**Project Two: Summary & Reflection**

Fnu Samim

Southern New Hampshire University

CS-320: Software Test, Automation QA

Dr. Joseph Martinez

April 20, 2025

**Unit Testing Approach for Contact, Task, and Appointment Services**

I designed unit tests for the contact, task, and appointment services to meet functional requirements. For the task service, I used Boundary Value Analysis (BVA) to enforce the 10-character task ID limit, catching violations with an IllegalArgumentException in testLongTaskId (TaskTest, lines 33–36). Null inputs were tested in testNullTaskName (TaskTest, lines 41–44), and thread safety was confirmed via testConcurrentUpdates (TaskServiceTest, lines 124–144). The appointment service applied BVA to ID/description lengths and Equivalence Partitioning to accept only current/future dates. The contact service focused on boundary checks and immutable collections to protect data.

**Alignment with Software Requirements**

The approach aligned closely with requirements. The task ID limit was validated by testLongTaskId, duplicate prevention by testAddTaskAndDuplicatePrevention (TaskServiceTest, lines 23–30), and task deletion by testAddDeleteSequence (TaskServiceTest, lines 148–157). Appointment date validation ensured only current/future dates were accepted, meeting specifications. These tests aligned with project goals, fully confirming the system’s behavior.

**Effectiveness of JUnit Tests**

The JUnit suite tackled edge cases, invalid inputs, and concurrency. Without formal coverage tools like JaCoCo, I relied on thorough test design, addressing input validation (testLeadingTrailingSpacesInTaskName, TaskTest, lines 89–92), thread safety (testThreadSafety, TaskServiceTest, lines 161–176), and concurrent performance (testConcurrentAdd, TaskServiceTest, lines 34–61). Null handling (testUpdateWithNullValues, lines 111–120) and duplicate prevention worked flawlessly, proving test effectiveness.

**Experience Writing JUnit Tests**

Writing JUnit tests was a grind, I recall wrestling with testConcurrentUpdates for hours, chasing a race condition that taught me to triple-check thread synchronization. I focused on edge cases like spaces (testLeadingTrailingSpacesInTaskName) and nulls (testNullTaskName), refining tests to cut redundancy while hitting all requirements. The process sharpened my system insights.

**Technical Soundness**

Defensive programming ensured robustness. The validateTaskId method (Task, lines 8–14) blocked invalid IDs, and a synchronized block in updateTask (TaskService, line 59) avoided race conditions, proven by testThreadSafety. Immutable collections in the appointment service, tested to block external changes, safeguarded data integrity..

**Efficiency**

I optimized performance with ConcurrentHashMap (TaskService, line 13) for O(1) operations, handling 100 simultaneous tasks in testThreadSafety. The putIfAbsent method (TaskService, line 29) enabled thread-safe insertions, stress-tested in testConcurrentAdd (TaskServiceTest, lines 34–61), ensuring scalability.  
**Software Testing Techniques Employed**

I used several testing techniques to ensure the reliability of the contact, task, and appointment service systems. Boundary Value Analysis (BVA) checked input limits, like max appointment ID lengths, catching invalid entries. Equivalence Partitioning sorted inputs into valid (current dates) or invalid (past dates) groups. Exception Testing confirmed the system handled null inputs and missing appointments gracefully. Defensive Programming ensured dates stayed immutable and collections were tamper-proof. State-Based Testing verified system updates after adding or removing appointments. Concurrency Testing tested multi-threaded access, keeping data safe from corruption.

**Software Testing Techniques Not Employed**

I skipped some advanced methods to nail down unit testing basics. Parameterized Testing (like JUnit 5’s @ParameterizedTest) could’ve made testing different inputs less repetitive. Property-Based Testing might’ve caught edge cases, like special characters. Mutation Testing with PITest could’ve tested my suite’s strength by adding fake bugs. I also didn’t do Performance Testing for heavy loads or Integration Testing with databases, but these could level up testing for bigger, complex systems.

**Practical Uses and Implications**

BVA suits projects with strict input rules, like healthcare systems (Kuhn, Kacker, & Lei, 2010). Defensive programming is ideal for security-sensitive applications, such as financial systems (Pashkovskaya, 2024). Concurrency testing supports multithreaded systems, like reservation platforms (Cohen, 2023). Parameterized testing excels in data-driven contexts, like e-commerce (Tillmann & Schulte, 2005). Mutation testing strengthens safety-critical software, such as aviation systems (Gurbuz, Tekinerdogan, Catal, & Er, 2024). UAT ensures consumer apps meet user needs (Content Square, 2025). My approach met unit-testing goals, but broader techniques would benefit larger systems.

**Caution as a Software Tester**

I stayed cautious, obsessing over edge cases after a null input slipped through early tests, crashing the task service. That mistake made me rigorous about validating every scenario, like in testNullTaskName. For example, testing null inputs (testNullTaskName) and concurrent updates (testConcurrentUpdates) ensured the system was strong against unexpected scenarios. Appreciating the complexity and interrelationships of the code was key, as concurrency issues in updateTask (TaskService, line 59) could have led to data corruption without proper synchronization. This caution prevented subtle bugs that could arise from overlooked interactions.

**Limiting Bias**

To avoid bias, I anchored my tests in requirements, not assumptions. How could I assume the code would behave as expected without rigorous validation? For instance, testLeadingTrailingSpacesInTaskName was driven by the requirement to trim spaces, not by my expectations of the implementation. If testing my own code, bias could arise from overconfidence in my design, potentially overlooking edge cases like special characters. To mitigate this, I relied on systematic techniques like BVA and equivalence partitioning, which forced me to consider inputs without bias.

**Commitment to Quality**

Discipline in maintaining quality is vital, as cutting corners in testing or coding can introduce technical debt, leading to costly fixes later. For example, skipping concurrency tests could have left race conditions undetected, compromising reliability. To steer clear of technical debt, I will stick to thorough testing, lean on code reviews, and refactor bit by bit to keep things clean. Honestly, I have learned the hard way that shortcuts are not worth it. I would like to incorporate test coverage tools and automated test runners as part of my workflow.

# **References**

Cohen, N. (2023, October 23). *Concurrency Testing: Benefits, How-To, and Tools To use*. Retrieved from BlazeMeter: https://www.blazemeter.com/blog/concurrency-testing#:~:text=Back%20to%20top-,What%20is%20Concurrency%20Testing?,monitor%20behavior%20under%20this%20load.

Content Square. (2025). *Usability Testing: what it is, its benefits, and why it matters*. Retrieved from Content Square: https://contentsquare.com/guides/usability-testing/

Gurbuz, H. G., Tekinerdogan, B., Catal, C., & Er, N. P. (2024, January 24). *Test suite assessment of safety-critical systems using safety tactics and fault-based mutation testing*. Retrieved from Springer: https://link.springer.com/article/10.1007/s10586-023-04229-x#author-information

Kuhn, R., Kacker, R., & Lei, Y. (2010, October 7). *Practical Combinatorial Testing*. Retrieved from NIST: https://csrc.nist.gov/pubs/sp/800/142/final

Pashkovskaya, E. (2024, July 15). *Defensive Coding: Key Security Practices in Software Development*. Retrieved from Neklo: https://neklo.com/blog/software-development-security#:~:text=Authentication%20and%20authorization%20control%20access,areas%20an%20attacker%20can%20target.&text=Input%20validation%20is%20a%20critical,the%20server%20before%20processing%20data.

Tillmann, N., & Schulte, W. (2005, September 5). *Parameterized unit tests*. Retrieved from ResearchGate: https://www.researchgate.net/publication/221560789\_Parameterized\_unit\_tests