Project Two: Summary and Reflections  
Alex Kastigar  
Southern New Hampshire University  
CS-320-10080-M01 Software Test  
April 20, 2025

Project Two: Summary and Reflections

**Summary**

For the mobile application project, I created JUnit tests for the contact, task, and appointment services. My testing approach focused heavily on aligning to the stated software requirements. Each test targeted specific input validations, business rules, and boundary conditions outlined in the prompt. For example, in ContactTest.java, I tested for null values and exceeded character limits on the contact ID, name fields, phone number, and address. These tests supported requirements like "the phone number must be exactly 10 digits" or "the address cannot exceed 30 characters."

In the task and appointment tests, I followed a similar pattern of testing every field constraint. In AppointmentTest.java, I also added logic to reject dates in the past using appointmentDate.before(new Date()), which matched the exact requirement. Across all service test files, I validated add, update, and delete functions with both valid and invalid input. These strategies ensured that my tests directly reflected the client’s expectations.

I used Eclipse's Run with Coverage tool to verify effectiveness. All three sets of tests reached at least 80% coverage, which gave me confidence that my tests were doing their job. I even adjusted and added tests (like ID and name checks in the task service) after receiving feedback. JUnit annotations like @Test and @BeforeEach helped keep my code organized and repeatable.

I ensured technical soundness by using strong assertions and exception testing. For instance, in TaskTest.java, I used assertThrows(IllegalArgumentException.class, () -> new Task(null, "Mario", "Jumping")); to test null task IDs. In the same file, I tested boundary conditions like name lengths and empty values. These tests were efficient because they reused setup logic and stuck to a single responsibility. For example, each method in ContactServiceTest.java tested just one behavior (like updating an address) without overreaching into unrelated areas. That made it easier to spot what was broken if a test failed.

**Reflection**

**Testing Techniques Used:** I relied primarily on unit testing with JUnit 5. These tests focus on verifying individual units of behavior in isolation. The main characteristics of unit testing include fast execution, repeatability, and fine-grained error identification. By targeting each method in each class, I could isolate bugs and ensure correctness early.

**Other Testing Techniques:** I didn’t use integration testing or static testing in this project. Integration testing checks whether multiple units or components work together. It might involve seeing if the Contact and Task services share data correctly, which wasn’t required for this project. Static testing, on the other hand, would include manual code reviews or using tools to analyze the source code without running it. These techniques are valuable in larger or collaborative projects where pre-runtime issues need to be flagged.

**Practical Uses of Techniques:** Unit testing is best for early-stage development and for TDD (Test Driven Development). Integration tests are useful during system integration or just before deployment. Static tests help in regulated industries (like aerospace or healthcare) where early error prevention is critical. Together, these techniques create a layered defense against bugs.

**Caution:** While testing my code, I constantly assumed something might go wrong. That mindset kept me from assuming that "if it compiles, it works." For example, I tested what would happen if I passed a null ID into Task.java. That helped me find issues I may have otherwise missed. Recognizing how each field could break helped me approach testing more critically.

**Bias:** Writing your own tests can lead to bias because you might subconsciously avoid edge cases that could fail. I fought against that by pretending my future job depended on catching bugs in someone else’s code. I included tests for invalid input I didn’t think I'd actually enter. For example, I tested updating a nonexistent contact ID in ContactServiceTest.java, even though that’s unlikely in real usage. That’s how I pushed past bias.

**Discipline:** I learned that being disciplined means not skipping testing just because "it looks fine." Every requirement was like a checklist. If the spec said, "no nulls," I tested that. If it said, "max 10 characters," I tested 11. Being lazy here would have meant shipping buggy code. Going forward, I plan to document all requirements and map them directly to tests. That makes maintenance easier and keeps technical debt low (Stack Overflow 2024). It also builds confidence that changes won’t break things silently.

**References**

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing: An ISTQB-BCS certified tester foundation guide* (4th ed.). BCS Learning & Development Limited.

García, B. (2017). *Mastering software testing with JUnit 5: Comprehensive guide to develop high quality Java applications* (1st ed.). Packt Publishing.

Stack Overflow. (2024). *2024 Stack Overflow Developer Survey: Professional developers*. https://survey.stackoverflow.co/2024/professional-developers#2-most-common-frustrations

TestDriven.io. (n.d.). *Test-driven development*. https://testdriven.io/test-driven-development/