Group Members:

Brandon Christian
Ethan Curtis
Ismail Ali
James Mundy

Summary:

Our final project is a shopping cart simulation app called “CartSafari.” This app is coded in Java, using Swing for the GUI. This app serves as a marketplace for customers as well as sellers and provides a simple way for sellers to sell their products and customers to buy them. Users can add and remove products to their cart, where the total price is calculated and the items are delivered straight to the customer's address. Sellers are able to quickly create and remove listings as well as edit their available stock all from the app.

Requirements:

CartSafari is coded in Java using Swing for the GUI. It is intended to be used on the Windows operating system but it has been tested on MacOS and works on there as well.

Project Demo:

https://www.youtube.com/watch?v=QmprsEgpQUo

CRC Cards:

User	
• Responsibilities: • -Store user credentials (username/password)	• Collaborators: • None

Customer	
• Responsibilities: • Access product information • Manage shopping cart • Initiate checkout process	• Collaborators: • Product • Shopping Cart • Payment Gateway

Seller	
• Responsibilities: • Manage inventory • View financial reports	• Collaborators: • Product • Inventory • Financial Reports

Product	
• Responsibilities: • Store product details	• Collaborators: • Customer • Seller

Shopping Cart	
• Responsibilities: • Store selected items • Calculate total price	• Collaborators: • Seller • Product

Inventory	
• Responsibilities • Store available products • Update product details	• Collaborators: • Seller • Product

Payment Gateway	
• Responsibilities: • Process payments securely	• Collaborators: • Customer

Email Service	
• Responsibilities: • Send email confirmations with transaction details	• Collaborators: • Customer

Financial Reports	
• Responsibilities: • Generate financial summaries and breakdowns	• Collaborations: • Seller

Source Code:

MainWindow.java:

```
package frontend;

import javax.swing.*;
import frontend.Login.*;
```

```java
/**
 * Represents the main window of the application and implements the IWrapper interface.
 */
public class MainWindow implements IWrapper {
	private JFrame windowFrame;
	private Object model, controller;
	private JComponent view;

	/**
    * Singleton instance of the MainWindow class.
    */
	public static MainWindow Instance;

	/**
    * Private constructor to enforce the Singleton pattern and initialize the main window.
    */
	private MainWindow() {
		windowFrame = new JFrame("CartSafari");
		windowFrame.setSize(1200, 720);
		windowFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		windowFrame.setVisible(true);

		// Add login window
		LoginModel loginModel = new LoginModel();
		LoginView loginView = new LoginView();
		LoginController loginController = new LoginController(loginModel, loginView);

		changeMVC(loginModel, loginView, loginController);
	}

	/**
    * Initializes the application's main window and starts the application.
    *
    * @param args The command-line arguments.
    */
	public static void main(String[] args) {
		Instance = new MainWindow();
	}

	/**
    * Changes the current Model-View-Controller (MVC) to another set of MVC components.
    * Removes the old MVC from memory (unless stored elsewhere).
    *
```

```
     * @param model      The new Model component.
     * @param view       The new View component (inherited from JComponent).
     * @param controller The new Controller component.
     */
    @Override
        public void changeMVC(Object model, JComponent view, Object controller) {
                this.model = model;
                this.controller = controller;

                if (this.view != null)
                        windowFrame.remove(this.view);
                this.view = view;
                windowFrame.add(view);
                windowFrame.revalidate();
                windowFrame.repaint();
        }
}
```

IWrapper.java:

```
package frontend;

import javax.swing.JComponent;

/**
 * Interface representing a wrapper for changing the Model-View-Controller (MVC) components.
 */
public interface IWrapper {
        /**
     * Changes the Model-View-Controller (MVC) components.
     *
     * @param model      The object representing the model component.
     * @param view       The JComponent representing the view component.
     * @param controller The object representing the controller component.
     */
        public void changeMVC(Object model, JComponent view, Object controller);
}
```

FinancialReportsScreenController.java:

```
package frontend.FinancialReportsScreen;

import frontend.LogoutWrapper.LogoutWrapperController;
import frontend.MainScreenSeller.*;

/**
 * Controls the functionality of the Financial Reports screen, connecting the view and model.
 */
```

```
public class FinancialReportsScreenController {
    private FinancialReportsScreenView view;
    private FinancialReportsScreenModel model;

    /**
     * Constructs a FinancialReportsScreenController.
     *
     * @param view  The view associated with the financial reports screen.
     * @param model The model containing data for the financial reports screen.
     */
    public FinancialReportsScreenController( FinancialReportsScreenView view,
FinancialReportsScreenModel model) {
        this.model = model;
        this.view = view;

        view.addBackButtonListener(e -> onBackButtonClick());
        view.setTotalSalesLabelText(Integer.toString(model.getTotalSales()));
        view.setTotalProfitLabelText(String.format("$%.2f", model.getTotalProfit()));
    }

    /**
     * Handles the action when the back button is clicked.
     * Redirects to the MainScreenSeller upon clicking the back button.
     */
    private void onBackButtonClick() {
            // back to MainScreenSeller

            MainScreenSellerView view = new MainScreenSellerView();
            MainScreenSellerModel model = new MainScreenSellerModel(this.model.getSeller());
            LogoutWrapperController.Instance.changeMVC(model, view, new
MainScreenSellerController(view, model));
    }
}
```

FinancialReportsScreenModel.java:

```
package frontend.FinancialReportsScreen;

import common.Seller;

/**
 * Represents the model containing data for the Financial Reports screen.
 */
public class FinancialReportsScreenModel {
    private Seller seller;
```

```
    /**
     * Constructs a FinancialReportsScreenModel with the associated Seller.
     *
     * @param seller The Seller associated with the financial reports.
     */
    public FinancialReportsScreenModel(Seller seller) {
        this.seller = seller;
    }

    /**
     * Retrieves the Seller associated with the financial reports.
     *
     * @return The Seller associated with the financial reports.
     */
    public Seller getSeller() {
        return seller;
    }

    /**
     * Simulates and retrieves the total sales for the financial reports.
     *
     * @return The total sales for the financial reports.
     */
    public int getTotalSales() {
        return 50000; // Simulated sales retrieved from the server
    }

    /**
     * Simulates and retrieves the total profit calculated for the financial reports.
     *
     * @return The total profit for the financial reports.
     */
    public double getTotalProfit() {
        return 123456.78; // Simulated profit calculation retrieved from the server
    }
}
```

FinancialReportsScreenView.java:

```
package frontend.FinancialReportsScreen;

import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
```

```java
import javax.swing.*;

/**
 * Represents the view component for displaying financial reports.
 */
public class FinancialReportsScreenView extends JComponent {
	private JButton backButton;
	private JLabel totalSalesLabel, totalProfitLabel;

	/**
    * Constructs the FinancialReportsScreenView.
    * Initializes and arranges the graphical components to display financial reports.
    */
	public FinancialReportsScreenView() {
		JPanel allContainer = new JPanel();
		allContainer.setLayout(new BorderLayout());

		backButton = new JButton("Back to Main Menu");
		totalSalesLabel = new JLabel("0");
		totalProfitLabel = new JLabel("$0.00");

		JPanel labelContainer = new JPanel(new GridLayout(2, 2, 10, 10));
		labelContainer.add(new JLabel("Total sales: "));
		labelContainer.add(totalSalesLabel);
		labelContainer.add(new JLabel("Total profit: "));
		labelContainer.add(totalProfitLabel);

		allContainer.add(backButton, BorderLayout.NORTH);
		allContainer.add(labelContainer);

		add(allContainer);

		setLayout(new FlowLayout());
	setVisible(true);
	}

	/**
    * Adds an ActionListener to the back button.
    *
    * @param listener The ActionListener to be added to the back button.
    */
   public void addBackButtonListener(ActionListener listener) {
      backButton.addActionListener(listener);
   }
```

```
    /**
     * Sets the text for the total sales label.
     *
     * @param text The text to be set for the total sales label.
     */
    public void setTotalSalesLabelText(String text) {
        totalSalesLabel.setText(text);
    }

    /**
     * Sets the text for the total profit label.
     *
     * @param text The text to be set for the total profit label.
     */
    public void setTotalProfitLabelText(String text) {
        totalProfitLabel.setText(text);
    }

}
```

LoginController.java:

```
package frontend.Login;

import javax.swing.SwingUtilities;
import frontend.MainWindow;
import frontend.LogoutWrapper.*;
import frontend.MainScreenCustomer.*;
import frontend.MainScreenSeller.*;
import frontend.SignUp.*;
import common.*;

/**
 * Controls the login functionality, handling interactions between the LoginView and LoginModel.
 */
public class LoginController {
        LoginModel model;
        LoginView view;

        /**
     * Constructs a LoginController with the associated LoginModel and LoginView.
     *
     * @param model The model containing login-related functionality and data.
     * @param view  The view responsible for displaying the login interface.
     */
```

```
	public LoginController(LoginModel model, LoginView view) {
		this.model = model;
		this.view = view;

		view.addLoginButtonListener(e -> onLoginButtonClick());
		view.addSignUpButtonListener(e -> onSignUpButtonClick());
	}

	/**
     * Handles the action when the login button is clicked.
     * Performs login verification, switches views based on user type, and handles the login process.
     */
	private void onLoginButtonClick() {
		view.showBuffering();
		User user = view.getUserType().equals("Customer") ?
		new Customer(view.getUsername(), view.getPassword()) :
		new Seller(view.getUsername(), view.getPassword());

		model.verifyCredentials(user)
			.whenComplete((response, throwable) -> {
				SwingUtilities.invokeLater(() -> {
					if (response.verified) {
						System.out.println("Successfully logged in!");

						// TODO:
						//	- Create LogoutWrapper
						//	- Get user type from Model, then create either:
						//		- MainScreenCustomer
						//		- MainScreenSeller
						//	- Switch window using MainWindow.changeMVC
						LogoutWrapperView lgwView = new LogoutWrapperView();
						LogoutWrapperModel lgwModel = new LogoutWrapperModel();
						LogoutWrapperController lgwController = new LogoutWrapperController(lgwView, lgwModel);

						if (response.user.getClass() == Customer.class) {
							// Customer
							Customer customer = (Customer) user;
```

```
                                                MainScreenCustomerView mscView = new
MainScreenCustomerView();
                                                MainScreenCustomerModel mscModel =
new MainScreenCustomerModel(customer);
                                                MainScreenCustomerController
mscController = new MainScreenCustomerController(mscView, mscModel);

                                                lgwController.changeMVC(mscModel,
mscView, mscController);
                                        } else {
                                                // Seller
                                                Seller seller = (Seller) user;

                                                MainScreenSellerView mssView = new
MainScreenSellerView();
                                                MainScreenSellerModel mssModel = new
MainScreenSellerModel(seller);
                                                MainScreenSellerController mssController =
new MainScreenSellerController(mssView, mssModel);

                                                lgwController.changeMVC(mssView,
mssView, mssController);
                                        }

                                        MainWindow.Instance.changeMVC(lgwModel,
lgwView, lgwController);

                                } else {
                                        System.out.println("Failed to login!");
                                        if (throwable != null)

view.displayLoginFailure(throwable.toString());
                                        else
                                                view.displayLoginFailure("Failed to login!");
                                }
                                view.hideBuffering();
                        });
                });
        }

        /**
     * Handles the action when the sign-up button is clicked.
     * Navigates to the sign-up view for user registration.
     */
```

```
        private void onSignUpButtonClick() {
                // navigate to SignUpView
                SignUpView signUpView = new SignUpView();
                SignUpModel signUpModel = new SignUpModel();
                SignUpController signUpController = new SignUpController(signUpModel,
signUpView);

                MainWindow.Instance.changeMVC(signUpModel, signUpView,
signUpController);
        }
}
```

LoginModel.java:

```
package frontend.Login;

import java.util.concurrent.CompletableFuture;

import common.*;

public class LoginModel {

        /**
         * Verifies the provided user's credentials asynchronously.
         *
         * @param user The User object containing username and password to be registered.
         * @return A CompletableFuture<VerifyCredentialsResponse> representing the
asynchronous registration process.
         *         The CompletableFuture will complete with a VerifyCredentialsResponse
indicating success or failure.
         */
        public CompletableFuture<VerifyCredentialsResponse> verifyCredentials(User user) {
      return CompletableFuture.supplyAsync(() -> {
        try {
          // Simulating a delay for connection and verification
          Thread.sleep(2000);
          return new VerifyCredentialsResponse(true, user);
        } catch (InterruptedException e) {
          e.printStackTrace();
          return new VerifyCredentialsResponse(false, user);
        }
      });
    }
}
```

LoginView.java:

```
package frontend.Login;
```

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

/**
 * Represents the view for the login functionality.
 */
public class LoginView extends JComponent {
	private JTextField usernameField;
	private JPasswordField passwordField;
	private JButton loginButton;
	private JButton signUpButton;
	private JToggleButton userTypeButton;

	// for buffering symbol
	private JProgressBar progressBar;

	// for login failure window
	private JFrame failureFrame;

	/**
    * Constructs the LoginView with graphical components for the login interface.
    */
	public LoginView() {
      usernameField = new JTextField(20);
      passwordField = new JPasswordField(20);
      loginButton = new JButton("Login");
      signUpButton = new JButton("Sign up");

      userTypeButton = new JToggleButton("Customer");
      userTypeButton.addActionListener(new ActionListener() {
         @Override
         public void actionPerformed(ActionEvent e) {
            if (userTypeButton.isSelected()) {
                  userTypeButton.setText("Seller");
            } else {
                  userTypeButton.setText("Customer");
            }
         }
      });
```

```
        JPanel inputPanel = new JPanel(new GridLayout(0, 2, 5, 5)); // 0 rows for variable number
of components
        inputPanel.add(new JLabel("Username:"));
        inputPanel.add(usernameField);
        inputPanel.add(new JLabel("Password:"));
        inputPanel.add(passwordField);
        inputPanel.add(new JLabel("User type:"));
        inputPanel.add(userTypeButton);

        JPanel buttonPanel = new JPanel();
        buttonPanel.add(loginButton);
        buttonPanel.add(signUpButton);

        JPanel mainPanel = new JPanel();
        mainPanel.setLayout(new BorderLayout());
        mainPanel.setPreferredSize(new Dimension(300, 150)); // Set preferred size

        mainPanel.add(inputPanel, BorderLayout.CENTER);
        mainPanel.add(buttonPanel, BorderLayout.SOUTH);

        this.setLayout(new GridBagLayout()); // Use GridBagLayout for LoginView
        GridBagConstraints gbc = new GridBagConstraints();
        gbc.gridx = gbc.gridy = 0;
        gbc.weightx = gbc.weighty = 1;
        gbc.fill = GridBagConstraints.BOTH; // Fill available space
        this.add(mainPanel, gbc);
           }

           /**
     * Displays a login failure window when a user fails to log in.
     * Creates a new window that the user will close once they read it.
     *
     * @param failureMessage Message to display on the failure frame.
     */
           public void displayLoginFailure(String failureMessage) {
              if (failureFrame != null) {
                 failureFrame.dispose();
              }

              failureFrame = new JFrame("Login failed!");
              failureFrame.setSize(200, 120);
              failureFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

              JPanel panel = new JPanel(new FlowLayout(FlowLayout.CENTER));
```

```
            JLabel label = new JLabel(failureMessage);
            label.setFont(label.getFont().deriveFont(Font.PLAIN, 18f));
            label.setHorizontalAlignment(JLabel.CENTER);
            panel.add(label);

            Box verticalBox = Box.createVerticalBox();
            verticalBox.add(Box.createVerticalGlue()); // Add glue to center vertically
            verticalBox.add(panel);
            verticalBox.add(Box.createVerticalGlue()); // Add additional glue for spacing

            failureFrame.add(verticalBox, BorderLayout.CENTER);

            failureFrame.setVisible(true);
        }

        /**
         * Shows a symbol to let user know the system is buffering.
         */
        public void showBuffering() {
                //throw new UnsupportedOperationException();
                progressBar = new JProgressBar();
                progressBar.setIndeterminate(true);
                progressBar.setString("Attempting to login...");
                progressBar.setStringPainted(true);
                progressBar.setPreferredSize(new Dimension(200, 30));

    int x = (this.getWidth() - progressBar.getPreferredSize().width) / 2;
    int y = (this.getHeight() - progressBar.getPreferredSize().height) / 2;

    // Set the position of the progress bar
    progressBar.setBounds(x, y, progressBar.getPreferredSize().width,
progressBar.getPreferredSize().height);
    this.add(progressBar);

    this.setVisible(true);
        }

        /**
         * Hides the buffering symbol.
         */
        public void hideBuffering() {
                this.remove(progressBar);
                this.setVisible(true);
                this.revalidate();
```

```
                this.repaint();
        }

        /**
 * Retrieves the entered username from the username field.
 *
 * @return The entered username.
 */
        public String getUsername() {
                return usernameField.getText();
        }

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

        /**
 * Retrieves the type of user selected (Customer or Seller) from the toggle button.
 *
 * @return The type of user selected.
 */
public String getUserType() {
        return userTypeButton.getText();
}

        /**
 * Adds an ActionListener to the login button.
 *
 * @param listener Listener to add to the login button.
 */
        public void addLoginButtonListener(ActionListener listener) {
                loginButton.addActionListener(listener);
        }

        /**
 * Adds an ActionListener to the sign-up button.
 *
 * @param listener Listener to add to the sign-up button.
 */
```

```
	public void addSignUpButtonListener(ActionListener listener) {
		signUpButton.addActionListener(listener);
	}
}
```

LogoutWrapperController.java:

```
package frontend.LogoutWrapper;

import javax.swing.JComponent;
import frontend.*;
import frontend.Login.*;

/**
 * Controls the logout wrapper functionality, implementing the IWrapper interface.
 */
public class LogoutWrapperController implements IWrapper {
	public static LogoutWrapperController Instance;

	private LogoutWrapperView view;
	private LogoutWrapperModel model;

	private Object customModel, customController;

	/**
    * Constructs a LogoutWrapperController with the associated LogoutWrapperView and
LogoutWrapperModel.
    *
    * @param view  The view component for the logout wrapper.
    * @param model The model containing logout-related functionality and data.
    */
	public LogoutWrapperController(LogoutWrapperView view, LogoutWrapperModel
model) {
		Instance = this;

		this.view = view;
		this.model = model;

		this.view.addLogoutButtonListener(e -> onLogoutButtonClick());
	}

	/**
    * Handles the action when the logout button is clicked.
    * Performs the logout operation and navigates back to the login view.
    */
	private void onLogoutButtonClick() {
```

```
                model.logout();

                LoginModel model = new LoginModel();
                LoginView view = new LoginView();

                MainWindow.Instance.changeMVC(model, view, new LoginController(model,
view));
        }

        /**
    * Changes the Model-View-Controller (MVC) components for a custom view in the logout
wrapper.
    *
    * @param model     The custom model component.
    * @param view      The custom view component (inherited from JComponent).
    * @param controller The custom controller component.
    */
        public void changeMVC(Object model, JComponent view, Object controller) {
                this.customModel = model;
                this.customController = controller;

                this.view.updateView(view);
        }
}
```

LogoutWrapperModel.java:

```
package frontend.LogoutWrapper;

public class LogoutWrapperModel {

        /**
         * Ends the current user session.
         * Note: This method does nothing as the user session is not stored on the
server.
         */
        public void logout() {
           // Implementation: This method does nothing as the user session is not
stored on the server.
        }
}
```

LogoutWrapperView.java:

```
package frontend.LogoutWrapper;
```

```java
import java.awt.BorderLayout;
import java.awt.event.ActionListener;

import javax.swing.*;
import frontend.IWrapper;

/**
 * Represents the view component for the logout wrapper.
 */
public class LogoutWrapperView extends JComponent {
	private JComponent view;
	private JPanel wrapperPanel;
	private JButton logoutButton;

	/**
    * Constructs the LogoutWrapperView with a logout button and a wrapper panel.
    */
	public LogoutWrapperView() {
		setLayout(new BorderLayout());

		wrapperPanel = new JPanel(new BorderLayout());
		logoutButton = new JButton("Logout");
		wrapperPanel.add(logoutButton, BorderLayout.NORTH);

		this.add(wrapperPanel);
		this.setVisible(true);
	}

	/**
    * Adds an ActionListener to the logout button.
    *
    * @param listener The ActionListener to be added to the logout button.
    */
	public void addLogoutButtonListener(ActionListener listener) {
		logoutButton.addActionListener(listener);
	}

	/**
    * Updates the view within the logout wrapper.
    * Replaces the existing view with a new one in the wrapper panel.
    *
    * @param view The new view component to be displayed within the wrapper.
    */
	public void updateView(JComponent view) {
```

```
                if (this.view != null)
                        wrapperPanel.remove(this.view);
                this.view = view;
                wrapperPanel.add(view, BorderLayout.CENTER);
                this.revalidate();
                this.repaint();
        }

}
```

MainScreenCustomerController.java:

```
package frontend.MainScreenCustomer;

import java.util.*;
import common.*;
import frontend.LogoutWrapper.LogoutWrapperController;
import frontend.ShoppingCartScreen.*;
import frontend.Product.*;

/**
 * Controls the functionality of the main screen for customers, handling interactions between the
view and model.
 */
public class MainScreenCustomerController {
        private MainScreenCustomerView view;
        private MainScreenCustomerModel model;
        private List<ProductController> pControllers;

        /**
    * Constructs a MainScreenCustomerController with the associated
MainScreenCustomerView and MainScreenCustomerModel.
    *
    * @param view  The view component for the main screen of customers.
    * @param model The model containing data and functionality for the main screen of
customers.
    */
        public MainScreenCustomerController(MainScreenCustomerView view,
MainScreenCustomerModel model) {
                this.view = view;
                this.model = model;

                List<ProductView> views = new LinkedList<ProductView>();
                pControllers = new LinkedList<ProductController>();
```

```
                for (Product product : model.getProducts()) {
                        ProductView pView = new ProductView();
                        views.add(pView);

                        pView.setTitleLabelText(product.getName());
                        pView.setDescriptionLabelText(product.getDescription());
                        pView.setPriceLabelText(String.format("$%.2f", product.getPrice()));

                        ProductModel pModel = new ProductModel(product,
model.getCustomer().getShoppingCart());

                        pView.setCountLabelText(Integer.toString(pModel.getCount()) + " item(s)
in cart.");

                        ProductController pController = new ProductController(pView, pModel);
                        pControllers.add(pController);
                }

                view.addProductViews(views);

                view.addShoppingCartButtonListener(e -> onShoppingCartButtonClick());
        }

        /**
    * Handles the action when the shopping cart button is clicked.
    * Navigates to the shopping cart page.
    */
        private void onShoppingCartButtonClick() {
                // navigate to shopping cart page

                ShoppingCartScreenView view = new ShoppingCartScreenView();
                ShoppingCartScreenModel model = new
ShoppingCartScreenModel(this.model.getCustomer());
                LogoutWrapperController.Instance.changeMVC(model, view, new
ShoppingCartScreenController(view, model));
        }
}
```

MainScreenCustomerModel.java:

```
package frontend.MainScreenCustomer;

import java.util.*;
import common.*;

/**
```

```
 * Represents the model component for the main screen of a customer.
 */
public class MainScreenCustomerModel {
	private Customer customer;

	/**
    * Constructs a MainScreenCustomerModel with the associated Customer.
    *
    * @param customer The customer associated with the main screen.
    */
	public MainScreenCustomerModel(Customer customer) {
		this.customer = customer;
	}

	/**
    * Retrieves the customer associated with the main screen.
    *
    * @return The customer associated with the main screen.
    */
	public Customer getCustomer() {
		return customer;
	}

	/**
    * Retrieves a list of products available for display on the main screen.
    *
    * @return A list of products available for display.
    */
	public List<Product> getProducts() {
		// Would load from server

		return new LinkedList<Product>(Arrays.asList(
			new Product("Item 1", "An item you can buy", 100.00),
			new Product("Item 2", "Another item you can buy", 200.00),
			new Product("Item 3", "Last item you can buy", 300.00),
			new Product("Item 4", "test", 35.125),
			new Product("Item 5", "test2", 11.35),
			new Product("Item 6", "test3", 78.46)
		));
	}
}
```

MainScreenCustomerView.java:

```
package frontend.MainScreenCustomer;
```

```java
import javax.swing.*;
import frontend.Product.*;
import java.awt.*;
import java.awt.event.ActionListener;
import java.util.List;

/**
 * Represents the view component for the main screen of a customer.
 */
public class MainScreenCustomerView extends JComponent {
        private JPanel productContainer;
        private JButton shoppingCartButton;

        /**
         * Constructs the MainScreenCustomerView with components for displaying products and a shopping cart button.
         */
    public MainScreenCustomerView() {
        JPanel allContainer = new JPanel();
        allContainer.setLayout(new BorderLayout());

        productContainer = new JPanel(new GridLayout(0, 3, 10, 10));

        shoppingCartButton = new JButton("View Shopping Cart");

        JScrollPane scrollPane = new JScrollPane(productContainer);
        scrollPane.setPreferredSize(new Dimension(1100, 600));
        scrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

        allContainer.add(shoppingCartButton, BorderLayout.NORTH);
        allContainer.add(scrollPane);

        add(allContainer);

        setLayout(new FlowLayout());
        setVisible(true);
    }

    /**
     * Adds a single product view to the product container panel.
     *
     * @param productView The product view to be added.
     */
    public void addProductView(ProductView productView) {
```

```
            productContainer.add(productView);
            revalidate();
            repaint();
    }

    /**
     * Adds multiple product views to the product container panel.
     *
     * @param productViews The list of product views to be added.
     */
    public void addProductViews(List<ProductView> productViews) {
            for (ProductView view : productViews) {
                    productContainer.add(view);
            }
            revalidate();
            repaint();
    }

    /**
     * Clears the product container panel, removing all product views.
     */
    public void resetProductContainer() {
            productContainer.removeAll();
    }

    /**
     * Adds an ActionListener to the shopping cart button.
     *
     * @param listener The ActionListener to be added to the shopping cart button.
     */
    public void addShoppingCartButtonListener(ActionListener listener) {
            shoppingCartButton.addActionListener(listener);
    }
}
```

MainScreenSellerController.java:

```
package frontend.MainScreenSeller;

import frontend.LogoutWrapper.LogoutWrapperController;
import frontend.SellerInventoryScreen.*;
import frontend.FinancialReportsScreen.*;

/**
 * Controls the functionality of the main screen for sellers, handling interactions between the
view and model.
```

```
 */
public class MainScreenSellerController {
   private MainScreenSellerView view;
   private MainScreenSellerModel model;

   /**
    * Constructs a MainScreenSellerController with the associated MainScreenSellerView and
MainScreenSellerModel.
    *
    * @param view  The view component for the main screen of sellers.
    * @param model The model containing data and functionality for the main screen of sellers.
    */
        public MainScreenSellerController(MainScreenSellerView view,
MainScreenSellerModel model) {
                this.view = view;
                this.model = model;

                view.addSellerInventoryButtonListener(e -> onSellerInventoryButtonClick());
                view.addFinancialReportsButtonListener(e -> onFinancialReportsButtonClick());
   }

        /**
    * Handles the action when the seller inventory button is clicked.
    * Navigates to the seller inventory screen.
    */
        private void onSellerInventoryButtonClick() {
                // open seller inventory screen
                SellerInventoryScreenView view = new SellerInventoryScreenView();
                SellerInventoryScreenModel model = new
SellerInventoryScreenModel(this.model.getSeller());
                LogoutWrapperController.Instance.changeMVC(model, view, new
SellerInventoryScreenController(view, model));
        }

        /**
    * Handles the action when the financial reports button is clicked.
    * Navigates to the financial reports screen.
    */
        private void onFinancialReportsButtonClick() {
                // open financial reports screen
                FinancialReportsScreenView view = new FinancialReportsScreenView();
                FinancialReportsScreenModel model = new
FinancialReportsScreenModel(this.model.getSeller());
```

```
                LogoutWrapperController.Instance.changeMVC(model, view, new
FinancialReportsScreenController(view, model));
        }
}
```

MainScreenSellerModel.java:

```
package frontend.MainScreenSeller;

import common.*;

/**
 * Represents the model component for the main screen of a seller.
 */
public class MainScreenSellerModel {
        private Seller seller;

        /**
    * Constructs a MainScreenSellerModel with the associated Seller.
    *
    * @param seller The seller associated with the main screen.
    */
        public MainScreenSellerModel(Seller seller) {
                this.seller = seller;
        }

        /**
    * Retrieves the seller associated with the main screen.
    *
    * @return The seller associated with the main screen.
    */
        public Seller getSeller() {
                return seller;
        }
}
```

MainScreenSellerView.java:

```
package frontend.MainScreenSeller;

import javax.swing.*;
import java.awt.BorderLayout;
import java.awt.event.ActionListener;

/**
 * Represents the view component for the main screen of a seller.
 */
public class MainScreenSellerView extends JComponent {
```

```
        private JButton sellerInventoryButton, financialReportsButton;

        /**
     * Constructs the MainScreenSellerView with buttons for seller inventory and financial reports.
     */
    public MainScreenSellerView() {
        // Create UI components
        sellerInventoryButton = new JButton("Open Seller Inventory");
        financialReportsButton = new JButton("Open Financial Reports");

        // Create a panel to hold the buttons
        JPanel buttonPanel = new JPanel();
        buttonPanel.add(sellerInventoryButton);
        buttonPanel.add(financialReportsButton);

        // Set layout for the main component
        setLayout(new BorderLayout());

        // Add components to the main component
        add(buttonPanel, BorderLayout.CENTER);

        // Set component properties
        setPreferredSize(getPreferredSize());
    }

    /**
     * Adds an ActionListener to the seller inventory button.
     *
     * @param listener The ActionListener to be added to the seller inventory button.
     */
    public void addSellerInventoryButtonListener(ActionListener listener) {
        sellerInventoryButton.addActionListener(listener);
    }

    /**
     * Adds an ActionListener to the financial reports button.
     *
     * @param listener The ActionListener to be added to the financial reports button.
     */
    public void addFinancialReportsButtonListener(ActionListener listener) {
        financialReportsButton.addActionListener(listener);
    }
}
```

PaymentScreenController.java:

```java
package frontend.PaymentScreen;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JOptionPane;

import frontend.LogoutWrapper.LogoutWrapperController;
import frontend.ShipmentScreen.*;

/**
 * Controls the payment screen functionality, managing interactions between the view and
model.
 */
public class PaymentScreenController {
    private PaymentScreenView view;
    private PaymentScreenModel model;

    /**
     * Constructs a PaymentScreenController with the associated view and model.
     *
     * @param view  The view component for the payment screen.
     * @param model The model containing data and functionality for payment processing.
     */
    public PaymentScreenController(PaymentScreenView view, PaymentScreenModel model)
{
        this.view = view;
        this.model = model;

        // Set up the submit button listener
        view.setSubmitButtonListener(new SubmitButtonListener());

        view.setTotalCostLabelText(String.format("Total cost: $%.2f",
model.getCustomer().getShoppingCart().getCartTotal()));
    }

    /**
     * ActionListener implementation for the submit button on the payment screen.
     * Handles payment verification and transaction processing.
     */
    private class SubmitButtonListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            // Retrieve data from the view
            String cardNumber = view.getCardNumber();
```

```
            String expiryDate = view.getExpiryDate();
            String cvv = view.getCVV();
            // Verify payment details
            boolean paymentDetailsVerified = model.verifyPaymentDetails(cardNumber,
expiryDate, cvv);
            if (paymentDetailsVerified) {
                // Process transaction
                boolean transactionSuccess = model.processTransaction();
                if (transactionSuccess) {
                    JOptionPane.showMessageDialog(view, "Payment successful!");

                    model.getCustomer().getShoppingCart().removeAllProducts();

                    switchToShipmentScreen();
                } else {
                    JOptionPane.showMessageDialog(view, "Transaction failed. Please try again.");
                }
            } else {
                JOptionPane.showMessageDialog(view, "Invalid payment details. Please check and
try again.");
            }
        }
    }

    /**
     * Switches the view to the shipment screen after successful payment processing.
     */
    private void switchToShipmentScreen() {
            ShipmentScreenView view = new ShipmentScreenView();
            ShipmentScreenModel model = new ShipmentScreenModel(this.model.getCustomer());
            LogoutWrapperController.Instance.changeMVC(model, view, new
ShipmentScreenController(view, model));
    }
}
```

PaymentScreenModel.java:

```
package frontend.PaymentScreen;

import java.util.HashSet;
import java.util.Set;
import java.util.regex.Pattern;

import common.Customer;

import java.time.LocalDate;
```

```java
import java.time.YearMonth;

/**
 * Manages payment-related data and logic for the payment screen.
 */
public class PaymentScreenModel {
	private Customer customer;

	/**
    * Constructs a PaymentScreenModel associated with a specific customer.
    *
    * @param customer The customer initiating the payment.
    */
	public PaymentScreenModel(Customer customer) {
		this.customer = customer;
	}

	/**
    * Retrieves the customer associated with the payment process.
    *
    * @return The customer initiating the payment.
    */
	public Customer getCustomer() {
		return customer;
	}

	/**
    * Verifies payment details provided by the user.
    *
    * @param cardNumber  The card number entered by the user.
    * @param expiryDate  The expiration date entered by the user.
    * @param cvv         The CVV entered by the user.
    * @return True if payment details are valid, false otherwise.
    */
   public boolean verifyPaymentDetails(String cardNumber, String expiryDate, String cvv) {
		// testing REMOVE LATER

      // Verify card number (must be 16 digits)
      if (!isValidCardNumber(cardNumber) && false) {
         return false;
      }
      // Verify expiration date (must be in "MM/YY" format and between 2023 and 2026)
      if (!isValidExpiryDate(expiryDate) && false) {
         return false;
```

```
        }
        // Verify CVV (must be 3 digits)
        if (!isValidCVV(cvv) && false) {
            return false;
        }
        // Payment details are considered valid
        return true;
    }

    /**
     * Checks if the provided card number follows the standard format of 16 digits.
     *
     * @param cardNumber The card number to be validated.
     * @return True if the card number is in the correct format, false otherwise.
     */
    private boolean isValidCardNumber(String cardNumber) {
        return cardNumber != null && cardNumber.matches("\\d{16}");
    }

    /**
     * Validates the expiry date format and checks if it's within the next 3 years.
     *
     * @param expiryDate The expiry date string in "MM/YY" format.
     * @return True if the format is correct and within the next 3 years, false otherwise.
     */
    private boolean isValidExpiryDate(String expiryDate) {
        if (expiryDate == null || !expiryDate.matches("\\d{2}/\\d{2}")) {
            return false;
        }
        String[] parts = expiryDate.split("/");
        int month = Integer.parseInt(parts[0]);
        int year = Integer.parseInt(parts[1]);
        YearMonth currentYearMonth = YearMonth.now();
        YearMonth inputYearMonth = YearMonth.of(year, month);
        return inputYearMonth.isAfter(currentYearMonth) &&
inputYearMonth.isBefore(currentYearMonth.plusYears(3));
    }

    /**
     * Validates the Card Verification Value (CVV) format, ensuring it consists of 3 digits.
     *
     * @param cvv The CVV string to be validated.
     * @return True if the CVV has the correct format, false otherwise.
     */
```

```java
    private boolean isValidCVV(String cvv) {
        return cvv != null && cvv.matches("\\d{3}");
    }

    /**
     * Processes a transaction based on the provided card number.
     *
     * @param cardNumber The card number used for the transaction.
     * @return True if the transaction is successful, false otherwise.
     */
    public boolean processTransaction(String cardNumber) {
        // Check if cardNumber has 16 unique integers
        if (!hasUniqueIntegers(cardNumber)) {
            System.out.println("Transaction failed. Card must have 16 unique integers.");
            return false;
        }
        // Couldn't think of a fail state so I figured the 16 integers for a card need to be different and
not just the same number repeating
        System.out.println("Transaction successful.");
        return true;
    }

    /**
     * Verifies if the provided card number has 16 unique integers.
     *
     * @param cardNumber The card number to be checked.
     * @return True if the card number has 16 unique integers, false otherwise.
     */
    private boolean hasUniqueIntegers(String cardNumber) {
        Set<Character> uniqueDigits = new HashSet<>();
        for (char digit : cardNumber.toCharArray()) {
            if (!Character.isDigit(digit) || !uniqueDigits.add(digit)) {
                return false; // Not a digit or not unique
            }
        }
        return true;
    }

    /**
     * Processes a default transaction without specific card number validation.
     *
     * @return True for a successful transaction (default), as no validation is performed.
     */
    public boolean processTransaction() {
```

```
        return true;
    }
}
```

PaymentScreenView.java:

```
package frontend.PaymentScreen;

import java.awt.FlowLayout;
import java.awt.GridLayout;

import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.JComponent;
import java.awt.event.ActionListener;

/**
 * Represents the payment screen view for the application.
 * Allows users to input payment information for their purchase.
 */
public class PaymentScreenView extends JComponent {
    private JTextField cardNumberField, expiryDateField, cvvField;
    private JButton submitButton;
    private JLabel totalCostLabel;

    /**
     * Constructs a PaymentScreenView object, initializing the UI components.
     * Creates fields to input card details and a button to submit the payment.
     */
    public PaymentScreenView() {
        JPanel allContainer = new JPanel(new GridLayout(2, 1, 10, 10));

        totalCostLabel = new JLabel("Total cost: $0.00");
        allContainer.add(totalCostLabel);

        JPanel creditCardInfoPanel = new JPanel(new FlowLayout());
        creditCardInfoPanel.add(new JLabel("Card Number:"));
        cardNumberField = new JTextField(16);
        creditCardInfoPanel.add(cardNumberField);
        creditCardInfoPanel.add(new JLabel("Expiry Date:"));
        expiryDateField = new JTextField(5);
        creditCardInfoPanel.add(expiryDateField);
```

```
        creditCardInfoPanel.add(new JLabel("CVV:"));
        cvvField = new JTextField(3);
        creditCardInfoPanel.add(cvvField);
        submitButton = new JButton("Submit");
        creditCardInfoPanel.add(submitButton);

        allContainer.add(creditCardInfoPanel);

        add(allContainer);
        setLayout(new FlowLayout());
        setVisible(true);
    }

    /**
     * Retrieves the card number entered by the user.
     *
     * @return The entered card number as a string.
     */
    public String getCardNumber() {
        return cardNumberField.getText();
    }

    /**
     * Retrieves the expiry date entered by the user.
     *
     * @return The entered expiry date as a string in "MM/YY" format.
     */
    public String getExpiryDate() {
        return expiryDateField.getText();
    }

    /**
     * Retrieves the CVV entered by the user.
     *
     * @return The entered CVV as a string.
     */
    public String getCVV() {
        return cvvField.getText();
    }

    /**
     * Sets an ActionListener for the submit button.
```

```
     *
     * @param listener ActionListener to be set for the submit button.
     */
    public void setSubmitButtonListener(ActionListener listener) {
        submitButton.addActionListener(listener);
    }

    /**
     * Sets the label text displaying the total cost to be paid.
     *
     * @param text The text to be displayed as the total cost.
     */
    public void setTotalCostLabelText(String text) {
         totalCostLabel.setText(text);
    }
}
```

ProductController.java:

```
package frontend.Product;

/**
 * Manages the interaction between the ProductView and ProductModel for an individual product in the application.
 * Handles user actions on the product view and updates the associated model accordingly.
 */
public class ProductController {
    private ProductView view;
    private ProductModel model;

    /**
     * Constructs a ProductController object with the associated ProductView and ProductModel.
     * Registers listeners for the view's increment and decrement buttons.
     *
     * @param view  The ProductView object associated with this controller.
     * @param model The ProductModel object associated with this controller.
     */
    public ProductController(ProductView view, ProductModel model) {
        this.view = view;
        this.model = model;

        view.addIncrementButtonListener(e -> onIncrementButtonClicked());
        view.addDecrementButtonListener(e -> onDecrementButtonClicked());
    }
```

```
    /**
     * Handles the action when the increment button is clicked.
     * Increments the product quantity in the cart through the model and updates the view.
     */
    private void onIncrementButtonClicked() {
        model.incrementToCart();
        updateCountLabel();
    }

    /**
     * Handles the action when the decrement button is clicked.
     * Decrements the product quantity in the cart through the model and updates the view.
     */
    private void onDecrementButtonClicked() {
        model.decrementFromCart();
        updateCountLabel();
    }

    /**
     * Updates the count label in the view to display the current quantity of the product in the cart.
     */
    private void updateCountLabel() {
        view.setCountLabelText(Integer.toString(model.getCount()) + " item(s) in cart.");
    }
}
```

ProductModel.java:

```
package frontend.Product;
import common.*;
/**
* Manages the data and actions related to an individual product in the shopping cart.
* Interacts with the ShoppingCart to modify the quantity of the associated product in the cart.
*/
public class ProductModel {
  private Product product;
  private ShoppingCart shoppingCart;
  /**
   * Constructs a ProductModel object with the associated Product and ShoppingCart.
   *
   * @param product      The Product object associated with this model.
   * @param shoppingCart The ShoppingCart object associated with this model.
   */
  public ProductModel(Product product, ShoppingCart shoppingCart) {
        this.product = product;
        this.shoppingCart = shoppingCart;
  }
```

```
    /**
     * Retrieves the Product associated with this model.
     *
     * @return The Product object associated with this model.
     */
    public Product getProduct() {
            return product;
    }

    /**
     * Retrieves the count of the associated product in the shopping cart.
     *
     * @return The quantity of the product in the cart.
     */
    public int getCount() {
            return shoppingCart.getCount(product);
    }

    /**
     * Increments the quantity of the associated product in the shopping cart.
     */
    public void incrementToCart() {
            shoppingCart.incrementProduct(product);
    }

    /**
     * Decrements the quantity of the associated product in the shopping cart.
     */
    public void decrementFromCart() {
            shoppingCart.decrementProduct(product);
    }
}
```

ProductView.java:

```
package frontend.Product;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionListener;
import java.util.List;

/**
 * Represents the view for an individual product in the application.
 * Allows users to view product details and manipulate the quantity in their cart.
 */
public class ProductView extends JComponent {
        private JButton incrementButton, decrementButton;
        private JLabel titleLabel, descriptionLabel, priceLabel, countLabel;
```

```
        /**
     * Constructs a ProductView object, initializing the UI components for displaying product
details
     * and quantity manipulation buttons.
     */
    public ProductView() {
        JPanel allContainer = new JPanel();
        allContainer.setLayout(new BorderLayout());

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

        incrementButton = new JButton("+1");
        decrementButton = new JButton("-1");
        buttonPanel.add(incrementButton);
        buttonPanel.add(decrementButton);

        JPanel labelPanel = new JPanel();
        labelPanel.setLayout(new BoxLayout(labelPanel, BoxLayout.Y_AXIS));
        labelPanel.setPreferredSize(new Dimension(200, 100));

        titleLabel = new JLabel("Item Title");
        descriptionLabel = new JLabel("Item Description");
        priceLabel = new JLabel("$0.00");
        countLabel = new JLabel("0 item(s) in cart.");

        labelPanel.add(titleLabel);
        labelPanel.add(descriptionLabel);
        labelPanel.add(priceLabel);
        labelPanel.add(countLabel);

        allContainer.add(labelPanel);
        allContainer.add(buttonPanel, BorderLayout.EAST);

        add(allContainer);

        setLayout(new FlowLayout());
        setVisible(true);
    }

    /**
     * Sets the text for the title label displaying the product title.
     *
```

```
     * @param text The text to be displayed as the product title.
     */
    public void setTitleLabelText(String text) {
        titleLabel.setText(text);
    }

    /**
     * Sets the text for the description label displaying the product description.
     *
     * @param text The text to be displayed as the product description.
     */
    public void setDescriptionLabelText(String text) {
        descriptionLabel.setText(text);
    }

    /**
     * Sets the text for the price label displaying the product price.
     *
     * @param text The text to be displayed as the product price.
     */
    public void setPriceLabelText(String text) {
        priceLabel.setText(text);
    }

    /**
     * Sets the text for the count label displaying the quantity of the product in the cart.
     *
     * @param text The text to be displayed as the product quantity in the cart.
     */
    public void setCountLabelText(String text) {
        countLabel.setText(text);
    }

    /**
     * Adds an ActionListener for the increment button to increase the product quantity in the cart.
     *
     * @param listener ActionListener to be added to the increment button.
     */
    public void addIncrementButtonListener(ActionListener listener) {
        incrementButton.addActionListener(listener);
    }

    /**
```

```
    * Adds an ActionListener for the decrement button to decrease the product quantity in the cart.
    *
    * @param listener ActionListener to be added to the decrement button.
    */
   public void addDecrementButtonListener(ActionListener listener) {
        decrementButton.addActionListener(listener);
   }
}
```

ProductSellerController.java:

```
package frontend.ProductSeller;

import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * Controls the interaction between the ProductSellerView and ProductSellerModel.
 * Handles user input from the view and updates the model accordingly.
 */
public class ProductSellerController {
        private ProductSellerView view;
        private ProductSellerModel model;

        /**
    * Constructs a ProductSellerController object.
    *
    * @param view  The ProductSellerView associated with this controller.
    * @param model The ProductSellerModel associated with this controller.
    */
        public ProductSellerController(ProductSellerView view, ProductSellerModel model) {
                this.view = view;
                this.model = model;

                view.addTitleFieldListener(new KeyListener() {
                        @Override
                        public void keyTyped(KeyEvent e) {}
                        @Override
                        public void keyPressed(KeyEvent e) {
                                onTitleFieldEdit();
                        }
                        @Override
                        public void keyReleased(KeyEvent e) {}
```

```
                });
                view.addDescriptionFieldListener(new KeyListener() {
                        @Override
                        public void keyTyped(KeyEvent e) {}
                        @Override
                        public void keyPressed(KeyEvent e) {}
                        @Override
                        public void keyReleased(KeyEvent e) {
                                onDescriptionFieldEdit();
                        }
                });
                view.addPriceFieldListener(new KeyListener() {
                        @Override
                        public void keyTyped(KeyEvent e) {}
                        @Override
                        public void keyPressed(KeyEvent e) {}
                        @Override
                        public void keyReleased(KeyEvent e) {
                                onPriceFieldEdit(); // i have no idea why it needs to be here but it
does
                        }
                });
        }

        /**
    * Updates the product name in the model when the title field is edited.
    */
        private void onTitleFieldEdit() {
                model.getProduct().setName(view.getTitleFieldText());
        }

        /**
    * Updates the product description in the model when the description field is edited.
    */
        private void onDescriptionFieldEdit() {
                model.getProduct().setDescription(view.getDescriptionFieldText());
        }

        /**
    * Updates the product price in the model when the price field is edited.
    */
        private void onPriceFieldEdit() {
                double price = extractFloatingNumber(view.getPriceFieldText());
                model.getProduct().setPrice(price);
```

```
        }

        /**
     * Extracts a floating-point number from the given input string.
     *
     * @param input The input string from which the number is extracted.
     * @return The extracted floating-point number.
     */
        private double extractFloatingNumber(String input) {
       // Regular expression to match a floating-point number pattern
       Pattern pattern = Pattern.compile("^\\d*\\.\\d+|\\d+\\.\\d*$");
       Matcher matcher = pattern.matcher(input);

       if (matcher.find()) {
           String numberStr = matcher.group(); // Get the matched number string
           return Double.parseDouble(numberStr); // Convert the string to a double
       }

       // Return a default value (or throw an exception) if no floating number found
       return 0.0; // Or throw an exception indicating no floating number found
   }
}
```

ProductSellerModel.java:

```
package frontend.ProductSeller;

import common.Product;

/**
 * Represents the model for a product within the context of a seller.
 * Manages the product details for the seller's view.
 */
public class ProductSellerModel {
        private Product product;

        /**
     * Constructs a ProductSellerModel object.
     *
     * @param product The product associated with this model.
     */
        public ProductSellerModel(Product product) {
                this.product = product;
        }

        /**
```

```
    * Retrieves the product associated with this model.
    *
    * @return The product object.
    */
        public Product getProduct() {
                return product;
        }
}
```

ProductSellerView.java:

```
package frontend.ProductSeller;

import java.awt.*;
import java.awt.event.*;
import java.text.DecimalFormat;
import javax.swing.*;

/**
 * Represents the view for a product within the context of a seller.
 * Manages the graphical representation of the product details in the seller's view.
 */
public class ProductSellerView extends JComponent {
        private JTextField titleField, descriptionField, priceField;
        private JButton deleteProductButton;

        /**
    * Constructs a ProductSellerView object.
    * Initializes UI components for displaying and editing product details.
    */
        public ProductSellerView() {
                JPanel allContainer = new JPanel();
        allContainer.setLayout(new BorderLayout());

        titleField = new JTextField();
        descriptionField = new JTextField();
        priceField = new JTextField();

        deleteProductButton = new JButton("Delete product");

        JPanel labelPanel = new JPanel();
        labelPanel.setLayout(new BoxLayout(labelPanel, BoxLayout.Y_AXIS));
        labelPanel.setPreferredSize(new Dimension(200, 100));

        labelPanel.add(titleField);
        labelPanel.add(descriptionField);
```

```
        labelPanel.add(priceField);

        allContainer.add(labelPanel);
        allContainer.add(deleteProductButton, BorderLayout.EAST);

        add(allContainer);

        setLayout(new FlowLayout());
        setVisible(true);
        }

        /**
    * Retrieves the text entered in the title field.
    *
    * @return The text in the title field.
    */
        public String getTitleFieldText() {
                return titleField.getText();
        }

        /**
    * Retrieves the text entered in the description field.
    *
    * @return The text in the description field.
    */
        public String getDescriptionFieldText() {
                return descriptionField.getText();
        }

        /**
    * Retrieves the text entered in the price field.
    *
    * @return The text in the price field.
    */
        public String getPriceFieldText() {
                return priceField.getText();
        }

        /**
    * Sets the text displayed in the title field.
    *
    * @param title The text to set in the title field.
    */
        public void setTitleFieldText(String title) {
```

```
                titleField.setText(title);
        }

        /**
* Sets the text displayed in the description field.
*
* @param description The text to set in the description field.
*/
        public void setDescriptionFieldText(String description) {
                descriptionField.setText(description);
        }

        /**
* Sets the text displayed in the price field.
*
* @param price The price to set in the price field.
*/
        public void setPriceFieldText(double price) {
                priceField.setText(String.format("%.2f", price));
        }

        /**
* Adds a key listener to the title field.
*
* @param listener The KeyListener to add to the title field.
*/
        public void addTitleFieldListener(KeyListener listener) {
                titleField.addKeyListener(listener);
        }

        /**
* Adds a key listener to the description field.
*
* @param listener The KeyListener to add to the description field.
*/
        public void addDescriptionFieldListener(KeyListener listener) {
                descriptionField.addKeyListener(listener);
        }

        /**
* Adds a key listener to the price field.
*
* @param listener The KeyListener to add to the price field.
*/
```

```
        public void addPriceFieldListener(KeyListener listener) {
                priceField.addKeyListener(listener);
        }

        /**
     * Adds an action listener to the delete product button.
     *
     * @param listener The ActionListener to add to the delete product button.
     */
        public void addDeleteProductButtonListener(ActionListener listener) {
                deleteProductButton.addActionListener(listener);
        }
}
```

SellerInventoryScreen.java:

```
package frontend.SellerInventoryScreen;

import common.Product;
import frontend.LogoutWrapper.LogoutWrapperController;
import frontend.MainScreenSeller.*;
import java.util.*;
import frontend.ProductSeller.*;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * Controls interactions between the SellerInventoryScreenView and
SellerInventoryScreenModel.
 * Manages the behavior and actions related to the seller's inventory screen.
 */
public class SellerInventoryScreenController {
    private SellerInventoryScreenModel model;
    private SellerInventoryScreenView view;
    private List<ProductSellerController> pControllers;

    /**
     * Constructs a SellerInventoryScreenController object.
     * Initializes the controller with the associated view and model.
     *
     * @param view  The SellerInventoryScreenView to be controlled.
     * @param model The SellerInventoryScreenModel to be managed.
     */
    public SellerInventoryScreenController(SellerInventoryScreenView view,
SellerInventoryScreenModel model) {
```

```
        this.model = model;
        this.view = view;

        pControllers = new LinkedList<ProductSellerController>();

        view.addBackButtonListener(e -> onBackButtonClick());
        view.addAddProductButtonListener(e -> onAddProductButtonClick());

        updateProductViews();
    }

    /**
     * Handles the action when the back button is clicked.
     * Navigates back to the MainScreenSeller view and model.
     */
    private void onBackButtonClick() {
            // back to MainScreenSeller

            MainScreenSellerView view = new MainScreenSellerView();
            MainScreenSellerModel model = new MainScreenSellerModel(this.model.getSeller());
            LogoutWrapperController.Instance.changeMVC(model, view, new
MainScreenSellerController(view, model));
    }

    /**
     * Handles the action when the add product button is clicked.
     * Creates a new product and adds it to the seller's inventory.
     */
    private void onAddProductButtonClick() {
            Product product = new Product();

            model.getSeller().addProduct(product);

            updateProductViews();
    }

    /**
     * Updates the product views in the SellerInventoryScreenView based on the
SellerInventoryScreenModel.
     * Refreshes the displayed list of products in the seller's inventory.
     */
    private void updateProductViews() {
            view.removeAllProductViews();
```

```
        List<ProductSellerView> views = new LinkedList<>();

        for (Product product : model.getSeller().getProducts()) {
                ProductSellerView pView = new ProductSellerView();
                views.add(pView);

                pView.setTitleFieldText(product.getName());
                pView.setDescriptionFieldText(product.getDescription());
                pView.setPriceFieldText(product.getPrice());

                pView.addDeleteProductButtonListener(new ActionListener() {
                        @Override
                        public void actionPerformed(ActionEvent e) {
                                model.getSeller().removeProduct(product);
                                updateProductViews();
                        }
                });

                ProductSellerModel pModel = new ProductSellerModel(product);

                ProductSellerController pController = new ProductSellerController(pView,
pModel);
                pControllers.add(pController);
        }

        view.addProductViews(views);
    }
}
```

SellerInventoryScreenModel.java:

```
package frontend.SellerInventoryScreen;

import common.Seller;

/**
 * Represents the model component for the seller's inventory screen.
 * Manages the data related to the seller's inventory.
 */
public class SellerInventoryScreenModel {
        private Seller seller;

        /**
     * Constructs a SellerInventoryScreenModel object.
     * Initializes the model with the associated seller.
     *
```

```
    * @param seller The seller whose inventory is being managed.
    */
        public SellerInventoryScreenModel(Seller seller) {
                this.seller = seller;
        }

        /**
    * Retrieves the seller associated with the inventory model.
    *
    * @return The seller object managed by this model.
    */
        public Seller getSeller() {
                return seller;
        }
}
```

SellerInventoryScreenView.java:

```
package frontend.SellerInventoryScreen;

import java.awt.*;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.util.List;
import frontend.ProductSeller.*;

/**
 * Represents the view for the seller's inventory screen.
 * Manages the graphical representation of the seller's inventory, including products and related
actions.
 */
public class SellerInventoryScreenView extends JComponent {
        private JButton backButton, addProductButton;
        private JPanel productContainer;

        /**
    * Constructs a SellerInventoryScreenView object.
    * Initializes UI components for displaying the seller's inventory and related buttons.
    */
        public SellerInventoryScreenView() {
                JPanel allContainer = new JPanel();
        allContainer.setLayout(new BorderLayout());

        productContainer = new JPanel(new GridLayout(0, 3, 10, 10));

        JPanel buttonContainer = new JPanel();
```

```
        buttonContainer.setLayout(new BoxLayout(buttonContainer, BoxLayout.X_AXIS));

        backButton = new JButton("Back to Main Menu");
        addProductButton = new JButton("Add Product");

        buttonContainer.add(backButton);
        buttonContainer.add(Box.createRigidArea(new Dimension(10, 0)));
        buttonContainer.add(addProductButton);

        JScrollPane scrollPane = new JScrollPane(productContainer);
        scrollPane.setPreferredSize(new Dimension(1100, 600));
        scrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

        allContainer.add(buttonContainer, BorderLayout.NORTH);
        allContainer.add(scrollPane);

        add(allContainer);

        setLayout(new FlowLayout());
        setVisible(true);
        }

        /**
    * Adds an ActionListener to the back button.
    *
    * @param listener The ActionListener to add to the back button.
    */
        public void addBackButtonListener(ActionListener listener) {
                backButton.addActionListener(listener);
        }

        /**
    * Adds an ActionListener to the add product button.
    *
    * @param listener The ActionListener to add to the add product button.
    */
        public void addAddProductButtonListener(ActionListener listener) {
                addProductButton.addActionListener(listener);
        }

        /**
    * Removes all product views from the product container.
    */
        public void removeAllProductViews() {
```

```
                productContainer.removeAll();
        }

        /**
     * Adds a list of ProductSellerView instances to the product container for display.
     *
     * @param productViews The list of ProductSellerView instances to be added.
     */
        public void addProductViews(List<ProductSellerView> productViews) {
                for (ProductSellerView view : productViews) {
                        productContainer.add(view);
                }
                revalidate();
                repaint();
        }
}
```

ShipmentScreenController.java:

```
package frontend.ShipmentScreen;

import javax.swing.SwingUtilities;

import frontend.LogoutWrapper.LogoutWrapperController;
import frontend.MainScreenCustomer.*;

/**
 * Controller responsible for managing the Shipment Screen view and model.
 * Handles the shipment details display and navigation back to the home screen.
 */
public class ShipmentScreenController {
    private ShipmentScreenModel model;
    private ShipmentScreenView view;

    /**
     * Constructs a ShipmentScreenController object.
     * Initializes the controller with the associated view and model.
     *
     * @param view  The view component for the shipment screen.
     * @param model The model component for managing shipment-related data.
     */
    public ShipmentScreenController(ShipmentScreenView view, ShipmentScreenModel
model) {
        this.model = model;
        this.view = view;
```

```
        view.addHomeButtonListener(e -> onHomeButtonClick());
        displayShipmentDetails();
    }

    /**
     * Displays shipment details asynchronously on the view.
     * Invokes the model to load shipment information and updates the view accordingly.
     */
    public void displayShipmentDetails() {
        SwingUtilities.invokeLater(() -> {
            model.loadShipmentInfo();
            view.setShipmentDetails(model.getShipmentInfo());
        });
    }

    /**
     * Handles the action upon clicking the home button.
     * Navigates back to the main customer screen when the home button is clicked.
     */
    private void onHomeButtonClick() {
        // Return to home screen
            MainScreenCustomerView view = new MainScreenCustomerView();
                    MainScreenCustomerModel model = new
MainScreenCustomerModel(this.model.getCustomer());
                    LogoutWrapperController.Instance.changeMVC(model, view, new
MainScreenCustomerController(view, model));
    }
}
```

ShipmentScreenModel.java:

```
package frontend.ShipmentScreen;

import common.Customer;

/**
 * Manages the shipment-related data for the shipment screen view.
 * Provides methods to load shipment information and retrieve customer data.
 */
public class ShipmentScreenModel {
    private String shipmentInfo;
    private Customer customer;

    /**
     * Constructs a ShipmentScreenModel object.
     * Initializes the shipment information as an empty string and sets the associated customer.
```

```
     *
     * @param customer The customer associated with the shipment.
     */
    public ShipmentScreenModel(Customer customer) {
        this.shipmentInfo = ""; // Empty shipping information
        this.customer = customer;
    }

    /**
     * Simulates loading shipment information.
     * Updates the shipment information data within the model.
     * (In a real application, this method would fetch data from an external source.)
     */
    public void loadShipmentInfo() {
        // Simulate shipment information
        this.shipmentInfo = "Shipment ID: \nStatus: ";
    }

    /**
     * Retrieves the current shipment information.
     *
     * @return The shipment information as a string.
     */
    public String getShipmentInfo() {
        return shipmentInfo;
    }

    /**
     * Retrieves the associated customer.
     *
     * @return The customer associated with the shipment.
     */
    public Customer getCustomer() {
            return customer;
    }
}
```

ShipmentScreenView.java:

```
package frontend.ShipmentScreen;

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

/**
```

```
 * Represents the graphical view for the shipment screen.
 * Displays shipment details and provides a button to navigate back to the home screen.
 */
public class ShipmentScreenView extends JPanel {
    private JTextArea shipmentDetailsTextArea;
    private JButton homeButton;

    /**
     * Constructs a ShipmentScreenView object.
     * Sets up the graphical components, such as a text area to display shipment details and a
button to navigate home.
     */
    public ShipmentScreenView() {
        shipmentDetailsTextArea = new JTextArea(10, 20);
        shipmentDetailsTextArea.setEditable(false);
        JScrollPane scrollPane = new JScrollPane(shipmentDetailsTextArea);

        homeButton = new JButton("Home");

        setLayout(new BorderLayout());
        add(scrollPane, BorderLayout.CENTER);
        add(homeButton, BorderLayout.SOUTH);
    }

    /**
     * Sets the displayed shipment details in the text area.
     *
     * @param details The shipment details to be displayed.
     */
    public void setShipmentDetails(String details) {
        shipmentDetailsTextArea.setText(details);
    }

    /**
     * Adds an ActionListener to the home button.
     * Invokes the provided ActionListener when the home button is clicked.
     *
     * @param listener The ActionListener to be added to the home button.
     */
    public void addHomeButtonListener(ActionListener listener) {
        homeButton.addActionListener(listener);
    }
}
```

ShoppingCartScreenController.java:

```java
package frontend.ShoppingCartScreen;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.LinkedList;
import java.util.List;
import common.Product;
import frontend.Product.*;
import frontend.LogoutWrapper.LogoutWrapperController;
import frontend.MainScreenCustomer.*;
import frontend.PaymentScreen.*;

/**
 * Controls the interactions between the ShoppingCartScreenView and ShoppingCartScreenModel.
 * Manages the display of products in the shopping cart, handles cart changes, and navigation options.
 */
public class ShoppingCartScreenController {
	private ShoppingCartScreenView view;
	private ShoppingCartScreenModel model;
	private List<ProductController> pControllers;
	private ActionListener cartChangedAction;

	/**
    * Constructs a ShoppingCartScreenController.
    * Initializes the view, model, and sets up listeners for cart updates and navigation buttons.
    *
    * @param view  The view representing the shopping cart screen.
    * @param model The model containing shopping cart information.
    */
	public ShoppingCartScreenController(ShoppingCartScreenView view, ShoppingCartScreenModel model) {
		this.view = view;
		this.model = model;

		List<ProductView> views = new LinkedList<ProductView>();
		pControllers = new LinkedList<ProductController>();

		for (Product product : model.getCustomer().getShoppingCart().getProducts()) {
			ProductView pView = new ProductView();
			views.add(pView);

			pView.setTitleLabelText(product.getName());
```

```
                        pView.setDescriptionLabelText(product.getDescription());
                        pView.setPriceLabelText(String.format("$%.2f", product.getPrice()));

                        ProductModel pModel = new ProductModel(product,
model.getCustomer().getShoppingCart());

                        pView.setCountLabelText(Integer.toString(pModel.getCount()) + " item(s)
in cart.");

                        ProductController pController = new ProductController(pView, pModel);
                        pControllers.add(pController);
                }

                view.addProductViews(views);

                view.addBackButtonListener(e -> onBackButtonClick());
                view.addCheckoutButtonListener(e -> onCheckoutButtonClick());

                cartChangedAction = new ActionListener() {
                        @Override
                        public void actionPerformed(ActionEvent e) {
                                updatePriceTotalLabelText();
                        }
                };

model.getCustomer().getShoppingCart().addCartUpdatedListener(cartChangedAction);

                updatePriceTotalLabelText();
        }

        /**
    * Handles the action when the back button is clicked.
    * Removes the cart update listener and navigates back to the main customer screen.
    */
        private void onBackButtonClick() {

model.getCustomer().getShoppingCart().removeCartUpdatedListener(cartChangedAction);

                MainScreenCustomerView view = new MainScreenCustomerView();
                MainScreenCustomerModel model = new
MainScreenCustomerModel(this.model.getCustomer());
                LogoutWrapperController.Instance.changeMVC(model, view, new
MainScreenCustomerController(view, model));
        }
```

```
        /**
     * Handles the action when the checkout button is clicked.
     * Removes the cart update listener and navigates to the payment screen.
     */
        private void onCheckoutButtonClick() {

model.getCustomer().getShoppingCart().removeCartUpdatedListener(cartChangedAction);

                PaymentScreenView view = new PaymentScreenView();
                PaymentScreenModel model = new
PaymentScreenModel(this.model.getCustomer());
                LogoutWrapperController.Instance.changeMVC(model ,view, new
PaymentScreenController(view, model));
        }

        /**
     * Updates the total price label text on the shopping cart screen.
     * Displays the current total price of items in the cart.
     */
        private void updatePriceTotalLabelText() {
                view.setPriceTotalLabelText(String.format("Total: $%.2f",
model.getCustomer().getShoppingCart().getCartTotal()));
        }
}
```

ShoppingCartScreenModel.java:

```
package frontend.ShoppingCartScreen;

import common.*;

/**
 * Manages the shopping cart data for the ShoppingCartScreen.
 * Stores the customer's shopping cart information.
 */
public class ShoppingCartScreenModel {
    private Customer customer;

    /**
     * Constructs a ShoppingCartScreenModel.
     *
     * @param customer The customer whose shopping cart is being managed.
     */
    public ShoppingCartScreenModel(Customer customer) {
        this.customer = customer;
```

```
    }

    /**
     * Retrieves the customer associated with this shopping cart.
     *
     * @return The customer object.
     */
    public Customer getCustomer() {
        return customer;
    }
}
```

ShoppingCartScreenView.java:

```
package frontend.ShoppingCartScreen;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.util.List;
import javax.swing.*;
import frontend.Product.ProductView;

/**
 * Represents the view for the shopping cart screen.
 * Displays the list of products in the shopping cart and provides options for navigation and checkout.
 */
public class ShoppingCartScreenView extends JComponent {
        private JPanel productContainer;
        private JButton backButton, checkoutButton;
        private JLabel priceTotalLabel;

        /**
     * Constructs a ShoppingCartScreenView and initializes its components.
     * Sets up the layout and components for displaying products, navigation buttons, and the total price label.
     */
    public ShoppingCartScreenView() {
        JPanel allContainer = new JPanel();
        allContainer.setLayout(new BorderLayout());

        productContainer = new JPanel(new GridLayout(0, 3, 10, 10));
```

```
        backButton = new JButton("Back to Main Menu");

        JPanel checkoutContainer = new JPanel(new BorderLayout());

        checkoutButton = new JButton("Proceed to Checkout");
        priceTotalLabel = new JLabel("Total: $0.00");

        checkoutContainer.add(priceTotalLabel, BorderLayout.NORTH);
        checkoutContainer.add(checkoutButton);

        JScrollPane scrollPane = new JScrollPane(productContainer);
        scrollPane.setPreferredSize(new Dimension(1100, 400));
        scrollPane.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

        allContainer.add(backButton, BorderLayout.NORTH);
        allContainer.add(checkoutContainer, BorderLayout.SOUTH);
        allContainer.add(scrollPane);

        add(allContainer);

        setLayout(new FlowLayout());
        setVisible(true);
    }

    /**
     * Adds a single product view to the product container.
     *
     * @param productView The view of the product to be added.
     */
    public void addProductView(ProductView productView) {
        productContainer.add(productView);
        revalidate();
        repaint();
    }

    /**
     * Adds multiple product views to the product container.
     *
     * @param productViews The list of product views to be added.
     */
    public void addProductViews(List<ProductView> productViews) {
        for (ProductView view : productViews) {
                productContainer.add(view);
        }
```

```
        revalidate();
        repaint();
    }

    /**
     * Removes all products from the product container.
     */
    public void resetProductContainer() {
        productContainer.removeAll();
    }

    /**
     * Adds a listener for the back button.
     *
     * @param listener The ActionListener for the back button.
     */
    public void addBackButtonListener(ActionListener listener) {
        backButton.addActionListener(listener);
    }

    /**
     * Adds a listener for the checkout button.
     *
     * @param listener The ActionListener for the checkout button.
     */
    public void addCheckoutButtonListener(ActionListener listener) {
        checkoutButton.addActionListener(listener);
    }

    /**
     * Sets the text for the price total label.
     *
     * @param text The text to be set for the price total label.
     */
    public void setPriceTotalLabelText(String text) {
        priceTotalLabel.setText(text);
    }
}
```

SignUpController.java:

```
package frontend.SignUp;

import javax.swing.SwingUtilities;

import frontend.MainWindow;
```

```java
import frontend.Login.*;
import frontend.LogoutWrapper.*;
import frontend.MainScreenCustomer.*;
import frontend.MainScreenSeller.*;
import common.*;

/**
 * Controller responsible for handling the sign-up process, creating user accounts,
 * and managing the view interactions for signing up.
 */
public class SignUpController {
    SignUpModel model;
    SignUpView view;

    /**
     * Constructs a SignUpController with the associated SignUpModel and SignUpView.
     * Registers listeners for sign-up and login actions in the provided view.
     *
     * @param model The SignUpModel associated with this controller.
     * @param view  The SignUpView associated with this controller.
     */
    public SignUpController(SignUpModel model, SignUpView view) {
        this.model = model;
        this.view = view;

        view.addSignUpButtonListener(e -> onSignUpButtonClick());

        view.addLoginButtonListener(e -> onLoginButtonClick());
    }

    /**
     * Action to be performed when the sign-up button is clicked.
     * Initiates the sign-up process, creating a user account based on the provided information.
     * Completes the sign-up action asynchronously and switches to the appropriate main screen
upon success.
     */
    private void onSignUpButtonClick() {
        view.showBuffering();
        User user = view.getUserType().equals("Customer") ?
                    new Customer(view.getUsername(), view.getPassword()) :
                    new Seller(view.getUsername(), view.getPassword());

        model.register(user)
            .whenComplete((response, throwable) -> {
```

```
            SwingUtilities.invokeLater(() -> {
                if (response.verified) {
                    System.out.println("Successfully signed up!");

                    // TODO:
                                                //      - Create LogoutWrapper
                                                //      - Get user type from Model, then create
either:
                                                //              - MainScreenCustomer
                                                //              - MainScreenSeller
                                                //      - Switch window using
MainWindow.changeMVC
                                                LogoutWrapperView lgwView = new
LogoutWrapperView();
                                                LogoutWrapperModel lgwModel = new
LogoutWrapperModel();
                                                LogoutWrapperController lgwController = new
LogoutWrapperController(lgwView, lgwModel);

                                                if (response.user.getClass() == Customer.class) {
                                                        // Customer
                                                        MainScreenCustomerView mscView = new
MainScreenCustomerView();
                                                        MainScreenCustomerModel mscModel =
new MainScreenCustomerModel((Customer) user);
                                                        MainScreenCustomerController
mscController = new MainScreenCustomerController(mscView, mscModel);

                                                        lgwController.changeMVC(mscModel,
mscView, mscController);
                                                } else {
                                                        // Seller
                                                        Seller seller = (Seller) user;
                                                        seller.addProduct(new Product("test 1",
"lol", 0.0));
                                                        seller.addProduct(new Product("test 2",
"lol", 10.0));
                                                        seller.addProduct(new Product("test 3",
"lol", 20.0));
                                                        seller.addProduct(new Product("test 4",
"lol", 30.0));

                                                        MainScreenSellerView mssView = new
MainScreenSellerView();
```

```
                                                    MainScreenSellerModel mssModel = new
MainScreenSellerModel(seller);
                                                    MainScreenSellerController mssController =
new MainScreenSellerController(mssView, mssModel);

                                                    lgwController.changeMVC(mssView,
mssView, mssController);
                                                }

                                                MainWindow.Instance.changeMVC(lgwModel,
lgwView, lgwController);
                } else {
                    System.out.println("Failed to sign up!");
                    if (throwable != null)
                        view.displaySignUpFailure(throwable.toString());
                    else
                        view.displaySignUpFailure("Failed to sign up!");
                }
                view.hideBuffering();
            });
        });
    }

    /**
     * Action to be performed when the login button is clicked.
     * Navigates the user to the login view to sign in after attempting to sign up.
     */
    private void onLoginButtonClick() {
        // navigate to Login
        LoginView loginView = new LoginView();
        LoginModel loginModel = new LoginModel();
        LoginController loginController = new LoginController(loginModel, loginView);

        MainWindow.Instance.changeMVC(loginModel, loginView, loginController);
    }
}
```

SignUpModel.java:

```
package frontend.SignUp;

import java.util.concurrent.CompletableFuture;
import common.*;

public class SignUpModel {
```

```
        /**
         * Registers the provided user asynchronously.
         *
         * @param user The User object containing username and password to be registered.
         * @return A CompletableFuture<VerifyCredentialsResponse> representing the
asynchronous registration process.
         *         The CompletableFuture will complete with a VerifyCredentialsResponse
indicating success or failure.
         */
        public CompletableFuture<VerifyCredentialsResponse> register(User user) {
    return CompletableFuture.supplyAsync(() -> {
        try {
            // Simulating a delay for connection and verification
            Thread.sleep(2000);
            return new VerifyCredentialsResponse(true, user);
        } catch (InterruptedException e) {
            e.printStackTrace();
            return new VerifyCredentialsResponse(false, user);
        }
    });
  }
}
```

SignUpView.java:

```
package frontend.SignUp;

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

public class SignUpView extends JComponent {
  private JTextField usernameField;
  private JPasswordField passwordField;
  private JButton signUpButton;
  private JButton loginButton;
  private JProgressBar progressBar;
  private JFrame failureFrame;
  private JToggleButton userTypeButton;

  /**
   * Constructs the SignUpView UI, consisting of input fields, buttons, and panels
   * to facilitate user registration.
   */
```

```
    public SignUpView() {
        usernameField = new JTextField(20);
        passwordField = new JPasswordField(20);
        signUpButton = new JButton("Sign up");
        loginButton = new JButton("Return to login");

        userTypeButton = new JToggleButton("Customer");
        userTypeButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                if (userTypeButton.isSelected()) {
                        userTypeButton.setText("Seller");
                } else {
                        userTypeButton.setText("Customer");
                }
            }
        });

        JPanel inputPanel = new JPanel(new GridLayout(0, 2, 5, 5)); //0 rows for variable number
of components
        inputPanel.add(new JLabel("Username:"));
        inputPanel.add(usernameField);
        inputPanel.add(new JLabel("Password:"));
        inputPanel.add(passwordField);
        inputPanel.add(new JLabel("User type:"));
        inputPanel.add(userTypeButton);

        JPanel buttonPanel = new JPanel();
        buttonPanel.add(signUpButton);
        buttonPanel.add(loginButton);

        JPanel mainPanel = new JPanel();
        mainPanel.setLayout(new BorderLayout());
        mainPanel.setPreferredSize(new Dimension(300, 150)); //Set size

        mainPanel.add(inputPanel, BorderLayout.CENTER);
        mainPanel.add(buttonPanel, BorderLayout.SOUTH);

        this.setLayout(new GridBagLayout()); //Use GBL for SignUpView
        GridBagConstraints gbc = new GridBagConstraints();
        gbc.gridx = gbc.gridy = 0;
        gbc.weightx = gbc.weighty = 1;
        gbc.fill = GridBagConstraints.BOTH; //fill blank space
        this.add(mainPanel, gbc);
```

```
}

/**
 * Displays a window when the sign-up process fails, showing an error message.
 *
 * @param failureMessage The message to display on the failure frame.
 */
public void displaySignUpFailure(String failureMessage) {
    if (failureFrame != null) {
        failureFrame.dispose();
    }

    failureFrame = new JFrame("Sign Up Failed!");
    failureFrame.setSize(200, 120);
    failureFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    JPanel panel = new JPanel(new FlowLayout(FlowLayout.CENTER));
    JLabel label = new JLabel(failureMessage);
    label.setFont(label.getFont().deriveFont(Font.PLAIN, 18f));
    label.setHorizontalAlignment(JLabel.CENTER);
    panel.add(label);

    Box verticalBox = Box.createVerticalBox();
    verticalBox.add(Box.createVerticalGlue());
    verticalBox.add(panel);
    verticalBox.add(Box.createVerticalGlue());

    failureFrame.add(verticalBox, BorderLayout.CENTER);

    failureFrame.setVisible(true);
}

/**
 * Displays a buffering symbol to indicate that the sign-up process is ongoing.
 */
public void showBuffering() {
    progressBar = new JProgressBar();
    progressBar.setIndeterminate(true);
    progressBar.setString("Signing up...");
    progressBar.setStringPainted(true);
    progressBar.setPreferredSize(new Dimension(200, 30));

    int x = (this.getWidth() - progressBar.getPreferredSize().width) / 2;
    int y = (this.getHeight() - progressBar.getPreferredSize().height) / 2;
```

```
        progressBar.setBounds(x, y, progressBar.getPreferredSize().width,
progressBar.getPreferredSize().height);
        this.add(progressBar);

        this.setVisible(true);
    }

    /**
     * Hides the buffering symbol when the sign-up process is complete.
     */
    public void hideBuffering() {
        this.remove(progressBar);
        this.setVisible(true);
        this.revalidate();
        this.repaint();
    }

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

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

    /**
     * Retrieves the type of user selected (Customer or Seller) from the toggle button.
     *
     * @return The type of user selected.
     */
    public String getUserType() {
            return userTypeButton.getText();
    }
```

```
    /**
     * Adds an ActionListener to the login button.
     *
     * @param listener The ActionListener to add to the login button.
     */
        public void addLoginButtonListener(ActionListener listener) {
            loginButton.addActionListener(listener);
        }

        /**
         * Adds an ActionListener to the sign-up button.
         *
         * @param listener The ActionListener to add to the sign-up button.
         */
    public void addSignUpButtonListener(ActionListener listener) {
        signUpButton.addActionListener(listener);
    }
}
```

MVC/UML Diagrams:

IWrapper (interface)

- Contains visual elements
- Methods:
    - public static void ChangeMVC(Object model, Object controller, JComponent view)
        - Changes the current MVC components to the MVC specified in the parameters

MainWindow implements IWrapper

- Static root window component (Singleton)

Login screen

LoginView

Responsibilities:

- Contain input elements
- Display login information
- Display login failure
- Display buffering

LoginController

Responsibilities:

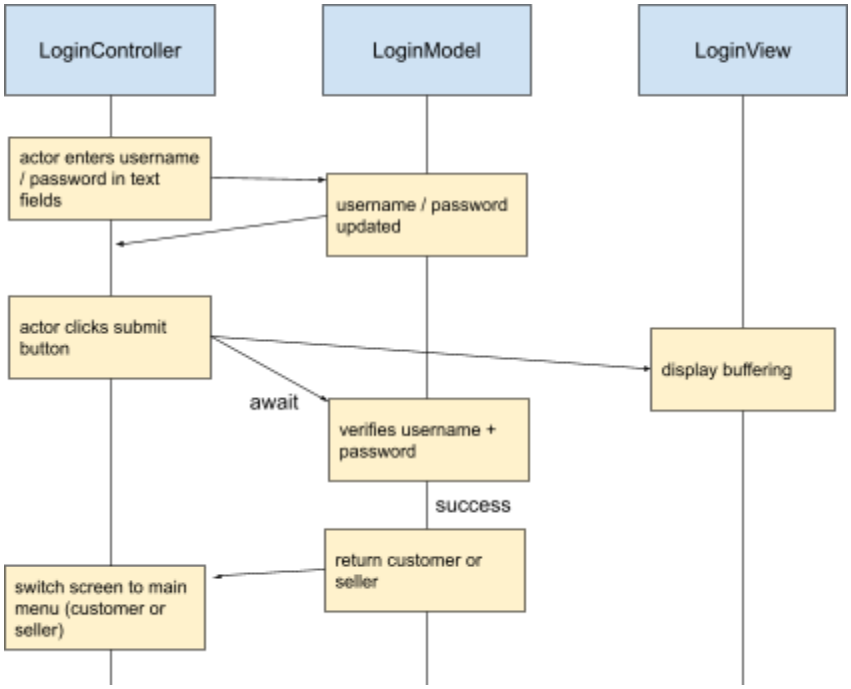
- Handle username/password input
- Handle login submission
- Handle login success / failure
- Handle sign up button clicked

LoginModel

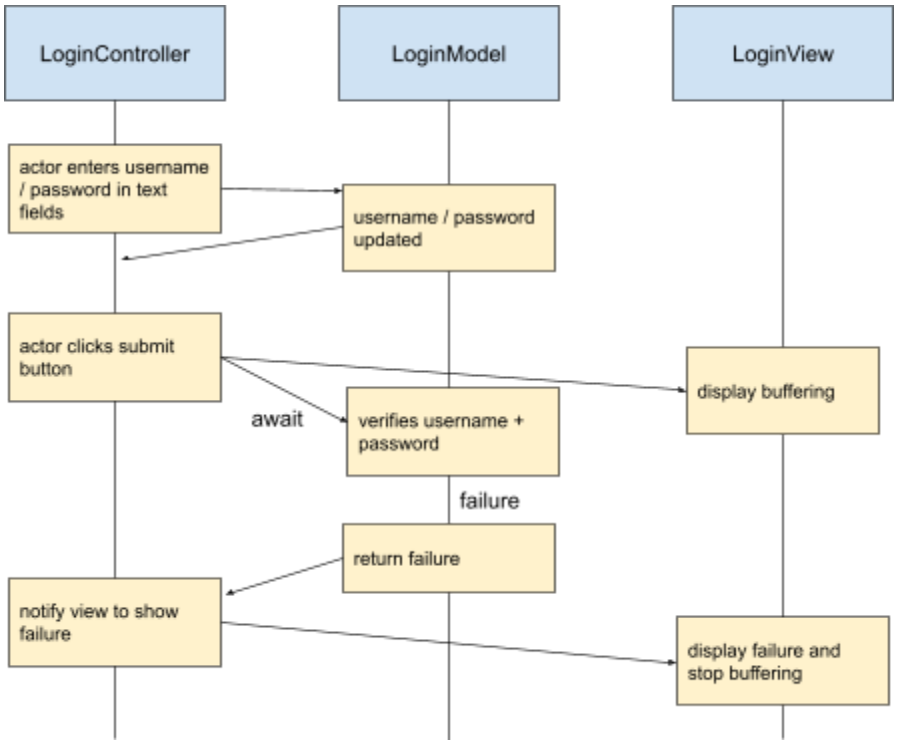
Responsibilities:

- Verify username/password

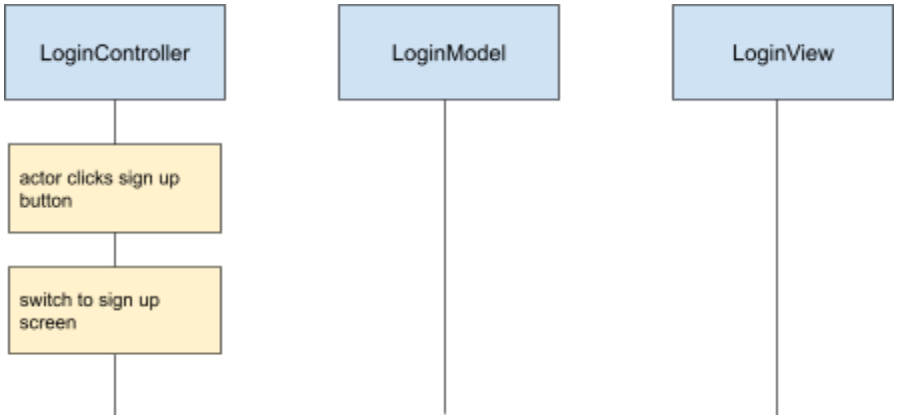
Use case: Actor logs in (success).



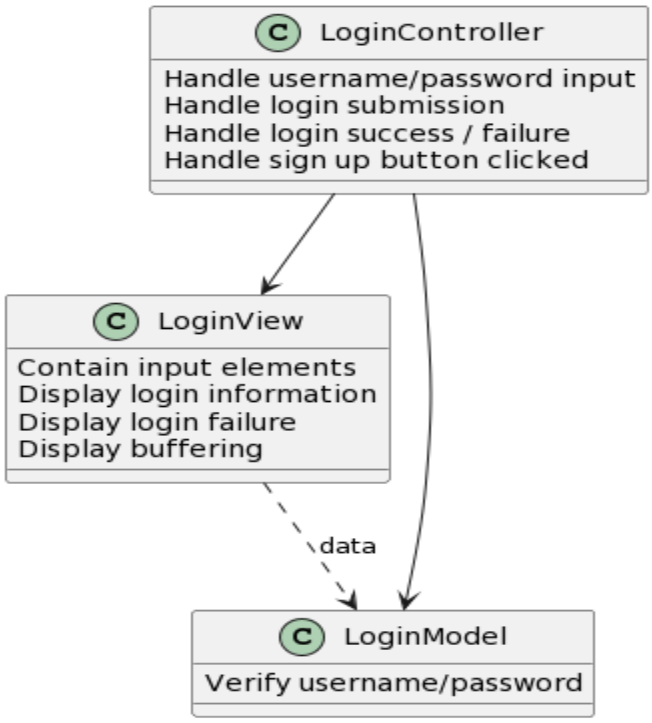
Use case: Actor logs in (failure).



Use case: Actor clicks sign up button.



UML:



Sign up screen

SignUpView

Responsibilities:

- Contain input elements
- Display sign up information
- Display sign up failure
- Display buffering

SignUpController

Responsibilities:

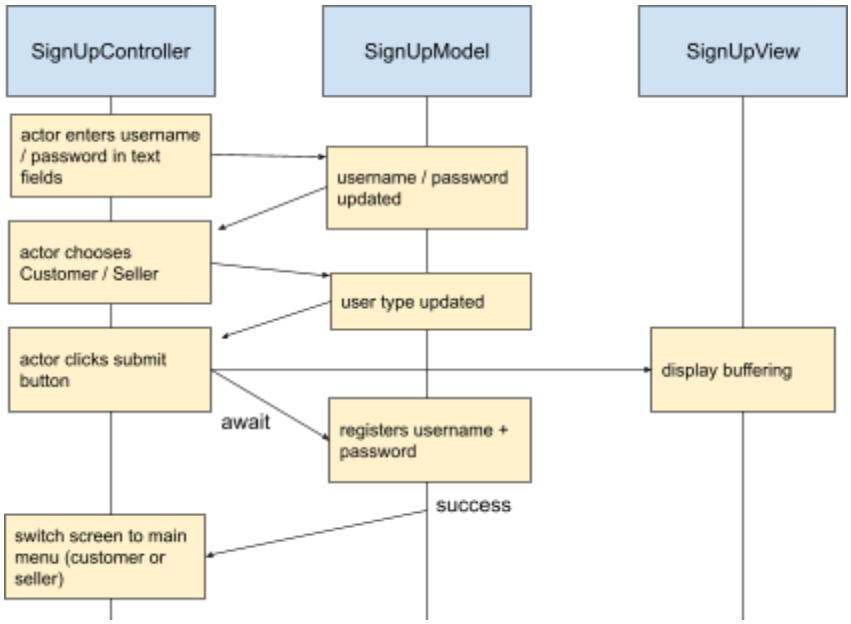
- Handle username/password input
- Handle type of user input (customer/seller)
- Handle sign up submission
- Handle sign up success / failure
- Handle login button clicked

SignUpModel

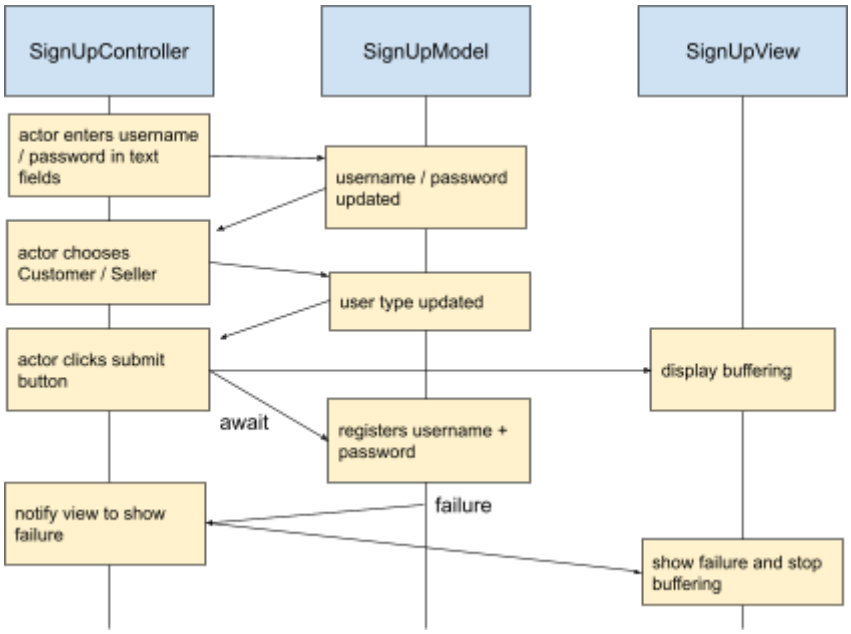
Responsibilities:

- Register username/password

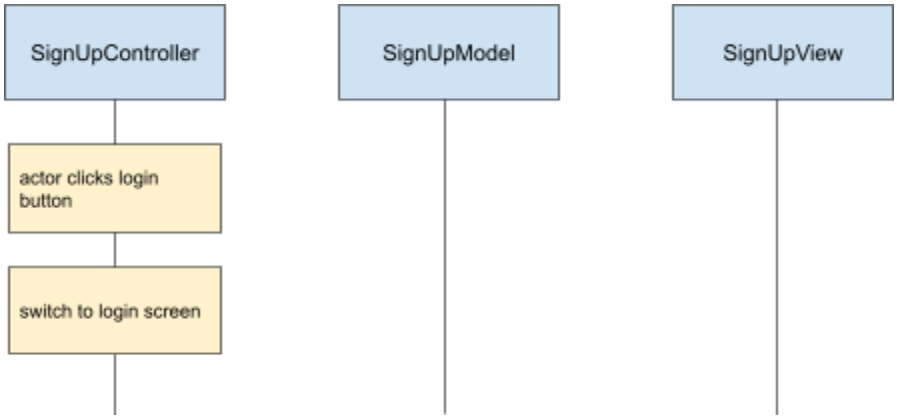
Use case: Actor signs up (success)



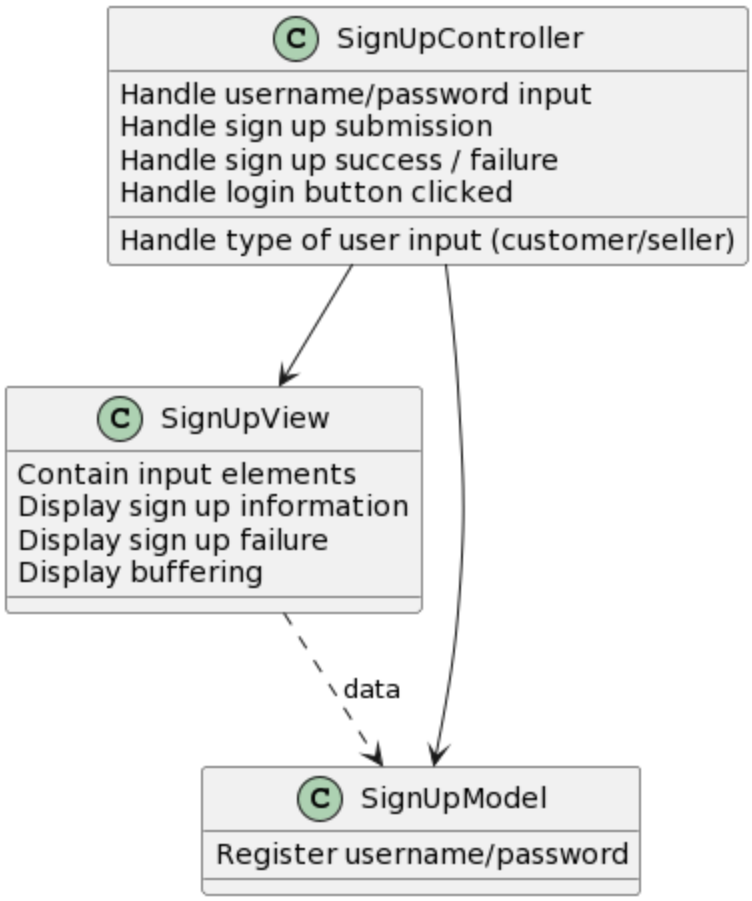
Use case: Actor signs up (failure)



Use case: Actor clicks login button.



UML:


SignUpController
Handle username/password input
Handle sign up submission
Handle sign up success / failure
Handle login button clicked
Handle type of user input (customer/seller)
SignUpView
Contain input elements
Display sign up information
Display sign up failure
Display buffering
data
SignUpModel
Register username/password

Main screen (Customer)

MainScreenCustomerView

Responsibilities:

- Display relevant UI elements
- Handle displaying a list of Products

MainScreenCustomerController

Responsibilities:

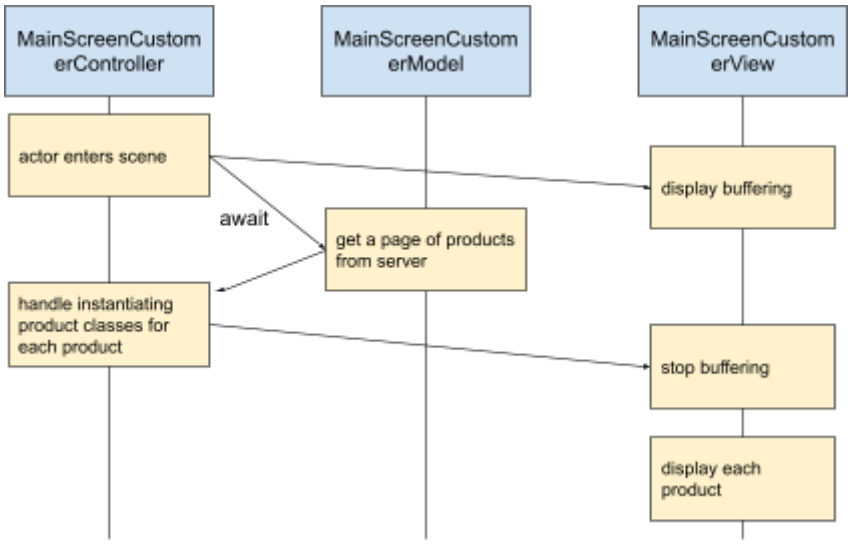
- Handle shopping cart button
- Handle instantiating Products
  - ProductView(s)
  - ProductModel(s)
  - ProductController(s)
- Handle next page / previous page buttons

MainScreenCustomerModel

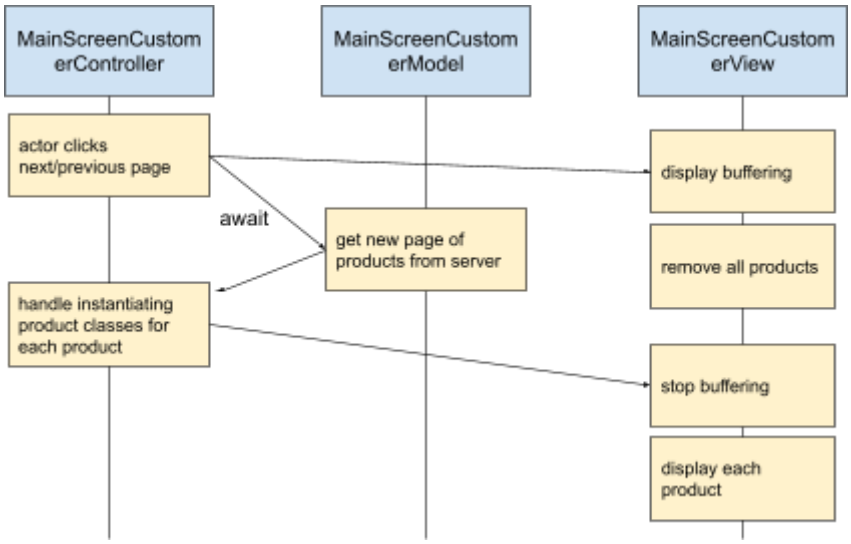
Responsibilities:

- Load product entries

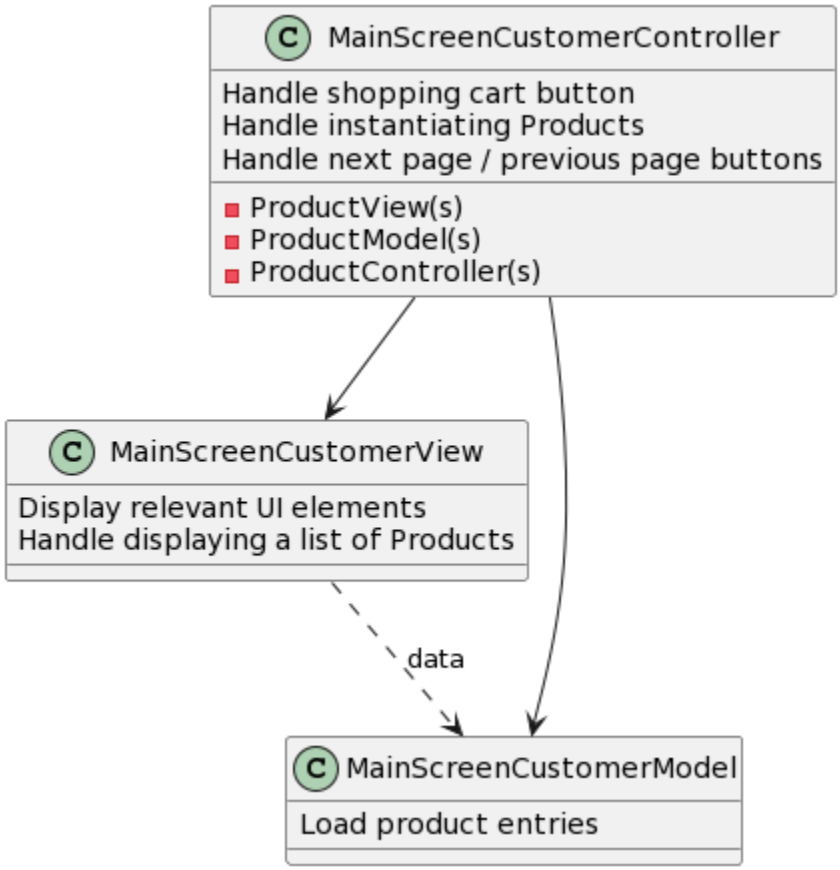
Use case: Actor enters Main screen (Customer)



Use case: Actor clicks next/previous page



UML:



Product

ProductView

Responsibilities:

- Display relevant UI elements
- Display based upon when cart contains product or not

ProductController

Responsibilities:

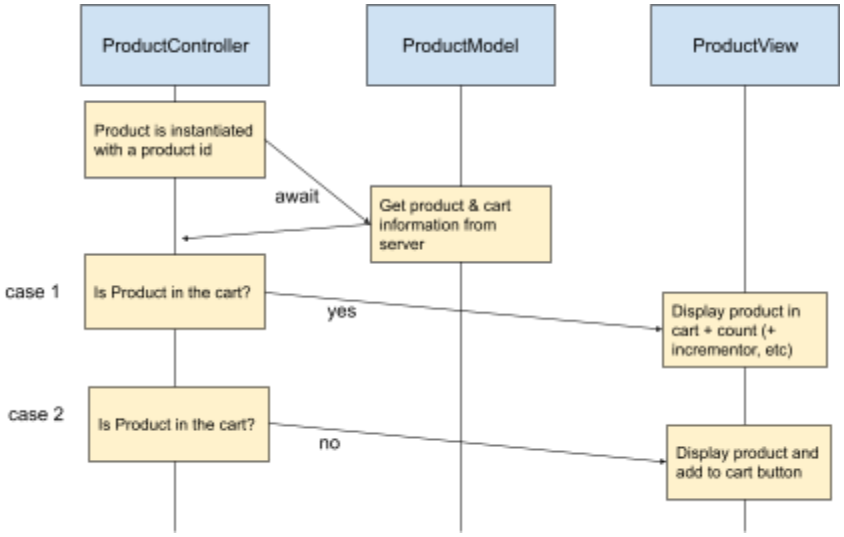
- Handle adding a product to the cart
- Handle removing a product from the cart
- Handle incrementing/decrementing number of a product

ProductModel

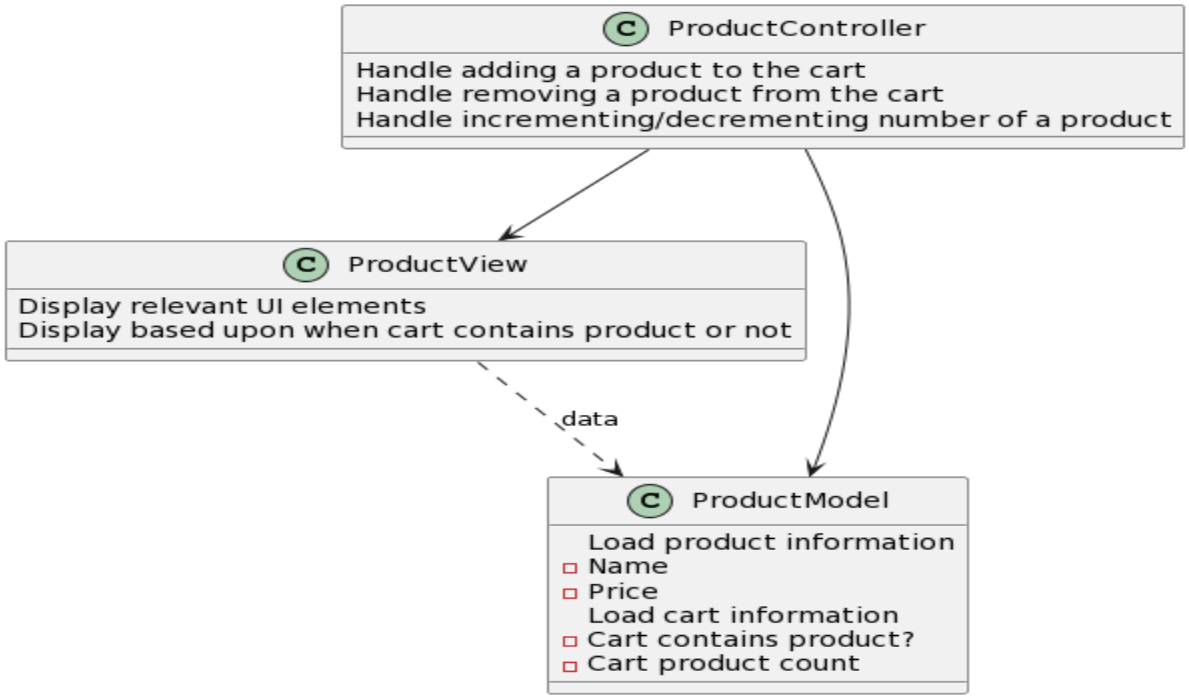
Responsibilities:

- Load product information
  - Name
  - Price
- Load cart information
  - Cart contains product?
  - Cart product count

Use case: Product gets instantiated



UML:



Main screen (Seller)

MainScreenSellerView

Responsibilities:

- Display relevant UI elements
- Contain:
  - SellerInventoryScreenView
  - FinancialReportsScreenView

MainScreenSellerController

Responsibilities:

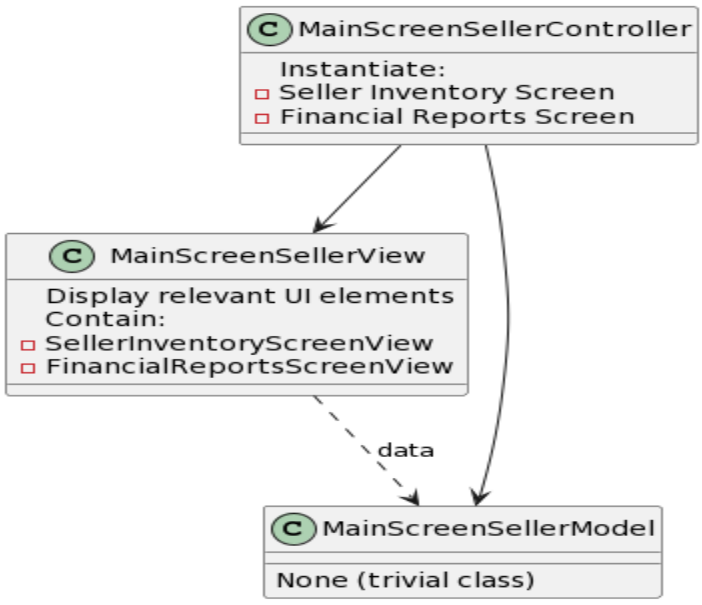
- Instantiate:
  - Seller Inventory Screen
  - Financial Reports Screen

MainScreenSellerModel

Responsibilities:

- None (trivial class)

UML:



Shopping Cart Screen

ShoppingCartScreenView

Responsibilities:

- Display relevant UI
- Display a list of Products vertically

ShoppingCartScreenController

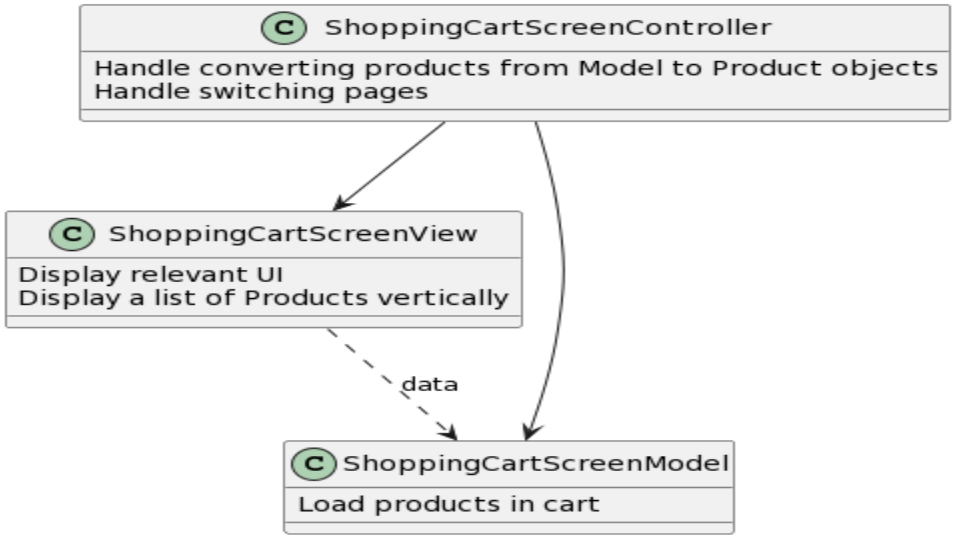
Responsibilities:

- Handle converting products from Model to Product objects
- Handle switching pages

ShoppingCartScreenModel

Responsibilities:

- Load products in cart



Shipment Screen

ShipmentScreenView

Responsibilities:

- Display relevant UI
  - Home button
  - Shipment details

ShipmentScreenController

Responsibilities:

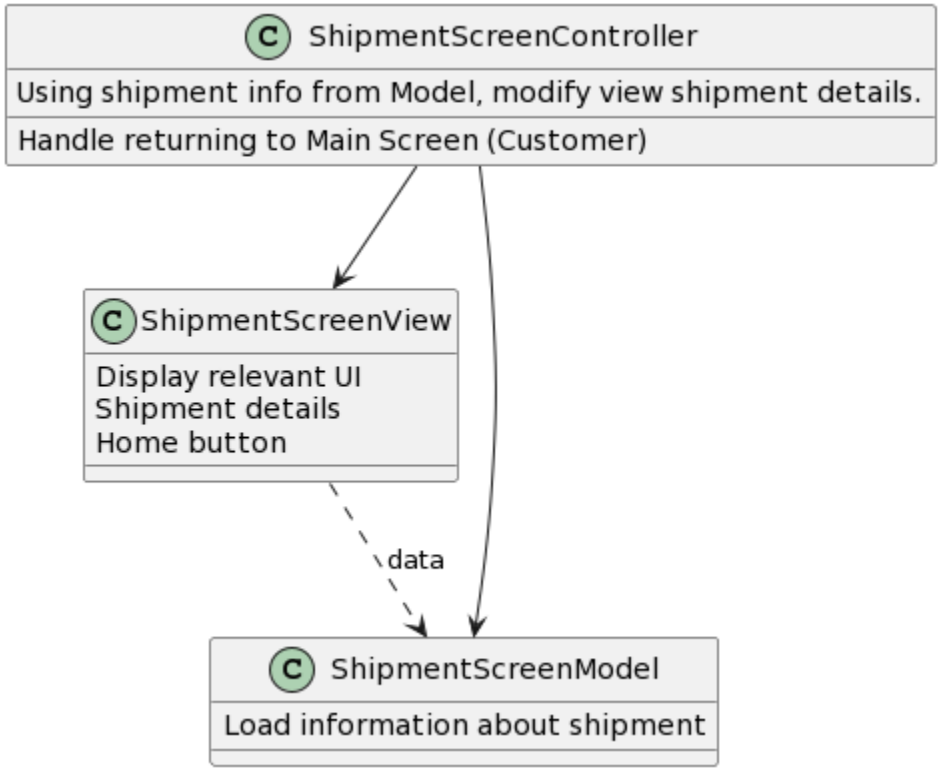
- Using shipment info from Model, modify view shipment details.
- Handle returning to Main Screen (Customer)

ShipmentScreenModel

Responsibilities:

- Load information about shipment

UML:



Payment Screen

PaymentScreenView

Responsibilities:

- Display relevant UI elements
  - Credit card input

PaymentScreenController

Responsibilities:

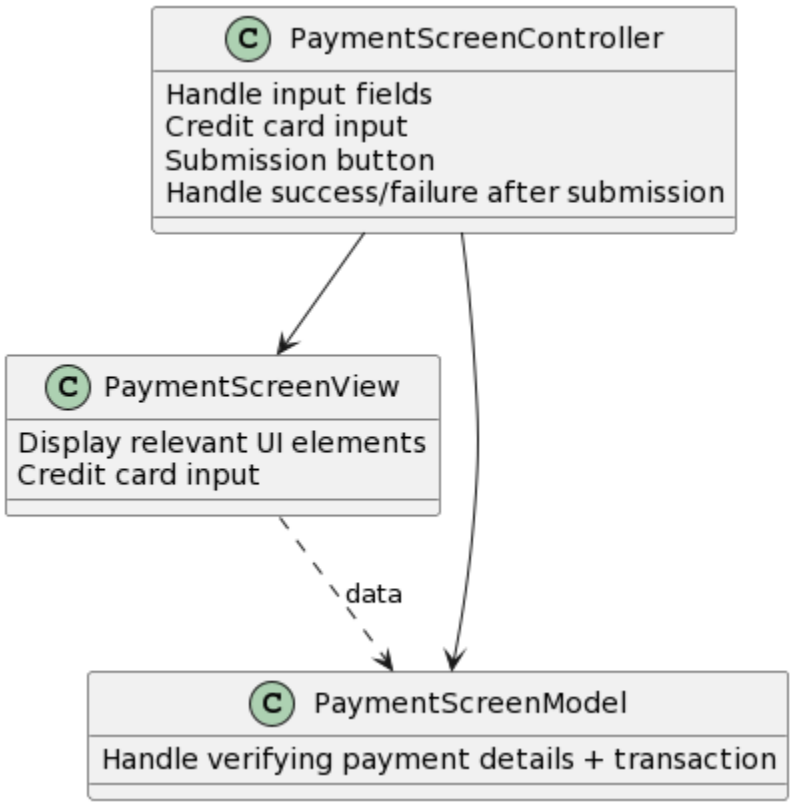
- Handle input fields
  - Credit card input
  - Submission button
- Handle success / failure after submission

PaymentScreenModel

Responsibilities:

- Handle verifying payment details + transaction

UML:



Seller Inventory Screen

SellerInventoryScreenView

Responsibilities:
- Contain a list of ProductSellerViews

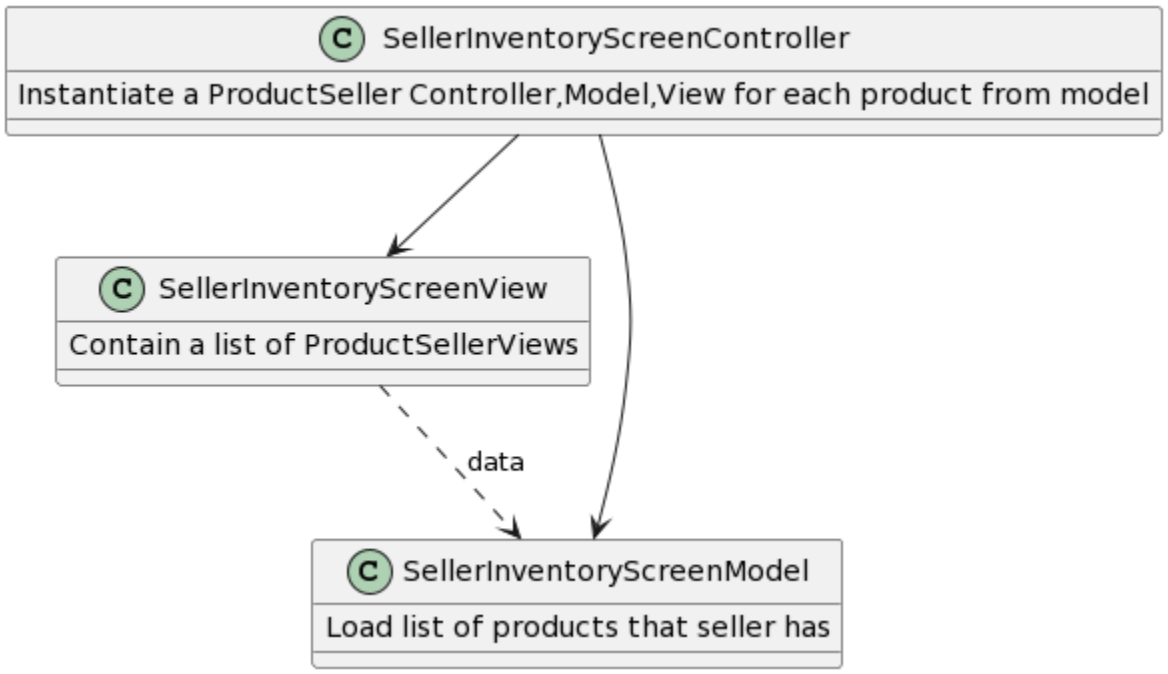
SellerInventoryScreenController

Responsibilities:
- Instantiate a ProductSeller Controller,Model,View for each product from model

SellerInventoryScreenModel

Responsibilities:
- Load list of products that seller has

UML:


SellerInventoryScreenController
Instantiate a ProductSeller Controller,Model,View for each product from model
SellerInventoryScreenView
Contain a list of ProductSellerViews
data
SellerInventoryScreenModel
Load list of products that seller has

Product (Seller)

ProductSellerView

Responsibilities:

- Display relevant UI elements
  - Name
  - Price
  - Sales
  - Total profit (= price * sales)
- Save details button

ProductSellerController

Responsibilities:

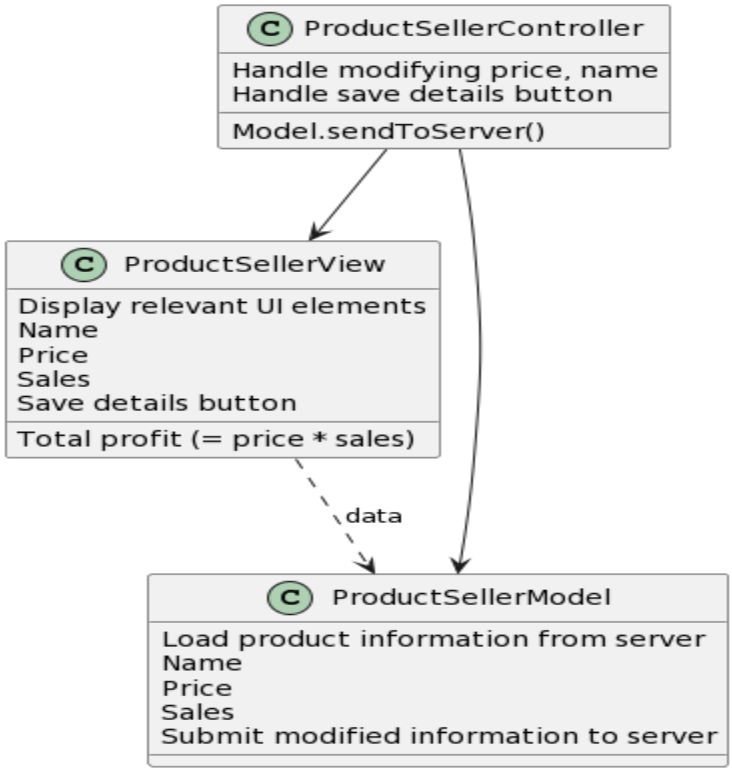
- Handle modifying price, name
- Handle save details button
  - Model.sendToServer()

ProductSellerModel

Responsibilities:

- Load product information from server
  - Name
  - Price
  - Sales
- Submit modified information to server

UML:



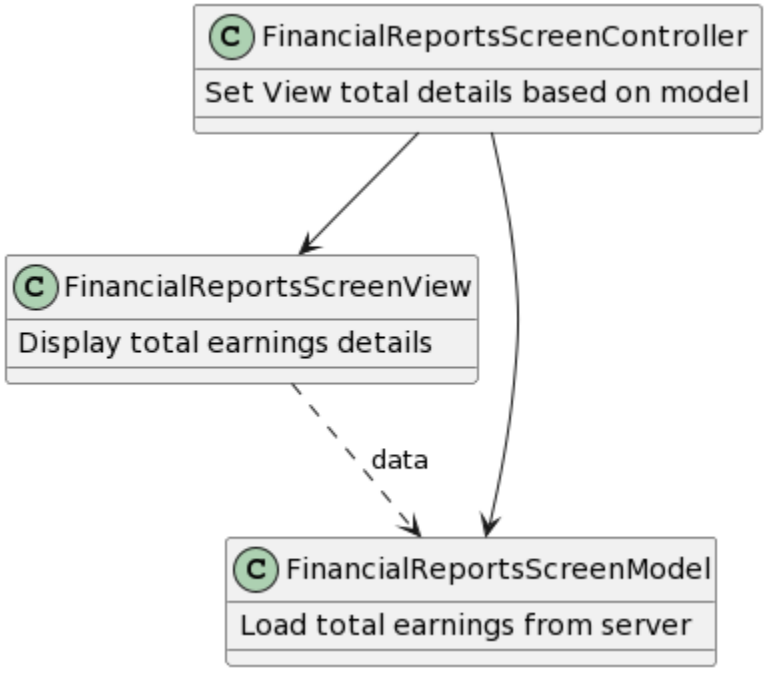
Financial Reports Screen

FinancialReportsScreenView
Responsibilities:
- Display total earnings detials

FinancialReportsScreenController
Responsibilities:
- Set View total details based on model

FinancialReportsScreenModel
Responsibilities:
- Load total earnings from server

UML:



Logout Wrapper

LogoutWrapperView

Responsibilities:
- Contain other views internally
- Show a logout button

LogoutWrapperController

Responsibilities:
- Handle logout button
  - Return to login screen
- Handle switching internal views

LogoutWrapperModel

Responsibilities:
- Erase current user information from memory

UML: