Summary and Reflections Report

Summary

Unit Testing Approach

For this project, I developed and tested three core services for a mobile application: ContactService, TaskService, and AppointmentService. Each class featured corresponding test classes designed to verify functionality using unit testing via JUnit 5.

AppointmentService testing emphasized validation constraints. For instance, appointment IDs were limited to 10 characters, dates had to be in the future, and descriptions could not exceed 50 characters. My approach focused on verifying these constraints using assertThrows() and checking successful object creation using assertEquals() and assertNotNull(). This was closely aligned with the software requirements, which mandated robust validation and error-handling.

ContactService testing focused on verifying CRUD functionality with valid and invalid input data. Tests validated phone number length, ZIP code format, and null checks for names. Similarly, TaskService testing ensured unique task IDs, deadlines in the future, and appropriate constraints on task descriptions. In all cases, testing methods adhered to specified requirements, ensuring correctness and reliability.

Effectiveness of JUnit Tests

Test effectiveness was measured using coverage reports from JaCoCo. I ensured each method and edge case had at least one associated test. All core methods in the service classes had corresponding positive and negative tests, including tests for exceptions and boundary values. For example:

- AppointmentServiceTest included both valid appointments and failed validations like past dates.
- ContactServiceTest checked invalid input for phone numbers (too long or null), ensuring those exceptions were thrown correctly.

The coverage report reflected over 80% coverage for all services, indicating high test effectiveness.

Experience Writing JUnit Tests

Ensuring Technically Sound Code

To ensure soundness, I applied clear test naming conventions (e.g., testInvalidPhoneNumberThrowsException) and arranged each test using the AAA (Arrange, Act, Assert) pattern. For example:

```
@Test
public void testInvalidId() {
   Date futureDate = getFutureDate(1);
   assertThrows(IllegalArgumentException.class, () -> new Appointment(null,
```

```
futureDate, "Meeting"));
}
```

This code confirms that null IDs are correctly rejected.

In ContactTest, this logic appeared in:

```
@Test
public void testInvalidPhone() {
   assertThrows(IllegalArgumentException.class, () -> new Contact("1", "John", "Doe",
"12345678901", "12345"));
}
```

This ensured technical correctness through negative testing.

Ensuring Efficient Code

Efficiency came from isolating responsibilities and minimizing duplication. Helper methods like getFutureDate(daysAhead) allowed reusable date generation. For instance, AppointmentTest reused a method for generating valid future dates, rather than duplicating logic in each test:

```
private Date getFutureDate(int daysAhead) {
   Calendar cal = Calendar.getInstance();
   cal.add(Calendar.DAY_OF_YEAR, daysAhead);
   return cal.getTime();
}
```

Additionally, tests only focused on one assertion per test to make debugging easier and execution faster.

Reflection

Testing Techniques

I employed the following techniques:

1. Black-box testing - I validated service class behavior without relying on their internal implementation. This was seen in tests like testAddAppointment or testDeleteContact.
2. Boundary value analysis - I tested edge inputs like 10-character IDs, 50-character descriptions, and invalid 11-character phone numbers.
3. Exception testing - Tests confirmed that invalid data triggered IllegalArgumentException, ensuring validation enforcement.

Techniques I did not employ include:

- Mocking/stubbing - Not used as these were in-memory tests without external dependencies.
- Integration testing - My tests were purely unit-level, not covering multiple services together.

Practical Use Cases:

- Black-box testing fits most CRUD-based services, like backend APIs or service layers.

- Boundary value testing is crucial in user-input-heavy systems, like form validators or parsers.
- Exception testing is valuable when strict validation rules must be enforced.
- Mocking becomes important in service-layer testing with database or network dependencies.

Mindset

Caution

I approached testing with caution by treating each input constraint as critical. For instance, in TaskTest, I wrote cases to fail if the due date was null or in the past. This ensured high code reliability and safeguarded future maintenance.

```
@Test
public void testInvalidDueDate() {
   assertThrows(IllegalArgumentException.class, () -> new Task("123", "Title",
"Description", null));
}
```

Appreciating complexity was key. Even simple services had subtle interactions between fields (For example, a task must have a unique ID and valid date). Recognizing that interdependencies exist helped prevent silent bugs.

Bias

I reduced bias by testing invalid and edge cases as rigorously as valid ones. It was tempting to only test the "happy path" where object creation succeeds. Instead, I treated each input as potentially harmful and wrote tests to ensure incorrect values failed reliably.

As a developer testing my own code, bias could creep in if I assumed my implementation was correct. Writing fail-first tests helped keep my focus on objectivity.

Discipline

I remained disciplined by keeping test coverage above 80% and not skipping validation checks even if time was limited. For example, I ensured ZIP code validation tests were present even though they were relatively simple.

Avoiding technical debt meant documenting utility methods like getFutureDate and isolating test logic by feature. By doing so, I ensured future developers could extend tests easily. I also resisted commenting out tests that initially failed as each failing test was a sign of a problem to fix, not avoid.

Conclusion

This project strengthened my understanding of JUnit testing, disciplined development, and defensive programming. I learned the value of methodical, thorough testing in validating even the simplest service logic. By focusing on sound, efficient, and requirement-aligned unit tests, I ensured reliable behavior for all core service features of the mobile application. Going forward, I will carry this mindset into all future software engineering projects, emphasizing test coverage, validation, and quality assurance as first-class concerns.

References

GarcíaB. (2017). *Mastering software testing with JUnit 5 : comprehensive guide to develop high quality Java applications*. Packt Publishing.

Hambling, B., & Al, E. (2015). *Software testing : an ISTQB-BCS certified tester foundation guide*. Bcs.

Jakubiak, N. (2018, August 2). *JUnit Tutorial With Examples: Setting Up, Writing, and Running Java Unit Tests*. Parasoft. https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/