Q1.

i) a)

To determine the minimum tests to achieve 100% decision/condition coverage for the given scenario, we need to identify the decision points or conditions involved. Based on the information provided, we have two conditions: the light being red (RED) and the front wheels of the car being over the line marking the beginning of the intersection (WHEELS).

Let's analyze the conditions and their possible outcomes:

Condition 1: RED (Light is red)

Possible outcomes: True, False
Condition 2: WHEELS (Front wheels over the line)

Possible outcomes: True, False
To achieve 100% decision/condition coverage, we need to cover all possible combinations of these conditions. Here are the minimum test cases required:

Test Case 1:

Condition 1 (RED): True
Condition 2 (WHEELS): True
Test Case 2:

Condition 1 (RED): True
Condition 2 (WHEELS): False
Test Case 3:

Condition 1 (RED): False
Condition 2 (WHEELS): True
Test Case 4:

Condition 1 (RED): False
Condition 2 (WHEELS): False

By testing all four possible combinations of the conditions, we achieve 100% decision/condition coverage for the given scenario.

It's important to note that the provided minimum test cases assume that there are no other conditions or constraints in the code that affect the outcome. If there are additional conditions or dependencies, further test cases may be required to achieve complete coverage.

ii)

b)

Modified condition/decision coverage (MC/DC) is a software testing methodology that aims to ensure that each condition in a decision statement has been evaluated both to true and false, and that each decision has taken all possible outcomes. MC/DC is particularly useful for safety-critical systems where a failure could have severe consequences.

Here are the key principles and steps involved in the MC/DC methodology:

1. Decision Coverage: The first step is to achieve 100% decision coverage, which means that every decision point in the software should be exercised during testing. This involves creating test cases that cover all possible outcomes of each decision.
2. Condition Coverage: Once decision coverage is achieved, the next step is to focus on condition coverage. Each condition within a decision should be evaluated to both true and false during testing. This ensures that all possible combinations of conditions are tested.
3. Modifying Conditions: To achieve MC/DC, modifications are made to the conditions within the decision. A single condition is changed while keeping the other conditions constant, and test cases are created to exercise this modified condition.
4. Evaluating Modified Conditions: The test cases created in the previous step are executed to evaluate the modified condition. This helps determine if the decision behaves correctly when a specific condition is changed.
5. Observability of Outcome: It is important to ensure that the outcome of the decision is observable during testing. This means that the test environment should be set up to capture and record the outcome of each decision.
6. Independent Evaluation: Each condition should be able to independently affect the outcome of the decision. This means that changing the value of one condition should not impact the evaluation of other conditions.
7. Traceability: It is crucial to have traceability between the test cases and the conditions being tested. This allows for easy identification and analysis of the specific conditions covered by each test case.

By following these steps, the MC/DC methodology helps in achieving thorough testing of decision statements and their conditions, ensuring that potential faults and errors are detected. This methodology is often used in safety-critical industries such as aviation, automotive, and medical devices where the consequences of software failures can be critical.

To achieve 100% modified condition/decision coverage, we need to test all possible combinations of the conditions RED, SPEED, and WHEELS. There are 8 possible combinations, so we need 8 test cases. The following sets of values provide the minimum tests to achieve 100% modified condition/decision coverage:

- RED = True, SPEED = True, WHEELS = True
- RED = True, SPEED = True, WHEELS = False
- RED = True, SPEED = False, WHEELS = True
- RED = True, SPEED = False, WHEELS = False
- RED = False, SPEED = True, WHEELS = True
- RED = False, SPEED = True, WHEELS = False
- RED = False, SPEED = False, WHEELS = True
- RED = False, SPEED = False, WHEELS = False

These test cases will ensure that all possible combinations of the conditions are tested, and that each condition is shown to independently affect the outcome of the decision.

Here is a more detailed explanation of each test case:

- Test Case 1: RED = True, SPEED = True, WHEELS = True This test case ensures that the photo is taken when the signal light is red, the car is speeding, and the front wheels of the car are over the line marking the beginning of the intersection.
- Test Case 2: RED = True, SPEED = True, WHEELS = False This test case ensures that the photo is taken when the signal light is red, the car

is speeding, and the front wheels of the car are not over the line marking the beginning of the intersection.

- Test Case 3: RED = True, SPEED = False, WHEELS = True This test case ensures that the photo is taken when the signal light is red, the car is not speeding, and the front wheels of the car are over the line marking the beginning of the intersection.
- Test Case 4: RED = True, SPEED = False, WHEELS = False This test case ensures that the photo is taken when the signal light is red, the car is not speeding, and the front wheels of the car are not over the line marking the beginning of the intersection.
- Test Case 5: RED = False, SPEED = True, WHEELS = True This test case ensures that the photo is taken when the signal light is green, the car is speeding, and the front wheels of the car are over the line marking the beginning of the intersection.
- Test Case 6: RED = False, SPEED = True, WHEELS = False This test case ensures that the photo is not taken when the signal light is green, the car is speeding, and the front wheels of the car are not over the line marking the beginning of the intersection.
- Test Case 7: RED = False, SPEED = False, WHEELS = True This test case ensures that the photo is not taken when the signal light is green, the car is not speeding, and the front wheels of the car are over the line marking the beginning of the intersection.
- Test Case 8: RED = False, SPEED = False, WHEELS = False This test case ensures that the photo is not taken when the signal light is green, the car is not speeding, and the front wheels of the car are not over the line marking the beginning of the intersection.

By testing all of these combinations, we can be sure that the photo is taken in all of the cases where it should be taken, and that it is not taken in any of the cases where it should not be taken. This ensures that the system is working as intended.

c)

Multiple condition coverage is a software testing methodology that focuses on ensuring that all possible combinations of conditions in a decision statement have been tested. It is also known as "all-terms" or "all-pairs" testing. The goal of multiple condition coverage is to identify faults that may occur due to the interaction of multiple conditions.

To achieve 100% multiple condition coverage, we need to test all possible combinations of the conditions RED, SPEED, and WHEELS. There are 8 possible combinations, but we can reduce this to 5 test cases by using the following sets of values:

- RED = True, SPEED = True, WHEELS = True : Photo taken : T
- RED = True, SPEED = True, WHEELS = False : Photo taken : F
- RED = False, SPEED = True, WHEELS = True : Photo taken : T
- RED = False, SPEED = False, WHEELS = True : Photo taken : T
- RED = False, SPEED = False, WHEELS = False : Photo taken : F

These test cases will ensure that all possible combinations of the conditions are tested, and that each condition is shown to independently affect the outcome of the decision.

Here is a more detailed explanation of each test case:

- Test Case 1: RED = True, SPEED = True, WHEELS = True This test case ensures that the photo is taken when the signal light is red, the car is speeding, and the front wheels of the car are over the line marking the beginning of the intersection.

- Test Case 2: RED = True, SPEED = True, WHEELS = False This test case ensures that the photo is taken when the signal light is red, the car is speeding, and the front wheels of the car are not over the line marking the beginning of the intersection.
- Test Case 3: RED = False, SPEED = True, WHEELS = True This test case ensures that the photo is taken when the signal light is green, the car is speeding, and the front wheels of the car are over the line marking the beginning of the intersection.
- Test Case 4: RED = False, SPEED = False, WHEELS = True This test case ensures that the photo is not taken when the signal light is green, the car is not speeding, and the front wheels of the car are over the line marking the beginning of the intersection.
- Test Case 5: RED = False, SPEED = False, WHEELS = False This test case ensures that the photo is not taken when the signal light is green, the car is not speeding, and the front wheels of the car are not over the line marking the beginning of the intersection.

By testing all of these combinations, we can be sure that the photo is taken in all of the cases where it should be taken, and that it is not taken in any of the cases where it should not be taken. This ensures that the system is working as intended.

The reason we can reduce the number of test cases from 8 to 5 is because of the way that multiple condition coverage works. Multiple condition coverage requires that all possible combinations of the conditions are tested, but it does not require that each condition be tested in every combination. In this case, we can achieve 100% multiple condition coverage by testing all combinations of the conditions where the photo is taken, and all combinations of the conditions where the photo is not taken.

d)

To achieve 100% path coverage with the given data flow diagram and conditions where a photo should be taken if the signal light is red (RED) or the car is speeding (SPEED), and the front wheels of the car are over the line marking the beginning of the intersection (WHEELS), we need to consider all possible paths through the data flow.

Here is a textual representation of the data flow diagram:

```
Start
     |
    V
   / \
  V   V
RED   SPEED
  \   /
   V V
  WHEELS
   |
   V
 Take Photo
   |
   V
   End
```

```
    Start
      |
      V
  RED = True?
 /   \
V     X
|     |
V     V
SPEED = True?
 /   \
V     X
|     |
V     V
WHEELS = True?
 /   \
V     X
|     |
V     V
Take Photo
      |
    V
   End
```

Based on the data flow diagram, we can identify the following paths:

Path 1: Start -> RED -> WHEELS -> Take Photo -> End

Path 2: Start -> SPEED -> WHEELS -> Take Photo -> End

To achieve 100% path coverage, we need to cover both paths with our tests. Here are the minimum test cases required:

Test Case 1:

- Start -> RED = True -> WHEELS = True -> Take Photo -> End

Test Case 2:

- Start -> SPEED = True -> WHEELS = True -> Take Photo -> End

By testing both paths through the data flow, we achieve 100% path coverage for the given scenario.

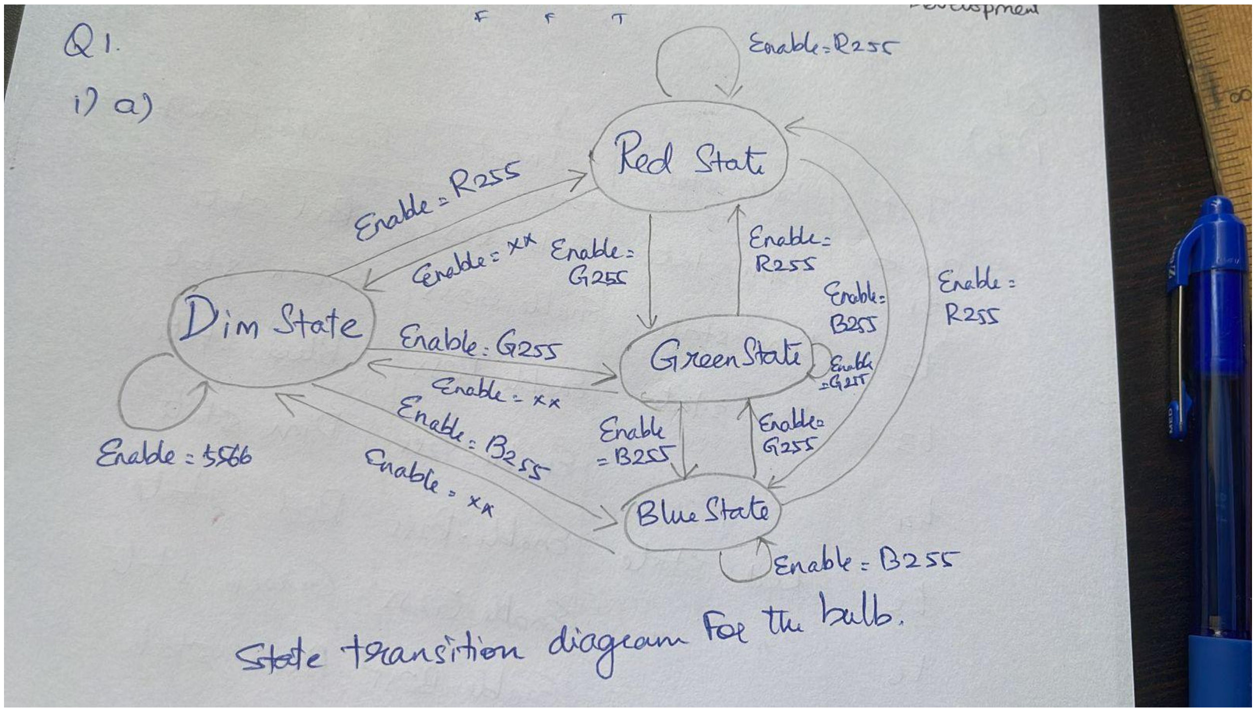
It's important to note that the provided minimum test cases assume that there are no other conditions or dependencies in the code that affect the outcome. If there are additional paths or constraints, further test cases may be required to achieve complete path coverage.

—--------------------

Q2

i) a)

Here is the state transition diagram for the IOT-enabled bulb:



```
R255 Start ------------> Red State | | | | V | Green State <--------- | | | |
V | Blue State <---------- | | V Dim State (Default)
```

In the state transition diagram:

- The Start state represents the initial state of the bulb.
- The arrows indicate the possible transitions between states based on the input signals received via Wi-Fi.
- The Enable = R255 signal transitions the bulb to the Red state.
- The Enable = G255 signal transitions the bulb to the Green state.
- The Enable = B255 signal transitions the bulb to the Blue state.
- Any other input signal at the Enable transitions the bulb to the Dim state, which is the default state.

This state transition diagram illustrates the possible states of the IOT-enabled bulb and the transitions between them based on the input signals received.

ii) b)

To test all the states of the IoT-enabled bulb based on the input signals and the "Dim state" condition, we can design the following test cases:

1. Test Case: Enable = R255 (Red state)
   - Input: Enable = R255
   - Expected Output: The bulb should change its state to red.
2. Test Case: Enable = G255 (Green state)
   - Input: Enable = G255
   - Expected Output: The bulb should change its state to green.
3. Test Case: Enable = B255 (Blue state)
   - Input: Enable = B255
   - Expected Output: The bulb should change its state to blue.
4. Test Case: Enable = Y255 (Dim state)
   - Input: Enable = Y255 (input other than R, G, or B)
   - Expected Output: The bulb should enter a dim state.
5. Test Case: Enable = "" (Dim state)
   - Input: Enable = "" (empty input)
   - Expected Output: The bulb should enter a dim state.
6. Test Case: Enable = R200 (Dim state)
   - Input: Enable = R200 (valid color with a different intensity)
   - Expected Output: The bulb should enter a dim state.
7. Test Case: Enable = G0 (Dim state)
   - Input: Enable = G0 (valid color with intensity zero)
   - Expected Output: The bulb should enter a dim state.
8. Test Case: Enable = B300 (Dim state)
   - Input: Enable = B300 (valid color with intensity beyond 255)

  - Expected Output: The bulb should enter a dim state.

9. Test Case: Enable = R255, then Enable = G255 (Red state followed by Green state)
   - Input: Enable = R255, Enable = G255
   - Expected Output: The bulb should change its state to red and then to green.
10. Test Case: Enable = B255, then Enable = R255, then Enable = G255 (Blue state followed by Red state followed by Green state)
    - Input: Enable = B255, Enable = R255, Enable = G255
    - Expected Output: The bulb should change its state to blue, then to red, and finally to green.

These test cases cover various input scenarios to test all the states of the IoT-enabled bulb, including valid color states (red, green, blue), the "Dim state" for invalid colors or intensity values, and transitions between different states.

These test cases cover the different input signals to transition the bulb between states as described in the provided information. It's important to verify that the bulb transitions to the correct state based on the input signal. Additionally, you may consider testing the behavior of the bulb in each state, such as testing the brightness or color emitted by the bulb in each state.

Remember to ensure that you have the necessary setup to simulate or send the input signals to the bulb for testing, such as a functioning Wi-Fi connection or a simulated environment.

—---

Q3.

a) The range of valid inputs for N is 0 to 255, inclusive.

1. Normal BVA:
    - Range: 1 to 254
    - Number of test cases: 254 - 1 + 1 = 254
2. Robust BVA:
    - Range: 0 to 255
    - Number of test cases: 255 - 0 + 1 = 256
3. Worst case BVA:
    - Range: 0 to 255 (same as Robust BVA)
    - Number of test cases: 256
4. Robust worst case BVA:
    - Range: -1 to 256
    - Number of test cases: 256 - (-1) + 1 = 258

b) Test cases for Robust BVA:

Considering the range of 0 to 255, the following test cases cover the Robust BVA scenario:

1. Test case 1: N = 0 (minimum valid input)
2. Test case 2: N = 1 (minimum valid input + 1)
3. Test case 3: N = 254 (maximum valid input - 1)
4. Test case 4: N = 255 (maximum valid input)
5. Test case 5: N = -1 (invalid input)
6. Test case 6: N = 256 (invalid input + 1)

These test cases cover the boundary values as well as invalid inputs for the Robust BVA scenario.

c) DU pairs and Use Nodes:

1. Variable: sum
   - DU Pairs: (declaration, initialization) -> (increment operation)
   - Use Nodes: C-Use in the increment operation because it is used to calculate the new value of 'sum'.
2. Variable: count
   - DU Pairs: (declaration, initialization) -> (increment operation)
   - Use Nodes: C-Use in the increment operation because it is used to calculate the new value of 'count'.
3. Variable: N
   - DU Pairs: (parameter passing)
   - Use Nodes: P-Use as it is used as a parameter in the function call.

In this case, the DU (Definition-Use) pairs for 'sum' and 'count' are created during the declaration, initialization, and increment operations. The variable 'N' is a parameter and its value is used as a P-Use in the function call.
In the given algorithm, the variables sum, count, and N are used in different contexts. Let's identify all the Def-Use (DU) pairs for these variables and specify whether each use node is a C-Use (Computation) or P-Use (Predicate) and why.

DU pairs for the variables:

1. Variable: sum
   - Def: sum = sum + count;
   - Use: sum (in the return statement)
2. DU Pair:
   - (Def) sum = sum + count; --> (Use) sum (return statement)

Use Node Type: C-Use (Computation)

3. Explanation: The variable "sum" is used to store the result of the computation (sum = sum + count) and is later used in the return statement. Hence, it is a C-Use.
4. Variable: count
   - Def: count = 1;
   - Use: count <= N, count++
5. DU Pairs:
   - (Def) count = 1; --> (Use) count <= N
   - (Def) count = 1; --> (Use) count++

Use Node Types: P-Use (Predicate), C-Use (Computation)

6. Explanation:
    - The variable "count" is used as a predicate in the condition count <= N, which determines the loop continuation. Hence, it is a P-Use.
    - The variable "count" is also used in the computation count++. It is incremented during each iteration of the loop. Hence, it is a C-Use.
7. Variable: N
    - Def: N (input parameter)
    - Use: count <= N
8. DU Pair:
    - (Def) N (input parameter) --> (Use) count <= N

Use Node Type: P-Use (Predicate)

9. Explanation: The variable "N" is used as a predicate in the condition count <= N, which determines the loop continuation. Hence, it is a P-Use.

To summarize:

Variable: sum

- DU Pair: (Def) sum = sum + count; --> (Use) sum (return statement)
- Use Node Type: C-Use (Computation)

Variable: count

DU Pairs: (Def) count = 1; --> (Use) count <= N

- (Def) count = 1; --> (Use) count++
- Use Node Types: P-Use (Predicate), C-Use (Computation)

Variable: N

- DU Pair: (Def) N (input parameter) --> (Use) count <= N
- Use Node Type: P-Use (Predicate)

Note: In DU analysis, C-Use refers to the use of a variable in a computation (such as arithmetic operations), while P-Use refers to the use of a variable in a predicate (such as conditions or loop controls).

—-------------------------------------

Q4.

Given the critical nature of the Air Traffic Control (ATC) software and the potential risk it poses to the lives of thousands of travelers, it is of utmost importance to ensure a comprehensive and thorough testing approach. Among the provided options, the most suitable structure-based testing technique for this project situation is:

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.

The combination of branch coverage and MC/DC provides a robust testing approach for the ATC software. It ensures that all decision points are tested, covering different scenarios and potential branches within the code. Additionally, MC/DC ensures thorough testing of each condition's outcome, which is crucial for critical systems where even minor errors in decision-making can have severe consequences.

The government's request for testing the ATC software beyond standard regulatory standards indicates the need for a higher level of assurance. By adopting both branch coverage and MC/DC, the testing team can achieve a more rigorous level of test coverage, reducing the likelihood of undetected defects and minimizing the risk to the lives of travelers using the system.

In summary, the combination of branch coverage and MC/DC provides an appropriate level of test coverage for the ATC software. It ensures comprehensive testing of decision points and conditions, helping to identify potential flaws in control flow and decision-making logic. This approach aligns with the criticality of the system and the government's request for heightened testing measures to mitigate risks and protect the lives of thousands of travelers.

Structure-based testing techniques, also known as white-box testing techniques, focus on deriving test cases based on the internal structure of the software being tested. These techniques aim to ensure that all paths, conditions, and statements within the software are exercised to achieve high test coverage. Some commonly used structure-based testing techniques include:

1. Statement coverage: This technique ensures that each statement in the software code is executed at least once during testing. Test cases are designed to cover all statements, providing a basic level of coverage. However, it does not guarantee comprehensive testing of all possible scenarios.
2. Branch coverage: Branch coverage focuses on testing all possible decision outcomes within the code. It aims to ensure that each branch (decision point) in the code is executed at least once. By testing both true and false outcomes of each decision, potential issues related to control flow and decision-making can be identified.
3. Condition coverage: Condition coverage aims to test all possible outcomes of each individual condition within a decision. It ensures that each condition within a decision has been evaluated to true and false independently, providing more thorough coverage than branch coverage alone.
4. Path coverage: Path coverage involves testing all possible paths through the software code. It aims to exercise every possible combination of branches and statements, providing a higher level of coverage than statement or branch

coverage alone. Path coverage can be time-consuming and challenging to achieve for complex software.

5. Modified Condition/Decision coverage (MC/DC): MC/DC is an advanced technique that focuses on **testing each individual condition within a decision independently**. It ensures that each condition has been evaluated to both true and false while other conditions remain unchanged. MC/DC helps identify subtle errors or inconsistencies in decision-making logic.

When selecting a structure-based testing technique, it is important to consider the nature of the software, the level of risk associated with its failure, and the level of test coverage required. A combination of techniques, such as branch coverage along with MC/DC, often provides a more comprehensive testing approach, particularly for critical systems where the potential impact of failure is high.

It is worth noting that structure-based testing techniques complement other testing techniques, such as black-box testing techniques that focus on testing the functionality and behavior of the software without considering its internal structure. A combination of different testing techniques can help achieve thorough test coverage and increase confidence in the reliability and quality of the software.

Web Application:

1. Consider a web application that allows users to sign up and log in. The application has different code branches to handle user authentication and authorization. Structure-based testing techniques, such as branch coverage, can be used to ensure that all possible branches within the authentication and

authorization logic are thoroughly tested. By designing test cases to cover different scenarios, including valid and invalid credentials, expired sessions, and various user roles, the testing team can verify that the application handles authentication and authorization correctly.

Financial Software:

2. Financial software often involves complex calculations and decision-making logic. Structure-based testing techniques, such as condition coverage and path coverage, can be used to ensure accurate calculation results and proper handling of various conditions. For example, in accounting software, different tax rates may apply based on income brackets. Test cases can be designed to cover different income ranges and verify that the correct tax calculation and handling of conditions, such as deductions or exemptions, are properly implemented.

Embedded Systems:

3. Embedded systems control critical hardware and perform essential functions in various domains, such as automotive, medical devices, or industrial control systems. Structure-based testing techniques play a crucial role in ensuring the reliability and safety of these systems. For example, in an automotive engine control unit (ECU), branch coverage and condition coverage can be applied to test different control paths and validate the behavior of the ECU under various conditions, such as engine temperature, speed, or sensor inputs.

Compiler/Interpreter:

4. A compiler or interpreter is responsible for translating or executing programming code. Structure-based testing techniques are valuable in testing these software components to ensure proper code interpretation and execution. For example, in a compiler, statement coverage and path coverage can be applied to test different language constructs, error handling, and control flow. Test cases can be designed to cover different language features, such as loops, conditionals, and variable declarations, to verify the correctness of code translation or execution.

Gaming Software:

5. Gaming software often has complex game logic, including different levels, scenarios, and interactions. Structure-based testing techniques can be used to ensure that all possible paths, decision points, and outcomes are thoroughly tested. For instance, branch coverage and path coverage can be applied to test different game states, player actions, and win/lose conditions. Test cases can be designed to cover different gameplay scenarios and verify that the game behaves as expected under various conditions.

These are just a few examples where structure-based testing techniques can be applied to improve the quality and reliability of software. By analyzing the internal structure and logic of the software and designing test cases accordingly, you can achieve higher test coverage and uncover potential issues or vulnerabilities that might not be discovered through other testing techniques alone.

Q5.

a) 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 behavior: The element should be pushed onto the stack successfully.
2. Test pushing an even element onto an empty stack:
    - Input: element = 4
    - Expected behavior: The method should print the message "Cannot push even element onto an odd stack" and not push the element onto the stack.
3. Test pushing an odd element onto a non-empty stack:
    - Input: element = 7, stack = [3, 5]
    - Expected behavior: The element should be pushed onto the stack successfully.
4. Test pushing an element onto a full stack:
    - Input: element = 9, stack = [1, 3, 5, 7, 9]
    - Expected behavior: The method should print the message "Overflow error" and not push the element onto the stack.

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.
6. Test pushing an odd element when the stack has one empty space left:
    - Input: element = 13, existing stack = [1, 3, 5, 9]
    - Expected Output: The element 13 should be pushed onto the stack successfully.
7. Test pushing multiple odd elements until the stack is full:
    - Input: elements = [1, 3, 5, 7, 9], existing stack = []
    - Expected Output: Each element should be pushed onto the stack successfully, and the stack should be [1, 3, 5, 7, 9].
8. Test pushing a mix of odd and even elements onto an empty stack:
    - Input: elements = [1, 2, 3]
    - Expected Output: The method should print "Cannot push even element onto an odd stack" when trying to push 2, and the stack should only contain [1, 3].
9. Test pushing a mix of odd and even elements onto a non-empty stack:
    - Input: elements = [1, 4, 3], existing stack = [5, 7]
    - Expected Output: The method should print "Cannot push even element onto an odd stack" when trying to push 4, and the stack should remain unchanged.
10. Test pushing elements until the stack overflows:
- Input: elements = [1, 3, 5, 7, 9, 11], existing stack = [2, 4]
- Expected Output: The method should print "Overflow error" when trying to push 11, and the stack should remain unchanged.

These test cases cover various scenarios to ensure that the `push()` method of the `OddIntegerStack` class behaves as expected, including handling odd and even elements, stack overflow conditions, and interactions with an existing stack.

b) Test cases to test the pop() method of the OddIntegerStack class:

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) Test cases to test polymorphic behavior:

1. Test storing an OddIntegerStack object in an IntegerStack reference:
    - Input: stack = new IntegerStack(), oddStack = new OddIntegerStack()
    - Expected behavior: The oddStack object should be assigned to the stack reference.
2. Test storing an EvenIntegerStack object in an IntegerStack reference:
    - Input: stack = new IntegerStack(), evenStack = new EvenIntegerStack()
    - Expected behavior: The evenStack object should be assigned to the stack reference.

To test the polymorphic behavior of the `IntegerStack` class and its subclasses `EvenIntegerStack` and `OddIntegerStack`, you can perform the following test cases:

1. **Test polymorphic behavior using a reference of type `IntegerStack` to store an object of type `EvenIntegerStack`:**
    - **Create an `IntegerStack` reference and assign it an instance of `EvenIntegerStack`.**
    - **Call methods `push()` and `pop()` on the reference, passing appropriate inputs.**
    - **Verify that the `push()` and `pop()` methods of `EvenIntegerStack` are correctly executed.**
2. **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 `OddIntegerStack`.**
    - **Call methods `push()` and `pop()` on the reference, passing appropriate inputs.**
    - **Verify that the `push()` and `pop()` methods of `OddIntegerStack` are correctly executed.**
3. **Test polymorphic behavior by storing both `EvenIntegerStack` and `OddIntegerStack` objects in an array of type `IntegerStack`:**

- **Create an array of `IntegerStack` and initialize it with instances of `EvenIntegerStack` and `OddIntegerStack`.**
- **Iterate over the array and call methods `push()` and `pop()` on each element.**
- **Verify that the corresponding `push()` and `pop()` methods of each subclass are correctly executed based on the actual object type.**

These test cases will help verify that the polymorphic behavior of the `IntegerStack` class and its subclasses is functioning correctly, allowing an `IntegerStack` reference to store objects of both `EvenIntegerStack` and `OddIntegerStack` and invoking the appropriate methods based on the actual object type.

d) Integration testing of the system:

Integration testing for the given system, which involves multiple classes and their interactions, can be carried out using the following approach:

1. Identify key interactions: Determine the main interactions between the classes, such as method calls and data exchanges, to understand the flow of information and dependencies.
2. Define test scenarios: Create test scenarios that cover different combinations of interactions, focusing on critical paths and boundary cases. For example, test scenarios could include pushing and popping elements in different sequences, handling stack overflow and underflow conditions, and testing the polymorphic behavior.
3. Set up test environment: Prepare the necessary test environment, including creating instances of the relevant classes and initializing any required variables or data structures.
4. Execute test cases: Execute the defined test scenarios and observe the behavior of the system. Verify that the expected results are obtained and that the system behaves correctly under different conditions.
5. Validate results: Compare the actual results with the expected results for each test case. Identify any discrepancies or failures and analyze the root causes.
6. Debug and fix issues: If any failures or discrepancies are found, debug the system to identify the root causes of the issues. Once identified, fix the issues and repeat the testing process to ensure the fixes are effective.

7. Repeat and iterate: Iterate the integration testing process, incorporating additional test cases and scenarios to achieve thorough coverage of the system's functionality and interactions.

By following these steps, integration testing can help ensure that the different components of the system, such as the IntegerStack, OddIntegerStack, and EvenIntegerStack classes, work together correctly and fulfill the overall requirements of the software.

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.

Integration testing helps ensure that the interactions and collaborations between the classes work correctly, providing a comprehensive evaluation of the system's behavior as a whole. By focusing on the integration points and verifying the correct functioning of inherited methods, overridden methods, and polymorphic behavior, you can gain confidence in the overall functionality and coherence of the system.