**Maiza Falcon Rojas**

**March 12, 2024**

**Assignment: Activity 7 - Unit Testing**

**Class: CST-239**

**Loom Link:**

**Github Link:** **https://github.com/maiza02/Activity-7.git**

**Part 1**

**Figure 1: Screenshot of the JUnit results screen part a**

A screenshot of a computer

Description automatically generated

**Figure 2: Screenshot of the JUnit results screen part b**

A screenshot of a computer program

Description automatically generated

**Testing Questions**

**1. How many test cases do you write?**

For each operation (add, subtract, multiply, divide), I have two test cases in the `CalculatorTest` class. So, I have a total of 4 test cases.

**2. What error conditions would you need to test?**

Division by zero: I should include a test case to handle division by zero. It's a critical error condition that needs to be checked to avoid runtime exceptions.

Invalid input: Although not explicitly handled in the code, it's a good practice to validate input parameters to ensure they are within acceptable ranges.

**3. Can you think of a very important test case to validate division?**

Test dividing a non-zero number by zero. This will help ensure that the calculator handles division by zero gracefully and does not result in unexpected behavior or exceptions.

**4. What is White and Black box testing?**

White Box Testing:

- Involves testing the internal logic, structure, and code of the software.

- The tester has knowledge of the internal workings of the system.

- Test cases are designed based on code paths, conditions, and branches.

- In the code, testing individual methods like `add`, `subtract`, etc., with knowledge of their internal implementation, would be a form of white-box testing.

Black Box Testing:

- Focuses on the functionality of the software without knowledge of its internal code and logic.

- The tester is concerned with the inputs and expected outputs.

- Test cases are designed based on specifications, requirements, and functionality.

- The `CalculatorTest` class is mostly black-box testing, as it tests the calculator's functionality without delving into its internal implementation details.

**5. When would you use a Test Suite?**

- Test suites are used to group related tests together.

- They are helpful when you want to run multiple test cases or test classes together.

- Useful for organizing and categorizing tests, especially in larger projects with numerous test classes.

- In the code, the `AllTests` suite includes the `CalculatorTest` class, allowing to run all calculator-related tests in one go.

**Part 2**

**Figure 3: Screenshot of the JUnit results screen part 2**

A screenshot of a computer

Description automatically generated

**Testing Questions**

**How did Parameterized Test improve your testing?**

Parameterized Tests offer a more systematic and efficient approach to testing by allowing the reuse of test logic with different input values, resulting in improved coverage and maintainability of my test suite.

**Part 3**

**Figure 4: Screenshot of Console part 3**

**A screenshot of a computer

Description automatically generated**

**Testing Questions**

**How can you test that an exception is thrown in your code?**

To test that an exception is thrown in the code, I would use the @Test annotation in JUnit to mark a test method and then use the assertThrows method to assert that a specific exception is thrown during the execution of that method.

**What challenges can you think of that will make testing for all error conditions and exceptions in your code even possible?**

Identifying potential error conditions: It can be challenging to identify all possible error conditions and exceptions that may occur in my code, especially in complex systems with numerous dependencies.

Covering edge cases: Ensuring that the tests cover all edge cases and boundary conditions where errors or exceptions might occur can be challenging. It requires careful consideration and thorough testing to achieve comprehensive coverage.

Handling asynchronous operations: If the code were to involve asynchronous operations, testing for error conditions and exceptions in such scenarios can be complex due to the asynchronous nature of the operations.

**All UML Diagrams**

**Figure 5: UML Diagram for Part 1A**

**A screenshot of a computer program

Description automatically generated**

**Figure 6: UML Diagram for Part 1B**

**A screenshot of a computer program

Description automatically generated**

**Figure 7: UML Diagram for Part 2**

**A screenshot of a calculator

Description automatically generated**

**Figure 8: UML Diagram for Part 3**

**A screenshot of a graph

Description automatically generated**

**Follow Up Questions**

**What was challenging?**

Getting the JUnit to work properly in eclipse was quite difficult I had to do some adjustments to the way my project was setup in order for it to work properly.

**What did you learn?**

I learned how to use Junit testing and how to adjust eclipse to be able to use Junit.

**How would you improve on the project?**

What I would improve is on the directions in the project more specifically the tutorials that were given they were vague and not target for the app the class is required to use.

**How can you use what you learned on the job?**

Junit testing will help in the work fill to ensure a project is working properly before a user uses it.