**CST-239 Activity 7**

**Oct. 01,2023**

**Taofik Otiotio**

**PART 1: BASIC UNIT TESTS**

**TAKE A SCREENSHOT OF THE JUNIT RESULTS SCREEN.**

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

1. **HOW MANY TEST CASES DO YOU WRITE?**

I wrote 8 test cases, two for each of the four operations (add, subtract, multiply, divide).

**WHAT ERROR CONDITIONS WOULD YOU NEED TO TEST?**

For a calculator application, the following error conditions should be considered:

1. Division by zero.
2. Overflow and underflow conditions (e.g., when adding or multiplying very large numbers).
3. Invalid inputs (e.g., non-numeric values).

**CAN YOU THINK OF A VERY IMPORTANT TEST CASE TO VALIDATE DIVISION?**

One of the most crucial test cases for division is dividing by zero. It's vital to ensure that the calculator handles this scenario gracefully, either by throwing an exception, returning an error message, or some other meaningful response, instead of crashing or returning invalid data.

**B. WHITE AND BLACK BOX TESTING:**

White Box Testing, also known as Glass Box or Clear Box, is a type of testing where the tester is aware of the code's internal workings and logic. It focuses on code coverage, path coverage, function/method coverage, and loop coverage. The main goal is to test all code paths. Black Box Testing, on the other hand, is a type where the tester is only concerned with inputs and outputs, focusing on the software's functionality, invalid inputs, unexpected behaviors, boundary value analysis, and user experience.

1. **WHEN WOULD YOU USE A TEST SUITE?**
2. **REGRESSION TESTING:** After software updates, a Test Suite can be conducted to ensure that existing functionality is unaffected.
3. **ORGANISING TEST CASES:** Software often has many features and functions. Test suites organize test cases by features or user stories for easier management and gap identification.
4. **AUTOMATED TESTING:** CI/CD pipelines need Test Suites. They can be automatically triggered after every code commits to prevent bugs.
5. **SMOKE TESTING:** Before extensive testing, a simpler Test Suite (called a "smoke test" or "sanity test") can be done to verify fundamental and crucial functionality.
6. **PERFORMANCE TESTING:** You may have a Test Suite for performance testing to assess system behavior underload.
7. **PERIODIC CHECKS:** Test Suites can be run weekly or quarterly for applications in maintenance mode to ensure everything is working properly, especially if underlying platforms or dependencies are upgraded.
8. **REUSABLE AND REPRODUCIBLE:** A Test Suite allows the same collection of tests to be performed in the same order following a defect correction, assuring consistency.
9. **GROUPING COMPARABLE TEST SCENARIOS:** Test Suites can be created for comparable test scenarios or those that must be run in a certain order.
10. **SHARED TESTING:** In larger teams with numerous testers working on different features, each tester can construct their own Test Suite. Later, these suites can be integrated into a master suite for complete coverage.

**PART 2: PARAMETERIZED UNIT TESTS**

**TAKE A SCREENSHOT OF THE JUNIT RESULTS SCREEN.**

A screenshot of a computer

Description automatically generated

**HOW DID PARAMETERIZED TEST IMPROVE YOUR TESTING?**

The CalculatorTest class uses parameterized tests to optimize testing procedures. This single method allows for testing multiple scenarios, increasing test coverage without unnecessary code. It separates test data from underlying logic, ensuring uniformity and effectiveness. This simplifies the process of incorporating more test cases and simplifies error identification based on input parameters. Parameterized tests offer enhanced efficiency, scalability, and clarity in testing techniques.

**PART 3: ADVANCED UNIT TESTS**

**TAKE A SCREENSHOT OF THE JUNIT RESULTS SCREEN.**

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

1. **HOW CAN YOU TEST THAT AN EXCEPTION IS THROWN IN YOUR CODE?**

JUnit allows testing if a code piece throws an expected exception using the @Test annotation and its expected attribute, which indicates that the test method should throw the specified exception for success.

1. **WHAT CHALLENGES CAN YOU THINK OF THAT WILL MAKE TESTING FOR ALL ERROR CONDITIONS AND EXCEPTIONS IN YOUR CODE EVEN POSSIBLE**
2. **Complex Code Paths:** Multiple execution pathways increase testing complexity.
3. **External Dependencies:** Dependencies on outside systems can introduce unpredictability.
4. **Concurrent Code:** Issues like race conditions arise with multithreaded code.
5. **Unpredictable Inputs:** Anticipating every possible input, especially user-generated, is challenging.
6. **Non-Deterministic Behavior:** Randomness or time-based code is hard to test consistently.
7. **Legacy Code:** Old, undocumented code poses testing challenges.
8. **Resource Limitations:** Testing under extreme conditions might be infeasible.
9. **Hidden Bugs:** Some issues only appear under unique conditions.
10. **Time Constraints:** Limited time can prevent thorough testing.
11. **Over-Reliance on Unit Tests:** Focusing only on unit tests can overlook integration errors.