Connor Bailey

CS 320: Software Testing and Automation

Professor Lou

26 April 2025

Project Two

In this project, I crafted the contact, appointment, and task services for Grand Strand System’s client and wrote comprehensive unit tests for these services. My goal was to craft these tests closely with the software requirements, allowing me to validate the functionality of the services and ensure the service followed the requirements for the project. For example, in my *ContactServiceTest* class, I wrote tests to verify that contacts could be created, updated, and deleted correctly. In the *ContactTest* class, I validated that the Contact class meets the requirements of the software for project. Testing the *Contact* and *ContactService* classes in this way ensures that the software I created is correct, and meets the requirements given by the client. I can feel confident that my code is reliable due to the +90% test coverage across the Contact, Task, and Appointment services and base classes.

Writing the Unit tests was a meticulous, but smooth process. When constructing my tests, I used Java assertions to raise exceptions caught at runtime. These assertions ensured not just that the code ran, but that it behaved correctly across different inputs. I also made efforts to keep my tests efficient and focused by avoiding unnecessary setup or repeated logic. In the *AppointmentTest* class, for example, I created helper functions to abstract getting dates in the past and future. This keeps the tests concise and improves the readability and understanding of the tests, that way a developer doesn’t have to understand why multiple different calendar instances are declared across the tests. This approach helps to maintain a fast feedback loop during development while also ensuring that the unit tests protect the code base from regressions between software versions.

I primarily used unit testing and exception testing while working on this project. Unit testing focuses on verifying the functionality of individual components of a software project in isolation from other components. Each test was written to validate a specific behavior I wanted to validate. One example of this was in my *TaskTest* class where I validated that a task name could not be set longer than 20 characters. I also incorporated exception testing by deliberately passing invalid inputs and using assertions like the Java *assertThrows()* method to make sure the system responded as expected. These two testing techniques made sense for this project, but other testing methods like integration testing could also be incorporated to further verify the code. Integration testing involves verifying how different parts of the system work together. This would be important to test how these different services work with one another, and how these systems might work with a front-end framework that this mobile application is built on. Unit testing is most valuable in the early phases of development, as it catches bugs early on and keeps individual components reliable. Integration testing, on the other hand, become more critical in larger projects with multiple interconnected modules. By focusing on unit and exception testing for this project, I was able to deliver high-quality services quickly and efficiently. In the future, this code should undergo integration testing as it is integrated into the larger software project.

I approached my software testing with a mindset of caution and thoroughness. Small oversights or mistakes can lead to a larger problem in a software project down the road, so I had to make sure to carefully consider edge cases and unusual user interactions in my testing. When creating the *ContactService* class, I had to consider what would happen if a user tried to create a contact with a null value. Typically, we wouldn’t expect a contact to not be assigned an ID, but if an error happens elsewhere in the code (for example during Contact creation), malformed data could be saved to our data structure in memory. This malformed data then may not work with other portions of our code that rely on using this ID, and at worst could result in the corruption and loss of data. To limit bias in reviewing my own code, I tried to approach my tests as if I were an external user or reviewer. When manually testing my code, I had to think critically about what could go wrong instead of what I expected to go right. Bias can be a concern when developers test their own work because it’s easy to believe that code will behave correctly without challenging that assumption. Writing tests that deliberately tried to "break" my own code helped go against that tendency.

I believe that maintaining code through software testing is essential in software engineering. Cutting corners and writing incomplete tests or no tests at all may save time in the short term but create technical debt in the long term that is much harder and more expensive to fix. Going forward, I plan to avoid this technical debt through test driven software development, as a small investment in quality early can prevent much bigger problems down the road.