Chase Knoblock

CS-320

Project 2 – Summary and Reflections Report

19 June 2025

For the project, I implemented JUnit testing for the mobile application and the three services that were required, contact, task and appointment. I approached the project with a test and requirement-based mindset, using the specifications provided to create test cases ensuring the application behaves as expected. For example, for contact creation, I tested for valid input creation, and I tested for different types of invalid input such as null fields, empty inputs, or too long of inputs. I tested for successful updates to the different fields within a contact and ensured that those inputs themselves were tested to ensure they were within the limits set. I also tested for proper deletion of a contact and used testing to ensure that the contact being deleted exists, rather than allowing the deletion of a nonexistent contact. I used this approach with all three services and allowed for me to follow along with the requirements ensuring the application met every aspect that was expected. Each test was created to trace to a specific functional requirement.

I evaluated the effectiveness of the JUnit tests by utilizing a code-coverage tool within the Eclipse IDE. The code-coverage achieved over 90%, an industry standard, indicating that nearly all logic paths and exception paths were tested. Although a high coverage does not mean that all tests were perfect, it provides a confidence that the logic written is validated. The experience JUnit testing iterative and rewarding. I began each requirement with a basic functionality and tested inputs that were within limits and successful. When basic functionality was running properly, I began adding failure conditions, testing for those inputs that were not wanted. This experience became rewarding as the JUnit tests allowed me to find very subtle flaws such as a missed null check or testing for a task description that was longer than 50 characters.

I ensured that the code I wrote was technically sound by using industry best practices and using clear assertions. For example, I provided large comments throughout the test files separating the testing for each requirement and I used meaningful test names that provided the input being tested and the expected outcome from that input such as “testFirstName\_withNullValue\_throwIllegalArgumentException.” I utilized different exceptions such as the IllegalArgumentException or the NoSuchElementException throughout the test cases. I ensured that the code I wrote was efficient in a couple different ways. I utilized the @BeforeEach setup to avoid repetitive instantiation. I also did my best to keep the tests concise, writing as little test code as possible and keeping each test focused on a single behavior or input.

For the project, there were a few different software testing techniques that were used: unit testing, boundary testing, negative testing, and white-box testing. I used JUnit to conduct the unit testing that validated the individual methods within the different classes. Unit testing focuses on the smallest testable pieces of an application such as a method, ensuring they perform as expected, individually, before being combined with other methods. Unit testing is highly important for all projects, no matter the size of the project, as it is a foundation for test-driven development. Boundary testing was done when I tested the boundaries of expected inputs. For example, I tested the maximum boundary input for the length of IDs, which was ten characters, and I tested the phone number input to ensure that exactly ten digits was the only acceptable input. Exceptions were thrown for IDs longer than ten characters and phone numbers that were not exactly ten digits. I utilized negative testing by deliberately passing null, empty, or invalid inputs causing failures, testing and validating that the application would reject these unwanted inputs. The boundary and negative tests are practical as errors often occur when an unexpected input is provided. It is important for any application to be able to handle invalid inputs to avoid crashes and identify weaknesses. And lastly, I utilized white-box testing through the code-coverage tests that were ran. The code-coverage tests ensured that all methods were tested. White-box testing allows developers to ensure code quality and helps identify vulnerabilities, optimizing security and performance.

Some common testing techniques that were not used in this project are integration testing, performance testing, and usability and acceptance testing. Integration testing is used to examine how different components or systems of an application work together and is vital for multi-component systems or when APIs are used. Because this project focused on individual unit tests, there was no opportunity to test behaviors across different modules or systems. Performance testing focuses on the behavior of a system or application under a load, testing the speed and responsiveness. This was not required in this project but is highly important in applications where a high number of users is expected. Usability and acceptance testing utilizes users and stakeholders and tests whether the application is user-friendly and/or meets business needs. This project focused on backend logic and usability testing was not required with this backend work.

While working on this project, I put myself into a tester’s mindset and tried to adopt caution while writing the code. I tried to think of each requirement as user or a critic, trying to anticipate different things that could go wrong and what I would expect of the application when that problem occurs. For example, I asked, “What would happen if a user tried to delete a contact or a task that did not already exist?” and I created those deletion tests that could handle an input where the ID did not exist, and no such contact or task could be found. Understanding the complexity and interrelationships of the code was very important. While it may seem simple to update the information of a contact, it is crucial to understand that the updateContact method must interact with the list of contacts and could cause data to not be saved properly or the application could fail to validate the changes that were made. And to write tests for methods, a tester must understand what the method does and how the method is used by other methods or classes. Understanding interrelationships prevent regression error where new code that is written will not break code that was already written and functioning.

To limit bias in my review of the code, I used test-driven development. I would write my test cases before writing and finalizing the methods, helping me remain objective and prevented me from writing tests confirming the implementations, rather than testing and challenging the implementations. If all the logic is written before the tests, some tests, specifically edge cases, could have unintentionally been avoided. Developers always want to believe in their code, and they want the code they write to be correct and perfect. By creating the tests first, it can avoid this confirmation bias. For example, if I had thought that my logic for deleting a contact, task or appointment was perfect, I could have missed testing for an input of an ID that does not exist, an important issue that could have been missed.

Having discipline is a foundation that every developer should have. Cutting corners or taking the easy route could save some time in the present but could cost so much more later in production as bugs that are found through poor testing or lack-thereof could cause downtime, dissatisfied users and security issues such as data breaches. To avoid technical debt, I plan to follow test-driven development whenever possible, use and maintain proper documentation of the test and edge cases and conditions, and to refactor code regularly. For this project, I made sure that the codebase was clean and adaptable.