**Southern New Hampshire University**

**Bryan Howard**

**CS-320-15800-M01 Software Test, Automation QA 2025**

**Unit Testing Approach for Each Feature**

To ensure software reliability and correctness, unit testing was an integral part of the development process. For the contact service, unit tests were designed to verify the creation, updating, and deletion of contact entries while ensuring data consistency. The primary requirements mandated that each contact must have a unique ID, a non-null name, and a valid phone number. JUnit tests were structured to validate these constraints by implementing assertions that checked invalid phone numbers, null values, and boundary cases.

Similarly, for the task service, tests were created to validate task creation, updates, and status changes. Since the requirement specified that tasks must have descriptions under 50 characters, boundary testing was conducted to ensure compliance. An example of this approach included writing test cases that deliberately exceeded this limit, ensuring that the system correctly rejected invalid input.

For the appointment service, the focus was primarily on date validation. The requirement that all appointments must be scheduled for future dates necessitated the implementation of tests to reject past dates while accepting valid ones. These tests played a crucial role in preventing erroneous or outdated appointments from being added to the system.

**Effectiveness and Quality of JUnit Tests**

The effectiveness of the JUnit tests was assessed based on code coverage, reliability, and adherence to best practices. The test coverage achieved was over 85%, ensuring that most major functionalities and edge cases were properly validated. Assertions such as assertEquals, assertThrows, and assertNotNull were extensively used to confirm expected behaviors and error handling. Additionally, Mockito was employed to simulate dependencies, allowing unit tests to be executed in isolation without unnecessary external dependencies interfering with results.

To further ensure quality, tests were incorporated into CI/CD pipelines, enabling automated validation during development cycles. This approach helped to prevent regressions and maintain software stability as new features and updates were introduced. By applying a structured testing approach, defects were identified early, reducing debugging time and improving the overall development workflow.

**Experience Writing JUnit Tests**

Writing JUnit tests for this project was a valuable learning experience that reinforced the importance of designing testable and maintainable code. One of the primary challenges faced was managing exception handling efficiently. To address this, dedicated test cases were implemented to verify that invalid inputs, such as null values or incorrect data formats, triggered appropriate exceptions.

As the testing process evolved, improvements were made to increase efficiency. Initially, some tests were redundant and overly verbose, but refactoring efforts led to the introduction of parameterized tests. By leveraging JUnit’s parameterized testing capabilities, multiple input scenarios could be tested within a single method, reducing code duplication and improving readability. This refinement helped in streamlining the test suite and making it more maintainable.

**Ensuring Technical Soundness and Code Efficiency**

Technical soundness was achieved by ensuring that all exception handling, input validation, and logic checks were thoroughly tested. For example, a test case was written to verify that an invalid task ID would trigger an exception, ensuring that the system rejected improperly formatted input. Similarly, edge cases such as empty strings, null values, and excessively long inputs were tested to prevent unexpected runtime errors.

Efficiency was also a key consideration in writing test cases. To eliminate redundancy, parameterized tests were used to validate multiple variations of invalid input in a concise manner. This approach not only made the test suite more manageable but also significantly reduced the need for repetitive test methods. Additionally, setup and teardown methods were optimized using JUnit’s @BeforeEach annotation, ensuring that each test case started with a clean environment while avoiding unnecessary reinitialization of objects.

**Reflection on Testing Techniques and Mindset**

Several software testing techniques were employed in this project to ensure comprehensive coverage. Black-box testing was used to validate expected behaviors based on functional requirements without analyzing the internal structure of the code. White-box testing was applied to evaluate control flow and logic, ensuring that exception handling and validation mechanisms functioned as expected. Boundary value analysis was also implemented to test limits such as minimum and maximum allowable values, helping to identify potential edge-case failures.

Some techniques were not applied in this project, such as integration testing and exploratory testing. Integration testing was not necessary since unit testing focused on individual components rather than system-wide interactions. Exploratory testing, which involves manual testing without predefined test cases, was omitted since structured JUnit tests provided the required validation.

A cautious and methodical approach was adopted while working on this project. Understanding the complexity and interrelationships of the code being tested was essential to ensuring its reliability. One example of this was testing null input handling in the contact service, which helped prevent runtime errors that could compromise application stability. Recognizing these dependencies reinforced the need for comprehensive test coverage and early bug detection.

Efforts were also made to limit bias in the testing process. Peer reviews and diverse test cases helped ensure an objective evaluation of the code. If a developer is solely responsible for testing their own code, bias can become a concern, as they may unintentionally overlook flaws or assume correct behavior without verifying edge cases. To counteract this, structured test plans and independent validation steps were followed to minimize subjective assumptions.

Maintaining a strong commitment to quality is a fundamental responsibility of a software engineering professional. Cutting corners in testing can lead to significant issues down the line, including undetected bugs, system failures, and increased maintenance costs. To avoid technical debt, best practices such as maintaining comprehensive test documentation, automating tests in CI/CD workflows, and regularly refactoring code to align with evolving requirements must be followed. Prioritizing these principles ensures that software remains robust, scalable, and maintainable over time.

**Conclusion**

The unit testing approach applied in Project One was instrumental in ensuring the quality, stability, and reliability of the developed mobile application. JUnit tests provided a structured validation mechanism that significantly reduced defects and improved maintainability. By implementing black-box and white-box testing techniques, handling edge cases, and refining test efficiency, the software was able to meet its intended requirements with high confidence. Moving forward, additional automation and integration testing will further enhance the development process and minimize future risks. A disciplined approach to testing and quality assurance will remain a core focus in delivering reliable and scalable software solutions.

**References**

**Beck, K., & Gamma, E. (2000).** *JUnit: A Cook’s Tour.* Addison-Wesley.

**Myers, G. J., Sandler, C., & Badgett, T. (2011).** *The Art of Software Testing (3rd ed.).* John Wiley & Sons.

**Fowler, M. (2018).** *Refactoring: Improving the Design of Existing Code (2nd ed.).* Addison-Wesley.