Braydon Woodward

CS-320

June 23, 2024

Project 2

**Summary:**

Unit Testing Approach

For the project involving the ContactService, TaskService, and AppointmentService, my unit testing approach was focused on ensuring comprehensive coverage and validating the correctness of each service.

1. **ContactService**:

* **Test Cases**: I developed test cases to validate the creation, retrieval, update, and deletion of contacts. This included testing valid and invalid inputs for each operation.
* **Specific Tests**: For instance, I wrote tests to ensure that duplicate contacts could not be created and that contacts could not be updated with invalid data.

1. **TaskService**:
   * **Test Cases**: Tests were created to ensure proper creation of tasks as well as the addition, update, and removal of tasks.
   * **Specific Tests**: For instance, I wrote tests to ensure that an error was thrown when trying to create a task with a null name or when trying to delete a nonexistent task.
2. **AppointmentService**:
   * **Test Cases**: I designed tests to validate the creation, addition, and removal of appointments.
   * **Specific Tests**: For instance, I wrote tests that ensured appointments could not be scheduled in the past and that duplicate appointments were correctly handled.

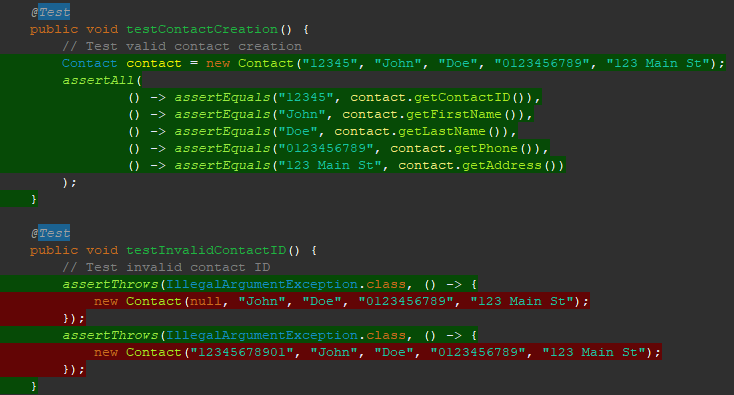
**Alignment with Software Requirements**

My approach was closely aligned with the software requirements. Tests were created directly from the requirements reference sheet, ensuring that each feature's critical functionalities were thoroughly tested. For instance, in ContactService, the requirement to prevent duplicate contacts was directly tested by attempting to add contacts with the same ID and verifying that an exception was thrown. In the AppointmentService test, an old date was created and directly passed into the appointment creation code to ensure no appointments could be scheduled for dates in the past and an exception was thrown. After tests were created and ran, I again checked that the tests were aligned with the requirements to ensure none were missed.

Defending Overall Quality of JUnit Tests

The overall quality of the JUnit tests is high, as indicated by an EclEmma coverage result of 82.6%. This suggests that a significant portion of the code was executed during testing, ensuring that most paths and conditions were verified. After a more detailed review of the results, I have determined that most, if not all, of the coverage “missed” by the test were within lambda expressions in assertThrows, which is a known limitation of the JaCoCo library used by EclEmma (Mountainminds GmbH & Co., et al. (2017)). This increases my confidence that the coverage is greater than the 82.6% result reported in the test.

Experience Writing JUnit Tests

Writing the JUnit tests was an insightful experience. I ensured the code was technically sound by validating various scenarios and edge cases. For example, in the ContactTest I efficiently tested creating a valid Contact.

The second test ensures efficiency by directly targeting the edge case of invalid contact IDs and verifying the exception inside the same method.

**Reflection:**

Testing Techniques

1. **Techniques Employed:**
   * **Unit Testing**: Focused on individual units of code to ensure they work as expected in isolation.
   * **Exception Testing**: Verified that methods correctly handle and throw exceptions under invalid conditions.
   * **Coverage Testing:** Used EclEmma to measure code coverage and ensure that a high percentage of the codebase was exercised during testing.
2. **Techniques Not Used:**
   * **Integration Testing**: Tests involving multiple components interacting together were not implemented.
   * **System Testing**: End-to-end testing of the entire application was not performed within this project scope.

Practical Uses and Implications

* + **Unit Testing**: Essential for catching issues early during the development phase. It ensures that individual components function as intended and helps maintain code quality.
  + **Exception Testing**: Vital for verifying robust error handling in the application. Proper exception handling is crucial for maintaining application stability and providing meaningful error messages to users.
  + **Coverage Testing:** Important for assessing the effectiveness of the test suite. High code coverage indicates thorough testing, which can lead to higher software reliability and fewer bugs in production.
  + **Integration Testing:** Useful for validating the interactions between different components of the application. Ensures that modules work together as expected.
  + **System Testing:** Critical for ensuring the entire application works according to the requirements. It validates the system’s compliance with its specifications and overall functionality.

**Mindset:**

Caution in Testing

Acting as a software tester, I employed caution by meticulously checking each feature against potential edge cases and invalid inputs. For instance, testing for invalid phone numbers in ContactTest ensured robustness against common user errors.

Limiting Bias

I consciously tried to limit bias by reviewing the code objectively and considering all possible failure points. Testing code I have written is not ideal, however it does happen, and it is very important to keep an open-mind when testing. This was particularly important in exception handling tests to ensure I did not overlook any potential issues. Getting up to take a break and clear my head helped ensure I was focused and did not miss simple errors.

Commitment to Quality

Being disciplined in my commitment to quality involved not cutting corners and ensuring comprehensive coverage of my tests. For example, I resisted the urge to skip a detailed review of my EclEmma coverage results, since 82.6% is considered high coverage. However, this would have created doubt when further developing. Instead, by completing the detailed review it increased my confidence in my code, as well as highlighted the lambda function deficiency in EclEmma. In the future, I plan to avoid technical debt by maintaining thorough documentation, rigid and professional coding practices, and regularly refactoring the code to address any emerging issues.

References:

Jakubiak Nathan. (2022). *JUnit Tutorial: Setting Up, Writing, and Running Java Unit Tests*. Parasoft.com. https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/

Morgan, P., Samaroo, A., Thompson, G., & Williams, P. (2019). *Software testing : An istqb-bcs certified tester foundation guide - 4th edition*. BCS Learning & Development Limited.

Mountainminds Gmbh & Co., et al. (2017). *Java Code Coverage for Eclipse: Frequently Asked Questions*. EclEmma.org. <https://www.eclemma.org/faq.html#trouble02>

SNHU Media. (2020, September 11). *JUnit Tutorial*. Youtube. https://www.youtube.com/watch?v=KbXhK9HUng4

*Tutorial Using JUnit PDF*. (n.d). Southern New Hampshire University.