**Alignment to Requirements**

The testing approach used was closely aligned to the requirements, as almost every unit test is based on the listed requirements. Common requirements were non-null fields and a length restriction, which can be found in each model test class. For the contact model, an exact length was required for the phone number, which is also reflected in the test class. The services also had common requirements of adding and removing model instances, as well as being able to update the model fields, which can be found in the service test classes. Although the test for if a service contains a model instance may not look as if it is aligned to the requirements at first, it is aligned to the ability to add a model instance.

**Effective Tests**

All the models and the appointment service have 100% coverage, while the contact and task services have 95.9% and 97.1% coverage, respectively. For most projects, numbers above 80% would be satisfactory (Pittet, n.d.) if the required functionality was also present. For this project, the requirements have been fulfilled, so based on code coverage it could be said that the tests are very effective.

**Technically Sound Code**

To ensure the code was technically sound, I utilized the @BeforeEach annotation to set up the test environment (lines 15-20 in the task/contact service tests, lines 19-26 in the appointment service test) in each of the service test classes. I also made sure that most test functions used only one assertion (lines 36-64 of the contact service test) so that it was easy to track down the source of a failure.

**Efficient Code**

Although the tests were not designed with efficient code in mind, there are examples that can be found in the models and services themselves. Using the task model as an example, the ‘final’ keyword was used on line 6 to fulfill the requirement that the ID could not be updated, and the setter functions were used on lines 16 and 17 to re-use the logic that checks for invalid lengths and null values. Additionally, the services utilized a hash-table as the backing data structure, which can be seen in the contact service on line 12. Because the models already have a unique ID assigned to them, a hash-table is an excellent choice for the speed of insertion, retrieval, and removal.

**Techniques Employed**

Many of the techniques used were specification-based (or black-box), as they did not test how the program achieved a result, only that it achieved the expected result based on the customer specifications. For example, the task service test does not care how the task is stored after it is added, only that it is stored and can be retrieved or deleted. One specific technique used was equivalence partitioning, as seen in the appointment test. The tests for length use strings that are only slightly too long and assume that the program will handle longer strings in the same manner. This takes advantage of the fact that you can split the valid inputs and invalid inputs into partitions (in this case, the partitions are null, strings longer than 50 characters, and strings that are 50 characters or less), and test for only one input within each partition to get an idea of how the program handles every input within each partition.

**Other Techniques**

One technique that wasn’t used was use case testing. This type of testing focuses on a goal that a user might have when interacting with a program and tests how well the program allows the user to fulfill that goal, usually with a set of steps that the user takes. Another technique that wasn’t used is decision table testing. This technique involves creating a table that holds a set of conditions and a set of actions for a system function defined by the specifications and determining which combination of true/false conditions can result in each action happening. Each combination is called a “business rule” and can be used to test whether the system handles the rule correctly.

**Uses and Implications of Techniques**

The main use of equivalence partitioning is to save time, as it drastically reduces the number of inputs that need to be tested. It does produce some uncertainty however, as there could be edge cases within those partitions that do not produce the expected result, which would lead to difficult to find errors. Use case testing is excellent for seeing how effective the program is at fulfilling customer needs, as well as testing how well components of the system have been integrated. This ensures that components of a system have been designed with the end goal of a whole system based on customer needs, instead of just functioning to individual specifications. Decision table testing is helpful to keep track of exactly what conditions need to be active for a certain action to take place, so that a systematic approach to testing that action can be performed. Since it identifies which conditions are irrelevant to an action, it can also increase efficiency of testing.

**Mindset – Caution**

One example of how I employed caution is by using the same values for each variable within the model classes, aside from the one being tested at the time. Using the contact test as an example, the ID is always “testID”, except in the tests checking for an ID too long or a null ID. Although it is unlikely that different values will cause a failure if they fulfill the requirements for each variable, this method provides another layer of assurance that the fault can be isolated based on the test that failed.

**Mindset – Bias**

I attempted to limit bias by evaluating the code based on whether it fulfilled the software requirements, when possible. For more complex projects, I can easily see bias being a large concern when testing my own code, especially when using types of tests besides unit tests. One example in this project that could end up being a problem in other projects is the initial test in each model. In the contact model test (lines 11-19), I used several assertions in one function because I couldn’t see a way that they could fail with the way I created the class.

**Mindset – Discipline**

The example from before regarding the model tests is a good example here, as I cut corners on those tests for short term benefits. Although it had no impact on the completion of this project, cutting corners can easily add up to a codebase that is not maintainable, especially with projects that have a long life. I think keeping a couple things in mind while writing and testing code would help greatly with avoiding technical debt. The first thing is how difficult it would be to understand and modify the code or tests if I returned to them after a couple years of ignoring them. The second is if the structures or architecture I’m using are the best for what I’m trying to accomplish and the restrictions I have at the time.

**References**

Pittet, S. (n.d.) *An introduction to code coverage.* Atlassian. Retrieved October 15, 2021, from https://www.atlassian.com/continuous-delivery/software-testing/code-coverage