**Summary**

**How I Tested Each Feature**

Contact service. I took the requirements and turned them straight into checks. IDs must be unique and not null. First and last names have max length. Phone is exactly 10 digits. Address has a max length. I wrote happy‑path tests that pass at the exact limits and negative tests that fail just past them. I used parameterized tests so I could hit a bunch of lengths and formats without copy‑pasting. I also tested update and delete so the collection never ends up with duplicates or missing items.

Task service. I treated the ID as the identity and kept it immutable. Name and description have length and null rules. I tested create with max‑length values, then tried values that are too long or blank. I verified updates change only the allowed fields. I added tests for “task not found” so the service throws the right error instead of failing silently.

Appointment service. Dates are the risky part. I injected a “clock”/supplier so tests don’t depend on my real machine time. I checked that today is rejected, tomorrow is accepted, and weird edges like end‑of‑month and leap‑year still behave. I kept the same pattern as the other services: unique ID, description length rules, create/find/delete happy paths plus negatives.

**How This Lines Up With the Requirements**

I mapped tests one‑to‑one with the rules from Project One. Examples: contactId is non‑null, unique, and ≤ 10; firstName/lastName non‑null and ≤ 10; phone exactly 10 digits; address non‑null and ≤ 30. For Task, taskId non‑null, unique, ≤ 10; name ≤ 20; description ≤ 50; ID never changes. For Appointment, appointmentId ≤ 10 and unique; date strictly in the future; description ≤ 50. Test names read like the rule so it’s easy to trace: createContact\_rejectsPhoneNotExactly10Digits, updateTask\_disallowsIdChange, createAppointment\_rejectsToday\_acceptsTomorrow.

**Why I’m Confident in the Tests (Coverage + Behavior)**

I checked JaCoCo and recorded both line and branch coverage. My final run showed: overall line coverage = ⟨add your %⟩ and branch coverage = ⟨add your %⟩. By service: Contact ⟨%⟩ line / ⟨%⟩ branch; Task ⟨%⟩ line / ⟨%⟩ branch; Appointment ⟨%⟩ line / ⟨%⟩ branch. I also did a quick “mutation‑style” sanity check: I flipped a comparison or removed a validation and confirmed a test failed. That told me the suite doesn’t just execute code—it catches real mistakes.

**Writing the Tests: keeping them sound and efficient**

Technically sound. I used assertThrows for specific exceptions so I’m not hiding bugs behind a generic failure. I kept names explicit so failures point to the rule that broke. I injected a clock for time logic so tests don’t flake.

Efficient. I used @BeforeEach to reset a small, in‑memory service so tests are fast. Parameterized tests cover many inputs in one shot. I avoided shared mutable state across tests.

**Examples (trimmed for clarity).**

class ContactServiceTests {

private ContactService service;

@BeforeEach

void init() { service = new ContactService(); }

@ParameterizedTest(name = "firstName length = {0}")

@CsvSource({"0,false","1,true","10,true","11,false"})

void firstName\_boundaries(int len, boolean accept) {

String first = "A".repeat(len);

if (accept) {

assertDoesNotThrow(() -> service.create("ID1", first, "Lo", "0123456789", "123 Main"));

} else {

assertThrows(IllegalArgumentException.class,

() -> service.create("ID2", first, "Lo", "0123456789", "123 Main"));

}

}

@ParameterizedTest

@CsvSource({"123456789,false","1234567890,true","12345678901,false","abcdefghij,false"})

void phone\_mustBeExactly10Digits(String phone, boolean accept) {

if (accept) {

assertDoesNotThrow(() -> service.create("ID3", "Al", "Lo", phone, "123 Main"));

} else {

assertThrows(IllegalArgumentException.class,

() -> service.create("ID4", "Al", "Lo", phone, "123 Main"));

}

}

@Test

void duplicateIds\_areRejected() {

service.create("DUP", "Al", "Lo", "0123456789", "123 Main");

assertThrows(IllegalStateException.class,

() -> service.create("DUP", "Bo", "Li", "0123456789", "456 Oak"));

}

}

class TaskServiceTests {

private TaskService service;

@BeforeEach void init() { service = new TaskService(); }

@Test

void accepts\_maxBoundaries() {

assertDoesNotThrow(() -> service.create("T1", "N".repeat(20), "D".repeat(50)));

}

@Test

void rejects\_nameTooLong() {

assertThrows(IllegalArgumentException.class,

() -> service.create("T2", "X".repeat(21), "ok"));

}

@Test

void id\_isImmutable() {

service.create("T3", "Ok", "Ok");

assertThrows(UnsupportedOperationException.class,

() -> service.update("NEWID", "T3", "New", "New"));

}

}

class AppointmentServiceTests {

private AppointmentService service; private LocalDate today;

@BeforeEach

void init() {

today = LocalDate.of(2025, 1, 31); // fixed point in time

service = new AppointmentService(() -> today);

}

@Test

void rejects\_today\_accepts\_tomorrow() {

assertThrows(IllegalArgumentException.class,

() -> service.create("A1", today, "desc"));

assertDoesNotThrow(() -> service.create("A2", today.plusDays(1), "desc"));

}

@Test

void leapYear\_edge\_still\_ok() {

LocalDate leap = LocalDate.of(2028, 2, 29);

assertDoesNotThrow(() -> service.create("A3", leap, "desc"));

}

}

**Reflection**

**Techniques I Used**

I leaned on equivalence partitioning (valid vs. invalid buckets), boundary value analysis (max, max+1), negative testing with clear exceptions, simple state‑based checks for the in‑memory collections, and a deterministic clock for date rules. These gave me wide coverage with a small, readable suite.

**Techniques I Did Not Use (but know when to)**

I didn’t add full integration tests, mutation testing with a tool, property‑based testing, fuzzing, contract/consumer‑driven tests, or UI/end‑to‑end tests. Those shine when services talk to databases or other services, when safety/compliance demands stronger guarantees, or when you need confidence across whole user flows. They’re slower and heavier, so I’d add them when the architecture or risk profile calls for it.

**Where These Techniques Fit in Real Projects**

Equivalence + boundary testing is great for validation‑heavy apps and APIs. Negative testing is essential anywhere bad input can slip in (security, forms, imports). A controllable clock is a must for scheduling, billing, or anything time‑based. Mutation testing is worth it for critical code or when you need to prove test quality. Integration/contract tests are key in microservices or when teams depend on each other’s APIs.

**Mindset**

**Caution. I asked “how would this break?” and wrote tests for those edges—duplicates, max+1 lengths, and off‑by‑one dates. That kept me from over‑trusting the happy path.**

**Limiting bias. I tried to make tests fail first on tricky rules, then coded to make them pass. I also did small intentional breaks to prove the tests actually catch issues. Naming tests after the rule kept me focused on requirements, not my implementation.**

**Discipline and avoiding debt. Skipping validation tests always comes back as messy data or bug hunts. My plan: keep a Definition of Done with a branch‑coverage floor (e.g., ⟨set your % here⟩), run tests in CI on every commit, stabilize time and randomness, and add a regression test before changing behavior. Small habits now prevent big rewrites later.**

**Conclusion**

My test suite mirrors the requirements, hits boundaries and negatives, and stays deterministic. Coverage plus the deliberate break checks give me confidence the tests catch real problems, not just run code. As the project grows, I’d layer mutation testing and a few integration/contract tests for even more confidence.

**References (APA)**

Beck, K. (2002). Test-driven development: By example. Addison‑Wesley.

Gamma, E., & Beck, K. (2019). JUnit 5 user guide. The JUnit Team. https://junit.org/junit5/docs/current/user-guide/

ISTQB®. (2018). Glossary of testing terms (v3.2). International Software Testing Qualifications Board. https://glossary.istqb.org/

OWASP. (2023). Testing guide. Open Worldwide Application Security Project. https://owasp.org/www-project-web-security-testing-guide/