**7-2 Project Two**

Rodrey McCoin

CS-320: Software Test, Automation

Professor Smith, Travis

August 18, 2024

**Summary:**

For each of the three features of the Grand Strand System, I approached each with a unit testing strategy that met each of their requirements. Each feature had specific requirements that needed testing to make sure that they ran correctly. For the Contact class I had to create tests that would test to make sure that the contact ID, contact firstName and contact lastName was no longer than 10 characters and was not null and for the contact ID, it should not be able to be updated. The phone string should be exactly 10 digits, the address should be no longer than 30, and neither should be null. To test this, I used a test that created a contact and tested to make sure the contact was not null, and a test that confirmed each value met the requirements. I also had to create tests to add, delete, and update the contact and its information.

For the Task class there were similar requirements for the objects identified. The task ID should be no longer than 10 characters, not null, or updateable. The task name should be no longer than 20 characters and not be null, and the description should not be longer than 50 characters or null. To test the task requirements, I ran tests that exceeded each of these requirements and threw an error message.

For the Appointment class the appointment ID cannot be longer than 10 characters, not null, and cannot be updateable. The date object cannot be in the past and cannot be null. The description string cannot be longer than 50 characters and cannot be null. The appointment service class should be able to add appointments with a unique appointment ID and delete appointments per appointment ID. To test the objects, I ran a test that exceeded the requirements and threw an error message. For the Appointment Service class, I ran tests that ensured that updated, added, and deleted appointments, and then ensured that each of these were performed correctly.

To ensure that my code was technically sound I used code that tested each object and made sure they met the requirements. For example, I used a test that input a task ID that exceeded 10 characters and threw an error message.

@Test

@DisplayName("Task ID cannot have more than 10 characters")

void testTaskIDWithMoreThanTenCharacters() {

Task task = new Task("12345678912", "name", "description");

if (task.getTaskID().length() > 10) {

fail("Task ID has more than 10 characters.");

}

}

To ensure that my code was efficient, I ran JUNIT tests that checked the accuracy of the tests I conducted. When performing the JUNIT tests, I ran into problems where some of my tests were not covered, meaning they were not efficient when testing my code. Examples of this included

@Test

@DisplayName("Description cannot be longer than 50 characters")

void testDescriptionWithMoreThanFiftyCharacters() {

Appointment appointment = new Appointment(null, Date(2024, Calendar.AUGUST, 4), "123456789 is nine characters long" + "123456789 is another nine characters long" + "123456789 is another nine characters long" + "123456789 is another nine characters long");

if (appointment.getDescription().length() > 50) { fail("Appointment Description has more than 50 characters.");

}

}

Even though the code ran with no errors, the JUNIT test had trouble covering this test.

**Reflection:**

The software testing techniques that I employed on this project are static testing, unit testing and JUNIT testing. The unit testing techniques allowed me to write test cases so that I could test the objects and their requirements. The unit tests also allowed me to run test cases to make sure that each service class was able to add, delete, and update their objects. A unit test example to make sure the Task name is not null is,

@Test

@DisplayName("Task Name shall not be null")

void testNameNotNull() {

Task task = new Task("TaskID", "null", "description");

assertNotNull(task.getName(), "Task Name was null.");

}

The JUNIT testing techniques allowed me to verify that the individual unit tests, were covered, so that I could see how effective my tests were to the code I wrote for each class.

Software techniques that I did not use were integration testing and regression testing. Integration testing would not work for this project because integration testing works best when testing how well different pieces of the application work together. For this specific project, we are testing how each piece of the applications works, rather than how they work together as a whole. This testing would work better down the line, once the rest of the application is created. Regression testing is a type of performance testing, that tests the application as a whole. This wouldn’t work, because the application is not yet complete.

Unit testing is a good technique to use for this application because it allows each function or class to be tested individually. This type of test works because for the development of projects before the whole application can be tested to see how well it works together. Regression testing is used to retest functions that have preciously been tested to ensure stability and reliability. “Regression testing helps maintain the overall quality of the software” (GeeksforGeeks, 2024). Integration and regression testing work best when all the different classes of the project are combined and are tested to see how well they work with each other.

During this process I had to pay close attention to detail and adapt to critical thinking. When creating my classes I would create code that would look like it had no flaws. But once running the JUNIT tests, I would see just how much of my code was not covered by my tests and must think critically on how to overcome these obstacles. This also shows how I had to employ caution. When creating my test cases I had to be aware of how each unit was going to work with each other, which determined how the test should be written. When running the JUNIT test, I came across low coverage on each of the three cases we were to create. For example, my appointment and appointment service classes. In the beginning each of these classes were getting low coverage percentages with the test classes I created for each of them. After working through each I was able to organize the code and create test cases that were covered and tested each function correctly.

The way that I tried to limit bias in my review of the code was to think about how the code should run and now how I think my code should work. If I came across low coverage when testing my code, I did not hesitate to change things and make them work. I kept an open mind and came up with solutions on how to fix the code, rather than get stuck on how it was already written. Just like I mentioned before, in my appointment classes I ran into low coverage, just like I did in my other classes the first-time testing. But once I figured out where I was going wrong, I was able to increase the coverage by editing my tests, along with organizing my main classes.

“An undisciplined developer will not be able to ship on time and will not write code that is easy to maintain. A disciplined developer will not only enable the success of a project but will raise the level of productivity in others” (Atwood, 2023). Having discipline is an important skill to maintain in most careers. As a software engineer professional, discipline plays an impact on your productivity as well as the team working alongside you. Cutting corners when writing or testing code can cause issues down the road in the development process for other members of your team. The errors can be harder to find as the code becomes more developed, and becomes more time-consuming, wasting the team as well as the clients’ time and money. To avoid technical debt as a practitioner in the field I can run unit tests on each element of the program along the way, as well as other tests once each piece of the program comes together. That way flaws can be detected ahead of time throughout each phase of the process, keeping the project on task and avoiding wasted time and money in the outcome. Examples of this include unit tests such as those included in the project we conducted, to test object requirements and tests to make sure that specific functions are performing correctly.

**References**

Atwood, J. (2023, May 20). *Discipline makes strong developers*. Coding Horror. https://blog.codinghorror.com/discipline-makes-strong-developers/

GeeksforGeeks. (2024, June 13). *Regression testing - software engineering*. https://www.geeksforgeeks.org/software-engineering-regression-testing/