For this project, I developed and tested three key features of a mobile application: the contact service, task service, and appointment service. My unit testing approach was focused on verifying that each object and its corresponding service class met the requirements provided by the customer.

For the contact service, my approach was to write the JUnit tests that confirmed every requirement was satisfied. The contact ID had to be unique, non-null, and immutable. First and last names could not exceed 10 characters, the phone number had to be exactly 10 digits, and the address could not exceed 30 characters. For example, I included tests such as assertThrows(IllegalArgumentException.class, () -> new Contact (“12345678901”, “John”, “Doe”, “1234567890”, “123 Main St”)); to validate that IDs longer than 10 characters were rejected. This matched the requirement directly and gave me confidence that my tests aligned with customer expectations.

For the task service, I used the same strategy of aligning tests to requirements, confirming that task IDs were unique, immutable, and under 10 characters. I also validated the name and description fields, ensuring they were non-null and within their character limits. Additionally, I tested updates to name and description fields through the service. For example, in TaskServiceTest.java, I included service.updateTaskName(“1”, “UpdatedName”); followed by assertEquals(“UpdateName”, service.getTask(“1”).getName()); to show that the update worked as intended.

For the appointment service, the main requirement was that appointments could not be scheduled in the past. I wrote tests that created an appointment with a past date, and I confirmed that the constructor threw an IllegalArgumentException. I also tested valid future dates and confirmed they were accepted. By tying my unit tests directly to requirements, I ensured my approach consistently matched customer needs.

Writing the JUnit tests gave me a clearer picture of how requirements map to code. For example, when testing the phone number validation in Contact, I included both valid (“1234567890”) and invalid (“123”) numbers. This process reinforced that my code was technically sound and covered likely failure points. Efficiency was also part of my goals, so I avoided repetitive code by reusing helper methods in my tests when possible, such as creating default objects to cut down on extra.

In terms of test coverage, I achieved 79% coverage across the classes. This meant that the majority of code paths, including edge cases, were executed during testing. Because I included both valid and invalid test cases, I can argue that my JUnit tests were mostly effective.

The main testing technique I employed in this project was unit testing. Unit testing focuses on verifying that individual classes or methods work correctly in isolation. The strength of this technique is its precision, as it allowed me to test specific requirements, such as rejecting null values or ensuring immutability without needing a full system or user interface. As AWS highlights, unit testing helps detect input, output, and logic errors early, saving time and effort before they reach production (AWS).

Other techniques I didn’t use include integration testing, which verifies that multiple components work together as expected, and system testing, which evaluates the entire application as a whole. For example, integration testing would have been useful if my services depended on a database, since I would need to check that connections and queries worked properly. I also did not use regression testing, which focuses on ensuring that new code changes do not break existing functionality.

Unit testing is most useful during the early development phases, when catching small issues quickly saves time and money. Integration and system testing are better suited for larger projects where multiple modules interact and the overall behavior must be validated.

While working on this project, I approached testing with caution and thoroughness. It was important to treat each requirement as achievable, since the customer’s trust depended on accuracy. For example, when testing appointment dates, I considered edge cases that went beyond obvious tests, utilizing dates exactly equal to new Date (). This was to ensure that the code functioned beyond simplicity.

Limiting bias in my testing was another focus. As the developer of the code, I knew how it was supposed to work and that my tests were more likely to give me the outcomes I wanted. To counteract that, I intentionally wrote tests that tried to break the code, such as providing nulls or overly long input strings. This helped me step back and test like any other person would. If I had only tested the code in the ways I expected to succeed, I would have missed potential errors.

Bias is a real concern when developers test their own code. It’s easy to unconsciously design tests that confirm what you expect rather than what you don’t. For example, I might have only tested valid phone numbers to check if they went through, but this would have hidden the possibility that my validation logic could break. Recognizing this tendency made me more disciplined in designing thorough test cases.

Finally, I learned the importance of being disciplined and not cutting corners in testing. Quality software requires consistent attention to detail, and skipping through testing leads to having to fix it later when the problems are bigger. My plan to avoid this includes writing clear, requirement-driven tests and to incorporate practices like regression testing and code reviews in future projects. These steps ensure that each new feature doesn’t just work but is also sustainable in the long run.

Citations

What is unit testing? - unit testing explained - AWS. (n.d.). https://aws.amazon.com/what-is/unit-testing/