Increasing test coverage to support the build pipeline

Introduction

Having completed the Makers Academy 12 week bootcamp- the intensive training portion of the apprenticeship- my employer, a tech consultancy, has placed me with a public sector client. This offers me a hands on education in working within a client's operational requirements. This so far includes working from their office five days a week, completing all documentation required for onboarding. Concomitantly I have gained access to: Confluence an internal collaborative wiki for all project information including business requirements, meeting notes and architecture design decisions and diagrams, also Jira, a tracking tool for issues (which can be stories, bugs, research spikes, etc.), and Slack for internal communication.
I am now beginning work on an already established codebase for the first time, rather than building smaller personal projects from scratch. This codebase implements a micro-service, built with the Spring framework and Java 8, that will allow users to create and update the details of organisations and associated workers that interact with our client. To start understanding the code, our tech lead suggested that adding test coverage to the service would be a good way to begin.

The Development Process

Jenkins is an automation tool written in Java used to build and test software projects continuously, making it easier for developers to integrate changes to the project. This supports the development practice Continuous Integration, in which developers working as a team check code into a shared repository frequently and on each occasion the changes are verified by an automated build, such as Jenkins which is preferred by our client. Depending on the needs of the project, Jenkins can be used to integrate a variety of processes, including build, document, test, package, stage, deploy, static analysis, etc. (See figure 1).
Understanding this and working to support the build pipeline was my first task on joining the project, focussing initially on the build and test stages. This enabled me to get both a fairly high level view of the development lifecycle and a close up view of some of the discrete tasks that support this process.

Figure 1 – An example Jenkins build pipeline.

As well as adhering to the client approach to continuous integration, this work also offered me the opportunity interpret and follow company defined coding standards and testing frameworks. For the purposes of this particular project, test coverage needs to meet a certain threshold. PITest coverage[1] has to be above 90%, this is a requirement set by our team for pushing and merging code, and overall line coverage must be above 80% in order for the build pipeline to pass the tests stage. Should this fall under the minimum threshold, the pipeline would fail at that point and would not continue on to the other stages. In both cases, coverage is increased and maintained through unit tests using JUnit and Mockito[2].

I'd briefly used JUnit before, but Mockito was entirely new. I could infer how it worked to an extent from looking at the pre-existing examples of its usage, but to better understand its purpose pressing Ctrl and clicking the '@Mock' annotation yielded a helpful message (see figure 2).

[1] PIT is a mutation testing framework for Java. It randomly modifies (or mutates) parts of the application code and runs unit tests to check that they fail. This ensures that unit tests can detect problems in application code and do not just execute the code.
[2] A testing framework for Java that allows the creation of mock objects.

```
/**
 * Mark a field as a mock.
 *
 * <ul>
 * <li>Allows shorthand mock creation.</li>
 * <li>Minimizes repetitive mock creation code.</li>
 * <li>Makes the test class more readable.</li>
 * <li>Makes the verification error easier to read because the <b>field name</b> is used to identify the mock.</li>
 * </ul>
 *
```

Figure 2 – Mockito documentation

This clarifies the value of the annotation, but to learn why it was worth creating mocks in the first instance required some further research online. I then learned that because a unit test should test functionality in isolation, a mock object could therefore be configured to perform a certain behaviour and used to replace real dependencies.

Figure 3 below shows an example of a unit testing I wrote, with my newfound knowledge of Mockito, to help maintain coverage. On lines 13 and 16 the '@Mock' annotation works in lieu of mock creation code, marking the ensuing fields as mocks of the validators, and makes the test class more readable.

```
10  @RunWith(MockitoJUnitRunner.class)
11  public class UpdateOrganisationRequestValidatorTest {
12
13      @Mock
14      UpdateOrganisationValidator updateOrganisationValidatorMock1;
15
16      @Mock
17      UpdateOrganisationValidator updateOrganisationValidatorMock2;
18
19      UpdateOrganisationRequestValidator updateOrganisationRequestValidator;
20
21      Organisation dummyOrganisation = new Organisation(
22              "dummyName",
23              OrganisationStatus.ACTIVE,
24              "sraId",
25              "12345678",
26              Boolean.FALSE,
27              "dummySite.com");
28
29      @Before
30      public void setUp() throws Exception {
31          updateOrganisationRequestValidator =
32                  new UpdateOrganisationRequestValidator(asList(updateOrganisationValidatorMock1, updateOrganisationValidatorMock2));
33      }
34
35      @Test
36      public void validateStatusTest() {
37          updateOrganisationRequestValidator.validateStatus(dummyOrganisation, dummyOrganisation.getStatus(), dummyOrganisation.getOrganisationIdentifier());
38
39          verify(updateOrganisationValidatorMock1, times(1))
40                  .validate(dummyOrganisation, dummyOrganisation.getStatus(), dummyOrganisation.getOrganisationIdentifier());
41          verify(updateOrganisationValidatorMock2, times(1))
42                  .validate(dummyOrganisation, dummyOrganisation.getStatus(), dummyOrganisation.getOrganisationIdentifier());
43      }
44  }
```

Figure 3 - My unit test class.

I've then given the mocked validators as parameters to the updateOrganisationRequestValidator object, in order to focus on verifying that the method under test, 'validateStatus', behaves as expected.

Intellij, our IDE of choice, provided some functionality that I found useful for this task. When running the unit tests, you could right click on the run symbol and choose run 'with Coverage' instead (see figure 4).

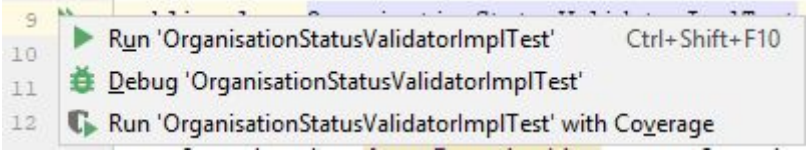

Figure 4 – Options for running tests.

This would run the tests and offer some additional information. First, it would show the percentage of test coverage that had been achieved for each of the main classes (figure 5).

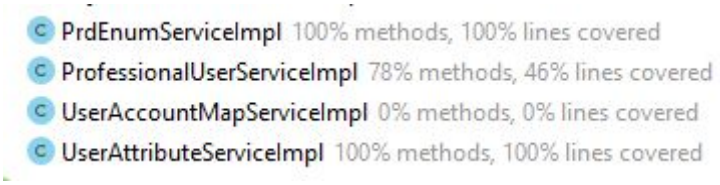

Figure 5 – Test coverage by percentage.

As you can see it categorised this information further by methods and lines covered. This was incredibly useful for helping to keep track of how much I'd elevated the test coverage by as I worked. It also helped me see if I'd failed to cover a method adequately though I'd written a test. In this way, it enabled me ensure the quality of my testing. Finally, and perhaps most usefully, running the tests with coverage would highlight the individual line numbers in the classes tested (see figure 6).

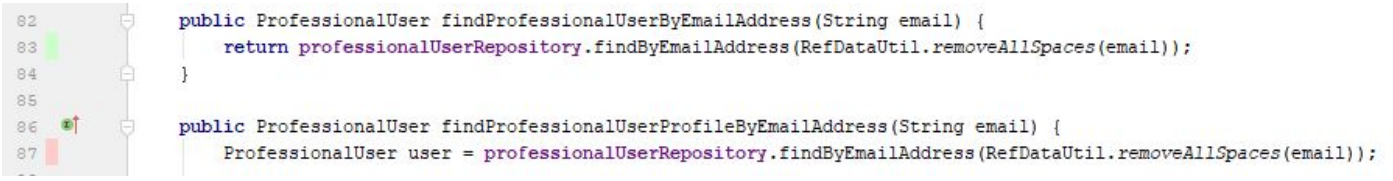

Figure 6 – Highlighting by the line numbers showing test coverage.

In figure 6 line 83 is adequately covered by a test in the corresponding unit test class, but line 87 isn't. This made it very easy to zero in on the areas where testing was weak, so that I could fill the gaps.

Reflective Statement

The main challenge I encountered while doing this work was building an understanding of the source code- which was already pretty established and growing increasingly labyrinthine- in order to know what everything was doing and be able to begin thinking about how to test specific functionality. To overcome this there was nothing for it but to spend time studying the source code, pressing Ctrl and clicking on unfamiliar objects to bring up their classes and examine their contents. Ultimately, anything that seemed too opaque to understand, I would ask my team mates about.

Another challenge to an extent was writing the tests with Mockito, a framework I had never used before. This I overcame by searching online for tutorials, and bookmarking the websites that seemed to have the most helpful and/or most comprehensive range of tutorials for future reference. Also as my tech lead said, "copy paste is your friend", meaning it's always worth looking at how it had already been applied in the context of this project elsewhere in the source code.

Some of these problem solving strategies, such as searching for tutorials and examining the source class of an object, I had already adopted from my time at boot camp, but there's an art to knowing what to look for in both cases, so the more I do it the more my process is improved. Further to that, the practice of searching within an existing codebase for examples of a framework already being applied is new to me. This is notably the most useful new strategy that I have adopted as a result of this project.

Owing to this work I started to understand the service the team is building a lot better, and there were two main business benefits. First, for the client, it enabled us to meet the previously mentioned requirement of maintaining test coverage above a certain threshold, ensuring build pipeline success. Secondly, for the team, it helped clear fast accruing code debt. Though it was acknowledged that developers should ordinarily assume responsibility for their test cases, the team had come to this client project shortly before myself to find a considerable commitment had been made, and an unforgiving deadline to deliver by. Though this was not ideal, it allowed me to make a meaningful contribution straight away to support the team and the client's goals.