Tania Boursiquot

CS-320 Software Test, Automation QA

October 18, 2025

7-2 Project Two

Throughout the development of the mobile application for Grand Strand Systems, I was responsible for implementing and testing three key back-end features: the contact service, task service, and appointment service. Each feature required distinct logic to meet the customer’s requirements, including data validation and storage management. This report summarizes my unit testing approach, the effectiveness and quality of my JUnit tests, and the mindset and testing techniques I employed while ensuring that each service met functional and quality expectations. It also reflects on the importance of testing discipline, bias reduction, and long-term software quality.

For the Contact Service, my unit testing approach focused on validating input constraints and ensuring data integrity. I tested that each contact included a unique ID, a name not exceeding ten characters, a properly formatted phone number, and a valid address. These test cases directly aligned with the system requirements for contact creation and modification. Each test verified both valid and invalid scenarios—for instance, testing whether the constructor correctly threw an IllegalArgumentException when a null or excessively long name was provided.  
  
In the Task Service, I applied similar principles by testing the creation and updating of tasks. The system required that each task contain a unique ID, a title no longer than twenty characters, and a description no longer than fifty characters. To ensure alignment with requirements, I created test cases that verified boundary values and handled both successful and failing conditions. For example, when testing the updateTaskDescription() method, I confirmed that overly long descriptions triggered an exception, ensuring that validation rules were enforced.  
  
For the Appointment Service, my testing approach centered on verifying correct date handling and the prevention of null or past dates. The test cases confirmed that valid appointments could be created with a unique ID and a future date while invalid inputs (such as null dates or those in the past) raised exceptions. These scenarios directly reflected the requirements outlined by the customer.

The effectiveness of my JUnit tests was measured by the coverage percentage achieved within Eclipse’s JUnit coverage tool. My combined test suites for all three services achieved coverage rates above 90%, demonstrating that the tests successfully exercised nearly all methods and conditional branches in the code. The inclusion of negative test cases, such as testing invalid input conditions, further improved reliability. High coverage alone does not guarantee quality, but in conjunction with meaningful assertions and exception checks, it indicated that the code was thoroughly validated and met both functional and structural requirements.

To ensure technical soundness, I followed clean testing practices, including the use of reusable setup methods and consistent assertions. Each test class used the @BeforeEach annotation to initialize objects, preventing redundancy and promoting isolation between test cases. For example:  
  
@BeforeEach  
void setUp() {  
   contact = new Contact("001", "Tania", "7725551234", "Port St Lucie");  
}  
I also used assertion methods such as assertEquals(), assertNotNull(), and assertThrows() to verify expected behavior. For instance:  
assertThrows(IllegalArgumentException.class, () -> contact.setPhoneNumber("123"));  
This ensured that invalid data was properly handled, protecting the integrity of the application. Clear, modular test methods improved maintainability and aligned with technical soundness standards.

Efficiency in my tests was achieved by minimizing redundancy and avoiding unnecessary object creation. I implemented helper methods to streamline repetitive tasks, improving readability and execution speed. For example, I used small utility functions to generate valid task and contact objects, which reduced boilerplate code. Efficient testing allowed faster test execution and made it easier to identify issues without adding unnecessary complexity. Overall, the test structure emphasized clarity, maintainability, and fast feedback—key principles of effective software testing.

The main techniques I employed were unit testing, boundary value analysis, and equivalence partitioning. Unit testing involved verifying each service independently, ensuring that methods worked as intended without reliance on other classes. Boundary value analysis focused on testing data limits, such as name and description lengths. Equivalence partitioning grouped valid and invalid input data to reduce redundant testing while still covering key scenarios. These approaches allowed me to test efficiently while maintaining accuracy and depth, ensuring that the services satisfied functional requirements.

Although my focus was on unit testing, there were other techniques I did not employ, such as integration testing, system testing, and regression testing. Integration testing examines the interaction between multiple modules, which would be appropriate after combining the services. System testing validates the entire system against requirements, often including user interface or API-level checks. Regression testing ensures that new changes do not introduce defects into previously working functionality. While these techniques were beyond the scope of this project, they are essential in larger systems or during later stages of development.

Each testing technique has practical implications depending on the project’s stage and scope. For instance, unit testing is most useful during early development for validating individual components, while integration testing is critical when those components begin interacting. System and regression testing are essential for ensuring stability during maintenance and release cycles. The combination of these techniques supports continuous quality assurance across the software lifecycle and helps identify issues early, reducing long-term costs.

While working on this project, I adopted a cautious and detail-oriented mindset. Software testing requires anticipating potential failures and designing tests to uncover them before users encounter them. I carefully reviewed the interrelationships among the classes—such as how the TaskService interacts with stored Task objects—to ensure that modifying one feature would not unintentionally affect another. Employing caution helped maintain code reliability and prevented cascading errors.

Testing my own code presented a natural challenge: the potential for developer bias. It is easy to assume that the code will work as intended because I wrote it. To reduce this bias, I reviewed each test from a neutral perspective, as if I were evaluating another developer’s work. I deliberately included invalid and unexpected input cases that might reveal hidden assumptions. For example, I tested null and empty string inputs even though the main code logic already prevented them. This objectivity improved the robustness of my tests.

Being disciplined in software testing is essential for maintaining long-term quality and avoiding technical debt—the accumulation of shortcuts and unfinished work that can cause future problems. I ensured that each test was meaningful, avoided redundant assertions, and properly documented. Cutting corners in testing could lead to hidden bugs, increased maintenance costs, and decreased user trust. Moving forward as a software professional, I plan to maintain this discipline by using version control, automating tests with CI/CD tools, and regularly refactoring code to uphold clean design principles. These practices will help me deliver reliable software and uphold professional integrity.

This project reinforced my understanding of how disciplined testing contributes to software reliability and maintainability. By aligning unit tests with requirements, achieving strong coverage, and practicing caution and objectivity, I ensured that each service met quality expectations. Reflecting on my approach also deepened my appreciation for the broader testing lifecycle and the importance of a professional testing mindset. As I continue developing software in the future, I will remain committed to thorough testing, limiting bias, and maintaining a consistent focus on quality to avoid technical debt and support sustainable, reliable software solutions.

Reference:

Myers, G. J., Sandler, C., & Badgett, T. (2011). The art of software testing (3rd ed.). John Wiley & Sons.  
Burnstein, I. (2003). Practical software testing: A process-oriented approach. Springer.