Software Testing, Quality Assurance & Maintenance—Lecture 25

Patrick Lam

March 8, 2017

Today

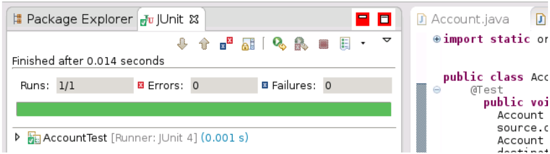
Result Verification for Tests.

Reference: Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*.

Goal

Good tests are *self-checking*:

no errors, no failures = successful test.



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
```

Why Self-Checking Tests?

Tests automatically report status.

Enables “keep the bar green”
coding style.

Worry less about introducing bugs.

Plus: Tests help document system specs.

Today's Plan

HOWTO make your tests self-checking.

“Isn’t it just calling asserts?”

“Isn’t it just calling asserts?”

[sadly, no.]

Two questions about asserts:

1. Q: what for?
   A: check method call results

2. Q: where?
   A: usually after calling SUT
   (System Under Test)

Counter Example

```
public class Counter {
    int count;

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

Counter Test

```
// 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());
  }
}
```

State or Behaviour?

Was Counter Test verifying state or behaviour?

State vs Behaviour

State: e.g. object field values.
Call accessor methods to verify.

Behaviour: which calls SUT makes.
Insert observation points,
monitor interactions.

Flight example

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

Flight example: state verification

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

What Is State Verification?

1. Exercise SUT.
2. Verify state & check return values.

Inspect only outputs;
only call methods from SUT.

Do not instrument SUT.

Do not check interactions.

Implementing State Verification

Two options:

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

Flight Example: discussing state verification

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.

Flight example: procedural behaviour verification

```
// Meszaros, p. 472
// procedural behaviour verification
public void testRemoveFlightLogging_PBV() {
 // fixture setup:
 FlightDto expectedFlightDto=createRegisteredFlight();
 FlightMgmtFacade=new FlightMgmtFacadeImpl();
 // 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());
}
```

Alternative: Expected Behaviour Specification

Use a mock object framework (e.g. JMock)
to define expected behaviour.

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

Kinds of Assertions

Three built-in choices:

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

note: `assertTrue` can give
hard-to-diagnose error messages
(must try harder when using).

Using Assertions

Assertions are good:

- to check all things that should be true

  (more = better)
- to serve 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.
- to allow failure diagnosis

  (include assertion messages!)

Not Using Assertions

Can also do external result verification:

Write output to files, diff (or custom diff) expected and actual output.

Twist: expected result then not visible when looking at test.

(What's a good workaround?)

Verifying Behaviour

Observe actions (calls) of the SUT.

- procedural behaviour verification; or, (challenge: recording & verifying behaviour)
- via expected behaviour specification. (also captures outbound calls of SUT)

So far. . .

Seen the basics of result verification.

Next: how to improve your tests!

Reducing Test Code Duplication

Usual cause: copy-pasta.

Mitigating duplication in result verification:

- Expected Objects
- Custom Assertions
- Verification Methods

Duplication-Prone Test Method

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:

```
// 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);
}
```

Need:

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

Potential Issues

Perhaps we:

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

Solutions:

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

Custom Assertions Example

```
// 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());
}
```

Pick a good, declarative name.
Obtain by refactoring,
using usual techniques.

Benefits of Custom Assertions

- Hide irrelevant detail.
- Label actions with a good name.
- Are themselves testable.

Variant: Outcome-describing Verification Method

```
// 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);
}
```

Differences: a verification method

- also interacts with SUT;
- may have arbitrary parameters.

Going Further: Parameterized, Data-Driven Tests

While we're at it,
we can have entire tests
that differ only in input data.

Concrete tests invoke parametrized tests.

Avoiding Logic in Tests

Problem: tests are untestable.

Including ifs and loops in tests = danger!

```
// 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

Practical techniques for writing tests.

Today's focus: result verification.

- state verification
- behaviour verification

Also, techniques for improving your tests.

- reducing duplication
- simplifying tests