**CS 320 Project Two**

**Summary**

Over the course of Project One I built three small, in‑memory services: Contact, Task, and Appointment, and treated unit testing as a contractual proof that every item in the requirements list was honored. Each constraint (a ten‑character limit on IDs, immutability of IDs once assigned, or a prohibition on null values) became a JUnit test in two flavors: one that proves the rule holds when the input is valid and one that confirms the system fails safely when the input is not.

In ContactService my first test created a contact with an ID of “AB123,” stored it, and retrieved it to verify persistence. Immediately after, another test tried an eleven‑character value, asserting that the constructor threw an IllegalArgumentException. The pair of tests embodies the broader pattern: success paths ensure business logic behaves; failure paths confirm that the same logic cannot be bypassed. TaskService and AppointmentService followed the same rhythm, including date‑related edge‑case tests that attempted to schedule appointments in the past and text‑length tests that fed fifty‑one‑character descriptions to the task constructor.

Because every requirement appeared in at least two tests, coverage climbed above 80 percent—a figure verified with IntelliJ’s built‑in coverage meter. Beyond raw coverage, the tests share a uniform, self‑documenting naming scheme (methodUnderTest\_scenario\_expectedOutcome). That consistency makes future failures easy to trace because the broken rule is spelled out in the test header itself.

Writing the tests reminded me that “technically sound” does not just mean passing but also clear. I avoided magic numbers by declaring constants like MAX\_ID\_LENGTH = 10 at the top of each suite, then referencing them in both production and test code. Efficiency came from isolating state: I spun up fresh service instances in the @BeforeEach block so no test depended on data left behind by a previous run.

**Reflection**

***Testing techniques employed:***

Black‑box testing: Focused on inputs and outputs without peeking at implementation details.

Boundary‑value testing: IDs at exactly ten characters and at eleven, descriptions at fifty and fifty‑one, phone numbers at ten and eleven digits.

Exception testing: Nulls, past dates, and jumbo strings all had to raise explicit errors.

***Techniques not applied (and why):***

Integration testing: Everything lives in memory and calls no external resources, so nothing to integrate yet.

Exploratory testing: Scope was small and requirements unambiguous, leaving little room for free‑form discovery.

Formal regression testing: The same unit suite doubles as the regression net; no separate harness needed for now.

These omissions are contextual, not permanent. As soon as a persistence layer or REST controller appears, integration tests that span repositories, controllers, and services will become essential. Likewise, a future UI will benefit from exploratory sessions to uncover usability quirks.

***Practical implications:***

Techniques map to risk profiles. Boundary checks and exception assertions are vital for any application that consumes user‑supplied data. Large microservice ecosystems depend more on integration suites that verify messages or HTTP payloads traveling between bounded contexts. Matching the technique to the highest‑risk area ensures testing time is spent where it matters most.

**Mindset**

I approached testing with constructive paranoia: assume the code is broken until proven otherwise. That stance led me to simulate “abusive” user behavior early; null inputs, absurdly long strings, duplicate IDs; to see where seams might split. Bias was another danger: because I also wrote the production code, I risked crafting tests that merely confirmed my own logic. To blunt that tendency, I paired every positive test with a negative counterpart. Discipline tied it together. Even when constraints repeated across classes, I resisted copy‑and‑paste testing. Each service received its own independent tests, shielding us from hidden coupling and future technical debt.

**Conclusion**

Project One taught me that unit tests aren’t just a box to check, they’re a roadmap for reliable code. By turning each requirement into both a “works-as-designed” test and a “fails-when-it-should” test, and by keeping tests small and self‑contained, I ended up with a suite I actually trust. Clear naming, fresh state for every run, and a bit of healthy skepticism mean these tests will keep paying dividends as we add persistence, UIs, or whatever comes next. In short, good unit tests are less about hitting coverage numbers and more about preventing surprises when real people use the software.