls());
 // ...
 assertEquals("detail",
              expectedFlightDto.getFlightNumber(),
              logSpy.getDetail());
}
```

As an alternative, we can use a mock object framework (e.g. JMock) to define expected behaviour.

Idea. Observe calls to the logger, make sure right calls happen.

Assertions

We build tests using assertions. In JUnit, there are three basic built-in choices:

1. assertTrue(aBooleanExpression)
2. assertEquals(expected, actual)
3. assertEquals(expected, actual, tolerance)

(There are others too, but let's start with these.)

`assertTrue` is more flexible, since you can write anything with a boolean value. However, it can give hard-to-diagnose error messages—you need try harder when using it if you want good tests.

Using Assertions. Why use assertions? Assertions are good for:

- checking all the things that should be true (more = better);
- serving as documentation: when system in state S_1, and I do X, assert that the result should be R, and that system should be in S_2.
- allowing failure diagnosis (include assertion messages!)

There are alternatives to using assertions. For instance, one can also do external result verification:

- write output to files; and
- use diff (or your own custom diff) to compare expected and actual output.

The twist is that the expected result is then not visible when looking at test's source code. (What's a good workaround?)

Verifying Behaviour. The key is to observe actions (calls) of the SUT. Some options for doing this:

- procedural behaviour verification (the challenge in that case: recording and verifying behaviour); or
- expected behaviour specification (capturing the outbound calls of the SUT).

How to Improve Your Tests

Next, we'll talk about some techniques for improving your test cases. Some tests are just better designed than others, making them easier to maintain and to understand. Applying these techniques will help.

Reducing Test Code Duplication. Copy-pasting is common when writing tests. This results in duplicate code in test cases, which has some undesirable side effects (bloat, unnecessary asserts). We'll talk about some techniques to mitigate duplication:

- Expected Objects
- Custom Assertions
- Verification Methods

Let's start with an example. One might expect many test methods like this one.

```
// Meszaros, p115
public void testInvoice_addLineItem7() {
  LineItem expItem = new LineItem(...);
  inv.addItemQuantity(product, QUANTITY);
  List lineItems = inv.getLineItems();
  LineItem actual = (LineItem) lineItems.get(0);
  assertEquals(expItem.getInv(), actual.getInv());
  assertEquals(expItem.getProd(), actual.getProd());
  assertEquals(expItem.getQuantity(),
               actual.getQuantity());
}
```

Using an Expected Object. We can compare objects instead:

```java
// Meszaros, p115
public void testInvoice_addLineItem8() {
  LineItem expItem = new LineItem(...);
  inv.addItemQuantity(product, QUANTITY);
  List lineItems = inv.getLineItems();
  LineItem actual = (LineItem) lineItems.get(0);
  assertEquals("Item", expItem, actual);
}
```

What we need:

- a way to create the Expected Object;
- a suitable `equals()` method.

Here are some potential barriers:

- we might need a special `equals()` method,
  e.g. to compare subset of fields; or,
- we may only have an `equals()` that checks identity; or,
- we can't create the desired expected object.

Some solutions:

- we can create a custom assertion; or,
- we can provide special `equals()` on expected object.

Custom Assertions Example. Consider the following code:

```java
// Meszaros, p116
static void assertLineItemsEqual(String msg, LineItem exp, LineItem act) {
  assertEquals(msg+"␣Inv", expItem.getInv(),
               actual.getInv());
  assertEquals(msg+"␣Prod", expItem.getProd(),
               actual.getProd());
  assertEquals(msg+"␣Qty", expItem.getQuantity(),
               actual.getQuantity());
}
```

Tips:

- Pick a good, declarative name.
- Create the custom assertion by refactoring, using usual techniques.

Benefits of Custom Assertions. Writing custom assertions can help with your test design. They:

- hide irrelevant detail;
- label actions with a good name (names are super important); and
- are themselves testable;

Variant: Outcome-describing Verification Method. Instead of using a custom assertion, you might use a verification method, like this one:

```java
// Meszaros, p117
static void assertInvoiceContainsOnlyThisLineItem(Invoice inv, LineItem exp) {
  List lineItems = inv.getLineItems();
  assertEquals("number␣of␣items", lineItems.size(), 1);
  LineItem actual = (LineItem)lineItems.get(0);
  assertLineItemsEqual("", expItem, actual);
}
```

Note that the verification method also interacts with SUT, but may have arbitrary parameters.

Going Further: Parameterized, Data-Driven Tests. While we're at it, there might be entire tests that differ only in input data.

Concrete tests invoke parametrized tests.

Other Best Practices for Tests

Avoid logic in tests.

The root problem is that tests are pretty much untestable. If you including ifs and loops in tests, you're asking for trouble. How are you going to make sure they're right?

Conditionals. For example,

```
// BAD
List lineItems = invoice.getLineItems();
if (lineItems.size() == 1) {
  // ...
} else {
  fail("Invoice should have exactly 1 line item");
}
```

Instead, do this:

```
// GOOD
List lineItems = invoice.getLineItems();
// (guard assertion:)
assertEquals("number of items", lineItems.size(), 1);
// ... proceed as before
```

The guard keeps you out of trouble.

Loops. Don't put loops directly in tests. Use a well-named, testable Test Utility Method instead.

Summary

In this lecture, we saw practical techniques for writing tests. This included techniques for result verification (using state verification and behaviour verification), as well as techniques for improving your tests by reducing duplication and by simplifying your tests.