Shawn Plaisted

CS-320

Southern New Hampshire University

19 Apr 2025

Project 2

In Project One, I worked on building three main features for a mobile app: ContactService, TaskService, and AppointmentService. My approach to testing focused on making sure objects were created, updated, and deleted properly, and that all inputs followed the rules we were given. For each service, I wrote test cases that matched the specific requirements, helping me make sure the code did what it was supposed to.

For ContactService, I made sure that contact IDs were always unique, and that names, phone numbers, and addresses weren’t too long or left empty. I also wrote tests to check that updates and deletions worked without issues. I followed a similar process for TaskService, where I made sure that tasks had a name and a description, and that the description didn’t go over 50 characters. For AppointmentService, I tested that new appointments could be added, changed, or removed, and that the date and description fields were never left blank.

Each test I wrote was designed to reflect exactly what the software requirements asked for. I also built in validations so that if something went outside the allowed limits like a name that was too long or a missing phone number the system would throw an error. This helped make sure problems were caught early on instead of slipping through.

I think my tests were effective because they covered all the key features: adding, updating, and deleting. I tested each of those with both good and bad data to make sure everything worked correctly and failed safely when needed. I also reused objects between test cases to keep the code clean and easy to read, and I made sure to include edge cases to cover all the different scenarios that might come up later.

The testing techniques I used most were unit testing with both positive and negative test cases, plus some boundary value testing. The positive tests made sure valid data worked as expected. Negative tests were there to check how the system reacted to bad input, like null values or strings that were too long. Boundary testing helped me make sure inputs right at the limit like a description with exactly 50 characters were still handled properly.

I didn’t use integration testing or system testing in this project. Integration testing is more useful in bigger systems where different parts of the code interact, like connecting a service to a database. System testing, on the other hand, focuses on testing the entire application from a user’s perspective. Since this project was just focused on backend services, those techniques didn’t really apply here.

Each technique has its place depending on the size and purpose of the project. Unit testing is fast and focused, which makes it perfect for small services like the ones I built. Integration testing becomes important when those services need to work together or rely on external systems. System testing is best when you’re wrapping everything together and need to make sure the whole app works smoothly for the end user.

While working on this project, I really tried to think like a tester and not just a developer. Even though I wrote the code, I didn’t just assume it would work. I asked myself how it might break, and then wrote tests to try and trigger those failures. For example, I checked what would happen if I tried to create a contact with an empty name or a phone number that was too short. That mindset helped me spot a few issues I might’ve missed if I wasn’t being cautious.

To avoid bias, I separated the coding and testing parts of my workflow. Just because I wrote the method didn’t mean I trusted it completely I still wrote tests that challenged my logic. As developers, it’s easy to overlook mistakes in our own work because we’re too close to it. That’s why writing tests from a different perspective helped me stay objective and thorough.

Being consistent with testing is a big deal in software development. It’s tempting to cut corners when you're short on time, but skipping tests usually causes more problems down the road. That kind of technical debt adds up fast. I plan to avoid that by sticking to habits like test-driven development, doing regular code reviews, and using tools that help automate testing. Making quality a priority from the start is how I’ll build better software and save time in the long run.

Work Cited

JUnit Team. JUnit 5 user guide. JUnit. Retrieved April 19, 2025, from <https://junit.org/junit5/docs/current/user-guide/>​

Microsoft. Java testing | Java for beginners. Microsoft Learn. Retrieved April 19, 2025, from <https://learn.microsoft.com/en-us/shows/java-for-beginners/java-testing-java-for-beginners>