Taaquan Smith  
10/19/2025  
  
  
  
  
  
  
  
For the three features in the project which were ContactService, TaskService, and AppointmentService, I used a structured unit testing approach with the JUnit 5 framework. Each service was tested individually to verify that its methods performed according to the software requirements. My focus was on testing the core functionalities, such as creating, updating, deleting, and retrieving objects. I designed tests for both valid and invalid inputs to ensure that constraints such as unique IDs, non-null values, and string length requirements were enforced.

In the ContactServiceTest class, I confirmed that adding a contact with a duplicate ID would result in an exception using the following test:

assertThrows(IllegalArgumentException.class, () -> {  
 contactService.addContact(new Contact("123", "John", "Doe", "1234567890", "123 Main St"));  
});

This test reflected the software requirement that each contact must have a unique identifier. In the TaskService tests, I verified that tasks could not be updated with invalid descriptions or names. For example, I used the following test:

assertThrows(IllegalArgumentException.class, () -> {  
 taskService.updateTaskDescription("001", "This description exceeds the allowed length limit...");  
});

These tests showed that the program correctly enforced field validation rules. For the AppointmentService, I tested that appointments could not be scheduled in the past by using:

assertThrows(IllegalArgumentException.class, () -> {  
 appointmentService.addAppointment("A01", "2020-01-01", "Old date");  
});

This ensured that all time-sensitive data followed the required business rules.

My testing approach was closely aligned with the software requirements. Each JUnit test was mapped to a specific requirement or constraint, which allowed for clear traceability between the requirements and the test cases. This alignment ensured that every requirement was validated through testing and provided strong confidence in the reliability of the system.

The overall quality of my JUnit tests was evaluated using code coverage analysis, which showed coverage above 90 percent for all three services. This high coverage demonstrated that nearly all lines and conditions in the code were executed during testing. The inclusion of negative test cases, such as those that intentionally triggered exceptions, improved the robustness of my test suite and confirmed that the program could handle edge cases and invalid inputs effectively.

Writing the JUnit tests for this project helped me understand how structured testing supports reliable software development. I learned the importance of isolating logic, using mock data when necessary, and writing clear and descriptive assertions. For example, I used the statement **assertEquals("John", contact.getFirstName**()); to verify specific expected outcomes. To ensure my code was technically sound, I followed the Arrange-Act-Assert (AAA) pattern in all tests. Each test created clean, isolated data before execution and verified the outcomes afterward to prevent interference between tests.

I also worked to make my tests efficient by reducing redundant setup code. I used the @BeforeEach annotation to initialize shared objects before each test, such as with contactService = new ContactService();. This helped maintain consistency and improved readability while reducing repetitive code throughout my test suite.

### **Reflection**

The main testing techniques I used in this project were unit testing, boundary value testing, and negative testing. Unit testing allowed me to verify that each method performed correctly in isolation. Boundary value testing confirmed that inputs at the limits of acceptable ranges were handled properly, such as testing maximum and minimum string lengths. Negative testing ensured that the system responded appropriately to invalid or unexpected inputs, such as null values or duplicate IDs.

There were several testing techniques that I did not use in this project, including integration testing, system testing, and acceptance testing. Integration testing would have verified how multiple components interact with each other, such as how the ContactService communicates with a potential database or other modules. System testing would have validated the entire mobile application workflow as one unified system, while acceptance testing would have focused on ensuring that the final product met user and business expectations. Each of these techniques plays an important role in different stages of software development. Unit testing is most useful during early development, integration testing is ideal for verifying component communication, and acceptance testing ensures that the final product aligns with stakeholder needs.

Throughout this project, I maintained a cautious and analytical mindset. I approached every piece of code as potentially flawed until proven otherwise through testing. This mindset helped me identify edge cases that might have been overlooked during development. For example, when testing the appointment feature, I considered the possibility of null or past dates that could break the scheduling logic.

It was also important to limit bias in my testing approach, especially since I was both the developer and the tester for the project. To minimize bias, I reviewed the software requirements before writing each test to ensure that I was testing the intended functionality rather than what I assumed the code should do. This helped me approach the tests objectively and focus on potential failure points rather than confirming success. Bias is a common issue when developers test their own code because it is easy to overlook mistakes in logic or assumptions. By deliberately trying to “break” my own code, I ensured a more balanced and thorough testing process.

Discipline and commitment to quality are essential traits for software engineers. Cutting corners during testing might save time in the short term, but it often results in technical debt and future rework. I learned that being thorough in writing and testing code helps prevent defects from carrying over into production. I plan to avoid technical debt by maintaining high test coverage, using automated testing tools, and conducting regular code reviews. Consistent testing practices, well-documented test cases, and adherence to best practices will allow me to produce reliable software and uphold professional standards in the field.

### **Conclusion**

This project reinforced the importance of structured, requirement-driven testing. My JUnit tests effectively verified the functionality of each service, maintained alignment with the project requirements, and achieved high levels of coverage. The testing mindset I developed through this process emphasized precision, objectivity, and a disciplined approach to software quality. These lessons will guide me as a software engineering professional, ensuring that I approach future projects with both technical and ethical responsibility.