Jacob Rethmeier

Professor Bermudez

CS-320

2/23/2025

7-2 Project Two

In Project One, I used a structured unit testing approach for each of the three primary features: contact management, task management, and appointment scheduling. For contact management, I used unit tests to verify that contacts could be successfully created and manipulated. While doing this I also made sure to ensure that each function adhered to the constraints of valid inputs. The task management tests validated task creation, deletion, and modification while enforcing the character limit and required fields were filled out. Appointment scheduling unit tests checked that there were no duplicate entries and ensuring valid time constraints. Each test/requirement was aligned with our tasks by validating expected inputs and outputs. By focusing on boundary conditions and edge cases, I ensured that the application handled both normal and abnormal situations properly. For instance, testing attempted input of null values or exceeding character limits confirmed that the application correctly threw exceptions and handled errors. An example would be this test case in which the purpose is to verify that there is no null entry or entry over the length limit.

@Test

void testSetName() {

Task task = new Task("12345", "Valid Task", "This is a task description.");

assertThrows(IllegalArgumentException.class, () -> task.setName(null));

assertThrows(IllegalArgumentException.class, () -> task.setName("This name is longer than allowed."));

task.setName("Updated Task");

assertEquals("Updated Task", task.getName());

}

To ensure that my Junit test was completed in industry the best practices I focused on having good readability and comments as well as verifying that I had test coverage for every essential function that I required. To ensure technical soundness, I wrote test cases that verified compliance with requirements and exception handling. While verifying my tests were up to the quality, I wanted I also was making sure that efficiency was a priority to reduce on redundancy.

@BeforeEach

void setUp() {

taskService = new TaskService();

}

The main testing techniques that I used in this project were static testing and unit testing. However, I was still mindful of boundary value analysis, equivalence partitioning, and exception handling testing. For the static testing I took a good portion of time and just went through each line verifying I didn’t make any minor mistakes that would be larger issues later. The unit testing focused on testing individual methods and classes in isolation by utilizing a separate testing file.

Some of the testing techniques that I didn’t use included integration testing, security testing, system testing, and performance testing. While all of these are very important, they each have scenarios where they are required and for this one I had more beneficial methods. Integration testing is used to ensure that different components of a system are all working together correctly. Security testing is used to identify vulnerabilities in a system and is used to protect them from threats.

While working on this project it was a priority for me to adopt a cautious and meticulous mindset as I approached the requirements. Which made me realize the need to test all the edge cases. One example to prevent potential runtime errors was:

assertThrows(IllegalArgumentException.class, () -> new Task(null, "Task", "Description"));

Bias can be a concern when testing one’s own code, and as someone pursuing to be a developer I might unconsciously assume my implementation is correct and overlook certain failure conditions. To combat this, I plan on befriending coworkers and often consulting them for peer reviews because I could miss something they might see.

Cutting corners in testing can lead to technical debt, among many worse things and resulting in harder-to-maintain code. As we just discussed in our discussion board, cutting corners can lead to major failures which could be avoided. For example, skipping validation tests might cause issues when new features are integrated. I plan to avoid technical debt by following my favorite practice of Test-Driven Development (TDD), which ensures a clear focus on requirements before implementation. I will also use code reviews to catch potential issues before they become significant problems. Additionally, automating testing with continuous integration tools will help maintain consistent validation of code changes.

Conclusion

This project reinforced the importance of structured unit testing and adopting a meticulous, unbiased mindset as a software tester. By applying best practices and ensuring high-quality test coverage, I was able to confirm the robustness of the application while maintaining efficiency and scalability. Moving forward, I will continue prioritizing thorough testing to enhance software reliability and reduce technical debt.