**Ivan Isip**

**06/17/25**

**CS320 – Summary and Reflections Report**

**Summary**

**Unit Testing Approach**

For this project, I implemented unit tests for the contact, task, and appointment services using JUnit. My approach was aligned closely with the software requirements provided in Project One. For example, in the ContactServiceTest, I verified that each contact could be added, updated (first name, last name, phone, and address), and deleted as specified. This matched the functionality outlined in the requirements. Similarly, the TaskServiceTest ensured that task name and description updates were possible while keeping the task ID immutable. The AppointmentServiceTest followed the same pattern, validating the unique ID and checking that the date was not in the past (Salem Press, 2023).

**JUnit Test Quality**

I believe my JUnit tests were effective because they provided high coverage across the core functions. All test cases covered valid and invalid inputs, edge cases (e.g., null or too-long strings), and exception handling. Based on the coverage reports, my tests achieved close to or full coverage on all methods. This gave me confidence that most of the functionality was tested and that any changes to those methods would cause a test failure, if appropriate (Coder Factory Academy, n.d.).

**Experience Writing JUnit Tests**

Writing the tests helped reinforce my understanding of each service’s logic. To ensure the code was technically sound, I wrote tests like the following:

@Test

void testUpdatePhone() {

service.addContact("001", "Ivan", "Isip", "1234567890", "123 Main St");

service.updatePhone("001", "0987654321");

assertEquals("0987654321", service.getContact("001").getPhone());

}

This test ensures that the update function correctly changes the phone number. To keep my code efficient, I reused the setup steps (like creating objects) and avoided repeating the same code. For example:

@BeforeEach

void setup() {

service = new ContactService();

service.addContact("001", "Ivan", "Isip", "1234567890", "123 Main St");

}This @BeforeEach method reduced repetition and kept the tests clean and efficient.

**Reflection**

**Testing Techniques**

In this project, I used unit testing with both black-box and white-box approaches. I didn’t rely on internal implementation knowledge alone; I also tested for expected input/output behavior. This helped ensure that even if internal changes were made later, the external behavior would still be tested.

Techniques I did not use include integration testing and system testing. Integration testing checks how multiple components work together, and system testing evaluates the complete system in a production-like environment. While those are useful in larger projects, unit testing was most appropriate for our isolated services (University of Minnesota, 2000).

Each technique fits different situations. Unit testing is great for catching bugs early during development. Integration testing is important when multiple modules interact. System testing helps before deployment to ensure the application works as a whole.

**Mindset**

While testing, I tried to approach the code like a skeptical user. I was cautious and assumed things could go wrong. This mindset helped me catch cases like null input, ID mismatches, and invalid dates (e.g., in the appointment service). Understanding how the methods connected also helped me write more complete tests. For example, I realized that if the updatePhone method didn’t first check if the contact ID existed, it could throw a null pointer error (Computer Weekly, 2002).

To limit bias, I took breaks and reviewed my tests with fresh eyes. I also thought about how someone else might try to use the classes incorrectly. If I had only trusted my own logic, I might have missed some error conditions (Leveson & Turner, 1993).

Being disciplined about quality matters a lot. It’s easy to skip a test or rush through an update, but that often leads to bugs or technical debt. To avoid that in the future, I plan to always write tests before or right after building features. I’ll also use code reviews and static analysis tools to catch problems early. For example, even when under time pressure, I commit to writing at least the basic unit tests before moving forward.

**References**

Coder Factory Academy. (n.d.). *When coding goes wrong*. Retrieved from class materials.

Computer Weekly. (2002, February 7). *Clear and present danger: Why we refused to give up*. *Gale Business: Insights*.

Gelles, D., & Kitroeff, N. (2020, September 16). *What went wrong with the Boeing 737 Max?* *The New York Times*. https://www.nytimes.com/2020/09/16/business/boeing-737-max-crashes.html

Leveson, N. G., & Turner, C. S. (1993). *An investigation of the Therac-25 accidents*. *IEEE Computer*, 26(7), 18–41.

Salem Press. (2023). *Mars Climate Orbiter and Mars Polar Lander are launched and lost*. *Salem Press Encyclopedia*.

University of Minnesota. (2000). *The explosion of the Ariane 5*. https://www-users.cse.umn.edu/~arnold/disasters/ariane5rep.html