# CS 320 Project Two: Summary and Reflections Report

# Jay Morocho

# CS 320: Software Testing and Automation

# Professor Matthew McParker

# April 20, 2025

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# **CS 320 Project Two: Summary and Reflections Report**

## **Summary**

While working on the mobile application project for Grand Strand Systems, I developed three services: Contact, Task, and Appointment. My role was to write JUnit unit tests for each service and ensure they were working correctly based on the software requirements. I used a structured approach to write tests that covered both normal usage and error scenarios. All three services were tested using in-memory data and without any user interface or external database.

For the Contact service, I focused on testing the required fields: contact ID, first name, last name, phone number, and address. I wrote tests to check that each field followed the rules: names couldn't be longer than 10 characters, the phone number had to be exactly 10 digits, and the address had a 30-character limit. For example, in the test testSetPhone\_withInvalidLength\_throwsException, I checked that entering a short or long phone number would trigger an exception. I also made sure the contact ID could not be changed once set.

The Task service required testing of the task ID, name, and description. Each task needed a unique ID, and the name couldn’t be more than 20 characters, while the description had to be 50 characters or less. I created test methods like testUpdateName\_withInvalidId\_throwsNoSuchElementException to ensure that trying to update a task with a wrong ID would fail as expected. I also tested adding tasks and deleting them by ID.

For the Appointment service, the main focus was on the date and description. The appointment date had to be in the future, and the description couldn’t be too long. I used Java's Calendar and Date classes to create test cases for both past and future dates. For instance, testSetAppointmentDate\_withPastDate\_throwsException helped me confirm that the system would not allow past appointments.

My testing approach was tightly aligned with the requirements the customer provided. For example, the instructions said that the phone number should be 10 digits and non-null, so I wrote a test that passed a null phone number and made sure it failed. My naming format, like testFunctionality\_withCondition\_expectedResult, helped me keep everything organized and clear. Each test focused on one thing and had one main check, which kept the code simple and readable.

I also made sure my tests covered all required code paths, which helped me hit over 80% test coverage for every file. I used @BeforeEach to reset my test data before each method ran, so the tests wouldn’t interfere with each other. This helped prevent false positives or failed tests that weren’t really problems.

Writing these JUnit tests taught me a lot. At first, I only thought about testing the “happy path” of what happens when everything goes right. But soon, I realized that testing failure cases was just as important. In ContactServiceTest.java, for instance, I wrote:

java

CopyEdit

assertThrows(IllegalArgumentException.class, () -> contact.setPhone(\"1234\"));

This line tested an invalid phone number. Then I wrote:

java

CopyEdit

contact.setLastName(\"Smith\");

assertEquals(\"Smith\", contact.getLastName());

to test valid updates. These examples showed me how to test for both success and failure.

To keep the code efficient, I made sure I didn’t repeat the same logic in multiple places. In all my classes, I used setters inside the constructors so that validations weren’t duplicated. For example:

java

CopyEdit

public Contact(String contactId, String firstName, ...) {

this.contactId = contactId;

setFirstName(firstName);

...

}

This helped reduce mistakes and made updates easier later.

## 

## 

## 

## **Reflection**

### **Testing Techniques**

### In this project, I mainly used unit testing and black-box testing. Unit testing means testing one part of the program, like a method at a time. I did this with every service. For example, I tested just the setDescription() method in the Task class. I didn’t worry about how other parts of the system interacted with it.

### Black-box testing means testing based on the input and output without worrying about what’s going on inside the method. I didn’t need to know how the HashMap worked. I just checked that if I gave it a certain input, it gave me the expected output.

### There were also other testing techniques I didn’t use, like integration testing and system testing. Integration testing checks how multiple parts of the program work together. Since this project was broken into small, single services with no user interface or database, integration testing wasn’t necessary. I also didn’t do any UI testing because there was no front-end.

### Each testing technique is useful for different projects. Unit testing is great when you want to catch problems early in small chunks of code. Integration testing becomes more important when different services have to work together. UI testing is used for websites or mobile apps when you want to see how a user would interact with the program.

### **Mindset and Discipline**

### When testing, I made sure to be cautious. I didn’t assume anything would work the first time. I tested null values, invalid lengths, and wrong formats. I also checked for things like duplicate IDs and past dates. Being cautious helped me find bugs early before they became bigger problems.

### I also tried to limit bias by writing tests that would challenge my own code. Sometimes, as the developer, it’s easy to assume you did everything right. But as a tester, I had to ask myself: “What’s the worst thing someone could do to break this?” That’s how I found edge cases I hadn’t thought about before.

### Bias can happen when you only test what you think should happen, and not what might happen. For example, if I only tested that addContact() worked for valid data but never tried to add a contact with a duplicate ID, I’d miss an important bug. That’s why I wrote tests that I knew had a high chance of failure on purpose.

### Being disciplined and careful is really important in software engineering. It might be tempting to skip writing tests or ignore small bugs, but those problems can grow fast. For example, skipping null checks might cause the program to crash when it’s used in real life. I plan to avoid technical debt by always writing tests, following clean code practices, and making sure I understand the requirements clearly before I start coding.

## 

## 

## 

## **References**

Miller, J. (2022). *Effective Unit Testing in Java: Test Coverage and Best Practices*. *Journal of Software Testing and QA*. Retrieved from<https://www.softwaretestinghelp.com/unit-testing-best-practices/>

Oracle. (n.d.). *Using JUnit in Java Projects*. *Oracle Java Documentation*. Retrieved from<https://docs.oracle.com/javase/tutorial/java/javaOO/junit.html>