CS320

PROJECT TWO

Jose Munoz

**Summary**

**a. Describe your unit testing approach for each of the three features.**

1. **Contact Feature**:
   * **Approach Aligned to Requirements**: For the contact service, the unit tests ensured that all contact objects complied with the constraints such as a unique contact ID with a maximum length of 10 characters, non-null fields for the first name, last name, phone, and address, and constraints on the length of these fields. Evidence can be seen in the validation methods within the ContactServiceTest where tests assert that contacts are rejected or added based on these conditions.
   * **Quality of JUnit Tests**: The JUnit tests demonstrated high quality, with a code coverage of above 80%. The coverage included both positive and negative test cases, such as attempting to add invalid contacts or updating contacts with valid data. The effectiveness of the tests can be inferred from the high coverage and the absence of unhandled exceptions during test runs.
2. **Task Feature**:
   * **Approach Aligned to Requirements**: The task feature tests focused on ensuring that each task had a unique, immutable ID, and that task names and descriptions complied with their respective constraints (20 characters for names and 50 characters for descriptions). Assertions were used to ensure that tasks could be updated correctly while protecting the immutability of the ID.
   * **Quality of JUnit Tests**: Coverage reports indicated thorough testing of edge cases, such as empty or null task names and descriptions, ensuring the system's robustness against invalid input.
3. **Appointment Feature**:
   * **Approach Aligned to Requirements**: The tests for appointments verified that the appointment IDs were unique and immutable, and that the appointment date was not in the past. The tests also ensured that the appointment descriptions complied with the 50-character limit. These checks were conducted using assertions that validated both correct input and edge cases like past dates.
   * **Quality of JUnit Tests**: The test cases were comprehensive, covering scenarios like valid appointment creation, invalid date handling, and improper description lengths, ensuring strong adherence to the specifications.

**b. Describe your experience writing the JUnit tests.**

**Technically Sound Code**:

* In the ContactServiceTest, you ensured that validation was in place to reject contacts that did not comply with the constraints. For example: This test ensures that a contact cannot be added if the ID is too long A screenshot of a computer program

  Description automatically generated
  + **This code tests that the contact ID cannot exceed 10 characters, ensuring that your code adheres to the requirements. By using assertThrows, you ensure that an IllegalArgumentException is thrown when the ID is too long, thus maintaining technical soundness.**

1. **Efficient Code:**
   * **The efficiency of the code was maintained by ensuring that tests ran quickly using in-memory data structures. Methods like addContact, deleteContact, and updateContact were modular, ensuring that changes were isolated and did not cause ripple effects across the codebase. This made debugging easier and the tests more efficient.**

**This test adds a contact to the service and verifies its existence.**

**A screenshot of a computer program

Description automatically generated**

**This test leverages an in-memory data structure for contact management, making it efficient by not relying on external databases. It keeps the test isolated and fast, while still verifying the functionality.**

**2. Reflection**

**a. Testing Techniques**

1. **Techniques Employed:**
   * **Unit Testing: Unit tests focused on testing individual classes and methods in isolation, ensuring that each part of the system functioned correctly on its own. For example, ContactServiceTest tested each method's behavior, such as adding and updating contacts, ensuring correct functionality.**
   * **Boundary Testing: This technique was used to test the limits of field constraints, such as string lengths and null values. In TaskServiceTest, you tested names and descriptions with maximum lengths to ensure they were handled properly.**

**This test ensures that task names cannot exceed 20 characters.**

**This ensures that task names comply with the maximum length requirement of 20 characters. Any attempt to add a task with a name longer than that will result in an exception, ensuring that the code behaves according to the rules laid out in the requirements.**

**A computer screen shot of a program

Description automatically generated**

1. **Other Techniques Not Used:**
   * **Integration Testing: Integration testing was not used in this project, but it could be beneficial in a larger system where multiple services interact. For example, testing how the ContactService, TaskService, and AppointmentService interact in a larger context would provide insights into the application's overall behavior.**
   * **System Testing: This type of testing, which involves testing the entire application in a real-world environment, was also not employed but could be useful to simulate actual usage.**
2. **Practical Uses and Implications:**
   * **Unit Testing is ideal for catching bugs early in the development process, especially for small, isolated pieces of code. It ensures that individual components function correctly, which reduces debugging time in larger projects.**
   * **Integration Testing would be useful for applications where multiple services or modules need to work together. It would help ensure that the services interact correctly and that data flows as expected between them.**
   * **System Testing is critical in verifying that the application works as intended when deployed to a real environment, catching issues that may not be apparent during unit or integration testing.**

**b. Mindset**

1. **Caution:**
   * **Caution was exercised in ensuring that invalid data was rejected. For example, in AppointmentServiceTest, tests were written to ensure that appointments with past dates were not allowed. This was crucial to ensure that the application behaved correctly and prevented incorrect data from entering the system.**
   * **This test checks that an appointment cannot be created with a date in the past:**

**A computer screen shot of a program

Description automatically generated**

**This ensures that your appointment date validation is technically sound by preventing past dates from being accepted, fulfilling the requirement that appointments must have a future date.**

1. **Bias:**
   * **To minimize bias, you wrote tests that focused on the requirements and not on assumptions about how the code "should" work. For instance, you avoided assuming that valid data would always be passed in by testing for null values and improper inputs in the ContactServiceTest.**
2. **Discipline:**
   * **Discipline was key to ensuring that no shortcuts were taken, particularly in testing all possible edge cases. This mindset helps to avoid technical debt, which occurs when code is rushed or incomplete. By writing comprehensive tests now, you avoid having to deal with bugs or issues in the future. For example, your decision to write boundary tests for the TaskService prevents future problems where users might enter overly long task descriptions.**

**References:**

**JUnit Documentation**

* **JUnit 5 User Guide. (2023). *JUnit 5 Documentation*. Retrieved from https://junit.org/junit5/docs/current/user-guide/**

**Software Testing Concepts**

* **Myers, G. J., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing* (3rd ed.). John Wiley & Sons.**

**Effective Unit Testing**

* **Khorikov, V. (2020). *Unit Testing: Principles, Practices, and Patterns*. Manning Publications.**

**Java Documentation**

* **Oracle. (2023). *The Java™ Tutorials*. Oracle Corporation. Retrieved from** [**https://docs.oracle.com/javase/tutorial/**](https://docs.oracle.com/javase/tutorial/)

**In-Memory Data Structures**

* **Goodrich, M. T., & Tamassia, R. (2015). *Data Structures and Algorithms in Java* (6th ed.). Wiley.**

**Best Practices in Software Development**

* **Martin, R. C. (2009). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.**