Caesar Yepiz

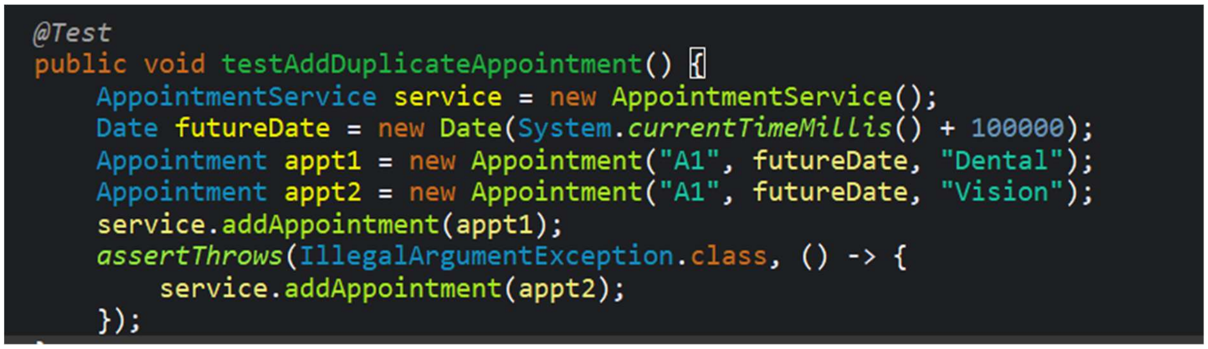
10/19/2025

CS 320

Professor Handlos

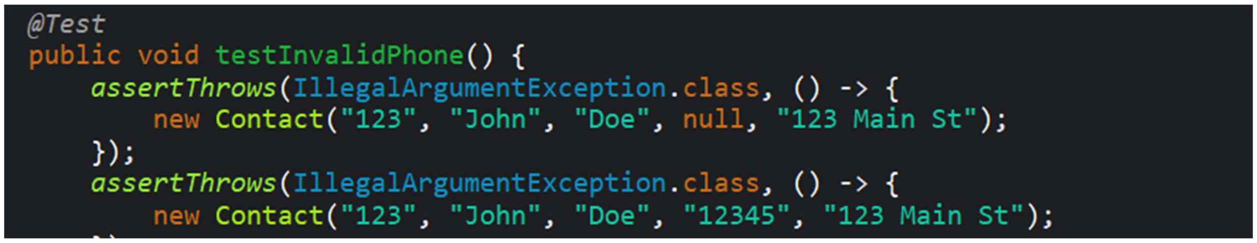
Summary

Throughout Project One, I developed and tested the contact, task, and appointment services for a mobile application. My unit testing approach focused on validating object creation, enforcing input constraints, and confirming service-level operations such as adding, deleting, and updating records. By using my experience in retail work, I made sure that nothing was left out and I was thorough. For the Contact feature, I tested constructor validation for all fields including null values and maximum lengths. I also verified that setter methods correctly enforced constraints, such as rejecting phone numbers that were not exactly ten digits. For the Task feature, I ensured that tasks could be created, updated, and deleted, and that invalid inputs such as overly long names or descriptions triggered exceptions. The Appointment feature followed a similar pattern, with tests confirming that appointments could be added and deleted, and that duplicate IDs or missing entries were handled appropriately as can be seen here:



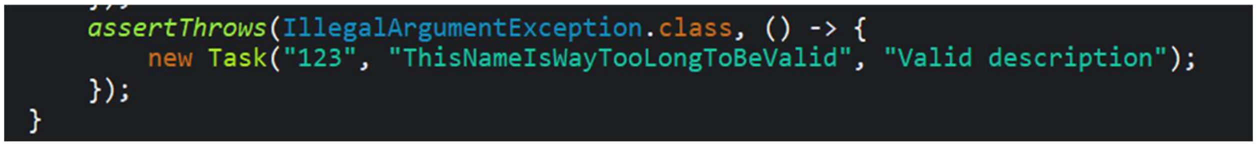
```
@Test
public void testAddDuplicateAppointment() {
    AppointmentService service = new AppointmentService();
    Date futureDate = new Date(System.currentTimeMillis() + 100000);
    Appointment appt1 = new Appointment("A1", futureDate, "Dental");
    Appointment appt2 = new Appointment("A1", futureDate, "Vision");
    service.addAppointment(appt1);
    assertThrows(IllegalArgumentException.class, () -> {
        service.addAppointment(appt2);
    });
```

My testing strategy was closely aligned with the software requirements. For example, the Contact class required that phone numbers be exactly ten digits. I implemented this in both the constructor and setter tests using assertions like



```
@Test
public void testInvalidPhone() {
    assertThrows(IllegalArgumentException.class, () -> {
        new Contact("123", "John", "Doe", null, "123 Main St");
    });
    assertThrows(IllegalArgumentException.class, () -> {
        new Contact("123", "John", "Doe", "12345", "123 Main St");
```

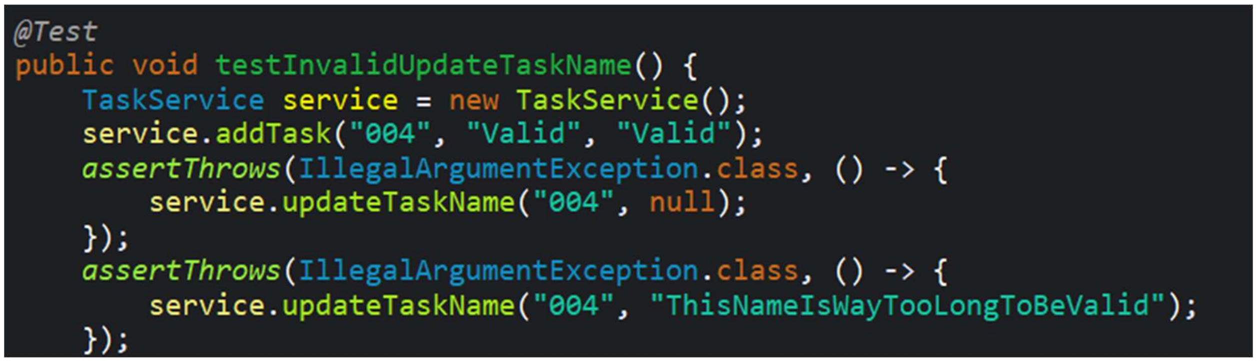
Similarly, the Task class required name and description length limits, which I validated with tests such as



```
    assertThrows(IllegalArgumentException.class, () -> {
        new Task("123", "ThisNameIsWayTooLongToBeValid", "Valid description");
    });
}
```

These tests directly reflect the constraints defined in the project specifications and demonstrate that my approach was grounded in the actual requirements.

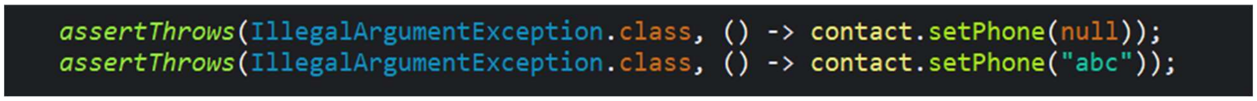
The overall quality of my JUnit tests was strong, with coverage extending across all public methods and validation logic. I tested both positive and negative scenarios, including valid object creation, field updates, duplicate ID handling, null inputs, and boundary values. Based on manual review and IDE coverage tools, my tests achieved near-complete coverage. For example, in TaskServiceTest, I tested both successful updates and failure cases using assertions like



```
@Test
public void testInvalidUpdateTaskName() {
    TaskService service = new TaskService();
    service.addTask("004", "Valid", "Valid");
    assertThrows(IllegalArgumentException.class, () -> {
        service.updateTaskName("004", null);
    });
    assertThrows(IllegalArgumentException.class, () -> {
        service.updateTaskName("004", "ThisNameIsWayTooLongToBeValid");
    });
```

which confirmed that the service correctly delegated validation to the model class.

Writing JUnit tests helped me think critically about edge cases and user behavior. I started with basic functionality, then expanded to cover invalid inputs and boundary conditions. I used assertThrows to confirm that exceptions were thrown as expected, and assertEquals and assertNull to verify state changes. I ensured technical soundness by isolating each test case and using clear assertions, such as assertEquals("Updated Name", service.getTask("001").getName()), which confirmed that the update method worked and that the object state was correctly modified. I also kept my tests efficient by avoiding redundancy and grouping related assertions. For example, in ContactTest, I tested multiple invalid inputs in one method:



```
    assertThrows(IllegalArgumentException.class, () -> contact.setPhone(null));
    assertThrows(IllegalArgumentException.class, () -> contact.setPhone("abc"));
```

This approach reduced repetition while maintaining clarity.

Reflection

In terms of testing techniques, I primarily used unit testing, boundary testing, and negative testing. Unit testing allowed me to isolate individual methods and verify their behavior. Boundary testing helped confirm that inputs at the edge of acceptable ranges, such as ten-character IDs or fifty-character descriptions, were handled correctly. Negative testing ensured that invalid inputs triggered exceptions, which was critical for validating the robustness of my model classes. I did not use integration testing or mocking frameworks like Mockito, as these techniques are more appropriate for testing interactions between multiple components or external dependencies, which were not part of this project. I also did not use automated user interface testing, since the project focused on back-end services.

Each technique has practical uses depending on the scale and complexity of the project. Unit testing is ideal for verifying logic in isolated classes, such as model validation and service operations. Boundary testing is useful for validating input constraints, especially in user-facing applications. Integration testing becomes essential when services interact with databases, APIs, or other modules. Mocking helps simulate external systems and isolate test behavior in complex environments. These techniques are tools that I plan to use selectively based on the needs of future projects.

I approached this project with a cautious and detail-oriented mindset. I knew that missing a single validation could lead to runtime errors or data corruption. For example, failing to test null phone numbers would have left a gap in my Contact validation logic. I appreciated the complexity of interrelated code, especially how service classes relied on model validation, and made sure that each service method delegated correctly and that exceptions were thrown when expected. To limit bias, I reviewed my own code critically and tested both success and failure paths. I did not assume that my constructors or setters were flawless. I wrote tests to prove it. For example, I tested invalid updates even after confirming that valid updates worked, using assertions like assertThrows(IllegalArgumentException.class, () -> contact.setFirstName(null)). Bias can be a concern when developers test their own code, as it is easy to overlook edge cases or assume correctness. I countered this by writing tests that intentionally broke the rules.

Being disciplined about quality is essential in software engineering. Cutting corners in testing leads to technical debt, which can snowball into bugs, rework, and lost trust. I avoided shortcuts by writing complete test suites and validating every field. To prevent technical debt, I plan to write tests alongside code, use coverage tools to identify gaps, refactor and simplify logic when needed, and document assumptions and constraints clearly. This project reinforced the value of thorough testing and the mindset required to deliver reliable software. I now feel more confident in my ability to build scalable, testable systems and apply these principles in real-world development.