**CS 320 Project Two: Summary and Reflections Report  
Jorge Cisneros Rodriguez  
Southern New Hampshire University**

**Summary**

I built the Contact, Task, and Appointment services with JUnit tests that focused on constructors and service methods. Each service had specific rules about field lengths, null checks, and uniqueness. My tests covered normal usage and boundary scenarios, such as 10-digit phone numbers or 50-character descriptions.

The requirements for each service were addressed through targeted checks. For instance, the Contact object enforced a maximum of 10 characters for the ID and exactly 10 digits for the phone field. Task objects had a 20-character limit for names, and Appointment objects required non-past dates. Each method in the service classes, like addContact() or addAppointment(), was tested to make sure invalid inputs produced exceptions.

I measured coverage with EclEmma and maintained over 80% for each main class. That meant constructors, setters, and specific error conditions were invoked by my tests. When coverage dipped, I added test methods for missing edge cases, like null fields or IDs that exceeded length requirements.

My experience writing the tests included separating multiple negative checks into distinct methods. Initially, combining them hampered clarity and coverage. I preferred assertThrows(IllegalArgumentException.class, ...) to confirm each constraint actually triggered an exception. An example in ContactTest was:

java

assertThrows(IllegalArgumentException.class, () -> {

new Contact("12345", "John", "Doe", "abcde12345", "123 Main St");

});

This approach let me confirm that non-numeric phones were correctly rejected. I also wrote concise tests to check performance without introducing unnecessary loops.

In ContactServiceTest, for instance:

java

ContactService service = new ContactService();

Contact contact = new Contact("12345", "John", "Doe", "5555555555", "123 Main St");

service.addContact(contact);

assertEquals(contact, service.getContact("12345"));

**Reflection**

I used unit testing, boundary testing, and coverage checks. Boundary testing included verifying exact field limits, while coverage checks pointed out potential gaps (like checking exactly 11-digit phone numbers). I did not use integration or system testing because the focus was on back-end logic, and no acceptance testing was performed. Those approaches might be necessary if I integrated external databases or confirmed real user requirements.

Approaching each test with caution allowed me to uncover issues I might have overlooked otherwise. For example, I ensured the phone was not just 10 characters, but also numeric. Writing tests for my own code risked assuming all was fine. I tried to remain skeptical by testing weird or extreme inputs. Maintaining this stance helped avoid missed corner cases.

Maintaining discipline in coverage and thorough checks reduced the chance of issues slipping into production code. Cutting corners might save short-term effort but leads to more fixes later. My goal was to keep writing tests that cover constraints one by one, so the final product is consistent with specifications and less prone to hidden flaws.

**Conclusion**

In this project, I tested the Contact, Task, and Appointment services according to each requirement, reaching over 80% coverage for the main classes. Unit and boundary testing checked lengths, null fields, and date validity. Dividing negative checks into separate tests helped me catch errors before they escalated. Writing clear and focused tests, promote coverage, and minimize long-term maintenance issues.