**Project Two**

**Summary and Reflection**

Taylor R. Jones

CS320 – Software Testing, Automation

SNHU

Professor David Handlos

2/22/2025

**Alignment to Requirements**

My approach to designing unit tests for each of the three features was entirely driven by the software requirements, employing a requirements-based testing methodology for the Contact, Appointment, and Task functionalities. “Making testing strategies align with the requirements enhances the chances of delivering a software product that complies with predefined benchmarks and user expectations” (*Requirement-Based Testing in Software Development*, 2024). For example, in the testPhoneNumberLength() method for the Contact class, I used assertThrows() to test an input of nine digits, ensuring that an exception was correctly thrown, as the system requires exactly ten-digit phone numbers. Similarly, I used tests like testPhoneNull() and testAddressNull() with the same assertion method to validate that an exception is thrown when fields contain null values. By aligning my tests with these constraints, I ensured that all software functionalities met the predefined system requirements, preventing invalid data from being processed.

**Effective Tests**

With 79% test coverage, my tests effectively validated most of the system’s functionality. Google’s guidelines suggest that 60% is acceptable, 75% is commendable, and 90% is exemplary (Arguelles et al., 2020), positioning my test coverage in the commendable range. While 79% test coverage is commendable, additional tests for things like overlapping appointments would prevent scheduling conflicts, ensuring that users cannot double-book a time slot. Without these tests, the system could allow conflicting appointments, leading to potential errors in real-world usage.

**Technically Sound Code**

To maintain technically sound code, I implemented strict input validation, exception handling, and enforcement of business rules. In the constructors for each feature, I used conditional if statements to verify input constraints to ensure that all input met their respective requirements. I validated these conditions through JUnit tests, using annotations like assertThrows() to confirm that exceptions were correctly handled. For example:



JUnit tests using assertThrows() ensured proper exception handling, as seen in testFirstNameNull(), which validated that a null first name triggers an IllegalArgumentException.

**Efficient Code**

To ensure efficiency, I optimized test logic and minimized redundancy. I used the @BeforeEach annotation in TaskServiceTest to set up reusable test data, reducing redundant object creation and system overhead. Instead of writing separate tests for each invalid input, I combined multiple assertions within a single test, as seen in testInvalidPhoneNumber(), which checks both too-short and too-long phone numbers. Additionally, I used @DisplayName to improve test readability. For in-memory data storage, I chose ArrayList due to its dynamic resizing, efficient indexed access, and seamless modifications, making it ideal for managing objects.

**Techniques Employed**

The primary testing technique used in this project was unit testing with JUnit, which verifies that individual software components function correctly in isolation (What Is Dynamic Testing? (Types and Methodologies), 2023). Unit tests are deterministic, consistently producing the same result for a given input, and execute quickly, making them ideal for continuous testing.

To validate functionality, I used assertions throughout my test classes, ensuring that validation logic worked as expected. I wrote unit tests for all user inputs across the Contact, Appointment, and Task features, achieving 79% test coverage. For example, in testAddAndRetrieveAppointment(), I used:

to confirm that appointments were assigned unique IDs which was a software requirement.

**Other Techniques and Implications**

Other software testing techniques not used in this project include integration testing, system testing, performance testing, and user acceptance testing (UAT).

Integration testing verifies interactions between multiple components, ensuring they work together correctly and catching issues that unit tests might miss. Unlike unit testing, which isolates individual components, integration testing focuses on system dependencies, such as APIs, databases, or external services (GeeksforGeeks, 2018). Since this project relied on in-memory data structures rather than external systems, integration testing was unnecessary.

System testing evaluates the entire application to ensure all features function correctly when combined. For example, in this project, a system test might simulate a user creating a contact, updating the contact’s information, and scheduling an appointment through a user interface. System testing typically involves multiple services working together, such as ContactService, AppointmentService, and TaskService, in a single test case. However, because this project focused on backend logic rather than a fully deployed application, system testing was not relevant at this stage.

Performance testing measures speed, responsiveness, and scalability under different conditions, ensuring the system can handle high loads efficiently (What Is Performance Testing?, n.d.). From a practical standpoint, this may include things like testing the system under load. Since this project does not require real-time performance validation or large-scale data processing, performance testing is not necessary.

User Acceptance Testing (UAT) ensures the software meets business and user requirements by having end-users validate real-world functionality. In this project, UAT might involve a user manually verifying that they can create tasks, schedule appointments, and update contact information. Since UAT is typically performed late in the development process, it was not applicable at this stage of unit testing.

**Caution**

Working on this project required me to adopt an analytical and detail-oriented mindset, ensuring that my unit tests were not only functional but also aligned with the software requirements. As a tester, I exercised a high degree of caution to identify potential defects early and proactively addressed them. To enhance reliability, I also incorporated edge-case unit tests to prevent future issues. For example, by writing tests that check for null values, I ensured that invalid data could not propagate through the system, reducing the risk of failures. In the Contact class, I verified that each field properly threw an exception when assigned a null value, preventing the creation of invalid objects and enforcing data integrity.

**Bias**

Developers naturally carry some degree of bias toward their own work, often testing for expected outcomes rather than exploring edge cases. Ideally, a separate software tester would be responsible for validating the code. However, as both the developer and tester in this project, I had to take extra steps to limit bias. Initially, I found myself writing tests primarily for scenarios I knew would work, sometimes overlooking edge cases that could expose potential failures. Over time, I realized that I needed to test both successful and unsuccessful scenarios, ensuring that the system handled invalid inputs correctly. To mitigate bias, I adopted the strategy of writing my code and tests on different days, allowing me to approach testing with a fresh perspective.

**Discipline**

As far are being disciplined, I did my best to try to adhere to best practices. In retrospect, I took some shortcuts and skipped some unit tests, especially edge cases, that may have offered a greater degree of test coverage. If I were to approach this project again, I would use Jacoco to visually highlight untested areas, ensuring closer to 100% coverage. Additionally, by parameterizing tests using JUnit’s @ParameterizedTest annotation, I could eliminate hardcoded values, making tests more reusable and reduce technical debt.

References

Arguelles, C., Ivanković‎, M., & Bender, A. (2020, August 7). *Code Coverage Best Practices*. Google Testing Blog. https://testing.googleblog.com/2020/08/code-coverage-best-practices.html

GeeksforGeeks. (2018, October 10). *Integration Testing Software Engineering*. GeeksforGeeks. https://www.geeksforgeeks.org/software-engineering-integration-testing/#what-is-integration-testing

*Requirement-Based Testing in Software Development*. (2024, January 10). GeeksforGeeks. https://www.geeksforgeeks.org/requirement-based-testing-in-software-development/

*What is Dynamic Testing? (Types and Methodologies)*. (2023, February 15). BrowserStack. https://www.browserstack.com/guide/dynamic-testing

*What is Performance Testing?* (n.d.). OpenText. https://www.opentext.com/what-is/performance-testing