In Project One, my approach to unit testing revolved around validating the core functionality of three major application features: the Appointment, Contact, and Task services. Each unit test suite was designed with a strong emphasis on the associated software requirements, ensuring correctness, reliability, and resilience under edge-case scenarios.

For the Appointment feature, I wrote unit tests that validated the creation of appointments with appropriate IDs, dates, and descriptions. The AppointmentTest.java file uses parameterized tests to assert validity across a range of acceptable ID inputs. The testing approach adhered strictly to the requirement that appointment IDs must not exceed 10 characters and that the appointment date must not be in the past. As emphasized by Garcia (2017), "unit tests are most powerful when they precisely target the boundaries of application behavior." The following test validates the proper construction of appointment objects:

@ParameterizedTest

@ValueSource(strings = { "0", "01", "0123456789" })

void testCreateAppointmentWithValidId(String testId) {

assertTrue(new Appointment(testId, testApptDate, "description").getAppointmentId().equals(testId));

}

The Contact feature followed a similar structure. Each unit test ensured all five required fields ID, first name, last name, phone number, and address and were validated against null or invalid input. Requirements were enforced by testing for both length and null constraints. According to Hambling et al. (2015), “validation ensures the software meets the needs of stakeholders and handles all potential usage scenarios.” For example, one parameterized test case in ContactTest.java ensures that invalid phone numbers shorter or longer than 10 digits are not accepted. These tests directly aligned with the constraints outlined in the functional requirements, such as:

@ParameterizedTest

@ValueSource(strings = { "1234567890" })

void testCreateContactWithValidPhone(String phone) {

assertEquals(phone, new Contact("1", "fname", "lname", phone, "address").getPhone());

}

With the Task feature, the testing strategy involved verifying unique task IDs, mandatory names, and descriptions. As seen in TaskServiceTest.java, tests such as testAddTaskThatAlreadyExists check for service logic that prevents duplicate tasks from being added. This aligns with guidance from TutorialsPoint (n.d.), which states that “unit tests should guard against redundancy and unintended side effects.” The following snippet checks the effectiveness of the add method and ensures no redundancy:

@Test

void testAddTaskThatAlreadyExists() {

assertFalse(testTaskService.addTask(new Task("1", "Duplicate", "Test")));

}

All test suites were designed to maintain high test coverage. I verified effectiveness using both visual inspection and Eclipse’s built-in coverage analysis tool. My coverage results exceeded 90% across all three service layers, a strong indicator that my tests adequately exercised the system under test.

To ensure the code was technically sound, I implemented comprehensive edge case checks. For instance, null and empty string scenarios were thoroughly tested using:

@ParameterizedTest

@NullSource

@ValueSource(strings = { "", " " })

void testCreateTaskWithInvalidNullOrEmptyId(String testId) {

Exception exception = assertThrows(IllegalArgumentException.class, () -> new Task(testId, "name", "description"));

}

This confirms that invalid inputs trigger appropriate exception handling, aligning with Java best practices for defensive programming (Garcia, 2017). Efficiency was also a key factor as each test method is concise, leveraging the JUnit 5 framework’s parameterized tests to minimize redundancy and keep execution fast.

Throughout the project, I primarily employed black-box testing and boundary value testing techniques. Black-box testing allowed me to validate application behavior without needing to delve into internal implementation. Hambling et al. (2015) support this technique by stating, “black-box testing allows testers to focus on input/output validation rather than internal code paths.” For example, I tested how the system reacted to invalid input formats, such as malformed IDs or null values, ensuring that constraints were enforced as specified.

Boundary value testing was essential for ensuring compliance with field length restrictions. By testing the exact length, I was able to confidently assert that the application adhered to requirement specifications under edge conditions. As noted by TutorialsPoint (n.d.), “boundary testing is a critical technique in determining how systems handle edge cases and input extremes.”

In larger or more complex systems, these additional techniques become especially important. For example, white-box testing is ideal when working with financial systems requiring precise conditional logic, while equivalence partitioning is valuable when validating forms with many fields and input rules.

Throughout this project, I adopted a more cautious and meticulous mindset. I treated testing not as a checkbox, but as an opportunity to improve reliability and prevent future issues. This mindset was especially important because even a single overlooked bug in an appointment or task scheduler could compromise data integrity. As Coder Academy (n.d.) aptly states, “when coding goes wrong, the smallest bug can derail a project.”

To avoid bias, I made an effort to test all code, including areas I had previously written and assumed to be correct. I approached the task with a critical eye and expected failure where appropriate. For example, one might assume that a date parser will always succeed and neglect to test parsing exceptions as an oversight that can easily be avoided with rigorous unit testing.

Finally, I learned the importance of being disciplined about quality. Cutting corners in testing may seem to save time, but it usually leads to technical debt. Hambling et al. (2015) warn that “untested software accumulates defects that cost more to fix later than to prevent initially.” In this project, ensuring that all service classes passed strict unit tests before integration prevented much larger issues.

Going forward, I plan to manage technical debt by enforcing code reviews, automated test coverage thresholds, and consistent use of CI/CD pipelines with test gates. For example, before merging a pull request, I will require that all unit tests pass and at least 90% code coverage be maintained.

References

Garcia, B. (2017). *Mastering software testing with JUnit 5: Comprehensive guide to develop high quality Java applications*. eBook Collection (EBSCOhost).

Hambling, B., Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2015). *Software testing: An ISTQB-BCS certified tester foundation guide* (4th ed.). BCS, The Chartered Institute for IT.

TutorialsPoint. (n.d.). *JUnit - Test framework*. TutorialsPoint. <https://www.tutorialspoint.com/junit/junit_test_framework.htm>

Coder Academy. (n.d.). *When coding goes wrong*. Medium. <https://medium.com/@coderacademy/when-coding-goes-wrong-e46d84c6565f>