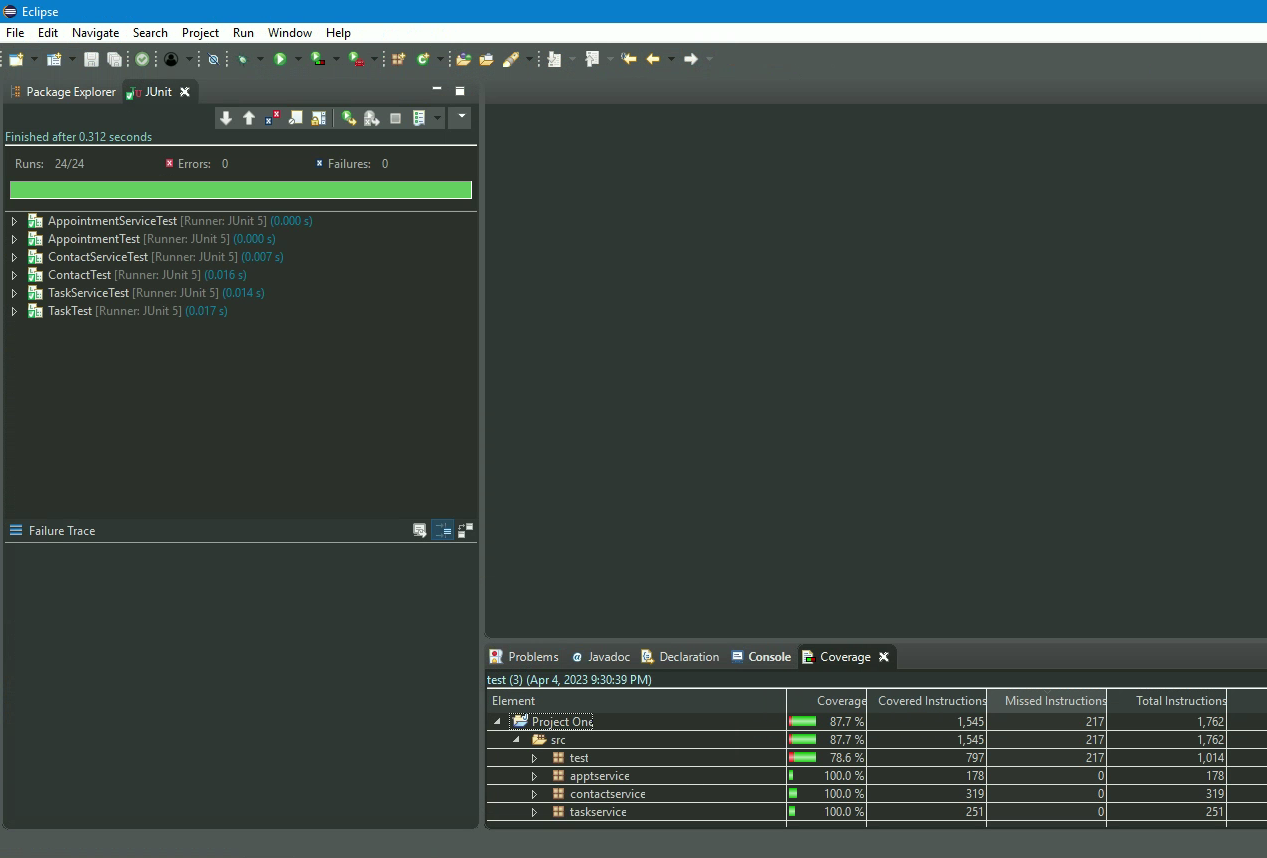
Nicholas Glover

4/12/2023

**CS-320: Project Two Summary and Reflections Report**

My software testing aligned with the software requirements because I used the lists of software requirements as a checklist for creating tests for the classes. The Contact, Task, and Appointment objects were required to have an ID number that couldn’t be null or exceed 10 characters. I made sure to write tests that would make sure a valid ID would pass through and that an ID that was too long or null would be rejected with the proper exception and error message. I followed that same logic when designing the tests for every requirement of each object and service. I needed to write the test slightly differently when I was testing the Appointment object and service due to the heavy amount of deprecation in the java.util.date object. I needed multiple date objects to compare against so I created a future date object and a past date object so that I could make sure the Appointment object and service would react properly and throw the exception it was supposed to if the date wasn’t valid. I know that I wrote high quality tests for the assigned objects and services because the testing coverage for them was 100%.



I ensured that my code was technically sound by testing the necessary edge cases as listed in the software requirements. As an example:

“@Test

void testTaskCreateIsNull() {

assertThrows(IllegalArgumentException.class, () -> {

new Task(null, "Task #1", "Completes Task #1");

});

assertThrows(IllegalArgumentException.class, () -> {

new Task("12345", null, "Completes Task #1");

});

assertThrows(IllegalArgumentException.class, () -> {

new Task("12345", "Task #1", null);

});

}”

I have a full block of testing code reserved to only testing if the Task object constructor properly throws an IllegalArgumentException if any of its parameters are null. I’ve made sure that my testing code is efficient by making sure to write the fewest tests required for total testing coverage.

I employed a mix of both black- and white-box testing in the milestone assignments. I implemented those testing techniques through the use of unit testing, specifically JUnit 5 testing. Black-box testing involves not knowing the internal processes of the function or method being tested. It simply takes the input and output of the function and compares it against the expected output given the input. I used that form of testing for the basics of the code I was writing like testing the getters and setters for the object code I wrote. White-box testing involves knowing and testing the internals of the function or method being tested. I used white-box testing to determine that the requirements that specified that the object would throw a specific exception if any of the data were null, worked properly.

One type of technique that I didn’t implement in my testing were experience-based techniques. I didn’t use them because I don’t have the expertise necessary to make them worthwhile yet. I expect those techniques will become more and more viable as I learn the ins and outs of writing code to follow specific requirements.

The three types of software testing techniques I’ve discussed all have their practical uses and implications. Black-box testing is very useful when testing pre-existing code that you might not be as familiar with and has been tested before. Things like legacy code or code that’s been grandfathered into a system come to mind. White-box testing is very useful for code that has been newly written and is in the beginning stages of testing. It’s very important because it will thoroughly test all the aspects of the code’s inner workings. Experience-based testing is important when there is too little time for the more structured techniques to be entirely useful. It relies on the expertise of the team testing the software so it can be used to find defects in the system that the other two techniques might not find easily. Because of this, experience-based testing can be implemented in addition to the other two software testing techniques.

When writing the code for the objects, I tried to be cautious when deciding how to address the software requirements. Since all of the objects had the requirement of not allowing an input over a certain length, I made sure to include input length checking into the constructor, so that it would be verified at the earliest point in the object’s initialization. I tried to limit bias when writing the tests for my own code. I don’t know how successful I was because bias is always something that’s difficult to test for on a one-off basis. I think it’s very important to limit bias when writing tests. Whether that’s addressed by having someone else write the tests or by changing the approach to testing, it limits the chance of simple mistakes or errors get passed testing phases. Discipline is incredibly important when it comes to software testing. Cutting corners can lead to some drastic flaws being released in the consumer product and leading to much larger issues over time. Since flaws in the code are much easier to solve and fix the earlier in the development phase they’re found, a lack of discipline can lead to very expensive problems indeed.