Reed Murray

CS – 320 Summary and Reflection

**SUMMARY**

When approaching testing for the contact, task, and appointment services, I tailored the unit tests to exactly align with the software requirements provided. For example, one of the requirements for all three object classes is that there be a unique ID which cannot be null. These are two separate requirements: the ID cannot be null, and there cannot be two ID with the same string assigned. I created a J Unit test called ‘**testApptIdIdNull()’** for the appointment class which creates an appointment object and assigns **null** to the variable for ID; this test aligns with the requirements because it uses **assertThrows()** to assert that a specific exception will be thrown when constructing an appointment object with a null ID:

*Assertions.assertThrows(IllegalArgumentException.class, () -> {*

*LocalDate appt\_date = LocalDate.of(2025, 8, 9);*

*Appointment appTest = new Appointment(****null,*** *"This is the summary", appt\_date);*

Each J Unit test I created was testing a method or constructor in the class and aligned with the software requirements. By keeping tests modular, each requirement can be analyzed individually. The JUnit test coverage for **Task.java** and **Contact.java** is 100%, meaning every line of code is run by a JUnit test. The constructor is tested, the accessors and mutators (setters and getters) are all tested as well. The three service classes have coverage ranging from 83-91%. This is due to a couple lines which are unused branches in the unit tests. For example, when deleting an appointment, the program checks if the appointment exists, because otherwise it cannot be deleted. The branch that is taken when the appointment requested to be deleted doesn’t exist simply returns the same appointment list given to the method as an argument. This ensures that we won’t be running into **out of array index** exceptions when trying to delete appointments that don’t exist. In the J Unit test, only the first return option is tested because that is the software requirement outlined for testing.

A close-up of a computer code

Description automatically generated

Keeping the tests simple and modular is what allowed me to write technically sound code. I used only 2-3 lines of code for most of the tests and stuck to testing one software requirement at a time. Each test would declare and instantiate an object which needed to be tested. I manually entered in the arguments for methods to ensure the arguments I wanted to pass were passed. For example, the contact class requires the phone number to be exactly 10 digits. One unit test I wrote for this was **‘testPhoneNumTooShort()’** which creates contact and tries to set a phone number that is too short. I assert that an exception will be thrown away when the argument is passed:

*Assertions.assertThrows(IllegalArgumentException.class, () -> {*

*Contact contact = new Contact("user123");*  ***// create contact object***

*contact.setPhoneNum("12345");* ***// Try to set phone number less than 10 digits***

The simplicity of a unit test like this meant I wrote several tests for one requirement sometimes. The combination of **testPhoneNumTooShort()** and **testPhoneNumTooLong()** make sure the phone number is exactly 10 digits. This is efficient because it allows the tester to immediately see which exceptions are working. If the exception were not caught for one or both tests, I would know how to look at the input validation code within the **Contact** class for defects.

**REFLECTIONS**

While writing JUnit tests I used several different testing techniques. Unit testing is the technique employed by writing the JUnit methods. Each ‘@Test’ within the **ContactTest** class tests the implementation of a method from the **Contact** class. By testing individual units, I employed unit testing. JUnit is a software tool used to automate the process of testing, which means its implementation also employs the technique ‘automated testing’.

Boundary value testing is a way to validate the input given to a method. It involves testing the boundaries of input values to identify defects. I used this technique to ensure that character limits were enforced by the program.

There are several types of testing which would only be applicable in a commercial or enterprise level project. Security testing detects software vulnerabilities to unauthorized access. One way to test security is to try to ‘hack’ into the software to see where the vulnerabilities exist; this is called penetration testing. Performance testing can determine the speed and scalability of software, which is important when working on a larger scale. I’ve used performance testing in the past for determining the efficiency of algorithms, however performance testing would be beyond the scale of this project, so I did not employ it.

While working on the first milestone I developed a user interface because I didn’t fully understand what the JUnit tests were going to accomplish. I have done most of my software testing using a user interface to test input validation and that methods are working properly. After learning more about JUnit tests and watching a video explicitly explaining how to utilize them, I realized the relationship between the class methods and the test methods was what facilitates automated testing.

For example, my input validation for **Contact ID** was within the constructor for creating a contact object. This meant within my JUnit test I had to assert that an exception would be thrown, then attempt to create an object with an invalid ID as a constructor argument. It was extremely important to know the structure of the **Contact** method which I was writing to get the correct assertion within the unit test. If I didn’t use the constructor in the assertion but received an exception for another reason, the test would be misleading.

When reviewing my own code, I have the mindset that I will find mistakes because I know I made them. I double check something even if I am confident in myself because testing isn’t about ego, it’s about getting the software to work. Bias would get in the way if the developers’ ego were influencing their decisions about testing their own code. While writing the **Appointment** class and test, I realized many inefficiencies in my **Contact** class code. Instead of thinking about how I spent time on that code, and I should find a way to fix it and keep it, I deleted it, specifically the user interface portion. By not allowing myself to be biased towards my previous scripts, I can make better decisions like deleting the user interface I developed. This turned out to be a positive decision because it improved my test coverage and made more sense because my role is a back-end developer in this scenario.

An important part of the mindset while developing software and tests is to not rush or cut corners. If you develop tests that don’t work properly, that is worse for the project than not having a test because the code would appear covered when it isn’t. For me, taking breaks is an important part of staying disciplined. If I am working on something without taking my eyes off it for a while, I feel more frantic and less focused. Being self-aware while working is one way I plan to avoid technical debt. For example, I ran into a bug while creating tests for the **Appointment** class. I kept getting a failed test even though I was confident while coding it. I took a break for a small time and came back to look at the code again. I found a simple error where I compared two strings without using the **compareTo** method. This led to the strings not being properly compared, which meant the exception I asserted would not get thrown during runtime.