**Joshua Massey**

Southern New Hampshire University

06/20/21

Project 2 Reflection

In order to create a product with the most value for the client, one must first fully understand their needs. After taking time to break down the requirements of the backend service requested for this mobile application into smaller user “actions” with the system, I began creating test cases based on those requirements.

One example of this is is the requirement for updating a field in the Task object. The request was that only the name and description of the task could be updated, although there is actually a third field as well for the object’s ID. When the update function is called, a switch statement is used to change the specified field, given that the object exists and that the field’s value is valid based on a set of defined parameters. In testing this feature, the test case was fed both valid and invalid requests, including attempting to change static fields like ID. The end result was that our test case proved the rigidity of the core logic as none of the edge case or failing scenarios created unintended outcomes.

Applying the same strategy as the one described above to the rest of the code base, I was able to ensure that the source code was built to cover all the business requirements set by the client. Further boosting the notion of a robust testing strategy is the fact that there is 100% code coverage across the entire application. What this signifies is that the unit tests have been written to cover every passing *and* failing scenario which applies to the service.

To ensure that the code was technically sound, many of the tests covered the same logic multiple times. For example, in the contact service, one of the requirements is that the first and last name can neither be null nor be more than 10 characters long. This functionality is used by both the functions for adding a new Contact and editing an existing object. Since the application uses an in-memory database to store the data, each test case uses a new instance of the of the ContactService, as shows by lines 15-18 of the ContactServiceTest. You can see some of the helper methods for performing data validation be also used in the tests for the Contact object as well. In ContactTest, beginning on line 32, we can see that the same kind of testing is performed on the setter methods, which also validate the data passed in to each method.

Although certain parts of the codebase were tested multiple times, I was able to leverage the capabilities of JUnit 5 in order to ensure that they were written succinctly. Certain functions such as *AssertAll()* allowed for passing in multiple assertion function calls for a single case. Using this, as well as leaving comments for each of the failing scenarios in each test, enabled brevity across all test cases as many fewer lines were needed to test the same functionality. The update method testing mentioned in the latter part of he prior paragraph is also a prime example of this test case construction being employed.

With this set of services being self-contained, as in it does not reach out to any external sources, and not fully integrated into the mobile application it is designed for, the kinds of testing which would be meaningful are few. As demonstrated, unit and acceptance testing were the sole drivers of the testing strategy within the scope of this project. Each test in the suite has its own set of stubbed data, and its functionality is tested based on the business requirements set forth. Both the techniques used, while very similar, ultimately have two different goals. Unit testing is meant to ensure that we achieve as close to 100% code coverage as possible. Acceptance testing, on the other hand, would ensure that the methods we call on actually achieve intended final result.

Although the code is meant to act as a service for a mobile application, none of the services, that being the ContactService, TaskService, and AppointmentService, interact with each other. Once they are connected to a UI and these components begin interacting with each other, it would be in the best interest of maintaining our robust testing strategy to do some integration testing. While also similar to acceptance - and by proxy unit - testing, integration testing’s goals is to test the functionality of the system as a whole. This would encompass creating a set of tasks for a “user” interacting with the application to complete, and seeing that the data both persists, maintains its integrity, and that the application is sufficiently protected by the proper handling of edge case scenarios, including critical errors.

As the requirements only make use of an in-place database in the form of a HashMap and it is not a RESTful service, there is no place in this application for any kind of performance testing as the operations stay within the context of the user-facing client. As mentioned before, the fact that we don’t reach out to any external sources or use any third party dependencies, security testing in this context would fail to bring much more value in terms of the overall testing strategy. However, if we were to introduce external libraries to the codebase, I would reverse this recommendation and suggest performing security testing in the form of vulnerability scans. OWASP provides libraries for both Gradle and Maven projects which scans the build file and checks for known vulnerabilities in the version of dependencies being utilized.

One of the pitfalls I took care to avoid is that of failing to test scenarios which I, as the developer, had not thought of. This can also be described as writing tests for the code, instead of coding for the tests - the latter of which is a core tenet of test-driven development. Naturally people will lean on confirmation bias to assure themselves that, if a test case fails, the scenario wasn’t set up correctly as opposed having an inherent flaw in the core logic. My exposure to proper testing techniques in a professional setting enabled me to circumvent this common trap. Before writing even the first the branching statement, I begin by mapping out all the possible scenarios; next is developing tests based on them. Proceeding these steps is the actual building of the core logic and, ultimately, running the cases for acceptance testing.

The ability to resist these kinds of outcomes takes hours of practice and discipline. Although the process may be tedious, the end result is the development of code which is less prone to errors. I have personally seen the destructive power fully untested code can wreak on a production application when it is shipped with uncaught defects.