Jordan Bankston  
CS-320-11221-M01  
April 26, 2025

Professor Luo

## 1. Summary

### a. Unit Testing Approach

For Project One, I developed unit tests focusing on three main features: the `addContact`, `updateContact`, and `deleteContact` methods inside the `ContactService` class. My approach to unit testing was to create a variety of test cases that not only validated normal functionality but also intentionally challenged the boundaries of the specifications.

#### i. Alignment to Software Requirements

I ensured that each unit test was directly aligned to the project's requirements. For example, the `addContact` method was tested to ensure that a contact with a valid first name, last name, phone number, and address could be successfully added. Additionally, I created tests that attempted to add a contact with missing or invalid fields, such as an empty first name, to confirm that exceptions were thrown as expected.

Examples like “Assertions.assertThrows(IllegalArgumentException.class, () -> {

contactService.addContact("", "Smith", "1234567890", "123 Main St");

});”

#### ii. Overall Quality and Effectiveness

I defended the quality of my JUnit tests by targeting a high coverage percentage, aiming to test every possible outcome of each method. Instead of merely achieving success cases, I emphasized boundary testing and error handling. My test coverage included validating normal inputs, handling null or empty inputs, and checking size changes in the `contactList` after deletions.

### b. Experience Writing JUnit Tests

#### i. Ensuring Technically Sound Code

To ensure technical soundness, I wrote tests that validated both the expected results and the internal state changes. For instance, after deleting a contact, I not only verified the list size but also looped through the list to confirm the specific contact ID was no longer present.

“assertEquals(2, contactService.contactList.size());

boolean exists = contactService.contactList.stream()

.anyMatch(contact -> contact.getContactID().equals("1"));

assertFalse(exists);”

#### ii. Ensuring Code Efficiency

Efficiency was emphasized by using structures like `HashMap` where appropriate. For example, in the `TaskService` class, I used a `HashMap` instead of a `List` to optimize search and delete operations. This allowed constant-time access to tasks by ID, making operations far more efficient than linear searches.

“if(tasks.containsKey(taskID)) {

tasks.remove(taskID);

}”

## 2. Reflection

### a. Testing Techniques

#### i. Software Testing Techniques Used

In this project, I primarily used unit testing and static testing. Unit testing involved writing JUnit tests to verify that individual methods behaved according to the specifications. Static testing involved reviewing the source code manually to catch logic errors and ensure compliance with the requirement documents.

#### ii. Other Testing Techniques Not Used

I did not employ integration testing or automated build testing. Integration testing would have validated interactions between the `ContactService`, `TaskService`, and `AppointmentService` classes. Automated testing would have involved setting up continuous test execution after each code commit.

#### iii. Practical Uses and Implications

Unit testing is practical for early error detection and ensuring individual components are reliable. Static testing helps ensure code quality before runtime. Integration and automated tests are crucial in larger systems to catch issues between modules and maintain consistent quality in a continuous integration pipeline.

### b. Mindset

#### i. Employing Caution

I approached testing with a mindset of skepticism, trying to "break" my own code by thinking like a malicious user. Recognizing the complexity of the software helped me avoid oversights.

#### ii. Limiting Bias

To limit personal bias, I assumed that every function I wrote was flawed until proven otherwise through testing. Running extensive unit tests kept me honest and thorough.

#### iii. Importance of Discipline

Discipline in testing is vital to prevent technical debt from accumulating. Cutting corners by skipping test cases or ignoring failed tests can introduce bugs that are costly to fix later. By rigorously validating each method, I built stronger confidence in my code's quality. Moving forward, I plan to integrate more automated testing and eventually add security scans to prevent vulnerabilities.

## Conclusion

Overall, the unit testing strategies I applied helped solidify the reliability of the mobile application features I developed. By using a combination of unit and static testing, maintaining discipline, and focusing on the true intent behind each requirement, I was able to create a well-tested and efficient solution for Project One.