Summary and Reflections Report

Unit Testing Approach

When working on unit testing for the **Contact, Task, and Appointment features**, I took a structured approach using JUnit. My strategy involved validating key constraints and functionalities:

- **Contact Service Tests:** Ensured that contacts were created correctly, that contact IDs were unique, and that name and phone number length constraints were enforced.
- **Task Service Tests:** Tested task creation, updates, and deletion, while ensuring unique task IDs and verifying that descriptions did not exceed the allowed character limit.
- **Appointment Service Tests:** Confirmed that appointments could be scheduled correctly, that appointment IDs remained unique, and that past-date scheduling was restricted.

Test Coverage & Maximization Strategy

To ensure **maximum test coverage**, I measured test execution using **Jacoco**, which reported an overall **92% test coverage**. This coverage was achieved by:

1. **Testing all critical paths** – verifying both valid and invalid inputs.
2. **Using parameterized tests** to check multiple cases efficiently.
3. **Employing edge-case testing** to validate input limits and constraints.

For example, I used **boundary value analysis** to ensure that task descriptions did not exceed **50 characters** and appointment dates were not scheduled in the past.

Alignment with Software Requirements

I ensured that my unit tests aligned with **software requirements** by covering key constraints and expected behaviors.
For example, in ContactServiceTest.java, I tested that an **exception was thrown** if a **phone number exceeded 10 digits**, ensuring adherence to the requirements.

```
java
CopyEdit
@Test
void testPhoneNumberTooLong() {
   assertThrows(IllegalArgumentException.class, () -> {
      new Contact("C001", "John Doe", "12345678901", "123 Street");
   });
}
```

Similarly, in TaskServiceTest.java, I validated that **task descriptions did not exceed 50 characters**, preventing invalid input from entering the system.

JUnit Test Quality and Effectiveness

I measured the **effectiveness of my JUnit tests** through **assertions and code coverage reports**.

- I used assertions such as **assertEquals**, **assertNotNull**, and **assertThrows** to validate expected behavior.
- By executing tests with a code coverage tool, I confirmed that my test suite covered all major functionalities, ensuring robustness.

Experience Writing JUnit Tests

Writing JUnit tests was an insightful experience. One of the biggest challenges was ensuring that **negative test cases** were properly handled. Debugging failures helped me refine the tests and improve overall code robustness.
For instance, I initially assumed that all inputs would be valid, but testing **invalid and edge cases** revealed **potential system weaknesses** that required fixes.

Code Soundness and Efficiency

Ensuring Technical Soundness

I ensured that my code was **technically sound** by incorporating **clear and meaningful assertions**.
For example, in ContactServiceTest.java, I used:

java
CopyEdit
assertThrows(IllegalArgumentException.class, () ->
contactService.addContact(duplicateContact));

This confirmed that **duplicate contact IDs were not allowed**.

Efficiency in Code Design

Efficiency was demonstrated by **reducing redundant code** and **optimizing function reuse**.
For example, instead of writing **multiple validation methods**, I created a single method that could validate **multiple input fields**, improving code **maintainability and performance**:

java
CopyEdit
private boolean validateInput(String item, int length) {
  return (item != null && item.length() <= length);
}

This method allows validation of **ID, Name, Address, Phone Number, and Task Description**, avoiding unnecessary duplication.

Reflections

Testing Techniques Used

The primary **testing techniques** I used were:

- **Unit Testing** – for verifying individual functionalities.
- **Boundary Testing** – to validate **input constraints** like **max character limits**.
- **Exception Testing** – to ensure **invalid inputs** were properly handled.

Techniques Not Used & Justification

I did **not use integration testing** or **performance testing** in this project.

- **Integration Testing** focuses on **interactions between components** but was not required since I was testing **individual services**.
- **Performance Testing** was not included since the system **does not handle high-load scenarios**.

Performance Testing: Why It Matters

Performance testing ensures that a system performs optimally under **expected workloads**.
For example, a scheduling system must handle **thousands of concurrent users** without lag.
Performance testing evaluates:

1. **Speed:** Response time under **expected and peak loads**.
2. **Scalability:** How well the system **handles increased users or data**.
3. **Stability:** System behavior under **continuous use**.

Example of a performance test in JUnit:

java
CopyEdit

```
@Test
@Timeout(1) // Ensures method execution completes within 1 second
void testAppointmentServicePerformance() {
   for (int i = 0; i < 1000; i++) {
      appointmentService.addAppointment(new Appointment("A" + i, new Date()));
   }
}
```

Practical Implications of Testing Techniques

Each technique is useful in different situations:

- **Unit Testing** is ideal for **early-stage development** to prevent errors.
- **Integration Testing** is critical when multiple services **communicate with each other**.
- **Performance Testing** is necessary when software must **handle heavy loads efficiently**.

Mindset and Bias Prevention in Testing

To minimize **bias**, I varied test data to **simulate real-world conditions**.

Example of Bias Prevention

If I expected **task descriptions to always be valid**, I might **neglect testing excessively long descriptions**.
To counteract this bias, I created a **negative test case**:

```
java
CopyEdit
@Test
void testTaskDescriptionTooLong() {
   assertThrows(IllegalArgumentException.class, () -> {
      new Task("T001", "This is a very long description that exceeds fifty characters.");
   });
}
```

This ensured **unexpected user inputs** were **handled properly**.

Commitment to Quality and Discipline in Testing

I maintained **high-quality standards** by:

- **Using automated testing tools** for **continuous validation**.
- **Conducting code reviews** to **catch logical errors**.
- **Maintaining clear documentation** to ensure future developers can **understand test cases**.

Example of Disciplined Testing Approach

I ensured **repeated test setups were optimized** using @BeforeEach:

java
CopyEdit
```
@BeforeEach
void setUp() {
    contactService = new ContactService();
}
```

References

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing: An ISTQB-BCS certified tester foundation guide* (4th ed.). BCS Learning & Development Limited.

García, B. (2017). *Mastering software testing with JUnit 5: Comprehensive guide to develop high-quality Java applications.* Packt Publishing.