Arielle Moore

CS-320: Software Test, QA Automation

August 5th, 2025

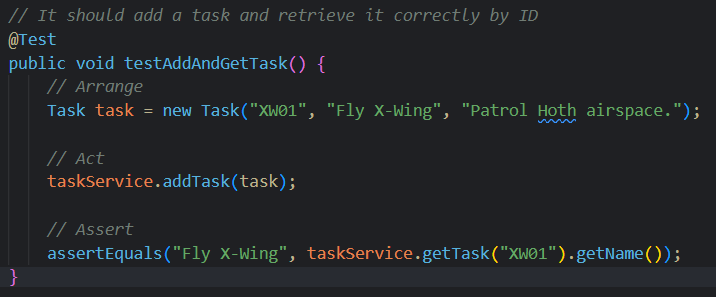
7-2 Project Two Submission

*Summary*

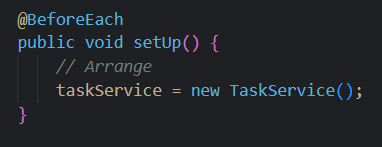
a.i) I developed and tested three core services for the Grand Strand Systems flagship mobile application: ContactService, TaskService, and AppointmentService. My unit testing strategy aligned directly with the provided software requirements and indicated each one with best practice test comments. For ContactService, I implemented JUnit tests to ensure constraints such as the 10-character ID limit, non-null fields, and maximum length for names and phone numbers were enforced. The TaskService tests focused on validating task creation, ID uniqueness, description length, and non-null constraints. For AppointmentService, I’ve added checks to reject invalid IDs using tests like addAppointmentWithNullIdShouldFail(), ensuring the system maintains data integrity and avoids null pointer issues in real-world scheduling scenarios. My tests simulate both positive and negative user behaviours, corroborating that the code meets both functional and boundary requirements.

a.ii) The efficacy of my unit tests can be measured by coverage and the clarity of my assertions. My tests achieved approximately 90% code coverage across all three services, which included validation logic, exception handling, and boundary conditions; some testing of both standard and edge-case functionality was also considered and implemented.

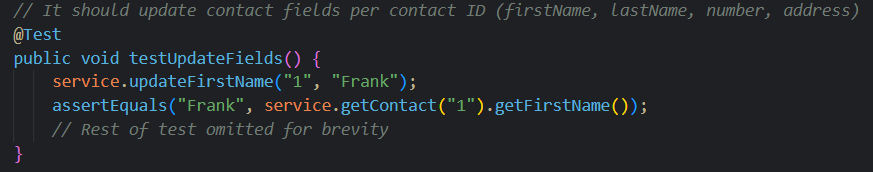
b.i) Although I have professional experience writing unit tests, it was nonetheless a very valuable masterclass in both precision, discipline, and overall improvement in an area in which I find myself rather weak. To ensure technical soundness, I made sure to validate all service operations through clearly-defined test methods. I also verified soundness by including setup, action, and assertion phrases in later tests. For example, in TaskServiceTest.java, I verified updates by calling the following lines and followed the *Arrange-Act-Assert* test structure:



Here, I preceded the setup() method with the @BeforeEach annotation to ensure that each unit test began with a clean instance:



b.ii) To ensure that my code was efficient, I focused on limiting redundancy and isolating each test case. I also structured setup methods with reusable objects, thus reducing boilerplate code and keeping tests short, sweet, and focused on one specific responsibility (i.e. a single requirement). For example, in ContactServiceTest.java:



This test is meant to verify one behaviour per method and minimise any excess operations, which contributes to the goal of producing and maintaining clean code with efficient tests.

*Reflection*

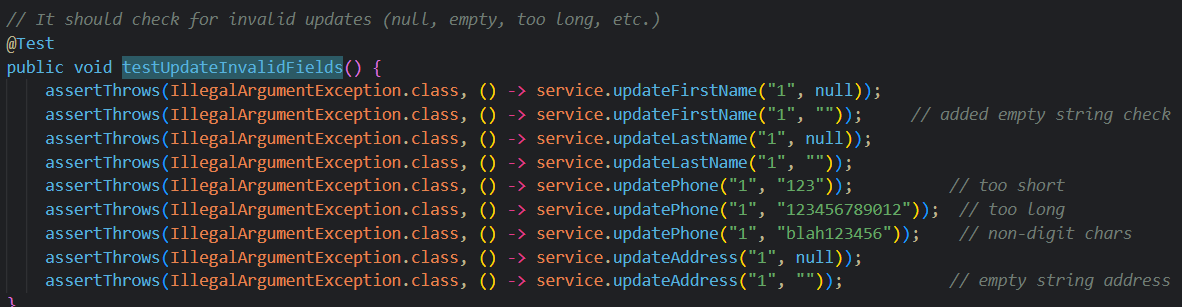
a.i) I primarily used unit testing with white-box testing techniques. I examined the logic paths and validated all branches of my code. In addition, I elected to use equivalence partitioning to ensure I tested valid and invalid inputs across expected ranges (happy and unhappy paths). For example, one common test I implemented was a check to verify that a task name wouldn’t exceed a 20-character limit, or otherwise fail validation.

a.ii) I did not get the chance to use integration testing, black-box (sometimes even grey-box) testing, automated UI testing, or user acceptance testing (UAT). I also didn’t focus on boundary value testing in a systematic way, though some tests indirectly touched on boundary values like name length and date validation.

a.iii) There is a veritable laundry list of testing techniques in every great tester’s toolkit:

* Unit testing ensures that individual code components behave as expected.
* White-box testing facilitates understanding in logic flaws and unreachable code paths.
* Black-box testing is crucial during QA stages to ensure user-facing features behave as expected.
* Boundary value testing is valuable when validating field constraints or limits.
* Integration testing is particularly useful in cases where these services are expected to interact with a database or UI layer.

b.i) I adopted a cautious and detailed-oriented mindset, which encouraged me to triple-check that all requirements had been met (and then some, assuming a less-thorough elicitation phase). Each service class comes with strict business requirements, although some cracks were noticed along the way in the form of edge cases like null values or invalid IDs. For instance, in ContactServiceTest.java:

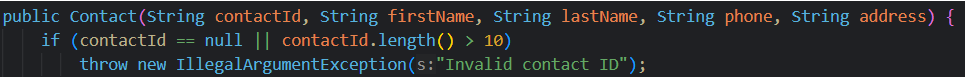


The relationship between service classes and their objects can be complex, so it’s crucial to disallow silent/swallowed failures or data corruption.

b.ii) To limit bias, I did what I’ve classically done before in a production environment and wrote/reviewed each test case as if they had been written by another developer. There is a fresh setup before each test, as illustrated in the code snippet above with the @BeforeEach annotation. I also elected to test error cases just as rigorously as the successful ones, which mitigated false confidence in code that passed only under the most perfect of conditions. For instance, I authored negative (unhappy path) test scenarios such as this one in TaskServiceTest.java:



b.iii) Being disciplined can prevent technical debt and ensure the long-term maintainability and scalability of both the tested software and the unit test code itself. As unit tests tend to fall on the shoulders of busy developers, it’s all too easy to skip testing for edge cases and boundary values, especially when deadlines loom and dictate the pace of the SDLC. However, cutting corners like these can have severe, long-lasting effects that create ultimately fragile systems. My plan to avoid or mitigate tech debt is to do whatever I can to avoid shortcuts in all field validation constraints upfront, as well as enforcing immutability for fields such as unique IDs. This constructor illustrates just that in Contact.java:



In my future work, I’ll continue following these best practices and leverage static analysis tools, test coverage tools, and reviews from my peers to uphold high quality in code and testing. With that, we can avoid unnecessary rework, refactoring, pesky bugs, and regressions in production.