# Summary & Reflections Report

# Testing Junit Strategy Defense & Reflection

**1.1 Contact Feature Tests:**

The testing approach for each feature evolved each time and became more robust while remaining within the software requirements. For the contact feature, which created contact objects in the contact class, and the contact service class which managed those contacts, we first tested the use of our constructor. This entailed hard coding in some values that met the software requirements. The test then validated that the information given for the different variables was in fact those values. We used assertion methods like assert true to verify this. We used an assertion for each field, which when called or added to various data structures, ensured that the values we put in, remained unchanged. One useful application of this test would be, in a productionized environment, would be to ensure data integrity.

In a similar way we updated our contact with a new name and then validated that the update was made. Finally, we deleted the contacts and then verified we pull a null value for the key we created. One thing we could’ve done to make the unit testing more robust would have been to include the length tests in the test rather than in the contact class. We failed to do this but the methods were created in lines 80 on inside the contact class. This would’ve been as simple as to call them. If these validation methods were included we could have run a unit test checked for length and null status simultaneously using if/ or statements.

**1.2 Task Service Feature Tests**

Our task service testing approach was built on validation of the features core assumptions around the existence of a value which, when combined with other data create the task object. The first unit test on adding tasks used assert true combined with “.equals()” to ensure that the object that was created and added was indeed that value. The next method was the delete method. Here we used the same approach but checked for all null values. To test update we sent in an updated value as an argument to our constructor, and then called “updateTask” from the task service class to update the values.

This feature was coded well and made use of hash mapping so that data could be efficiently searched. While it was more robust than the testing employed in the contact service the tests to check lengths was missing. We did check for null values. Taking this into consideration, we say that only ½ of the requirements were tested. This could be more robustly coded by adding in the assert methods used in the appointment test which allowed us to use inequalities in our assertions because the static length variables were defined but not used.

**1.3 Appointment Service Tests**

The appointment feature’s unit tests checked for variable lengths along with null values. as well as “assert” to and runtime exceptions to determine whether or not the test passed or failed. This use of pointed error messages on lines 104 and / or 123 help us understand what exactly happened rather than just a “fail.” This makes these unit tests a little more user friendly, in the case that I as the developer didn’t test the code.

In the application the variable lengths had maximum size requirements so we checked to make sure that the size of the inputs was not over this length using assert throws which will tell us whether or not the argument was invalid per the requirements. We proved this out by hardcoding invalid arguments so as to ensure a failed test. Using the same type of method we checked for null values.

We tested our methods for adding, deleting, and updating appointments in the appointment service test class. What we did here was walkthrough and test the methods by calling them and providing arguments. We used validation methods we coded in to the contact service class to validate the contact, create a key, and then add it to the hash list we created.

One key difference between the appointment feature’s requirement list and the other two was the use of dates. We had to include a test to see if the date was in the past. We solved this on lines 125 to 131 by creating an instance of LocalDate and calling the “.now” method.

# Summary

In summary, the above mentioned unit tests covered most of the requirements for the application’s features. By checking for length, using exception throws for null values, and then calling our methods within the unit tests, we can be confident that inputs provided would be validated, keys would be created, and we can add, update and delete a contact, appointment, and or task to the various hash lists we created. This was done in various lines within the program application and became the chief way we would store data. It can be found anytime a new object is created. One thing that was not done was setting up necessary annotations such as @beforeall, @before each. This would have helped us run these tests more consistently each time the application was launched.

# New Mindsets on Software Testing

Ultimately, before coming into this class I always knew they tested things, but I believed that they must code a lot more than I thought before testing. I also learned a lot more about unit testing. I had a slight understanding of it just being in the industry, but never to the extent of knowing there was already a devoted library to it! I tried to eliminate bias in my testing by ensuring I tested each piece independently, and then together. By dividing the unit tests into discrete parts I am able to assess each piece one by one and determine a pass / fail. This is opposed to testing one or two of the fields and determining a pass. As the developer, since you code and write your own tests, you may leave the more buggier parts to less strenuous tests so as not to have to put the appropriate amount of work into it to make it as least error prone as possible. Being disciplined takes time. The less time you give yourself the messier and less complete the application will be. It’s important not to cut corners, and the one way you can ensure that is to put the time into it and socializing it with others. I think it is important in cases like this to get feedback from others at work or in the software community. This can reveal issues without even having to run tests and can also be a great way to eliminate bias. This evolved my understanding of software development in such an important way because this saves you time as a developer and reduces technical debt and also eliminating re-work. It is an important step and one I will always look back on as I continue my learning journey to be a software developer.