Application Option #1: Shopping Cart

COP4331 02

Group 4:

George Martinez

Freddy Ingle

Sydney Tivoli

Bryan Cooke

George's Shopping Cart

Platform: Swing UI

Glossary of Domain Concepts:

1. **User**: An entity that can access and interact with the system. Users can be further categorized as customers or sellers.
2. **Authentication**: The process of verifying the identity of a user based on the provided credentials (username and password).
3. **Product**: An item available for sale on the platform. It has attributes such as name, price, description, and quantity.
4. **Product Catalog**: A collection or listing of available products.
5. **Shopping Cart**: A virtual basket where customers can add products they intend to purchase.
6. **Inventory**: A list or collection of products maintained by a seller, detailing quantities, prices, and other relevant information.
7. **Checkout**: The final step in the buying process where customers review their selected items and proceed to payment.
8. **Credit Card Information**: Data related to a customer's credit card which is used to process payments.
9. **Seller**: A user who lists products for sale on the platform and manages their inventory.
10. **Customer**: A user who browses the product catalog, adds items to the shopping cart, and makes purchases.
11. **Transaction**: A record of a completed sale, including details of the products purchased, total amount, payment method, and customer information.
12. **Invoice Price**: The amount that the seller pays to obtain the product.

13. **Selling Price**: The price at which the product is listed for sale to customers.
14. **Error Message**: A message displayed by the system to indicate a problem or inform the user about a particular situation, e.g., incorrect login credentials.
15. **Pop-up Window**: A secondary window that appears over the primary application window to display additional information or options.

Essential Use Cases:

1. User Authentication

- **Description:** Authenticate users (customers and sellers) based on their provided username and password.
- **Actors:** System, Customer, Seller
- **Trigger:** User attempts to log in.
- **Preconditions:** User accounts and authentication data exist.

Basic Flow:

1. User enters their username and password.
2. System validates the credentials.
3. User gets logged into system

Variant 1: Invalid Credentials Entered:

1.1. System displays an error message and suggests the user retry.

Variant 2: Security Question

2.1. After entering the primary credentials (username and password), the system prompts the user with a predefined security question.

2.2 User enters the answer to the security question**.**

2.3 The system validates the answer.

a. If the answer is correct, proceed to step 3 in basic flow

b. If the answer is incorrect, an error message is displayed, and the user has to try again

2. Browse Products

- **Description:** Allow customers to view available products in the catalog.
- **Actors:** Customer
- **Trigger:** Customer logs in or accesses the product catalog.
- **Preconditions:** Product catalog exists.

Basic Flow:

1. Customer views the product catalog.
2. The system displays a list of available products with names, prices, and quantities.

3. View Product Details

- **Description:** Provide detailed information about a selected product.
- **Actors:** Customer
- **Trigger:** Customer selects a product in the catalog.
- **Preconditions:** Product catalog exists, and the selected product is available.

Basic Flow:

1. Customer clicks on a product in the catalog.
2. A pop-up window displays detailed information about the product, including description, price, and availability.

4. Add Product to Cart

- **Description:** Allow customers to add products to their shopping cart.
- **Actors:** Customer
- **Trigger:** Customer selects a product and chooses to add it to the cart.
- **Preconditions:** Product catalog exists, and the selected product is available.

Basic Flow:

1. Customer clicks an "Add to Cart" button for a product.
2. The system adds the selected product to the shopping cart.
3. The shopping cart's total amount is updated.

5. Update Shopping Cart

- **Description:** Enable customers to modify the quantity of items in their shopping cart.
- **Actors:** Customer
- **Trigger:** Customer accesses their shopping cart.
- **Preconditions:** Customer has items in the shopping cart.

Basic Flow:

1. Customer adjusts the quantity of each item in the cart.
2. The system updates the shopping cart's total amount accordingly.

6. Checkout

- **Description:** Allow customers to review their shopping cart and complete the purchase.
- Actors: Customer
- **Trigger:** Customer initiates the checkout process.
- **Preconditions:** Customer has items in the shopping cart.

Basic Flow:

1. Customer proceeds to the checkout section.
2. They review the shopping cart contents.
3. Customer provides credit card information.
4. The system processes the payment and confirms the order.

Variant 1: Failed Payment:

1.1. System displays an error message about the failed payment.

1.2. Customer is prompted to re-enter credit card details or use a different payment method.

7. Seller Views Inventory

- **Description:** Enable sellers to view the current state of their product inventory.
- **Actors:** Seller
- **Trigger:** Seller logs in or accesses the inventory section.
- **Preconditions:** Product inventory exists.

Basic Flow:

1. Seller views the inventory.
2. The system displays product information, including ID, type, quantity, invoice price, and selling price.

8. Seller Updates Inventory

- **Description:** Allow sellers to add new products or update existing product information.
- **Actors:** Seller
- **Trigger:** Seller selects an option to add/update products.
- **Preconditions:** Seller is logged in.

Basic Flow:

1. Seller provides necessary product details (name, invoice price, selling price, quantity).
2. The system updates the inventory with the new information.

Variant 1: Error in Data Entry:

1.1. System identifies an error in the input data (e.g. negative product quantity or price).

1.2. Seller is prompted to correct the invalid data before submission.

9. Seller Adds New Product

Basic Flow:

1. Seller accesses inventory page
2. Seller chooses to add a new product
3. Seller provides product details: product name, invoice price, selling price, and available quantity.
4. The application adds the product to inventory

10. Seller Views Sales Page

Basic Flow:

1. Seller accesses sales page
2. Seller reads revenue
3. Seller reads total sales by month
4. Seller reads profit

Detailed Use Cases:

1. Detailed Use Case: User Authentication

- **Description:** Authenticate users (customers and sellers) based on their provided username and password.
- **Actors:** System, Customer, Seller
- **Trigger:** User attempts to log in.
- **Preconditions:** User accounts and authentication data exist.

Basic Flow:

1. Customer or seller enters their username and password.
2. System validates the credentials by checking against the stored user data.
3. For customers, the system grants access to customer functionalities.
4. For sellers, the system grants access to seller functionalities.

Variant 1: Invalid Credentials Entered:

1.1. System displays an error message and suggests the user retry.

Variant 2: Security Question

2.1. After entering the primary credentials (username and password), the system prompts the user with a predefined security question.

2.2 User enters the answer to the security question**.**

2.3 The system validates the answer.

a. If the answer is correct, the user is granted access.

b. If the answer is incorrect, an error message is displayed, and the user has to try again.

2. Detailed Use Case: Browse Products

- **Description:** Allow customers to view available products in the catalog.
- **Actor:** Customer
- **Trigger:** Customer logs in or accesses the product catalog.
- **Preconditions:** Product catalog exists.

Basic Flow:

1. Customer logs in or accesses the product catalog.
2. The system displays a list of available products in a user-friendly interface.
3. Each product in the catalog is listed with its name, price, and available quantity.
4. The customer can scroll through the catalog to view products.

3. Detailed Use Case: View Product Details

- **Description:** Provide detailed information about a selected product.
- **Actor:** Customer
- **Trigger:** Customer selects a product in the catalog.
- **Preconditions:** Product catalog exists, and the selected product is available.

Basic Flow:

1. Customer clicks on a product in the catalog.
2. The system opens a pop-up window displaying detailed information about the selected product.
3. The information includes the product's name, description, price, and availability.

4. Detailed Use Case: Add Product to Cart

- **Description:** Allow customers to add products to their shopping cart.
- **Actor:** Customer
- **Trigger:** Customer selects a product and chooses to add it to the cart.
- **Preconditions:** Product catalog exists, and the selected product is available.

Basic Flow:

1. Customer selects a product from the catalog.
2. The system displays product details and an "Add to Cart" button.
3. Customer clicks the "Add to Cart" button.
4. The system adds the selected product to the customer's shopping cart.
5. The shopping cart's total amount is updated to reflect the added product.

5. Detailed Use Case: Update Shopping Cart

- **Description:** Enable customers to modify the quantity of items in their shopping cart.
- **Actor:** Customer
- **Trigger:** Customer accesses their shopping cart.
- **Preconditions:** Customer has items in the shopping cart.

Basic Flow:

1. Customer goes to the shopping cart section.
2. The system displays the list of items in the cart along with their quantities.
3. Customer can change the quantity of each item in the cart.
4. The system automatically updates the shopping cart's total amount as quantities are adjusted.

6. Detailed Use Case: Checkout

- **Description:** Allow customers to review their shopping cart and complete the purchase.
- **Actor:** Customer
- **Trigger:** Customer initiates the checkout process.
- **Preconditions:** Customer has items in the shopping cart.

Basic Flow:

1. Customer proceeds to the checkout section.
2. The system displays a summary of the shopping cart contents.
3. Customer reviews the items, quantities, and total amount.
4. Customer provides credit card information (assuming successful payment).
5. The system processes the payment and confirms the order.
6. Customer receives a confirmation of the order, and the shopping cart is cleared.

Variant 1: Failed Payment:

1.1. System displays an error message about the failed payment.

1.2. Customer is prompted to re-enter credit card details or use a different payment method.

7. Detailed Use Case: Seller Views Inventory

- **Description:** Enable sellers to view the current state of their product inventory.
- **Actors:** Seller
- **Trigger:** Seller logs in or accesses the inventory section.
- **Preconditions:** Product inventory exists

Basic Flow:

1. Seller navigates to inventory page
2. Application displays current state of the product inventory:
    - Product name
    - Product ID
    - Type
    - Quantity
    - Invoice price
    - Selling price

8. Detailed Use Case: Seller Updates Inventory

- **Description:** Allow sellers to add new products or update existing product information
- **Actors:** Seller
- **Trigger:** Seller selects an option to add/update products.

Basic Flow:

1. Seller accesses the inventory page. Application displays the current inventory with existing product details.
2. Seller selects an existing product to update or chooses to add a new product.
3. If updating, the application displays a pre-filled form with the current product details.
4. If adding, the application displays a blank product form.
5. Seller modifies product details or fills in the new product details as needed:
    - Product Name
    - Invoice Price
    - Selling Price
    - Available Quantity

6. Seller confirms and submits the updated or new product details.
7. Application validates the input data.
8. Seller can continue updating other products or return to the main dashboard.

Variant 1: Error in Data Entry:

1.1. System identifies an error in the input data (e.g. negative product quantity or price).

1.2. Seller is prompted to correct the invalid data before submission.

9. Detailed Use Case: Seller Adds New Product

- **Description:** Allow sellers to introduce new products to the inventory.
- **Actors:** Seller
- **Trigger:** Seller decides to add a new product to the inventory.

Basic Flow:

1. Seller accesses the inventory page.
2. Application displays the current inventory with existing product details.
3. Seller clicks on the "Add New Product" button.
4. Application displays a blank product form.
5. Seller enters the required product details:
    - Product Name
    - Invoice Price
    - Selling Price
    - Available Quantity
6. Seller confirms and submits the new product details.
7. Application validates the input data.
8. If data is valid, the new product is added to the inventory, and a confirmation message is shown.
9. If data is invalid, an error message is displayed prompting the seller to correct the information.
10. Seller can continue adding other products or return to the main dashboard.

10. Detailed Use Case: Seller Views Sales Page

Description: Seller views the information about total sales, revenue, and profit

Actor: Seller

Trigger: Seller needs to access information about sales

Preconditions:

- Seller has an active account on the platform.
- Seller has sold items which have recorded sales data.

Basic Flow:

1. **Seller accesses sales page:**
    - 1.2. Seller navigates to the sales dashboard.
    - 1.3. Seller clicks on the “Sales Page” link or button.
2. **Seller reads revenue:**
    - 2.1. Seller views the “Total Revenue” section displaying the overall revenue generated from all sales.
    - 2.2. Seller might see a breakdown of revenue from different products or categories if available.
3. **Seller reads total sales by month:**
    - 3.1. Seller navigates to the “Monthly Sales” section.
    - 3.2. Graph or table presents sales figures for each month.
    - 3.3. Seller may have an option to view sales of specific months by selecting the desired month from a dropdown or calendar picker.
4. **Seller reads profit:**
    - 4.1. Seller moves to the “Profit” section.
    - 4.2. System displays total profit calculated by subtracting expenses from revenue.
    - 4.3. There might be a detailed breakdown showing sources of expenses (production costs, shipping, platform fees, etc.).

“Walkthrough” Use Cases

1. User Authentication

- **Description:** Authenticate users (customers and sellers) based on their provided username and password.
- **Actors:** System, Customer, Seller, AuthenticationController, LoginView
- **Trigger:** User attempts to log in.
- **Preconditions:** User accounts and authentication data exist.

Flow:

1. **View:** Customer or seller enters their username and password in the LoginView.
2. **Controller:** LoginView sends the credentials to AuthenticationController.
3. **Controller:** AuthenticationController validates the credentials by checking against the stored user data in the UserModel.
4. **Model:** UserModel confirms or denies the validation.
5. **View:** If credentials are valid, AuthenticationController directs the user to the appropriate dashboard (CustomerView or SellerView).
6. **View:** If credentials are invalid, LoginView displays an error message and suggests the user retry.

Variant 1: Invalid Credentials Entered:

1.1. LoginView shows an error message and offers the user a chance to retry.

Variant 2: Security Question:

2.1. After entering the primary credentials, AuthenticationController prompts the user with a security question through the SecurityQuestionView.

2.2. User enters the answer.

2.3. AuthenticationController validates the answer against UserModel.

2.4. If correct, proceed to step 5 in the basic flow. If incorrect, display an error message via SecurityQuestionView.

2. Browse Products

- **Description:** Allow customers to view available products in the catalog.
- **Actors:** Customer, ProductCatalogView, ProductCatalogController, ProductModel
- **Trigger:** Customer logs in or accesses the product catalog.
- **Preconditions:** Product catalog exists.

Flow:

1. **View:** Customer accesses the ProductCatalogView.
2. **Controller:** ProductCatalogView requests the list of products from ProductCatalogController.
3. **Model:** ProductCatalogController retrieves product information from ProductModel.
4. **View:** ProductCatalogView displays the products with names, prices, and quantities.

Variant 1: Empty Catalog:

1.1. If the product catalog is empty, ProductCatalogView displays a message indicating that no products are available.

Variant 2: Network Error:

2.1. If there's a network error or inability to retrieve data, ProductCatalogView displays an error message and suggests the customer try again later.

3. **View Product Details**

- **Description:** Provide detailed information about a selected product.
- **Actors:** Customer, ProductDetailView, ProductController, ProductModel
- **Trigger:** Customer selects a product in the catalog.
- **Preconditions:** Product catalog exists, and the selected product is available.

Flow:

1. **View:** Customer selects a product in ProductCatalogView.
2. **Controller:** ProductCatalogView sends the selected product ID to ProductController.
3. **Model:** ProductController fetches detailed product information from ProductModel.
4. **View:** ProductDetailView displays detailed information about the product.

Variant 1: Product Unavailable:

1.1. If the selected product is no longer available, ProductDetailView displays a message indicating the product is out of stock or unavailable.

Variant 2: Incomplete Product Information:

2.1. If certain product details are missing, ProductDetailView shows available information and indicates missing data.

4. **Add Product to Cart**

- **Description:** Enable customers to add products to their shopping cart.
- **Actors:** Customer, ProductCatalogView, ShoppingCartController, ShoppingCartModel
- **Trigger:** Customer selects a product and chooses to add it to the cart.
- **Preconditions:** Product catalog exists, and the selected product is available.

Flow:

1. **View:** Customer selects a product from the ProductCatalogView.
2. **Controller:** ProductCatalogView sends the selected product information to ShoppingCartController.
3. **Controller:** ShoppingCartController checks the availability of the product in ShoppingCartModel.
4. **Model:** ShoppingCartModel updates the cart with the new product.
5. **View:** ShoppingCartController updates the ShoppingCartView to reflect the new cart contents.

5. **Update Shopping Cart**

- **Description:** Enable customers to modify the quantity of items in their shopping cart.
- **Actors:** Customer, ShoppingCartView, ShoppingCartController, ShoppingCartModel
- **Trigger:** Customer accesses their shopping cart.
- **Preconditions:** Customer has items in the shopping cart.

Flow:

1. **View:** Customer accesses ShoppingCartView.
2. **View:** ShoppingCartView displays the items in the cart with editable quantities.
3. **View:** Customer adjusts the quantities and submits changes.
4. **Controller:** ShoppingCartController updates the quantities in ShoppingCartModel.
5. **View:** ShoppingCartView updates to show the new total amount.

Variant 1: Product Quantity Exceeds Availability:

1.1. If a customer tries to increase the quantity beyond what's available, ShoppingCartView alerts the customer and prevents the update.

Variant 2: Cart Empty:

2.1. If the shopping cart is empty, ShoppingCartView displays a message indicating the cart is empty and suggests browsing products.

6. Checkout

- **Description:** Enable customers to review their shopping cart and complete the purchase.
- **Actors:** Customer, CheckoutView, PaymentController, ShoppingCartModel, PaymentModel
- **Trigger:** Customer initiates the checkout process.
- **Preconditions:** Customer has items in the shopping cart.

Flow:

1. **View:** Customer navigates to the CheckoutView.
2. **View:** CheckoutView displays a summary of the shopping cart contents using data from ShoppingCartModel.
3. **View:** Customer reviews items and enters credit card information in CheckoutView.
4. **Controller:** PaymentController processes the payment information with PaymentModel.
5. **Model:** PaymentModel confirms or denies the transaction.
6. **View:** If the payment is successful, CheckoutView displays a confirmation message, and ShoppingCartModel clears the cart.

Variant 1: Failed Payment:

1.1. CheckoutView displays an error message about the failed payment.

1.2. Customer is prompted to re-enter credit card details or use a different payment method.

7. **Seller Views Inventory**

- **Description:** Enable sellers to view the current state of their product inventory.
- **Actors:** Seller, InventoryView, InventoryController, InventoryModel
- **Trigger:** Seller logs in or accesses the inventory section.
- **Preconditions:** Product inventory exists.

Flow:

1. **View:** Seller accesses InventoryView.
2. **Controller:** InventoryView requests inventory data from InventoryController.
3. **Model:** InventoryController retrieves data from InventoryModel.
4. **View:** InventoryView displays product information (ID, type, quantity, prices).

Variant 1: Empty Inventory:

1.1. If the inventory is empty, InventoryView displays a message indicating no products are currently available in the inventory.

Variant 2: Data Retrieval Error:

2.1. If there's an error in fetching inventory data, InventoryView shows an error message.

8. Seller Updates Inventory

- **Description:** Allow sellers to add new products or update existing product information.
- **Actors:** Seller, InventoryManagementView, InventoryController, InventoryModel
- **Trigger:** Seller selects an option to add/update products.
- **Preconditions:** Seller is logged in.

Flow:

1. **View:** Seller accesses the InventoryManagementView.
2. **Controller:** Seller inputs new or updated product details in InventoryManagementView.
3. **Model:** InventoryController validates and updates the information in InventoryModel.
4. **View:** InventoryManagementView displays updated inventory or prompts for corrections.

Variant 1: Duplicate Product Entry:

1.1 If the seller tries to add a product that already exists, InventoryManagementView prompts the seller to update the existing product instead.

Variant 2: Invalid Product Information:

2.1. If entered product details are invalid (e.g., negative price), InventoryManagementView prompts for correction.

9. Seller Adds New Product

- **Description:** Allow sellers to introduce new products to the inventory.
- **Actors:** Seller, NewProductView, ProductController, InventoryModel
- **Trigger:** Seller decides to add a new product to the inventory.

Flow:

1. **View:** Seller accesses NewProductView.
2. **View:** Seller fills in new product details in NewProductView.
3. **Controller:** ProductController validates and adds the new product to InventoryModel.
4. **View:** NewProductView displays confirmation or prompts for correction.

Variant 1: Incomplete Product Details:

1.1. If the seller submits the form with incomplete details, NewProductView prompts for all necessary information.

Variant 2: Duplicate Product:

2.1. If the new product closely matches an existing product, NewProductView suggests updating the existing product instead.

10. Seller Views Sales Page

- **Description:** Seller views information about total sales, revenue, and profit.
- **Actors:** Seller, SalesView, SalesController, SalesModel
- **Trigger:** Seller needs to access information about sales.
- **Preconditions:** Seller has an active account and recorded sales data.

Flow:

1. **View:** Seller accesses SalesView.
2. **Controller:** SalesView requests sales data from SalesController.
3. **Model:** SalesController retrieves data from SalesModel.
4. **View:** SalesView displays total revenue, sales by month, and profit.

Variant 1: No Sales Data:

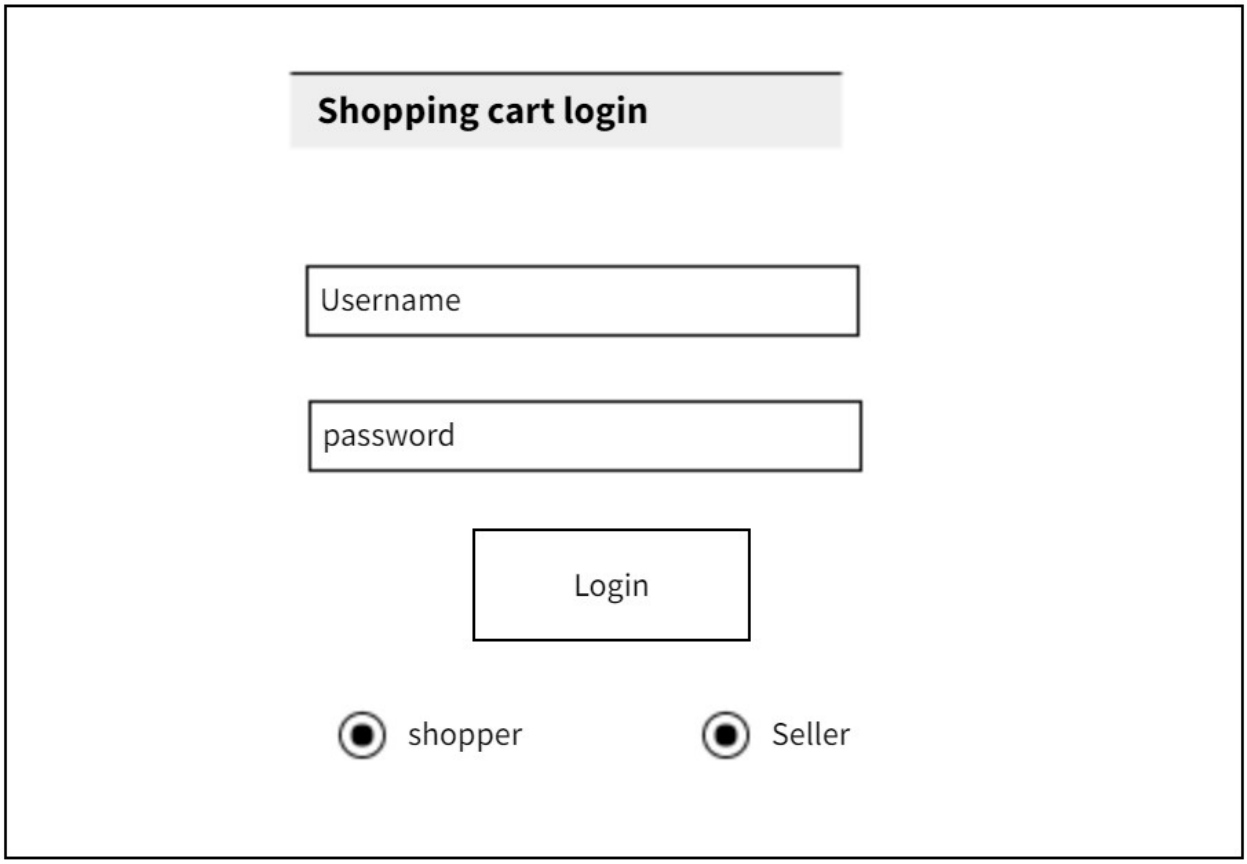
1.1. If there are no sales records, SalesView displays a message indicating no sales data is available.

Variant 2: Data Access Error:

2.1. If there's an error accessing sales data, SalesView displays an error message and suggests trying again later.

Use Cases with GUI

1.User Authentication



- **Description:** Authenticate users (customers and sellers) based on their provided username and password.
- **Actors:** System, Customer, Seller, AuthenticationController, LoginView
- **Trigger:** User attempts to log in.
- **Preconditions:** User accounts and authentication data exist.

Flow:

1. The user navigates to the shopping cart system's login page.
2. The login interface is displayed, showing fields for "Username" and "Password," a "Login" button, and options to select either "customer" or "seller."
3. The user enters their username into the "Username" field.
4. The user enters their password into the "Password" field.

5. The user selects their role by clicking the radio button next to "Customer" or "Seller."
6. The user clicks the "Login" button to proceed.
7. The system validates the entered credentials against the stored user database.
8. If the credentials are correct and the user selected "customer," the system redirects the user to the customer homepage where they can browse and purchase products.
9. If the credentials are correct and the user selected "seller," the system redirects the user to the seller dashboard where they can manage their product listings and view sales statistics.

Variant 1: Invalid Credentials Entered:

2.1. LoginView shows an error message and offers the user a chance to retry.

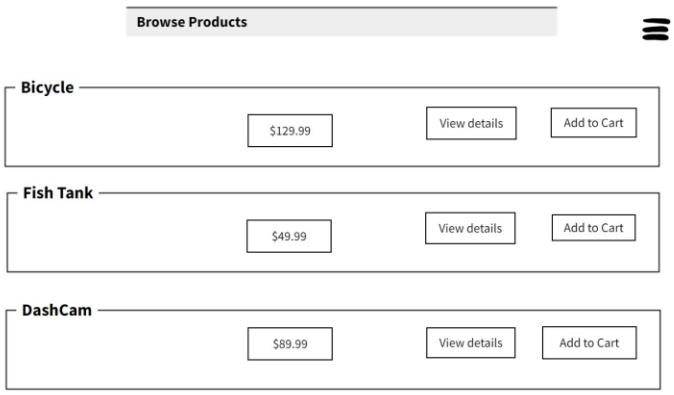
Variant 2: Security Question:

2.5. After entering the primary credentials, AuthenticationController prompts the user with a security question through the SecurityQuestionView.

2.6. User enters the answer.

2.7. AuthenticationController validates the answer against UserModel.

2.Browse Products

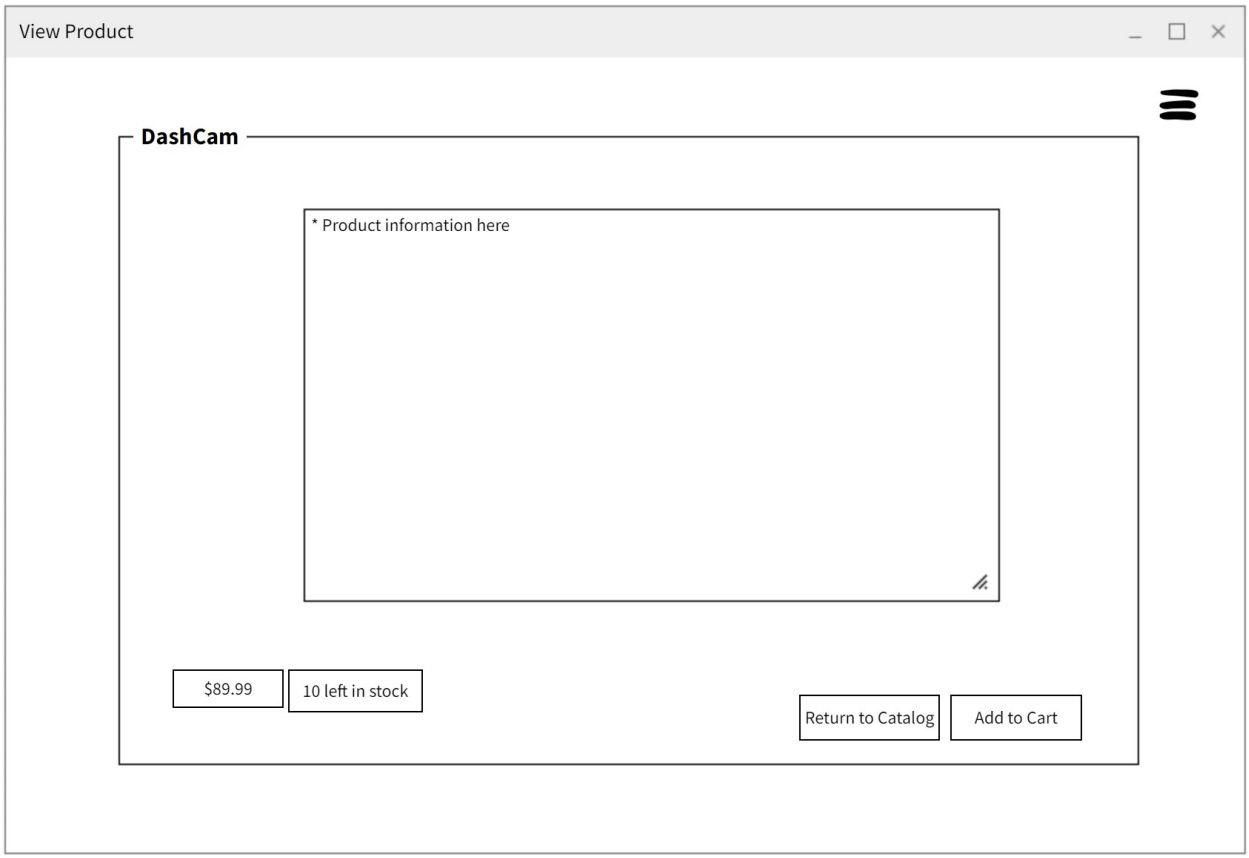


- **Description:** Allow customers to view available products in the catalog.
- **Actors:** Customer, ProductCatalogView, ProductCatalogController, ProductModel
- **Trigger:** Customer logs in or accesses the product catalog.
- **Preconditions:** Product catalog exists.

Flow:

1. The shopper is on the "Browse Products" page after logging in.
2. The page displays a list of products, each with a title, price, "View details" button, and "Add to Cart" button.
3. The shopper browses through the list of products.
4. The shopper decides to purchase a "Bicycle" and reviews the price listed as $129.99.

3. View Product Details



- **Description:** Provide detailed information about a selected product.
- **Actors:** Customer, ProductDetailView, ProductController, ProductModel

- **Trigger:** Customer selects a product in the catalog.
- **Preconditions:** Product catalog exists, and the selected product is available.

Flow:

1. The shopper is presented with the "View Product" page for a DashCam.
2. The page displays the product name "DashCam" at the top.
3. Below the image placeholder, there is a text placeholder "*Product information here" where detailed information about the DashCam should be displayed.
4. The price of the DashCam ($89.99) is displayed.
5. An indicator shows that there are "10 left in stock," informing the shopper of the available quantity.
6. The shopper can choose to return to the product catalog by clicking the "Return to Catalog" button.
7. If the shopper decides to purchase the DashCam, they click the "Add to Cart" button.
8. Upon clicking "Add to Cart," the product is added to the shopper's cart, and a confirmation message is displayed.
9. The shopper can either proceed to checkout or return to the catalog to continue shopping.

Variant 1: Product Unavailable:

1.1. If the selected product is no longer available, ProductDetailView displays a message indicating the product is out of stock or unavailable.

Variant 2: Incomplete Product Information:

2.1. If certain product details are missing, ProductDetailView shows available information and indicates missing data.

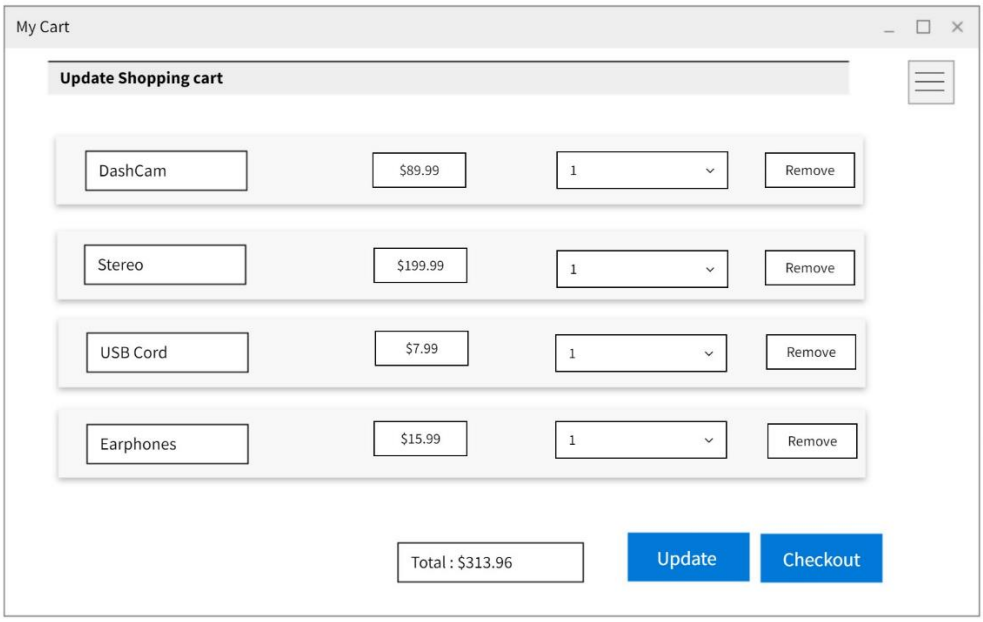
4. Add Product to Cart

- **Description:** Enable customers to add products to their shopping cart.
- **Actors:** Customer, ProductCatalogView, ShoppingCartController, ShoppingCartModel
- **Trigger:** Customer selects a product and chooses to add it to the cart.
- **Preconditions:** Product catalog exists, and the selected product is available.

Flow:

1. The shopper is on the "View Product" page for the DashCam, which shows the product name, price, available stock, and a detailed product description.
2. The shopper reviews the information provided about the DashCam, including the price ($89.99) and the note indicating there are "10 left in stock."
3. The shopper decides to purchase the DashCam.
4. The shopper clicks the "Add to Cart" button located below the stock information.
5. The system checks the availability of the product in real-time to ensure it can be added to the cart.
6. Upon successful verification, the DashCam is added to the shopper's shopping cart.
7. A confirmation message or icon indicates the addition of the product to the cart.
8. An updated cart count is displayed, typically on the top right corner of the page or beside the shopping cart icon, indicating the number of items in the cart.
9. The shopper is given the option to "Return to Catalog" to continue shopping or proceed to checkout.

5. Update Shopping Cart



- **Description:** Enable customers to modify the quantity of items in their shopping cart.
- **Actors:** Customer, ShoppingCartView, ShoppingCartController, ShoppingCartModel
- **Trigger:** Customer accesses their shopping cart.
- **Preconditions:** Customer has items in the shopping cart.

Flow:

1. The shopper is presented with the "My Cart" page, displaying a list of currently selected products: DashCam, Stereo, USB Cord, and Earphones.

2. Each product listing shows the item name, price, a dropdown for quantity selection, and a "Remove" button.

3. The total price of all items in the cart is displayed at the bottom of the list.

4. To change the quantity of an item, the shopper clicks on the quantity dropdown next to the item and selects the desired number.

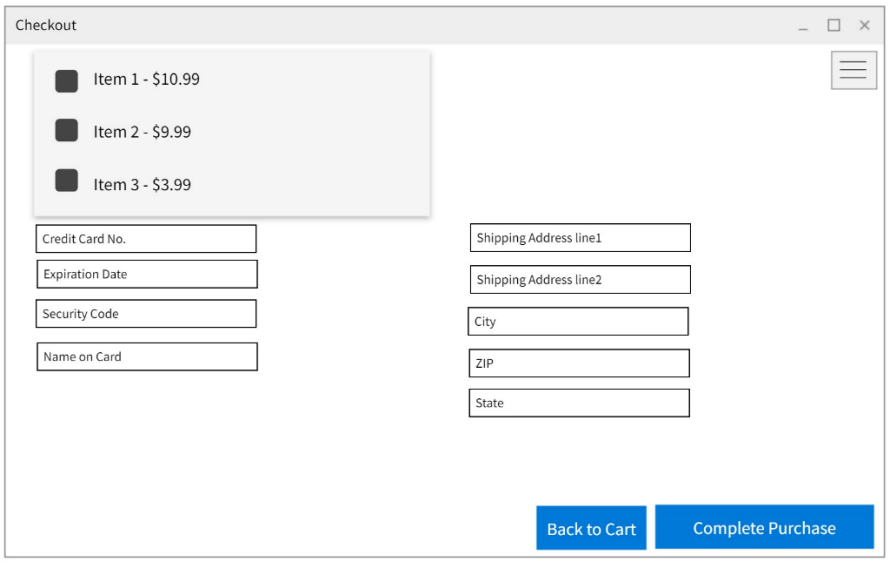
5. If the shopper decides to remove an item entirely, they click the "Remove" button next to that item.

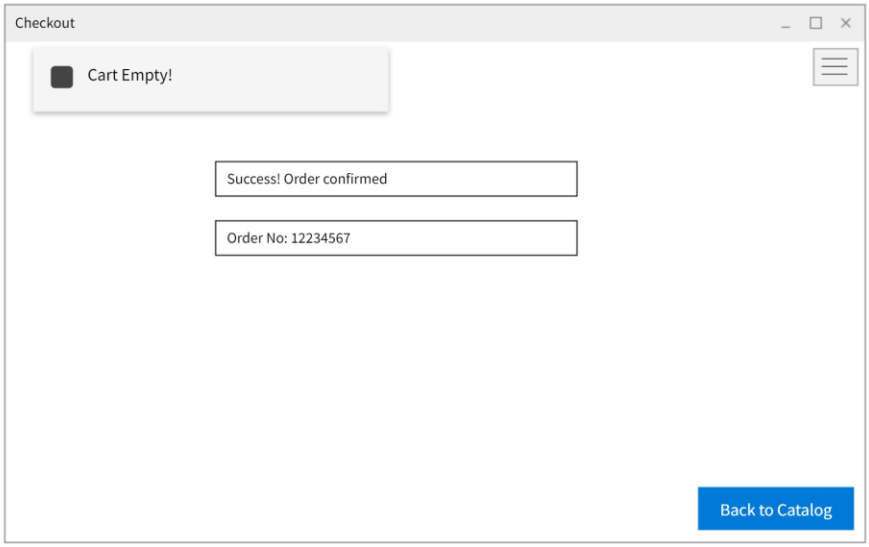
6. After making changes to quantities or removing items, the shopper clicks the "Update" button to refresh the cart and the total price.

7. The system updates the shopping cart to reflect the changes made by the shopper, including the new total price.

8. The shopper reviews the updated cart to ensure it reflects the desired products and quantities.

6. Checkout





- **Description:** Enable customers to review their shopping cart and complete the purchase.
- **Actors:** Customer, CheckoutView, PaymentController, ShoppingCartModel, PaymentModel

- **Trigger:** Customer initiates the checkout process.
- **Preconditions:** Customer has items in the shopping cart.

Flow:

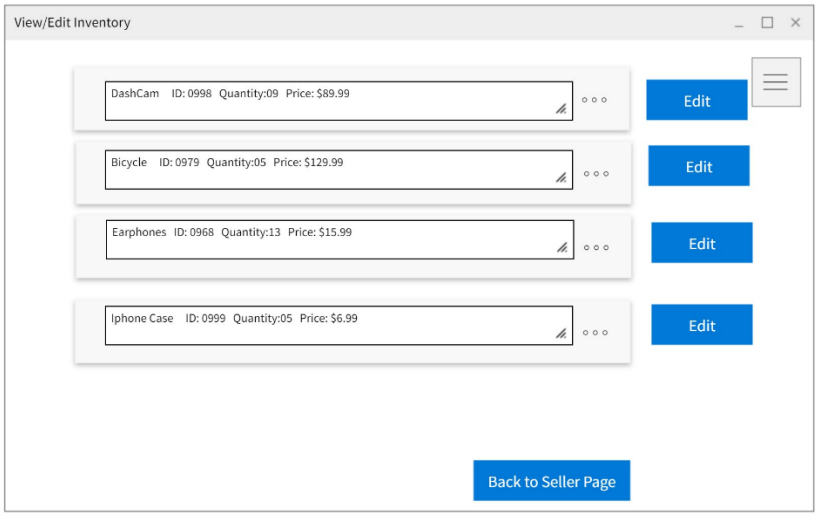
1. The shopper is directed to the "Checkout" page, which lists the items they have selected for purchase along with their individual prices.
2. The checkout interface is divided into two sections: one for payment information and the other for shipping details.
3. The shopper begins by entering their payment information, including the "Credit Card No.", "Expiration Date", "Security Code", and the "Name on Card" into the appropriate fields.
4. Next, the shopper fills out the shipping details, including "Shipping Address line1", "Shipping Address line2" (if necessary), "City", "ZIP", and "State".
5. The shopper has the option to review their order by clicking "Back to Cart", which will take them back to the cart page to review or modify their order.
6. Once the payment and shipping details are filled in correctly, the shopper clicks the "Complete Purchase" button.
7. The system processes the payment, verifies the shipping details, and if everything is in order, confirms the purchase.
8. A confirmation message is displayed to the shopper with an order number.
9. The shopper is then either given the option to return to Catalog or can exit the app.

Variant 1: Failed Payment:

1.1. CheckoutView displays an error message about the failed payment.

1.2. Customer is prompted to re-enter credit card details or use a different payment method.

7. Seller Views Inventory



- **Description:** Enable sellers to view the current state of their product inventory.
- **Actors:** Seller, InventoryView, InventoryController, InventoryModel
- **Trigger:** Seller logs in or accesses the inventory section.
- **Preconditions:** Product inventory exists.

Flow:

1. The seller navigates to the "View/Edit Inventory" section of the seller dashboard.
2. A list of products is displayed with product ID, quantity in stock, and current price.
3. The seller reviews the information for accuracy and determines if any updates are needed based on stock levels or price changes.
4. If an update is needed, the seller clicks the "Edit" button next to the respective item.
5. After reviewing or editing the inventory, the seller can return to the main seller dashboard by clicking "Back to Seller Page".

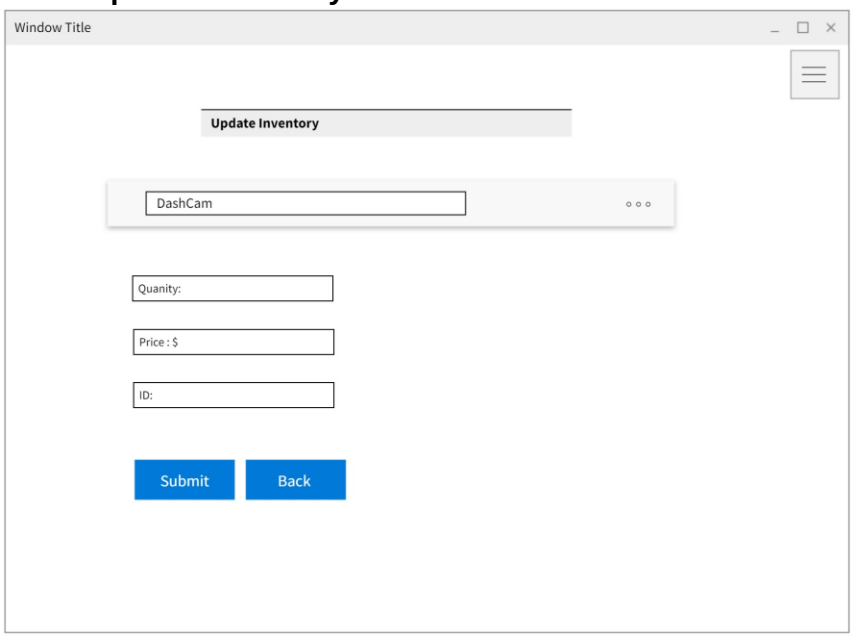
Variant 1: Empty Inventory:

1.1. If the inventory is empty, InventoryView displays a message indicating no products are currently available in the inventory.

Variant 2: Data Retrieval Error:

2.1. If there's an error in fetching inventory data, InventoryView shows an error message.

8. Seller Updates Inventory



- **Description:** Allow sellers to add new products or update existing product information.
- **Actors:** Seller, InventoryManagementView, InventoryController, InventoryModel
- **Trigger:** Seller selects an option to add/update products.
- **Preconditions:** Seller is logged in.

Flow:

1. The seller accesses the "View/Edit Inventory" page, where they can edit the details of products they have listed.
2. The seller is presented with editable fields for a specific product, including the product name, quantity, price, and ID.

3. The seller enters the new information into the appropriate fields:
    - “Product Name”
    - "Quantity"
    - "Price"
    - "Product ID"
4. After reviewing the changes and ensuring all information is accurate, the seller clicks the "Save Changes" button to save the updates.
5. The system processes the changes and updates the product details in the inventory database.
6. A confirmation message appears to notify the seller that the inventory has been successfully updated.
7. The updated product details are now reflected in the seller's inventory.
8. If the seller decides not to make changes at any point, they can click the "Cancel" button to return to the previous screen without saving any changes.

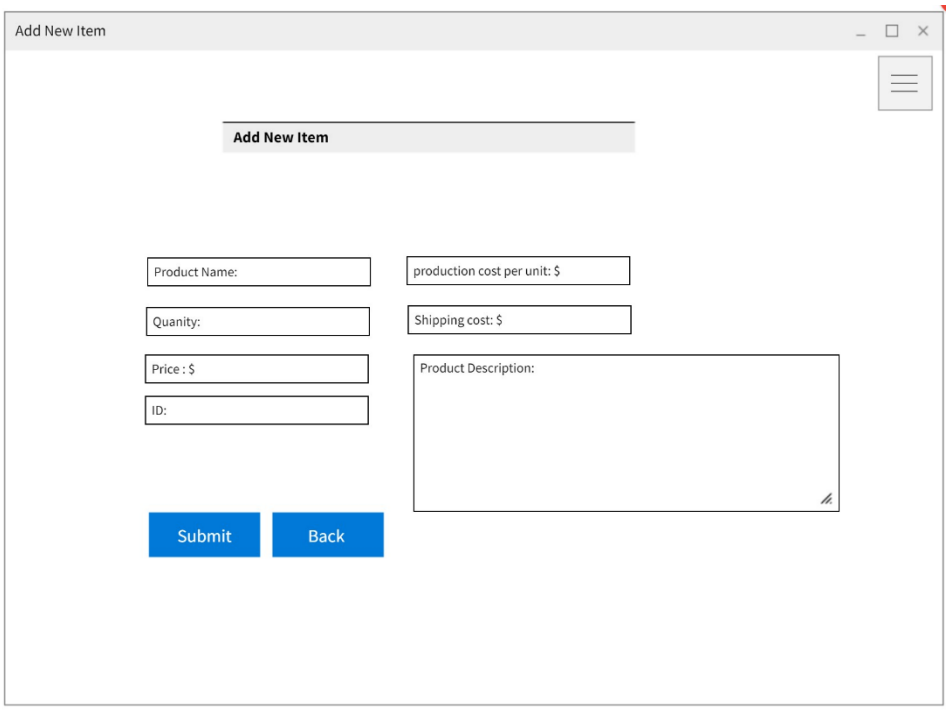
Variant 1: Duplicate Product Entry:

1.1 If the seller tries to edit/add a product that already exists, InventoryController, AddInventory, and AddNewItem prompts the seller to update the existing product instead.

Variant 2: Invalid Product Information:

2.1. If entered product details are invalid (e.g., negative price), InventoryController, prompts for correction.

9. Seller Adds New Product



- **Description:** Allow sellers to introduce new products to the inventory.
- **Actors:** Seller, NewProductView, ProductController, InventoryModel
- **Trigger:** Seller decides to add a new product to the inventory.

Flow:

1. The seller navigates to the "Add New Item" page within the seller dashboard.
2. The seller is presented with a form to fill out the new product's details, including "Product Name," "Quantity," "Price," "ID," "Production cost per unit," "Shipping cost," and "Product Description."
3. The seller enters the name of the product into the "Product Name" field.
4. The seller inputs the number of items available for the product in the "Quantity" field.
5. The seller determines the selling price and enters it into the "Price" field.
6. The seller assigns a unique identifier to the product and enters it into the "ID" field.
7. The seller calculates the production cost per unit and enters it into the respective field to keep track of the product's profitability.
8. The seller inputs the shipping cost for the product, which could be used to calculate shipping charges for the customer.

9. The seller provides a detailed description of the product in the "Product Description" field, which will be visible to customers on the product page.
10. After reviewing all the details and ensuring they are correct, the seller clicks the "Submit" button to add the product to the inventory.
11. If the seller decides to go back or needs to fetch more information before submitting, they can click the "Back" button.

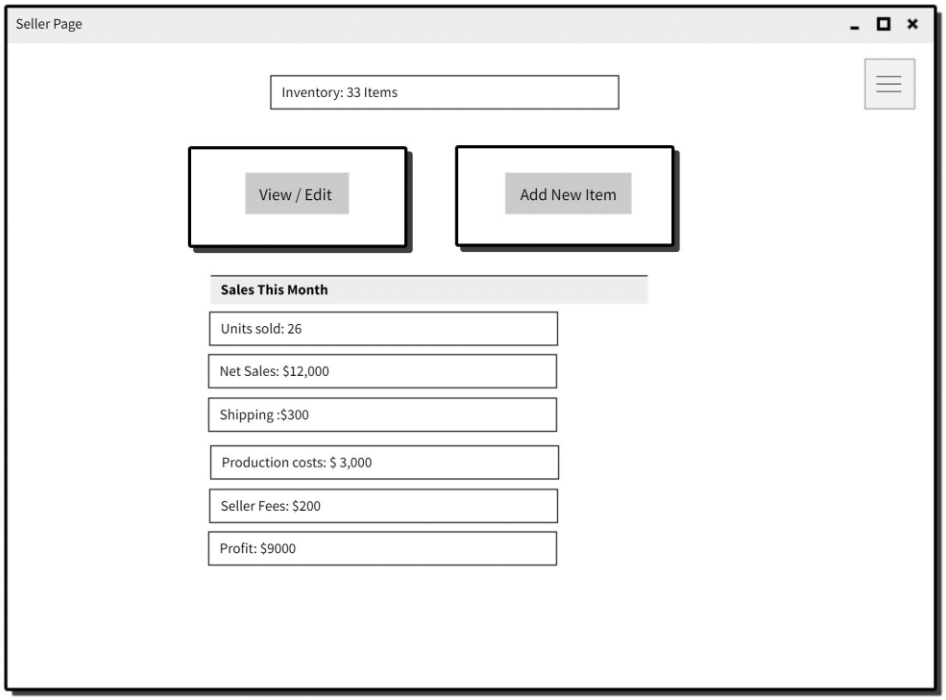
Variant 1: Incomplete Product Details:

1.1. If the seller submits the form with incomplete details, NewProductView prompts for all necessary information.

Variant 2: Duplicate Product:

2.1. If the new product closely matches an existing product, NewProductView suggests updating the existing product instead.

10. Seller Views Sales Page



- **Description:** Seller views information about total sales, revenue, and profit.
- **Actors:** Seller, SalesView, SalesController, SalesModel
- **Trigger:** Seller needs to access information about sales.
- **Preconditions:** Seller has an active account and recorded sales data.

Flow:

1. The seller accesses the "Seller Page" after logging in as a Seller
2. The seller views a summary of their inventory, which currently shows "33 Items."
3. Two main options are presented: "View / Edit" which allows the seller to manage current listings, and "Add New Item" for listing additional products.
4. Below the inventory management options, there is a "Sales This Month" section.
5. The seller reviews the sales metrics provided, including "Units sold," "Net Sales," "Shipping," "Production costs," "Seller Fees," and "Profit."
6. The seller examines the "Units sold" to understand the volume of sales.
7. The seller assesses the "Net Sales" to see the total revenue generated from sales before expenses.
8. The seller considers the "Shipping" costs incurred for the month.
9. The seller evaluates the "Production costs" to ensure they are in line with the expected margins.
10. The seller notes the "Seller Fees" which may include marketplace commissions, transaction fees, or any other applicable charges.
11. The seller calculates their "Profit" by subtracting the total costs from the net sales, which is also displayed on the page.
12. If the seller wishes to adjust prices, production strategies, or marketing efforts based on this data, they may navigate to the "View / Edit" or "Add New Item" sections to make changes.
13. After reviewing the sales data, the seller can navigate elsewhere or log out

Variant 1: No Sales Data:

1.1. If there are no sales records, SalesView displays a message indicating no sales data is available.

Variant 2: Data Access Error:

2.1. If there's an error accessing sales data, SalesView displays an error message and suggests trying again later.

Design Specification

1. User Authentication

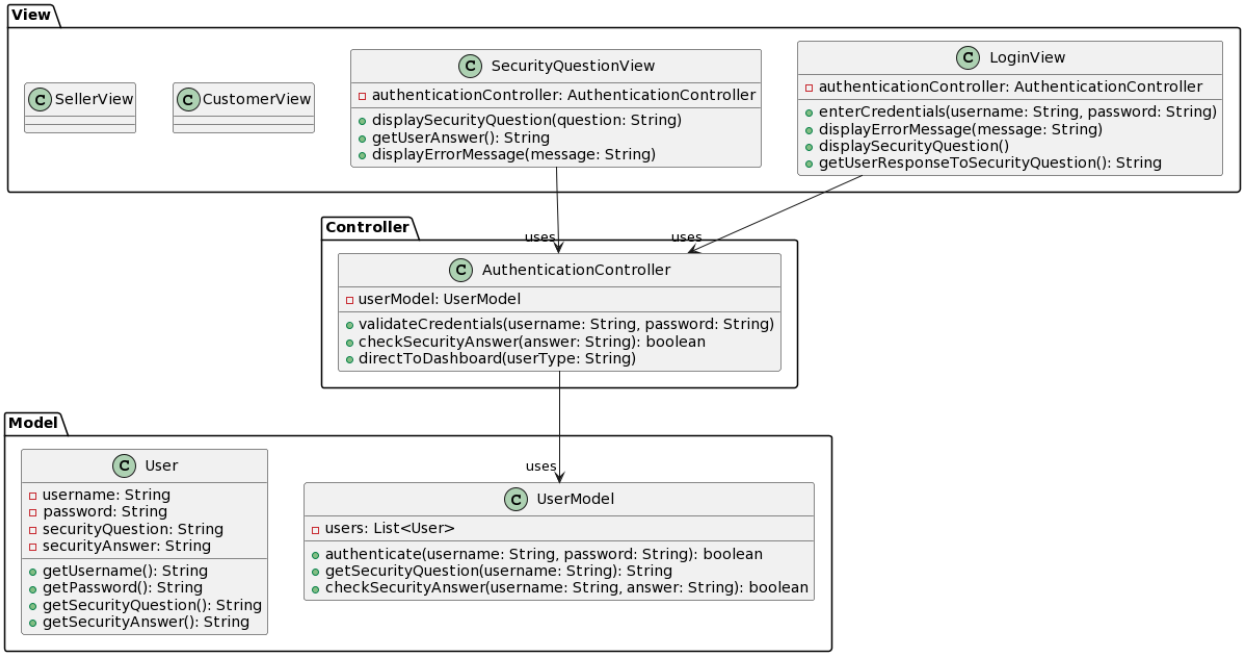
- **CRC Cards:**

LoginView	
• Display login form • Show error messages	• AuthenticationController

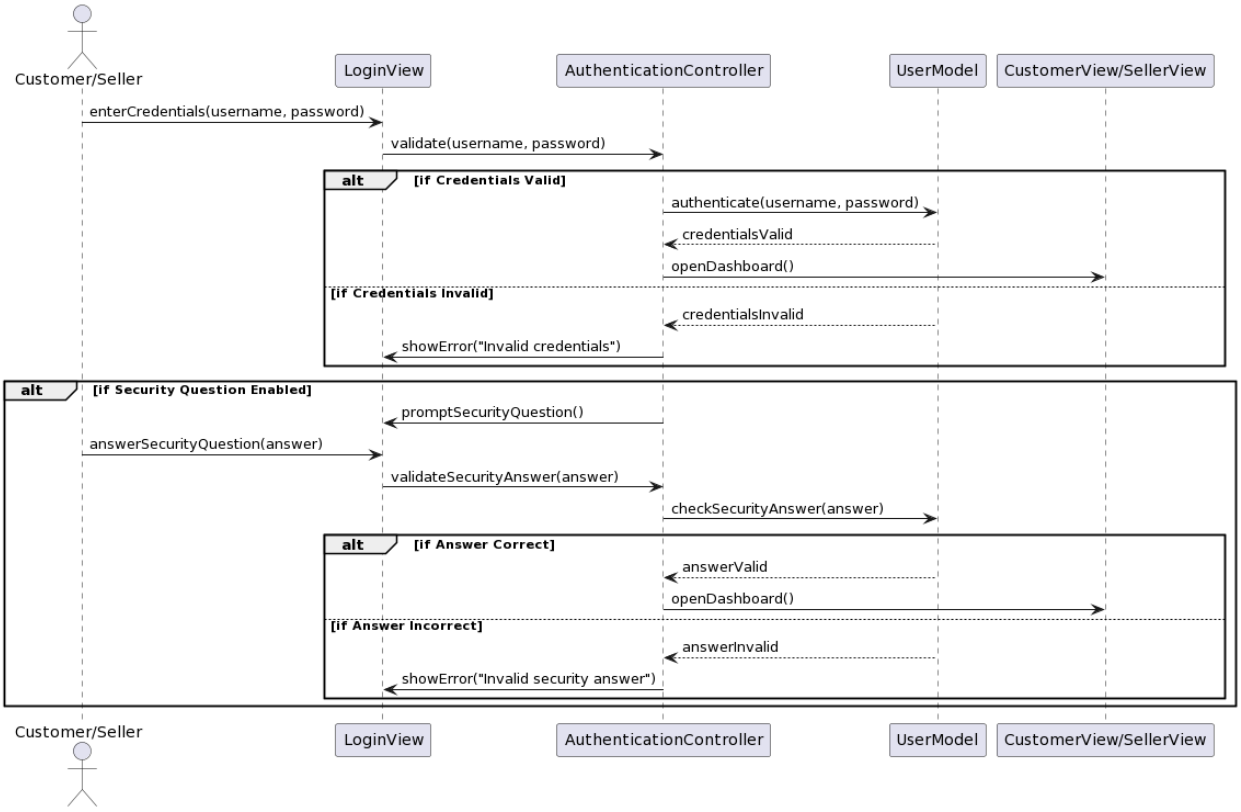
AuthenticationController	
• Validate user credentials • Handle security questions	• UserModel • CustomerView • SellerView

UserModel	
• Store user credentials and security questions • Authenticate user credentials	• AuthenticationController

- **UML Diagram:**



- **Sequence Diagram:**



2. Browse Products

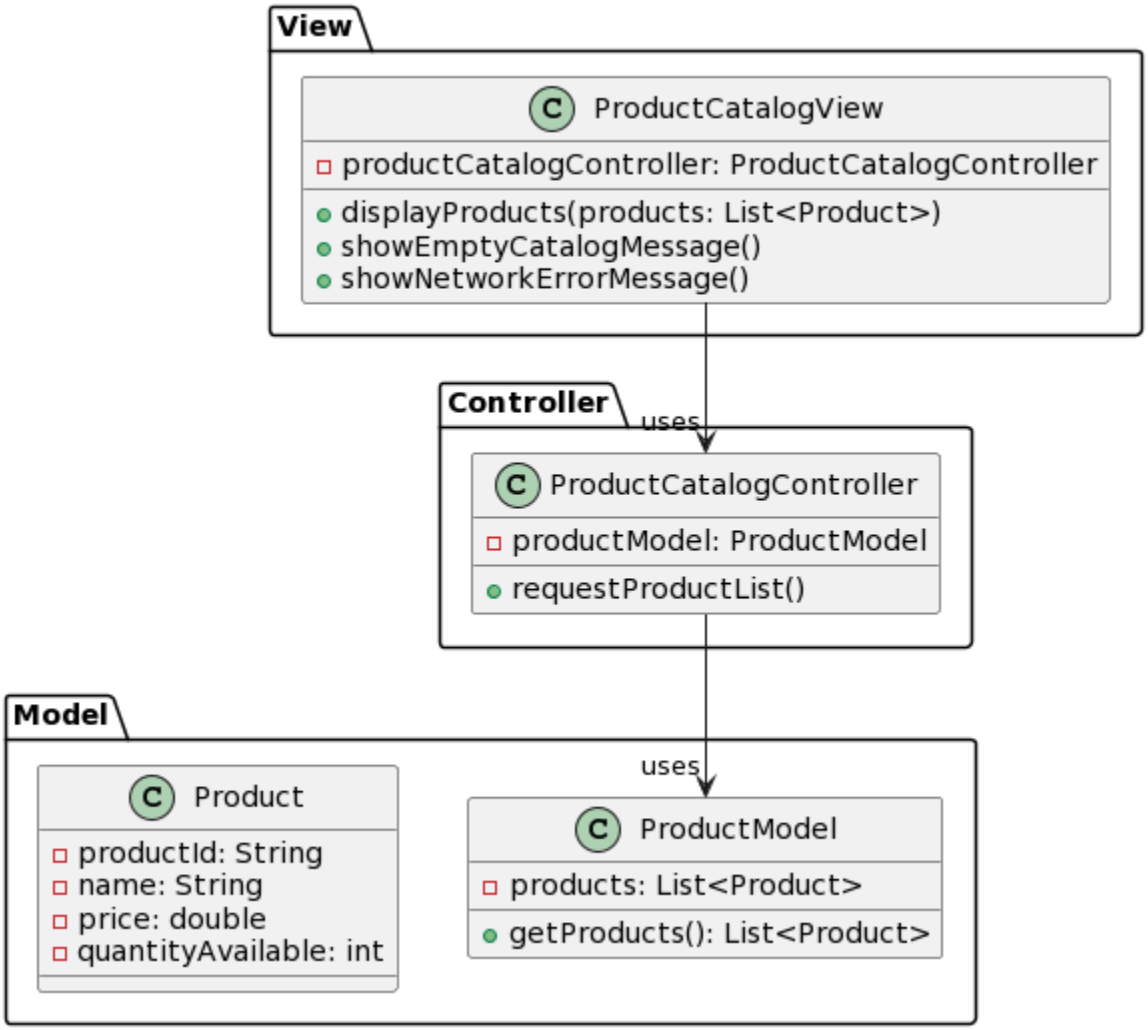
- **CRC Cards:**

ProductCatalogView	
• Display list of products • Handle empty catalog or errors	• ProductCatalogController

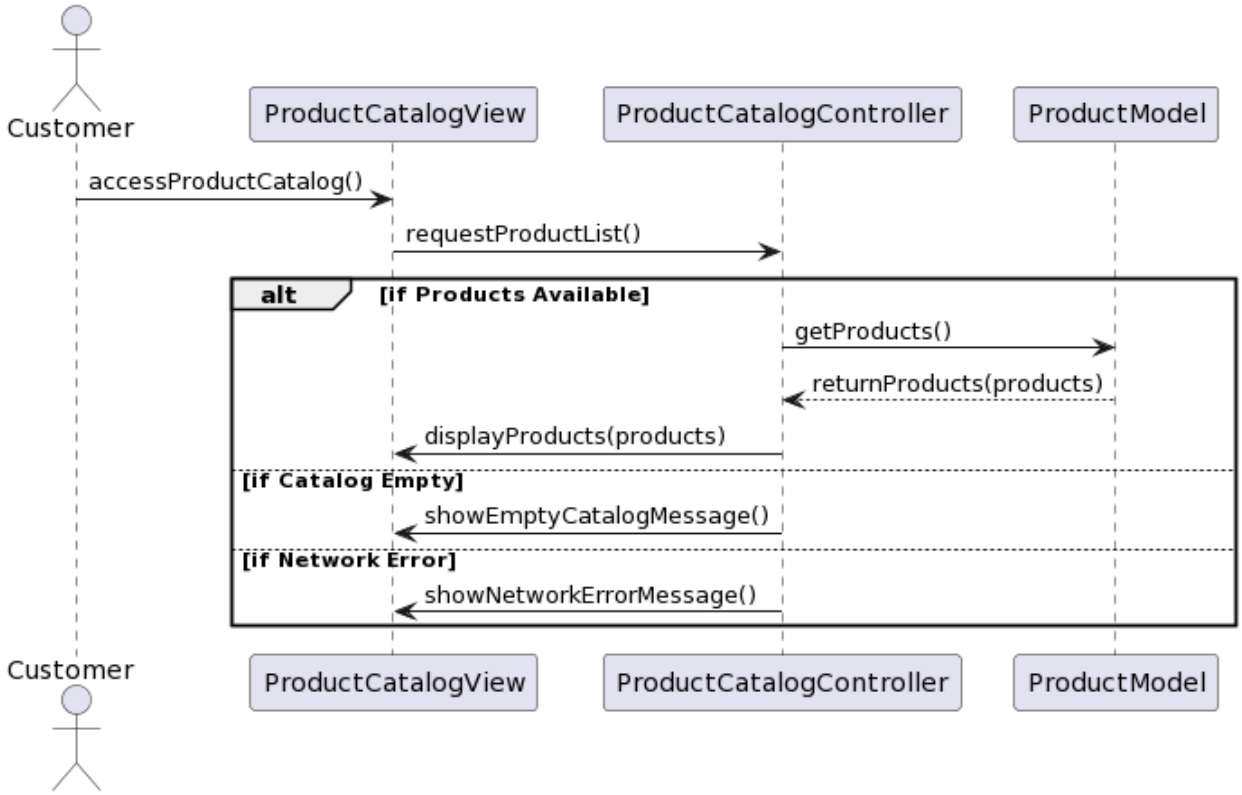
ProductCatalogController	
• Request product list from model • Handle user selection	• ProductModel

ProductModel	
• Store product information • Provide product list	• ProductCatalogController

- **UML Diagram:**


View
ProductCatalogView
productCatalogController: ProductCatalogController
displayProducts(products: List<Product>)
showEmptyCatalogMessage()
showNetworkErrorMessage()
uses
Controller
ProductCatalogController
productModel: ProductModel
requestProductList()
uses
Model
Product
productId: String
name: String
price: double
quantityAvailable: int
ProductModel
products: List<Product>
getProducts(): List<Product>

- **Sequence Diagram:**



3. View Product Details

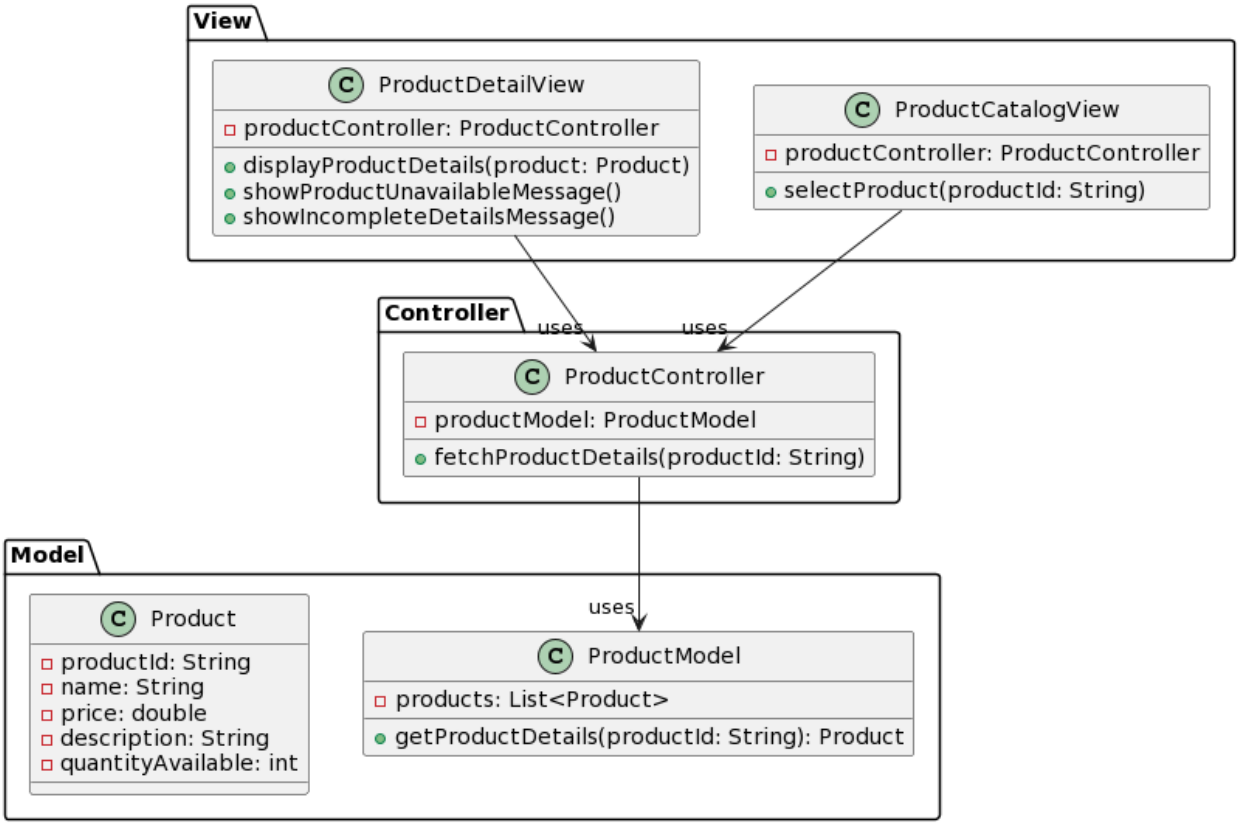
- **CRC Cards:**

ProductDetailView	
• Display detailed information of a product • Handle unavailable or incomplete product details	• ProductController

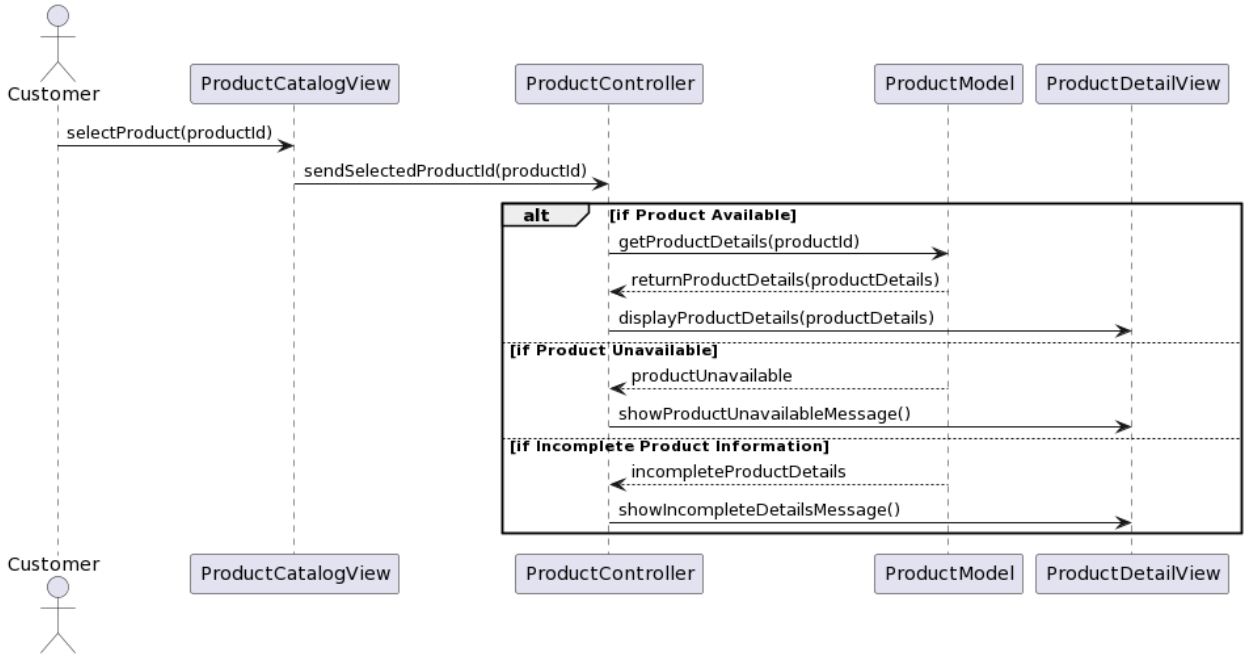
ProductController	
• Fetch product details from the model	• ProductModel

ProductModel	
• Provide detailed product information	• ProductController

- **UML Diagram:**



- **Sequence Diagram:**



4. Add Product to Cart

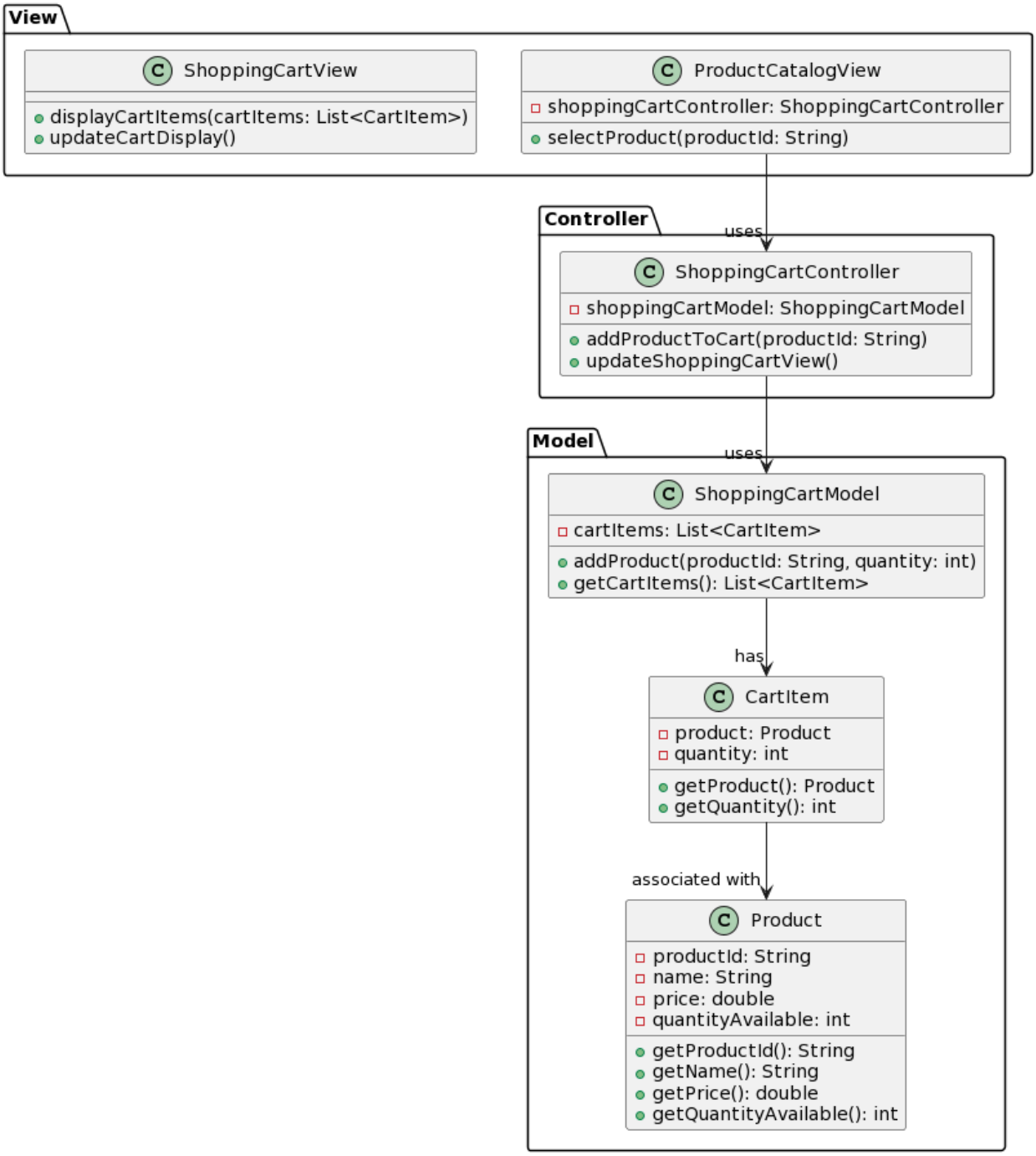
- **CRC Cards:**

ProductCatalogView	
• Display available products • Send selected product info to controller	• ShoppingCartController

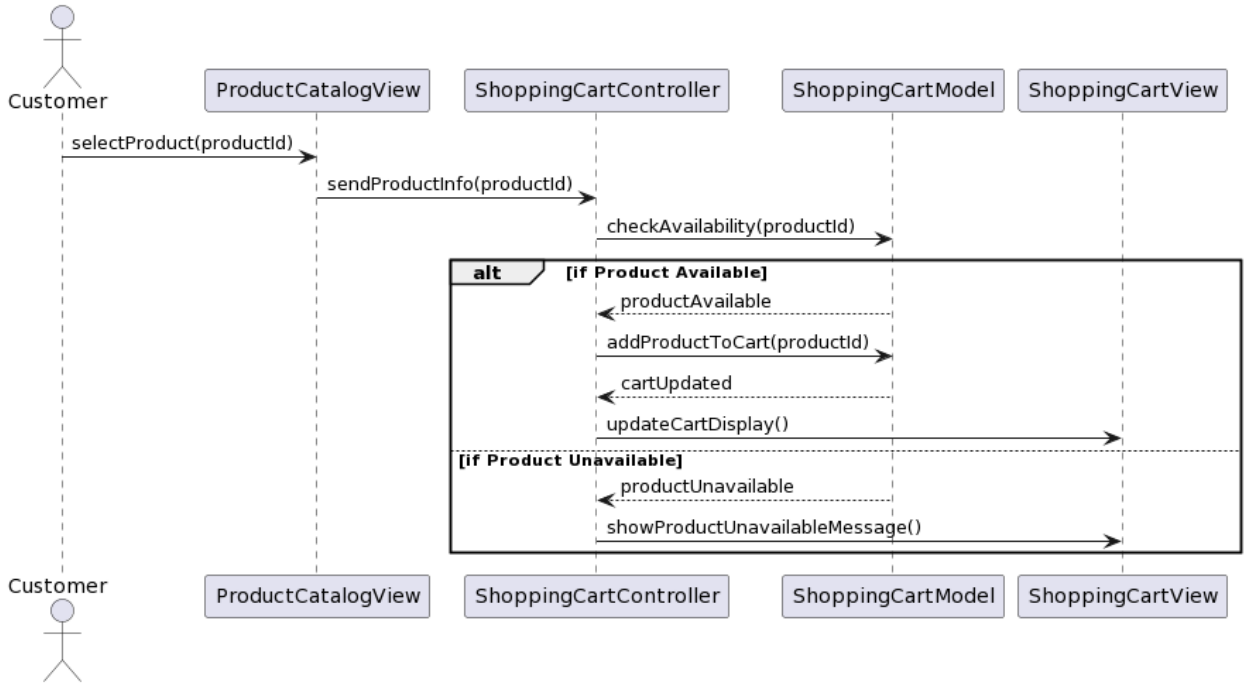
ShoppingCartController	
• Manage adding products to the cart • Update cart view	• ShoppingCartModel

ShoppingCartModel	
• Store cart items • Update quantities and total	• ShoppingCartController

- **UML Diagram:**


View
ShoppingCartView
displayCartItems(cartItems: List<CartItem>)
updateCartDisplay()
ProductCatalogView
shoppingCartController: ShoppingCartController
selectProduct(productId: String)
uses
Controller
ShoppingCartController
shoppingCartModel: ShoppingCartModel
addProductToCart(productId: String)
updateShoppingCartView()
uses
Model
ShoppingCartModel
cartItems: List<CartItem>
addProduct(productId: String, quantity: int)
getCartItems(): List<CartItem>
has
CartItem
product: Product
quantity: int
getProduct(): Product
getQuantity(): int
associated with
Product
productId: String
name: String
price: double
quantityAvailable: int
getProductId(): String
getName(): String
getPrice(): double
getQuantityAvailable(): int

- **Sequence Diagram:**



5. Update Shopping Cart

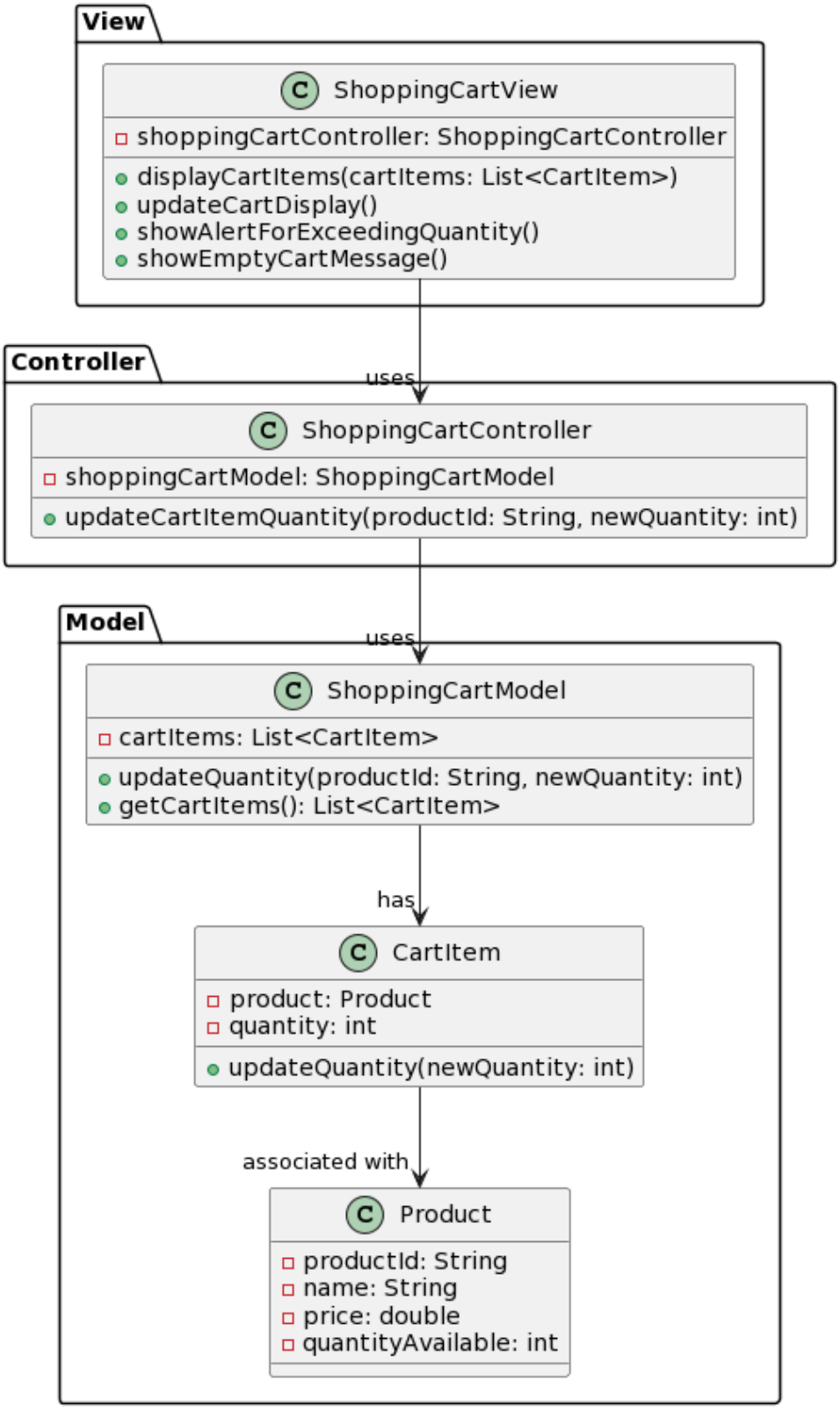
- **CRC Cards:**

ShoppingCartView	
• Display cart items with quantities • Allow quantity adjustments • Show updated cart total	• ShoppingCartController

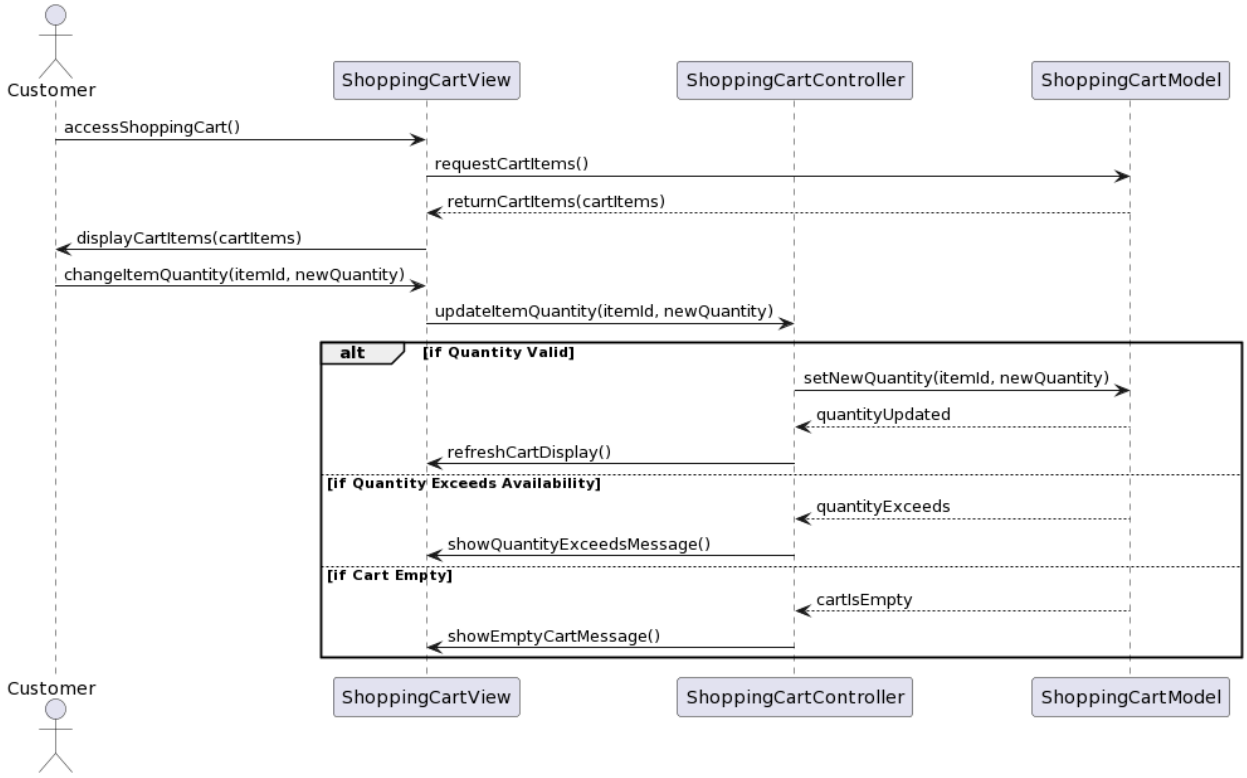
ShoppingCartController	
• Manage cart item quantity updates	• ShoppingCartModel • ShoppingCartView

ShoppingCartModel	
• Store and manage cart items and quantities	• ShoppingCartController

- **UML Diagram:**


View
ShoppingCartView
shoppingCartController: ShoppingCartController
displayCartItems(cartItems: List<CartItem>)
updateCartDisplay()
showAlertForExceedingQuantity()
showEmptyCartMessage()
uses
Controller
ShoppingCartController
shoppingCartModel: ShoppingCartModel
updateCartItemQuantity(productId: String, newQuantity: int)
uses
Model
ShoppingCartModel
cartItems: List<CartItem>
updateQuantity(productId: String, newQuantity: int)
getCartItems(): List<CartItem>
has
CartItem
product: Product
quantity: int
updateQuantity(newQuantity: int)
associated with
Product
productId: String
name: String
price: double
quantityAvailable: int

- **Sequence Diagram:**



6. Checkout

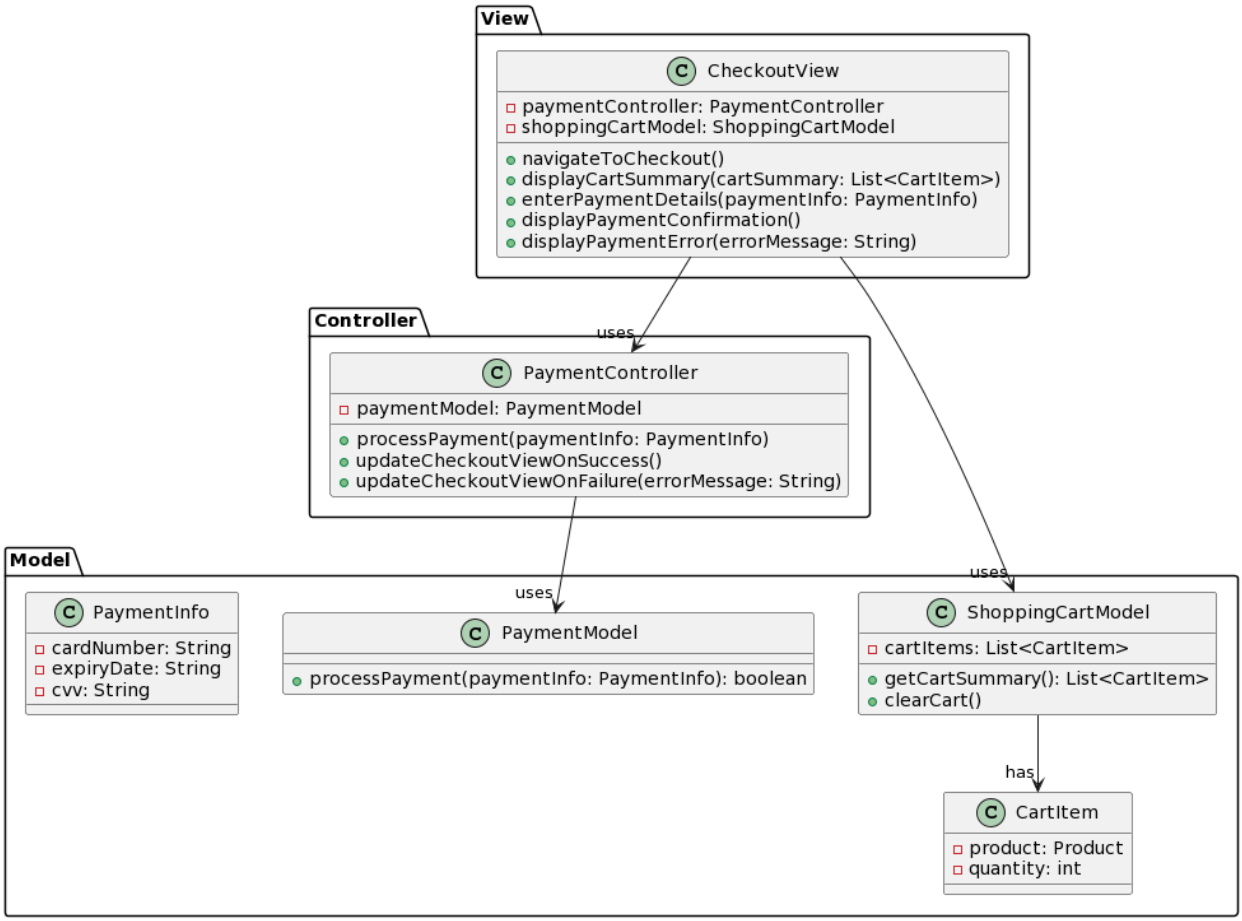
- **CRC Cards:**

CheckoutView	
• Display cart summary • Collect payment information • Show payment confirmation or errors	• PaymentController

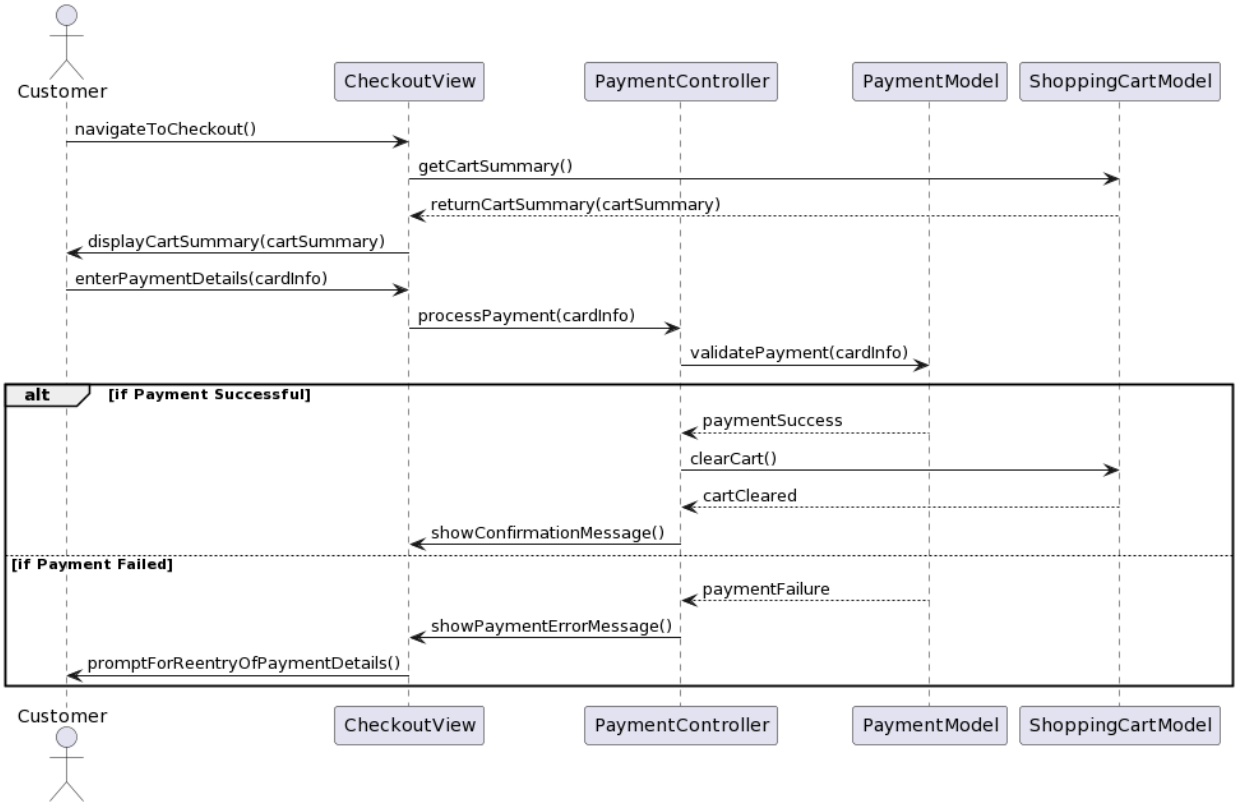
PaymentController	
• Process payment information • Communicate with payment model	• PaymentModel • CheckoutView

PaymentModel	
• Validate and process payment details	• PaymentController

- **UML Diagram:**


View
CheckoutView
paymentController: PaymentController
shoppingCartModel: ShoppingCartModel
navigateToCheckout()
displayCartSummary(cartSummary: List<CartItem>)
enterPaymentDetails(paymentInfo: PaymentInfo)
displayPaymentConfirmation()
displayPaymentError(errorMessage: String)
uses
Controller
PaymentController
paymentModel: PaymentModel
processPayment(paymentInfo: PaymentInfo)
updateCheckoutViewOnSuccess()
updateCheckoutViewOnFailure(errorMessage: String)
uses
uses
Model
PaymentInfo
cardNumber: String
expiryDate: String
cvv: String
PaymentModel
processPayment(paymentInfo: PaymentInfo): boolean
ShoppingCartModel
cartItems: List<CartItem>
getCartSummary(): List<CartItem>
clearCart()
has
CartItem
product: Product
quantity: int

- **Sequence Diagram:**



7. Seller Views Inventory

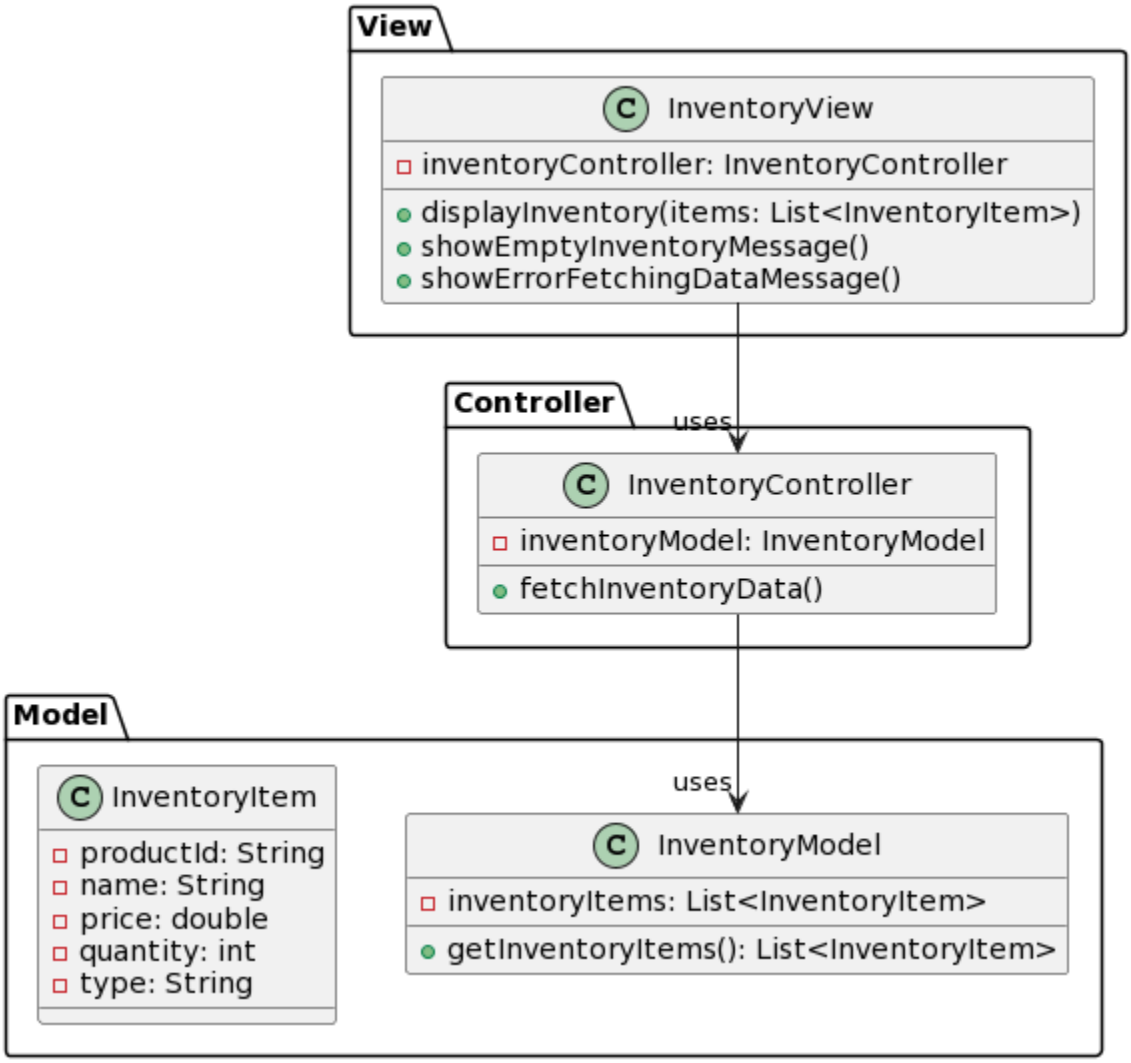
- **CRC Cards:**

InventoryView	
• Display inventory items • Show empty inventory message or errors	• InventoryController

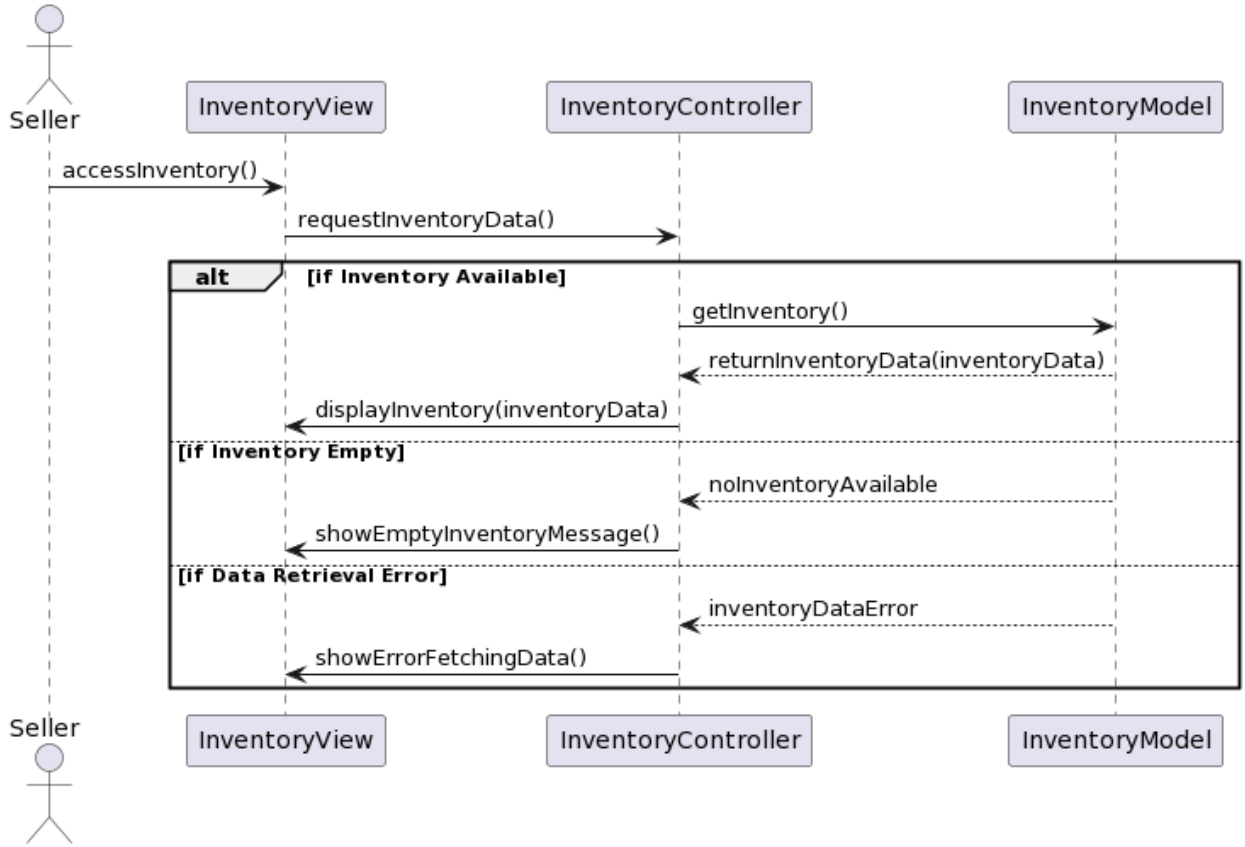
InventoryController	
• Fetch inventory data from the model	• InventoryModel • InventoryView

InventoryModel	
• Store and provide inventory data	• InventoryController

- **UML Diagram:**


View
InventoryView
inventoryController: InventoryController
displayInventory(items: List<InventoryItem>)
showEmptyInventoryMessage()
showErrorFetchingDataMessage()
uses
Controller
InventoryController
inventoryModel: InventoryModel
fetchInventoryData()
uses
Model
InventoryItem
productId: String
name: String
price: double
quantity: int
type: String
InventoryModel
inventoryItems: List<InventoryItem>
getInventoryItems(): List<InventoryItem>

- **Sequence Diagram:**



8. Seller Updates Inventory

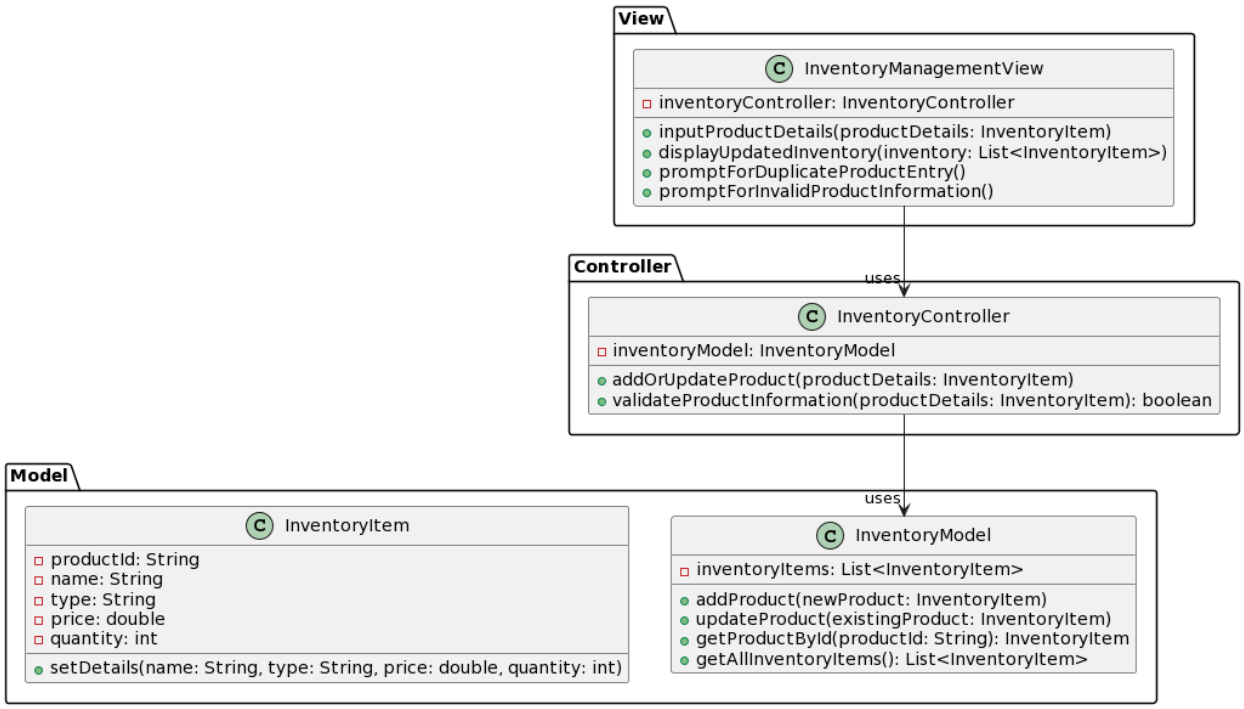
- **CRC Cards:**

InventoryManagementView	
• Allow input of new/updated product details • Display confirmation or correction prompts	• InventoryController

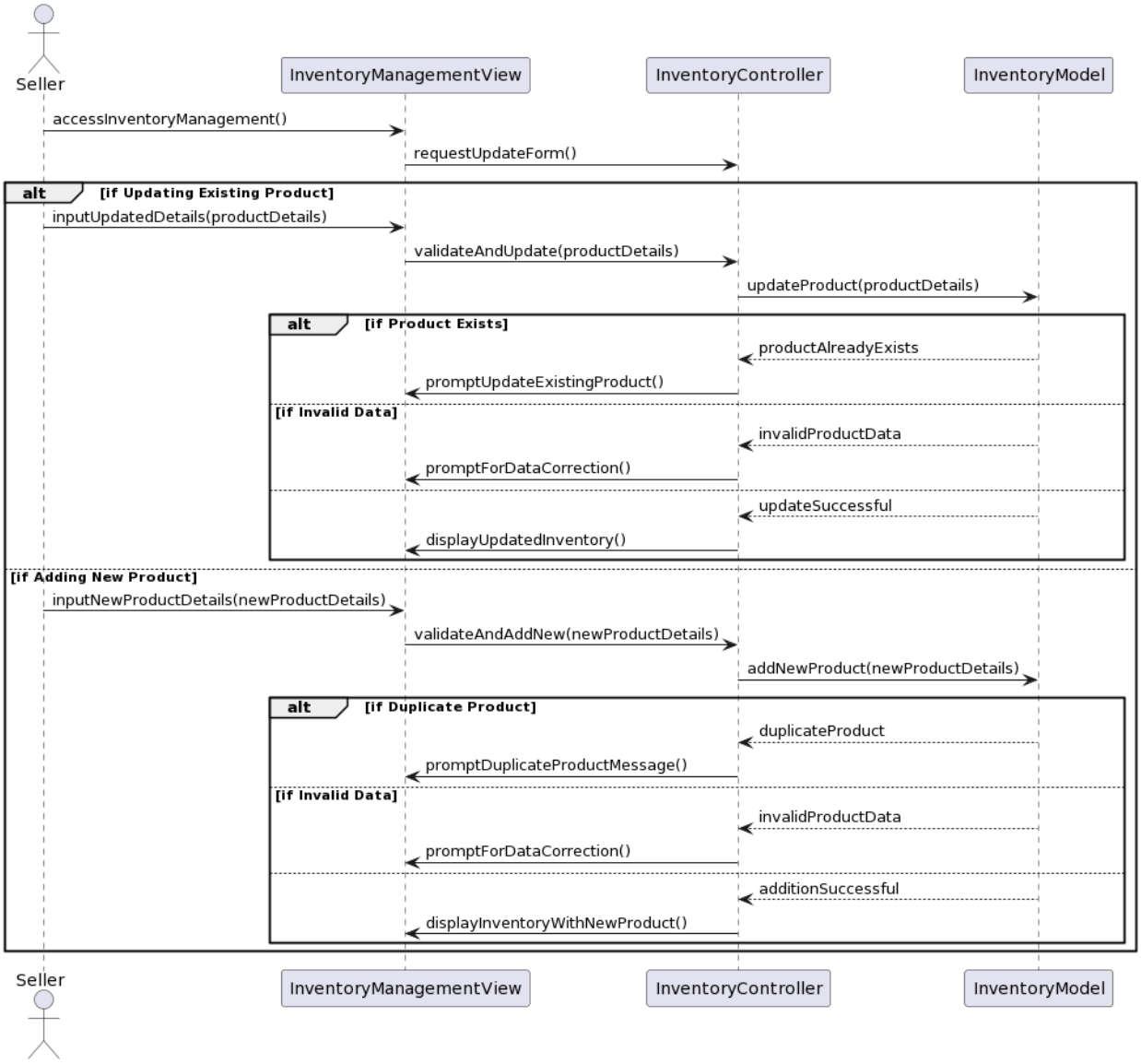
InventoryController	
• Validate and update inventory information	• InventoryModel • InventoryManagementView

InventoryModel	
• Manage inventory details (add/update products)	

- **UML Diagram:**



- **Sequence Diagram:**



9. Seller Adds New Product

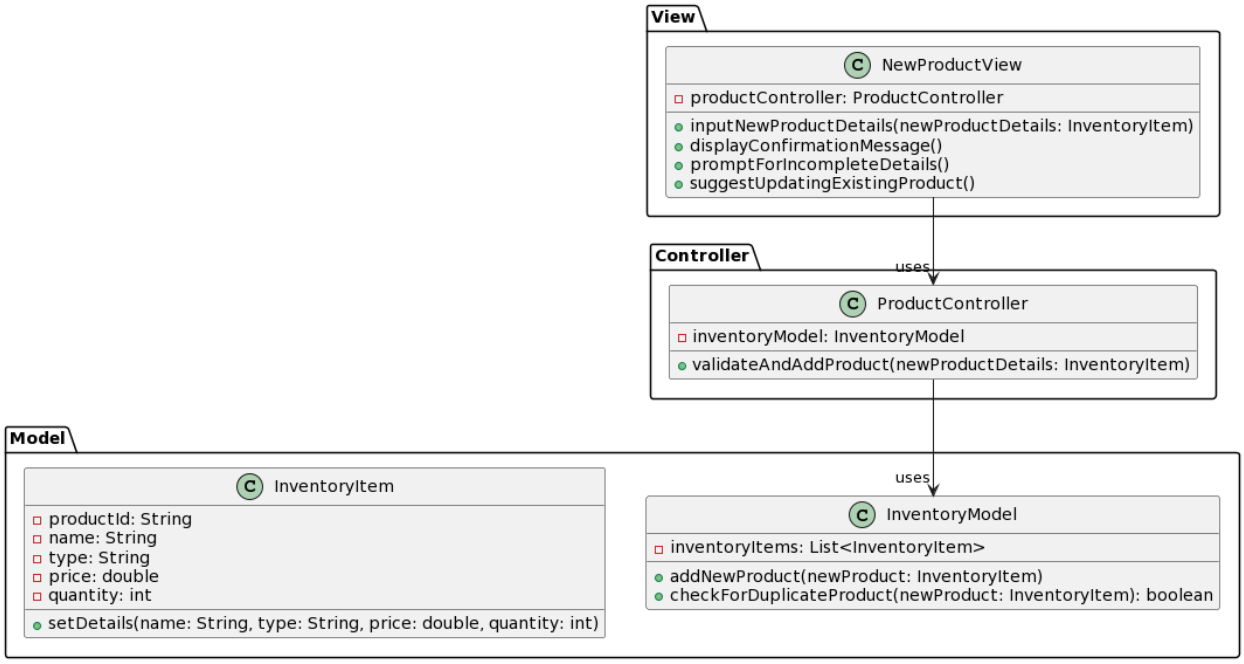
- **CRC Cards:**

NewProductView	
• Display form for new product details • Show confirmation or error messages	• ProductController

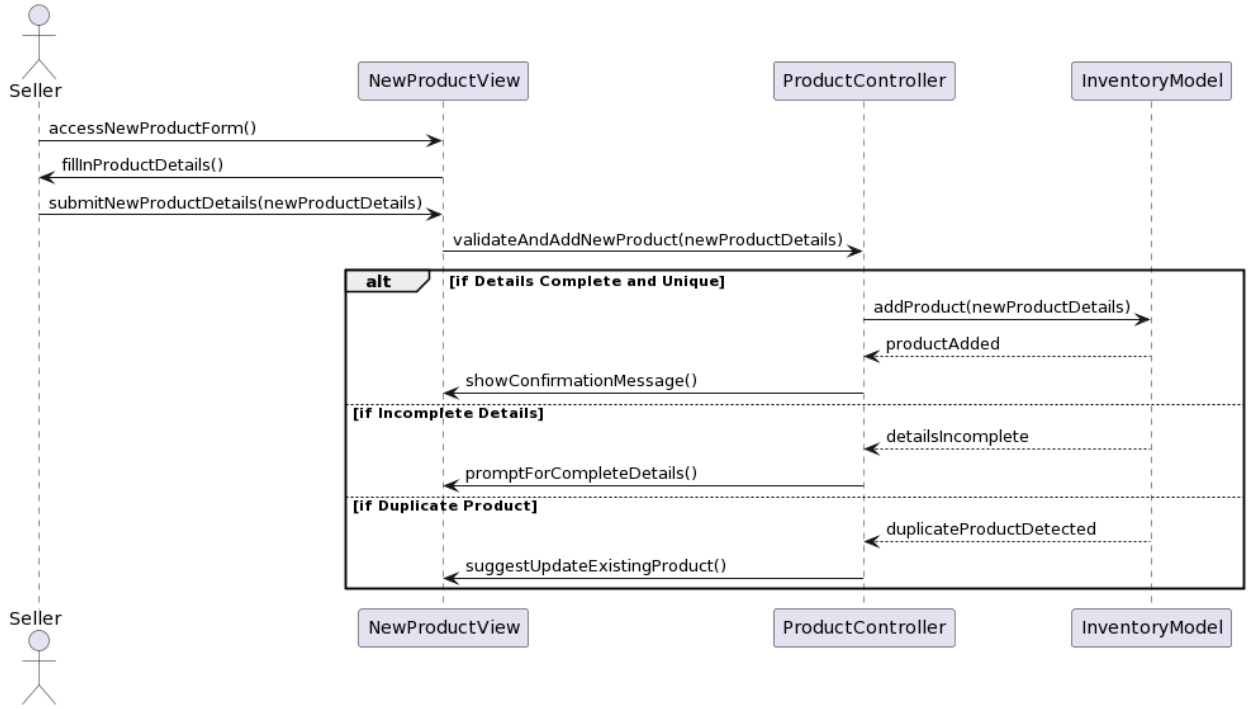
ProductController	
• Validate and add new product to inventory	• InventoryModel • NewProductView

InventoryModel	
• Store and manage new product entries	• ProductController

- **UML Diagram:**



- **Sequence Diagram:**



10. Seller Views Sales Page

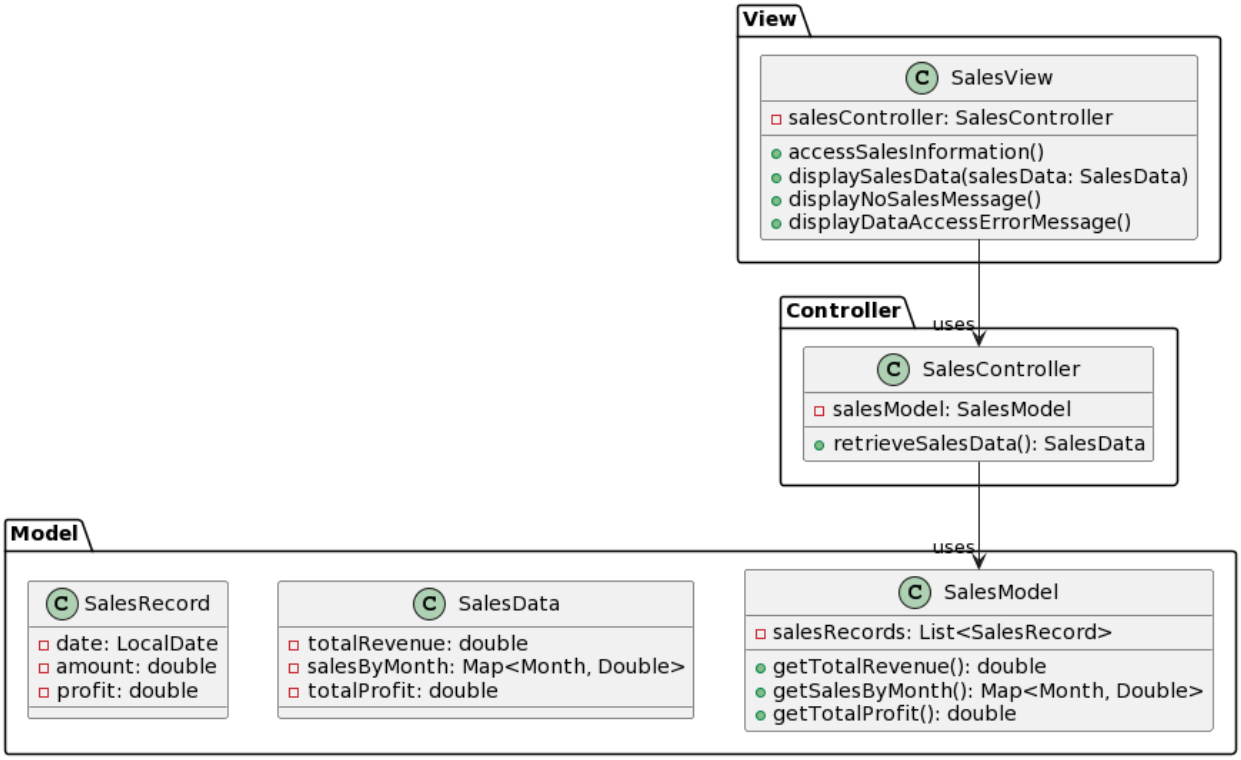
- **CRC Cards:**

SalesView	
• Display sales data (revenue, monthly sales, profit) • Show no data or error messages	• SalesController

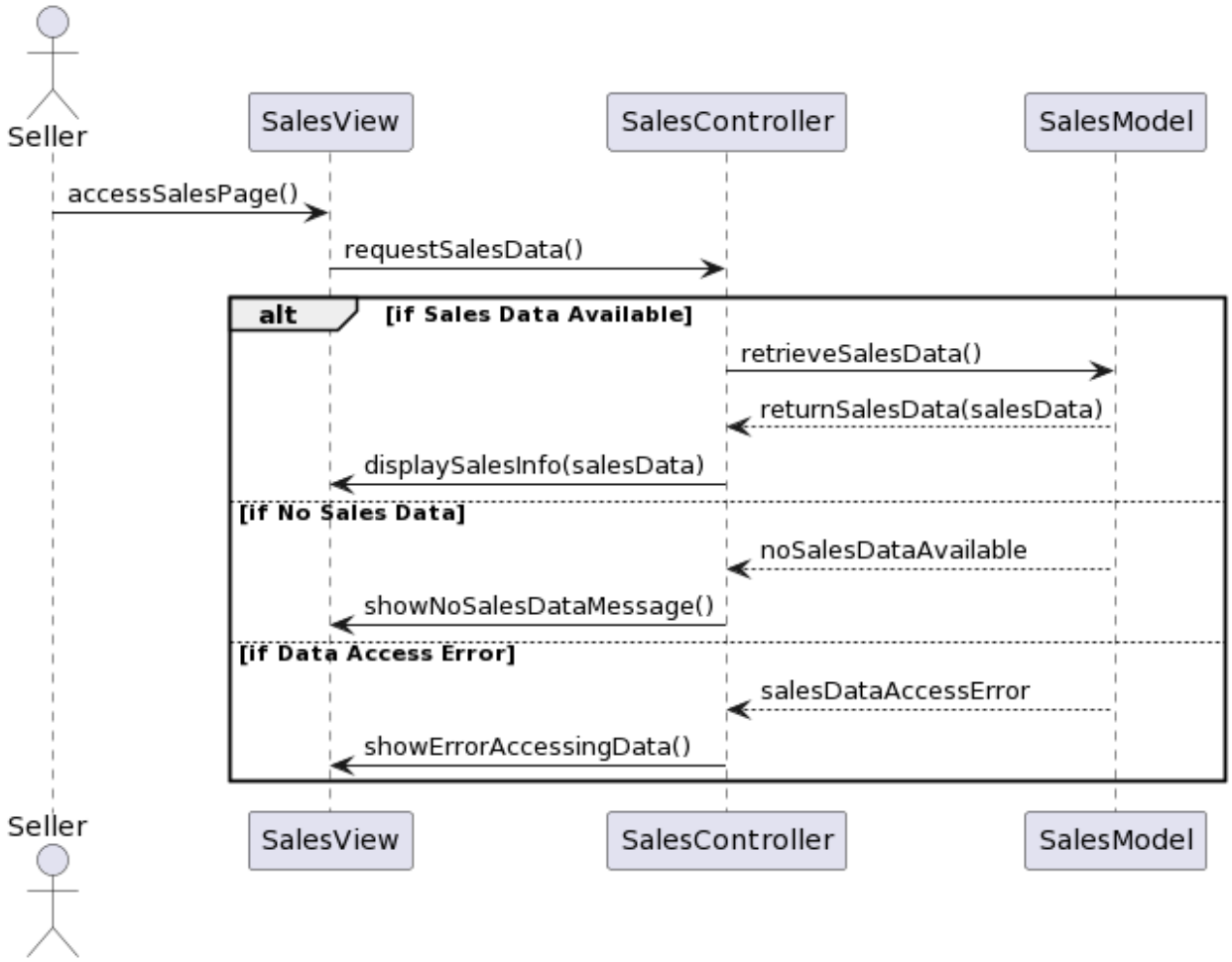
SalesController	
• Retrieve and process sales data for display	• SalesModel • SalesView

SalesModel	
• Store and provide sales records and calculations	• SalesController

- **UML Diagram:**

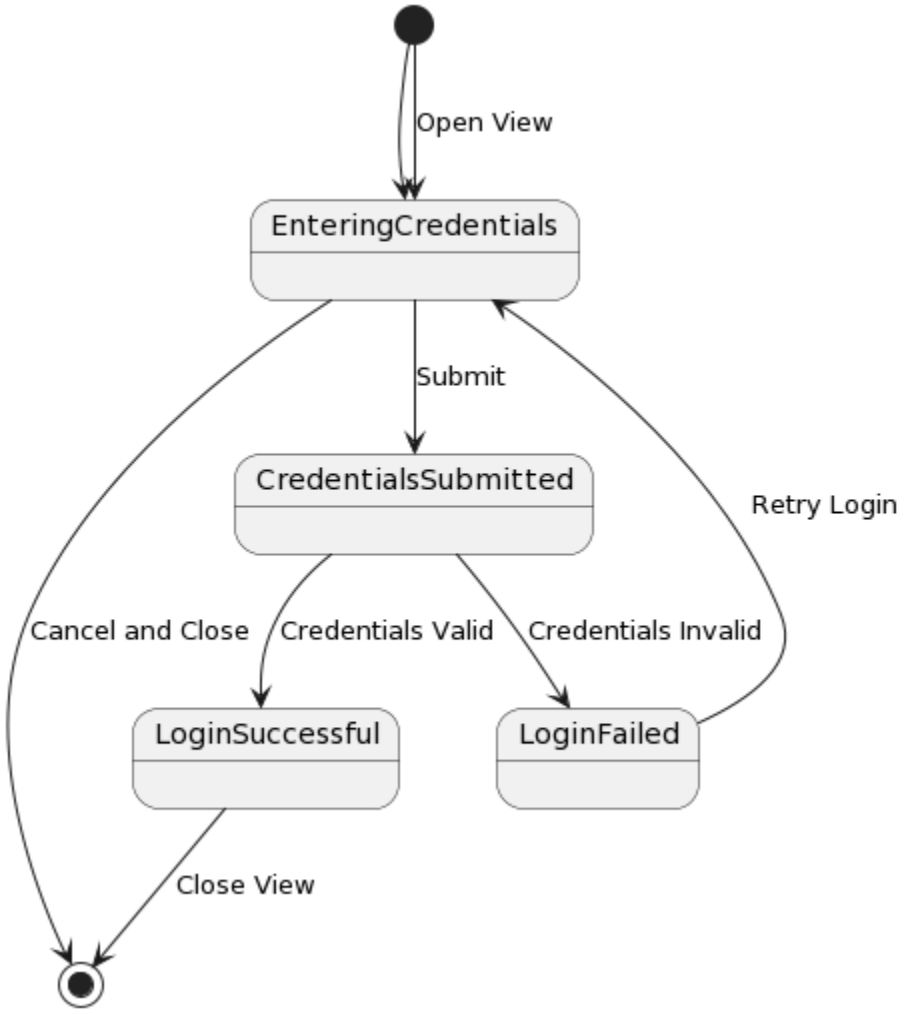

View
SalesView
salesController: SalesController
accessSalesInformation()
displaySalesData(salesData: SalesData)
displayNoSalesMessage()
displayDataAccessErrorMessage()
uses
Controller
SalesController
salesModel: SalesModel
retrieveSalesData(): SalesData
uses
Model
SalesRecord
date: LocalDate
amount: double
profit: double
SalesData
totalRevenue: double
salesByMonth: Map<Month, Double>
totalProfit: double
SalesModel
salesRecords: List<SalesRecord>
getTotalRevenue(): double
getSalesByMonth(): Map<Month, Double>
getTotalProfit(): double

- **Sequence Diagram:**

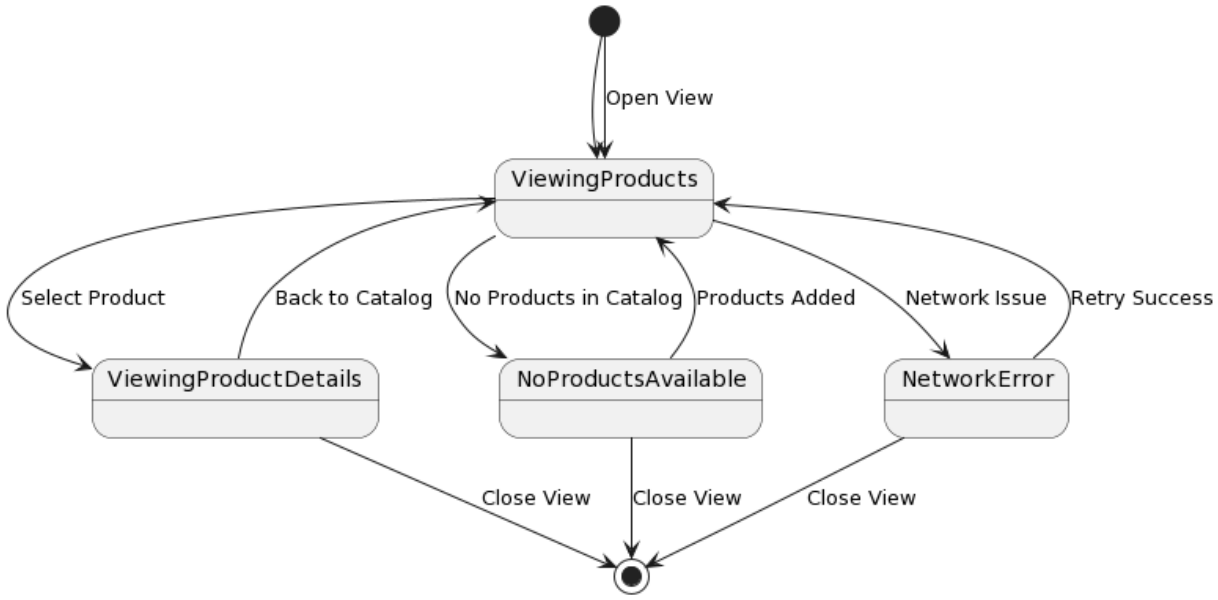


State Diagrams

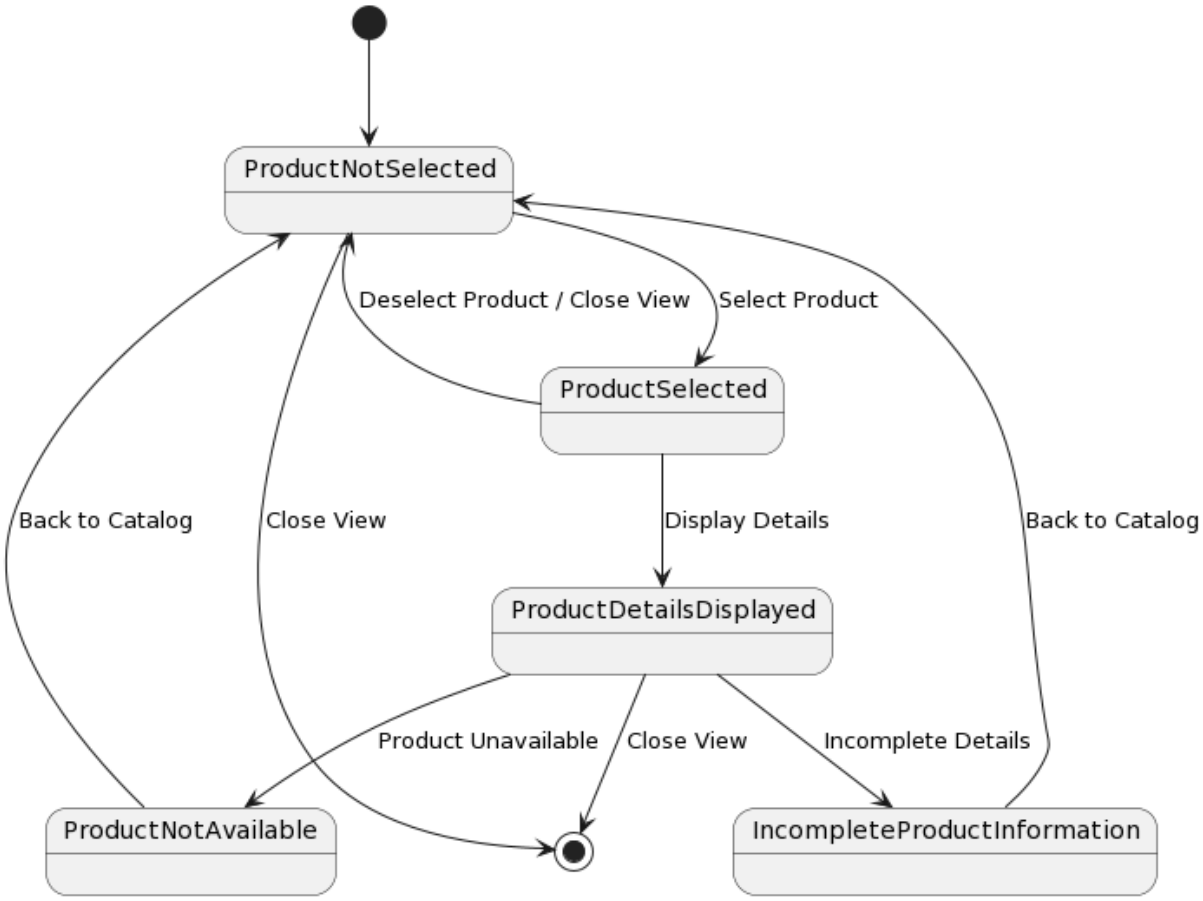
State Diagram for LoginView:



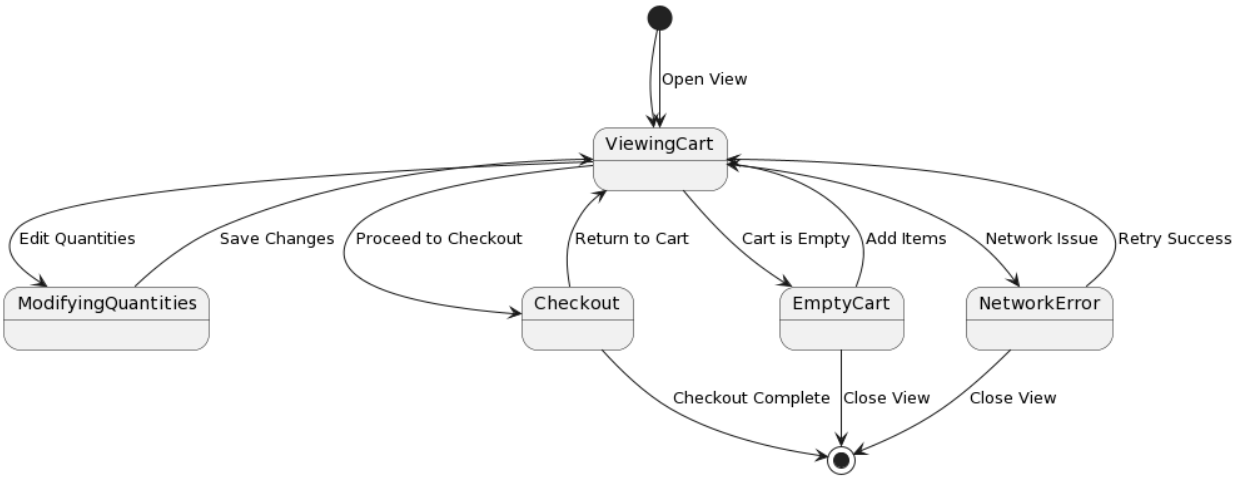
State Diagram for ProductCatalogView:



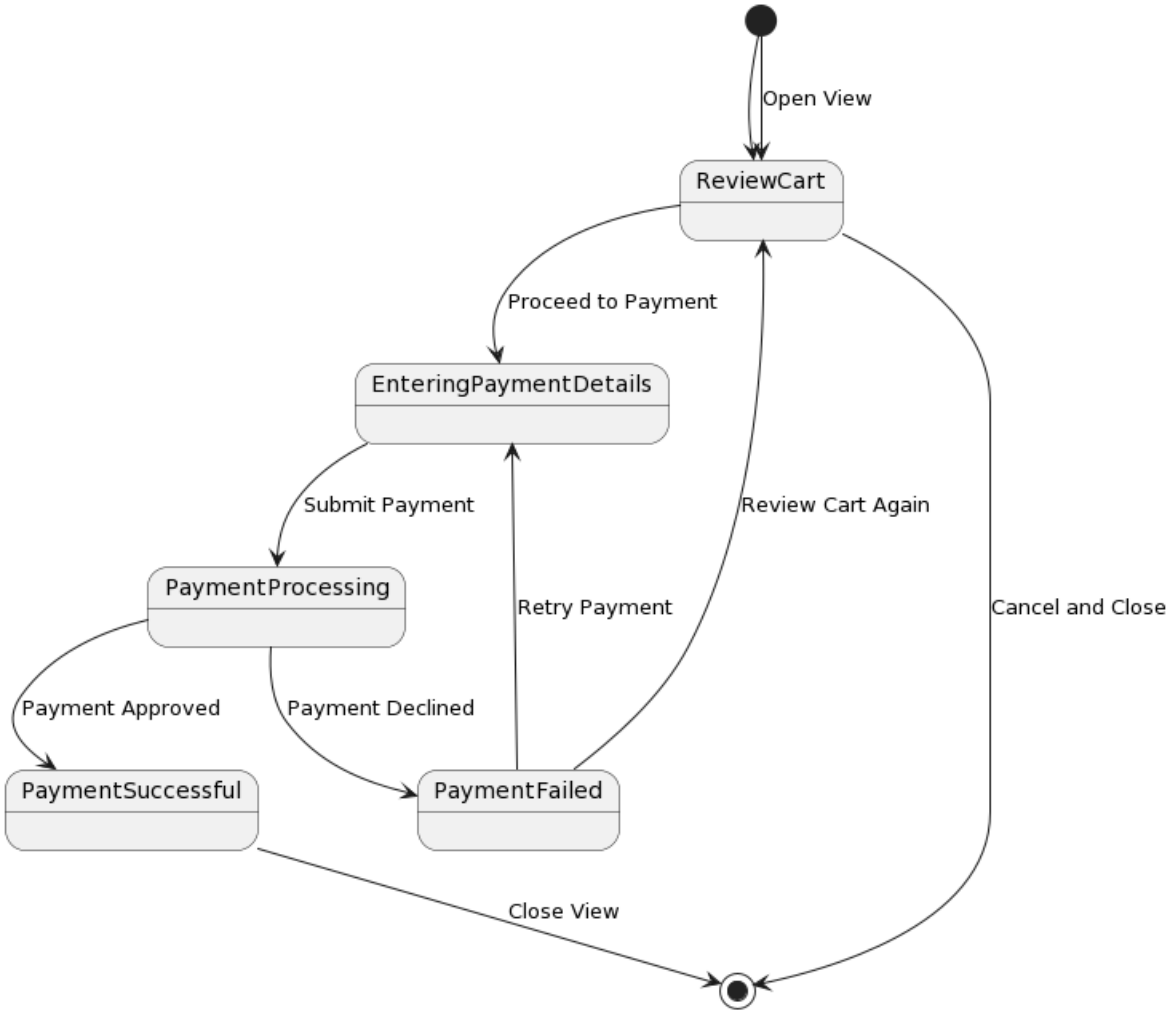
State Diagram for ProductDetailView:



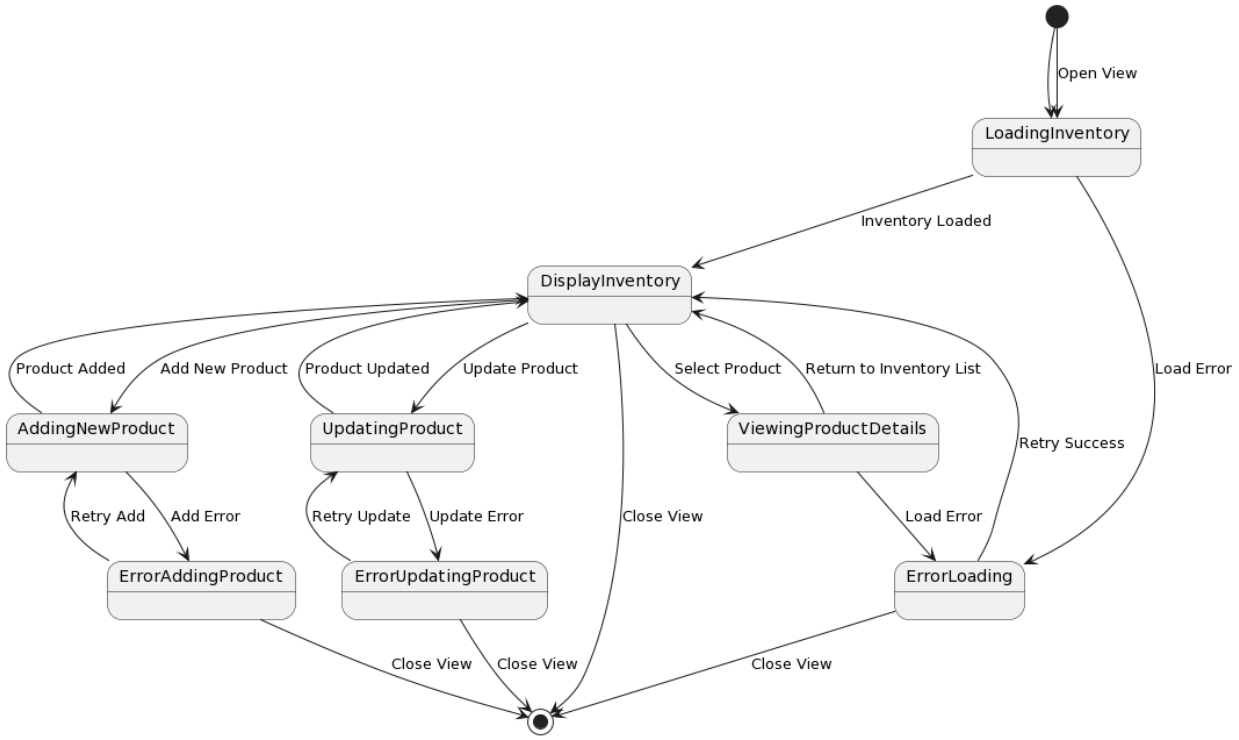
State Diagram for ShoppingCartView:



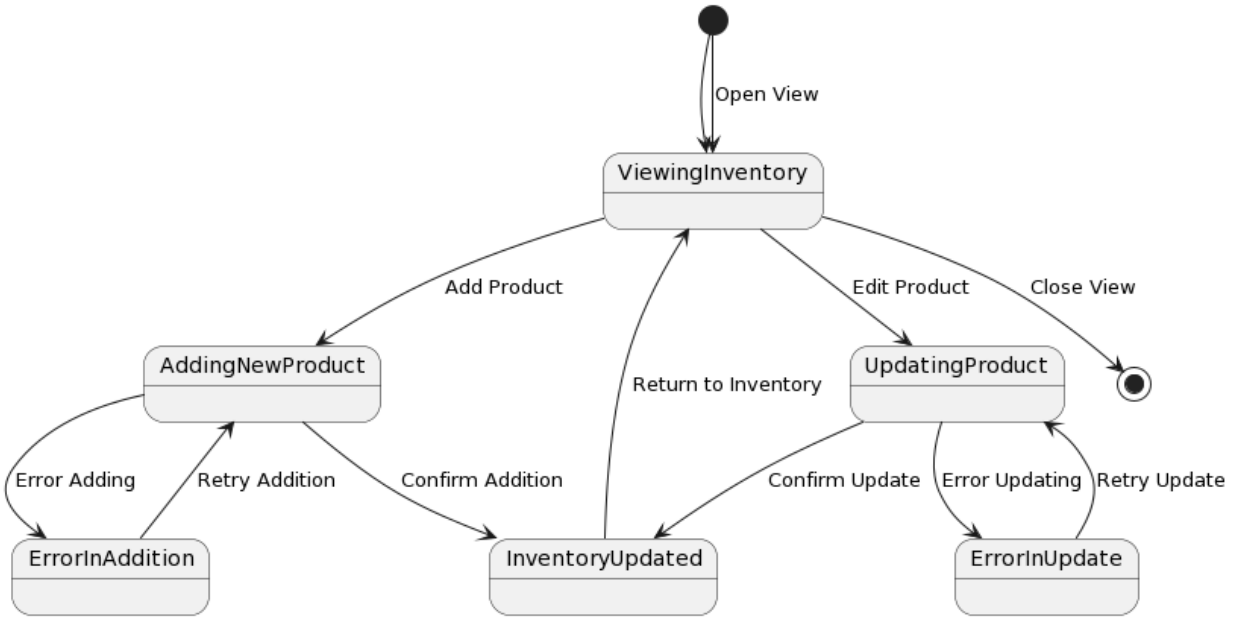
State Diagram for CheckoutView:



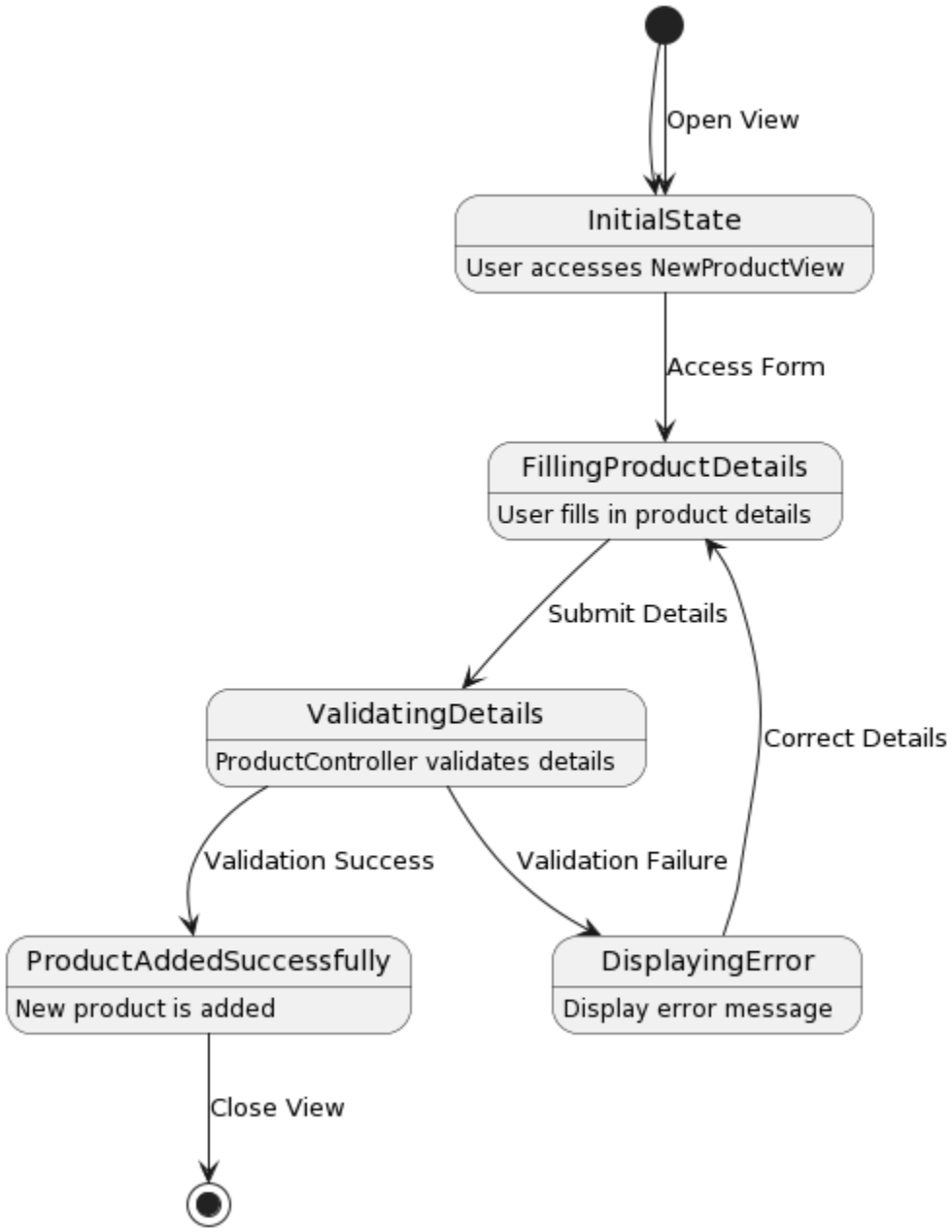
State Diagram for InventoryView:



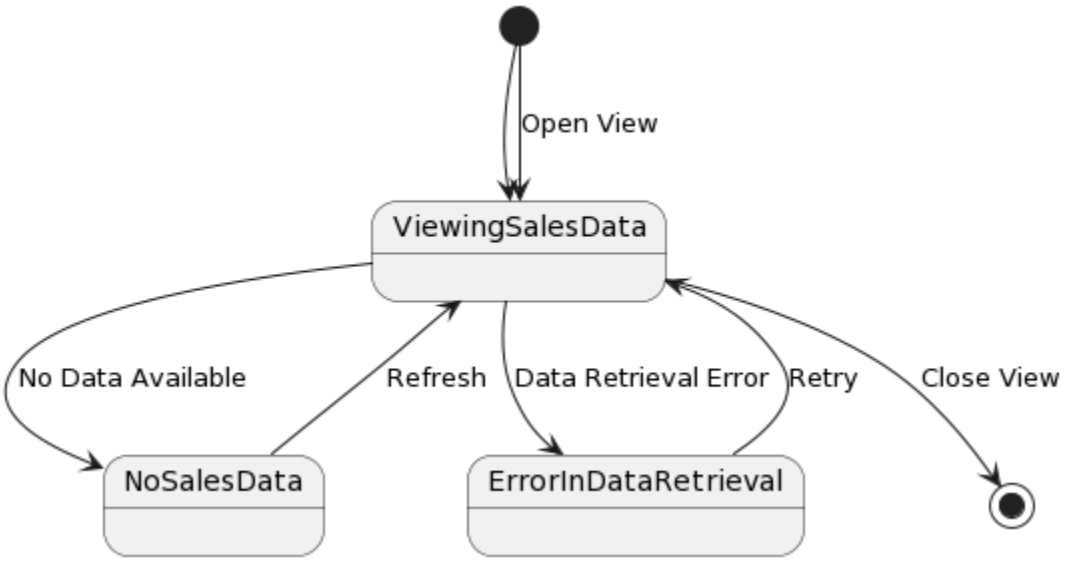
State Diagram for InventoryManagementView:



State Diagram for NewProductView:



State Diagram for SalesView:



Design Patterns

1. User Authentication

Design Pattern Element	Swing UI Components/Concepts
Model	UserModel
View	JPanel, JFrame
Controller	AuthenticationController
View Components	JTextField, JPasswordField
Observer	ChangeListener
Concrete Observer	The class that implements the ChangeListener interface type
Attach Observer	addChangeListener()
Notify Observers	Method calls when state changes (e.g., stateChanged())

2.Browse Products

Design Pattern Element	Swing UI Components/Concepts
Model	ProductModel (holds data about products)
View	ProductCatalogView (displays the product information)
Controller	ProductCatalogController (handles the interaction logic)
Data Transfer Object	Product (serves to transfer product data)
List Model	DefaultListModel (stores the list of Product objects for the JList)
List Component	JList (displays the products)
Action Listener	ActionListener (handles user actions like button clicks)
List Selection Listener	ListSelectionListener (handles the event of a product being selected from the list)
Observer Pattern	Any Swing component that registers listeners
Composite Pattern	JPanel containing other components
Adapter Pattern	Could be used to adapt the interface of the ProductModel for use with Swing components

3. View Product Details

Design Pattern Element	Swing UI Components/Concepts
Model	ProductModel (data and logic about products)
View	ProductDetailView, ProductCatalogView (display information to the user)
Controller	ProductController (handles business logic and user interaction)
Observer Pattern	PropertyChangeListener
Strategy Pattern	Could be used for different ways to display products (e.g., different detail views)
Adapter Pattern	If needed to adapt ProductModel methods to Swing components' expected interfaces
Builder Pattern	Could be used to construct complex Product instances step by step

3. Add Product to Cart

Design Pattern Element	Swing UI Components/Concepts
Model	ShoppingCartModel, Product
View	ShoppingCartView, ProductCatalogView
Controller	ShoppingCartController
View Components	JList, JButton
Action Listener	ActionListener
Action Event	ActionEvent
Observer	ListModelListener
Concrete Observer	The class that implements ListModelListener
Attach Observer	addListDataListener()
Notify Observers	Method calls when list data changes (e.g., contentsChanged())

5. Update Shopping Cart

Design Pattern Element	Swing UI Components/Concepts
Model	ShoppingCartModel
View	ShoppingCartView
Controller	ShoppingCartController
Command	Interface for cart update operations
Concrete Command	Implementation of update cart item quantity
Observer - Subject	ShoppingCartModel (observable)
Observer - Observer	Components within ShoppingCartView (observers)
Strategy	Interface for strategies to update quantities
Concrete Strategy	Specific implementations for update strategies
Factory Method - Creator	Interface for creating CartItem instances
Factory Method - ConcreteCreator	Implementation of CartItem creation

6. Checkout

Design Pattern Element	Swing UI Components/Concepts
Model	PaymentModel, ShoppingCartModel
View	CheckoutView
Controller	PaymentController
Strategy	Interface for payment processing strategies
ConcreteStrategy	Specific implementations for payment processing (e.g., credit card, PayPal)
Observer - Subject	PaymentModel (observable)
Observer - Observer	Components within CheckoutView (observers)
State - Context	CheckoutView (maintains instance of state)
State - State	Interface for different states of the checkout process
State - ConcreteState	Specific states (e.g., viewing cart, entering payment, confirmation)

7. Seller Views Inventory

Design Pattern Element	Swing UI Components/Concepts
Model	InventoryModel
View	InventoryView
Controller	InventoryController
Observer - Subject	InventoryModel (observable)
Observer	Components within InventoryView (observers)
Concrete Command	Implementation of a command to fetch inventory data

8. Seller Updates Inventory

Design Pattern Element	Swing UI Components/Concepts
Model	InventoryModel
View	InventoryManagementView
Controller	InventoryController
Command	Method addOrUpdateProduct in InventoryController
ConcreteCommand	Actual implementation of addOrUpdateProduct within the controller
Strategy	Method validateProductInformation in InventoryController
Concrete Strategy	Specific validation logic within validateProductInformation
Observer - Subject	InventoryModel
Observer	Any GUI component that updates based on changes in InventoryModel

9. Seller Adds new Product

Design Pattern Element	Swing UI Components/Concepts
Model	InventoryModel
View	NewProductView
Controller	ProductController
Command	Interface for product creation operations
ConcreteCommand	Implementation of the command to add a new product
Strategy	Interface for validation strategies
ConcreteStrategy	Specific validation strategies for new product details
Observer - Subject	InventoryModel (observable)
Observer	Components within NewProductView (observers)
Builder Pattern - Builder	Interface for step-by-step construction of InventoryItem
Builder Pattern - ConcreteBuilder	Implementation for constructing instances of InventoryItem
Factory Method - Creator	Interface for creating InventoryItem instances
Factory Method - ConcreteCreator	Implementation of InventoryItem creation

10. Seller Views Sales Page

Design Pattern Element	Swing UI Components/Concepts
Model	SalesModel
View	SalesView
Controller	SalesController
Observer	SalesView
Template Method - AbstractClass	SalesModel could define a template method for calculating sales metrics
Template Method - ConcreteClass	Specific calculations within SalesModel following the template method
Facade	SalesController acting as a facade to simplify the interaction between the view and multiple models
Builder Pattern - Builder	SalesRecord, SalesData
Builder Pattern - ConcreteBuilder	The specific construction process of SalesRecord or SalesData
Factory Method - Creator	Creation methods within SalesModel for SalesData objects
Factory Method - ConcreteCreator	method in SalesModel that creates a SalesData object

Source Code

AddInventory.Java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.FocusAdapter;
import java.awt.event.FocusEvent;

/**
 * This class represents the user interface for adding inventory to a
product.
 * It provides fields for entering product ID and the quantity to add, and
updates the inventory accordingly.
 * @author george martinez
 * @author freddy ingle
 * @author bryan cooke
 */
public class AddInventory extends JFrame {
    private JTextField productIdField, quantityField;
    private JButton submitButton;

    /**
     * Constructs the AddInventory interface.
     */
    public AddInventory() {
        createUI();
    }
     /**
     * Creates the user interface components and layout.
     */
    private void createUI() {
        setTitle("Add New Inventory");
        setSize(300, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new GridLayout(8, 2));

        add(new JLabel("Product ID:"));
        productIdField = new JTextField(10);
        add(productIdField);

        JLabel itemName = new JLabel("Product Name:");
        add(itemName);

        JLabel label = new JLabel("Current Quantity:");
        add(label);


        add(new JLabel("Add Quantity:"));
                quantityField = new JTextField();
```

```
                add(quantityField);



                productIdField.addFocusListener(new FocusAdapter() {
                    @Override
                    public void focusLost(FocusEvent e) {
                        try {
                            int productId =
Integer.parseInt(productIdField.getText());
                            Product product = Product.getProduct(productId);
                            if (product != null) {
                                label.setText("Current Quantity: " +
product.getQuantity());
                                itemName.setText("Product Name: " +
product.getName());
                            } else {
                                label.setText("Current Quantity: Not found");
                                itemName.setText("Product Name: Not found");
                            }
                        } catch (NumberFormatException ex) {
                            label.setText("Current Quantity: Invalid ID");
                            itemName.setText("Product Name: Invalid ID");
                        }
                    }
                });

        submitButton = new JButton("Submit");
        submitButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                updateInventory();

            }
        });
        add(submitButton);

        JButton backButton = new JButton("Back");
        backButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                dispose();
            }
        });
        add(backButton);

        setLocationRelativeTo(null); // Center on screen
    }

    /**
     *  Updates the inventory based on the input provided in the text fields.
     */
    private void updateInventory() {
```

```
        try {
            int productId = Integer.parseInt(productIdField.getText());
            int quantityToAdd = Integer.parseInt(quantityField.getText());
            Product product = Product.getProduct(productId);

            if (product != null) {
                product.setQuantity(product.getQuantity() + quantityToAdd);
                JOptionPane.showMessageDialog(this, "Inventory Updated");
            } else {
                JOptionPane.showMessageDialog(this, "Product not found");
            }
        } catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog(this, "Invalid input");
        }
    }

}
```

AddItemtoCartTest.Java

```java
import org.junit.Before;
import org.junit.Test;

import javax.swing.*;
import java.util.ArrayList;

import static org.junit.Assert.*;

public class AddItemtoCartTest {

    private CatalogView catalogView;
     Product sampleProduct;

    int shoppingCartSize =0;

    @Before
    public void setUp() {
        // Initialize CatalogView
        catalogView = new CatalogView();
        ArrayList<Object> shoppingCart = new ArrayList<>(); // Initialize the
shopping cart list
        // Sample product to be used in the tests
        sampleProduct = new Product(1010101, "Test", 99.99, 10,
"https://m.media-amazon.com/images/I/81zKcC5wJ6L._AC_UF1000,1000_QL80_.jpg",
"Sample.");
    }

    @Test
    public void whenProductAddedToCart_thenCartSizeShouldIncrease() {
        catalogView.addToCart(sampleProduct);

        assertEquals("Cart size should increase by 1", 1,
catalogView.getCartSize());


    }


}
```

AddNewItem.Java

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Random;

/**
 * This class represents a GUI form for adding new items to an inventory.
 * It allows the user to input details about a product, such as its name,
 * quantity, price, ID, image URL, and description. It also handles the
 * submission of this data to update or add a new product in the inventory.
 * @author freddy ingle
 * @author George martinez
 * @author Bryan Cooke
 */
public class AddNewItem extends JFrame {
    private JTextField productNameField, quantityField, priceField,
productIdField, productImageURLField, productDescribeField;
    private JButton submitButton;

    /**
     * Constructor for AddNewItem. It initializes the user interface.
     */
    public AddNewItem() {
        createUI();
    }

    /**
     * Creates the user interface for the AddNewItem window.
     * This method sets up the layout and components of the GUI.
     */
    private void createUI() {
        setTitle("Add New Item");
        setSize(300, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new GridLayout(7, 2));

        add(new JLabel("Product Name:"));
        productNameField = new JTextField(10);
        add(productNameField);

        add(new JLabel("Quantity:"));
        quantityField = new JTextField(10);
        add(quantityField);

        add(new JLabel("Price:"));
        priceField = new JTextField("00.00", 10); // Example current price
        add(priceField);

        Random randomNum = new Random();
```

```java
        int min = 100;
        int max = 1000;
        int randomNumber = randomNum.nextInt(max - min + 1) + min;

        add(new JLabel("Product ID:"));
        productIdField = new JTextField(String.valueOf(randomNumber), 10);
        add(productIdField);

        add(new JLabel("Product Image URL:"));
        productImageURLField = new JTextField(10);
        add(productImageURLField);

        add(new JLabel("Product Description:"));
        productDescribeField = new JTextField(10);
        add(productDescribeField);

        submitButton = new JButton("Submit");
        submitButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                updateInventory();
            }
        });
        add(submitButton);

        JButton backButton = new JButton("Back");
        backButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                dispose();
            }
        });
        add(backButton);

        setLocationRelativeTo(null); // Center on screen
    }

    /**
     * Updates the inventory with the information provided in the form.
     * If a product with the same ID already exists, it offers an option to
     * update the existing product. Otherwise, it adds a new product.
     */
    private void updateInventory() {
        try {
            String productName = productNameField.getText();
            int quantity = Integer.parseInt(quantityField.getText());
            double price = Double.parseDouble(priceField.getText());
            int productId = Integer.parseInt(productIdField.getText());
            String productImageURL = productImageURLField.getText(); //
Example image URL
            String productDescribe = productDescribeField.getText();
            // Check for negative quantity and price
            if (quantity < 0 || price < 0) {
```

```
                JOptionPane.showMessageDialog(this, "Please enter non-
negative values for quantity and price.");
                return;  // Exit the method to prevent further processing
            }
            // Check for duplicate product entry
            Product existingProduct = Product.getProduct(productId);
            if (existingProduct != null) {
                int option = JOptionPane.showConfirmDialog(null,
                        "A product with the same ID already exists. Do you
want to update the existing product?",
                        "Duplicate Product Entry",
JOptionPane.YES_NO_OPTION);
                if (option == JOptionPane.YES_OPTION) {
                    // Update the existing product
                    existingProduct.setName(productName);
                    existingProduct.setQuantity(quantity);
                    existingProduct.setPrice(price);
                    existingProduct.setImageURL(productImageURL);
                    existingProduct.setDescription(productDescribe);
                    JOptionPane.showMessageDialog(this, "Product updated
successfully!");
                } else {
                    JOptionPane.showMessageDialog(this, "Product not
updated.");
                }
            } else {
                // No duplicate found, add the new product to inventory
                Product.addProduct(new Product(productId, productName, price,
quantity, productImageURL, productDescribe));
                JOptionPane.showMessageDialog(this, "Product added
successfully!");
            }
        } catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog(this, "Please enter valid numbers
for quantity, price, and product ID.");
        }
    }
}
```

CatalogView.Java

```
import javax.swing.*;
import java.awt.*;
import java.net.URL;
import java.util.ArrayList;
import java.util.List;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
import java.awt.event.FocusAdapter;
import java.awt.event.FocusEvent;

/**
 * This class represents the main view of the product catalog in a GUI
application.
 * It displays products and allows users to add items to a shopping cart.
 * @author Freddy Ingle
 * @author George Martinez
 */
class CatalogView {
    private JFrame frame;
    private JPanel productPanel;
    private static JLabel cartItemCountLabel; //keep count of cart items
    private static List<Product> shoppingCart; //list of cart items


    /**
     * Gets the current size of the shopping cart.
     *
     * @return the number of items in the shopping cart.
     */
    public int getCartSize() { //use for junit test
        return shoppingCart.size();
    }

    /**
     * Constructs the CatalogView by initializing the GUI components.
     */
    public CatalogView() {
        frame = new JFrame("Product Catalog");
        productPanel = new JPanel(new GridLayout(0, 3, 10, 10)); // 3
columns, auto rows, 10px gaps

        // Cart Panel at the top-right
        JPanel cartPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
        cartItemCountLabel = new JLabel("Cart: 0 items"); // Initialize the
label with 0 items
        cartPanel.add(cartItemCountLabel);
        // "View Cart" button
        JButton viewCartButton = new JButton("View Cart");
        viewCartButton.addActionListener(e -> viewCart());
```

```java
        // Add the label and button to the cart panel
        cartPanel.add(cartItemCountLabel);
        cartPanel.add(viewCartButton);
        frame.add(cartPanel, BorderLayout.NORTH); // Add the cart panel to
the frame


        JScrollPane scrollPane = new JScrollPane(productPanel);
        frame.add(scrollPane, BorderLayout.CENTER);
        frame.setSize(1000, 600);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        shoppingCart = new ArrayList<>(); // Initialize the shopping cart
list
        refreshCartDisplay();
    }


    /**
     * Sets the visibility of the main frame.
     *
     * @param visible boolean flag to set the frame's visibility.
     */
    public void setVisible(boolean visible) {
        frame.setVisible(visible);
    }

      /**
     * Sets and displays the list of products in the catalog.
     *
     * @param products the list of products to be displayed.
     */
    public void setProducts(List<Product> products) {
        productPanel.removeAll(); // Remove all previous components
        for (Product product : products) {
            JPanel card = createProductCard(product);
            productPanel.add(card);
        }
        productPanel.revalidate();
        productPanel.repaint();
    }

    /**
     * Creates and returns a JPanel representing a single product card.
     *
     * @param product the product to create a card for.
     * @return a JPanel representing the product.
     */
    private JPanel createProductCard(Product product) {
        JPanel card = new JPanel(new BorderLayout(5, 5));
        JLabel nameLabel = new JLabel(product.getName(),
SwingConstants.CENTER);
```

```
        JLabel priceLabel = new JLabel(String.format("$%.2f",
product.getPrice()), SwingConstants.CENTER);

        // Adjust the description label to wrap text
        JLabel descriptionLabel = new JLabel("<html><body style='width:
100px'>" + product.getDescription() + "</body></html>");

        // Buttons Panel
        JPanel buttonsPanel = new JPanel(new FlowLayout(FlowLayout.CENTER,
10, 0));
        JButton btnAddToCart = new JButton("Add to Cart");
        JButton btnViewDetails = new JButton("View Details");

        btnAddToCart.addActionListener(e -> addToCart(product));
        btnViewDetails.addActionListener(e -> viewProductDetails(product));

        buttonsPanel.add(btnAddToCart);
        buttonsPanel.add(btnViewDetails);

        // Use a SwingWorker to load image in the background
        new SwingWorker<ImageIcon, Void>() {
            @Override
            protected ImageIcon doInBackground() throws Exception {
                URL imageUrl = new URL(product.getImageURL());
                ImageIcon imageIcon = new ImageIcon(imageUrl);
                Image image = imageIcon.getImage().getScaledInstance(100,
100, Image.SCALE_SMOOTH);
                return new ImageIcon(image);
            }

            @Override
            protected void done() {
                try {
                    JLabel imageLabel = new JLabel(get());
                    card.add(imageLabel, BorderLayout.WEST);
                    card.validate();
                    card.repaint();
                } catch (Exception e) {
                    e.printStackTrace();
                    // Handle the error here
                }
            }
        }.execute();

        card.add(nameLabel, BorderLayout.NORTH);
        card.add(priceLabel, BorderLayout.SOUTH);
        card.add(descriptionLabel, BorderLayout.CENTER);
        card.add(buttonsPanel, BorderLayout.PAGE_END);

        card.setBorder(BorderFactory.createLineBorder(Color.BLACK, 1));

        return card;
    }
```

```java
    /**
     * Adds a given product to the shopping cart and refreshes the cart
display.
     *
     * @param product the product to add to the shopping cart.
     */
    public void addToCart(Product product) {
        shoppingCart.add(product);
        refreshCartDisplay();
        JOptionPane.showMessageDialog(frame, product.getName() + " added to
cart!");
    }
    /**
     * Clears all items from the shopping cart and updates the display.
     */
    public void clearShoppingCart() {
        shoppingCart.clear();
        refreshCartDisplay();
    }


      /**
     * Displays details of a specific product in a dialog.
     *
     * @param product the product whose details are to be displayed.
     */
    private void viewProductDetails(Product product) {
        // Implementation of product details view
        if (product.isAvailable()) {
            String productDetails = product.getProductDetails();
            JOptionPane.showMessageDialog(frame, "Product Details:\n" +
productDetails);
        } else {
            JOptionPane.showMessageDialog(frame, "Product is out of stock or
unavailable.");
        }
    }

     /**
     * Displays the shopping cart view.
     */
    private void viewCart() {
        // Implementation to view the cart contents
                refreshCartDisplay();

        ShoppingCartView cartView = new ShoppingCartView(shoppingCart);
        cartView.displayCartItems();
    }

    /**
     * Displays the customer homepage view.
```

```java
     */
    public void displayCustomerHomepage() {
        refreshCartDisplay();
        SwingUtilities.invokeLater(() -> {
            CatalogMain catalogMain = new CatalogMain();
            catalogMain.display();
        });
    }

     /**
     * Updates the cart display label with the current number of items in the
cart.
     */
    public static void refreshCartDisplay() {

        cartItemCountLabel.setText("Cart: " + shoppingCart.size() + "
items");
        // Any other UI updates related to the cart
    }





}
/**
 * Controller for managing the product catalog view and interactions.
 */
class ProductCatalogController {
    private List<Product> products;
    private CatalogView view;

     /**
     * Constructor for the product catalog controller.
     *
     * @param view the CatalogView to control.
     */
    public ProductCatalogController(CatalogView view) {

        this.view = view;
        this.products = Product.getProducts();
        initController();
    }

    /**
     * Initializes the controller by setting up the view with data.
     */
    private void initController() {

        updateView();
    }
```

```java
    /**
     * Updates the view with the list of products.
     */
    private void updateView() {
        view.setProducts(products);
    }

    // methods for handling user actions
}
/**
 * Main class for launching the catalog view as part of a GUI application.
 */// CatalogMain (Main class for customer homepage)
class CatalogMain {
    private CatalogView view;

    /**
     * Constructs the main application view and controller.
     */
    public CatalogMain() {
        this.view = new CatalogView();
        new ProductCatalogController(view);
    }

     /**
     * Displays the main application view.
     */
    public void display() {
        view.setVisible(true);
    }
}
```

CheckoutView.Java

```java
import javax.swing.*;
import java.awt.*;
import java.util.List;

/**
 * This class represents the checkout view in a GUI application for a
shopping cart system.
 * It displays the items in the shopping cart and allows the user to complete
the purchase.
 *
 * @author Freddy Ingle
 * @author George Martinez
 *
 */
public class CheckoutView {
    private JFrame frame;
    private List<Product> shoppingCart; // The shopping cart items
    private double total; // Total price of items in the cart
    private CatalogView catalogView; // Reference to the CatalogView

    /**
     * Constructs the CheckoutView with a given shopping cart.
     *
     * @param shoppingCart The list of products in the shopping cart.
     */
    public CheckoutView(List<Product> shoppingCart) {
        this.shoppingCart = shoppingCart;
        this.catalogView = catalogView;
        frame = new JFrame("Checkout");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new BorderLayout());

        // Calculate the total
        total = shoppingCart.stream().mapToDouble(Product::getPrice).sum();

        // Create UI components
        createCartItemsPanel();
        createCheckoutInfoPanel();
        createBottomPanel();

        // Finalize and show the frame
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

    /**
     * Creates and adds the cart items panel to the frame.
     */
    private void createCartItemsPanel() {
```

```
        JPanel cartItemsPanel = new JPanel();
        cartItemsPanel.setLayout(new BoxLayout(cartItemsPanel,
BoxLayout.Y_AXIS));
        shoppingCart.forEach(product -> {
            JPanel productPanel = new JPanel(new
FlowLayout(FlowLayout.LEFT));
            JLabel productLabel = new JLabel(product.getName() + " - $" +
product.getPrice());
            productPanel.add(productLabel);
            cartItemsPanel.add(productPanel);
        });

        frame.add(new JScrollPane(cartItemsPanel), BorderLayout.CENTER);
    }

    /**
     * Creates and adds the checkout information panel to the frame.
     */
    private void createCheckoutInfoPanel() {
        JPanel infoPanel = new JPanel(new GridLayout(0, 2, 10, 10));

        // Add labels and text fields for checkout information
        infoPanel.add(new JLabel("Name on Card:"));
        infoPanel.add(new JTextField());
        infoPanel.add(new JLabel("Credit Card Number:"));
        infoPanel.add(new JTextField());
        infoPanel.add(new JLabel("Expiration Date:"));
        infoPanel.add(new JTextField());
        infoPanel.add(new JLabel("CVV:"));
        infoPanel.add(new JTextField());
        infoPanel.add(new JLabel("Shipping Address:"));
        infoPanel.add(new JTextField());

        frame.add(infoPanel, BorderLayout.NORTH);
    }

    /**
     * Creates and adds the bottom panel (including total and buttons) to the
frame.
     */
    private void createBottomPanel() {
        JPanel bottomPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
        JLabel totalLabel = new JLabel("Total: $" + total);
        JButton checkoutButton = new JButton("Complete Purchase");
        JButton backButton = new JButton("Back to Cart");

        checkoutButton.addActionListener(e -> completePurchase());
        backButton.addActionListener(e -> goBackToCart());

        bottomPanel.add(backButton);
        bottomPanel.add(totalLabel);
        bottomPanel.add(checkoutButton);
```

```
        frame.add(bottomPanel, BorderLayout.SOUTH);
    }


     /**
     * Handles the completion of the purchase, updating inventory and
clearing the cart.
     */
    private void completePurchase() {
        // Update Inventory
        shoppingCart.forEach(product -> {

Product.getProduct(product.getProductID()).setQuantity(Product.getProduct(pro
duct.getProductID()).getQuantity() - 1);
        });
        shoppingCart.clear();
        if (shoppingCart.size() == 0) {
            CatalogView.refreshCartDisplay();
        }

        JOptionPane.showMessageDialog(frame, "Thank you for your purchase!");

        frame.dispose();
    }


     /**
     * Handles the action to go back to the cart view.
     */
    private void goBackToCart() {
        // Placeholder for going back to the cart
        frame.dispose();
        // Potentially re-open the cart view or go back to the main shopping
view
    }

}
```

ImageTester.Java

```
import javax.swing.*;
import java.awt.*;
import java.io.IOException;
import java.net.URL;
import javax.imageio.ImageIO;


/*This class is purely for testing and debugging images */
public class ImageTester {

    JFrame frame;
    JLabel displayField;
    Image image;

    public ImageTester() {
        frame = new JFrame("Image Test");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        try {
            URL imageURL = new URL("https://m.media-
amazon.com/images/I/81zKcC5wJ6L._AC_UF1000,1000_QL80_.jpg");
            image = ImageIO.read(imageURL);

            // Resize the image
            int newWidth = 200;
            int newHeight = 200;
            Image resizedImage = image.getScaledInstance(newWidth, newHeight,
Image.SCALE_SMOOTH);

            displayField = new JLabel(new ImageIcon(resizedImage)); // Use
resizedImage here
            frame.add(displayField);
        } catch (IOException e) {
            e.printStackTrace(); // Print the exception stack trace
            System.out.println("Image not found");
        }
        frame.setSize(400, 400);
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        // Create an instance of ImageTester
        ImageTester i = new ImageTester();
    }
}
```

LoginApplication.Java

```
import javax.swing.*;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.util.List;

/**
 * Represents the user model in the application. It handles user
authentication
 * and security questions related to the user's role.
 *
 * @author Freddy Ingle
 * @author George Martinez
 */
// UserModel (Model)
class UserModel {
    private List<ChangeListener> listeners = new ArrayList<>();
    private boolean authenticated;
    private String role;


    /**
     * Authenticates a user based on the provided credentials and role.
     *
     * @param username The username input.
     * @param password The password input.
     * @param role     The role of the user (e.g., Customer, Seller).
     */
    public void authenticate(String username, String password, String role) {
        boolean oldAuthenticated = this.authenticated;
        this.authenticated = "admin".equalsIgnoreCase(username) &&
"12345".equalsIgnoreCase(password);

        // Check if authentication is successful
        if (authenticated) {
            this.role = role;

            // Check if security question is needed
            if ("Customer".equals(role)) {
                setSecurityQuestion("What is your favorite color?", "Blue");
            } else if ("Seller".equals(role)) {
                setSecurityQuestion("What is the name of your first pet?",
"Fluffy");
            }

            setSecurityQuestionRequired(true);
        }
```

```java
        // Notify listeners only if authentication status changes
        if (oldAuthenticated != this.authenticated) {
            notifyListeners();
        }
    }


    /**
     * Returns the authentication status of the user.
     *
     * @return true if the user is authenticated, false otherwise.
     */
    public boolean isAuthenticated() {
        return authenticated;
    }

      /**
     * Adds a ChangeListener to the list of listeners.
     *
     * @param listener The ChangeListener to be added.
     */
    public void addChangeListener(ChangeListener listener) {
        listeners.add(listener);
    }

    /**
     * Notifies all registered listeners of a change in the user's
authentication status.
     */
    private void notifyListeners() {
        ChangeEvent event = new ChangeEvent(this);
        for (ChangeListener listener : listeners) {
            listener.stateChanged(event);
        }
    }

     /**
     * Gets the role of the currently authenticated user.
     *
     * @return A string representing the user's role.
     */
    public String getRole() {
        return role;
    }

    private boolean securityQuestionRequired;
    private String securityQuestion;
    private String securityAnswer;

     /**
     * Sets the security question and answer for the user.
     *
```

```java
     * @param question The security question.
     * @param answer   The answer to the security question.
     */
    public void setSecurityQuestion(String question, String answer) {
        this.securityQuestion = question;
        this.securityAnswer = answer;
    }

     /**
     * Checks if answering a security question is required for the user.
     *
     * @return true if a security question is required, false otherwise.
     */
    public boolean isSecurityQuestionRequired() {
        return securityQuestionRequired;
    }

     /**
     * Sets whether answering a security question is required for the user.
     *
     * @param required true if a security question is required, false
otherwise.
     */
    public void setSecurityQuestionRequired(boolean required) {
        this.securityQuestionRequired = required;
    }


    /**
     * Gets the security question for the user.
     *
     * @return A string representing the security question.
     */
    public String getSecurityQuestion() {
        return securityQuestion;
    }

     /**
     * Validates the user's answer to the security question.
     *
     * @param answer The answer to validate.
     * @return true if the answer is correct, false otherwise.
     */
    public boolean validateSecurityAnswer(String answer) {
        return securityAnswer.equalsIgnoreCase(answer);
    }
}

/**
 * This class represents the login view in the application. It creates the
user
 * interface for user login, including fields for username, password, and
role selection.
```

```java
 *
 */
// LoginView (View)
class LoginView extends JFrame {
    private JLabel userLabel = new JLabel("Username");
    private JTextField userTextField = new JTextField();
    private JLabel passwordLabel = new JLabel("Password");
    private JPasswordField passwordField = new JPasswordField();
    private JButton loginButton = new JButton("Login");
    private JButton resetButton = new JButton("Reset");
    private JLabel messageLabel = new JLabel();
    private JLabel titleLabel = new JLabel("George's Shopping Cart App",
SwingConstants.CENTER);
    private JRadioButton sellerRadioButton = new JRadioButton("Seller");
    private JRadioButton customerRadioButton = new JRadioButton("Customer");
    private ButtonGroup roleButtonGroup = new ButtonGroup();

    /**
     * Constructs the login view with all UI components.
     */
    public LoginView() {
        setTitle("Login Form");
        setBounds(10, 10, 370, 600);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setResizable(false);

        Container container = getContentPane();
        container.setLayout(null);

        titleLabel.setFont(new Font("Serif", Font.BOLD, 20));
        titleLabel.setBounds(50, 30, 270, 30);
        container.add(titleLabel);

        sellerRadioButton.setBounds(150, 270, 80, 30);
        customerRadioButton.setBounds(230, 270, 100, 30);

        roleButtonGroup.add(sellerRadioButton);
        roleButtonGroup.add(customerRadioButton);

        container.add(sellerRadioButton);
        container.add(customerRadioButton);
        customerRadioButton.setSelected(true);

        userLabel.setBounds(50, 150, 100, 30);
        container.add(userLabel);

        userTextField.setBounds(150, 150, 150, 30);
        container.add(userTextField);

        passwordLabel.setBounds(50, 220, 100, 30);
        container.add(passwordLabel);

        passwordField.setBounds(150, 220, 150, 30);
```

```java
        container.add(passwordField);

        loginButton.setBounds(150, 300, 100, 30);
        container.add(loginButton);

        resetButton.setBounds(150, 340, 100, 30);
        container.add(resetButton);

        messageLabel.setBounds(50, 380, 270, 30);
        container.add(messageLabel);
    }

     /**
     * Gets the entered username.
     *
     * @return The text entered in the username field.
     */
    public String getUsername() {
        return userTextField.getText();
    }

     /**
     * Gets the entered password.
     *
     * @return The text entered in the password field.
     */
    public String getPassword() {
        return new String(passwordField.getPassword());
    }

    /**
     * Returns the login button.
     *
     * @return The login button.
     */
    public JButton getLoginButton() {
        return loginButton;
    }

     /**
     * Returns the reset button.
     *
     * @return The reset button.
     */
    public JButton getResetButton() {
        return resetButton;
    }

    /**
     * Displays an error message dialog.
     *
     * @param errorMessage The error message to be displayed.
     */
```

```java
    public void displayErrorMessage(String errorMessage) {
        JOptionPane.showMessageDialog(this, errorMessage);
    }

    /**
     * Displays a message in the view with the specified color.
     *
     * @param message The message to display.
     * @param color   The color of the message text.
     */
    public void displayMessage(String message, Color color) {
        messageLabel.setForeground(color);
        messageLabel.setText(message);
    }

     /**
     * Attaches the given action listener to the login and reset buttons.
     *
     * @param actionListener The ActionListener to attach.
     */
    public void attachController(ActionListener actionListener) {
        loginButton.addActionListener(actionListener);
        resetButton.addActionListener(actionListener);
    }

     /**
     * Updates the login status message in the view.
     *
     * @param isAuthenticated The authentication status of the user.
     */
    public void updateLoginStatus(boolean isAuthenticated) {
        String message = isAuthenticated ? "Login successful" : "Invalid
username or password";
        displayMessage(message, isAuthenticated ? Color.GREEN : Color.RED);
    }
    /**
     * Resets the fields in the login form.
     */
    public void resetFields() {
    userTextField.setText("");
    passwordField.setText("");
}

    /**
     * Gets the selected role from the radio buttons.
     *
     * @return The selected role, either "Seller" or "Customer".
     */
    public String getRole() {
        if (sellerRadioButton.isSelected()) {
            return "Seller";
        } else if (customerRadioButton.isSelected()) {
            return "Customer";
```

```java
        } else {
            return null;
        }
    }
}

/**
 * This class serves as the controller in the MVC pattern. It handles user
interactions
 * from the LoginView and updates the UserModel and other views accordingly.
 */
// LoginController (Controller)
class LoginController implements ActionListener, ChangeListener {
    private UserModel model;
    private LoginView view;
    private CatalogView catalogView;
    private SecurityQuestionView securityQuestionView;
    private boolean securityQuestionAnswered = false;


    /**
     * Constructs a LoginController with the specified model, view, and
catalog view.
     *
     * @param model       The UserModel to manage authentication logic.
     * @param view        The LoginView to interact with the user.
     * @param catalogView The CatalogView for displaying the product catalog.
     */
    public LoginController(UserModel model, LoginView view, CatalogView
catalogView) {
        this.model = model;
        this.view = view;
        this.catalogView = catalogView;
        this.securityQuestionView = new
SecurityQuestionView(model.getSecurityQuestion());
        this.view.attachController(this);
        this.model.addChangeListener(this);
        this.securityQuestionView.attachController(this); // Attach the
controller to handle the submit button
    }

     /**
     * Handles action events triggered in the LoginView.
     *
     * @param e The action event.
     */
    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == view.getLoginButton()) {
            String username = view.getUsername();
            String password = view.getPassword();
            String role = view.getRole();
            model.authenticate(username, password, role);
```

```
            if (model.isAuthenticated()) {
                // Check if security question is needed
                if (model.isSecurityQuestionRequired()) {
                    showSecurityQuestion();
                } else {
                    // Proceed with role-based redirection
                    handleAuthenticationSuccess();
                }
            }
        } else if (e.getSource() == view.getResetButton()) {
            view.resetFields();
        } else if (e.getSource() == securityQuestionView.getSubmitButton()) {
            handleSecurityQuestionSubmission();
        }
    }

    /**
     * Displays the security question view.
     */
    private void showSecurityQuestion() {
        // Set the security question text before showing the view

securityQuestionView.setSecurityQuestion(model.getSecurityQuestion());
        securityQuestionView.setVisible(true);
    }

     /**
     * Handles the submission of the security question.
     */
    private void handleSecurityQuestionSubmission() {
        String answer = securityQuestionView.getAnswer();
        if (model.validateSecurityAnswer(answer)) {
            securityQuestionAnswered = true;
            securityQuestionView.dispose();  // Close the security question
dialog
            redirectToRoleHomepage(model.getRole());
        } else {
            securityQuestionView.dispose();  // Close the security question
dialog
            view.displayErrorMessage("Incorrect answer to the security
question. Authentication failed.");
        }
    }


    /**
     * Manages the flow after successful authentication, including security
question handling.
     */
    private void handleAuthenticationSuccess() {
        String role = model.getRole();
        if ("Customer".equals(role) || "Seller".equals(role)) {
```

```
            // Display security question and wait for submission
            showSecurityQuestion();
        } else {
            // Handle other roles or scenarios
            redirectToRoleHomepage(role);
        }
    }

     /**
     * Redirects the user to the appropriate homepage based on their role.
     *
     * @param role The role of the user.
     */
    private void redirectToRoleHomepage(String role) {
        securityQuestionView.dispose();  // Close the security question
dialog

        if ("Customer".equals(role)) {
            // Redirect to customer homepage
            catalogView.displayCustomerHomepage();
        } else if ("Seller".equals(role)) {
            // Redirect to seller dashboard
            showSellerDashboard();
        }
    }

    /**
     * Responds to state changes in the UserModel.
     *
     * @param e The change event.
     */
    @Override
    public void stateChanged(ChangeEvent e) {
        boolean isAuthenticated = model.isAuthenticated();
        view.updateLoginStatus(isAuthenticated);

        if (isAuthenticated && securityQuestionAnswered) {
            String role = model.getRole();
            redirectToRoleHomepage(role);
            securityQuestionAnswered = false;  // Reset the flag for future
logins
        }
    }

     /**
     * Shows the seller dashboard.
     */
    private void showSellerDashboard() {
        SwingUtilities.invokeLater(() -> {
            SellerModel sellerModel = new SellerModel();
            SellerView sellerView = new SellerView();
            SellerController sellerController = new
SellerController(sellerModel, sellerView);
```

```java
            sellerView.display();
        });
    }
}
/**
 * This is the main class for the login application. It sets up the
application
 * by initializing the model, view, and controller, and then makes the view
visible.
 */
// Main Application
public class LoginApplication {
    /**
     * The main method that serves as the entry point for the application.
     *
     * @param args Command line arguments (not used).
     */
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            UserModel model = new UserModel();
            LoginView view = new LoginView();
            CatalogView catalogView = new CatalogView();  // Create an
instance of CatalogView
            new LoginController(model, view, catalogView);  // Pass
catalogView to the constructor
            view.setVisible(true);
        });
    }
}
```

Product.Java

```
import java.util.ArrayList;
import java.util.List;
/**
 * class for storing information about the products
 * @author freddy ingle
 * @author george martinez
 * @author Bryan Cooke
 * contributor: Sydney tivoli provided the image URLs
 */
public class Product {
    private int productID;
    private String name;
    private String description;
    private double price;
    private int quantity;
    private String imageURL;
    private static List<Product> products = new ArrayList<>();
    public Product(int productID, String name, double price, int quantity,
String imageURL, String description) {
        this.productID = productID;
        this.name = name;
        this.price = price;
        this.quantity = quantity;
        this.imageURL = imageURL;
        this.description = description;
    }

    /**
     *
     * @return product id
     */
    public int getProductID(){
        return productID;
    }
    /**
     *
     * @param productID
     * @return product
     */
    public static Product getProduct(int productID) {
        for (Product product : products) {
            if (product.getProductID() == productID) {
                return product;
            }
        }
        return null;
    }

    /**
     *
```

```java
     * @return price
     */
    public double getPrice(){
        return price;
    }
    /**
     *
     * @return name
     */
    public String getName(){
        return name;
    }
    /**
     *
     * @return desc
     */
    public String getDescription(){
        return description;
    }
    /**
     *
     * @return quantity
     */
    public int getQuantity(){
        return quantity;
    }

    /**
     *
     * @return imageURL
     */
    public String getImageURL(){
        return imageURL;
    }
    /**
     *
     * @param imageURL
     */
    public void setImageURL(String imageURL) {this.imageURL = imageURL;}
    /**
     *
     * @param productID
     */
    public void setProductID(int productID) {
        this.productID = productID;
    }
    /**
     *
     * @param name
     */
    public void setName(String name) {
        this.name = name;
    }
```

```java
    /**
     *
     * @param description
     */
    public void setDescription(String description) {
        this.description = description;
    }
    /**
     *
     * @param price
     */
    public void setPrice(double price) {
        this.price = price;
    }
    /**
     *
     * @param quantity
     */
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
    /**
     *
     * @return
     */
    public boolean isAvailable() {return quantity > 0;}
    /**
     *
     * @return
     */
    public String getProductDetails() {
        StringBuilder details = new StringBuilder();
        details.append("Name: ").append(name).append("\n");
        details.append("Price: $").append(String.format("%.2f",
price)).append("\n");
        // Check and include other product details if available
        if (description != null && !description.isEmpty()) {
            details.append("Description: ").append(description).append("\n");
        } else {
            details.append("Description: Information not available\n");
        }
        if (imageURL != null && !imageURL.isEmpty()) {
            details.append("Image URL: ").append(imageURL).append("\n");
        } else {
            details.append("Image URL: Information not available\n");
        }
        details.append("Quantity: ").append(quantity).append(" left in
stock\n");
        return details.toString();
    }
    static {
        initializeProducts();
    }
```

```java
    public static List<Product> getProducts() {
        return new ArrayList<>(products); // Return a copy to avoid
modification of the original list
    }
    // Static method to create a list of products
    public static void initializeProducts() {
        products.add(new Product(101, "Laptop", 999.99, 10, "https://m.media-
amazon.com/images/I/81zKcC5wJ6L._AC_UF1000,1000_QL80_.jpg", "A high-
performance laptop."));
        products.add(new Product(102, "Smartphone", 499.99, 15,
"https://store.storeimages.cdn-apple.com/4982/as-images.apple.com/is/iphone-
15-pro-finish-select-202309-6-7inch-naturaltitanium?wid=5120&hei=2880&fmt=p-
jpg&qlt=80&.v=1692845702708", "An innovative smartphone with the best
camera."));
        products.add(new Product(103, "DashCam", 99.99, 30, "https://m.media-
amazon.com/images/I/61-ouW+YFlL.jpg", "A Dashcam for your car."));
        products.add(new Product(3, "Bluetooth Headphones", 89.99, 25,
"https://m.media-amazon.com/images/I/61PAHKjnjCL.jpg", "Bluetooth headphones
with noise cancellation."));
        products.add(new Product(4, "Electric Toothbrush", 39.99, 40,
"https://m.media-amazon.com/images/I/71Ipo1ZbMFL._AC_UF1000,1000_QL80_.jpg",
"Rechargeable toothbrush with smart timer."));
        products.add(new Product(5, "Gaming Laptop", 1299.99, 30,
"https://m.media-amazon.com/images/I/712g5R0vkbL._AC_UF894,1000_QL80_.jpg",
"The latest gaming laptop with 24GB Ram and 1TB SSD."));
        products.add(new Product(6, "Smart Watch", 199.99, 20,
"https://m.media-amazon.com/images/I/71LfnkRgZ4L._AC_UF894,1000_QL80_.jpg",
"A smartwatch with health and fitness tracking."));
        products.add(new Product(7, "Tablet", 329.99, 15, "https://m.media-
amazon.com/images/I/61goypdjAYL._AC_UF1000,1000_QL80_.jpg", "A versatile
tablet perfect for work and play."));
        products.add(new Product(8, "Bluetooth Mouse", 24.99, 50,
"https://m.media-amazon.com/images/I/61Mk3YqYHpL.jpg", "A comfortable mouse
for everyday use."));
        products.add(new Product(9, "E-Reader", 129.99, 30,
"https://media.wired.com/photos/648ba6dff2de86183cf5b4d7/191:100/w_2580,c_lim
it/Kobo-Libra-2-Gear.jpg", "An e-reader with a paper-like display."));
        products.add(new Product(11, "Portable Speaker", 59.99, 35,
"https://m.media-amazon.com/images/I/51kQntfQvPL._AC_UF894,1000_QL80_.jpg",
"A portable speaker with excellent sound quality."));
        products.add(new Product(12, "Fitness Tracker", 49.99, 45,
"https://m.media-amazon.com/images/I/51JNsCR32BL._AC_UF1000,1000_QL80_.jpg",
"Track your daily activity and sleep."));
        products.add(new Product(13, "Espresso Machine", 249.99, 15,
"https://cb.scene7.com/is/image/Crate/BrevilleBrstExEsprsAVSSS21_VND/$web_pdp
_main_carousel_low$/210409125354/breville-barista-express-espresso-
machine.jpg", "Brew cafe-quality espresso at home with ease."));
        products.add(new Product(14, "Yoga Mat", 19.99, 30,
"https://images.lululemon.com/is/image/lululemon/LU9AKES_054348_4", "Eco-
friendly yoga mat with non-slip surface."));
        products.add(new Product(15, "LED Desk Lamp", 45.99, 20,
"https://m.media-amazon.com/images/I/519PNsCh2OL.jpg", "Adjustable desk lamp
with multiple brightness settings."));
```

```
        products.add(new Product(17, "Cookware Set", 129.99, 25,
"https://www.lecreuset.com/dw/image/v2/BDRT_PRD/on/demandware.static/-/Sites-
le-creuset-master/default/dw21ea7695/images/cat_cookware_sets/ECOM1901-new-
g1.jpg?sw=650&sh=650&sm=fit", "Stainless steel cookware set for all your
cooking needs."));
        products.add(new Product(18, "Sunglasses", 89.99, 50,
"https://ampere.shop/cdn/shop/files/Dusk-
Blackframewithdarktint_polarizedlenses_969c55e5-54b3-44bc-ad49-
3c0eac2e49f5_1100x.jpg?v=1700533974", "Stylish sunglasses with UV
protection."));
        products.add(new Product(19, "Backpack", 59.99, 35, "https://m.media-
amazon.com/images/I/81d1YjW6z-L._AC_UY1000_.jpg", "Durable backpack for
travel and everyday use."));
        products.add(new Product(20, "Leather Wallet", 49.99, 45,
"https://media.gucci.com/style/DarkGray_Center_0_0_490x490/1463502620/428726_
DJ20T_1000_001_080_0000_Light.jpg", "Elegant leather wallet with RFID
blocking."));
        products.add(new Product(22, "Noise-cancelling Earplugs", 25.99, 50,
"https://m.media-amazon.com/images/I/615FjNyNApL.jpg", "Reusable earplugs
with superior noise cancellation."));
    }
    public static void addProduct(Product product) {
        products.add(product);
    }


    @Override
    public String toString() {
        return
                "productID: " + productID +
                        ", name: '" + name + '\'' +
                        ", description: '" + description + '\'' +
                        ", price: $" + price +
                        ", quantity: " + quantity ;
    }
}
```

ProductTest.java

```
import org.junit.Test;
import static org.junit.Assert.*;

public class ProductTest {

    @Test
    public void productCreation_withValidDetails_shouldSetCorrectAttributes()
{
        // Create a product with the full constructor details
        Product product = new Product(1010101, "Test", 99.99, 10,
                                      "https://m.media-
amazon.com/images/I/81zKcC5wJ6L._AC_UF1000,1000_QL80_.jpg",
                                      "Sample.");

        // Assert that each attribute is set correctly
        assertEquals("Product ID should be 1010101", 1010101,
product.getProductID());
        assertEquals("Product name should be 'Test'", "Test",
product.getName());
        assertEquals("Product price should be 99.99", 99.99,
product.getPrice(), 0.001);
        assertEquals("Product quantity should be 10", 10,
product.getQuantity());
        assertEquals("Product imageURL should match",
                     "https://m.media-
amazon.com/images/I/81zKcC5wJ6L._AC_UF1000,1000_QL80_.jpg",
                     product.getImageURL());
        assertEquals("Product description should be 'Sample.'", "Sample.",
product.getDescription());
    }

}
```

SecurityQuestionView.Java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * Represents a view for displaying and answering a security question.
 * This class creates the user interface for the security question prompt.
 *
 * @author George Martinez
 */
public class SecurityQuestionView extends JFrame {
    private JLabel securityQuestionLabel = new JLabel("Security Question");
    private JTextField answerTextField = new JTextField();
    private JButton submitButton = new JButton("Submit");

     /**
     * Constructs the SecurityQuestionView with the specified security
question.
     *
     * @param securityQuestion The security question to be displayed.
     */
    public SecurityQuestionView(String securityQuestion) {
        setTitle("Security Question");
        setBounds(10, 10, 370, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setResizable(false);

        Container container = getContentPane();
        container.setLayout(null);

        securityQuestionLabel.setBounds(50, 30, 270, 30);
        container.add(securityQuestionLabel);

        JLabel questionLabel = new JLabel(securityQuestion);  // Set the
security question text
        questionLabel.setBounds(50, 60, 270, 30);
        container.add(questionLabel);

        answerTextField.setBounds(50, 90, 270, 30);
        container.add(answerTextField);

        submitButton.setBounds(150, 130, 100, 30);
        container.add(submitButton);
    }


     /**
     * Returns the submit button.
     *
```

```java
     * @return The submit button.
     */
    public JButton getSubmitButton() {
        return submitButton;
    }

    /**
     * Gets the user's answer to the security question.
     *
     * @return The answer text entered by the user.
     */
    public String getAnswer() {
        return answerTextField.getText();
    }

     /**
     * Sets the security question text in the view.
     *
     * @param question The security question to display.
     */
    public void setSecurityQuestion(String question) {
        securityQuestionLabel.setText(question);
    }

    /**
     * Attaches an action listener to the submit button.
     *
     * @param actionListener The ActionListener to be added.
     */
    public void attachController(ActionListener actionListener) {
        submitButton.addActionListener(actionListener);
    }
}
```

SellerDashboard.Java

```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.List;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.util.Optional;

/**
 * This class represents the model for a seller in the application.
 * It handles the storage and management of the seller's total sales and
inventory count.
 *
 * Authors are written for each class in this file
 * @author freddy ingle
 * @author george martinez
 */
// Model
class SellerModel {
    private int totalSales;
    private int inventoryCount;

    /**
     * Gets the total sales of the seller.
     *
     * @return The total sales amount.
     */
    public int getTotalSales() {
        return totalSales;
    }

    /**
     *  set total sales
     * @param totalSales
     */
    public void setTotalSales(int totalSales) {
        this.totalSales = totalSales;
    }

    /**
     *
     * @return inventoryCount
     */
    public int getInventoryCount() {
        return inventoryCount;
    }

    /**
     *  set inventory count
```

```java
     * @param inventoryCount
     */
    public void setInventoryCount(int inventoryCount) {
        this.inventoryCount = inventoryCount;
    }
}

/**
 * This class represents the view for a seller in the application.
 * It creates and displays the seller dashboard interface including buttons
for various actions.
 * @author george martinez
 * @author freddy ingle
 * @author Bryan Cooke
 */
// View
class SellerView {
    private JFrame frame;
    private JButton btnTotalSales;
    private JButton btnInventory;
    private JButton btnAddNewItem;
    private JLabel titleLabel;
    private JButton btnAddNewInventory;

    /**
     * Constructs the SellerView with all its UI components.
     */
    public SellerView() {
        frame = new JFrame("Seller Dashboard");
        frame.setSize(600, 400);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new BorderLayout());

        titleLabel = new JLabel("Seller Dashboard", JLabel.CENTER);
        frame.add(titleLabel, BorderLayout.NORTH);
        titleLabel.setFont(new Font("Arial", Font.BOLD, 24));

        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new BoxLayout(buttonPanel, BoxLayout.Y_AXIS));

        btnTotalSales = new JButton("Check Total Sales");
        btnInventory = new JButton("View/Edit Inventory");
        btnAddNewItem = new JButton("Add New Item");
        //btnAddNewInventory = new JButton("Add New Inventory");


        buttonPanel.add(btnTotalSales);
        buttonPanel.add(btnInventory);
        buttonPanel.add(btnAddNewItem);
        //buttonPanel.add(btnAddNewInventory);

        frame.add(buttonPanel, BorderLayout.CENTER);
```

```java
        // Center align the buttons
        btnTotalSales.setAlignmentX(Component.CENTER_ALIGNMENT);
        btnInventory.setAlignmentX(Component.CENTER_ALIGNMENT);
        btnAddNewItem.setAlignmentX(Component.CENTER_ALIGNMENT);
       // btnAddNewInventory.setAlignmentX(Component.CENTER_ALIGNMENT);

        // Add buttons to the button panel with alignment and padding
        buttonPanel.add(Box.createVerticalGlue());
        buttonPanel.add(btnTotalSales);
        buttonPanel.add(btnInventory);
        buttonPanel.add(btnAddNewItem);
        //buttonPanel.add(btnAddNewInventory);
        buttonPanel.add(Box.createVerticalGlue());

        frame.add(buttonPanel, BorderLayout.CENTER);
    }

    /**
     *
     * @return frame
     */
    public JFrame getFrame() {
        return frame;
    }

    /**
     * set the frame
     */
    public void display() {
        frame.setVisible(true);
    }

    /**
     *
     * @return total sales button
     */
    public JButton getBtnTotalSales() {
        return btnTotalSales;
    }

    /**
     *
     * @return inventory button
     */
    public JButton getBtnInventory() {
        return btnInventory;
    }

    /**
     *
     * @return add new item button
     */
    public JButton getBtnAddNewItem() {
```

```java
        return btnAddNewItem;
    }

    /*public JButton getBtnAddNewInventory() {
        return btnAddNewInventory;
    }*/
}

// InventoryModel
/**
 * Model in MVC pattern for inventory
 * @author freddy ingle
 * @author george martinez
 */
class InventoryModel {
    private List<Product> products;

    public InventoryModel(List<Product> products) {
        this.products = products;
    }

    public List<Product> getProducts() {
        return products;
    }

    public boolean isInventoryEmpty() {return products.isEmpty();}

}

// InventoryView
/**
 * View in MVC pattern for inventory
 * It creates and displays the interface for viewing and editing the
inventory.
 * @author george martinez
 * @author freddy ingle
 */
class InventoryView {
    private JFrame frame;
    private JTextArea inventoryTextArea;
    private JButton btnBack;
    private JButton btnEdit;

    /**
     * Constructs the InventoryView with the specified list of products.
     *
     * @param products The list of products to display in the inventory view.
     */
    public InventoryView(List<Product> products) {
        frame = new JFrame("View/Edit Inventory");
        frame.setSize(600, 400);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new BorderLayout());
```

```java
        if (products.isEmpty()) {
            showMessage("No products are currently available in the
inventory.");
            return; // Don't create other components if the inventory is
empty
        }

        inventoryTextArea = new JTextArea();
        inventoryTextArea.setEditable(false);
        JScrollPane scrollPane = new JScrollPane(inventoryTextArea);
        frame.add(scrollPane, BorderLayout.CENTER);

        btnBack = new JButton("Back to Seller Page");
        frame.add(btnBack, BorderLayout.SOUTH);

        btnBack.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                // Handle back button click
                frame.dispose(); // Close the inventory view
            }
        });

        btnEdit = new JButton("Edit Selected Item");
        frame.add(btnEdit, BorderLayout.SOUTH);

        btnEdit.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                // Handle edit button click
                handleEdit();
            }
        });
    }

    /**
     *
     * @return frame
     */
    public JFrame getFrame() {
        return frame;
    }

    /**
     *
     * @return inventory text area
     */
    public JTextArea getInventoryTextArea() {
        return inventoryTextArea;
    }

    /**
```

```
     * set the frame
     */
    public void display() {
        frame.setVisible(true);
    }

    /**
     *
     * @return Edit button
     */
    public JButton getBtnEdit() {
        return btnEdit;
    }

    /**
     * show no product in inventory message
     */
    public void showEmptyInventoryMessage() {
        showMessage("No products are currently available in the inventory.");
    }

    /**
     * show error message : data retrieval error
     */
    public void showDataRetrievalError() {
        showMessage("Error fetching inventory data. Please try again
later.");
    }

    /**
     *
     * @param message
     */
    private void showMessage(String message) {
        JOptionPane.showMessageDialog(frame, message, "Inventory
Information", JOptionPane.INFORMATION_MESSAGE);
    }

    private void handleEdit() {
    }

    /**
     * edits the selected inventory with new user changes in string format
     * @param products
     */
    public void updateInventoryText(List<Product> products) {
        StringBuilder inventoryText = new StringBuilder();
        inventoryText.append(String.format("%-10s %-20s %-15s %-10s\n",
"Product ID", "Product Name", "Quantity", "Price"));

        for (Product product : products) {
            inventoryText.append(String.format("%-10d %-20s %-15d $%-
10.2f\n",
```

```
                    product.getProductID(), product.getName(),
product.getQuantity(), product.getPrice()));
        }

        inventoryTextArea.setText(inventoryText.toString());
    }

    /**
     * Shows a confirmation dialog for duplicate product entries.
     * @return true if the user chooses to update the existing product, false
otherwise.
     */
    public boolean showDuplicateProductConfirmation() {
        int option = JOptionPane.showConfirmDialog(frame,
                "A product with the same ID already exists. Do you want to
update the existing product?",
                "Duplicate Product Entry", JOptionPane.YES_NO_OPTION);

        return option == JOptionPane.YES_OPTION;
    }
}

// InventoryController
/**
 * This class serves as the controller for the InventoryView.
 * It handles user interactions from the InventoryView and updates the
InventoryModel accordingly.
 * @author george martinez
 */
class InventoryController {
    private InventoryModel model;
    private InventoryView view;

    /**
     * Constructs an InventoryController with the specified model and view.
     *
     * @param model The InventoryModel to manage inventory data.
     * @param view  The InventoryView to interact with the user.
     */
    public InventoryController(InventoryModel model, InventoryView view) {
        this.model = model;
        this.view = view;

        view.getBtnEdit().addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                // Handle edit button click
                handleEdit();
            }
        });

        if (model.isInventoryEmpty()) {
            view.showEmptyInventoryMessage();
```

```
            return; // Stop further initialization if the inventory is empty
        }

        // Update the inventory text when the view is created
        view.updateInventoryText(model.getProducts());

        // Add a listener to handle the selection of items in the JTextArea
        view.getInventoryTextArea().addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                handleSelection();
            }
        });
    }

     /**
     * Handles errors during data retrieval by showing an error message.
     */
    public void handleDataRetrievalError() {
        view.showDataRetrievalError();
    }


    private void handleEdit() {
        // Check if an item is selected
        String selectedText = view.getInventoryTextArea().getSelectedText();
        if (selectedText == null || selectedText.isEmpty()) {
            JOptionPane.showMessageDialog(view.getFrame(), "Please select an
item to edit.",
                    "Edit Item", JOptionPane.WARNING_MESSAGE);
            return;
        }

        // Retrieve the selected item based on the selected text
        String[] lines = view.getInventoryTextArea().getText().split("\n");

        // Find the index of the selected text in the lines
        int selectedRow = -1;
        for (int i = 0; i < lines.length; i++) {
            if (lines[i].contains(selectedText)) {
                selectedRow = i;
                break;
            }
        }

        // Check if selectedRow is valid
        if (selectedRow >= 0 && selectedRow < lines.length) {
            String selectedLine = lines[selectedRow];
            int selectedProductId =
Integer.parseInt(selectedLine.split("\\s+")[0]);

            // Find the corresponding product based on the selected product
ID
```

```
            Optional<Product> selectedProduct = model.getProducts().stream()
                    .filter(product -> product.getProductID() ==
selectedProductId)
                    .findFirst();

            if (selectedProduct.isPresent()) {
                // Create a dialog for editing
                JDialog editDialog = new JDialog(view.getFrame(), "Edit
Product", true);
                editDialog.setSize(400, 300);
                editDialog.setLayout(new GridLayout(6, 2));

                // Create labels and text fields for each property
                JLabel idLabel = new JLabel("Product ID:");
                JTextField idField = new
JTextField(String.valueOf(selectedProduct.get().getProductID()));
                JLabel nameLabel = new JLabel("Product Name:");
                JTextField nameField = new
JTextField(selectedProduct.get().getName());
                JLabel descriptionLabel = new JLabel("Description:");
                JTextField descriptionField = new
JTextField(selectedProduct.get().getDescription());
                JLabel priceLabel = new JLabel("Price:");
                JTextField priceField = new
JTextField(String.valueOf(selectedProduct.get().getPrice()));
                JLabel quantityLabel = new JLabel("Quantity:");
                JTextField quantityField = new
JTextField(String.valueOf(selectedProduct.get().getQuantity()));

                // Create buttons for saving changes and canceling
                JButton saveButton = new JButton("Save Changes");
                JButton cancelButton = new JButton("Cancel");

                // Add action listener to the save button
                saveButton.addActionListener(new ActionListener() {
                    @Override
                    public void actionPerformed(ActionEvent e) {
                        // Validate non-negative quantity and price
                        if (validateNonNegativeValues(quantityField,
priceField)) {
                            // Update the selected product with the new
values

selectedProduct.get().setProductID(Integer.parseInt(idField.getText()));

selectedProduct.get().setName(nameField.getText());

selectedProduct.get().setDescription(descriptionField.getText());

selectedProduct.get().setPrice(Double.parseDouble(priceField.getText()));

selectedProduct.get().setQuantity(Integer.parseInt(quantityField.getText()));
```

```
                            // Update the product in the list
                            updateProductInList(selectedProductId,
selectedProduct.get());

                            // Update the display in the inventory view
                            view.updateInventoryText(model.getProducts());
                        }

                        // Close the dialog
                        editDialog.dispose();
                    }
                });

                // Add action listener to the cancel button
                cancelButton.addActionListener(new ActionListener() {
                    @Override
                    public void actionPerformed(ActionEvent e) {
                        // Close the dialog without saving changes
                        editDialog.dispose();
                    }
                });

                // Add components to the dialog
                editDialog.add(idLabel);
                editDialog.add(idField);
                editDialog.add(nameLabel);
                editDialog.add(nameField);
                editDialog.add(descriptionLabel);
                editDialog.add(descriptionField);
                editDialog.add(priceLabel);
                editDialog.add(priceField);
                editDialog.add(quantityLabel);
                editDialog.add(quantityField);
                editDialog.add(saveButton);
                editDialog.add(cancelButton);

                // Set the layout and make the dialog visible
                editDialog.setLayout(new GridLayout(6, 2));
                editDialog.setVisible(true);
            }
        } else {
            // Handle the case where selectedRow is invalid
            JOptionPane.showMessageDialog(view.getFrame(), "Invalid
selection. Please try again.",
                    "Edit Item", JOptionPane.ERROR_MESSAGE);
        }
    }

    // Helper method to validate non-negative quantity and price
    private boolean validateNonNegativeValues(JTextField quantityField,
JTextField priceField) {
        try {
            int quantity = Integer.parseInt(quantityField.getText());
```

```
            double price = Double.parseDouble(priceField.getText());
            if (quantity < 0 || price < 0) {
                JOptionPane.showMessageDialog(view.getFrame(), "Please enter
non-negative values for quantity and price.",
                        "Validation Error", JOptionPane.ERROR_MESSAGE);
                return false;
            }
            return true;
        } catch (NumberFormatException ex) {
            JOptionPane.showMessageDialog(view.getFrame(), "Please enter
valid numbers for quantity and price.",
                    "Validation Error", JOptionPane.ERROR_MESSAGE);
            return false;
        }
    }

    /**
     * Updates a product in the model's product list.
     *
     * @param oldProductId   The old product ID.
     * @param updatedProduct The updated product object.
     */
    private void updateProductInList(int oldProductId, Product
updatedProduct) {
        List<Product> productList = model.getProducts();

        for (int i = 0; i < productList.size(); i++) {
            if (productList.get(i).getProductID() == oldProductId) {
                productList.set(i, updatedProduct);
                break;
            }
        }
    }


    /**
     * Checks for duplicate product ID in the inventory.
     *
     * @param editedProductId   The new product ID.
     * @param originalProductId The original product ID.
     * @return true if a duplicate product ID exists, false otherwise.
     */
    private boolean checkForDuplicateProductId(int editedProductId, int
originalProductId) {
        // Check if a product with the same ID already exists, excluding the
original product ID
        boolean isDuplicate = model.getProducts().stream()
                .anyMatch(product -> product.getProductID() ==
editedProductId && product.getProductID() != originalProductId);

        if (isDuplicate) {
            JOptionPane.showMessageDialog(view.getFrame(), "A product with
the same ID already exists. Please choose a different ID.",
```

```
                    "Duplicate Product ID", JOptionPane.ERROR_MESSAGE);
        }

        return isDuplicate;
    }


    // Add a method to handle the selection of items in the JTextArea
    private void handleSelection() {
        // You can add logic to handle selection if needed
    }

    /**
     * Handles updating the view after editing a product.
     *
     * @param product The edited product.
     */
    private void handleEditProduct(Product product) {
        // Update the display in the inventory view
        view.updateInventoryText(model.getProducts());
    }
}

// SellerController (Controller)
/**
 * This class serves as the controller for the SellerView.
 * It handles user interactions from the SellerView and updates the
SellerModel accordingly.
 * @author bryan cooke
 * @author george martinez
 * @author freddy ingle
 */
class SellerController {
    private SellerModel model;
    private SellerView view;

     /**
     * Constructs a SellerController with the specified model and view.
     *
     * @param model The SellerModel to manage seller data.
     * @param view  The SellerView to interact with the user.
     */
    public SellerController(SellerModel model, SellerView view) {
        this.model = model;
        this.view = view;

        view.getBtnTotalSales().addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                handleTotalSales();
            }
        });
```

```java
        view.getBtnInventory().addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                handleInventory();
            }
        });


        view.getBtnAddNewItem().addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                handleAddNewItem();
            }
        });

        /*view.getBtnAddNewInventory().addActionListener(new ActionListener()
{
            @Override
            public void actionPerformed(ActionEvent e) {
                handleAddNewInventory();
            }
        });*/

        // Initialize model data (for demonstration purposes)
        model.setTotalSales(500);
        model.setInventoryCount(50);
    }

    /**
     * Handles the display of total sales metrics.
     */
    private void handleTotalSales() {
        // Logic for total sales
        // Retrieve data from the model
        int totalSales = model.getTotalSales();
        int inventoryCount = model.getInventoryCount();

        // Calculate other metrics
        double netSales = calculateNetSales(totalSales);
        double shippingCosts = calculateShippingCosts();
        double productionCosts = calculateProductionCosts();
        double sellerFees = calculateSellerFees();
        double profit = calculateProfit(netSales, shippingCosts,
productionCosts, sellerFees);

        // Display the sales metrics in a dialog
        String message = "Total Sales: " + totalSales + "\n" +
                "Net Sales: $" + netSales + "\n" +
                "Shipping Costs: $" + shippingCosts + "\n" +
                "Production Costs: $" + productionCosts + "\n" +
                "Seller Fees: $" + sellerFees + "\n" +
                "Profit: $" + profit;
```

```java
        JOptionPane.showMessageDialog(view.getFrame(), message, "Total Sales
Information", JOptionPane.INFORMATION_MESSAGE);
    }


    // Add logic for calculating the sales metrics
    /**
     * Calculates the net sales.
     *
     * @param totalSales The total sales amount.
     * @return The net sales amount.
     */
    private double calculateNetSales(int totalSales) {
        return totalSales * 10; // For demonstration purposes
    }

    /**
     * calculates shipping costs
     *
     * @return shipping cost amount
     */
    private double calculateShippingCosts() {
        // Add logic
        return 200.0; // For demonstration purposes
    }

    /**
     *
     * @return production costs amount
     */
    private double calculateProductionCosts() {
        // Add logic
        return 3000.0; // For demonstration purposes
    }

    /**
     *
     * @return seller fees amount
     */
    private double calculateSellerFees() {
        // Add logic
        return 500.0; // For demonstration purposes
    }

    /**
     *
     * @param netSales
     * @param shippingCosts
     * @param productionCosts
     * @param sellerFees
     * @return Profit
     */
```

```java
    private double calculateProfit(double netSales, double shippingCosts,
double productionCosts, double sellerFees) {
        // Add logic
        return netSales - (shippingCosts + productionCosts + sellerFees);
    }

    /**
     * Handles the display and management of inventory.
     */
    private void handleInventory() {
        // Get or initialize your list of products
        List<Product> products = Product.getProducts();

        if (products.isEmpty()) {
            // Use the InventoryView to show the empty inventory message
            InventoryView inventoryView = new InventoryView(products);
            inventoryView.showEmptyInventoryMessage();
            return; // Stop further initialization if the inventory is empty
        }

        // Display the inventory view
        InventoryModel inventoryModel = new InventoryModel(products);
        InventoryView inventoryView = new InventoryView(products);
        InventoryController inventoryController = new
InventoryController(inventoryModel, inventoryView);

        // Add logic to update the inventoryTextArea in the view
        inventoryView.updateInventoryText(products);
        inventoryView.display();
    }


    /**
     * display new item
     */
    private void handleAddNewItem() {
        new AddNewItem().setVisible(true);
    }


    /**
     * display the seller Dashboard
     */
    public void displaySellerDashboard() {view.display();}


    /**
     * display the new inventory
     */
    private void handleAddNewInventory() {
        new AddInventory().setVisible(true);
    }
```

```java
}



/**
 * The main class for the application. It sets up the seller dashboard by
initializing
 * the model, view, and controller, and then makes the view visible.
 * @author freddy ingle
 * @author george martinez
 */
class Main {
    public static void main(String[] args) {
        List<Product> products = Product.getProducts();

        if (products.isEmpty()) {
            // Handle empty inventory condition if needed
            System.out.println("Empty inventory. Handle accordingly.");
            return;
        }

        SellerModel model = new SellerModel();
        SellerView view = new SellerView();
        SellerController controller = new SellerController(model, view);

        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                view.display();
            }
        });
    }
}
```

ShoppingCartView.Java

```
import javax.swing.*;
import java.awt.*;
import java.util.ArrayList;
import java.util.List;

/**
 * Represents the shopping cart view in the application.
 * This class creates the user interface for displaying the shopping cart and
initiating the checkout process.
 * @author freddy ingle
 */
class ShoppingCartView {
    private List<Product> shoppingCart;
    private JFrame frame;
    // Checkout button at the bottom of the dialog
    private JButton checkoutButton = new JButton("Checkout");


      /**
     * Constructs the ShoppingCartView with the specified list of products in
the shopping cart.
     *
     * @param shoppingCart The list of products in the shopping cart.
     */
    public ShoppingCartView(List<Product> shoppingCart) {
        this.shoppingCart = shoppingCart;
        this.frame = new JFrame("Shopping Cart");
        this.checkoutButton = new JButton("Checkout");

        // Set up the frame and add components...
        checkoutButton.addActionListener(e -> openCheckoutView());
        frame.add(checkoutButton, BorderLayout.SOUTH); // Or wherever it
needs to be placed
        frame.pack();
        frame.setVisible(true);
    }

     /**
     * Displays the items in the shopping cart in a dialog.
     */
     public void displayCartItems() {
        JDialog cartDialog = new JDialog();
        cartDialog.setTitle("Shopping Cart");
        cartDialog.setLayout(new BorderLayout());

        StringBuilder sb = new StringBuilder("<html>");
        for (Product product : shoppingCart) {
            sb.append(product.getName()).append(" -
$").append(product.getPrice()).append("<br>");
        }
```

```
        sb.append("</html>");

        JLabel itemsLabel = new JLabel(sb.toString());
        JScrollPane scrollPane = new JScrollPane(itemsLabel); // In case of
many items


        cartDialog.add(scrollPane, BorderLayout.CENTER);
        cartDialog.add(checkoutButton, BorderLayout.SOUTH);

        cartDialog.setSize(300, 400);
        cartDialog.setLocationRelativeTo(null); // Center the dialog
        cartDialog.setVisible(true);
    }
     /**
     * Opens the checkout view and disposes the current shopping cart frame.
     */
    private void openCheckoutView() {
        new CheckoutView(shoppingCart); // Create and display the checkout
view

        frame.dispose(); // Close or hide the shopping cart window
    }


}
```

UserLoginTest.Java

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class UserLogInTest {
    private UserModel model;

    @Before
    public void setUp() {
        model = new UserModel();
    }

    @Test
    public void testAuthenticateSuccess() {
        // Given
        String username = "admin";
        String password = "12345";
        String role = "Customer";

        // When
        model.authenticate(username, password, role);

        // Then
        assertTrue("User should be authenticated with correct credentials",
model.isAuthenticated());
    }

    @Test
    public void testAuthenticateFailure() {
        // Given
        String username = "wrongUser";
        String password = "wrongPass";
        String role = "Customer";

        // When
        model.authenticate(username, password, role);

        // Then
        assertFalse("User should not be authenticated with incorrect
credentials", model.isAuthenticated());
    }
}
```