RIPHAH INTERNATIONAL UNIVERSITY, GULBERG GREENS, ISLAMABAD

Faculty of computing

Software Construction & Development Project

BSSE Semester no 5

Submitted to: Mam Kausar

Submitted by:

Sap ID	Name	Email
44937	Amna Jamil	amnajamil@gmail.com
47235	Sabahat Qadeer	Sabahatqadeer@gmail.com
46481	Maryam Safdar	maryamsafdar@gmail.com
47054	Nimra Gul	nimragul@gmail.com

Project Title:

Smart Inventory and Order Management System (SIOMS)

Project Overview:

This system enables businesses to efficiently manage inventory, handle orders, and track deliveries. Different user roles include **Admin**, **Supplier**, and **Customer**. Suppliers manage inventory and restock items as needed, while customers can place orders and track their delivery status. The admin oversees operations, manages user roles, and generates sales and stock reports. The system architecture should use the MVC framework for structure and follow GRASP principles to ensure well-defined responsibilities.

Project Deliverables Using Topics:

Use Case Diagram:

❖ Design a use case diagram with key actors and interactions.

The following diagram represents a use case model for a "Smart Inventory and Order Management System (SIOMS)" and includes the interactions between the system and its actors.

Actors:

- **Admin (Secondary Actor):** Manages users and overall system configurations.
- **Supplier (Primary Actor):** Handles inventory management and order fulfillment.
- **Customer (Primary Actor):** Interacts with the system for product-related actions and order processing.

Key Use Cases:

1. Login: Required for all actors to access the system functionalities.
2. Manage User (Admin):

- Add/Delete User: Manage system users (e.g., Suppliers and Customers).

3. Inventory Management (Supplier and Admin):

- Add, Edit, Remove Product: Manage product details in the inventory.
- Track Stock Level: Monitor inventory levels.
- Restock Product: Replenish stock when necessary.

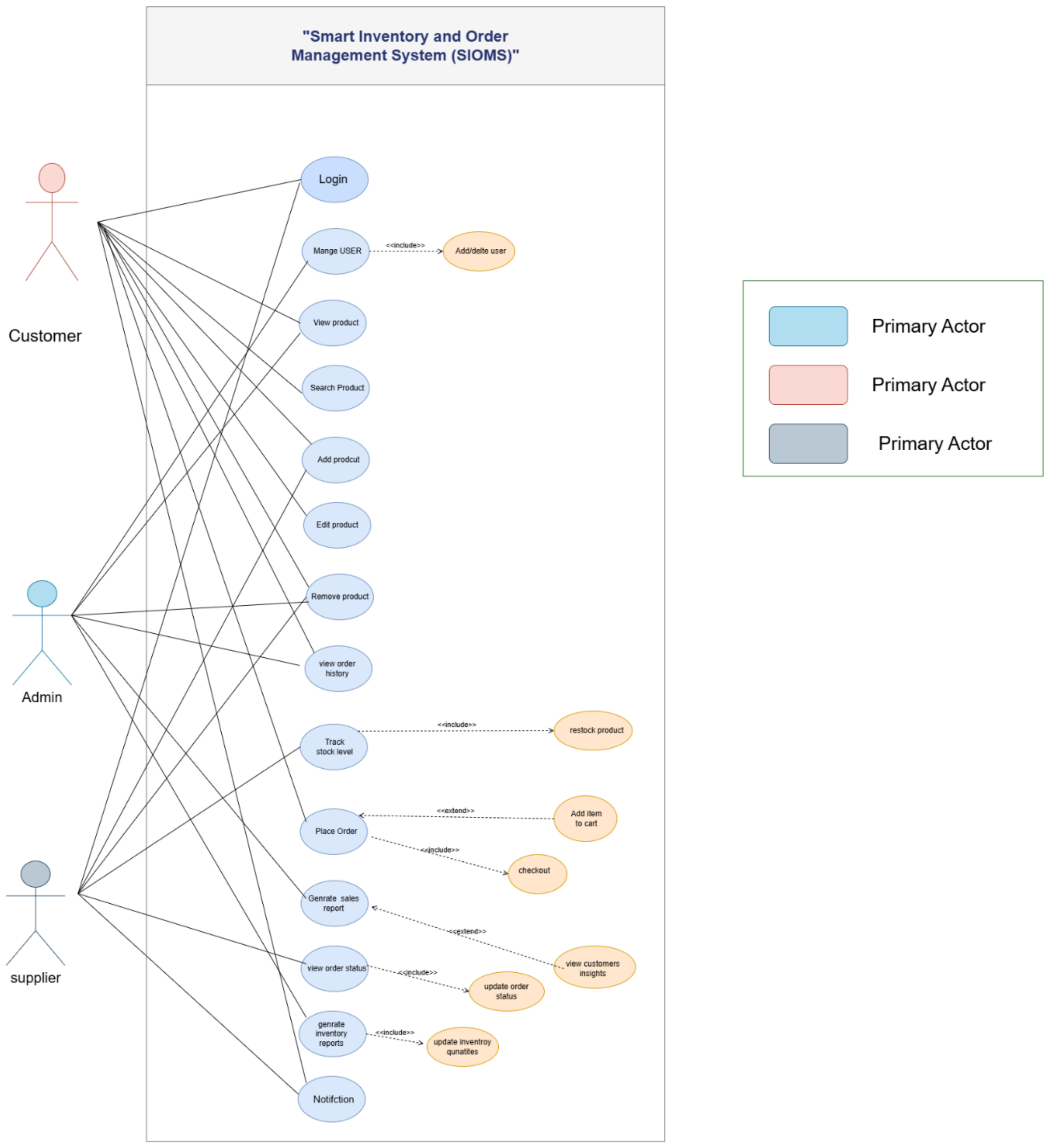
4. Order Management:

- Place Order (Customer): Process customer orders.
- Add Item to Cart (Customer): Prepare a shopping cart for purchase.
- Checkout (Customer): Complete the order process.
- View Order History (Customer and Supplier): Track past orders.
- Update Order Status (Supplier): Change the status of orders.

5. Product Search and Reports:

- Search Product (Customer): Find specific products in the system.
- Generate Sales Report (Admin and Supplier): Create sales analytics.
- Generate Inventory Reports (Admin): Report stock levels.
- View Customer Insights (Admin): Analyze customer behavior and data.

6. Notifications: Notify stakeholders about key events (e.g., stock updates, order changes)



Fully Dressed Use Cases:

❖ Develop detailed use cases for any three critical functionalities.

Use Case 1: User Authentication and Role-Based Access Control

Section	Content
Designation	UC-01
Name	User Authentication and Role-Based Access Control
Author	Ms. Maryam
Priority	Importance for system: High; Technological risk: Medium
Criticality	High
Source	System Requirement
Responsible	Development Team
Description	This use case facilitates secure login and role-based access for users. Users (Admin, Supplier, Customer) can only access features specific to their roles. It ensures a seamless login process with data protection using encryption and robust authentication mechanisms.
Trigger Event	The user attempts to log in through the system's login page.
Actors	Admin, Supplier, Customer
Precondition	The user must have valid login credentials stored in the system. Roles (Admin, Supplier, or Customer) and associated permissions must already exist in the database.
Postcondition	The user is authenticated and granted access to their respective dashboard with role-specific functionalities.
Result	The system successfully verifies the user's credentials, providing secure access to relevant features.
Main Scenario	1. User navigates to the login page and enters valid credentials (username and password). 2. The system encrypts the credentials and compares them with the stored data. 3. If valid, the system identifies the user's role. 4. Depending on the role: - **Admin:** Accesses dashboards for user management, orders, inventory, and reporting. - **Supplier:** Manages inventory (add/edit products, update stock). - **Customer:** Browses products, places orders, and tracks delivery. 5. The system redirects the user to their dashboard.
Alternative Scenario	1a. If the credentials are invalid, the system displays an error: *"Invalid username or password. Please try again."* 1b. If the user forgets their password, they can reset it using the *Forgot Password* link. A reset email is sent, and the user sets a new password.
Quality Requirements	- **Security:** The system must encrypt user credentials and sensitive data. - **Usability:** User interface should provide clear instructions and error messages. - **Performance:** Ensure login verification takes less than 2 seconds for smooth user experience.

Use Case 2: Inventory Management by Suppliers

Section	Content
Designation	UC-02
Name	Inventory Management
Author	Ms. Maryam
Priority	Importance for system: High; Technological risk: Medium
Criticality	High
Source	Supplier Requirement
Responsible	Development Team
Description	Suppliers manage inventory through this use case. They can add new products, edit existing product details, remove discontinued items, and monitor stock levels. The system provides alerts for low-stock products, ensuring timely restocking to prevent supply interruptions.
Trigger Event	The supplier logs in and accesses the inventory management module.
Actors	Supplier
Precondition	Suppliers must be logged in with valid credentials. The system must have existing product categories and data records.
Postcondition	Inventory details such as stock levels, product pricing, and descriptions are updated in the database.
Result	The supplier successfully manages their inventory, ensuring stock availability for customers.
Main Scenario	1. Supplier logs in and accesses the inventory module. 2. The system displays all products with their details (stock levels, price, and categories). 3. Supplier performs actions: - Add a new product by filling out product details (e.g., name, price, quantity, and description). - Update existing product details. - Remove products that are no longer offered. 4. The system processes changes, updates the database, and reflects the updated inventory. 5. Alerts are sent for low-stock products, prompting the supplier to replenish stock.
Alternative Scenario	1a. If invalid product details are entered, the system rejects the entry and displays an error: *"Invalid input. Please check product details."* 2a. If a low-stock product is not replenished within a defined timeframe, the system continues sending periodic notifications until resolved.
Quality Requirements	- **Accuracy:** All updates should immediately reflect in the system. - **Performance:** Inventory management actions should complete without noticeable delays. - **Reliability:** The system must send timely and accurate low-stock notifications to avoid supply chain issues.

Use Case 3: Order Placement and Tracking

Section	Content
Designation	UC-03
Name	Order Placement and Tracking
Author	Ms. Maryam
Priority	Importance for system: High; Technological risk: Low
Criticality	High
Source	Customer Requirement
Responsible	Development Team
Description	Customers can browse products, place orders, and track their delivery status in real-time. The system sends notifications for every order stage (Placed, Processed, Shipped, Delivered). This ensures a smooth shopping experience and transparency during delivery.
Trigger Event	The customer adds items to their cart and proceeds to checkout.
Actors	Customer
Precondition	Customers must have an account and be logged in. Products must be available in stock for the order to proceed.
Postcondition	The order is successfully placed, and the customer can track its progress until delivery.
Result	Customers can complete their purchases and receive timely delivery updates.
Main Scenario	1. Customer browses the product catalog and adds items to the cart. 2. The customer reviews the cart, selects a payment method, and clicks *Checkout*. 3. The system calculates the total amount (with taxes and shipping costs). 4. The customer enters payment and delivery details. 5. The system validates the data and confirms the order. 6. Notifications are sent as the order progresses through stages (Placed → Processed → Shipped → Delivered). 7. Customers can view the order status and track delivery via the system.
Alternative Scenario	1a. If payment fails, the system prompts the customer to retry or use a different payment method. 2a. If an item in the cart is out of stock, the system notifies the customer to remove or replace it. 3a. If delivery is delayed, the system updates the estimated delivery date and notifies the customer.
Quality Requirements	**- Reliability:** Ensure accurate and timely updates on order status. **- User Experience:** Notifications must be clear and delivered in real time. **- Security:** Payment and customer details must be securely handled to prevent data breaches.

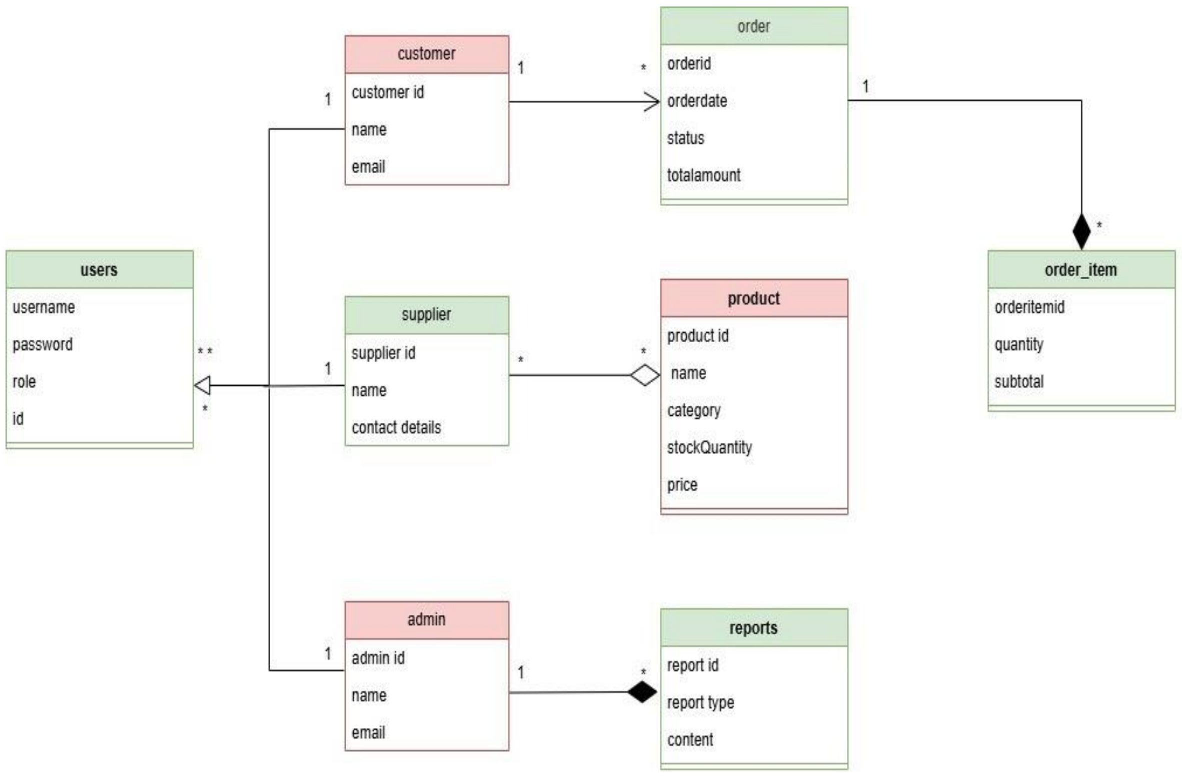
Domain Modeling:

❖ Create a domain model showing primary entities and their relationships.

Main Entities and Relationships:

This is a database schema for the SIOMS system:

1. **Users:** Central table managing username, password, and roles (Admin, Customer, Supplier).
2. **Customer:** Linked to Order (1-to-many); tracks customer details.
3. **Supplier:** Manages Products (many-to-many); tracks supplier details.
4. **Admin:** Generates Reports (1-to-many).
5. **Product:** Includes name, category, stockQuantity, and price.
6. **Order:** Tracks orderDate, status, totalAmount, linked to OrderItem (1-to-many).
7. **Order Item:** Links Order and Product, tracks quantity and subtotal.
8. **Reports:** Generated by Admin for insights.

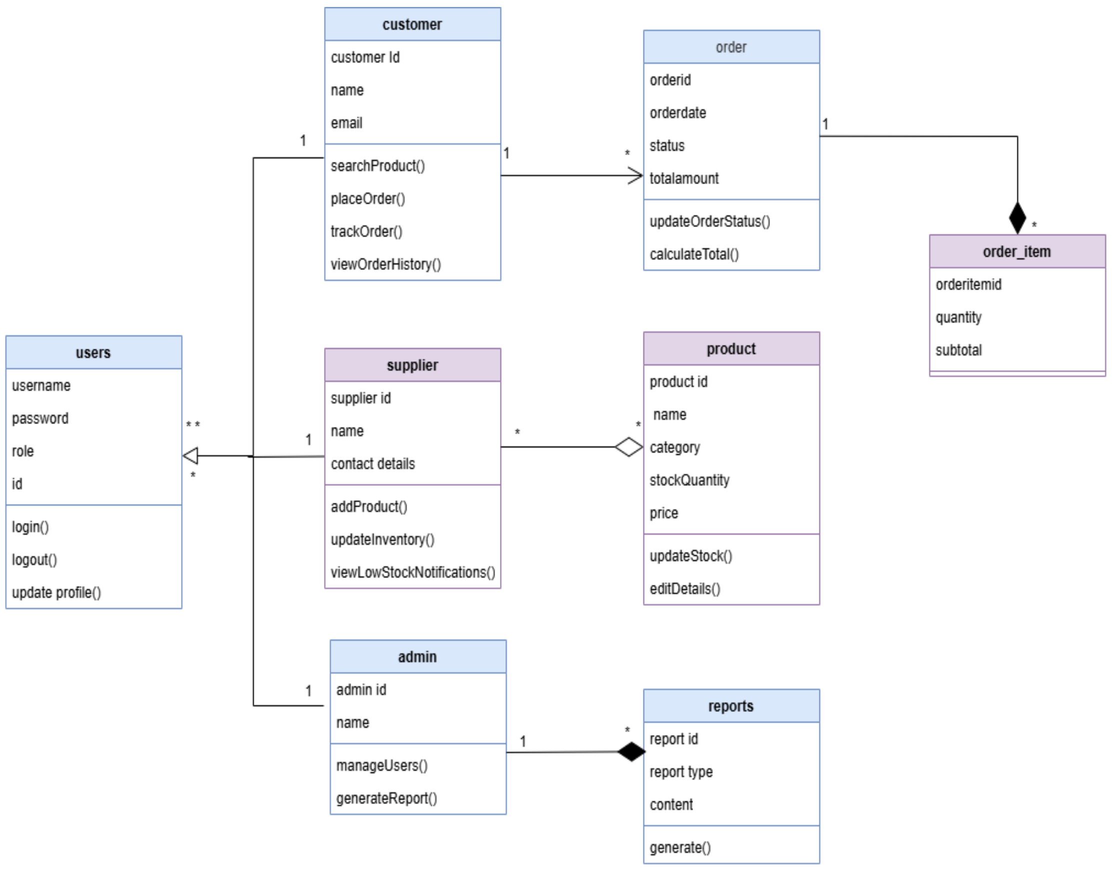


Class Diagram:

❖ Design a class diagram showing classes, attributes, methods, and associations.

This **Class Diagram** for SIOMS includes:

1. **Users**: Handles login (), logout(), and updateProfile().
2. **Customer**: Manages searchProduct(), placeOrder(), trackOrder(), and links to **Order**.
3. **Order**: Tracks status, totalAmount, and connects to **OrderItem**.
4. **OrderItem**: Links **Order** and **Product**, tracks quantity, subtotal.
5. **Supplier**: Handles addProduct(), updateInventory(), linked to **Product**.
6. **Product**: Tracks stockQuantity, price, and updates inventory.
7. **Admin**: Manages users and generates reports.
8. **Reports**: Stores report details, with generate() function

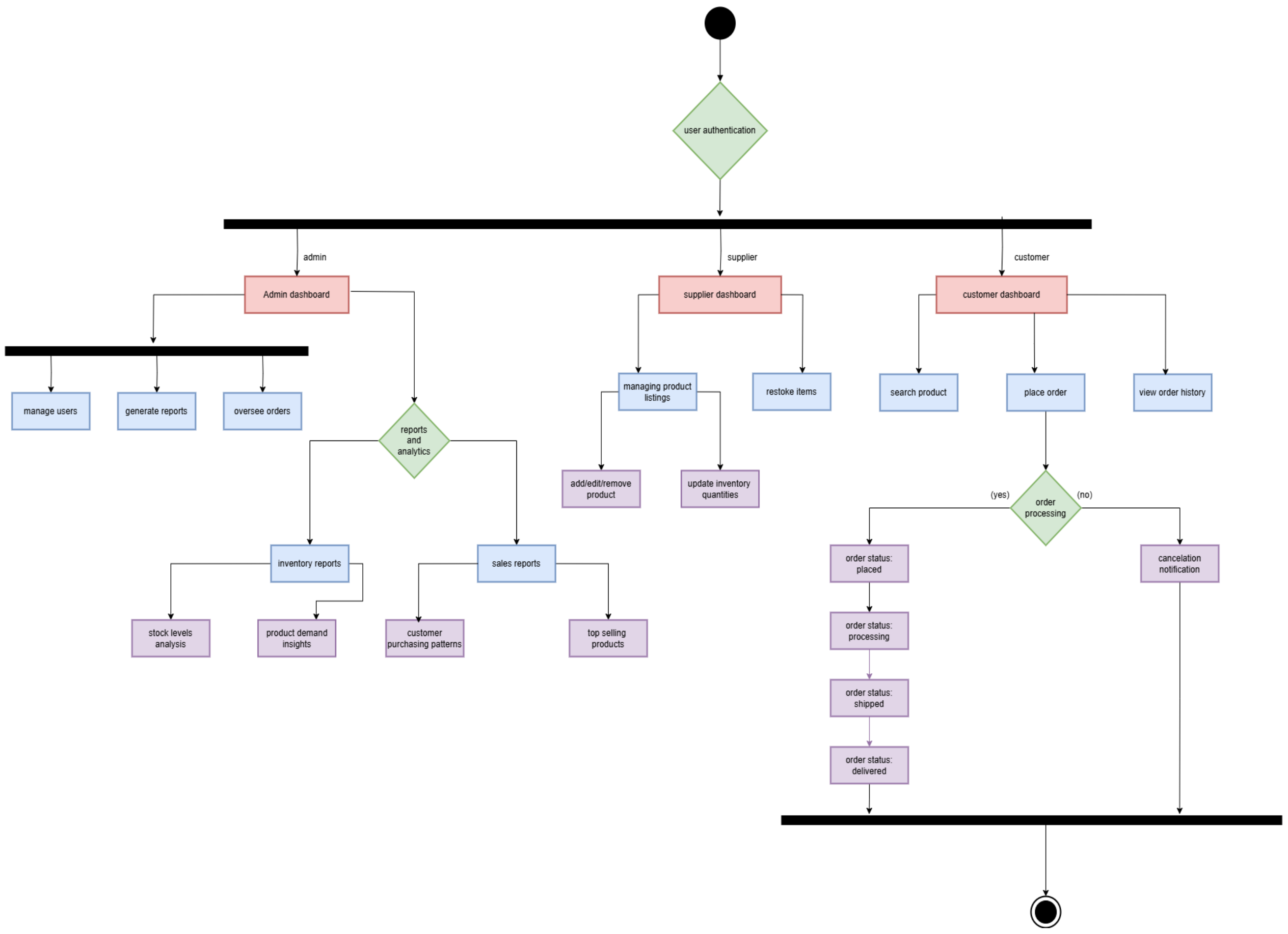


Activity Diagram:

❖ Create activity diagrams for workflows.

The diagram represents a system with three user roles:

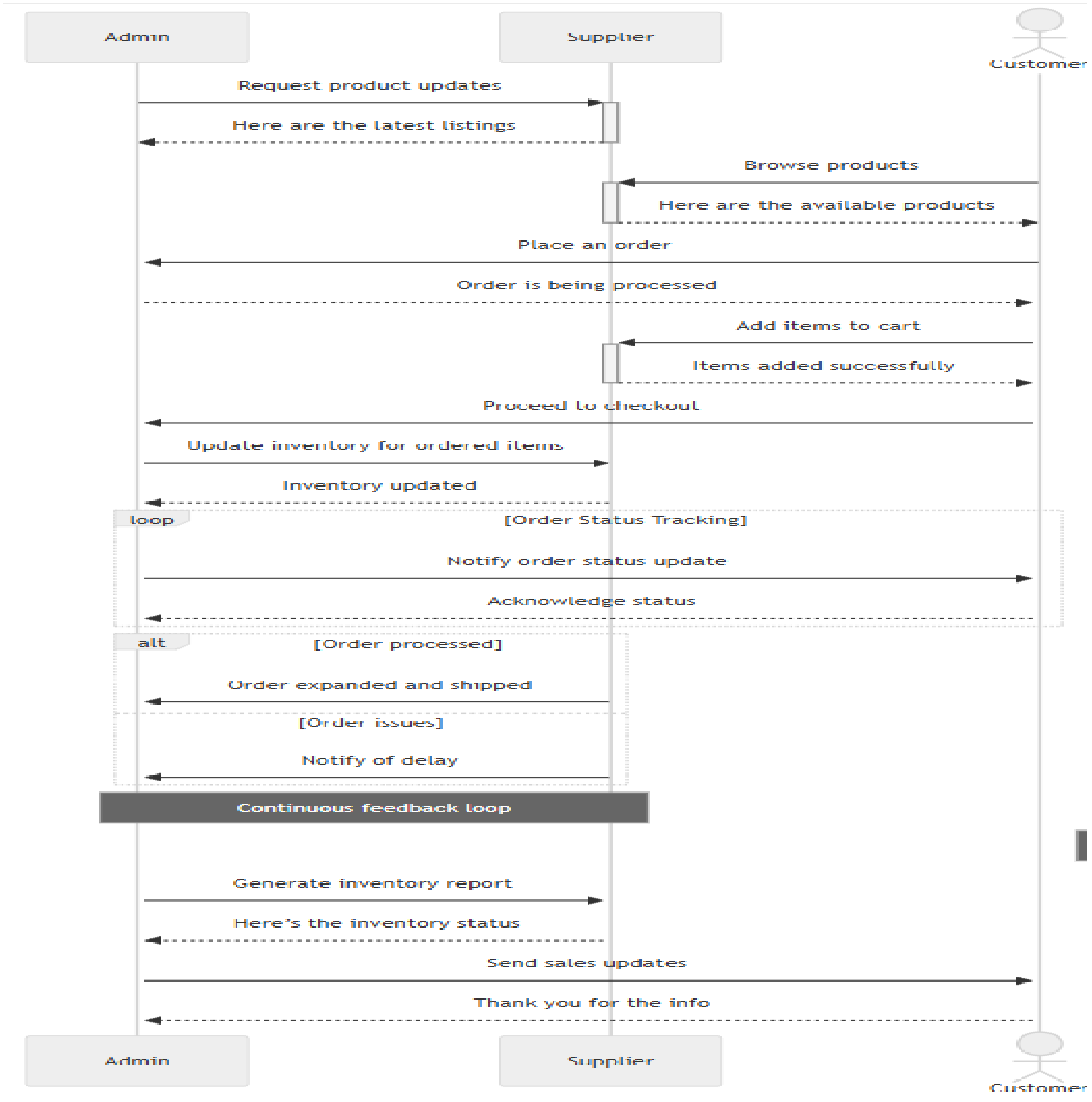
1. **Admin**:
    - Manages users.
    - Oversees orders.
    - Generates reports (inventory and sales insights).
2. **Supplier**:
    - Manages product listings (add, edit, delete).
    - Restocks and updates inventory.
3. **Customer**:
    - Searches products.
    - Places orders (tracked through statuses: Placed → Processing → Shipped → Delivered).
    - Views order history or cancels orders.



Sequence Diagram:

❖ Develop sequence diagrams illustrating object interactions over time.

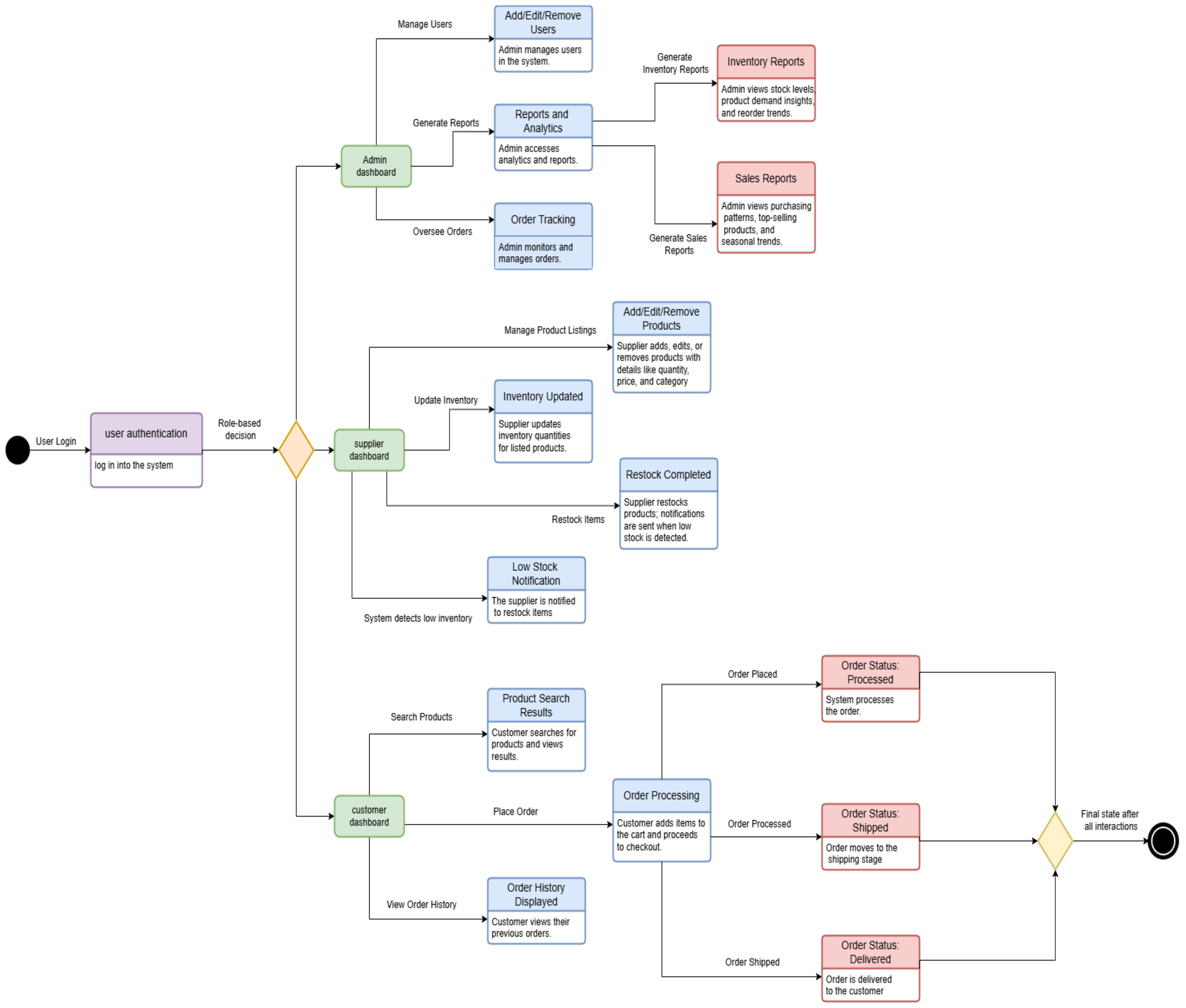
1. **Admin** requests product updates;
2. **Supplier** provides listings.
3. **Customer** browses, adds items to the cart, and places an order.
4. **Inventory** updates, and order status is tracked (notified to the customer).
5. **Orders** are shipped or delayed, with updates shared.
6. **Reports** are generated by the **Admin**, and feedback loops maintain system updates.



State Transition Diagram:

❖ Design state transition diagrams for entities.

- **Login:** Users authenticate and access role-based dashboards.
- **Admin:** Manages users, generates reports, and tracks orders.
- **Supplier:** Updates inventory, restocks, and manages products.
- **Customer:** Searches products, places orders, and views order history.
- **Orders:** Processed through statuses (Processed → Shipped → Delivered)



User Login
user authentication
log in into the system
Role-based decision
Admin dashboard
Manage Users
Add/Edit/Remove Users
Admin manages users in the system.
Generate Reports
Reports and Analytics
Admin accesses analytics and reports.
Generate Inventory Reports
Inventory Reports
Admin views stock levels, product demand insights, and reorder trends.
Generate Sales Reports
Sales Reports
Admin views purchasing patterns, top-selling products, and seasonal trends.
Oversee Orders
Order Tracking
Admin monitors and manages orders.
supplier dashboard
Manage Product Listings
Add/Edit/Remove Products
Supplier adds, edits, or removes products with details like quantity, price, and category
Update Inventory
Inventory Updated
Supplier updates inventory quantities for listed products.
Restock Items
Restock Completed
Supplier restocks products; notifications are sent when low stock is detected.
System detects low inventory
Low Stock Notification
The supplier is notified to restock items
customer dashboard
Search Products
Product Search Results
Customer searches for products and views results.
Place Order
Order Processing
Customer adds items to the cart and proceeds to checkout.
View Order History
Order History Displayed
Customer views their previous orders.
Order Placed
Order Status: Processed
System processes the order.
Order Processed
Order Status: Shipped
Order moves to the shipping stage
Order Shipped
Order Status: Delivered
Order is delivered to the customer
Final state after all interactions

MVC Framework:

❖ Implement the project using the MVC framework, dividing code into Model, View, and Controller layers for organized functionality.

Implementation of the MVC-based system for

User Authentication and Role-Based Access Control

- User Service contains predefined user data and verifies credentials.
- Users are authenticated and assigned their respective roles (Admin, Supplier, Customer).

1. **Model**

User.java:

```java
public class User {
    private String username;
    private String password;
    private String role;

    public User(String username, String password, String role) {
        this.username = username;
        this.password = password;
        this.role = role;
    }

    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }

    public String getRole() {
        return role;
    }
}
```

UserService.java:

```java
import java.util.HashMap;
import java.util.Map;

public class UserService {
    private Map<String, User> users = new HashMap<>();
    public UserService() {
        // Initialize with some users
        users.put("admin", new User("admin", "admin123", "Admin"));
        users.put("supplier", new User("supplier", "sup123", "Supplier"));
        users.put("customer", new User("customer", "cust123", "Customer"));
    }

    public User authenticate(String username, String password) {
        User user = users.get(username);
        if (user != null && user.getPassword().equals(password)) {
            return user;
        }
        return null;
    }
}
```

2. View

LoginView.java:

```
import java.util.Scanner;

public class LoginView {
    private Scanner scanner = new Scanner(System.in);

    public String[] getUserCredentials() {
        System.out.println("Enter Username: ");
        String username = scanner.nextLine();
        System.out.println("Enter Password: ");
        String password = scanner.nextLine();
        return new String[]{username, password};
    }

    public void displayMessage(String message) {
        System.out.println(message);
    }
}
```

DashboardView.java:

```
public class DashboardView {
    public void showDashboard(String role) {
        switch (role) {
            case "Admin":
```

```java
            System.out.println("Welcome to the Admin Dashboard.");
            System.out.println("1. Manage Users\n2. View Reports\n3. Manage Inventory");
            break;
        case "Supplier":
            System.out.println("Welcome to the Supplier Dashboard.");
            System.out.println("1. Manage Products\n2. Update Stock");
            break;
        case "Customer":
            System.out.println("Welcome to the Customer Dashboard.");
            System.out.println("1. Search Products\n2. View Order History\n3. Place Orders");
            break;
        }
    }
}
```

2. Controller

LoginController.java:

```java
public class LoginController {
    private UserService userService;
    private LoginView loginView;
    private DashboardView dashboardView;

    public LoginController(UserService userService, LoginView loginView, DashboardView dashboardView) {
        this.userService = userService;
        this.loginView = loginView;
        this.dashboardView = dashboardView;
    }

    public void authenticateUser() {
        String[] credentials = loginView.getUserCredentials();
        User user = userService.authenticate(credentials[0], credentials[1]);

        if (user != null) {
            loginView.displayMessage("Login successful! Welcome " + user.getRole());
            dashboardView.showDashboard(user.getRole());
        } else {
            loginView.displayMessage("Invalid credentials. Please try again.");
        }
```

Inventory Management

- Suppliers can add, edit, or remove products.
- View the list of products and their stock levels.
- Update product quantities.

1. **Model**

Product.java:

```
public class Product {
    private int id;
    private String name;
    private String category;
    private int quantity;
    private double price;

    public Product(int id, String name, String category, int quantity, double price) {
        this.id = id;
        this.name = name;
        this.category = category;
        this.quantity = quantity;
        this.price = price;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getCategory() {
        return category;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }
```

2. **View**

InventoryView.java:

```
public class InventoryView {
    private Scanner scanner = new Scanner(System.in);

    public void displayProducts(List<Product> products) {
        if (products.isEmpty()) {
            System.out.println("No products available.");
        } else {
            System.out.println("Current Products:");
            for (Product product : products) {
                System.out.println(product);
            }
        }
    }

    public int displayMenuAndGetChoice() {
        System.out.println("\nInventory Management:");
        System.out.println("1. Add Product");
        System.out.println("2. Update Product");
        System.out.println("3. Remove Product");
        System.out.println("4. View All Products");
        System.out.println("5. Exit");
        System.out.print("Enter your choice: ");
        return scanner.nextInt();
    }
```

3. **Controller**

InventoryController.java:

```java
public class InventoryController {
    private InventoryService inventoryService;
    private InventoryView inventoryView;

    public InventoryController(InventoryService inventoryService, InventoryView inventoryView) {
        this.inventoryService = inventoryService;
        this.inventoryView = inventoryView;
    }

    public void manageInventory() {
        while (true) {
            int choice = inventoryView.displayMenuAndGetChoice();
            switch (choice) {
                case 1:
                    String[] productDetails = inventoryView.getProductDetailsForAddition();
                    inventoryService.addProduct(
                        productDetails[0], productDetails[1],
                        Integer.parseInt(productDetails[2]), Double.parseDouble(productDetails[3])
                    );
                    System.out.println("Product added successfully.");
                    break;

                case 2:
                    int updateId = inventoryView.getProductId("update");
                    int newQuantity = inventoryView.getNewQuantity();
                    double newPrice = inventoryView.getNewPrice();
                    try {
                        inventoryService.updateProduct(updateId, newQuantity, newPrice);
                        System.out.println("Product updated successfully.");
                    } catch (IllegalArgumentException e) {
                        System.out.println(e.getMessage());
                    }
                    break;

                case 3:
                    int removeId = inventoryView.getProductId("remove");
                    inventoryService.removeProduct(removeId);
                    System.out.println("Product removed successfully.");
                    break;
                case 4:
                    List<Product> products = inventoryService.getAllProducts();
                    inventoryView.displayProducts(products);
                    break;

                case 5:
                    System.out.println("Exiting Inventory Management.");
                    return;

                default:
                    System.out.println("Invalid choice. Please try again.");
            }
        }
    }
}
```

Order Management and **Admin Reports**

- **Customer can place orders and view order history.**
- **Admin can view all orders, their status, and generate reports.**

1. Model

Order.java:

```
    public Order(int id, int productId, int quantity, String status, Date orderDate, double totalPrice) {
        this.id = id;
        this.productId = productId;
        this.quantity = quantity;
        this.status = status;
        this.orderDate = orderDate;
        this.totalPrice = totalPrice;
    }

    public int getId() {
        return id;
    }

    public int getProductId() {
        return productId;
    }

    public int getQuantity() {
        return quantity;
    }
    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }

    public Date getOrderDate() {
        return orderDate;
    }

    public double getTotalPrice() {
        return totalPrice;
    }
```

2. **View**

OrderView.java:

```
public class OrderView {

    public void displayOrders(List<Order> orders) {
        if (orders.isEmpty()) {
            System.out.println("No orders available.");
        } else {
            System.out.println("Current Orders:");
            for (Order order : orders) {
                System.out.println(order);
            }
        }
    }

    public int displayMenuAndGetChoice() {
        System.out.println("\nOrder Management:");
        System.out.println("1. Place Order");
        System.out.println("2. View All Orders");
        System.out.println("3. Exit");
        System.out.print("Enter your choice: ");
        return new java.util.Scanner(System.in).nextInt();
    }

    public int getProductIdForOrder() {
        System.out.print("Enter Product ID to order: ");
        return new java.util.Scanner(System.in).nextInt();
    }

    public int getQuantityForOrder() {
        System.out.print("Enter Quantity: ");
        return new java.util.Scanner(System.in).nextInt();
    }

    public void displayMessage(String message) {
        System.out.println(message);
    }
}
```

AdminReportView.java:

```
public class AdminReportView {

    public void displayInventoryReport(List<Product> products) {
        System.out.println("Inventory Report:");
        for (Product product : products) {
            System.out.println(product);
        }
    }

    public void displaySalesReport(List<Order> orders) {
        System.out.println("Sales Report:");
        double totalSales = 0;
        for (Order order : orders) {
            totalSales += order.getTotalPrice();
        }
        System.out.println("Total Sales: $" + totalSales);
    }

    public void displayOrders(List<Order> orders) {
        System.out.println("All Orders:");
        for (Order order : orders) {
            System.out.println(order);
        }
    }
```

3. **Controller**

OrderController.java:

```
    public OrderController(OrderService orderService, OrderView orderView) {
        this.orderService = orderService;
        this.orderView = orderView;
    }

    public void manageOrders() {
        while (true) {
            int choice = orderView.displayMenuAndGetChoice();
            switch (choice) {
                case 1:
                    int productId = orderView.getProductIdForOrder();
                    int quantity = orderView.getQuantityForOrder();
                    double totalPrice = 100 * quantity; // Assume a price per unit for simplicity
                    orderService.placeOrder(productId, quantity, totalPrice);
                    orderView.displayMessage("Order placed successfully.");
                    break;

                case 2:
                    orderView.displayOrders(orderService.getAllOrders());
                    break;

                case 3:
                    return;

                default:
                    orderView.displayMessage("Invalid choice. Please try again.");
            }
```

AdminReportController.java:

```
public class AdminReportController {
    private AdminReportView adminReportView;
    private InventoryService inventoryService;
    private OrderService orderService;

    public AdminReportController(AdminReportView adminReportView, InventoryService inventoryService,
OrderService orderService) {
        this.adminReportView = adminReportView;
        this.inventoryService = inventoryService;
        this.orderService = orderService;
    }
    public void generateReports() {
        adminReportView.displayInventoryReport(inventoryService.getAllProducts());
        adminReportView.displaySalesReport(orderService.getAllOrders());
        adminReportView.displayOrders(orderService.getAllOrders());
    }
}
```

We have successfully implemented an **MVC-based Inventory and Order Management System** with role-based access control

GRASP Patterns:

- Apply GRASP patterns to ensure good design principles, focusing on Controller, Information Expert, and Creator, ensuring low coupling and high cohesion among components.

  Controller: Manages requests and defines the flow of the system.

  Information Expert: Assigns responsibilities to the class that has the necessary information.

  Creator: Ensures the class responsible for creating objects has a strong association with them.

  Low Coupling: Minimizes dependencies between classes for better modularity.

The **Controller** pattern is applied here to manage system flow.

```
public class SIOMS {
    static Scanner scanner = new Scanner(System.in);
    static Inventory inventory = new Inventory(); // Information Expert
    static OrderManager orderManager = new OrderManager(inventory); // Creator

    public static void main(String[] args) {
        System.out.println("Welcome to Smart Inventory and Order Management System (SIOMS)");
        while (true) {
            System.out.println("\n1. Admin Menu");
            System.out.println("2. Supplier Menu");
            System.out.println("3. Customer Menu");
            System.out.println("4. Exit");
            System.out.print("Choose an option: ");
            int choice = scanner.nextInt();

            switch (choice) {
                case 1 -> adminMenu();
                case 2 -> supplierMenu();
                case 3 -> customerMenu();
                case 4 -> {
```

```
                System.out.println("Goodbye!");
                System.exit(0);
            }
            default -> System.out.println("Invalid option. Try again.");
        }
    }
}

static void adminMenu() {
    System.out.println("\nAdmin Menu:");
    System.out.println("1. Generate Reports");
    System.out.println("2. Back");
    System.out.print("Choose an option: ");
    int choice = scanner.nextInt();

    if (choice == 1) {
        System.out.println(inventory.generateReport());
    }
}

static void supplierMenu() {
    System.out.println("\nSupplier Menu:");
    System.out.println("1. Add Product");
    System.out.println("2. View Inventory");
    System.out.println("3. Back");
    System.out.print("Choose an option: ");
    int choice = scanner.nextInt();

    switch (choice) {
        case 1 -> inventory.addProduct();
        case 2 -> inventory.viewProducts();
    }
}

static void customerMenu() {
    System.out.println("\nCustomer Menu:");
    System.out.println("1. Place Order");
    System.out.println("2. View Orders");
    System.out.println("3. Back");
    System.out.print("Choose an option: ");
    int choice = scanner.nextInt();

    switch (choice) {
        case 1 -> orderManager.placeOrder();
        case 2 -> orderManager.viewOrders();
        }
    }
}
```

Product Class

The **Information Expert** pattern is applied here since the Product class encapsulates all information related to a product.

```
class Product {
    private String name;
    private int quantity;
    private double price;

    public Product(String name, int quantity, double price) {
        this.name = name;
        this.quantity = quantity;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public int getQuantity() {
        return quantity;
    }

    public double getPrice() {
        return price;
    }

    public void updateQuantity(int newQuantity) {
        this.quantity = newQuantity;
    }

    @Override
    public String toString() {
        return "Product{Name='" + name + "', Quantity=" + quantity + ", Price=" + price + "}";
    }
}
```

The **Information Expert** and **Low Coupling** patterns are used here. The Inventory class handles all operations related to product management.

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
class Inventory {
    private final List<Product> products = new ArrayList<>();
    void addProduct() {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter product name: ");
        String name = scanner.nextLine();
        System.out.print("Enter quantity: ");
        int quantity = scanner.nextInt();
        System.out.print("Enter price: ");
        double price = scanner.nextDouble();

        products.add(new Product(name, quantity, price));
        System.out.println("Product added successfully.");
    }

    void viewProducts() {
        System.out.println("\nCurrent Inventory:");
        for (Product product : products) {
            System.out.println(product);
        }
    }

    Product findProduct(String productName) {
        for (Product product : products) {
            if (product.getName().equalsIgnoreCase(productName)) {
                return product;
            }
        }
        return null;
    }

    String generateReport() {
        StringBuilder report = new StringBuilder("Inventory Report:\n");
        for (Product product : products) {
            report.append(product).append("\n");
        }
        return report.toString();
    }
}
```

Order and OrderManager Classes:

The **Creator** pattern is applied here: the OrderManager creates and manages orders.

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

class Order {
    private final String productName;
    private final int quantity;

    public Order(String productName, int quantity) {
        this.productName = productName;
        this.quantity = quantity;
    }

    @Override
    public String toString() {
        return "Order{Product='" + productName + "', Quantity=" + quantity + "}";}}
class OrderManager {
    private final List<Order> orders = new ArrayList<>();
    private final Inventory inventory;
public OrderManager(Inventory inventory) {
        this.inventory = inventory;}
 void placeOrder() {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter product name: ");
        String productName = scanner.nextLine();
        System.out.print("Enter quantity: ");
        int quantity = scanner.nextInt();
            Product product = inventory.findProduct(productName);
        if (product != null && product.getQuantity() >= quantity) {
            orders.add(new Order(productName, quantity));
            product.updateQuantity(product.getQuantity() - quantity);
            System.out.println("Order placed successfully!");
        } else {
            System.out.println("Product not available or insufficient quantity.");  } }

    void viewOrders() {
        System.out.println("\nYour Orders:");
        for (Order order : orders) {
            System.out.println(order);
        }
    }
```

Explanation of Patterns:

1. **Controller**:
    - The SIOMS class acts as the controller, routing user input to appropriate operations.
2. **Information Expert**:
    - The Product and Inventory classes handle data and logic relevant to their domain.
3. **Creator**:
    - The OrderManager creates Order objects because it manages the order process.
4. **Low Coupling**:
    - Dependencies are minimal, and the SIOMS class interacts only with high-level methods of Inventory and OrderManager.

This implementation balances simplicity and adherence to GRASP principles!