Project Two

**Summary**

* Unit Testing Approach
  + For the contact, task, and appointment services, my unit testing approach was focused on validating individual fields and behaviors based on the specific requirements given. For the Contact class, I wrote tests to make sure the ID was unique and no longer than 10 characters, first and last names were under 10 characters, the phone number was exactly 10 digits, and the address was under 30 characters. Each field had its own set of validation tests using JUnit's assertThrows() method for invalid cases and assertEquals() for valid cases. This approach was closely aligned with the requirements because each field validation was explicitly tested based on the specifications provided in the project directions. For example, in ContactTest.java, I wrote:

*assertThrows(IllegalArgumentException.class, () -> {*

*new Contact("12345678901", "John", "Doe", "1234567890", "123 Main Street");*

*});*

This ensured that IDs longer than 10 characters were properly rejected.

* + I can defend the overall quality of my JUnit tests based on the coverage percentage, as I achieved over 90% coverage across all classes. This means that nearly all branches, conditions, and possible exception cases were tested, not just the basic success paths. High coverage, according to García (2017), is a strong indicator that the tests are effectively validating the behavior of the code, helping to catch both obvious errors and hidden edge cases. While coverage alone doesn't guarantee perfect tests, it shows that the code was thoroughly exercised and that major issues would likely have been caught during testing.
* Experience Writing JUnit Tests
  + Writing the JUnit tests helped me make sure my code was technically sound by focusing heavily on validation. I wasn’t just checking if the methods worked under normal conditions, I also made sure to test edge cases, invalid inputs, and exceptions. For example, in AppointmentTest.java, I tested that dates in the past would throw exceptions:

*assertThrows(IllegalArgumentException.class, () -> {*

*new Appointment("A123", pastDate.getTime(), "Check-up appointment");*

*});*

This line confirmed that the appointment date logic was working correctly.

* + Efficiency was also a priority because instead of writing redundant tests, I reused setup methods and organized my tests logically with @BeforeEach, which kept the code clean and reduced unnecessary duplication. By setting up common objects and data before each test, I avoided repeating the same initialization code in every test method. This not only made the tests easier to read and maintain but also made it faster to update the tests if changes to the setup were needed later. An example from TaskServiceTest.java:

*@BeforeEach*

*public void setup() {*

*service = new TaskService();*

*task = new Task("123", "TaskName", "Task Description");*

*service.addTask(task);*

*}*

This made sure each test started with a fresh and consistent environment.

**Reflection**

* Testing Techniques
  + The main software testing technique I used was unit testing, which focuses on testing small, isolated parts of a program, like individual classes and methods. Unit testing is useful for verifying that a single component behaves as expected without relying on other parts of the system. In my project, unit tests allowed me to quickly catch mistakes in field validations and business logic before they could cause larger problems. Using assertions to check both valid and invalid inputs helped make sure that each method handled different scenarios properly and consistently (Jakubiak, 2022).
  + Other techniques that I did not use include integration testing, system testing, and static testing. Integration testing checks how different modules work together and would be practical if my services needed to interact with each other. System testing validates the entire application end-to-end and is usually done before full release. Static testing, like code reviews or inspections, could have been useful early on to catch simple mistakes before executing the code (García, 2017).
  + Unit testing is ideal for smaller, isolated services like the ones I built because it checks if individual pieces of code work as expected. However, in larger projects with complex systems, integration and system testing are critical. Integration testing makes sure different modules or APIs work together properly, while system testing checks that the whole application functions correctly from start to finish. Without these tests, it’s easy to miss bugs that happen when components interact.
* Mindset
  + While working on this project, I tried to adopt a cautious and thorough mindset. I approached testing with the idea that even simple-looking methods could fail in unexpected ways, so I made it a point not to assume anything without verifying it through tests. For example, even though the phone field validation in the Contact class seemed easy, I tested edge cases like null values, incorrect lengths, and non-numeric characters separately to be sure. It was important to appreciate the complexity of the code because even small validation errors could affect how the service handled updates or deletions later. If one part of the logic failed silently, it could cause cascading issues across the ContactService methods. By treating each method and its connections carefully, I was able to catch issues early and prevent small problems from turning into bigger ones.
  + Limiting bias was another important part of the project. It’s really easy to assume that your code just works because you wrote it. To avoid bias, I actively wrote negative tests, such as trying to break my code with invalid inputs, instead of only checking for success cases. If I had only tested happy paths, I could have easily missed edge cases. Bias would definitely be a concern if I was testing my own production code in the real world. Without independent testers, it’s easy to miss mistakes simply because you're too close to your own work. To deal with this, in a real job, I would make sure to get peer code reviews and have separate people review my tests.
  + Being disciplined about quality is extremely important as a software engineer. Cutting corners might save time at first, but it usually leads to technical debt as bad code that you’ll have to fix later. One way I plan to avoid technical debt is by sticking to strict testing and documentation habits even when under pressure to move quickly. It’s easy to skip writing tests or documenting changes when deadlines are tight, but that usually leads to bigger problems later. To avoid this, I plan to always write basic unit tests whenever I add a new feature, even for small changes, to make sure new code doesn’t introduce hidden bugs. I will also use tools like coverage reports to catch any untested areas before merging code into production. Taking the extra time to maintain good testing and documentation standards helps keep the codebase clean, easier to maintain, and less likely to cause unexpected failures down the road.

**References**

García, B. (2017). Mastering software testing with JUnit 5. Packt Publishing.

Jakubiak, N. (2022, December 6). JUnit tutorial: Setting up, writing, and running Java unit tests. Parasoft. https://www.parasoft.com/blog/junit-tutorial/