Jacob Winters

CS-320

December 8, 2021

**Project Two: Summary and Reflections Report**

While developing my tests for the contact, task, and appointment services, I ensured that my approach was aligned to the software requirements by developing test patterns that verified a specific criteria was met. All three services utilize JUnit tests to determine unique attributes. The requirements for the contact class were to create an object with a required unique ID string that is no longer than 10 characters in length, cannot be null, and cannot be updated. In addition to this, the contact object has a required firstName and lastName string field that cannot be longer than 10 characters, nor can it be null. The contact object also has a required phone string field that must be exactly 10 digits, and cannot be null. The remaining requirement for the contact object was to include a required address field, no longer than 30 characters, and as usual, cannot be null.

In a moment, we will take a look at a portion of the contact service test object. The contact service is designed to loop through the program and check each string within the contact object for valid entries. The contact service itself had to meet a different set of requirements. The first of which was the ability to add contacts with a unique ID. The contact service must also be able to delete contacts per ID, and update contact fields per contact ID. The updatable contact fields must include the firstName, lastName, phone/number, and address fields. Let’s look at an example of the JUnit test within the contact service object, which verifies whether a contact can be added successfully.

@Test

public void testAdd()

{

ContactService cs = new ContactService();

Contact test1 = new Contact("1928374", "Kenny", "Schrub", "1111111111", "12 Street");

assertEquals(true, cs.addContact(test1));

}

In addition to this, I worked on a task object that met specific criteria, similar to the contact object. The task object shared some common attributes, such as the required ID and name field. However, there was a required description string field with a limit of 50 characters, and the name string could not be longer than 20 characters. For the task service testing procedure, I developed code to ensure that the task service could add tasks with a unique ID, delete tasks, and update tasks per task ID. The updatable fields include the name and description strings. Below is a code snippet of the task object JUnit test that verifies valid task data, and the task service test that confirms functionality of the task duplicate operation.

class TaskTest {

@Test

void createValidTaskData() {

Task task = new Task("ID", "name", "description");

System.out.println(task);

}

}

@Test

void noDoubleIDTest() {

Task test1 = new Task("ID", "name", "description");

TaskService.addTask(test1);

Task test2 = new Task("ID", "name", "description");

assertEquals(true, TaskService.addTask(test2));

}

Finally, I created an appointment object that had to meet a few criteria. The object required a unique appointment ID that was no longer than 10 characters, a date field that could not be in the past, and a description field that cannot be longer than 50 characters. All three fields shared a common requirement where they could not accept null input. The appointment service was required to add appointments with a unique appointment ID, and delete appointments per ID. An example of the appointment service testing addition feature is shown below.

@Test

void apptAdd() {

Date date = new GregorianCalendar(2021, Calendar.NOVEMBER, 23).getTime();

Appointment test1 = new Appointment("ID", date, "description");

assertEquals(true, AppointmentService.addAppt(test1));

}

The overall quality of my JUnit tests for each service exceeds my personal expectations and are efficient in ensuring that my code meets the project requirements. The tests are a helpful way to ensure that the program meets the specific requirements, and allows you to determine whether additional testing is necessary based on the coverage percentage. I am very confident in the effectiveness of my JUnit tests based on the excellent coverage percentage, since the coverage refers to the amount of code that was ran through the tests.

In order to ensure that my code was technically sound and efficient, my program went through several processes. I try to maintain a consistent pattern of testing my code often, to make sure that it is of the highest quality standard, and free of errors. The JUnit tests were specifically designed to run my code through different scenarios to test each individual feature, and allow me to see how the program will react. While working on each service, my code was refactored many times, featuring modifications made from my JUnit test results. For example, I kept my code simple, while ensuring that my JUnit tests would cover an acceptable percentage. Take a look at a snippet from the appointment test object, which verifies whether the program has received valid appointment data.

class AppointmentTest {

@Test

void ApptTest(){

Date date = new GregorianCalendar(2021, Calendar.NOVEMBER, 23).getTime();

Appointment appt = new Appointment("id", date, "description");

System.out.println(appt);

}

}

During the development of each milestone assignment, I utilized JUnit tests to ensure that the software met specific requirements. As part of a software testing procedure, JUnit was used to test my code against predetermined input as part of a use-case scenario. The module milestone assignments had to meet a list of requirements, such as string field character count limitations. Each milestone also required a service that can perform several actions on class object IDs. For all of the milestone assignments, I wrote test files that use the JUnit test/assertion API to run my code through a pass/fail test case situation. Simply put, JUnit assertions are helpful in validating expected output with the actual program output.

One of the software testing techniques that I did not utilize during my work on the milestone assignments was regression testing. It helps determine whether certain changes to software did not break any existing functionality by testing the changes that were made. I also did not utilize experience-based techniques, which derive test cases based on the tester’s experience. I believe that each testing technique has it’s own pros and cons that change depending on the individual testing scenario. Each project has its own requirements and one technique may be a better option than another technique.

As a software tester, I employed caution by confirming that the tests maintained a high coverage percentage, and performing a manual review of the code. I believe that it is important to appreciate the complexity and interrelationships of the code that I am testing because it can help me provide effective feedback to the developer. It is one thing to confirm that a program works, but it is an entirely different thing to understand how it works. In the same sense, this would be helpful in determining why something may not work. Being knowledgable in the interrelationships of the code can also aid in creating effective test cases, which is beneficial to the entire team.

During the code review process, it is important to limit bias so that the final deliverable project will be of the highest quality. For this project, I limited any potential bias by sticking to the objective results from my JUnit tests, and constantly referring to the customer requirements. If I were testing my own code on the software developer side, I could see bias as a potential concern, but one that can be mitigated. Ways to limit bias in this scenario would be to stick to the requirements and use an objective testing procedure.

Finally, my commitment to quality as a software engineering professional is of the highest importance. Being disciplined in this commitment is a life-long process that I aim to improve every single day of my professional career. It is worth the time to focus on writing code that is efficient, technically sound, and error free, in order to meet both my personal and career goals of consistently providing a product of the highest quality.

As a practitioner in the field, I plan to avoid technical debt by working with my teams to prioritize proper coding solutions, rather than speedy, limited solutions. When writing or testing code, I do not believe in cutting corners or taking shortcuts, as one mistake can cost the entire project, or more. I cannot think of a scenario where releasing an incomplete product would be a good idea, as the code would eventually need to be refactored, costing more in time and resources.