Nicholas Harris

CS 320 Software Test, Automation & QA

October 19, 2025

Summary and Reflections Report

SUMMARY

Unit Testing Approach and Alignment to Requirements

For this project, I developed comprehensive JUnit tests for three back-end services: Contact Service, Task Service, and Appointment Service. My testing approach directly aligned with the software requirements by ensuring every requirement had corresponding test cases.

For Contact Service, I created 59 tests covering unique IDs (max 10 characters), names (max 10 characters), phone numbers (exactly 10 digits), and addresses (max 30 characters). In ContactTest.java lines 23-27, I tested that IDs exceeding 10 characters throw IllegalArgumentException, and in lines 29-33, I verified exactly 10 characters work correctly. This approach came from my work as a build engineer at a game startup - we can't ship builds with validation bugs because they'll crash on players' machines.

For Task Service, I wrote 40 tests validating unique task IDs (max 10 characters), names (max 20 characters), and descriptions (max 50 characters). TaskTest.java lines 47-51 verify names exceeding 20 characters get rejected. The requirement that task IDs aren't updatable was tested in lines 37-44 by confirming no setter method exists.

For Appointment Service, I developed 14 tests for unique IDs (max 10 characters), future dates, and descriptions (max 50 characters). AppointmentTest.java lines 95-99 test that past dates throw exceptions. My approach systematically covered positive tests (valid inputs), negative tests (invalid inputs), boundary tests (maximum lengths), null tests, and edge cases. Every requirement had at least one corresponding test case.

Quality and Effectiveness of JUnit Tests

I know my tests were effective because all 113 tests passed with zero failures: Contact Service (59 tests, 100% pass rate), Task Service (40 tests, 100% pass rate), and Appointment Service (14 tests, 100% pass rate). I tested every validation rule in constructors and setters. In ContactTest.java, I have dedicated tests for null validation (lines 40, 62, 98, 118) and length validation (lines 42, 64, 100, 120-124). I tested both valid and invalid inputs for every validation, ensuring error paths got exercised. Edge cases like exact 50-character boundaries (AppointmentTest.java lines 181-186) and multiple sequential updates (lines 203-214) verify state consistency. During development, I found validation bugs - I initially forgot to validate that phone numbers were numeric, and my test in ContactTest.java lines 108-111 caught it immediately.

Experience Writing Technically Sound and Efficient Code

I made my test code technically sound by using proper JUnit 5 annotations (@Test) and assertions (assertEquals, assertThrows, assertNotNull) throughout all test files. I wrote descriptive test method names like testContactIdTooLong() (ContactTest.java line 22) and testAppointmentDateInPast() (AppointmentTest.java line 95). At my startup, clear names mean faster debugging. I used assertThrows() to test exceptions (ContactServiceTest.java lines 43-47)

rather than try-catch blocks. I validated multiple object properties after operations - in ContactServiceTest.java lines 178-183, I verified all five fields after retrieval.

For efficiency, I used @BeforeEach to eliminate duplicate setup code. In ContactServiceTest.java lines 17-21, I initialize fresh test data before each test, eliminating repeated setup in 27 test methods - cutting approximately 150 lines of duplication. I created reusable helper methods in AppointmentTest.java lines 16-34 (getFutureDate, getPastDate) called 30+ times, eliminating redundant calendar code. In TaskService.java lines 19-20, I chose HashMap over ArrayList for O(1) lookup performance instead of O(n) linear search. With 40 task tests performing multiple operations, this efficiency adds up. I made ID fields final (Contact.java line 17, Task.java line 8) since requirements specified IDs aren't updatable, preventing bugs and enabling JVM optimizations.

REFLECTION

Testing Techniques Employed

I primarily used unit testing - testing individual components in isolation without external dependencies. Each test class tested a single class, ran quickly (under 2 seconds total), and provided fast feedback. I heavily used boundary value analysis, testing values at acceptable range edges where bugs often hide. In ContactTest.java, I tested phone numbers at 9, 10, and 11 digits (lines 100, 114, 104) and first names at exactly 10 and 11 characters (lines 52, 43). Boundary testing is critical in games - crashes at level 99 or 255 items are usually boundary bugs.

I used equivalence partitioning to reduce test cases while maintaining coverage. For phone validation, I picked representative invalid examples: too short, too long, containing hyphens, and

containing letters (lines 100, 104, 108, 113). I applied positive and negative testing systematically - positive tests verify valid inputs work (ContactTest.java line 16), negative tests verify invalid inputs get rejected (lines 40-44). I used exception testing with assertThrows() (ContactServiceTest.java lines 43-47, TaskServiceTest.java lines 111-121) to verify error handling. I also implemented integration testing at the service level - ContactServiceTest.java lines 219-242 exercises complete workflows that test how components work together.

Other Testing Techniques Not Used

System testing evaluates the complete application with end-to-end workflows across services. I didn't use it because this project focused on individual services. At my game startup, system tests simulate full player sessions and take 15-30 minutes versus seconds for unit tests. Performance testing measures execution speed and behavior under load. I didn't do it because requirements focused on functional correctness, not performance. In game development, we test extensively because games below 60 FPS feel laggy. User acceptance testing (UAT) involves end-users verifying the application meets their needs. I was both developer and tester, so true UAT wasn't possible. At my startup, playtests with actual gamers find issues we'd never think of. Security testing finds vulnerabilities like SQL injection and authentication bypasses. I didn't do it because these services lack authentication, databases, or web interfaces creating attack surfaces. Regression testing verifies new changes don't break existing functionality. I ran my test suite multiple times, but didn't have formal regression testing with CI/CD. At work, automated regression tests run on every commit - if code breaks 5,000+ tests, the build fails immediately.

Practical Uses and Implications of Techniques

Unit testing is ideal for all software projects during development, especially for long-term maintained code. At my startup, unit tests catch bugs before QA, saving days of debugging. The

implication is developers need testable code - small, focused functions not tightly coupled to dependencies. Boundary value analysis should be used whenever code has limits or validations. It's essential for financial software (transaction amounts), e-commerce (cart limits), and data validation. Requirements must clearly specify boundaries. Integration testing is critical for interconnected components before deployment. In games, it verifies inventory, quest, and achievement systems work together. Integration tests are slower than unit tests, so the test pyramid suggests many unit tests, fewer integration tests, even fewer system tests. System testing is necessary before major releases. It requires realistic test environments mirroring production. Performance testing is essential for applications serving many users or having strict response times. It requires specialized tools and infrastructure. Security testing is mandatory for applications handling sensitive data and is legally required in many industries due to GDPR, HIPAA, and PCI DSS regulations.

Mindset: Employing Caution

Working as a tester required extreme caution. I approached testing assuming my code probably has bugs I haven't found yet. This came from my build engineering work - one bad build can crash the game for thousands of players on launch day. I tested not just happy paths but edge cases and error conditions. In AppointmentTest.java lines 181-186, I tested descriptions at exactly 50 characters, one over 50, and empty strings. I appreciated code complexity by realizing changing one validation could break other parts. In Contact.java, the phone validation regex depends on the length check working correctly. My test in ContactTest.java lines 104-106 would catch bugs from incorrect changes. In ContactService.java, updatePhone() retrieves a contact and calls its setter. If getContact() returns null instead of throwing an exception, it causes NullPointerException. My test verifies it throws IllegalArgumentException

(ContactServiceTest.java lines 152-155). At my startup, inventory bugs cascade into quest and achievement systems, requiring the same cautious testing mindset.

Mindset: Limiting Bias

I limited bias by approaching tests as a skeptic, not the code's creator. I wrote tests before reading implementation code closely - when testing TaskService, I read requirements, wrote tests, then checked if my implementation matched. This prevented unconsciously writing tests matching my bugs rather than requirements. I tested from the user's perspective - users care about adding, updating, and deleting tasks without errors, not that I use HashMap. My integration tests (ContactServiceTest.java lines 219-242) simulate realistic workflows. I explicitly tested things I assumed would work - I tested that updating first name doesn't affect phone number by verifying unrelated fields stay unchanged.

Developer bias would definitely be a concern testing my own code. Developers see what they intended rather than what they wrote. When I first implemented phone validation, I thought "of course this works" but my test caught the bias (ContactTest.java lines 100-106). I initially wrote TaskService without null checks, assuming "I won't pass null task IDs," but a tester asks "what if someone does?" (TaskServiceTest.java lines 75-80). This is why teams have dedicated QA engineers - fresh eyes catch bias developers miss.

Mindset: Discipline and Quality Commitment

Being disciplined about quality is critical because cutting corners creates technical debt that grows over time. Skipping edge case tests because "probably nobody will enter exactly 50 characters" lets bugs slip to production. Fixing them later requires patches, user coordination, and handling corrupted data - exponentially more expensive than one extra test. At my startup, a

"small" collision bug QA didn't fully test caused thousands of complaints about falling through the world on launch day.

Quality isn't free, but lack of quality is really expensive. Writing 113 tests took hours, but that's cheaper than Grand Strand Systems' app crashing for thousands of users. IBM research found production bugs cost 100 times more to fix than bugs found during development. My tests are insurance - small investment preventing huge costs. As a professional, my reputation depends on quality work. At my startup, we've lost player trust from buggy releases - players who quit after bad patches rarely return.

To avoid technical debt, I'll write tests as I develop code, not after. I'll maintain test suites over time, updating tests when requirements change. I'll use automated testing in CI/CD pipelines like at work - every commit triggers tests, builds only deploy if all pass. I'll advocate for realistic deadlines including testing time. When managers pressure "just ship it, fix bugs later," I'll explain the business cost - lost users, support costs, emergency patches, reputation damage. I'll treat test code as production code - documented, reviewed, and maintained. Poorly written tests give false confidence, worse than no tests.

Looking back, I learned testing requires a completely different mindset than development. When coding, I'm trying to make things work. When testing, I'm trying to break things. This tension produces robust software handling edge cases, error conditions, and unexpected inputs gracefully. The discipline to write comprehensive tests even when deadlines are tight separates professional engineers from amateurs. My 113 tests with 100% pass rate show that commitment, and I'll carry this approach throughout my career as a build engineer and software developer.