Project Two: Summary and Reflections

Matthew Bridegroom

Southern New Hampshire University

CS 320: Software Test Automation QA

6/29/2025

**Summary**

The implementation of unit testing in each of the different features had slight variations to them. Taking a look at the Contact Service first, the unit testing approach is shown to be focused on helping to validate the actual core functions of the service such as adding, deleting, or updating the different contacts. This was done while also making sure that all the constraints were met, such as the 10 Digit phone numbers or the requirement for a unique ID. One example of one of the key test cases was testAddContactSuccess, this helped to verify that the contact was created successfully. Additionally, the different tests in ContactTest.java helped to validate the constructors and setters using different null checks and enforcing the format for the phone number.

The second service, being Task Service, also had the same approach as validating the different core functions of the service, such as adding, deleting, and updating the different tasks. The different testing in Task Service focused on ensuring that constraints with this service were enforced. These include a unique task ID, a maximum of 20 Characters for the name and a maximum of 50 Characters for the description. One important test case for the Task Service was the testAddTaskSuccess. This helped in verifying which of the tasks were added with a valid set of data. Like the Contact Service, TaskTest.java also helped here in testing the object’s constructor and setter methods in Task.

Finally, the third service, Appointment Service, also followed the same approach validating its core functionality in adding, deleting, and creating new appointments. With the Appointment Service, however, there were two main constraints which were tested for. There was the Unique Appointment ID, and that the date was set in the future. One test that was key to the service was the testValidAppointmentCreation. This test helped to verify that appointments were being created with valid input. AppointmentTest.java also helped in testing the constructor and setters for the Appointment Service, as well as the validation of the date and description.

The testing approach that I used in each of the different Services varied as they each had their own set of requirements. Throughout each of them I did my best to ensure that they aligned with every requirement of the software. Below are examples of each service and how they meet the requirements.

In Contact Service, there were three main requirements for the service. The first was the Unique ID, the second a 10-Digit Phone number.

assertThrows(IllegalArgumentException.class, () -> new Contact(null, "Matthew", "Bridegroom", "1234567890", "123 Main St"));

assertThrows(IllegalArgumentException.class, () -> c.setPhone("123"));

For Task Service, the requirements listed again were a Unique ID, as well as new constraints on the Name and the Description.

assertThrows(IllegalArgumentException.class, () -> service.addTask(new Task("task001", "Clean Dishes", "Do the dishes after dinner")));

Lastly, for the Appointment Service, there was again a requirement for a Unique ID for the appointments, a date set in the future and a limit of 50 Characters for the description.

assertThrows(IllegalArgumentException.class, () -> new Appointment("appt003", getPastDate(), "Check-up"));

I believe the JUnit Tests that I wrote were delivered in terms of quality. I believe this because I followed a comprehensive testing plan and aimed to cover both positive and negative testing scenarios for the different core functions in each of the services. For Contact Service, I tested the functions of adding, deleting and updating new contacts and followed that methodology in both Task and Appointment Services. I believe that the coverage of my test was greater than 90% as the different tests I incorporated covered all areas of the services. This can be seen with tests like testDeleteContactNotFound or in testInvalidSetters. Both of these helps to show the thoroughness of the tests incorporated.

The actual experience of writing the JUnit tests was a more detailed process as it required the key functionalities to be identified through analysis, testing, and implementation. When writing them I first started by covering the basics such as testAddContactSuccess then I followed that by testing different edge cases such as testAddContactDuplicateIDThrows. Doing this helps to ensure that the code is Technically sound. Another way of ensuring that the code was technically sound was by validating that all the different inputs and outputs all met the requirements and constraints that were set. One example of this was in ContactServiceTest.java where the test testUpdateContactFields helped to verify that the updates in the code were made and applied correctly. This can be seen below.

service.updateFirstName("abc123", "Alicia");

assertEquals("Alicia", c.getFirstName());

These tests showed that updateFirstName called upon the Contact setter and applied the update correctly. Efficiency within the code was also achieved by making use of the logic I had setup beforehand and reusing it later in the different services.

**Reflections**

There were three main types of software testing techniques that I used in the different Services, I used Unit Testing, Boundary Testing, and Exception Testing. Unit testing involves testing the individual classes such as Contact, or Task in Isolation. Boundary Testing involved testing edge cases and Exception testing was used in validating the way that errors were handled in the different Services.

Three forms of testing that I did not incorporate into any of the services were Integration Testing, Performance Testing, and Regressing Testing. Integration Testing can be used to test the interactions between the different components in a service such as Contact and Task Services. Performance Testing would test the load of the services on different resources such as memory. Regression testing would test to ensure that new features or changes in the services did not break the existing functions.

The different methods of testing that I employed all have their own practicial uses. Unit testing can help to reduce any early bugs in the system by making sure that the individual components function as they should. Boundary testing can be used practically to help ensure that any edge cases scenarios can be prevented, and exceptional testing helps in the case of user input by ensuring that any errors are handled reliably.

I believe that I employed a more robust level of caution. This is because I did my best to ensure that I was testing any possible scenario I could such as edge cases and any errors. One example of my more elevated level of caution was in ContactServiceTest.java where I tested to ensure that any duplicate IDs were rejected. This can be seen in my implementation of testAddContactDuplicateIdThrows. If I were testing my own code like down in this project, simply overlooking or having a greater level of confidence could allow me to overlook simple requirements such as null inputs in areas such as Contact.setPhone. One way I tried to avoid this and limit my own bias was by making sure I had some level of coverage over the entirety of the requirements. A future improvement could be the use of peer review.

I believe that it is important in software engineering to ensure that quality is top level and do corners are being cut in development as it helps to avoid any technical debt. This can be seen if one were to avoid or skip the testing of different edge case tests, which could allow tiny errors to pass along and become a greater problem down the line. An example of how I did my best to be committed to quality can be seen with my inclusion of testInvalidAppointmentDate. This allowed me to test correctly and not leave a possibility of an error down the line.