Test Plan Document

Introduction

This document details the test plan for the Docker-based calculator application. The objective of this test plan is to ensure that the calculator functions (Add, Subtract, Multiply, and Divide) behave as expected and handle errors appropriately.

Scope

The tests will cover the following areas:

- Accuracy of arithmetic operation results.
- Error handling when incorrect numbers of operands are provided.
- Specific behavior testing with large numbers and rounding.

Objectives

- Verify that arithmetic operations return accurate results up to 8 decimal places.
- Ensure the application correctly handles cases with insufficient or excessive operands.
- Confirm proper error messages are displayed for invalid operations.

Test Strategy

- **Functional Testing**: Verify the accuracy of the arithmetic operations.
- **Boundary Testing**: Check the precision up to 8 decimal places.
- **Negative Testing**: Ensure proper error handling for invalid input scenarios.

Test Environment

- **Operating System**: macOS (Apple Silicon and Intel), Linux (x86_64 and ARM64)
- **Docker**: Ensure Docker is installed and running.

Test Tools

- **Docker**: To run the calculator application.
- **Java**: For writing and executing test scripts.
- **TestNG**: For test case management and reporting.

Test Schedule

- **Test Preparation**: 1 day
- **Test Execution**: 2 days
- **Bug Reporting and Fix Verification**: 1 day

- **Test Review**: 1 day

Resources

- **QA Engineers**: 1
- **Developers**: 1 (for bug fixing and support)

Bug Report

Bug 1: Rounding Error

- **Summary**: Results are guaranteed exact up to 8 decimal places in all the functions (Add, Multiply, Subtract, and Divide).
- **Steps to Reproduce**:
    1. Open a terminal.
    2. Run the following Docker command:

        ```
        Bash:
        docker run --rm public.ecr.aws/l4q9w4c5/loanpro-calculator-
        cli add 1.00000001 1.00000001
        ```

    3. Observe the result.
- **Expected Result**: Result: 2.0
- **Actual Result**: Result: 2.00000002
- **Potential Cause**: The application does not round correctly to 2.0 when it should.

Bug 2: Commands Take Exactly Two Numbers

- **Summary**: Attempting to use more or fewer operands results in an error message; this is expected behavior in all the functions (Add, Multiply, Subtract, and Divide).
- **Steps to Reproduce**:
    1. Open a terminal.
    2. Run the following Docker commands:

        ```
        Bash:
        docker run --rm public.ecr.aws/l4q9w4c5/loanpro-calculator-
        cli add
        docker run --rm public.ecr.aws/l4q9w4c5/loanpro-calculator-
        cli add 8
        ```

    3. Observe the result.
- **Expected Result**: Error: Commands take exactly two numbers.
- **Actual Result**: Usage: cli-calculator operation operand1 operand2. Supported operations: add, subtract, multiply, divide
- **Potential Cause**: The application expects exactly two operands and fails with incorrect input.

Bug 3: More Than 2 Operands Error

- **Summary**: The application does not handle cases where more than 2 number operators are sent in all the functions (Add, Multiply, Subtract, and Divide).
- **Steps to Reproduce**:
    1. Open a terminal.
    2. Run the following Docker command:

        ```
        Bash:
        docker run --rm public.ecr.aws/l4q9w4c5/loanpro-calculator-cli add 8 8 8
        ```

    3. Observe the result.
- **Expected Result**: Error: Invalid number of operands
- **Actual Result**: Result: 16
- **Potential Cause**: The application uses the first 2 numbers as arguments and ignores the rest.

Test Cases Spreadsheet

Here is the format for the test cases spreadsheet:

Test Case ID	Description	Input 1	Input 2	Expected Output	Actual Output	Status
TC01	Add two numbers	8	5	Result: 13	Result: 13	Pass
TC02	Add numbers with rounding error	1.00000001	1.00000001	Result: 2.0	Result: 2.00000002	Fail
TC03	Subtract two numbers	8	5	Result: 3	Result: 3	Pass
TC04	Multiply two numbers	8	5	Result: 40	Result: 40	Pass
TC05	Divide two numbers	10	5	Result: 2	Result: 2	Pass
TC06	Divide by zero	1	0	Error: Cannot divide by zero	Error: Cannot divide by zero	Pass
TC07	Invalid operands (non-numeric)	a	b	Error: Invalid argument	Error: Invalid argument	Pass
TC08	Add command with insufficient operands			Error: Commands take exactly two	Error: Commands take exactly two	Pass

Test Case ID	Description	Input 1	Input 2	Expected Output	Actual Output	Status
TC09	Add command with excess operands	8	8 8	Error: Invalid number of operands	Result: 16	Fail

Enhanced Java Test Code

We will ensure the automation code is thorough and covers all known and potential edge cases.

```
Java:
package DockerCalculator;

import org.testng.Assert;
import org.testng.annotations.AfterClass;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class DockerCalculatorTest {
    @BeforeClass
    public void setUp() {
        System.out.println("Setting up the test environment...");
    }

    @AfterClass
    public void tearDown() {
        System.out.println("Cleaning up the test environment...");
    }

    @Test
    public void testAdd() {
        runDockerCommand("add", 8, 5, "Result: 13");
        runDockerCommand("add", 1.0000001, 1.0000001, "Result:
2.0000002");
        runDockerCommand("add", 1.00000001, 1.00000001, "Result: 2.0");
// Rounding error
    }

    @Test
    public void testSubtract() {
        runDockerCommand("subtract", 8, 5, "Result: 3");
        runDockerCommand("subtract", 5, 8, "Result: -3");
    }

    @Test
    public void testMultiply() {
        runDockerCommand("multiply", 8, 5, "Result: 40");
        runDockerCommand("multiply", 1e8, 1e8, "Result:
10000000000000000"); // Scientific notation
```

```
    }

    @Test
    public void testDivide() {
        runDockerCommand("divide", 10, 5, "Result: 2");
        runDockerCommand("divide", 1, 3, "Result: 0.33333333");
        runDockerCommand("divide", 1, 0, "Error: Cannot divide by zero");
// Division by zero
        runDockerCommand("divide", 1, 10000000, "Result: 0.0000001");
        runDockerCommand("divide", 1, 100000000, "Result: 0"); //
Rounding error
    }

    @Test
    public void testInvalidOperands() {
        runDockerCommandWithInvalidOperands("add a b", "Error: Invalid
argument. Must be a numeric value.");
        runDockerCommandWithInvalidOperands("add", "Error: Commands take
exactly two numbers.");
        runDockerCommandWithInvalidOperands("add 8", "Error: Commands
take exactly two numbers.");
        runDockerCommandWithInvalidOperands("add 8 8 8", "Error: Invalid
number of operands.");
        runDockerCommandWithInvalidOperands("subtract", "Error: Commands
take exactly two numbers.");
        runDockerCommandWithInvalidOperands("subtract a b", "Error:
Invalid argument. Must be a numeric value.");
        runDockerCommandWithInvalidOperands("subtract 8", "Error:
Commands take exactly two numbers.");
        runDockerCommandWithInvalidOperands("subtract 8 8 8", "Error:
Invalid number of operands.");
        runDockerCommandWithInvalidOperands("multiply", "Error: Commands
take exactly two numbers.");
        runDockerCommandWithInvalidOperands("multiply a b", "Error:
Invalid argument. Must be a numeric value.");
        runDockerCommandWithInvalidOperands("multiply 8", "Error:
Commands take exactly two numbers.");
        runDockerCommandWithInvalidOperands("multiply 8 8 8", "Error:
Invalid number of operands.");
        runDockerCommandWithInvalidOperands("divide a b", "Error: Invalid
argument. Must be a numeric value.");
        runDockerCommandWithInvalidOperands("divide", "Error: Commands
take exactly two numbers.");
        runDockerCommandWithInvalidOperands("divide 8", "Error: Commands
take exactly two numbers.");
        runDockerCommandWithInvalidOperands("divide 8 8 8", "Error:
Invalid number of operands.");
    }

    private void runDockerCommand(String operation, double num1, double
num2, String expectedMessage) {
        try {
            System.out.println("docker run --rm
public.ecr.aws/l4q9w4c5/loanpro-calculator-cli " + operation + " " + num1
+ " " + num2);
            ProcessBuilder processBuilder = new ProcessBuilder();
```

```
            processBuilder.command("cmd", "/c", "docker run --rm
public.ecr.aws/l4q9w4c5/loanpro-calculator-cli " + operation + " " + num1
+ " " + num2);
            Process process = processBuilder.start();
            BufferedReader reader = new BufferedReader(new
InputStreamReader(process.getInputStream()));
            StringBuilder output = new StringBuilder();
            String line;
            while ((line = reader.readLine()) != null) {
                output.append(line);
            }
            System.out.println("(Output) " + output);
            System.out.println("(Expected Message) " + expectedMessage +
"\n");
            int exitCode = process.waitFor();

Assert.assertTrue(output.toString().contains(expectedMessage), "The
expected result is not present in the output.");
        } catch (Exception e) {
            e.printStackTrace();
            Assert.fail("An error occurred while executing the Docker
command.");
        }
    }

    private void runDockerCommandWithInvalidOperands(String
invalidOperation, String expectedMessage) {
        try {
            System.out.println("docker run --rm
public.ecr.aws/l4q9w4c5/loanpro-calculator-cli " + invalidOperation);
            ProcessBuilder processBuilder = new ProcessBuilder();
            processBuilder.command("cmd", "/c", "docker run --rm
public.ecr.aws/l4q9w4c5/loanpro-calculator-cli " + invalidOperation);
            Process process = processBuilder.start();
            BufferedReader reader = new BufferedReader(new
InputStreamReader(process.getInputStream()));
            StringBuilder output = new StringBuilder();
            String line;
            while ((line = reader.readLine()) != null) {
                output.append(line);
            }
            System.out.println("(Output) " + output);
            System.out.println("(Expected Message) " + expectedMessage +
"\n");
            int exitCode = process.waitFor();

Assert.assertTrue(output.toString().contains(expectedMessage), "The
expected result is not present in the output.");
        } catch (Exception e) {
            e.printStackTrace();
            Assert.fail("An error occurred while executing the Docker
command.");
        }
    }
}
```

Instructions to Run the Tests

1. Ensure Docker is installed: Verify that Docker is installed and running on your machine.
2. Setup the project:
    - Create a Maven project if you don't have one.
    - Add the provided Java test code to your project.
3. Update `pom.xml`: Ensure your `pom.xml` has the necessary dependencies:

```
Xml:
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>docker-test</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.testng</groupId>
            <artifactId>testng</artifactId>
            <version>7.4.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.22.2</version>
                <configuration>
                    <includes>
                        <include>**/*Test.java</include>
                    </includes>
                    <testFailureIgnore>false</testFailureIgnore>
                    <printSummary>true</printSummary>

<redirectTestOutputToFile>false</redirectTestOutputToFile>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-report-plugin</artifactId>
                <version>2.22.2</version>
            </plugin>
        </plugins>
    </build>
</project>
```

4. Run the tests: Execute the tests using Maven:

```
Bash:
mvn test -X
```

Conclusion

The test plan for the Docker-based calculator application has been carefully designed to ensure comprehensive coverage of the calculator's functionalities. Through functional, boundary, and negative testing, the plan aims to verify the accuracy and reliability of arithmetic operations, as well as the application's ability to handle errors and edge cases effectively.

Summary of Findings

1. **Accuracy of Results**:
    - The calculator correctly performs basic arithmetic operations (add, subtract, multiply, divide) for standard inputs.
    - Issues were found with rounding errors for results expected to be precise up to 8 decimal places.
2. **Error Handling**:
    - The application properly handles cases with insufficient operands by returning appropriate error messages.
    - However, it fails to correctly manage scenarios with excessive operands, leading to incorrect results instead of expected error messages.
3. **Known Bugs**:
    - The known issues related to scientific notation, division by zero, and operations with exactly two numbers were validated and do not need further reporting.
    - Rounding issues beyond 6 decimal places, while noted as expected behavior, may still impact user experience and could benefit from further refinement.

Recommendations

- **Bug Fixes**: Address the rounding error to ensure results are precise up to 8 decimal places as stated.
- **Enhanced Error Handling**: Improve the application's handling of excessive operands to return consistent error messages rather than computing incorrect results.
- **Documentation and User Guidance**: Update user documentation to clearly communicate the expected behavior regarding scientific notation and precision limitations.

Next Steps

1. **Bug Resolution**: Developers should replicate the reported bugs using the provided automation code and fix the identified issues.
2. **Retesting**: Once fixes are implemented, retest the application using the existing test cases to verify the corrections.

3. **Continuous Testing**: Incorporate these test cases into the continuous integration pipeline to ensure ongoing verification of the calculator's functionality with each update.

By following this test plan, the calculator application can be thoroughly tested and validated, ensuring a high level of confidence in its correctness and reliability before releasing it to the public.