Name: Phan Vu – Student ID: 104222099

Assignment 1: Software Testing and Reliability

Swinburne University of Technology

Introduction

This assignment aims to strengthen your understanding of software testing activities and the process of generating test cases for a given program. The focus is on detecting any possible incorrect use of arithmetic operators within the provided program.

Program Under Test

The program to be tested is as follows:
Input A, B // A and B are real variables
A = A - B
C = A * 2
Output C // C is a real variable

Objective

The main testing objective is to detect any possible incorrect use of the arithmetic operators '-' and '*'. This involves verifying whether '-' is used correctly in 'A = A - B' and '*' is used correctly in 'C = A * 2'.

Task 1: Designing Test Cases

To design effective test cases, we need to consider all possible ways in which the operators could be incorrectly used. This includes:

1. Incorrect use of the '-' operator in 'A = A - B'.
2. Incorrect use of the '*' operator in 'C = A * 2'.
3. Incorrect use of both '-' and '*' operators.

Approach:

1. Identify Valid and Invalid Scenarios:
   - Valid: Both operators are used correctly.
   - Invalid: Either or both operators are used incorrectly.
2. Construct Test Cases:
   - Select values for 'A' and 'B' to cover the valid and invalid scenarios.
   - Ensure that the test cases include edge cases, such as 'A' and 'B', which are zero, positive, and negative numbers.

Task 2: Analysis of Test Case (A=3, B=1)

Using the test case (A=3, B=1):

Input	A = 3, B = 1
First operation	A = A - B => A = 3 - 1 = 2
Second operation	C = A * 2 => C = 2 * 2 = 4

Justification:

- This test case verifies that the subtraction and multiplication operators are used correctly.
- It provides a straightforward example where both operations are expected to produce correct results.

- However, this test case alone cannot guarantee the detection of all incorrect uses of arithmetic operators.

Task 3: Concrete Test Cases

Based on the design from Task 1, concrete test cases are:

Test Case	*Input A*	*Input B*	*Expected A after A = A - B*	*Expected C after C = A * 2*	*Purpose*
1	0	1	-1	-2	Verifying edge case where A = 0.
2	1	1	0	0	Ensuring the result matches C = 0.
3	3	1	2	4	Testing typical positive values.
4	2	1	1	2	Verifying correct multiplication.
5	-1	1	-2	-4	Testing negative values.
6	-3	1	-4	-8	Ensuring results match for negative inputs.
7	-2	1	-3	-6	Testing another set of negative values.
8	-5	1	-6	-12	Edge case for negative range.
9	4	1	3	6	Positive values ensuring multiplication.
10	5	1	4	8	Higher positive values for completeness.

Task 4: Values of A for Given B=1

Given B=1, we need to find all possible values of A so that the test cases cannot achieve the testing objective. These are the scenarios where the test cases fail to detect incorrect use of arithmetic operators.

Analysis:

1. Edge Numbers: These are numbers that are either too large or are floating-point numbers, leading to imprecise results. Precision is critical since the task involves real numbers. Examples include numbers like 1.11, 1.12, etc.

2. C = A (or C = B): When the result **C** is the same as the input **A** or **B**, it becomes challenging to detect errors. This scenario fails to meet the testing objective. For the program provided, with **C = (A - B) * 2** and **B = 1**, values of **A** such as -2, -1, 0, 1, 2, 3 might fall into this category.

3. C1 = C2 = Cn: If multiple inputs for **A** result in the same **C**, it becomes challenging to distinguish and detect errors in the operator usage. This necessitates identifying such values mathematically.

Identified Edge Cases:

Values like **A = -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5** and floating-point numbers like **1.11** might obscure errors and fail to achieve the testing objective.

Demonstration:

Python code:

```python
# Define the program under test
def program_under_test(A, B):
    A = A - B
    C = A * 2
    return C

# Given value of B
B = 1

# Function to check if the test case reveals an incorrect use of operators
def check_incorrect_operator_use(A, B, expected_C):
    C = program_under_test(A, B)
    return C != expected_C

# Expected results for the correct use of operators
def expected_result(A, B):
    return (A - B) * 2

# Finding values of A that do not reveal incorrect operator use
def find_edge_cases(B):
    A_values = range(-100, 101)  # Expanded range
    undetectable_A_values = []

    # Include specific identified values directly
    identified_values = [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]

    for A in A_values:
        expected_C = expected_result(A, B)
        if not check_incorrect_operator_use(A, B, expected_C):
            undetectable_A_values.append(A)

    # Add specific values identified
    for A in identified_values:
        expected_C = expected_result(A, B)
        if not check_incorrect_operator_use(A, B, expected_C):
            if A not in undetectable_A_values:
                undetectable_A_values.append(A)

    # Floating point precision issue
    float_A_values = [i / 100 for i in range(-10000, 10000)]  # Range from -100 to 100 in 0.01 increments
    for A in float_A_values:
        expected_C = expected_result(A, B)
        if not check_incorrect_operator_use(A, B, expected_C):
            undetectable_A_values.append(A)

    return undetectable_A_values

undetectable_A_values = find_edge_cases(B)
print(f"For B={B}, the values of A that do not reveal incorrect operator use are: {undetectable_A_values}")
```

Output (shortcut output first and last few results):

```
vufanity@Vus-Mac-Buc-Po-Ro desktop % python3 test.py
For B=1, the values of A that do not reveal incorrect operator use are: [-100, -99, -98, -97, -96, -95, -94, -93, -92, -91
, -90, -89, -88, -87, -86, -85, -84, -83, -82, -81, -80, -79, -78, -77, -76, -75, -74, -73, -72, -71, -70, -69, -68, -67,
-66, -65, -64, -63, -62, -61, -60, -59, -58, -57, -56, -55, -54, -53, -52, -51, -50, -49, -48, -47, -46, -45, -44, -43, -4
2, -41, -40, -39, -38, -37, -36, -35, -34, -33, -32, -31, -30, -29, -28, -27, -26, -25, -24, -23, -22, -21, -20, -19, -18,
 -17, -16, -15, -14, -13, -12, -11, -10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 4
4, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74,
 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, -100.0, -99.99,
```

```
9.1, 99.11, 99.12, 99.13, 99.14, 99.15, 99.16, 99.17, 99.18, 99.19, 99.2, 99.21, 99.22, 99.23, 99.24, 99.25, 99.26, 99.27,
 99.28, 99.29, 99.3, 99.31, 99.32, 99.33, 99.34, 99.35, 99.36, 99.37, 99.38, 99.39, 99.4, 99.41, 99.42, 99.43, 99.44, 99.4
5, 99.46, 99.47, 99.48, 99.49, 99.5, 99.51, 99.52, 99.53, 99.54, 99.55, 99.56, 99.57, 99.58, 99.59, 99.6, 99.61, 99.62, 99
.63, 99.64, 99.65, 99.66, 99.67, 99.68, 99.69, 99.7, 99.71, 99.72, 99.73, 99.74, 99.75, 99.76, 99.77, 99.78, 99.79, 99.8,
99.81, 99.82, 99.83, 99.84, 99.85, 99.86, 99.87, 99.88, 99.89, 99.9, 99.91, 99.92, 99.93, 99.94, 99.95, 99.96, 99.97, 99.9
8, 99.99]
```

Explanation:

1. **program_under_test(A, B):** Represents the provided program.
2. **check_incorrect_operator_use(A, B, expected_C):** Compares the actual output **C** with the expected result to identify discrepancies.
3. **expected_result(A, B):** Calculates the expected **C** using correct operators.
4. **find_edge_cases(B):**
    - Tests a broad range of integer values and floating-point numbers.
    - Integrates specific values identified.
    - Checks if the output **C** matches the expected result for various **A** values.
    - Collects values of **A** that do not reveal incorrect operator use.

⇨ For **B** = 1, the identified values of **A** that do not reveal incorrect operator use are comprehensive and include a broad range of integers and specific floating-point values. This approach ensures thorough coverage and detection of edge cases that might obscure errors in the program.