Michael Isenhour

CS 300

Assignment 5-2

Project 2 Summery and Reflection

Summery:

When it comes to the extent my testing approach aligned with the software requirements, I would have to say they aligned very closely to what was being required. In both the Task and Contact classes I made sure to create a unique ID that I also finalized making sure it couldn’t be altered. I also made to that I only had getter methods for getting the ID but no setter methods for setting it. This means that it could only be created once and not updated. After setting up the constructor and getters and setters in that Contact and Task classes I would test those classes and make sure things ran smoothly. For example these tests in TaskTest.java would check and make sure IllegalArgumentException would be thrown if ID, name, or description was too long or null therefore making sure requirements were met. Under ContactTest.java for testing my Contact class I did the same thing to make sure the ID, firstName, lastName, phoneNumber, and address were all correct. I also ran a test to check for phone number errors such as the user adding non digit characters. When these tests were all successful and made sure that the requirements were met I moved on to contact service and task service.

Thought this is my first time ever seeing and running Junit tests I believe that the quality of my tests were very successful. In contact service and task service I made sure that my IDs were unique by checking and making sure I would get an error if someone ever tried to make a duplicate ID. I also made sure that other requirements were met such as being able to add Contacts or Tasks as well as Deleting Contacts and Tasks or updating things such as Contact names and Task Descriptions, etc. In the end after all my tests were complete they successfully passed and made sure all requirements in the documentation were met.

@Test

public void testAddDuplicateContactID() {

TaskService service = new TaskService();

Task task1 = new Task("0002", "Second Task", "A Secnod Task");

service.addTask(task1);

Task task2 = new Task("0002", "Next Task", "Complete After First Task");

assertThrows(IllegalArgumentException.class, () -> service.addTask(task2));

}

This test in TaskServiceTest.java for checking and making sure there aren’t duplicate IDs is a good example of how my code is technically sound. In this test I make sure that if a user tried to duplicate an ID either accidentally or intentionally it would reject it and send an IllegalArgumentException therefore making sure that the ID remained unique for each task and no task had the same ID number.

I ensured that my code was efficient by making sure my test cases were specific and focused on specific functionalities such as adding a task, deleting a task, updating a task, etc. I also made sure that my assertions were easy to understand and very clear such as using assertNotNull after adding a task to make sure the added task wasn’t null. Then once knowing it wasn’t null I used assertEquals to make sure the information was correct. I also set up a new TaskService in each function to make sure that each test is independent from other tests.

// Tests that a new contact can be successfully added to the TaskService

@Test

public void testAddTask() {

TaskService service = new TaskService();

service.addTask(new Task("0002", "Second Task", "A Secnod Task"));

assertNotNull(service.getTask("0002"));

assertEquals("0002", service.getTask("0002").getTaskID());

assertEquals("Second Task", service.getTask("0002").getName());

assertEquals("A Secnod Task", service.getTask("0002").getDescription());

}

Reflection:

When it comes to the software techniques that I used for these three milestones the techniques I relied on was that of white box and black box testing. Since our milestones gave us the understanding of the internal logic and what to test for, white box testing was the best approach. With all my projects that I did I created the java files first making sure to keep to the boundaries and requirements. Once they were written, I then created the testing files to test that the logic I had was correct and ran as it should. For example, I coded Appointments.java making sure to put in checks to make sure things like the ID wasn’t null and was unique. I also checked to make sure that lengths were the right length etc. After doing this I created AppointmentTest.java to create an appointment with valid and invalid inputs to validate functionality. This turned out to be very beneficial as my unit tests worked flawlessly. Also, because the programs had relatively the same requirements, I was able to reuse a lot of the tests and code in each project. For example, testing my adding and deleting of contacts, tasks, and appointments withing the ServiceTest files.

Since these projects were just for the milestones and not real-world development, I didn’t bother to use non-functional testing or acceptance testing. Acceptance testing would involve end-user testing which wasn’t necessary because there would be no real user using these applications except our professor just testing to see if our JUnit tests were successful. I also didn’t use Integration testing as there was no need to see how the application would perform with other components such as a database or user interface since there was no database and no users would be using my program. I also didn’t worry about performance testing as response times and resource usage didn’t concern me since again this wasn’t going to be a deployable application and was just for these assignments.