Brad Jackson
CS320
February 20, 2022

Summary and Reflections Report

My approach to unit testing was aligned to the software requirements for each of the three features of project one. I created unit tests for each service per the customer requirements. I accomplished this by following the requirements laid out in the rubric for each of the services beginning with the contact service.

The contact service uses in-memory data structures to support storing contacts. The contact service contains a contact object along with the contact service. Requirements for the contact class began with a unique ID string which could not be longer than 10 characters. This attribute of that contact class was also not nullable nor modifiable. I created a unit test which generated a unique ID with a 10 character ID and a unit test which attempted to create two contacts with the same ID. The latter was meant to fail. I also created a unit test in which the ID was null. This was also meant to fail. In this way we had full coverage on that attribute. The remainder of the fields, firstName, lastName, phone, and address, were only constrained by character length and nullability. I created unit tests for each which asserted that null values would fail. In addition to this, I created unit tests for the character constraints. The requirements for the contact service aligned with the unit tests I created for the contact class. The tests were submitted to the contact service to validate the contact class.

The task service uses in-memory data structures to support storing tasks. The task service contains a task object along with the task service. The requirements for the task class related to nullability, writability, and character length. The task service had the requirements of being able to add, delete, and update tasks based on certain criteria. I created unit tests, similar to those for the contact service, to validate inputs to the task service and, in turn, the task class. This service had fewer requirements than the contact service and was easier to achieve full coverage on.

The appointment service uses in-memory data structures to support storing appointments. The appointment service contains an appointment object along with the appointment service. I created unit

tests for the appointment service which tested the typing, character length, nullability, and writability of the appointment class. A unit test unique to this class was one to check the date. The appointment date was required to be a java.util.date classand could not occur in the past. Unit tests were also written to test the ability to add appointments via a unique appointment ID, one which did not exist, and to delete appointments in this way.

I believe that the overall quality of my JUnit tests was high. By breaking down each requirement into several unit tests I was able to achieve a high level of coverage and handle unexpected exceptions in a controlled manner. I ensured that my code was technically sound by running the tests locally using Maven. Below is an example of one of my tests for the contact service,

```
@Test
 public void testUpdateContact() throws Exception {
   String id;
   svc = new ContactService();
   svc.addContact("Testy", "Guy", "1233211234", "Nowhere");
   id = svc.getByFirstName("Testy");
   svc.updateContact(id, "Tested", "Guy", "1233211234", "Nowhere");
 }
```

This tested the ability to update a contact which already existed in the instance of the contact service. By breaking down the requirements into multiple, smaller, tests, I was able to efficiently meet the requirements for each service.

I used static and dynamic testing techniques for this project. While writing the code for the models I validated that I was following requirements. I did this by reading the code once I was complete with each model. The same applied for the services because each service required different CRUD

methods and validation patterns. After fixing any visual errors, I ran the projects locally and addressed errors as needed, dynamically. After writing unit tests according to the requirements for each service I also did some exploratory testing, guessing other inputs which users may give in order to produce errors. I counteracted this behavior by handling those errors and creating a bucket for other errors. This could be improved by sending the users a generic message and logging specific inputs for the admin. This way they could go back to the logs and create new tests.

I adopted the same mindset for this project as I do when I am working on professional projects. It is important to consider all aspects of the software requirements as defined by the customer. This is done in order to meet and exceed their expectations. It is also important to realize where to end specific unit tests and employ error handling. When reviewing my own code I am usually unbiased. I am not afraid of self-improvement and realize that I always have more room to grow. The only downside of this project is that I could not have peers validate the services as well. It is always a great idea to get more opinions on a project before it proceeds to a production environment.