**Project Two**

**Summary**

For Contacts, I treated the service as a stateful repository keyed by an immutable ID> I wrote tests to confirm the happy path which is creating a valid *Contact*, adding it once, updating mutable fields, and deleting by ID, and then pressed on edge behavior: duplicate additions must be rejected by the service, updates must never rekey or change the contact ID, and field validations must fire on constructor and on setters. I exercised boundary values for name, phone, and address, including digit-only rules for phone and maximum lengths enforced by your *Validation* logic.

For Tasks, I followed a similar pattern but emphasized the strict length boundaries that your model imposes. I validated that *Task* accepts an ID of up to 10 characters, a name up to 20, and a description up to 50. Tests prove that creation fails when any field is null, blank, or exceeds those limits; that service updates apply only to *name* and *description* while the ID remains immutable; and that adding a duplicate ID is rejected by *putIfAbsent*. I used parameterized tests to compact many invalid cases without slowing the suite or duplicating code.

For Appointments, I focused on temporal correctness and uniqueness. Because the model uses java.util.Date, I avoided flakiness by creating dates relative to “now” with offsets rather than relying on the current instant. Tests confirm that past dates are rejected during construction, that description length stays within 1-50, that IDs are 1-10 characters and immutable, and that the service enforces uniqueness and predictable delete semantics.

**ID immutability and no re-key during update**

A computer code with text

AI-generated content may be incorrect.

**Phone rules for Contact (digits only, exactly 10)**

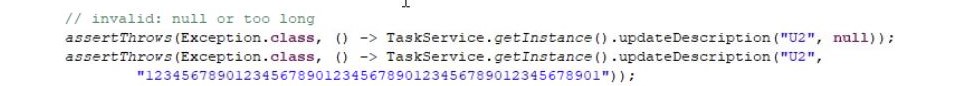
A computer code on a white background

AI-generated content may be incorrect.

These are some examples that show the tests directly verify the behaviors the requirements call for like uniqueness and immutability of IDs.

My experience writing tests began with the requirements, not from the implementation. That helped me identify the high-risk behaviors first, their uniqueness, immutability, and temporal rules and encode them into concise tests that either pass cleanly on the happy path or fail big on violations. The most delicate part was date handling; by consistently using “now + offset” for appointments and avoiding sleeps, the tests remain determined and fast. Parametrized tests paid off for Tasks, where there are many ways to violate string constraints; compressing those into tight tables kept the suite readable and maintainable.

I verified that validation truly fires at construction and during updates rather than trusting field setters implicitly:



I also asserted that uniqueness occurs at the service layer using *putIfAbsent*, not merely at the model layer:

A computer code with text

AI-generated content may be incorrect.

Finally, I confirmed that ID immutability is preserved across updates, since there is no API to change the key and the service never re-keys existing entries:

A computer code with text

AI-generated content may be incorrect.

The services use *ConcurrentHashMap* with *putIfAbsent* and *remove(id)*, so operations are keyed and constant time in practice. Tests are deterministic, no sleep or timing loops, keeping feedback fast.

A computer code with text

AI-generated content may be incorrect.

**Reflection**

I applied equivalence partitioning to divide inputs into valid and invalid classes for each field, then used boundary value analysis to probe edges (like 20 vs 21 characters for a task’s name and 50 vs 51 for descriptions). I leaned on negative and exception testing to exercise defensive paths such as duplicate IDs, null or blank strings, invalid phone formats, and past dates. Because these services are stateful repositories keyed by immutable IDs, I also used state-based testing to verify the map’s contents before and after add, update, and delete operations. Parameterized tests let me cover many invalid string variants compactly while keeping the suite fast and maintainable. Together, these techniques strike a pragmatic balance: they uncover common real-world failures with relatively few tests.

I did not run mutation testing, but it would be an excellent follow-up for these modules because mutating or removing a *Validation* check should immediately flip several tests from green to red. Property-based testing would be useful for generating random strings near length boundaries and for asserting global invariants such as IDs remain unique across many operations. Fuzz testing is more relevant to parsers and input protocols than to these domain objects, and performance or lead testing belongs at integration or system levels once persistence and I/O come into play. End-to-end and UI testing would be essential in the mobile application context but are out of scope for the course.

I assumed that simple code could hide complicated cases. Dates are a classic example: even with *javal.util.Date*, it is easy to let a past timestamp slip through, so I explicitly created both past and future cases to prove the guardrails work. Uniqueness and immutability are another: I verified ID integrity not only at creation but also across update, ensuring the service never re-keys or silently change identity. To limit bias, I wrote tests that I expected to fail, like null inputs, long strings, duplicate IDs, so I wouldn’t only confirm the correct code. There is a sort of confirmation biased when creating the code. But as we learned throughout the course, if we are the testers, we check every detail that is required to make sure the code in on par with the requirements, but if we have to write the code only, we assume it is correct because we know the requirements, but might miss certain details that the test team can and will discover as testing and deliberate adverse cases are added to fully test and validate the code.

Cutting corners will undoubtedly convert to hotfixes, manual debugging, and refactoring. I tried to keep tests short, readable, and fast so they can run on each change. I favor boundaries and negative cases because they can catch the most defects per line of test code. When the stakes rise, such as in modules that touch money, privacy, or safety, I would add mutation testing to prove the suite’s strength. My plan to avoid technical debt is to keep validations centralized and explicit, preserve the keyed access pattern in services, maintain deterministic tests (like no sleeps, no waiting on loops), and run the suite often to support safe refactoring. I would not wish to be part negatively in history of a team that cut corners in software development and testing that cost millions of dollars or worse, lives.

**References**

Vladimir Khorikov, & Roy Osherove. (2024). *The Art of Unit Testing, Third Edition*. Manning Publications.

Vocke, H. (n.d.). *The practical test pyramid*. martinfowler.com. https://martinfowler.com/articles/practical-test-pyramid.html