Project One Summary and Reflection

My unit testing approach for the contact, task, and appointment features stayed very close to the written requirements. I treated each requirement like a small contract, then wrote tests that proved the contract held for both valid and invalid input. For contacts, the requirements set hard limits on field lengths and did not allow nulls. My tests created a valid Contact and checked that getters returned the exact values that were passed in. Then I wrote negative tests that tried to break every rule. A simple example from my ContactTest shows the pattern I used throughout the project:

```
@Test
public void testValidContactCreation() {
  Contact contact = new Contact('001”, “John”, “Doe”, “1234567890”, “123 Main St”);
  assertEquals(“001”, contact.getContactId());
  assertEquals(“John”, contact.getFirstName());
  assertEquals(“Doe”, contact.getLastName());
  assertEquals(“1234567890”, contact.getPhone());
  assertEquals(“123 Main St”, contact.getAddress());
}

@Test
public void testInvalidContactId() {
  assertThrows(IllegalArgumentException.class, () -> {
  new Contact(null, “John”, “Doe”, “1234567890”, “123 Main St”);
  });
}
```

This pair shows how I aligned the tests to the requirement. The first test confirms that a valid object is accepted and stored correctly. The second proves that a null id is rejected as required. I repeated the same idea for too long names, short or long phone numbers, and empty addresses, always checking both the happy path and the failure path. For the task and appointment features I followed the same structure. For tasks I verified that name and description limits were enforced and that updates respected those same limits. For appointments I focused on immutability of the appointment id, non null fields, and the date rule that prevents invalid or past entries, then I confirmed that updates only change allowed fields.

I used coverage results to defend test quality. My tests executed constructors, all public getters and setters, and the service methods for add, delete, and update. I wrote both success and failure tests for each branch, so exception paths were exercised too. The result was high method and line coverage across all three features, with coverage staying above ninety percent and exception branches included. More important than the number, the suite covered every requirement statement with at least one positive and one negative test. When I added a check or saw a new branch in the code, I made sure there was a test that would hit it.

Writing the Junit tests taught me to keep things technical and simple. I relied on assertEquals, assertTrue and assertFalse for normal behavior, and assertThrows for invalid input. These small focused assertions kept the tests readable and forced me to think about a single outcome at a time. For example, when testing updates in the service classes, I first added a record, asserted that it existed, performed the update, then asserted that the change took effect and that unrelated fields did not change. A typical pattern looked like this:

```
TaskService service = new TaskService();
Task t = new Task(“T1”, “Pay bills”, “Monthly bills due”);
Service.addTask(t);
assertEquals(“Pay bills”, service.getTask(“T1”).getName());

service.updateTaskName(“T1”, “Pay rent”);
assertEquals(“Pay rent”, service.getTask(“T1”).getName());

assertThrows(IllegalArgumentException.class, () -> {
  service.updateTaskName(“T1”, null);
  });
```

These lines show technical soundness because they isolate one behavior per assertion, use the public API only, and exercise both valid and invalid paths that map cleanly to the written rules. They also show efficiency, because each test sets up only what it needs, uses simple in memory data, and tears down automatically since there are no external resources. I kept object creation small, I avoided duplication by reusing a local service instance inside a single test, and I avoided any I O or time based checks. The suite runs fast and produces clear failures when a rule is broken.

I also hit a few practical issues and folded those lessons into my approach. Early on I fixed a pom.xml header problem and a Maven path issue on Windows. Running tests inside Eclipse solved the path problem, and that kept my feedback loop short. When I moved to the service classes, I ran into a case where duplicate ids were not rejected. The test caught it first, which reminded me to always write the failure case even when I feel confident about the code. These little bumps shaped my habits. Fail first when it is cheap, then make the code pass in a clean way.

The techniques I used most were requirement based testing, boundary value analysis, equivalence partitioning, and negative testing with exceptions. Requirement based testing kept me honest. I took each sentence in the spec and turned it into one or more tests. Boundary value analysis drove the string length checks for names, phone numbers, and descriptions. I tested exact limits, one under, and one over to make sure the edges behaved correctly. Equivalence partitioning helped me avoid over testing. Once I proved that any string longer than the limit fails, I did not need to try ten more random long strings. Negative testing with assertThrows was the backbone for all the invalid cases. It confirmed that the code enforced rules by throwing the right exception when given a null, an empty sting, an over length string, or an invalid date.

There were several techniques I did not use. I did not use mocks or stubs because these services stayed in memory and had no external dependencies. I did not use parameterized tests, although they would have been a clean way to express length based boundaries across many inputs. I did not use mutation testing, which could have measured how well my suite detects small code changes. I also did not write integration or system tests, since the scope here was unit level. Each of these has a place. Parameterized tests are useful when the same logic repeats with many inputs. Mutation testing is valuable when you want deep confidence that tests will break if the code changes in subtle ways. Integration tests matter when services talk to databases or web APIs, and mocks help isolate those boundaries.

These were several techniques I did not use. I did not use mocks or stubs because these services stayed in memory and had no external dependencies. I did not ue parameterized tests, although they would have been a clean way to express length based boundaries across many inputs. I did not use mutation testing, which could have measured how well my suite detects small code changes. I also did not write integration or system tests, since the scope here was unit level. Each of these has a place. Parameterized tests are useful when the same logic repeats with many inputs. Mutation testing is valuable when you want deep confidence that tests will break if the code changes in subtle ways. Integration tests matter when services talk to databases or web APIs, and mocks help isolate those boundaries.

In practical terms, requirement based and boundary focused tests work well for data model and service layers like these, where rules are clear and side effects are small. Parameterized tests shine when the same rule applies to many cases, such as validating many phone number formats. Mutation testing is a good fit when quality needs are high and the team wants to catch logic gaps that simple coverage cannot reveal. Integration tests matter when the service layer grows to include persistence or messaging. Picking a mix depends on risk, cost, and the shape of the code.

My mindset through this project was cautious and curious. I tried to assume that any unchecked path could hide a defect, so I looked for edges and strange inputs. When I tested the services, I paing attention to how objects flowed through add, update, and delete, because a small mistake there can ripple across the rest of the app. One example was the duplicate id behavior in the services. It is easy to think add will always be called with a new id, but in real life users click buttons more than once or two requests race each other. Writing the duplicate id failure test forced me to think about that relationship and fix it before it reached the user interface. Another example was appointments. Valid dates sound simple until you remember time zones, past versus future, and different date formats. Keeping the checks simple and writing tests that prove the rule helped me respect the complexity without overbuilding.

Limiting bias took real effort. When I write code, I naturally believe it works because I know what I meant to write. To fight that, I wrote tests that assumed nothing, then I tried to break my own assumptions. I wrote the failure test before finishing the implementation whenever I could. I also read my tests out lout in plain language. For example, update name should change only the name, not any other field. That little habit caught a case where a setter touched the wrong property. If I were the only person testing my own code in a production setting, bias would be a concern. I would want another engineer to review my tests, I would add mutation testing to challenge the suite, and I would require code coverage thresholds in continuous integration so that I cannot skip the hard parts.

Staying disciplined about quality matters because cutting corners always leaves a bill to pay later. In small projects the bill looks like rework and late nights. In bigger systems it turns into outages and lost trust. My plan to avoid technical debt starts with habits that scale. I will keep tests small, fast, and tied to requirements. I will keep code simple and readable, because clear code is easier to test and maintain. I will add coverage checks in the pipeline, not because coverage is perfect, but because it keeps attention on untested code. I will use parameterized tests where they fit, I will add mutation testing when the risk is high, and I will pull in integration tests as soon as the code reaches across a boundary. Finally, I will keep the feedback loop short. The day I fixed the Maven path and ran tests directly in Eclipse was a good reminder that fast feedback beats guesswork.

This project pulled together the basics of unit testing and the mindset behind it. I wrote tests that matched the rules, I pushed on edges, and I learned to slow down at the right moments. The experience with real roadblocks, from a broken build setup to a missed duplicate rule, made the lessons stick. The result is a test suite that supports the code instead of getting in the way, and a clearer plan for how I will carry these habits into the rest of my work.

References

• Black, R., van Veenendaal, E., & Graham, D. (2020). Foundations of software testing ISTQB certification. Cengage Learning

• García, B. (2017). Mastering software testing with JUnit 5. Packt Publishing.