Lab 1 Report

ECE 322
Diepreye Charles-Daniel
Student Number: 1766907

Introduction:

The purpose of this lab is to serve as a practical introduction to black box testing techniques like dirty testing, error guessing, and partition testing discussed in the lecture. The completion of the exercises we have below, for the 2 applications involved, provide relevant experience in identifying the problem and user requirements, understanding the theory involved in black box testing, and applying them properly to effectively provide test cases that aim to find errors in the systems/applications provided.

Part One – Calculator Program

Q1 (3 marks): Require an explanation of the application under test, they should describe the problem that is being tested :
Answer:
The application under test is a **simple calculator program** designed to evaluate arithmetic expressions using numeric inputs and the standard order of operations (**BEDMAS** — Brackets, Exponents, Division/Multiplication, Addition/Subtraction). The goal of testing is to verify that the calculator correctly handles both valid mathematical expressions and invalid input scenarios.

According to the given requirements**, non-numeric characters are considered invalid inputs**, and the program must handle such cases gracefully without crashing. The testing therefore focuses on correctness of operator precedence, handling of invalid characters, divide-by-zero protection, and robustness against malformed or extreme expressions.

Q2 (12 marks): Explain what testing methods you are using. The testing methods should be explained, what do they test, what are they good for etc ..
Errors in the application should be identified, and some justification given for what may be causing these errors. :
Answer:

To test the calculator, I used two complementary black box testing techniques:

1. **Failure / Dirty Testing**
   This approach involves testing the program destructively — deliberately entering inputs that break expected assumptions. It focuses on **finding crashes, unhandled exceptions, or incorrect outputs** by using malformed, unexpected, or extreme input values.

- **What it tests:** Robustness of input handling, exception management, and numerical stability.
- **Why it's useful:** It mimics real-world user behavior and often reveals hidden validation issues that structured tests might miss.
- **Advantage:** Uncovers real-world failures that occur under misuse or boundary stress.
- **Limitation:** Unsystematic and relies heavily on tester creativity and experience.

2. **Error Guessing**
   This technique relies on **intuition, experience, and prior knowledge** of common programming mistakes. For example, I expected possible logic errors around operator precedence (exponent vs multiplication), handling of consecutive operators (e.g., "++" or "^^"), and missing operands.
   - **What it tests:** Logical correctness and parser consistency under error-prone input patterns.
   - **Why it's useful:** It focuses on areas where developers are most likely to make mistakes — such as mis ordered precedence rules or missing input validation.
   - **Advantage:** Uncovers real-world failures that occur under misuse or boundary stress.
   - **Limitation:** Unsystematic and relies heavily on tester creativity.

Identified Errors and Justifications
Based on the test outcomes:

Error	*Example Input*	*Expected vs Actual*	*Possible Cause*
Operator Precedence Misinterpretation	2^3-3	Expected 5.0 but got 1.0	The program may be evaluated from the left to the right without respecting the expected order of operations
Negation and Exponent Handling	3*-2^2	Expected 12.0 but got 36.0	The multiplication operation happens before the exponent operand which goes against the order of operations
Invalid Character Handling	3*abc	The application correctly returns NaN as the result	The calculator properly detects the invalid input but the messaging

			could be handled better
Divide By Zero	123/(1-1)	The result is NaN as expected	We don't have an explicit divide by zero error message showing
Duplicate Operators	2^^3 and 3++4	The calculator should show and error but gave numerical results	The application's input parser fails to enforce any strict syntax validation
Missing Operand / Leading Operator	3+, *7	Should return error but return NaN and 0.0 respectively	The input evaluator is likely defaulting missing operands to 0 instead of reporting the invalid syntax
Numeric Overflow	100000000000000000001 * 9	Should either return big integer or warn the user of an overflow error	Result was 9.0E20 which is suggesting that the application is auto converting to scientific notation and/or a truncated precision

Q3 (3 marks): Discuss the effectiveness of the testing methods used
Answer:
Both failure testing and error guessing were highly effective for this application.

Failure testing successfully revealed robustness and error-handling issues, such as divide-by-zero and invalid operator sequences, by simulating destructive or malformed user inputs.

Error guessing exposed logical and precedence errors that structured partition testing might miss, particularly in cases involving exponents and chained operators.

However, these methods are unsystematic and depend heavily on tester intuition. To achieve complete coverage, they should be complemented with equivalence class partitioning and boundary value analysis in future tests.

Q4 (14 marks): Test Cases Table:
Answer:

Expected invalid test cases, order do not matter

ID	*Inputs*	*Description*	*Expected Result*	*Actual Result*
1	2^3-3	Testing the exponent order of operations in BEDMAS in the application	5.0	1.0
2	3*-2^2	Testing the exponent order of operations in BEDMAS in the application	12.0	36.0
3	3*abc	Testing no characters allowed in the calculator requirements	NaN	NaN
4	123/(1-1)	Testing division by 0 to break the calculator	NaN	NaN
5	2^^3	Testing duplicate operator tokens	NaN	1.0
6	3++4	Testing duplication operator tokens	NaN	7.0
7	3+	Testing missing operand	NaN	NaN
8	*7	Testing missing leading operand	NaN	0.0
9	100000000000000000001 * 9	Testing an extreme numeric size	Either a defined big integer answer of overflow but no crashing	9.0E20

Part 2 – Triangle Classification Program (32 marks)

Q1 (3 marks): explanation of the application under test, and a description of the problem
Answer:
The Triangle Classification application reads 3 side lengths of a triangle and classifies the type of the triangle-Equilateral, Isosceles, or Scalene-or reports an error. The testing problem we are provided with is to properly and systematically cover all the **valid triangle categories** and **invalid input** especially testing the **boundary cases** around the triangle inequality and natural number constraints.

This ensures the program correctly distinguishes between valid triangles and invalid configurations while handling both format and geometric errors.

Q2 (8 marks): List all the Partition classes
Answer:

1. **Using the specs**:
    a. **Valid**: exactly 3 space separated **positive integers** on one line
    b. **Invalid**: fewer or more that 3 input tokens, non-integer values, 0 or negative values, non-numeric values, and an extra space should be handled
2. **By triangle inequality**:
    a. **Valid**: a > 0 & b > 0 & c > 0 & a+ b > c & a + c > b & b+ c > a
    b. **Invalid**: any input that invalidates the above composite condition
3. **Using triangle type**:
    a. **Equilateral Triangle**: a = b = c
    b. **Isosceles**: exactly 2 sides equal and satisfies triangle inequality
    c. **Scalene**: all three sides different and satisfies triangle inequality
4. **Boundary-focused subclasses:**
    a. **Near equality:** a + b = c which is invalid and a + b = c + 1 which just valid
    b. **Large values / potential overflow in** a+b **computations**

Q3 (4 marks): Identification of error(s) in the program. Must be identified and justified

Answer:
Our testing revealed a few flaws in how the Triangle program enforces the triangle inequality and other edge cases

Observed Issue	*Example Input*	*Expected Result*	*Actual Result*	*Possible Cause*
Triangle inequality not followed	2 2 4, 1 1 2, 10 10 20	Error – not a triangle	Isosceles	The condition a + b > c appears to use be >= instead or isn't checked properly
Input validation robust for negative or zero values	0 3 3, -1 3 3	Error – non-positive	Correctly flagged	Validations for positive integers work as intended
Overflow or type-check failure on large inputs	10000000000 10000000000 10000000000	Either handled gracefully or classified as equilateral	Error: Invalid argument – non-integer	Program likely casts to integer type and fails parsing
Correct classification for valid cases	5 5 5, 5 5 3, 5 4 3	Accurate types	Accurate	Main classification logic for valid triangles works well

Q4 (3 marks): Discuss the effectiveness of the testing methods used
Answer:
The testing methods-**Equivalence Partitioning** and **Boundary Value Analysis (BVA)**-were effective in finding errors with our minimal test cases.

- Equivalence Partitioning ensured that each logical input class was represented without much redundancy.
- Boundary Value Analysis was particularly useful for identifying the logic faults around the boundaries (a + b = c, a + b = c + 1)
- Advantage: Uncovers real-world failures that occur under misuse or boundary stress.
- Limitation: Unsystematic and relies heavily on tester creativity.

Q5 (14 marks): Test Cases Table :
Answer:

ID	*Inputs a b c*	*Description*	*Expected Result*	*Actual Result*
1	5 5 5	All equal sidess	Equilateral	Equilateral
2	5 5 3	Two equal sides, valid	Isosceles	Isosceles
3	5 4 3	All different, valid	Scalene	Scalene
4	2 2 4	Boundary, a + b = c	Error not a triangle	Isosceles
5	2 3 6	Invalid Triangle Inequality: a + b < c	Error not a triangle	ERROR: Invalid triangle
6	1 2 2	Boundary values: Just valid	Isosceles	Isosceles
7	1 1 2	Boundary: a + b = c	Error not a triangle	Isosceles
8	0 3 3	Zero side	Error non-positive side	ERROR: Invalid argument - non positive value
9	-1 3 3	Negative side	Error non-positive side	ERROR: Invalid argument - non positive value
10	3 3 10000	Large skew value	Error not a triangle	ERROR: Invalid triangle

11	7 8 9	Scalene	Scalene	Scalene
12	10 10 19	Boundary values: valid	Isosceles	Isosceles
13	10 10 20	a + b = c	Error not a triangle	Isosceles
14	10 11 20	Just valid	Scalene	Scalene
15	A 3 4	Not an integer	Error not an integer	ERROR: Invalid argument - non integer
16	3 4	Too few arguments	Error too few args	ERROR: Not enough arguments
17	3 4 5 6	Too many arguments	Error too many args	ERROR: Too many arguments
18	3 4 5	Trailing whitespaces at the end of the input which should be handled properly	Scalene	Scalene
19	10000000000 10000000000 10000000000	Large integer values testing overflow	Overflow warning	ERROR: Invalid argument - non integer

Conclusion (3 marks)

In this lab, we applied different **black box testing techniques** such as failure testing, error guessing, equivalence partitioning, and boundary value analysis to evaluate two programs. These methods helped identify several faults and verify the programs' handling of both valid and invalid inputs.

For the **Calculator program**, the tests exposed issues with operator precedence, duplicate operator handling, and weak input validation. For the **Triangle Classification Program**, the tests revealed errors in enforcing the triangle inequality and improper handling of degenerate triangles and large integer inputs.

Overall, the lab showed how combining creative (failure/error guessing) and systematic (partitioning/BVA) testing approaches provides better coverage and helps uncover both logical and boundary-related faults efficiently.