Question No: 01

You are testing a photo-enforcement system for traffic control in an intersection. Consider the following scenarios

[4*2.5 marks]

(Scenario

i) A photo will be taken if the light is red (RED) and the front wheels of the car are over the line marking the beginning of the intersection (WHEELS).

a) Which sets of values provides the minimum tests to achieve 100% decision/condition coverage?

(Scenario ii) A photo should be taken if the signal light is red (RED) or the car is speeding (SPEED) and if the front wheels of the car are over the line marking the beginning of the intersection (WHEELS).

b) Which sets of values provide the minimum tests to achieve 100% modified condition/decision coverage?

c) Which sets of values provide the minimum tests to achieve 100% multiple condition coverage?

d) Which sets of values provide the minimum tests to achieve 100% path coverage?

Ans:

a)

Variables – RED and WHEEL **(Condition)**

Action – PHOTO TAKEN

	Test case 1	Test case 2	Test case 3	Test case 4
RED	F	T	F	T
WHEEL	F	F	T	T
PHOTO TAKEN	F	F	F	T

Condition should be evaluated to condition 1 and 4

b)

(Scenario ii) A photo should be taken if the signal light is red (RED) or the car is speeding (SPEED) and if the front wheels of the car are over the line marking the beginning of the intersection (WHEELS).

b) Which sets of values provide the minimum tests to achieve 100% modified condition/decision coverage?

(R -> T **OR** S ->T) **AND (**W -> T) => P -> T

2^n , n is action

2^3 = 8 (2 to power of 3)

Variables – RED and WHEEL (Condition)

Action – PHOTO TAKEN

	Test case 1	Test case 2	Test case 3	Test case 4	Test case 5	Test case 6	Test case 7	Test case 8
RED	F	F	F	F	T	T	T	T
SPEED	F	F	T	T	F	F	T	T
WHEEL	F	T	F	T	F	T	F	T
PHOTO TAKEN	F	F	F	T	F	T	F	T

1,2,4 & 6 minimum test is required

2,4,6,7 ?

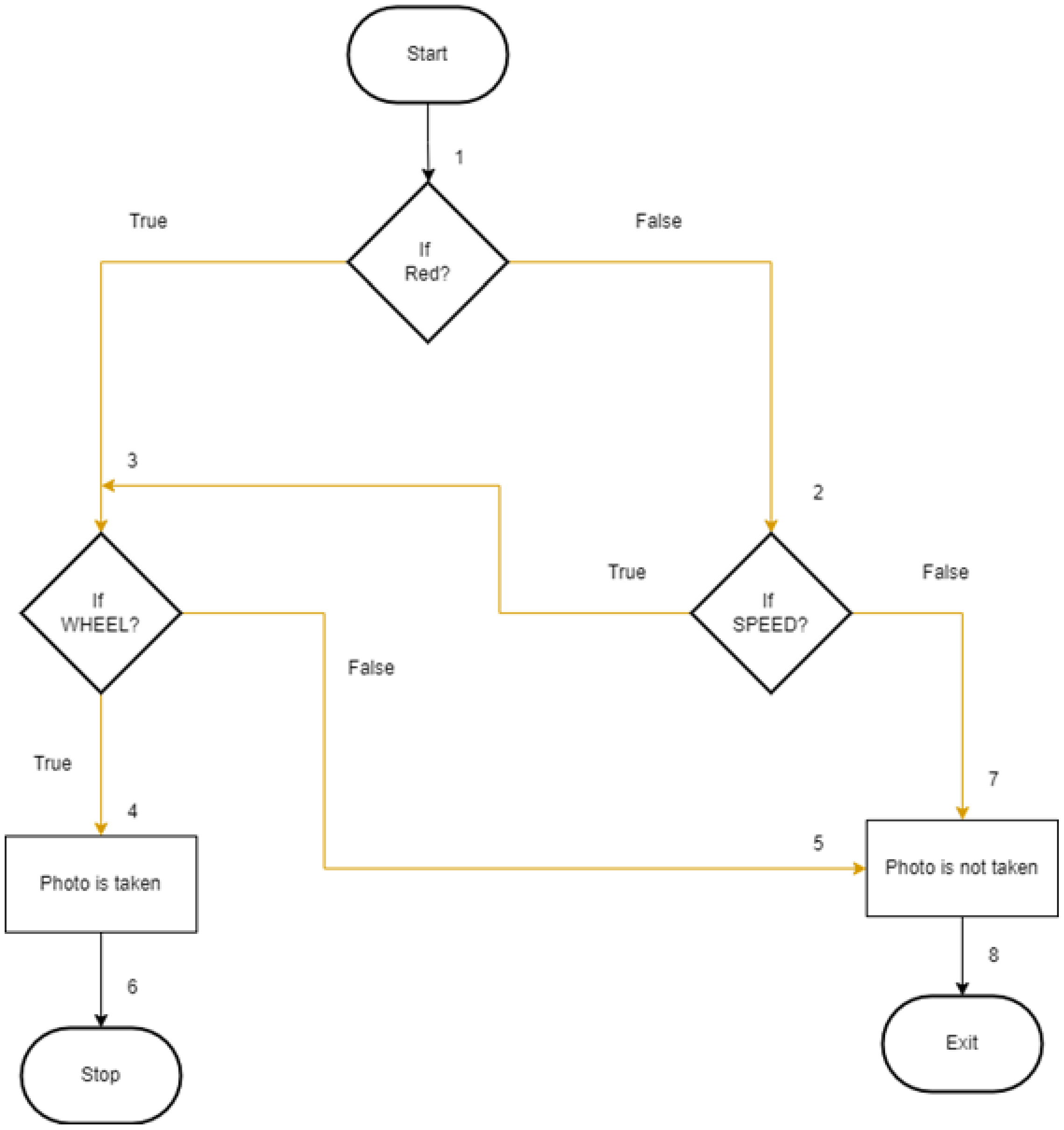
Note: number of test case is n+1, where n is number of condition = 3+1 = 4

c)

	Test case 1	Test case 2	Test case 3	Test case 4	Test case 5	Test case 6	Test case 7	Test case 8
RED	F	F	F	F	T	T	T	T

SPEED	F	F	T	T	F	F	T	T
WHEEL	F	T	F	T	F	T	F	T
PHOTO TAKEN	F	F	F	T	F	T	F	T

d) Which sets of values provide the minimum tests to achieve 100% path coverage?



Path coverage – 1, 2, 3 and 8

	Test case 1	Test case 2	Test case 3	Test case 4	Test case 5	Test case 6	Test case 7	Test case 8
RED	F	F	F	F	T	T	T	T

SPEED	F	F	T	T	F	F	T	T
WHEEL	F	T	F	T	F	T	F	T
PHOTO TAKEN	F	F	F	T	F	T	F	T

Question No. 02

An IOT enabled bulb changes states between red, green and blue on receiving the following input signals via Wi-Fi

[6 +4 marks]

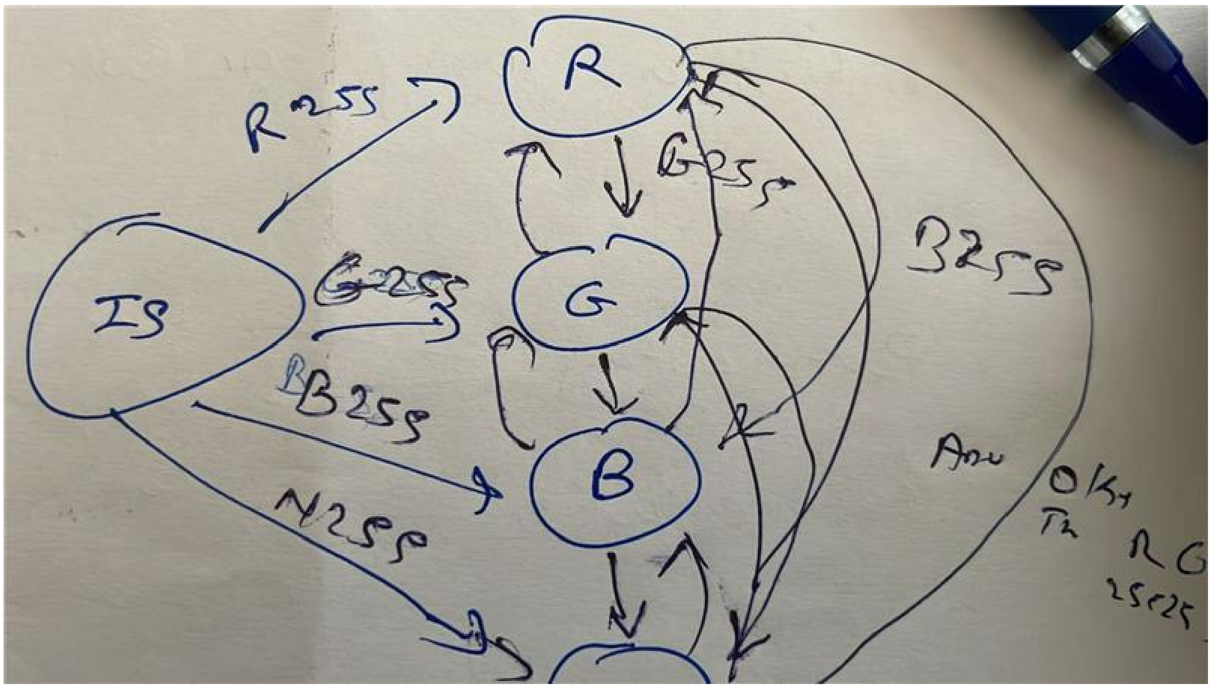
Enable = R255 ------ Red state

Enable = G255 ------ Green state

Enable = B255 ------ Blue state

In case of other input being sent at the Enable - the bulb takes a 'Dim state'.

a) Draw the state transition diagram for the bulb.



b) Write test cases to test all the states.

Test cases

Current state	Enable i/p	[Expected o/p] Next state.
Dim	R255	Red
Dim	G255	Green
Dim	B255	Blue
Dim	R100	Dim
Red	R255	Red
Red	B255	Blue
Red	G255	Green
Red	R100	Dim

(ii) The bulb needs to interact with a Wi-Fi enabled device in order to receive the 'enable' input. Write system-level test cases to test this system.

Test Wi-Fi Connectivity:

- Verify that the tricolor bulb can establish a Wi-Fi connection with the device.
- Verify that the Wi-Fi connection remains stable and does not disconnect unexpectedly.
- Test the tricolor bulb's ability to reconnect to the Wi-Fi network after a power cycle or network disruption.

Test Enable Input:

- Verify that the tricolor bulb can receive the "enable" input from the Wi-Fi enabled device.
- Test the tricolor bulb's response time to the "enable" input and ensure it reacts promptly.
- Verify that the tricolor bulb interprets the "enable" input correctly and activates the appropriate behavior (e.g., changing colors).

Test Disabling Functionality:

- Verify that the tricolor bulb can receive a "disable" input from the Wi-Fi enabled device.
- Test the tricolor bulb's response time to the "disable" input and ensure it reacts promptly.
- Verify that the tricolor bulb interprets the "disable" input correctly and deactivates the behavior initiated by the "enable" input (e.g., stops changing colors).

Test Multiple Device Interactions:

- Test the tricolor bulb's ability to handle simultaneous "enable" inputs from multiple Wi-Fi enabled devices.
- Verify that the tricolor bulb correctly identifies and responds to each device's "enable" input independently.
- Test scenarios where different devices send conflicting "enable" or "disable" inputs, and verify that the tricolor bulb handles them appropriately.

Test Error Handling:

- Simulate scenarios where the Wi-Fi connection between the tricolor bulb and the device is weak or intermittent.
- Verify that the tricolor bulb handles such situations gracefully, maintains functionality, and reconnects automatically when the connection is restored.

Test Compatibility:

- Verify the tricolor bulb's compatibility with different Wi-Fi enabled devices (e.g., smartphones, tablets, smart hubs).
- Test the tricolor bulb's ability to receive the "enable" input from different types of devices and ensure consistent behavior.

Test Security Measures:

- Verify that the tricolor bulb implements proper security measures, such as encryption protocols, to protect the Wi-Fi communication.
- Test the tricolor bulb's resistance to unauthorized access or malicious attempts to manipulate the "enable" input.
-

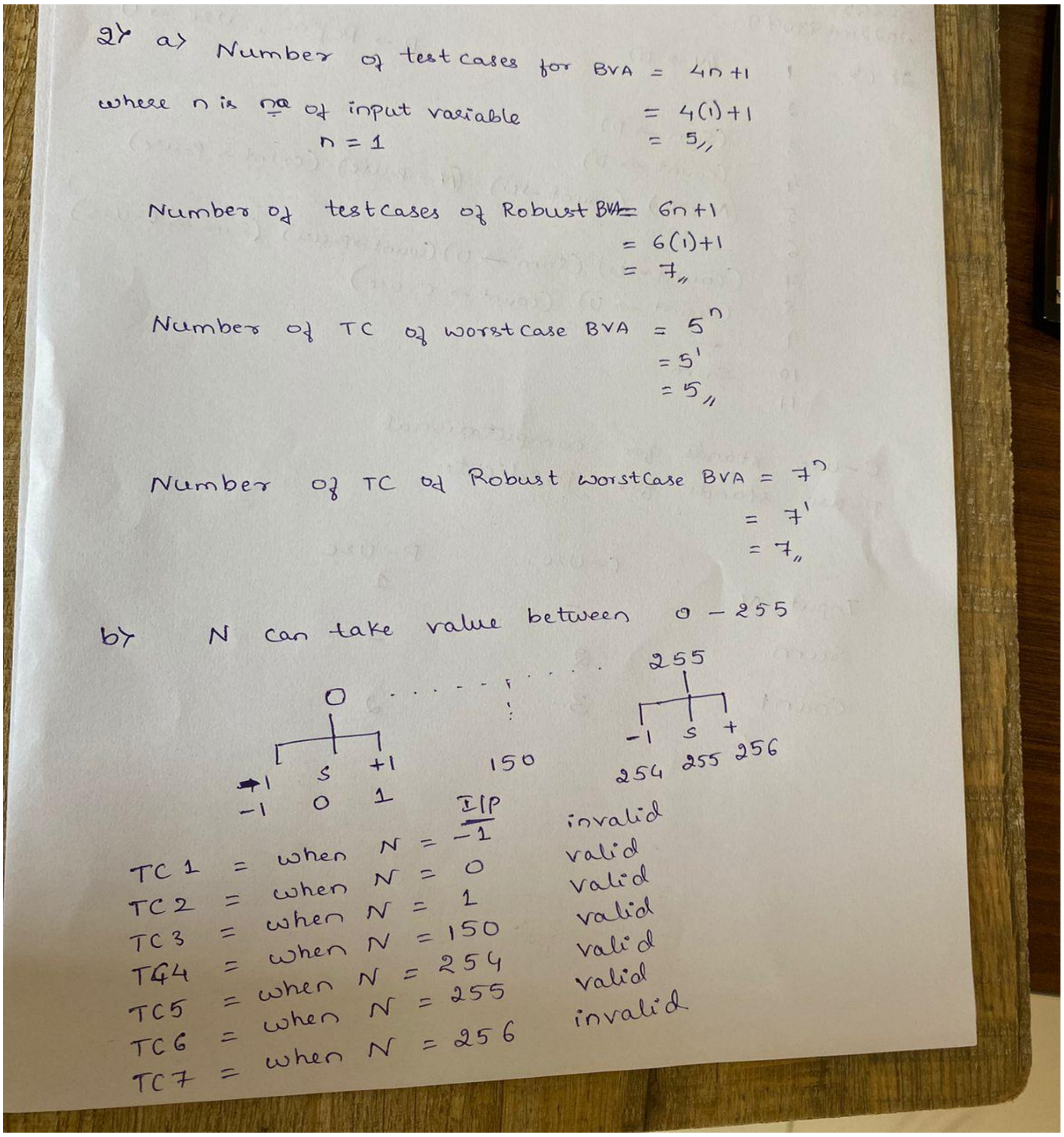
Question No: 03

Consider the following algorithm that calculates the sum of all even natural numbers between 1 to N. Answer the following questions:

```
int evenNaturalSum(int N)
{
 int sum = 0;
int count = 1;
while(count<=N)
{
if (count%2==0) {sum = sum + count;}
count++;
}
return sum;
}
```

a) Assume that the range of signed int is 2^8=256. Then N can take values only in range 0 to 255 [including end points]. Calculate the number of test cases - Normal BVA, Robust BVA, Worst case BVA and Robust worst case BVA.

b) Write all the test cases for Robust BVA for the above algorithm.



2) a) Number of test cases for BVA = $4n+1$

where n is no of input variable
$n = 1$

$= 4(1)+1$
$= 5$

Number of testcases of Robust ~~BVA~~ = $6n+1$
$= 6(1)+1$
$= 7$

Number of TC of worst case BVA = 5^n
$= 5^1$
$= 5$

Number of TC of Robust worstCase BVA = 7^n
$= 7^1$
$= 7$

b) N can take value between 0 – 255

0 . . . . . . 150 255

0: -1, s, +1 → -1, 0, 1

150

255: -1, s, + → 254, 255, 256

		I/P	
TC 1	= when	N = -1	invalid
TC 2	= when	N = 0	valid
TC 3	= when	N = 1	valid
TC 4	= when	N = 150	valid
TC 5	= when	N = 254	valid
TC 6	= when	N = 255	valid
TC 7	= when	N = 256	invalid

min value : 100

close to min : 101

nominal : 300

close to max : 499

max : 500

lesser than min value : 99

larger than max value : 501

c) Identify all DU pairs for the variables sum, count and N. For all the use nodes, specify whether it is a C-Use or P-Use and why?

D-> def

u : use

i) N : DU : n

Variable: N

- Def: N (input parameter) DU -> 1,5
- Use: count <= N
- DU Pair:
    i. (Def) N (input parameter) --> (Use) count <= N
- Use Node Type: P-Use (Predicate)

ii) count DU -> 4,8 | 4,5

<4,5> : Predicate

<4,7> : Predicate, Computation

<4,8>: Computation

<8,5> : Computation

<8,7>: Computation

<8,8>: Computation

iii) sum

Sum:

<3,7> : Computation

<7,7>: Computation

Question 4

Select a structure-based technique according to a given project situation and justify your answer.

You are the Senior Test Analyst working on the testing of Air traffic control software that helps to prevent collisions, organize and expedite the flow of air traffic, and provide information and other support for pilots. A failure analysis has shown that if the software system fails then it may risk the life of thousands of travelers. The government has requested that the level of testing for this software exceeds that normally required by the relevant regulatory standards.

Which is the level of test coverage you would expect to be achieved in of the control software? Also, justify your answer. [5 marks]

A. Multiple Condition coverage

B. Branch coverage + Modified Condition/Decision coverage

C. Branch coverage + Statement coverage

D. Modified Condition/Decision coverage

B. Branch coverage + Modified Condition/Decision coverage

Justification:

1. Branch coverage: This technique ensures that all branches (decision points) in the software code are executed at least once during testing. In the context of ATC software, it is crucial to test all decision points thoroughly to cover different scenarios and ensure that critical decisions, such as collision

prevention and traffic flow organization, are handled correctly. Branch coverage helps identify potential control flow issues, missing conditions, or unintended behavior within the software.

2. Modified Condition/Decision coverage (MC/DC): MC/DC is an advanced testing technique that focuses on ensuring that each individual condition within a decision has been tested independently. It requires exercising different combinations of true/false outcomes for each condition in a decision to test all possible combinations. MC/DC helps identify potential errors or inconsistencies in decision-making logic, ensuring that even subtle flaws in the software's decision points are detected. Given the potential risk to life, it is crucial to achieve a high level of confidence in the decision-making capabilities of the ATC software.

Question 5

Class Integer Stack provides implementation of Stack data structure, with the implementation of two methods: push() and pop(). Moreover, classes OddIntegerStack and EvenIntegerStack extend the class IntegerStack and overrides the implementation of the push() method. Class Odd Integer Stack is shown below:

```
public class OddIntegerStack extends IntegerStack{


public void push (int element)

{

if (element % 2 == 0)

{

System.out.println("Cannot push even element onto an odd stack");


return;

}
If  (top != maxSize-1){

arrayStack [++top] =element;
}
else{ System.out.println("Overflow error");}
}
}
```

[2+2+3+3 marks]

a) Write test cases to test the push()method of the Odd IntegerStack class. Assume maxSize =5

Here are all the test cases to test the `push()` method of the `OddIntegerStack` class:

1. Test pushing an odd element onto an empty stack:
   - Input: element = 3
   - Expected Output: The element 3 should be pushed onto the stack successfully.
2. Test pushing an odd element onto a non-empty stack:
   - Input: element = 5, existing stack = [1, 3]
   - Expected Output: The element 5 should be pushed onto the stack successfully.
3. Test pushing an even element onto an empty stack:
   - Input: element = 2
   - Expected Output: The method should print "Cannot push even element onto an odd stack" and the stack should remain empty.
4. Test pushing an even element onto a non-empty stack:
   - Input: element = 4, existing stack = [1, 3]
   - Expected Output: The method should print "Cannot push even element onto an odd stack" and the stack should remain unchanged.
5. Test pushing an odd element when the stack is already full:
   - Input: element = 7, existing stack = [1, 3, 5, 9, 11]
   - Expected Output: The method should print "Overflow error" and the stack should remain unchanged.

b) Write test cases to test the pop()method of the OddIntegerStack class. Assume maxSize = 5.

1. Test popping an element from a non-empty stack:
   - Input: stack = [3, 5, 7]
   - Expected behavior: The top element (7) should be removed from the stack and returned.
2. Test popping an element from an empty stack:
   - Input: stack = []
   - Expected behavior: The method should print the message "Underflow error" and return a default value (e.g., -1).

c) An object reference of class IntegerStack can be used to store an object of class

Even Integer Stack or OddIntegerStack. Write test cases to test this polymorphic behaviour.

IntegerStack obj2 = new OddIntegerStack();

1. **Test polymorphic behavior using a reference of type `IntegerStack` to store an object of type** OddIntegerStack**:**
   - **Create an `IntegerStack` reference and assign it an instance of `EvenIntegerStack`.**

     IntegerStack obj1 = new OddIntegerStack**();**

   - **Call methods `push()` and `pop()` on the reference, passing appropriate inputs.**

     obj1 .push(1); =>
2.

     obj1 .pop();

   - **Verify that the `push()` and `pop()` methods of `EvenIntegerStack` are correctly executed.**
   -

d) Discuss, in brief, how you would carry out integration testing of such a system.

To carry out integration testing of the system consisting of the `IntegerStack`, `OddIntegerStack`, and `EvenIntegerStack` classes, you can follow these steps:

1. Identify the integration points: Determine the points where the classes interact with each other. In this case, the integration points are the inheritance relationship between `IntegerStack` and its subclasses (`OddIntegerStack` and `EvenIntegerStack`).
2. Define integration test scenarios: Based on the identified integration points, define test scenarios that cover the interactions between the classes. Some scenarios may include:
    - Testing the inheritance relationship: Verify that the subclasses inherit the methods and attributes correctly from the `IntegerStack` class.
    - Testing method overriding: Ensure that the overridden `push()` method in `OddIntegerStack` and `EvenIntegerStack` behaves as expected and does not interfere with the functionality of the base class's `push()` method.
    - Testing polymorphic behavior: Verify that an `IntegerStack` reference can store objects of both `OddIntegerStack` and `EvenIntegerStack`, and the appropriate methods are called based on the actual object type.
3. Create integration test cases: Based on the defined scenarios, design and implement specific integration test cases. For example:
    - Test that calling the `push()` method on an `OddIntegerStack` object with an odd element correctly adds the element to the stack.
    - Test that calling the `push()` method on an `OddIntegerStack` object with an even element prints the error message and does not modify the stack.
    - Test that calling the `pop()` method on an `IntegerStack` reference assigned to an `EvenIntegerStack` object correctly removes and returns the top element from the stack.
4. Execute integration tests: Run the integration test cases and observe the behavior of the system. Ensure that the interactions between the classes function as expected and that the integration points are properly integrated.
5. Analyze results: Verify whether the integration tests pass or fail. If any issues or failures are encountered, investigate and debug the code to address the problems. Repeat the integration testing process as necessary until all integration issues are resolved.