# WHITE BOX TESTING

White box testing, also known as structural testing or glass box testing, is a software testing technique that focuses on the internal structure, implementation details, and logic of the software. Unlike black box testing, which examines the software from an external perspective without knowledge of its internal workings, white box testing delves into the code, algorithms, and system architecture.

The main objective of white box testing is to uncover defects and ensure that the software functions correctly according to its design and specifications. By analyzing the internal structure, white box testing aims to verify the proper execution of individual code statements, decision branches, and overall program flow.

In white box testing, various coverage methods are utilized to ensure comprehensive testing of the codebase. Here are some commonly employed coverage techniques:

# Code Coverage:

Code coverage is a metric used to measure the extent to which the source code of a program is tested. It helps assess the effectiveness of the test suite in terms of code execution. Code coverage can be measured at different levels, such as statement, branch, and path coverage.

# Method Coverage:

Method coverage focuses on testing individual methods or functions within the code. The goal is to ensure that all methods are called and executed during testing. By achieving high method coverage, you can verify the correctness and behavior of each method in isolation.

# Branch Coverage:

Branch coverage aims to test all possible outcomes of conditional statements or decision points in the code. It ensures that both the true and false branches of each decision are executed at least once. This technique helps uncover potential logical errors and ensures that the software behaves as expected under different conditions.

# Statement Coverage:

Statement coverage involves testing each statement in the code at least once during the testing process. It ensures that all statements are executed and helps identify dead code (unreachable code) or code segments that may have been inadvertently skipped during development.

# Condition Coverage:

Condition coverage focuses on evaluating the different Boolean conditions within the code. It aims to test all possible combinations of conditions and their outcomes. By doing so, it helps uncover issues related to complex conditional logic, such as missing conditions, incorrect Boolean operators, or inconsistent behavior.

These coverage techniques enable testers and developers to assess the thoroughness of their testing efforts and identify potential areas of improvement. By applying these techniques, white box testing aims to increase the reliability, quality, and overall effectiveness of the software.

1. **Code Coverage:**

// Global variables to store order details

var loggedInCustomer = null;

var order = [];

// Function to check if the customer is registered

function isCustomerRegistered() {

  // Implement your logic to check if the customer is registered

  return loggedInCustomer !== null;

}

// Function to register the customer

function registerCustomer() {

  // Implement your logic to register the customer

  loggedInCustomer = { name: "John Doe", customerId: "123456" };

}

// Function to place an order

function placeOrder() {

  // Check if the customer is registered

  if (!isCustomerRegistered()) {

    var registerOption = confirm("You are not registered. Do you want to register now?");

    if (registerOption) {

      registerCustomer();

    } else {

      console.log("Order placement canceled.");

      return;

    }

  }

  // Ask the customer to select order type: pickup or delivery

  var orderType = prompt("Please select order type: (1) Pickup or (2) Delivery");

  var deliveryLocation = null;

  if (orderType === "2") {

    // If the order is for delivery, prompt for the delivery location

    deliveryLocation = prompt("Please enter the delivery location:");

  }

  // Display the menu for the specified date

  var menuDate = prompt("Please enter the menu date:");

  displayMenu(menuDate);

  // Prompt the user to select the number of units for each menu item

  for (var i = 0; i < menu.length; i++) {

    var menuItem = menu[i];

    var units = parseInt(prompt("Enter the number of units for " + menuItem.name + ":"));

    // Check if the number of units exceeds the available inventory

    if (units > menuItem.inventory) {

      units = menuItem.inventory;

      console.log("Maximum number of units available for " + menuItem.name + " is " + units);

    }

    // Add the ordered item to the order array

    order.push({ item: menuItem.name, price: menuItem.price, units: units });

  }

  // Display the order details and prompt for confirmation

  displayOrder();

  // Prompt for order confirmation

  var confirmation = confirm("Confirm the meal order?");

  if (confirmation) {

    console.log("Order confirmed!");

  } else {

    var choice = prompt("Choose an option:\n1. Edit the order\n2. Cancel the order");

    if (choice === "1") {

      // Edit the order

      editOrder();

    } else if (choice === "2") {

      // Cancel the order

      cancelOrder();

    } else {

      console.log("Invalid choice. Order canceled.");

    }

  }

}

// Function to display the menu for a specified date

function displayMenu(date) {

  // Implement your logic to retrieve and display the menu for the specified date

  var menu = getMenuForDate(date);

  console.log("Menu for " + date + ":");

  for (var i = 0; i < menu.length; i++) {

    console.log(menu[i].name + " - $" + menu[i].price);

  }

}

// Function to display the order details

function displayOrder() {

  console.log("Food Items Ordered:");

  for (var i = 0; i < order.length; i++) {

    console.log(order[i].item + " - $" + order[i].price + " - Units: " + order[i].units);

  }

  var paymentAmount = calculateTotal();

  console.log("Total Payment Amount: $" + paymentAmount);

}

// Function to calculate the total payment amount

function calculateTotal() {

  var total = 0;

  for (var i = 0; i < order.length; i++) {

    total += order[i].price \* order[i].units;

  }

  return total.toFixed(2);

}

// Function to edit the order

function editOrder() {

  // Implement your logic to edit the order

  console.log("Order edited successfully.");

}

// Function to cancel the order

function cancelOrder() {

  // Implement your logic to cancel the order

  console.log("Order canceled.");

}

# Method Coverage:

In the Order food Module code, the following methods are covered:

We will execute all these methods separately to check if these are working individually or not.

**isCustomerRegistered() -** Checks if the customer is registered.

**registerCustomer() -** Registers the customer.

**placeOrder() -** Initiates the process of placing an order.

**displayMenu(date) -** Displays the menu for a specified date.

**displayOrder() -** Displays the order details.

**calculateTotal() -** Calculates the total payment amount.

**editOrder() -** Allows the user to edit the order.

**cancelOrder() -** Cancels the order.

* 1. **Test for isCustomerRegistered():**

**Test case 1:**

When the customer is not registered, isCustomerRegistered () should return false.

**Test case 2:**

When the customer is registered, isCustomerRegistered () should return true.

* 1. **Test for registerCustomer ():**

**Test case:**

After calling registerCustomer (), isCustomerRegistered () should return true.

* 1. **Test for placeOrder ():**

**Test case 1:**

Simulated the scenario where the customer is not registered and chooses not to register. The order placement should be canceled.

**Test case 2:**

Simulated the scenario where the customer is not registered and chooses to register. After registration, the order placement should proceed as expected.

**Test case 3:**

Simulated the scenario where the customer is registered and goes through the entire order placement process, confirming the order at the end.

* 1. **Test for displayMenu(date):**

**Test case:**

Provided different valid and invalid dates and verify that the menu is displayed correctly for each date.

* 1. **Test for displayOrder ():**

**Test case:**

Created a sample order and verify that the order details are displayed correctly.

* 1. **Test for calculateTotal ():**

**Test case:**

Create a sample order with different item prices and quantities. Verify that the total payment amount is calculated correctly.

* 1. **Test for editOrder():**

**Test case:**

Create a sample order, call editOrder() to modify the order, and verify that the changes are reflected correctly.

* 1. **Test for cancelOrder():**

**Test case:**

Create a sample order, call cancelOrder(), and verify that the order is canceled and the necessary actions are taken.

# Branch Coverage:

Branch coverage refers to testing each possible branch or decision point in the code. Here are the branch coverage points in the provided code:

* 1. **Branch in isCustomerRegistered():**

**Branch 1:**

If the customer is not registered.

* **Test case 1:**

When the customer is not registered, isCustomerRegistered() should return false.

* **Test case 2:**

When the customer is registered, isCustomerRegistered() should return true.

* 1. **Branches in placeOrder():**

**Branch 1:**

If the customer is not registered and chooses not to register.

**Branch 2:**

If the customer is not registered and chooses to register.

**Branch 3**

If the order type is for delivery.

* **Test case 1:**

Simulate the scenario where the customer is not registered and chooses not to register. The order placement should be canceled.

* **Test case 2:**

Simulate the scenario where the customer is not registered and chooses to register. After registration, the order placement should proceed as expected.

* **Test case 3:**

Simulate the scenario where the customer is registered and chooses the order type as pickup. The order placement should proceed without any issues.

* **Test case 4:**

Simulate the scenario where the customer is registered and chooses the order type as delivery. Provide a valid delivery location. The order placement should proceed without any issues.

* 1. **Branch in displayOrder():**

**Branch 1:**

If the user confirms the meal order.

**Branch 2:**

If the user does not confirm the meal order.

* **Test case 1:**

Simulate the scenario where the user confirms the meal order. The confirmation message should be displayed.

* **Test case 2:**

Simulate the scenario where the user does not confirm the meal order. The user should be prompted to either edit or cancel the order.

# Condition Coverage:

Condition coverage refers to testing all possible outcomes of conditional expressions within the code. Here are the condition coverage points in the provided code:

* 1. **Condition in isCustomerRegistered():**
* Test case 1: When loggedInCustomer is null (customer not registered).
* Test case 2: When loggedInCustomer is not null (customer registered).
  1. **Conditions in placeOrder():**
* Test case 1: When the customer is not registered and chooses not to register (registerOption is false).
* Test case 2: When the customer is not registered and chooses to register (registerOption is true).
* Test case 3: When the order type is pickup (orderType is "1").
* Test case 4: When the order type is delivery (orderType is "2").
  1. **Condition in displayMenu(date):**
* Test case: When a valid date is provided.
  1. **Condition in placeOrder() within the loop:**
* Test case 1: When the units entered by the customer exceed the available inventory (units > menuItem.inventory).
* Test case 2: When the units entered by the customer do not exceed the available inventory (units <= menuItem.inventory).
  1. **Condition in the prompt within placeOrder():**
* Test case 1: When the customer confirms the meal order (confirmation is true).
* Test case 2: When the customer does not confirm the meal order (confirmation is false).
  1. **Conditions in the prompt within the else statement of placeOrder():**
* Test case 1: When the customer chooses to edit the order (choice is "1").
* Test case 2: When the customer chooses to cancel the order (choice is "2").
* Test case 3: When the customer selects an invalid choice.

-------------------------------------------------------------------------------------------------------------------