Michael Kibler

CS 320 Software Test, Automation QA

20 April 2025

Project Two: Summary and Reflections Report

**1) Summary**

a) *Unit Testing Approach and Alignment to Requirements*

For project one, I implemented JUnit tests for each of the contact, task, and appointment service classes. I began with a close analysis of the software requirements to ensure each test case matched a specified function. For instance, in the ContactService class, I ensured that tests covered all required behaviors such as creating, adding, deleting, and updating contact fields like name, phone, and address.

In the TaskService package, I created unit tests that validated the ability to create new tasks with unique identifiers, prevent duplicate IDs, and accurately update task details. I also tested the AppointmentService class for a unique appointment ID creation, proper date formatting, and conflict management.

I made sure that my implementation aligned with the requirements of the project. As an example, in the ContactServiceTest.java file, I verified that duplicate IDs cannot be added to the system, enforcing the uniqueness of the ContactID:

Assertions.assertThrows(IllegalArgumentException.class, () -> {

service.addContact(contact2);

});

I measured effectiveness through code coverage metrics. Each of the three services had test coverage of at least 80%, which aligns with industry standards for unit testing. All of the core logic branches for the program were tested, reducing the risk of defects.

b) *Experience Writing JUnit Tests*

Each test followed the Arrange-Act-Assert (AAA) pattern for clarity and maintainability. As an example, in TaskServiceTest.java:

*Arrange:*

Task task = new Task("task7", "Task Name", "Task Description");

service.addTask(task);

*Act:*

Task returned = service.getTask("task7");

*Assert:*

assertTrue(returned.getTaskID().equals("task7"));

assertTrue(returned.getName().equals("Task Name"));

assertTrue(returned.getDescription().equals("Task Description"));

I have updated my project’s code to ensure efficiency by removing redundant setup code and reusing objects via the @BeforeEach annotation. As an example, the AppointmentServiceTest.java class now has:

AppointmentService service;

Appointment appointment;

@BeforeEach

void setup() {

appointmentService = new AppointmentService();

appointment = new Appointment("1", LocalDate.now());

}

This setup eliminated repeated instantiations across tests, enhancing code efficiency, maintainability and readability.

**2) Reflection**

a) *Testing Techniques*

The primary techniques I used were black-box testing and equivalence partitioning. Black-box testing allowed me to focus on expected input/output behavior without needing internal code structure knowledge. For example, I tested whether an appointment date in the past would throw an exception:

Assertions.assertThrows(IllegalArgumentException.class, () -> {

new Appointment("app2", new Date(120, 4, 6), "Description of app2");

});

Equivalence partitioning helped reduce redundant tests. As an example, I tested phone numbers by checking only one valid and one invalid format, while assuming the behavior is uniform across similar inputs.

White-box testing techniques such as statement and decision coverage were not directly applied, but coverage reports showed that most logical paths were executed. Techniques like state transition testing or mutation testing were also not utilized in my project. I believe these are better suited for complex systems with stateful interactions or safety-critical domains.

In different contexts, state transition testing would be useful for a UI login system, while mutation testing would benefit a security-sensitive application to check the resilience against altered logic.

b) *Mindset*

Throughout the project, I tried my best to keep a cautious and skeptical mindset. I did not assume my code would work. I assumed it could break and designed tests to prove otherwise. One specific example was in AppointmentServiceTest.java, where I tested creating an appointment with a null date to validate robustness:

Assertions.assertThrows(IllegalArgumentException.class, () -> {

new Appointment("app3", null, "Description of appointment3");

});

Appreciating the interrelationship between components was key here. An update to the Task class could cause problems throughout the TaskService class, so I reran all related tests whenever a change was made. This ensured stability and protected against any sort of regression errors.

To limit bias in code review, I wrote tests before finalizing the implementation, which portrays a test-driven development mindset. This prevented me from unconsciously ignoring potential edge cases. Bias is a real concern when developers test their own code. A developer might unconsciously test only the path they expect to pass. As an example, without deliberate intention, I may have neglected testing duplicate contact IDs if I assumed it “wouldn’t happen.” By recognizing this I forced myself to test all of the edge cases and invalid inputs.

Being disciplined about quality is critical when writing code. Cutting corners may save time in the now, but it leads to technical debt and future headaches. For example, skipping null checks or not testing error conditions can lead to runtime exceptions that are hard to debug later down the line.

To avoid technical debt, I plan to continue using automated test suites and code coverage tools. I will also apply code reviews in an attempt to catch style or logic issues early. As a software engineer, being committed to writing clean, testable code needs to be a main priority.

**References**

Garcia, B. (2017). *Mastering Software Testing with JUnit 5: Comprehensive Guide to Develop High Quality Java Applications*. Packt Publishing.

Hambling, B., et al. (2019). *Software Testing: An ISTQB-BCS Certified Tester Foundation Guide (4th Edition)*. BCS Learning & Development Limited.