John Chomen  
CS 320 Software Test, Automation

Professor Angelo Luo  
October 06, 2025

Project Two

This report summarizes my software testing approach for the contact, task, and appointment services, reflects on the effectiveness of my JUnit tests, and explains the techniques and tester mindset I applied. The evidence below references specific unit tests from the project to demonstrate how the tests align with the functional requirements and how they promote both technical soundness and efficiency.

Unit testing approach by feature

Contact service: I used equivalence partitioning, boundary-value analysis, and negative testing to exercise constructor validation and all update methods. Tests cover valid object creation and field updates, plus invalid inputs for null, length, and format (10-digit phone).

Task service: I verified unique ID enforcement, field-length validation, and update paths for name and description. Service tests cover add/update/delete and error paths for nonexistent IDs.

Appointment service: I validated that dates cannot be in the past (per requirement using java.util.Date and before(new Date())), checked description length and nulls, and tested service add/delete and update helpers. A defensive-copy test guards against external Date mutation.

Alignment to software requirements

My tests were written directly from the acceptance criteria:  
Contact: immutable ID ≤10; first/last ≤10; phone exactly 10 digits; address ≤30. Task: immutable ID ≤10; name ≤20; description ≤50.  
• Appointment: immutable ID ≤10; non-past java.util.Date; description ≤50.  
Each assertion and negative test maps back to one of these constraints or service behaviors (add, unique IDs, delete by ID, and updating allowed fields).

Effectiveness and coverage

The suite exercises normal and exceptional flows across constructors and service methods. Based on Codio’s requirement, the project is designed to achieve at least 80% line coverage. After running the tests in Codio, record the actual percentage here: [Coverage: \_\_\_\_\_\_%]. High coverage plus targeted boundary and error-path assertions increase defect-detection capability beyond a single percentage metric.

Experience writing the JUnit tests

To keep tests technically sound, I asserted both state and behavior: constructors reject bad input, services enforce uniqueness, and updates persist changes. I also favored small, focused tests to isolate failures and reduce flakiness. Below are representative snippets from my tests:

Valid Contact creation (technical soundness)

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

Invalid phone rejected (boundary/negative test)

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

Service updates persisted (ContactService)

}  
  
 @Test  
 void testUpdateContact() {  
 service.updateFirstName("1", "Mike");  
 service.updateLastName("1", "Adams");  
 service.updatePhone("1", "2223334444");  
 service.updateAddress("1", "987 Cedar Rd");

Task description update

Task task = new Task("404", "Name", "OldDesc");  
 service.addTask(task);  
 service.updateTaskDescription("404", "NewDesc");  
 assertEquals("NewDesc", service.getTask("404").getDescription());  
 }

Task deletion and null retrieval

Task task = new Task("202", "DeleteMe", "Desc");  
 service.addTask(task);  
 service.deleteTask("202");  
 assertNull(service.getTask("202"));  
 }

Past appointment date rejected

service.addAppointment("A5", future, "Good");  
 Date past = new Date(System.currentTimeMillis() - 60\_000);  
 assertThrows(IllegalArgumentException.class, () -> service.updateAppointmentDate("A5", past));  
 String longDesc = "x".repeat(51);  
 assertThrows(IllegalArgumentException.class, () -> service.updateAppointmentDescription("A5", longDesc));

Defensive copy of Date (immutability guard)

Appointment a = new Appointment("CONSTID", future, "desc");  
 // There is no setter; ensure getter still returns original value  
 assertEquals("CONSTID", a.getAppointmentId());  
 }  
  
 @Test  
 void defensiveCopy\_onDateGetter() {  
 Date future = new Date(System.currentTimeMillis() + 60\_000);  
 Appointment a = new Appointment("ID7", future, "desc");  
 Date obtained = a.getAppointmentDate();  
 obtained.setTime(obtained.getTime() + 86\_400\_000L); // attempt to mutate  
 // original should remain unchanged  
 assertTrue(a.getAppointmentDate().before(new Date(obtained.getTime())) || a.getAppointmentDate().equals(future));

Efficiency considerations included: (1) avoiding redundant fixtures by using @BeforeEach where appropriate; (2) testing one behavior per test for fast pinpointing; and (3) relying on in-memory Maps to keep tests fast. Assertions are minimal yet sufficient, keeping each test concise and quick to run.

Equivalence Partitioning: Grouped inputs into valid vs. invalid classes (e.g., phone numbers of exactly 10 digits vs. non-digit or wrong-length values).  
Boundary-Value Analysis: Targeted limits like 10-character IDs/names and 50-character descriptions, plus date at the current moment vs. past.  
Negative/Exception Testing: Verified that constructors and service methods throw when requirements are violated (duplicate IDs, past dates, nulls, excessive lengths).  
State-Based Testing: After updates, asserted that object state matched expectations.  
Immutability/Defensive Copy Checks: Ensured ID immutability and verified Appointment returns a defensive copy of Date.

Techniques not used (and why)

Integration/End-to-End UI Tests: Out of scope; the project is service-level with no UI.  
Mocking/Spies: Not needed because services use in-memory Maps; there are no external dependencies.  
Property-Based Testing: Useful for exploring large input spaces, but unnecessary for the small, fixed constraints here.  
Mutation Testing: Valuable for measuring test rigor, but not required by this assignment’s toolchain.  
Parameterized Tests: Could reduce repetition for boundary cases but standard tests were sufficiently clear and short.

Practical uses and implications

In microservices or domain layers with strict field constraints, boundary-value and negative tests offer high ROI by catching violations early. For systems with external dependencies, mocking and integration tests become essential. Property-based and mutation testing are beneficial on complex parsing or algorithm-heavy projects to improve robustness and validate that tests meaningfully fail when code is altered.

Mindset

I approached the project with caution, treating every requirement as a potential failure point if untested. For example, I verified that the Appointment date could not be set to the past and that Contact phone numbers are strictly 10 digits. Appreciating interrelationships (e.g., service updates invoking domain validation) ensured updates couldn’t bypass constructor rules.  
  
 To limit bias, I wrote negative tests first for many constraints and then verified the positive path. As a developer, testing my own code can introduce confirmation bias; countermeasures include writing tests from the requirement text, performing code reviews, and using mutation testing when available.  
  
 Discipline is central to software quality. Cutting corners on tests leads to technical debt—like allowing bad state into domain objects—which becomes expensive to fix later. To avoid debt, I plan to: keep tests close to requirements, enforce immutability where appropriate, cover error paths, and automate coverage gates (e.g., fail builds under 80% coverage).