**Summary and Reflections Report**

Jason Hney

CS 320

Software Test, Automation

02/22/2025

**Summary and Reflections Report**

**Summary**

**Unit Testing Approach**

For the mobile application project, I implemented **unit testing using JUnit** for the three core features: **contact management, task management, and appointment scheduling**. Each feature was tested individually to ensure proper functionality and adherence to software requirements.

* **Contact Management**: Tested for proper creation, modification, and deletion of contacts. Used assertions to check data integrity.
* **Task Management**: Verified task addition, status updates, and deletion. Ensured constraints like unique task IDs were met.
* **Appointment Scheduling**: Focused on date validation, conflicts, and correct retrieval of scheduled appointments.

**Alignment with Software Requirements**

My approach was closely aligned with the software requirements as each test case was derived from the functional specifications. For instance, the requirement that “contacts must have unique IDs and cannot exceed a certain character limit” was tested using:

@Test

public void testInvalidContactName() {

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

new Contact("123", "NameExceedingMaxLength", "Last", "1234567890");

});

assertEquals("Name exceeds character limit", exception.getMessage());

}

This test confirms that the validation constraints outlined in the requirements were enforced.

**JUnit Test Effectiveness and Coverage**

To evaluate the effectiveness of my tests, I relied on **code coverage metrics**. My test suite achieved **over 90% statement coverage**, indicating that the majority of code paths were executed during testing. The high coverage percentage suggests that edge cases and exceptions were adequately tested.

A specific example of achieving good coverage is:

@Test

public void testDeleteNonexistentTask() {

TaskManager manager = new TaskManager();

assertFalse(manager.deleteTask("nonexistentID"));

}

This ensures the software gracefully handles invalid deletions.

**Experience Writing JUnit Tests**

Developing the JUnit tests required careful planning to ensure both **technical soundness and efficiency**.

* **Ensuring Technical Soundness**: Used **boundary testing and exception handling** to cover all possible input cases.
* **Code Efficiency**: Avoided redundant tests and utilized parameterized testing where applicable. For example:

@ParameterizedTest

@CsvSource({

"task1, true",

"task2, false"

})

public void testTaskCompletionStatus(String taskId, boolean expected) {

Task task = new Task(taskId, "Sample Task");

task.setCompleted(expected);

assertEquals(expected, task.isCompleted());

}

This structure reduces code duplication while testing multiple conditions in one method.

**Reflection**

**Software Testing Techniques Employed**

1. **Black-box testing**: Focused on input/output behavior without inspecting internal implementation.
2. **White-box testing**: Used statement and branch coverage to test the internal logic.
3. **Boundary value testing**: Checked limits, such as maximum character lengths and numerical constraints.

**Other Testing Techniques**

1. **Integration Testing** (Not Used): Ensures multiple modules work together, but since this was unit testing, it was not within scope.
2. **Regression Testing** (Not Used): Would be beneficial for future iterations but was not a primary focus.
3. **Mutation Testing** (Not Used): Helps check test robustness by making small changes to code, though not applied here.

Each technique is useful in different scenarios. **Integration testing** would be critical for checking service interactions in a microservices-based application. **Regression testing** would help detect unintended side effects after code modifications.

**Mindset and Caution in Testing**

While testing, I maintained a **cautious and detail-oriented mindset** to anticipate possible errors. Recognizing the **interrelationships between features** was crucial. For example, modifying the **contact deletion function** also impacted **task assignments**, so I had to ensure task dependencies were handled properly:

@Test

public void testContactDeletionWithTasks() {

Contact contact = new Contact("001", "John", "Doe", "1234567890");

Task task = new Task("t001", "Linked Task", contact.getId());

contactManager.deleteContact("001");

assertNull(taskManager.getTask("t001"));

}

This ensured that deleting a contact would also handle assigned tasks correctly.

**Limiting Bias in Code Reviews**

To reduce bias, I **reviewed the tests from a third-person perspective**, focusing on **negative test cases** rather than assuming correctness. If a developer is testing their own code, they might unconsciously avoid **edge cases** that break functionality.

Example: Initially, I only tested valid task names, assuming users would follow the expected format. However, I later added:

@Test

public void testEmptyTaskName() {

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

new Task("t002", "");

});

assertEquals("Task name cannot be empty", exception.getMessage());

}

This mindset shift helped catch potential errors that could be overlooked due to developer bias.

**Commitment to Quality and Avoiding Technical Debt**

Being disciplined in writing and testing code is crucial to avoid **technical debt**. Cutting corners, such as skipping test cases for minor features, could lead to **cumulative failures** over time.

To prevent technical debt, I follow these strategies:

* **Automated Testing**: Running tests regularly prevents regressions.
* **Code Reviews**: Peer reviews help identify overlooked issues.
* **Documentation**: Clearly documenting test cases ensures maintainability.

For instance, I initially ignored **edge case testing for leap years** in the appointment scheduling module. Later, I added:

@Test

public void testLeapYearAppointment() {

LocalDate date = LocalDate.of(2024, 2, 29);

Appointment appt = new Appointment("a001", date, "Doctor Visit");

assertEquals("2024-02-29", appt.getDate().toString());

}

This test ensures the system correctly handles date validations, reducing future maintenance issues.

**Conclusion**

Through this project, I applied **structured unit testing techniques**, maintained a **quality-first approach**, and learned the importance of **caution, discipline, and limiting bias** when writing tests. Moving forward, I will incorporate **integration testing and automation** to further improve testing efficiency and reliability.