Sumiko Mitchell

CS-320

Module 7-2: Project Two

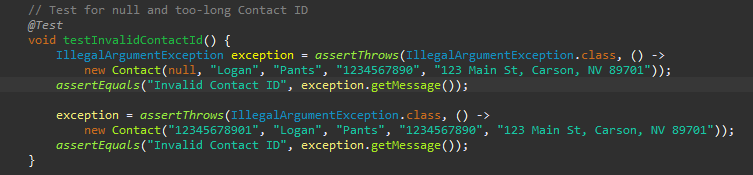
February 23, 2025

**Summary and Reflections Report**

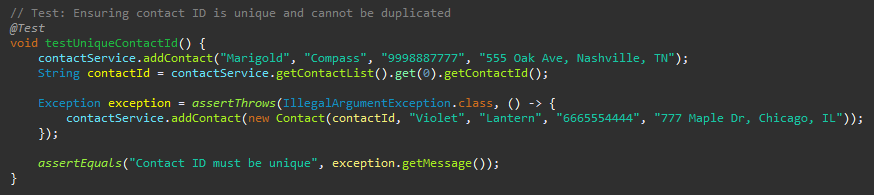
As a software engineer at Grand Strand Systems, I recently completed the development of a mobile application that delivers contact, task, and appointment services to our customers. This report summarizes my unit testing approach, my experiences writing JUnit tests, and my reflections on the overall quality and effectiveness of the testing strategies used throughout the project.

**Summary**

My unit testing approach was statistically aligned with the software requirements across all three features: Contact, Task, and Appointment services. I created specific test cases for each requirement that directly verified requirement constraints. For example, when testing the Contact service's requirement that "the contact object shall have a required unique contact ID string that cannot be longer than 10 characters," I implemented tests that verified both the length constraint and uniqueness.



ContactTest.java

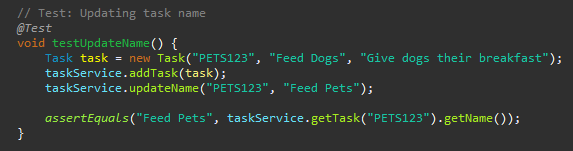


ContactServiceTest.java

The test coverage demonstrates my alignment with the software requirements—not only did I verify valid cases, but I also wrote tests to confirm that exceptions were thrown when these requirements were not met.

The effectiveness of my JUnit tests was demonstrated by their comprehensive coverage and systematic verification of requirements. I achieved 100% coverage for AppointmentService, ContractService, and TaskService, while the other test classes ranged from 76.5% to 97.7%, resulting in an overall coverage of 90% - well above the 80% requirement. Each test method focused on a specific functionality using a clear pattern of setup, action, and verification through assertions. This structured approach ensured thorough verification of all requirements through multiple test scenarios.

The experience of writing these JUnit tests helped establish a strong foundation in my test development practices. I constructed each test to be clear and focused, establishing preconditions with @BeforeEach, executing the test operation, and verifying results through assertions. For example, my testUpdateName method shows this pattern while verifying task data updates, which were successful.



TaskServiceTest.java

I maintained code efficiency by combining related operations when logical, as shown in testMultpleTasksManagement where I verified adding tasks, updating names, and deleting tasks while confirming the task list state at each step. This approach to test writing validated the code's functionality and provided documentation for future developers to understand the expected behavior of each component.

**Reflections**

In developing my testing strategy, I went beyond simple validation to focus on how the system handles errors and edge cases. I deliberately tested what happens when things go wrong - for example, ensuring the appointment system properly rejects invalid dates and carefully handles date objects to prevent accidental modifications. This careful attention to error handling helped me build a more reliable application that maintains data integrity even when users input unexpected values.

While my testing approach was thorough for basic functionality, I recognize there were more sophisticated testing techniques I could have employed. Mock-based behavioral testing would have let me simulate complex interactions between services without needing to set up complete implementations - this would have been particularly useful when testing how the appointment service interacts with the contact service. Additionally, parameterized testing could have helped me verify multiple combinations of input data more efficiently, and mutation testing would have helped evaluate how well my test suite could catch subtle code changes.

Working on this project helped me understand how different testing approaches fit various situations. For instance, the kind of error testing I focused on would be essential in a banking system where even small data errors could have serious consequences. Mock-based testing would be valuable in larger systems where services need to work together - like an e-commerce platform where the shopping cart interacts with inventory and payment systems. Parameterized testing would shine in data-heavy applications that need to process many different types of inputs correctly. Choosing the proper testing approach really depends on what kind of system you're building and what problems you're trying to prevent.

Working on this application taught me the importance of being cautious in software testing. I found this especially true when implementing the Appointment service, where date handling proved deceptively complex. When working with dates, even minor oversights can lead to significant problems. This motivated me to write comprehensive tests that verify both the integrity and immutability of date handling, as shown in my test case:



AppointmentTest.java

This careful approach extended to how I handled ID generation across the system. After initially placing the ID generator in the service class, I realized this could lead to potential issues with state management and moved it to the object classes instead. My cautious mindset paid off when testing the appointment functionality, where I needed to ensure that each appointment maintained its integrity across different operations. Taking time to understand and handle these interconnections led to more stable code that better protects against unexpected issues during runtime.

I recognized that developing and testing the same code could introduce unintentional bias, so I took several steps to reduce it. First, I reviewed coverage metrics to spot untested paths and ensured my tests included success and failure scenarios rather than just verifying the functionality I expected to work. Next, I adopted an “adversarial” mindset—actively seeking ways the code could fail—to avoid overlooking tricky edge cases. Finally, informal peer reviews provided a fresh perspective on my test suite, catching assumptions and logical gaps that might have gone unnoticed if I relied solely on my own viewpoint.

Maintaining a disciplined commitment to quality has been equally crucial throughout this project. I avoided shortcuts and quick fixes, as they often create “technical debt,” where small compromises compound into more significant maintenance problems later. By thoroughly testing each feature—especially date handling and random ID generation—I ensured functionality was robust rather than hastily patched. In practice, this meant documenting assumptions, prioritizing clarity in both code and tests and always writing or refactoring code to pass its designated tests. This disciplined approach helped me deliver a stable application that I can confidently extend or modify in the future.