CS – 320 Project 2 Write Up

1. Summary.

For the Contact and ContactService classes, I took a hands-on approach to testing by checking both how individual contacts were validated and how they were managed within the service. I made sure all the key requirements were covered, like limiting name and ID lengths, ensuring the phone number was exactly 10 digits, and only allowing updates to certain fields. My JUnit tests reflected these rules directly, and they all passed, which gave me confidence that the code worked as expected. I also tested how the service handled real-world actions like adding a new contact, preventing duplicates, deleting, and updating just parts of a contact's info. I knew my tests were effective because they covered all the major functions and edge cases, with nothing slipping through. For example, I tested updating just the first name while leaving everything else untouched, which helped confirm that the update logic was working properly. I kept the tests clean and efficient by reusing objects and writing direct, focused assertions, nothing overly complicated. Overall, writing these tests helped me feel good about the quality and reliability of the code.

For the Appointment and AppointmentService classes, my testing approach focused on ensuring the core functionality matched the requirements, such as making sure appointment IDs were unique, the date wasn’t in the past, and descriptions stayed within the length limit. My JUnit tests first focused on validating input to the Appointment constructor-checking that it throws exceptions when something’s off-and then shifted to the AppointmentService, where I tested adding, retrieving, and deleting appointments. Everything passed, which told me the code was behaving as expected. I aligned my tests closely with the requirements by creating scenarios that mimicked real use, like trying to add an appointment with a duplicate ID or a past date. I knew my tests were effective because they covered all possible fail cases and kept things clear and easy to follow. One specific example I tested was ensuring that addAppointment() throws an exception when a duplicate ID is used-that confirmed my logic for maintaining unique entries. I kept things efficient by only writing the tests needed to verify each rule, without adding extra fluff. Overall, the testing helped me feel confident that both classes are solid and ready for real use.

For the Task and TaskService classes, I focused my unit tests on making sure tasks were created and managed according to the rules-like ensuring the ID is under 10 characters, the name under 20, and the description under 50. I tested both good and bad inputs to the Task constructor to verify that invalid data was properly rejected. In the TaskService class, I tested all the core features: adding tasks, updating them, deleting them, and retrieving them by ID. Since all my tests passed, I saw that the logic in both classes was solid and matched the requirements. I knew the tests were effective because they covered different scenarios, like trying to update a non-existent task or checking that a task is properly removed. I also kept the tests efficient by directly testing what mattered, without unnecessary duplication. For example, in one test, I created a task, added it to the list, updated the name, and then confirmed the change-this kind of step-by-step validation gave me confidence in both the design and implementation. Writing these tests really helped reinforce the reliability of the task management code.

Writing the JUnit tests was a smooth experience overall. I made sure each test was focused, relevant, and directly tied to the functionality outlined in the project requirements. Since I had already implemented clear validation rules in the classes-like ensuring IDs weren't null and met length restrictions-it was easy to write tests confirming that those rules were enforced. For example, in my test for the Contact class, I included a case where I passed a phone number with fewer than 10 digits to confirm that the constructor would throw an IllegalArgumentException. Similarly, I tested the Appointment class by passing in a date in the past to ensure that new Appointment(...) would fail as expected. These kinds of checks helped me confirm that my code was technically sound and defensive against invalid input.

To make sure the code was efficient, I avoided redundant tests and made sure each test targeted a distinct part of the logic. For instance, in the TaskService tests, I created a single task, added it to the list, and then used updateTaskName(...) and updateTaskDesc(...) to verify that both updates worked as intended, all within one test case. This approach reduced overhead while still thoroughly checking the service’s core functionality. With all my tests passing and a coverage score of 81%, I felt confident in the quality of both my tests and the underlying code. That percentage falls in line with common industry standards, and since I tested every meaningful branch and edge case, I was satisfied that the tests provided solid, real-world validation of my project.

1. Reflection

In my project, I employed both dynamic and static testing techniques to ensure my code was reliable and functional. For static testing, I performed manual code reviews, carefully checking for logical errors, inconsistent naming conventions, and other potential issues before running any tests. This approach helped me catch problems early, reducing the risk of bugs later in the development process. On the dynamic side, I ran unit tests to ensure the functionality of each method and class, validating that the code worked under real conditions. By using these techniques together, I was able to identify issues from both a high-level design perspective (static testing) and a detailed, execution-focused perspective (dynamic testing). While I didn’t use other testing techniques like integration or regression testing for this project, these techniques have distinct advantages in larger or more complex software projects. Integration testing, for example, would be useful when ensuring that multiple components or services work together as expected, especially in applications with many interconnected modules. Similarly, regression testing helps ensure that new changes don’t inadvertently break existing functionality, making it invaluable in projects with frequent updates or ongoing development. Both techniques are practical for ensuring the robustness and scalability of software in larger teams or production environments.  
  
3. Mindset.

When working on this project, I adopted a mindset of caution and patience, fully appreciating the complexity of the code and its interrelationships. It’s easy to jump to conclusions and assume there’s a major error when something goes wrong, especially in a larger and more developed codebase. However, I recognized that not every issue is a catastrophic failure. It's important to take a step back, thoroughly review the problem, and approach it logically. For example, when testing the TaskService class, I didn’t immediately assume an issue when the task list wasn’t updated as expected. Instead, I traced through the method, reviewed the logic, and realized the issue was a minor oversight in the update method, not an overall system failure. This cautious approach helped prevent hasty decisions and ensured I wasn’t overlooking potential solutions that could be more straightforward.

Bias is a natural concern when reviewing your own code, especially as a developer. It’s easy to fall into the trap of thinking your code is perfect or, on the other hand, excessively criticizing it out of a lack of confidence. For instance, when testing my own code, I could have been too lenient and missed minor bugs because I assumed the logic was sound, or I could have overcomplicated things by assuming the code was wrong without fully testing it. I tried to limit bias by taking a step back and objectively reviewing each part of the code, as though I were testing someone else’s work. When I found a mistake in my code, I didn’t view it as a failure, but rather as an opportunity to learn and improve. This mindset helped me maintain a balanced perspective, preventing overconfidence or self-doubt that can often cloud judgment during testing.

Being disciplined is essential in both development and testing. When writing or testing code, cutting corners may seem tempting for the sake of speed or ease, but it can lead to significant issues down the road. For example, failing to implement proper input validation or rushing through testing could introduce security vulnerabilities or bugs that could become costly and time-consuming to fix later. As a practitioner, I believe it’s crucial to balance speed with thoroughness, following industry standards and best practices to ensure high-quality, maintainable code. By committing to this disciplined approach, I avoid the temptation of shortcuts and the risk of technical debt. For instance, even though my tests passed with 81% coverage, I still reviewed edge cases and scenarios that might not have been covered by standard tests, ensuring long-term stability and reliability in the project.