Hong Luu

Professor Karl Lewis

CS 320

October 15, 2023

Project Two

**1. Summary:**

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

**i. Alignment with Software Requirements:**

In my testing approach, I made sure to align the test cases with the software requirements to ensure that the code functions correctly. Here’s how my approach corresponds to the requirements:

ContactTest.java:

I created test methods that cover various scenarios, such as creating a valid contact, attempting to create contacts with null values, and testing getters and setters.

These scenarios align with the requirement of ensuring that contact creation, validation, and access work correctly.

ContactServiceTest.java:

I designed test methods to cover adding, updating, deleting, and retrieving contacts, as well as handling exceptions like adding duplicate contacts. These test cases align with the requirement that the ContactService should perform these operations correctly and handle exceptions as specified.

TaskTest.java:

I included test methods that validate creating valid tasks, attempting to create tasks with invalid parameters (null values and long strings), in line with the requirement that tasks should be created correctly with validation checks.

TaskServiceTest.java:

My test methods address adding, deleting, and updating tasks, as well as handling exceptions and duplicates. These align with the requirements that TaskService should manage tasks correctly and handle errors appropriately.

AppointmentTest.java:

I created test methods to validate the creation of valid appointments and check for exceptions when using invalid appointment IDs, which align with the requirement that appointments should be created correctly with proper validation.

AppointmentServiceTest.java:

My test methods cover adding appointments, adding duplicates, deleting appointments, and handling exceptions, aligning with the requirement that AppointmentService should manage appointments correctly and handle errors as specified.

**ii. Quality of JUnit Tests:**

I consider the quality of my JUnit tests to be high for several reasons:

* Coverage: While I didn’t mention specific code coverage numbers, I aimed to cover various scenarios, including edge cases and exceptional conditions. My tests covered a wide range of code paths.
* Assertions: I used assertions effectively to validate that the expected outcomes matched the actual results. For example, in testInvalidContactID, I assert that an IllegalArgumentException is thrown when creating a contact with a null contactID.
* Exception Handling: I thoroughly tested exception scenarios to ensure that the code behaves correctly when faced with invalid inputs. Each exception scenario was tested with relevant test cases.
* Clear Naming and Descriptions: I provided clear and descriptive names and descriptions for each test case, making it easy to understand the purpose of the test.

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

**i. Ensuring Technical Soundness:**

I ensured technical soundness in my code by using JUnit’s assertThrows method to validate that exceptions are thrown when necessary. For example, in ContactTest, I made sure that creating a contact with a null contactID results in an IllegalArgumentException being thrown:

assertThrows(IllegalArgumentException.class, () -> new Contact(null, "John", "Doe", "1234567890", "123 Main St"));

**ii. Ensuring Code Efficiency:**

While my tests themselves do not contain code that raises concerns about efficiency, I can point out that I avoided redundant or unnecessary test cases. For instance, in the Task feature, I did not duplicate tests for similar update operations but rather ensured that a single test comprehensively covered those scenarios.

**2. Reflection:**

**a. Testing techniques:**

i. The software testing techniques employed in this project include:

* Boundary Value Analysis: I used this technique to test the edge cases of input boundaries. For example, I tested the Task feature by creating tasks with both very long and null values for task IDs and names.
* Equivalence Partitioning: I divided the input domain into equivalence classes and tested representative values from each class. For instance, in the Contact feature, I tested valid and invalid phone numbers and addresses to cover different partitions.
* Negative Testing: I extensively tested for scenarios where invalid or unexpected inputs were provided. This includes passing null values, excessively long values, and duplicate IDs, which can help identify vulnerabilities and error handling.
* Exception Testing: I tested exceptions to ensure that the code correctly raises exceptions when it should. For example, in the Contact and Task features, I tested scenarios where IllegalArgumentException should be thrown for invalid inputs.
* Functional Testing: I validated that the functions and methods in each feature performed their intended functions correctly. For instance, adding, updating, and deleting contacts, tasks, and appointments.

ii. Testing techniques not used in this project include:

* Performance Testing: This project primarily focused on functional correctness rather than performance. Performance testing, such as load testing or stress testing, was not necessary for the simple functionality implemented here.
* Usability Testing: Usability testing, which evaluates how user-friendly the software is, wasn’t applicable in this backend-focused project with no user interface.

iii. Practical Uses and Implications:

* Boundary Value Analysis: This technique is crucial for identifying off-by-one errors or boundary-related issues. In scenarios like financial software, testing boundary values is critical to ensure accurate calculations.
* Equivalence Partitioning: It helps to ensure comprehensive coverage of different input cases without testing every possible value. In applications with extensive user inputs, like e-commerce websites, equivalence partitioning can significantly reduce the number of test cases needed.
* Negative Testing: Negative testing uncovers vulnerabilities and security risks. In healthcare software, where data security is paramount, extensive negative testing is essential to prevent data breaches.
* Exception Testing: Exception testing ensures that error conditions are handled gracefully. In safety-critical systems like autonomous vehicles, it’s vital to test how the software reacts to unexpected situations.
* Functional Testing: Functional testing verifies that the software performs its intended functions correctly. This technique is valuable in virtually all software development projects, from financial systems to video games.

**b. Mindset:**

**i. Example of Caution:**

Let’s take the Contact feature as an example. In the testInvalidPhoneNumber method, I tested the scenario of creating a contact with an invalid phone number (not exactly 10 digits) to ensure that the software handles this case correctly. I employed caution by setting up the test case to represent this specific scenario and then verifying that the code throws the expected IllegalArgumentException. This demonstrates the meticulous approach to testing and attention to potential issues.

Understanding the complexity and how different parts of the code work together was crucial in testing. For example, when checking the Contact feature, we needed to know how functions like changing a first name might affect other things like getting the first name. It’s like making sure all the pieces of a puzzle fit together correctly. We also needed to be aware of how data is stored, as seen in the ContactService part, so we don’t accidentally mess up the data. We had to be careful with things like really long task names in the Task feature because they might cause problems. For some tasks, like updating a task with the wrong information, knowing how the code makes decisions was important. Finally, understanding all of this helped us make sure that when we made changes to the code, we didn’t accidentally break other parts of it. So, appreciating how everything in the code connects was super important to make sure it all works correctly.

**ii. Bias:**

When reviewing the code, I aimed to limit bias by approaching it objectively and systematically. To avoid personal preferences or preconceived notions, I followed established coding standards and guidelines, focusing on adherence to best practices and requirements outlined in the project’s specifications.

Bias can become a significant concern when developers are responsible for testing their own code. They may unconsciously overlook flaws or assume that their code is correct, leading to confirmation bias. For example, a developer might skip certain test cases because they believe those parts of the code are robust, even when there might be hidden issues.

**iii. The importance of being disciplined in your commitment to quality as a software engineering professional:**

Being disciplined in our commitment to quality as software engineering professionals is paramount. Cutting corners in code writing or testing can lead to technical debt, a situation where quick, suboptimal solutions accumulate over time, making future maintenance and development more challenging. To avoid technical debt, it’s essential to invest time in code quality from the beginning. For instance, by writing clear and well-documented code, we ensure that others (or even our future selves) can understand and modify it easily without introducing errors. Comprehensive testing, as demonstrated in our JUnit tests, helps catch issues early in the development process, reducing the likelihood of costly bug fixes in the future.