## Summary

Unit Testing Approach

For Project One at Grand Strand Systems, my role was to implement and test the Contact, Task, and Appointment services for a mobile application. To ensure reliability, I employed JUnit testing to validate core functionalities, enforce data constraints, and assess system behavior under varying conditions.

Alignment to Software Requirements

Each test was designed to align directly with the project’s functional requirements:

- Appointments: Enforced a rule that prevented scheduling appointments in the past.

- Contacts: Verified that phone numbers adhered to the 10 digit requirement, names met length constraints, and addresses were within the expected range.

- Tasks: Ensured tasks were correctly instantiated with appropriate identifiers and descriptions.

These tests mirrored the software specifications, ensuring that improper data entry was systematically rejected.

Effectiveness of JUnit Tests

The effectiveness of my JUnit tests was measured by achieving over 80% code coverage, ensuring that:

-Each feature was rigorously tested through positive and negative test cases.

-Assertions like `assertEquals()`, `assertNotNull()`, and `assertThrows()` were employed to validate correct behavior and exception handling.

-Boundary conditions were examined to prevent unintended input acceptance.

For example, to validate phone number constraints in `ContactTest.java`:

@Test

void testInvalidPhoneNumber() {

assertThrows(IllegalArgumentException.class, () -> {

new Contact("1234567890", "John", "Doe", "12345", "123 Main St");

});

}

This ensured invalid numbers were immediately rejected, maintaining data integrity.

Experience Writing JUnit Tests

Ensuring Technically Sound Code

Each test case was designed to verify both valid inputs and edge cases. For example, appointment scheduling tests included conditions where:

- A valid appointment could be created successfully.

- Invalid dates (past appointments) triggered an exception.

Code Efficiency

To enhance efficiency, I optimized tests by:

-Minimizing redundant code by using reusable test data.

-Employing exception handling to reduce unnecessary computations.

-Eliminating redundant instantiations, conserving memory usage.

Example from `AppointmentTest.java`:

Date futureDate = new Date(System.currentTimeMillis() + 100000);

Appointment appointment = new Appointment("1234567890", futureDate, "Doctor Visit");

assertNotNull(appointment);

This prevents inefficient memory allocation by ensuring objects are created only when necessary.

## Reflection

Testing Techniques

I employed multiple testing techniques to ensure reliability:

-Unit Testing: Each feature was tested in isolation to confirm correct behavior.

-Black Box Testing: Tests focused solely on expected input/output behavior rather than internal logic.

-Boundary Testing: Examined edge cases (e.g., maximum/minimum string lengths, invalid dates).

Techniques not used but valuable for future iterations include:

-Integration Testing: Would test interactions between different components.

-System Testing: Would validate the entire application workflow.

-Regression Testing: Would ensure that changes did not break existing functionality.

Each technique plays a crucial role in the software development lifecycle. For example, regression testing is essential when releasing updates to prevent unintended breakages.

Mindset Reflection

Caution in Software Testing

Being meticulous in testing was essential to catch potential issues before deployment. By focusing on boundary conditions and negative test cases, I ensured that invalid inputs were handled gracefully rather than causing unexpected system failures.

Bias in Testing

Bias is a natural challenge in self testing. To mitigate this, I:

-Relied on well defined software requirements rather than assumptions.

-Tested unexpected edge cases rather than just confirming expected functionality.

-Ensured exceptions were actively validated rather than assuming correct execution.

Discipline in Software Development

Maintaining discipline in testing is critical to avoiding technical debt. Cutting corners in validation can lead to significant maintenance costs down the line. I addressed this by:

-Strictly adhering to error handling principles.

-Writing clear, reusable tests to ensure maintainability.

-Prioritizing correctness over speed, ensuring tests covered real-world scenarios.

## Conclusion

The JUnit tests for Project One ensured that each feature adhered to strict validation rules, preventing invalid data from corrupting the system. Future improvements could include:

-Expanding negative test cases for more robust validation.

-Incorporating integration testing to confirm inter-service communication.

-Automating test execution for continuous validation in future iterations.

With rigorous testing in place, the system remains reliable, efficient, and scalable for real world deployment.

## References

-JUnit. (n.d.). JUnit 5 user guide. JUnit. Retrieved from https://junit.org/junit5/docs/current/user-guide/

-Institute of Electrical and Electronics Engineers. (2008). IEEE standard for software and system test documentation (IEEE Std 829-2008). IEEE. Retrieved from https://standards.ieee.org/standard/829-2008.html