# Jordan Walker

CS-320

Kuang-Jung Huang

April 17th, 2025

**Project Two: Summary and Reflections Report**

In this project, I developed and tested three separate services for a mobile application. These included a contact service, a task service, and an appointment service. The main goal was to create code that followed specific validation rules while building a full suite of unit tests to ensure each service functioned properly. From beginning to end, the project focused on writing testable code and applying solid testing techniques to validate behavior. Each milestone had its own set of challenges, and through them, I gained a much clearer understanding of how to build reliable code and how to think like a software tester. This report reflects on the unit testing strategies I used, how I approached the design of each service, what techniques I did and did not use, and the mindset and discipline that helped me complete the project successfully.

The contact service was the first major part of the project. I started by creating a class that stored the fields required for a contact: an ID, first name, last name, phone number, and address. Each field had strict constraints. The ID had to be unique, no longer than ten characters, and it could not be null. The first name and last name also had ten-character limits, the phone number had to be exactly ten digits, and the address was limited to thirty characters. I wrote tests that confirmed each rule was enforced properly. I tried creating contacts with null values and with fields that were one character over the allowed length. I used assertions in JUnit to confirm that the constructor would throw an exception if any rule was broken. I also tested the methods that allowed updates to certain fields. While the ID had to remain unchanged, the other fields could be updated, and I wrote tests to confirm that updates were applied correctly and that invalid values were still blocked. These tests gave me a strong foundation in writing unit tests that focus on validation and structure.

After finishing the contact service, I moved on to the task service. The task class was a little simpler in terms of the number of fields, but the rules were still strict. The task ID had the same rule as the contact ID: it had to be unique, no longer than ten characters, and unchangeable after creation. The task name had to be no longer than twenty characters, and the task description could be up to fifty characters. I tested a wide range of inputs, including null names and descriptions, overly long descriptions, and names that were exactly at the boundary. I also wrote tests for the task service methods that allowed tasks to be added, updated, and deleted. These tests helped reinforce the importance of covering every possible behavior. For instance, I tested the update methods with valid new values and also with values that should be rejected. This process helped me better understand how important it is to test not just for what should work, but also for what should not.

The final service I built was the appointment service. This service was a little more complex because it introduced a new kind of data: dates. In addition to a unique appointment ID and a description, each appointment had to include a valid date. One of the rules was that the appointment date must be in the future. This required testing logic that compared the appointment date to the current date. To handle this, I used Java’s Date and Calendar classes to generate test dates. I wrote test cases that passed in a date set for tomorrow and made sure the constructor accepted it. Then I wrote tests that passed in a date set for yesterday and checked that an exception was thrown. I also tested for null values and overly long descriptions. On the service side, I created methods to add and delete appointments by ID. I used a HashMap to store the appointments for faster lookup, and I added a method that returned the stored data so I could access it during testing. This structure helped keep the class clean and testable without exposing too much of the internal implementation.

The testing techniques I relied on the most throughout the project were unit testing, boundary testing, and functional testing. Unit testing helped me isolate each part of the code to make sure it worked independently. For example, I wrote tests for each constructor to make sure object creation followed the rules. I also tested each setter method and each service method. Boundary testing helped me verify how the program handled inputs that were right at the edge of what was allowed. For instance, I tested ten-character IDs and compared them to eleven-character IDs. I did this with names, phone numbers, and descriptions too. Functional testing came in when I looked at how the services interacted with the objects. I tested adding contacts and appointments, updating them, and deleting them. I checked that these actions updated the stored data correctly and that invalid actions were blocked.

There were some testing strategies I did not use, mostly because the project did not include any outside systems. Since the project was self-contained and everything ran in memory, I did not use integration testing or user interface testing. If this had been a larger application connected to a database or user interface, those techniques would have been necessary. For example, integration testing would be important if I needed to make sure data was saved to a database and loaded correctly. UI testing would help confirm that a user interface behaved properly when interacting with the services. Although I did not use these techniques in this project, I now understand where they would fit in and how to apply them in the future.

Throughout the project, I made a strong effort to think like both a developer and a tester. As a developer, my job was to write code that worked. As a tester, my job was to make sure that code could handle incorrect input, edge cases, and unexpected behavior. I asked myself questions like what happens if someone enters a blank string, or what if the user enters a phone number with letters in it. These questions helped me write better tests and improve the quality of my code overall. I tried not to assume that the code would behave correctly just because it passed a few tests. Instead, I looked for ways to break it on purpose and then confirmed that the program responded with clear errors or safe defaults.

Another important part of this project was recognizing and avoiding bias. When writing tests, it is easy to focus on the situations you expect to work and forget about the ones that should fail. To avoid this, I made it a habit to write at least one test case for every rule that should throw an error. For example, I tested null input, input that was too long, and invalid dates. These tests helped make sure that I was not only covering the success paths, but also the failure paths that are just as important. The more I wrote, the more confident I felt that my code was truly solid and not just appearing to work by chance.

Discipline was a big part of getting through this project successfully. Testing every piece of code takes time and patience. It can be tempting to assume something works just because it looks right or because it worked once. I avoided that by taking time with each milestone to double check the rules and write clear, repeatable tests. I also made sure my test classes were well-organized, using meaningful method names and keeping everything readable. I took care to avoid hardcoding values and instead used reusable patterns that could adapt if the data changed later. Staying disciplined in this way helped me catch mistakes early and made debugging a lot easier.

Another important habit I practiced was keeping my code simple and maintainable. I avoided writing methods that tried to do too many things at once. I kept my logic focused and my classes easy to read. This helped a lot when it came time to test, since each method had a clear purpose and a predictable structure. When something failed, I could quickly tell where the issue was. It also made it easier to fix problems without creating new ones. That kind of clean design is something I want to carry forward into all of my future projects.

This project also helped me appreciate the role of testing in real software development. Before this, I sometimes thought of testing as something you do at the end, after the program is built. Now I see it as something that should be part of the process from the very beginning. Writing the tests first, or alongside the code, helped me think more clearly about what the code was supposed to do. It also helped me design my classes in a way that made testing easier. I can see now that testing is not just about checking results. It is about shaping the design of the program and helping you think more carefully about every decision you make.

Looking ahead, I know I will continue to use these techniques in both academic and professional projects. The habit of thinking about edge cases, writing tests for both success and failure, and reviewing your code from a tester’s point of view is something I plan to keep improving. As I move into larger projects with more complexity, I plan to add additional testing techniques such as integration testing, performance testing, and more advanced validation strategies. This project gave me a strong starting point, and I feel more prepared to tackle larger, more dynamic systems.

In the end, this project showed me that good code is not just about getting something to run. It is about building something that can handle real use, resist bad input, and remain stable over time. Testing gave me the tools to prove that my code worked, not just in theory, but in actual practice. It helped me catch mistakes early and made the entire development process smoother and more structured. I now see testing as an essential skill in software development, one that requires just as much focus and care as writing the code itself. The lessons I learned here will stick with me and continue to shape the way I write, test, and think about code in the future.