Testing for Fun and Profit

Camille Fournier

Outline

- Unit testing fundamental principles
- TDD
- Testing DB code
- Mocking
- The Build

The Platonic Ideal of Unit Testing

- Each test tests one “unit” of code
- A “unit” is the smallest testable part of a system
  - One path through one method, ideally
  - One path through a couple of methods, ok
  - Multiple paths through a single method... eh...
  - Multiple paths through multiple methods, yuck
  - Multiple paths through methods touching random external systems, not unit testing

But... I want to test MORE

- Fine. You can write integration tests.
  - Or functional tests.
  - Or performance tests.
  - Or regression tests.
- These are all wonderful, good to have, necessary to a complete testing suite...
  - But almost always slow, and often complex
- Unit tests are fast, small, and simple
  - Please don't make me tackle another 4 hour build

Basics of Unit testing

- For the method under test:
  - What inputs, if any?
  - What is the expected outcome given these inputs?
  - Assert that the outcome is as expected
- You need to have an assertion or it is not a unit test!
- This should include not just successful cases, but failures as well
  - Verify handling of bad data
  - Verify exceptions are thrown as expected

Sample Method, and Test

```
protected static boolean isAfter3pm(Calendar cal) {
	return (cal.get(Calendar.HOUR_OF_DAY) >=
	SHIPPING_CUTOFF_HOUR_24);

}
@Test
public void testIsAfter3pm() {
	Calendar day = Calendar.getInstance();
	day.set(Calendar.HOUR_OF_DAY, 16);
	Assert.assertTrue("Should be after 3", ReservationCoreApi.isAfter3pm(day)
	);
	day.set(Calendar.HOUR_OF_DAY, 3);
	Assert.assertFalse("Should be before 3", ReservationCoreApi.isAfter3pm
	(day));
	day.set(Calendar.HOUR_OF_DAY, 15);
	Assert.assertTrue("Should be after 3", ReservationCoreApi.isAfter3pm(day)
	);
}
```

Basics of JUnit (4.X)

- @Test – Marks this as a test to run
  - This takes options, for example:
    @Test(expected = MyException.class) (more on this...)
    @Test(timeout=1000) millisecond timeout for the test
- Assert: Assert the behavior you want
  - Lots of options, try to use the one that makes sense
  - assertTrue, assertNotNull, assertEquals...
  - Try to include a message about what you are testing
- @Ignore – don't run this test. Use this instead of commenting out your tests!

More JUnit

- @Before, @After – run this method before each test, run this method after each test
  - Useful if you have a common object or set of objects you want to work with
- @BeforeClass, @AfterClass – Static methods called before the entire test class is executed
  - Useful if you're writing tests that are more like integration tests

A note on exceptions

- If your test shouldn't throw an exception (ie, an exception indicates a failure) but there is a checked exception possible, the test method should throw Exception
  - Don't surround your code in try/catch blocks and then fail the test in the catch block!
  - JUnit will nicely fail the test for you if an unexpected exception is thrown, and you have much more information about the cause than you do from an Assert.fail()
- If you are testing the case of an exception, you may want to catch the exception and assert it contains expected messages or cause
  - You can use the (expected=ExceptionType.class) annotation, but if multiple places in the test could throw that exception you may want to be more precise

Cardinal Rules of Tests

- Be a good citizen: Leave the system state as you found it
  - IE, don't write things to the db without cleaning up!
- Every test should be able to run completely on its own
- Ordering of tests should not matter for success
- Tests are your best documentation. Write them to be readable by others
  - Copy-paste programming is just as bad when writing tests
- Tests should behave predictably
- Pay attention to the test results
  - If the test fails and you don't care, why run it?
- Test your own code, not the libraries you are calling

When to Write a Test

- Any time you fix a bug
  - Write a test FIRST that fails, then fix the bug
  - Do not check in a fix without a test
- Any time you add new functionality
- When you've found untested code that you don't understand
  - Great way to learn the code, and make sure it works
- Before you start on an aggressive new feature

Test-Driven Development

- Write a test, see it fail, then write code to make it green
- Forces you to think about the valid and invalid behavior up front
- Results in more testable code
  - Keeps you from baking in assumptions that are hard to fake for testing

Database Testing

- Go read the info on dbunit.org
  - Even if you don't use dbunit, the authors cover a bunch of the whys, hows, and best practices for database unit testing
- Think about grouping your data into fixtures
  - Related pieces of data needed by a test class or set of classes
- Data quality is important
  - If you need to test how your code handles various data from the db, it needs to be in your data set
- Try to use an in-memory database and mocks, if possible
  - DB interaction is damn slow

Mocking

- Enables unit testing even when your code wants to call external components
- Encourages development against interfaces
  - If you pass around interfaces instead of classes, you can easily write your own mocks by implementing the interface
- Mocking libraries (mockito!) are awesome
  - Unless all your code is static. Then they suck.

Example: The Power of Mocking

```
public class PlayJUnitRunnerTest {

    @Test
    public void testFilter() throws Exception {
        PlayJUnitRunner runner = mock(PlayJUnitRunner.class);
        runner.jUnit4 = new JUnit4(PlayJUnitRunnerTest.class);
        doCallRealMethod().when(runner).filter((Filter) any());

        runner.filter(new Filter() {

            @Override
            public boolean shouldRun(Description arg0) {
                return arg0.getMethodName().indexOf("testFilter") > -1;
            }

            @Override
            public String describe() {
                return "";
            }
        });

        when(runner.testCount()).thenCallRealMethod();
        when(runner.getDescription()).thenCallRealMethod();
        assertEquals(1, runner.testCount());
    }

}
```

Something Cool: Testing Logs

```
Layout layout = Logger.getRootLogger().getAppender("CONSOLE")
.getLayout();
ByteArrayOutputStream os = new ByteArrayOutputStream();
WriterAppender appender = new WriterAppender(layout, os);
appender.setImmediateFlush(true);
appender.setThreshold(Level.INFO);
Logger zlogger = Logger.getLogger("org.apache.zookeeper");
zlogger.addAppender(appender);
try { …
} finally {
zlogger.removeAppender(appender);
}

os.close();
LineNumberReader r = new LineNumberReader(new StringReader(os
.toString()));
String line;
Pattern p = Pattern.compile(".*Majority server found.*");
boolean found = false;
while ((line = r.readLine()) != null) {
if (p.matcher(line).matches()) {
found = true;
break;
}
}
Assert.assertTrue(
"Majority server wasn't found while connected to r/o server",
found);
}
```

The Build

- Tests are great but if your team is > 1, they need to be run regularly
  - When the code is changed
  - On a timer
- Running the suite needs to be scripted
  - For Java, ant or maven are good options
  - Play does this for us
- Poke around Jenkins. If you like Apache, poke around the Jenkins for the Apache projects (Esp Hadoop), they do some sick stuff
  - https://builds.apache.org/
- You must respect the build. Tests are worthless if you don't listen to them. When they fail, fix them. Don't leave them broken.

Build Bonuses

- We have a record of history, logs, timings
  - Can be useful to detect problem checkins
- We can use the build to run a bunch of stuff that might be too slow to run in our local env such as:
  - Code Coverage
    - How much of my code base is actually touched by my tests?
    - 100% might be impossible, but at least try to keep it over 80
    - Can show you areas to concentrate on covering better
  - FindBugs and other static analysis
  - Document generation
- The build is the system of record, and owns the publishing of artifacts