

Testing major task

-book store-



Siveen Said Sayed Ahmed 21P0149

Reem Alaa Ali Mohamed 2000207

Malak Mohamed Salem Elgeziry 21P0277

Hager Hesham Mohamed Elsawy 21P0297

Mariam Mahmoud Mohamed Abdou 21P0214

# Project Description

E-Commerce System is a robust online platform designed to facilitate seamless buying and selling of goods and services. With a focus on functionality, efficiency, and security, Our project offers the following key features:

1. **Product Catalogue**:
   * A comprehensive product catalog that showcases various items available for purchase.
   * Organized categories and subcategories for easy navigation.
   * Detailed product descriptions, images, and pricing information.
2. **User Accounts and Authentication**:
   * User registration and login functionality.
   * Secure authentication mechanisms (e.g., password hashing, session management).
   * User profiles with personal information and order history.
3. **Shopping Cart**:
   * Users can add products to their shopping cart while browsing the catalog.
   * Real-time updates on cart contents, quantities, and total price.
   * Ability to modify or remove items from the cart.
4. **Order Management**:
   * Users can place orders for selected items.
   * Each order includes a unique order ID, timestamp, and shipping details.
   * Stock management ensures that ordered items are deducted from available inventory.
5. **Payment Processing**:
   * Integration with secure payment gateways (e.g., credit card processing, PayPal).
   * Validation of credit card details (e.g., CVV, expiration date).
   * Confirmation emails sent to users after successful payment.
6. **Security Measures**:
   * Encryption of sensitive user data (passwords, credit card details).
   * Protection against common security threats (SQL injection, cross-site scripting).
   * Regular security audits and updates.
7. **User Experience**:
   * Responsive web design for seamless browsing on various devices.
   * Intuitive user interfaces with clear navigation and search functionality.
   * Error handling and informative messages for a smooth user experience.

# Unit Testing Description

## **EcomSystemTest Class:**

This test class is for testing an e-commerce system.

1. **Test Login**: First, we're testing if users can sign up and login. We're creating a user named Ahmed, and we're checking if we can log in with the username "Dino" and password "Danielo". We also check if someone with the wrong username and password can't log in.
2. **Test Signup**: Next, we're testing the signup process. We create a user named Siveen and check if it's successful. Then, we try to create the same user again to make sure it fails, as it should. Finally, we check if the user we created has the correct username and password.
3. **Test Get Current User**: Lastly, we're testing if we can get the current user. At the beginning, there shouldn't be any current user, so we check if it's null. Then, we sign up a user named Chery and make sure we can get the current user, and that it's the same as the one we signed up.

These tests help ensure that the e-commerce system is working as expected, allowing users to sign up, log in, and access their accounts correctly.

## **UserTest Class:**

This test class is for testing the functionality of the user class in an e-commerce system.

1. **Order Management Tests**:
   * **Test Add Order**: This test checks if an order can be added to the user's order list. It creates an order, adds it to the user, and then checks if the user's order list is not null and has a size of 1.
   * **Test Order Exists**: This test verifies if the user can retrieve an order by its ID after adding it. It creates an order, adds it to the user, and then checks if the retrieved order matches the added one.
   * **Test Getters**: This test validates various getters of the user object, including checking if the username, cart, and password are set correctly.
2. **Visa User Management Tests**:
   * **Test Add Visa**: This test ensures that the user can add multiple visas and that the list of added visas matches the expected list. It creates three visa objects, adds them to the user, and compares the expected and actual lists.
   * **Test Username**: This test simply checks if the username retrieved from the user object matches the one set during setup.

**.OrderTest Class:**

This test class is designed to validate the functionality of the order class in an e-commerce system

1. **Order Creation Tests**:
   * **Test Order ID Generation**: Checks if the order ID is generated properly. It creates two orders and ensures that their IDs are not null and follow the expected format.
   * **Test Total Price Calculation**: Verifies if the total price of the order is calculated correctly. Initially, it checks if the total price matches the expected value. Then, it adds another item to the cart and verifies that the total price changes accordingly.
   * **Test Order Status**: Ensures that the initial status of the order is "Order\_Placed".
   * **Test Time Difference in Seconds**: Validates the method that calculates the time difference in seconds. It schedules an end time 30 seconds from the current time and checks if the difference is correctly calculated.
   * **Test Schedule Status Update**: Tests if the status of the order changes to "Dispatch" after scheduling a status update.
2. **Stock Handling**:
   * **Test Cancel Order**: Checks if canceling an order restores the stock of items in the cart to their initial quantities.
   * **Test Remove from Stock**: Verifies if removing items from stock during order placement updates the stock quantities correctly.
3. **Test Getters and Setters**: Ensures that the getters and setters of the **Order** class work as expected. It validates if the cart, order ID, address, and phone number can be retrieved correctly, and if the status can be updated successfully.

## **CartTest Class:**

This CartTest class is focused on testing the functionalities of a shopping cart in an e-commerce system.

1. **Add To Cart Tests**:
   * **Valid and Invalid Amount**: these tests verify whether items can be added to the cart with valid and invalid quantities. The first test checks if items can be added successfully within the available stock limit, while the second test ensures that attempting to add more items than available stock results in an empty cart.
2. **Remove From Cart Test**:
   * This test checks if items can be successfully removed from the cart. It adds an item, removes it, and then verifies that the cart is empty and the total price is zero.
3. **Amount Increase and Decrease Tests**:
   * These tests validate the functionality to increase and decrease the quantity of items in the cart. They add items, increase or decrease their quantities, and verify if the changes are reflected correctly in the cart's contents and total price.
4. **Empty Cart Test**:
   * This test ensures that the cart can be emptied successfully. It adds items to the cart, empties it, and then verifies that the cart is indeed empty and the total price is zero.
5. **Tests for Getters**:
   * these tests validate the getters of the cart class, They add items to the cart, retrieve its contents and total price, and assert that they match the expected values.

# **VisaTest Class:**

This visa Test class is responsible for testing the functionality of the visa class, particularly focusing on validation checks for Visa card numbers and CVV (Card Verification Value).

1. **Test Valid Visa**:
   * This test ensures that the method correctly validates the expiration date of the Visa card. It checks if dates within the valid range return true and dates outside the valid range return false.
2. **Test Invalid Visa - Year, Month**:
   * This parameterized test verifies that method correctly handles invalid combinations of year and month. It uses a CSV source to provide different combinations of year and month, testing if the method returns false for all invalid cases.
3. **Visa CVV Check**:
   * This test validates the method. It checks if the method returns true for a correct CVV and false for an incorrect one.
4. **Visa Number Check**:
   * This test verifies that the Visa card number stored in the Visa object matches the expected value.

## **CatalogTest Class:**

This class is designed to test the functionality of the Catalog class.

1. **Add Item Test**:
   * This test validates the method. It adds two items to the catalog and checks if the size of the catalog increases accordingly and if the added items are present in the catalog.
2. **Tests for Getters**:
   * **Get Item Test**: Verifies if the method retrieves the correct item from the catalog based on its name.
   * **Get All Items Test**: Checks if the method returns a list containing all items in the catalog and if it includes the items that were added.
3. **Size Test**:
   * This test ensures that the method correctly returns the number of items in the catalog. It verifies if the size is 0 initially and if it increases after adding items.

## **ItemTest Class:**

This class is responsible for testing the functionality of the Item class, which likely represents individual items available for sale in an e-commerce system. Let's review the tests:

1. **Tests for Order and Unorder Methods**:
   * **Test for Order Method with Valid Amount**: Verifies if the ordered() method correctly decreases the stock of the item by the specified amount.
   * **Test for Unorder Method**: Checks if the unordered() method correctly increases the stock of the item by the specified amount.
2. **Tests for Getters**:
   * **Get Name Test**: Ensures that the getName() method returns the correct name of the item.
   * **Get Price Test**: Verifies that the getPrice() method returns the correct price of the item.
   * **Get Stock Test**: Checks if the getStock() method returns the correct stock quantity of the item.
   * **Get Image Test**: Validates that the getImage() method returns the correct image path of the item.

# Integration Testing

The integration testing process aims to combine various modules of the system to ensure they function together seamlessly. We are following a hybrid approach for integration testing, that combines both top-down and bottom-up strategies.

This approach ensures that each module seamlessly integrates into the larger system, thereby guaranteeing smooth functionality of the entire system.

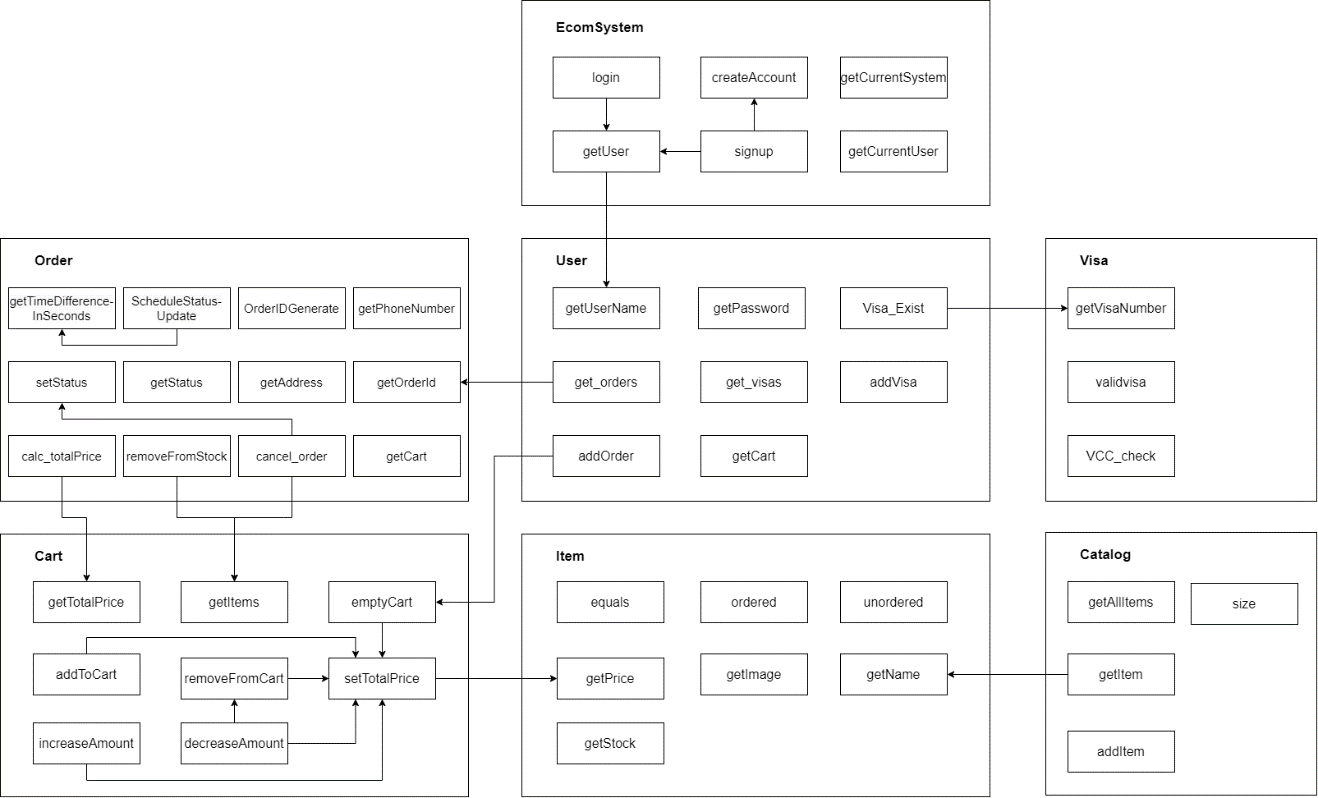
## Modules

This figure shows the structure of the system and how each class uses other classes. (e.g. E-commerce uses User)

A diagram of a product

Description automatically generated

The next figure shows how each module how each module depends on other modules. (e.g. getUser function in EcomSystem class depends on getUserName in User class)



### EcomSystem module

- Initializes user accounts and manages user-related functionalities.

- Dependencies: **User**.

### User module

- Represents a user in the system, handling user data and actions.

- Dependencies: **Order, Visa, Cart**.

### Visa module

- Handles payment information for users.

- Dependencies: None.

### Order module

- Manages orders placed by users on their carts.

- Dependencies: **Cart**.

### Cart module

- Stores items added by users for purchase.

- Dependencies: **Item**.

### Catalog module

- Stores and manages items.

- Dependencies: **Item**.

### Item module

- Represents individual items available for purchase.

- Dependencies: None.

## Implementation History

1. Implement **User** and **Order** Class
2. Implement User Controllers: SignupController and LoginController.
3. Implement Order Controllers: OrderListController, OrderRequestController, and OrderManagementController.
4. Implement **Visa** Class
5. Implement Visa Controllers: PaymentController and PaymentExistingCardController.
6. Implement **Item** Class
7. Implement **Catalog** Class
8. Implement ItemController
9. Implement CatalogController
10. Implement **Cart** Class
11. Implement CartController
12. Implement **EcomSystem** Class

## Top-down

* Most high-level module (**EcomSystem**) is implemented at the end.
* **User** is implemented before **Visa**, and **Cart**.
* **Order** uses **Cart**. **Order** is implemented first.

## Bottom-up

* Controllers are implemented after their classes.
* **Catalog** uses **Item**. **Item** is implemented first.
* **Item** is implemented before **Cart**.

## Integration Testing Approach

Integration testing ensures that the integrated modules work together correctly. conducting the following tests:

**Test the interfaces of each module with its controllers.**

Example: confirm that SignupController interacts with the signup() method.

**Test the functionality of the integrated modules, ensuring they perform as expected.**

Example: test if adding items to the cart reflects in the user's cart and updates the total price correctly.

**Test the interaction between dependent modules, ensuring seamless integration.**

Example: Test if the *Order* class correctly uses the *Cart* and *Item* classes to place orders.

## Integration Testing Result

The integration process successfully combined the e-commerce system modules. All modules of the system are integrated correctly and function smoothly together.

By employing a hybrid approach, we ensured that all modules of the system are integrated correctly and function smoothly together. That it allowed us to identify and address issues at both the component and system levels, ensuring a robust and reliable system.

Integration testing will further validate that the system functions reliably and meets the required specifications.

# White Box Testing:

1. **Control Flow Graph (CFG)**
2. **Cyclomatic Complexity**
3. **Path testing**

Control Flow Graph (CFG):

A Control Flow Graph (CFG) visually illustrates the organization and execution of a program's statements or instructions. It represents the program's control flow using nodes and directed edges.

Each node in the CFG corresponds to either a statement or a basic block—a sequence of statements with a singular entry and exit point.

The directed edges between nodes denote the potential flow of control from one statement to another.

**Cyclomatic Complexity:** Cyclomatic complexity is a software metric utilized to gauge the complexity of a program based on its control flow. It offers a quantitative assessment of the number of independent paths within a program's source code.

The higher the cyclomatic complexity, the greater the program's complexity, potentially necessitating increased testing and maintenance efforts.

The cyclomatic complexity of a program can be calculated using the following formula, known as McCabe's Cyclomatic Complexity (M):

M = E - N + 2P

Where:

E is the number of edges in the control flow graph.

N is the number of nodes in the control flow graph.

P is the number of connected components (exit points) in the graph.

**Guidelines:**

The cyclomatic complexity of 1 indicates a simple, linear program without any branching or decision points.

A cyclomatic complexity between 2 and 10 is considered reasonable and manageable.

A cyclomatic complexity above 10 suggests a higher level of complexity, indicating that the program may be harder to understand, test, and maintain.

**Path Testing:**

Path testing is a software testing method that methodically examines various paths or sequences of statements within a program. Its objective is to guarantee that all conceivable execution paths through the program's source code are scrutinized. This approach aims to enhance test coverage and uncover potential defects or errors.

In a Control Flow Graph (CFG), the number of paths corresponds to the cyclomatic complexity.

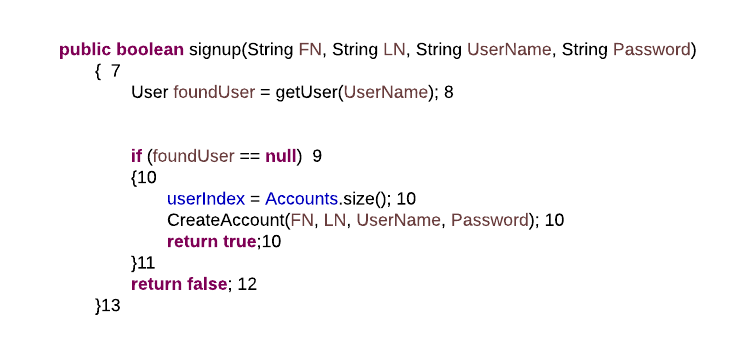
# Class EcomSystem:

A screenshot of a computer code

Description automatically generated

A diagram of a diagram

Description automatically generated



A diagram of a flowchart

Description automatically generated

A screenshot of a computer code

Description automatically generated

A diagram of a diagram

Description automatically generated

* + Cyclomatic **Complexity = 24-22+2\*3=8**
  + **Path 1 =1,2,4,6**
  + **Path 2 =1,2,3,5,6**
  + **Path 3 =7,8,9,10,11,13**
  + **Path 4 =7,8,9,12,13**
  + **Path 5 = 15,16,17,23**
  + **Path 6 =15,16,18,19,21,22,23**
  + **Path 7 = 15,16,18,19,20,23**
  + **Path 8 =15,16,18,19,21,20,23**

## User class

A diagram of a diagram

Description automatically generatedA close-up of a code

Description automatically generated

A screenshot of a computer code

Description automatically generated

A diagram of a diagram

Description automatically generated

* + Cyclomatic Complexity = 16-14+2\*2=6
  + Path 1 =1,2,3,4,7
  + Path 2 =1,2,3,5,2,6,7
  + Path 3 =1,2,6,7
  + Path 4 =8,9,10,11,14
  + Path 5 = 8,9,13,14
  + Path 6 =8,9,10,12,9,13,14

## Order Class

A diagram of a diagram

Description automatically generatedA screenshot of a computer program

Description automatically generated

A diagram of a diagram

Description automatically generatedA screenshot of a computer code

Description automatically generated

A diagram of a diagram

Description automatically generatedA computer code with text

Description automatically generated

A computer code with text

Description automatically generated with medium confidence

A diagram of a diagram

Description automatically generated

* + Cyclomatic Complexity = 33-31+2\*4=10
  + Path 1 =1,2,3,4,11
  + Path 2 =1,2,3,5,6,11
  + Path 3 =1,2,3,5,7,8,11
  + Path 4 =1,2,3,5,7,9,10,11
  + Path 5 = 12,13,14,15,18,19
  + Path 6 =12,13,14,16,17,18,19
  + Path 7 = 20,21,22,23,21,24
  + Path 8 = 20,21,24
  + Path 9 = 26,27,28,29,30,31
  + Path 10 = 26,27,28,29,30,28,31

## Cart Class

A computer code with text

Description automatically generated

A diagram of a diagram

Description automatically generated

A diagram of a number

Description automatically generatedA computer code with text

Description automatically generated

A computer code with text

Description automatically generated with medium confidence

A diagram of a flowchart

Description automatically generated

* + Cyclomatic Complexity = 21-21+2\*3=6
  + Path 1 =1,2,3,4,3,5,6
  + Path 2 =1,2,3,5,6
  + Path 3 =7,8,9,11,12,13
  + Path 4 =7,8,9,10,13
  + Path 5 = 14,15,16,17,20,21
  + Path 6 =14,15,16,18,19,20,21

## Visa Class

A diagram of a diagram

Description automatically generatedA computer code with text

Description automatically generated with medium confidence

A diagram of a diagram

Description automatically generatedA computer code with text

Description automatically generated with medium confidence

* + Cyclomatic Complexity = 14-13+2\*2=5
  + Path 1 =1,2,3,4,8
  + Path 2 =1,2,3,5,7,8
  + Path 3 =1,2,3,5,6,8
  + Path 4 =9,10,11,13
  + Path 5 = 9,10,12,13

# Coverage:

# statement coverage

## EcomSystem class:

    public User getUser(String UserName)

    {

    1-    for (int i = 0; i < Accounts.size(); i++) {

    2-        User user = Accounts.get(i);

    3-        if (user.getUserName().equalsIgnorecase(UserName)) {

    4-            userIndex = i;

    5-            return user;

            }

        }

    6-    userIndex = -1;

    7-    return null;

    }

**Test case 1:**

Accounts.size() 🡺 5

(user.getUserName().equalsIgnorecase(UserName)) 🡺 true

1->2->3->4->5

**Test case 2:**

Accounts.size() 🡺 0

1->6->7

    public boolean login(String UserName, String Password)

    {

    1-    if(Accounts.isEmpty())

    2-        return false;

    3-    User foundUser = getUser(UserName);

    4-    if (foundUser != null)

    5-        if(foundUser.getPassword().equals(Password))

    6-            return true;

    7-    return false;

    }

**Test case 1:**

Accounts.isEmpty() 🡺 true

foundUser = null

1->2

**Test case 2:**

Accounts.isEmpty() 🡺 false

foundUser != null

foundUser.getPassword().equals(Password)) 🡺 true

1->3->4->5->6

**Test case 3:**

Accounts.isEmpty() 🡺 false

foundUser = null

1->3->4->7

    public boolean signup(String FN, String LN, String UserName, String Password)

    {

    1-  User foundUser = getUser(UserName);

    2-    if (foundUser == null)

        {

    3-        userIndex = Accounts.size();

    4-        CreateAccount(FN, LN, UserName, Password);

    5-        return true;

        }

    6-    return false;

    }

**Test case 1:**

foundUser = null

1->2->3->4->5

**Test case 2:**

foundUser != null

1->2->6

    public User getCurrentUser(){

        1- if(userIndex >= 0 && userIndex < Accounts.size())

        2-    return Accounts.get(userIndex);

else

        3-     return null;

    }

**Test case 1:**

userIndex = 10

Account.size() 🡺 12

1->2

**Test case 2:**

userIndex = -10

Account.size() 🡺 12

1->3

## User class:

    public boolean Visa\_Exist(String num)

    {

        1- for (Visa v : Visas)

        2-    if(num.equals(v.getVisaNumber()))

        3-        return true;

        4- return false;

    }

**Test case 1:**

Visas != null

num.equals(v.getVisaNumber())) 🡺 true

1->2->3

**Test case 2:**

Visas = null

1->4

## Visa class:

    public boolean validvisa(int year, int month)

    {

        1- Calendar cal = Calendar.getInstance();

        2- int currentYear = cal.get(Calendar.YEAR)% 100;

        3- int currentMonth = cal.get(Calendar.MONTH) + 1;

        4- if (year < currentYear || year > currentYear + 5)

        5-    return false;

        6- else if (year == currentYear && month < currentMonth)

        7-    return false;

        8- return true;

    }

**Test case 1:**

year = 2020

month = 12

1->2->3->4->5

**Test case 2:**

year = 2024

month = 01

1->2->3->4->6->7

**Test case 3:**

year = 2025

month = 07

1->2->3->4->6->8

    public boolean CVV\_check(int cvv){

        1- if(this.cvv == cvv)

        2-    return true;

        3- return false;

    }

**Test case 1:**

this.cvv = 123

cvv = 456

1->3

**Test case 2:**

this.cvv = 555

cvv = 555

1->2

## Order class:

    public void scheduleStatusUpdate(LocalTime endTime) {

        1- int unitTime = 15;

        2- long TimeDiff = getTimeDifferenceInSeconds(endTime);

        3- if(TimeDiff < unitTime)

        4-    status = "Order\_Placed";

        5- else if(TimeDiff < 2\*unitTime)

        6-    status = "Packed";

        7- else if(TimeDiff < 3\*unitTime)

        8-     status = "Dispatch";

        else

        9-     status = "Delivered";

    }

**Test case 1:**

TimeDiff = 10

1->2->3->4

**Test case 2:**

TimeDiff = 20

1->2->3->5->6

**Test case 3:**

TimeDiff = 40

1->2->3->5->7->8

**Test case 4:**

TimeDiff = 50

1->2->3->5->7->9

    public double calc\_totalPrice()

    {

        1- double totPrice = cart.getTotalPrice();

        2- if(totPrice < 1000)

        3-     shippingPrice = 100;

        else

        4-     shippingPrice = 0;

        5- return totPrice + shippingPrice;

    }

**Test case 1:**

totPrice = 900

1->2->3->5

**Test case 2:**

totPrice = 1500

1->2->4->5

## Cart class:

public void addToCart(Item item, int amount) {  
 1- if(amount > item.getStock())  
 2- return;  
 3- int currentAmount = itemsAmounts.getOrDefault(item, 0);  
 4- itemsAmounts.put(item, currentAmount + amount);  
 5- setTotalPrice();  
}

**Test case 1:**

amount = 50

item.getStock()🡺20

1->2

**Test case 2:**

amount = 10

item.getStock()🡺20

1->3->4->5

public int increaseAmount(Item item) {   
 1- int currentAmount = itemsAmounts.get(item);  
 2- int newAmount = currentAmount + 1;  
 3- if (newAmount > item.getStock())  
 4- return currentAmount;  
 5- itemsAmounts.put(item, newAmount);  
 6- setTotalPrice();  
 7- return newAmount;  
}

**Test case 1:**

newAmount = 50

item.getStock()🡺20

1->2->3->4

**Test case 2:**

newAmount = 10

item.getStock()🡺20

1->2->3->5->6->7

public int decreaseAmount(Item item) {  
 1- int currentAmount = itemsAmounts.get(item);  
 2- int newAmount = currentAmount - 1;  
 3- if (newAmount == 0)  
 4- removeFromCart(item);  
 else  
 5- itemsAmounts.put(item, newAmount);  
  
 6- setTotalPrice();  
 7- return newAmount;  
}

**Test case 1:**

currentAmount = 1

1->2->3->4->6->7

**Test case 2:**

currentAmount = 10

1->2->3->5->6->7

## Catalog class

public static Item getItem(String bookName) {  
 1- for (Item item : *allitems*)  
 2- if (item.getName() == bookName)  
 3- return item;  
 4- return null;  
}

**Test case 1:**

item.getName()=> “Jane Eyre”

bookname= “Jane Eyre”

allitems not equal NULL

1->2->3

**Test case 2:**

item.getName()=> “War and Peace”

bookname= “Jane Eyre”

allitems not equal NULL

1->2->4

# Branch coverage

## EcomSystem class:

    public User getUser(String UserName)

    {

    1-    for (int i = 0; i < Accounts.size(); i++) {

    2-        User user = Accounts.get(i);

    3-        if (user.getUserName().equalsIgnorecase(UserName)) {

    4-            userIndex = i;

    5-            return user;

            }

        }

    6-    userIndex = -1;

    7-    return null;

    }

**Test case 1:**

Accounts.size() 🡺 5

(user.getUserName().equalsIgnorecase(UserName)) 🡺 true

1->2->3->4->5

**Test case 2:**

Accounts.size() 🡺 5

(user.getUserName().equalsIgnorecase(UserName)) 🡺 false

1->2->3->6->7

    public boolean login(String UserName, String Password)

    {

    1-    if(Accounts.isEmpty())

    2-        return false;

    3-    User foundUser = getUser(UserName);

    4-    if (foundUser != null)

    5-        if(foundUser.getPassword().equals(Password))

    6-            return true;

    7-    return false;

    }

**Test case 1:**

Accounts.isEmpty() 🡺 true

foundUser = null

1->2

**Test case 2:**

Accounts.isEmpty() 🡺 false

foundUser != null

foundUser.getPassword().equals(Password)) 🡺 true

1->3->4->5->6

**Test case 3:**

Accounts.isEmpty() 🡺 false

foundUser != null

foundUser.getPassword().equals(Password)) 🡺 false

1->3->4->5->7

**Test case 4:**

Accounts.isEmpty() 🡺 false

foundUser = null

1->3->4->7

    public boolean signup(String FN, String LN, String UserName, String Password)

    {

    1-  User foundUser = getUser(UserName);

    2-    if (foundUser == null)

        {

    3-        userIndex = Accounts.size();

    4-        CreateAccount(FN, LN, UserName, Password);

    5-        return true;

        }

    6-    return false;

    }

**Test case 1:**

foundUser = null

1->2->3->4->5

**Test case 2:**

foundUser != null

1->2->6

    public User getCurrentUser(){

        1- if(userIndex >= 0 && userIndex < Accounts.size())

        2-    return Accounts.get(userIndex);

else

        3-     return null;

    }

**Test case 1:**

userIndex = 10

Account.size() 🡺 12

1->2

**Test case 2:**

userIndex = -10

Account.size() 🡺 12

1->3

## User class:

    public boolean Visa\_Exist(String num)

    {

        1- for (Visa v : Visas)

        2-    if(num.equals(v.getVisaNumber()))

        3-        return true;

        4- return false;

    }

**Test case 1:**

Visas != null

num.equals(v.getVisaNumber())) 🡺 true

1->2->3

**Test case 2:**

Visas != null

num.equals(v.getVisaNumber())) 🡺 false

1->2->4

## Visa class:

    public boolean validvisa(int year, int month)

    {

        1- Calendar cal = Calendar.getInstance();

        2- int currentYear = cal.get(Calendar.YEAR)% 100;

        3- int currentMonth = cal.get(Calendar.MONTH) + 1;

        4- if (year < currentYear || year > currentYear + 5)

        5-    return false;

        6- else if (year == currentYear && month < currentMonth)

        7-    return false;

        8- return true;

    }

**Test case 1:**

year = 2020

month = 12

1->2->3->4->5

**Test case 2:**

year = 2024

month = 01

1->2->3->4->6->7

**Test case 3:**

year = 2025

month = 07

1->2->3->4->6->8

    public boolean CVV\_check(int cvv){

        1- if(this.cvv == cvv)

        2-    return true;

        3- return false;

    }

**Test case 1:**

this.cvv = 123

cvv = 456

1->3

**Test case 2:**

this.cvv = 555

cvv = 555

1->2

## Order class:

    public void scheduleStatusUpdate(LocalTime endTime) {

        1- int unitTime = 15;

        2- long TimeDiff = getTimeDifferenceInSeconds(endTime);

        3- if(TimeDiff < unitTime)

        4-    status = "Order\_Placed";

        5- else if(TimeDiff < 2\*unitTime)

        6-    status = "Packed";

        7- else if(TimeDiff < 3\*unitTime)

        8-     status = "Dispatch";

        else

        9-     status = "Delivered";

    }

**Test case 1:**

TimeDiff = 10

1->2->3->4

**Test case 2:**

TimeDiff = 20

1->2->3->5->6

**Test case 3:**

TimeDiff = 40

1->2->3->5->7->8

**Test case 4:**

TimeDiff = 50

1->2->3->5->7->9

    public double calc\_totalPrice()

    {

        1- double totPrice = cart.getTotalPrice();

        2- if(totPrice < 1000)

        3-     shippingPrice = 100;

        else

        4-     shippingPrice = 0;

        5- return totPrice + shippingPrice;

    }

**Test case 1:**

totPrice = 900

1->2->3->5

**Test case 2:**

totPrice = 1500

1->2->4->5

## Cart class:

public void addToCart(Item item, int amount) {  
 1- if(amount > item.getStock())  
 2- return;  
 3- int currentAmount = itemsAmounts.getOrDefault(item, 0);  
 4- itemsAmounts.put(item, currentAmount + amount);  
 5- setTotalPrice();  
}

**Test case 1:**

amount = 50

item.getStock()🡺20

1->2

**Test case 2:**

amount = 10

item.getStock()🡺20

1->3->4->5

public int increaseAmount(Item item) {   
 1- int currentAmount = itemsAmounts.get(item);  
 2- int newAmount = currentAmount + 1;  
 3- if (newAmount > item.getStock())  
 4- return currentAmount;  
 5- itemsAmounts.put(item, newAmount);  
 6- setTotalPrice();  
 7- return newAmount;  
}

**Test case 1:**

newAmount = 50

item.getStock()🡺20

1->2->3->4

**Test case 2:**

newAmount = 10

item.getStock()🡺20

1->2->3->5->6->7

public int decreaseAmount(Item item) {  
 1- int currentAmount = itemsAmounts.get(item);  
 2- int newAmount = currentAmount - 1;  
 3- if (newAmount == 0)  
 4- removeFromCart(item);  
 else  
 5- itemsAmounts.put(item, newAmount);  
  
 6- setTotalPrice();  
 7- return newAmount;  
}

**Test case 1:**

currentAmount = 1

1->2->3->4->6->7

**Test case 2:**

currentAmount = 10

1->2->3->5->6->7

## Catalog class

public static Item getItem(String bookName) {  
 1- for (Item item : *allitems*)  
 2- if (item.getName() == bookName)  
 3- return item;  
 4- return null;  
}

**Test case 1:**

item.getName()=> “Jane Eyre”

bookname= “Jane Eyre”

allitems not equal NULL

1->2->3

**Test case 2:**

item.getName()=> “War and Peace”

bookname= “Jane Eyre”

allitems not equal NULL

1->2->4

# Condition coverage

## EcomSystem class:

    public User getCurrentUser(){

        1- if(userIndex >= 0 && userIndex < Accounts.size())

        2-    return Accounts.get(userIndex);

else

        3-     return null;

    }

**Test case 1:**

userIndex = 10

Account.size() 🡺 12

1->2

**Test case 2:**

userIndex = -10

Account.size() 🡺 12

1->3

**Test case 3:**

userIndex = 20

Account.size() 🡺 12

1->3

## Visa class:

    public boolean validvisa(int year, int month)

    {

        1- Calendar cal = Calendar.getInstance();

        2- int currentYear = cal.get(Calendar.YEAR)% 100;

        3- int currentMonth = cal.get(Calendar.MONTH) + 1;

        4- if (year < currentYear || year > currentYear + 5)

        5-    return false;

        6- else if (year == currentYear && month < currentMonth)

        7-    return false;

        8- return true;

    }

**Test case 1:**

year = 2020

month = 12

1->2->3->4->5

**Test case 2:**

year = 2040

month = 01

1->2->3->4->5

**Test case 3:**

year = 2024

month = 02

1->2->3->4->6->7

**Test case 4:**

year = 2026

month = 08

1->2->3->4->6->8

# Multiple Condition coverage

## EcomSystem class:

    public User getCurrentUser(){

        1- if(userIndex >= 0 && userIndex < Accounts.size())

        2-    return Accounts.get(userIndex);

else

        3-     return null;

    }

**Test case 1:**

userIndex = 10

Account.size() 🡺 12

1->2

**Test case 2:**

userIndex = -10

Account.size() 🡺 12

1->3

**Test case 3:**

userIndex = 20

Account.size() 🡺 12

1->3

## Visa class:

    public boolean validvisa(int year, int month)

    {

        1- Calendar cal = Calendar.getInstance();

        2- int currentYear = cal.get(Calendar.YEAR)% 100;

        3- int currentMonth = cal.get(Calendar.MONTH) + 1;

        4- if (year < currentYear || year > currentYear + 5)

        5-    return false;

        6- else if (year == currentYear && month < currentMonth)

        7-    return false;

        8- return true;

    }

**Test case 1:**

year = 2020

month = 12

1->2->3->4->5

**Test case 2:**

year = 2040

month = 01

1->2->3->4->5

**Test case 3:**

year = 2024

month = 07

1->2->3->4->6->8

**Test case 4:**

year = 2024

month = 01

1->2->3->4->6->7

**Test case 5:**

year = 2026

month = 08

1->2->3->4->6->8

**Test case 6:**

year = 2026

month = 02

1->2->3->4->6->8

# MCDC

## EcomSystem class:

    public User getCurrentUser(){

        1- if(userIndex >= 0 && userIndex < Accounts.size())

        2-    return Accounts.get(userIndex);

else

        3-     return null;

    }

**Test case 1:**

userIndex = 10

Account.size() 🡺 12

1->2

**Test case 2:**

userIndex = -10

Account.size() 🡺 12

1->3

**Test case 3:**

userIndex = 20

Account.size() 🡺 12

1->3

## Visa class:

    public boolean validvisa(int year, int month)

    {

        1- Calendar cal = Calendar.getInstance();

        2- int currentYear = cal.get(Calendar.YEAR)% 100;

        3- int currentMonth = cal.get(Calendar.MONTH) + 1;

        4- if (year < currentYear || year > currentYear + 5)

        5-    return false;

        6- else if (year == currentYear && month < currentMonth)

        7-    return false;

        8- return true;

    }

**Test case 1:**

year = 2020

month = 12

1->2->3->4->5

**Test case 2:**

year = 2040

month = 01

1->2->3->4->5

**Test case 3:**

year = 2024

month = 07

1->2->3->4->6->8

**Test case 4:**

year = 2024

month = 01

1->2->3->4->6->7

**Test case 5:**

year = 2026

month = 02

1->2->3->4->6->8