In Project One, I developed three main features: Contact, Task, and Appointment services. For each feature, I employed unit testing via JUnit to verify that the software requirements were fulfilled. For instance, the TaskTest class checks for various constraints on task IDs, names, and descriptions. The tests ensured that the character length constraints and null inputs raised appropriate exceptions.

The testing approach was closely aligned with the software requirements. For example, in the TaskTest, I verified that any task ID exceeding 10 characters throws an IllegalArgumentException, as required by the specification:

void testTaskIDTooLong() {

Assertions.assertThrows(IllegalArgumentException.class, () -> {

new Task(elevenChar, tenChar, tenChar);

});

}

This demonstrates how my tests were tailored to specific functional constraints, such as the length of input fields.

The JUnit tests were highly effective, as demonstrated by the high code coverage achieved. For example, the Task tests covered edge cases (too long, null) and ensured the proper functioning of all methods, yielding a coverage of around 100%. This high coverage percentage indicates that the tests covered most of the critical code paths.

Writing the tests was straightforward but required caution to ensure complete coverage. For example, in the ContactTest class, I ensured that invalid inputs, such as a contact phone number with fewer than 10 digits, correctly triggered exceptions:

void testContactphoneTooShort() {

Assertions.assertThrows(IllegalArgumentException.class, () -> {

new Contact(tenChar, tenChar, tenChar, nineChar, tenChar);

});

}

This ensured that the code adhered to both functional and non-functional requirements.

To ensure soundness, I employed checks on null and incorrect values, such as:

if (ID == null || ID.length() > 10) {

throw new IllegalArgumentException("Invalid id");

}

Additionally, I avoided unnecessary operations in tests and focused on key failure points to keep the tests efficient.

To ensure that the code and tests were efficient, I focused on minimizing redundant operations and writing concise, focused tests. For example, when checking for invalid input in the TaskTest class, I avoided unnecessary instantiation of objects and reused variable values when appropriate:

void testTask() {

Task task = new Task(tenChar, tenChar, tenChar);

assertTrue(task.getID().equals(tenChar));

assertTrue(task.getname().equals(tenChar));

assertTrue(task.getdescription().equals(tenChar));

}

By using variables like tenChar for repeated values, I ensured that the test was not creating unnecessary new strings, reducing memory usage and improving the test's overall execution speed.

The main testing techniques employed were unit testing and boundary testing. Unit testing allowed me to validate individual methods' behavior, such as ensuring that null inputs and overly long strings raised appropriate exceptions. Boundary testing helped verify edge cases, such as input lengths exactly at or exceeding the maximum allowed values.

I did not use integration testing or system testing in this project. Integration testing could be valuable for checking how multiple modules interact, while system testing would assess the behavior of the entire application in a real-world scenario. These could be useful in larger-scale projects where components interact more frequently.

**Unit Testing**: Unit testing is ideal for verifying individual components in isolation, ensuring each works as expected. It helps catch bugs early but doesn't cover how components interact in a larger system.

**Boundary Testing**: Boundary testing ensures software handles edge cases correctly, like input limits. It's useful for validating user.

**Integration Testing**: Integration testing checks that multiple modules work together as intended. It's critical for detecting issues between components but can be complex to set up.

**System Testing**: System testing validates the entire system's behavior in a production-like environment. It provides full confidence before deployment but is resource-intensive and time-consuming.

Throughout this project, I approached testing with caution, especially when checking the relationships between inputs and constraints. It was essential to understand the impact of interdependencies, such as ensuring that a null Task ID not only failed the test but also didn’t cause other errors. For example, in the appointment tests, ensuring that an invalid date triggered the correct response was crucial:

void testAppointmentdateIsPast() {

Assertions.assertThrows(IllegalArgumentException.class, () -> {

new Appointment(tenChar, pastDate, tenChar);

});

}

I tried to limit bias by considering how another developer might misunderstand or misuse the code. This approach ensured that my tests were not overly optimistic or tailored to ideal inputs only. Had I been testing my own code without considering potential misuses, I might have missed edge cases like invalid date inputs in the Appointment class.

Quality is paramount in software engineering. Cutting corners can lead to technical debt, which is difficult to address later in the development lifecycle. To avoid technical debt, I plan to continue rigorous testing practices, always writing comprehensive unit tests before moving on to integration tests. For instance, ensuring that tasks are adequately validated at the service level helps prevent issues down the line, as illustrated by the TaskServiceTest:

void testAddDuplicateTask() {

TaskService taskService = new TaskService();

Task task = new Task(tenChar, tenChar, tenChar);

taskService.addTask(task);

assertFalse(taskService.addTask(task)); // Duplicate ID should return false

}

By consistently following these practices, I can avoid accumulating technical debt and maintain a high standard of code quality.