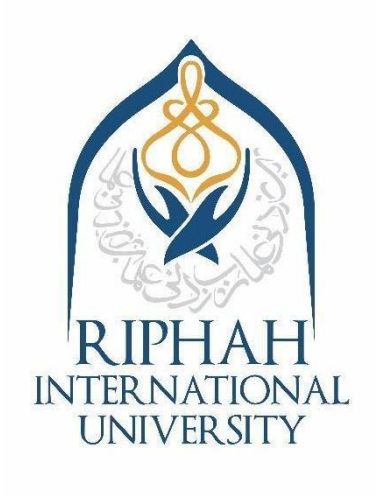
RIPHAH INTERNATIONAL UNIVERSITY, GG CAMPUS



Software Construction & Development

"Smart Inventory and Order Management System (SIOMS)"

Project Team

Name	Sap ID	Program	Email Address
Maryam Safdar	46481	BSSE	46481@students.riphah.edu.pk
Amna jamil	44937	BSSE	44937@students.riphah.edu.pk
Nimra Gul	47054	BSSE	47054@students.riphah.edu.pk
Sabahat Qadeer	47235	BSSE	47235@students.riphah.edu.pk

Date of Submission

24/11/2024

Table of Contents

Artifact 01

Project Proposal

Project Title:

Smart Inventory and Order Management System (SIOMS)

The **Smart Inventory and Order Management System (SIOMS)** enables businesses to efficiently manage inventory, handle orders, and track deliveries. The system includes distinct user roles—**Admin**, **Supplier**, and **Customer**—offering tailored functionalities for each role. Admins oversee operations, manage user roles, and generate reports; suppliers handle inventory management and restocking, while customers place orders and track deliveries. Designed using the **MVC framework** and **GRASP principles**, the system ensures well-defined responsibilities and scalable structure.

Literature Survey:

Features and Functionalities:

- **F-1** Secure login system with role-specific privileges.
- **F-2** Each role (Admin, Supplier, Customer) is assigned distinct tasks and capabilities.
- **F-3** Admins manage users, oversee inventory, and monitor sales transactions.
- **F-4** Suppliers add products, manage inventory, and update stock.
- **F-5** Customers browse product listings, create shopping carts, and place orders.
- **F-6** Track inventory levels and restock when necessary.
- **F-7** Search products by criteria like name, category, and availability.
- **F-8** Customers view order history for past purchases.
- **F-9** Suppliers can update order statuses (e.g., Pending, Shipped, Delivered).
- **F-10** Generate sales and inventory reports to analyze trends and performance.
- **F-11** Notifications alert users about low stock or order updates.

Artifact 02
Use Case Diagram

Use Case Diagram:

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 fulfilment.
- 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 behaviour and data.

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

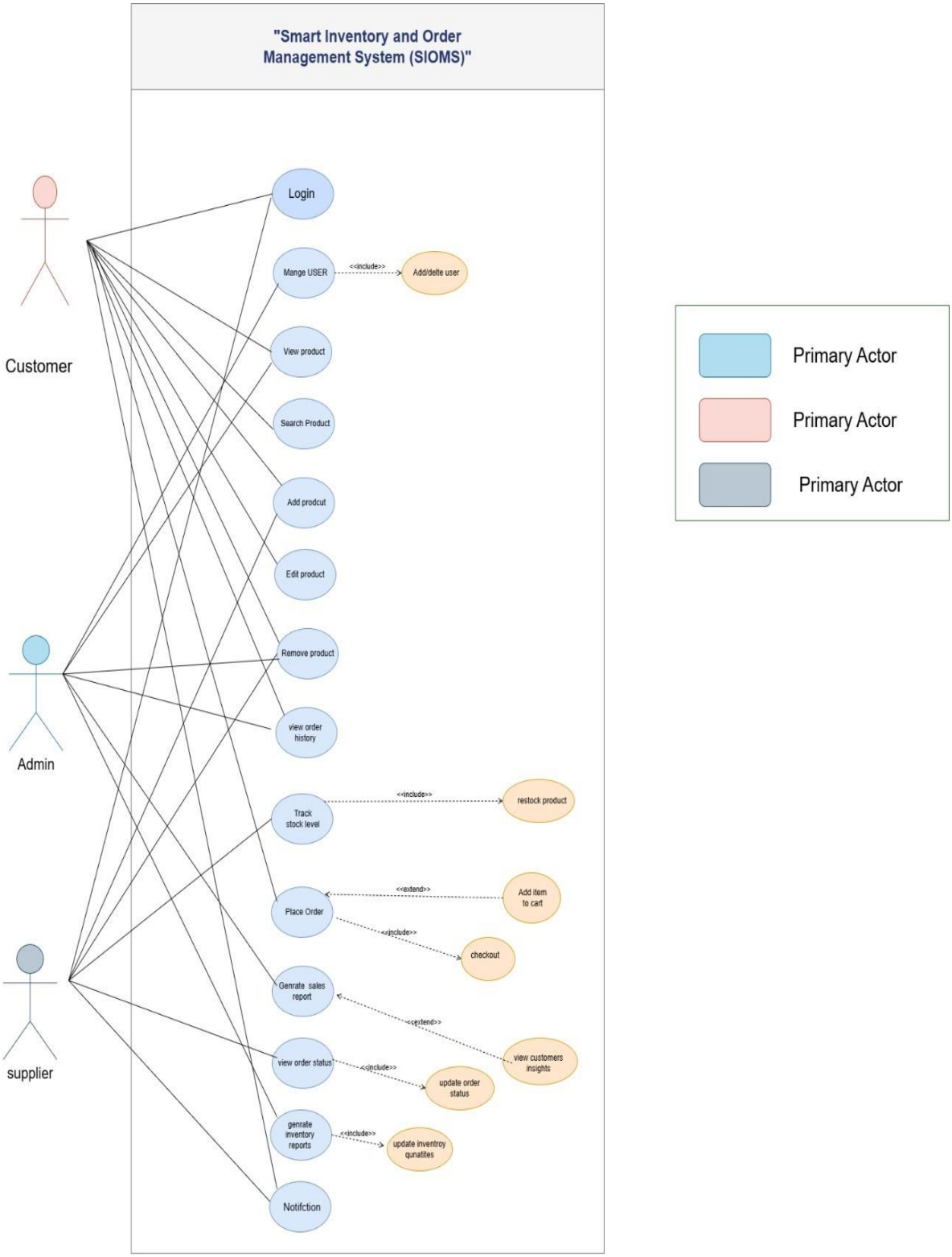


Figure 1 Use Case Diagram of Smart Inventory and Order Management System (SIOMS)

Artifact 03
Fully Dressed Use Cases

Use Cases:

Role-Based Functionality: Ensuring each role has specific functionality (though this is part of the login and role management, it’s a support feature). Here’s how

Critical Functionalities

UC-1 User Login: Secure login with role-based access control for Admin, Supplier, and Customer.

UC-2 Manage Users (Admin): Admin's ability to add, edit, or delete users and assign roles.

UC-3 Inventory Management (Supplier): Suppliers adding, updating, or removing product listings in inventory.

UC-4 Search Products (Customer): Customers searching and filtering products by categories, price, and stock availability.

UC-5 Place Order (Customer): Customers placing orders for available products.

UC-6 Track Stock Levels (Supplier/Admin): Monitoring stock levels and receiving alerts for low inventory.

UC-7 Process Orders (Supplier): Suppliers managing and updating the status of orders (Processing, Shipped, Delivered).

UC-8 View Reports (Admin): Admin viewing sales reports, inventory performance, and user activity reports.

UC-9 Notifications: System notifying users (e.g., order updates for customers, low-stock alerts for suppliers).

Medium Functionalities

UC-10 Update Inventory (Supplier): Suppliers modifying product details like price, stock quantity, and description.

UC-11 View Order History (Supplier/Customer): Viewing past orders for suppliers (fulfilled) and customers (purchased).

UC-12 Generate Sales Reports (Admin): Admin generating sales analytics and performance reports for the business.

UC-13 Restock Inventory (Supplier): Suppliers replenishing inventory based on low-stock alerts.

UC-14 Monitor Customer Insights (Admin): Admin analysing customer behaviour, purchasing trends, and frequently ordered products

UC-15 Order Status Updates (Supplier): Suppliers updating customers on order progress, including delays or shipment details.

Low Functionalities

UC-16 View Product Details (Customer): Customers viewing detailed product descriptions, prices, and availability.

UC-17 Role-Specific Dashboards: Ensuring each role (Admin, Supplier, Customer) has a tailored dashboard with specific 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.
Post condition	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.
Post condition	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.
Post condition	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 catalogue 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.

Artifact 04
Domain Modeling

Domain Model

A domain model is a conceptual representation of the main entities, their attributes, and relationships within a specific problem domain. It is often used during the early stages of software development to help developers and stakeholders understand the structure and behaviour of the system being designed. A domain model serves as a blueprint for how data and operations will be handled, ensuring clarity and alignment before detailed coding begins.

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, stock Quantity, and price.
6. **Order:** Tracks order Date, status, total Amount, linked to Order Item (1-to-many).
7. **Order Item:** Links Order and Product, tracks quantity and subtotal.

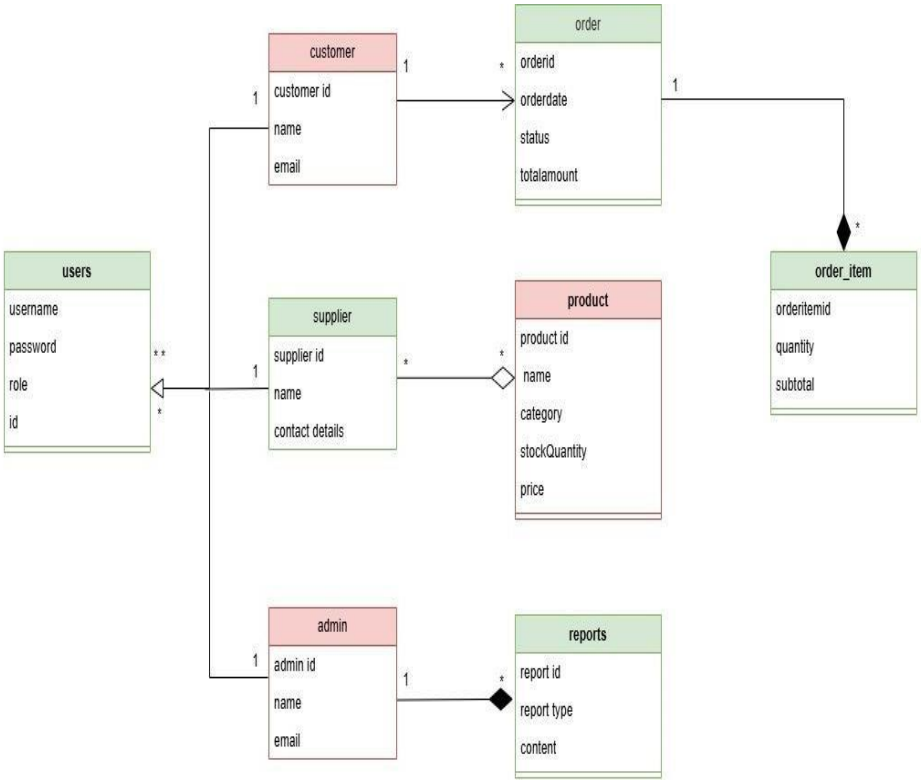


Figure 2 Domin Modeling Diagram

Key Classes and Their Roles

1. User

- **Description:** The base class representing any user in the system.
- **Attributes:** user ID, username, password, role.
- **Responsibilities:**
    - Provides a framework for Admin, Supplier, and Customer roles.

2. Admin

- **Inherits from:** User.
- **Responsibilities:**
    - Manage users using manage Users ().
    - Oversee inventory and sales transactions through oversee Inventory () and monitor Transactions ().
    - Generate sales and stock reports via generate Reports ().

3. Supplier

- **Inherits from:** User.
- **Responsibilities:**
    - Manage inventory through add Product (), update Stock (), and remove Product ().
    - Monitor stock levels using check Stock ().

4. Customer

- **Inherits from:** User.
- **Responsibilities:**
    - Place orders via place Order () and checkout ().
    - View order history using viewOrderHistory ().
    - Browse products with search Product ().

5. Product

- **Description:** Represents items in the inventory.
- **Attributes:** product ID, name, category, price, stock Quantity.
- **Responsibilities:**
    - Track and update product details using update Details ().
    - Monitor stock status with check Availability ().

6. Order

- **Description:** Represents customer orders.
- **Attributes:** order ID, order Date, status, total Amount.
- **Responsibilities:**
    - Link customers and products in orders.
    - Update order status using updateOrderStatus ().

7. Order Item

- **Description:** Links products to specific orders.
- **Attributes:** orderItemID, quantity, subtotal.
- **Responsibilities:**
    - Calculate subtotal for a specific product in the order.

8. Notification

- **Description:** Handles sending notifications to users.
- **Attributes:** notification ID, message, date Time.
- **Responsibilities:**
    - Notify stakeholders about order updates and stock alerts using send Notification ().

9. Report

- **Description:** Represents sales and inventory insights generated by Admin.
- **Attributes:** report ID, report Type, generated Date.
- **Responsibilities:**
    - Summarize data for sales and inventory with generate Report ().

Relationships Explained

- **Inheritance:**

- **Admin**, **Supplier**, and **Customer** inherit from the **User** class, representing their distinct roles.

- **Associations:**

- **Supplier** is associated with **Product** to manage inventory.
- **Customer** is associated with **Order** for placing orders and viewing history.
- **Order** is linked to **Order Item**, connecting products with quantities for each order
- **Product** is linked to **Order Item** to represent items purchased.

- **Notification** is associated with **User** to alert stakeholders about critical updates.
- **Admin** is linked to **Report** to generate insights about sales and inventory trends.

Artifact 05
Class Diagram

Class Diagram

A class diagram is a type of UML (Unified Modeling Language) diagram that visually represents the structure of a system by showing its classes, attributes, operations (methods), and the relationships between them. It is commonly used in object-oriented design to model the static structure of a system and is crucial during the planning and design phase of software development.

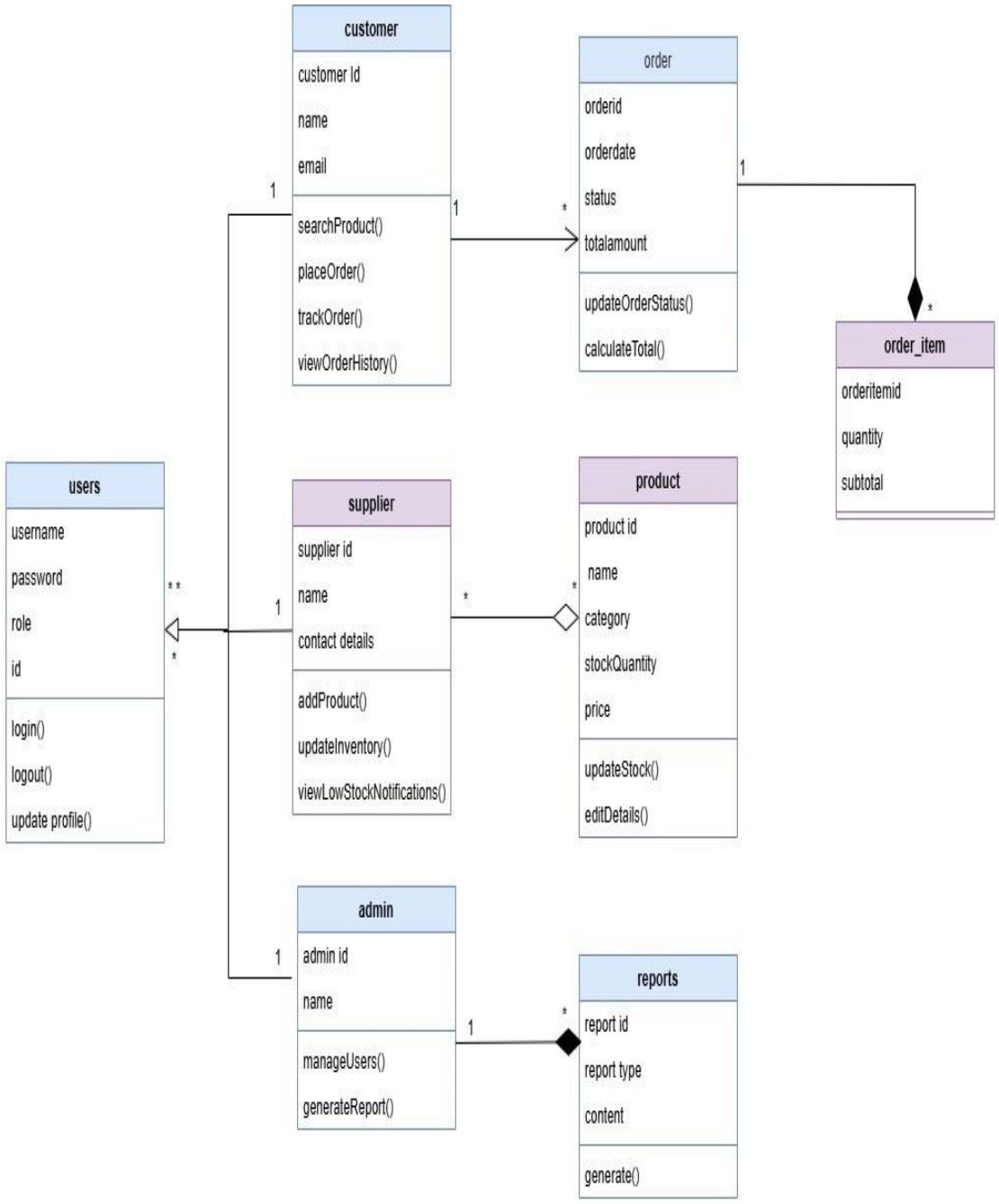


Figure 3 Class Diagram

Key Classes and Their Roles

1. User

- **Attributes:**
  - user ID: Unique identifier for the user.
  - username: Name of the user.
  - password: Secure login credential.
  - role: Defines the user's role (Admin, Supplier, or Customer).
- **Methods:**
  - login (): Authenticates users based on credentials.
  - logout(): Logs out users from the system.
  - viewProfile(): Allows users to view their profile details.
  - updateProfile(): Enables users to modify their profile information.

2. Admin (Inherits from User)

- **Attributes:**
  - Inherits attributes from User.
- **Methods:**
  - manageUsers(): Adds, edits, or deletes user accounts.
  - generateReports(): Creates sales and inventory reports.
  - monitorTransactions(): Tracks all system transactions.

3. Supplier (Inherits from User)

- **Attributes:**
  - Inherits attributes from User.

    - supplierID: Unique identifier for the supplier.
    - inventory: A list of products managed by the supplier.
- **Methods:**
    - addProduct(): Adds new products to the inventory.
    - updateStock(): Updates the stock level of products.
    - removeProduct(): Removes discontinued products.
    - viewInventory(): Displays the current product inventory.

4. Customer (Inherits from User)

- **Attributes:**
    - Inherits attributes from User.
    - orderHistory: List of previous orders placed by the customer.
    - cart: Temporary storage for selected products before checkout.
- **Methods:**
    - browseProducts(): Allows customers to search and filter products.
    - placeOrder(): Processes customer orders.
    - viewOrderHistory(): Displays details of past orders.

5. Product

- **Attributes:**
    - productID: Unique identifier for the product.
    - name: Name of the product.
    - category: Type or classification of the product.
    - price: Cost of the product.
    - stockQuantity: Available quantity in stock.
- **Methods:**
    - updateDetails(): Modifies product attributes like price or description.
    - checkAvailability(): Validates stock availability.

6. Order

- **Attributes:**
  - orderID: Unique identifier for the order.
  - orderDate: Date the order was placed.
  - status: Current status of the order (e.g., Pending, Shipped, Delivered).
  - totalAmount: Total cost of the order.
- **Methods:**
  - updateOrderStatus(): Changes the status of an order.
  - calculateTotal(): Computes the total cost of items in the order.

7. OrderItem

- **Attributes:**
  - orderItemID: Unique identifier for the order item.
  - productID: The product associated with this order item.
  - quantity: Number of units ordered.
  - subtotal: Cost of the order item.
- **Methods:**
  - calculateSubtotal(): Calculates the subtotal for the product.

8. Notification

- **Attributes:**
  - notificationID: Unique identifier for the notification.
  - message: Content of the notification.
  - dateTime: Time when the notification was created.
- **Methods:**
  - sendNotification(): Sends alerts about stock or order updates.

9. Report

- **Attributes:**
  - reportID: Unique identifier for the report.
  - type: Type of report (e.g., Sales, Inventory).
  - generatedBy: The admin who created the report.
  - data: Details and insights included in the report.
- **Methods:**
  - generateReport(): Creates a new report based on selected criteria.
  - viewReport(): Displays the generated report.

Relationships Explained

- **Inheritance:**
  - Admin, Supplier, and Customer inherit from User.
- **Associations:**
  - **Admin** generates Reports to analyze sales and inventory trends.
  - **Supplier** manages Products in the inventory.
  - **Customer** places Orders and views past purchases.
  - **Order** links Customers with Products through OrderItems.
  - **Product** is associated with Orders through OrderItems.
  - **Notification** alerts Users about order updates or low stock.
  - **Reports** provide insights into system performance.

Artifact 06
Activity Diagram

Activity Diagram

An activity diagram is a type of UML (Unified Modeling Language) diagram that represents the flow of activities or actions within a system. It focuses on the sequence of actions, decision points, and parallel processes, making it useful for Modeling workflows, business processes, or complex algorithms.

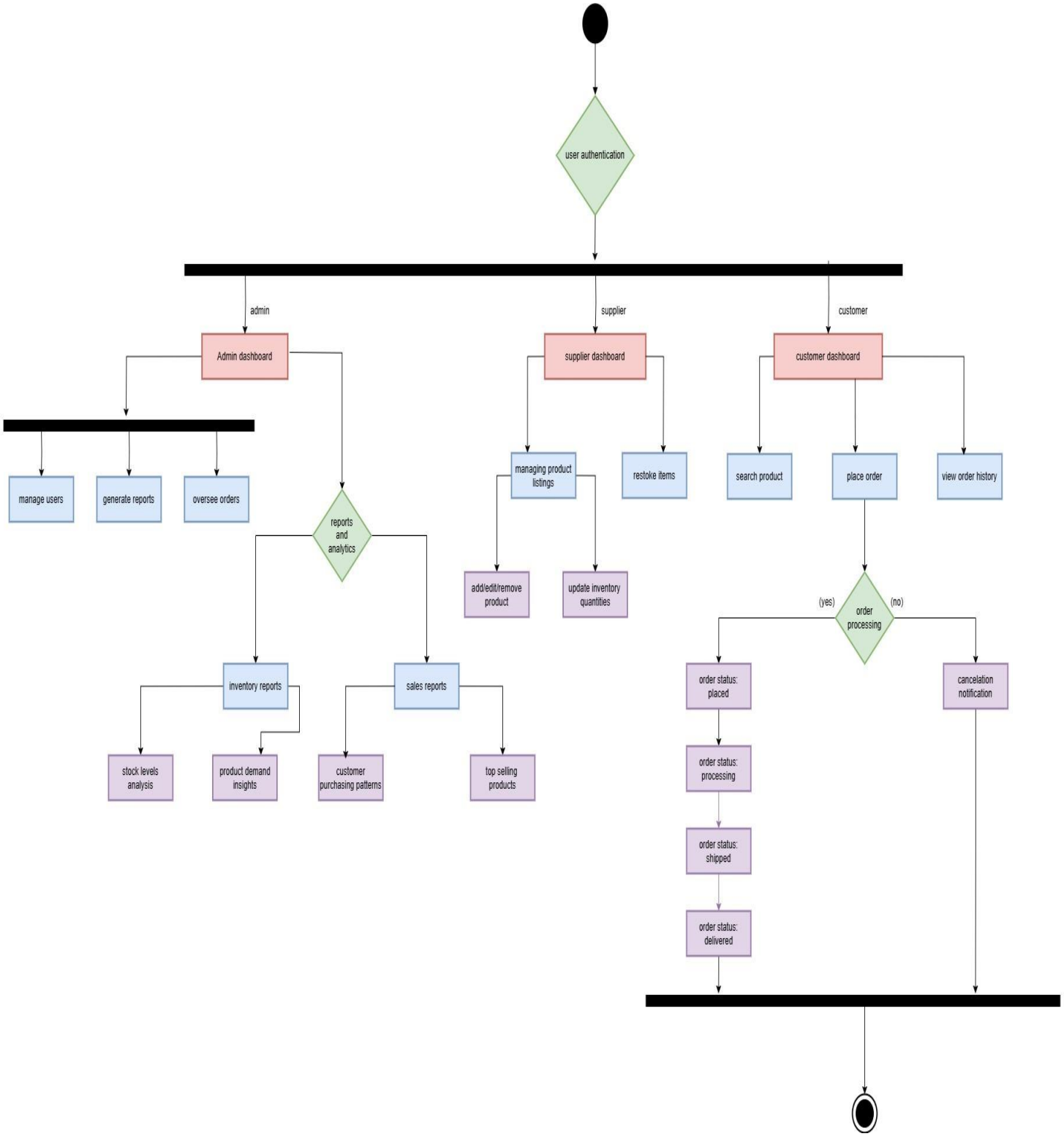


Figure 4 Activity Diagram

1. Login Process

- **Actors:** Customer, System, Authentication Service
- **Flow:**
    1. The **Customer** initiates a login request to the **System**.
    2. The **System** sends the login information to the **Authentication Service**.
    3. An **alt block** checks the credentials:
        - If the **credentials are correct**, the **System** grants access, and the **Authentication Service** authenticates the user.
        - If the **credentials are incorrect**, access is denied, and authentication fails.

2. Product Addition to Inventory

- **Actors:** Supplier, System, Database
- **Flow:**
    1. The **Supplier** adds a new product to the **System**.
    2. The **System** stores the product details in the **Database**.
    3. The **System** confirms the product has been successfully added to the inventory.

3. Order Placement Process

- **Actors:** Customer, System, Order Service, Inventory Service
- **Flow:**
    1. The **Customer** places an order through the **System**.
    2. The **System** processes the order and checks product availability using **Inventory Service**.
    3. An **alt block** checks if the product is in stock:
        - If the product is in stock, the order is confirmed, and the **Order Service** creates the order.
        - If the product is out of stock, the **Customer** is notified of the unavailability.

4. Order Processing and Shipment

- **Actors:** Supplier, System, Shipping Service
- **Flow:**

1. The **System** notifies the **Supplier** of a new order.
2. The **Supplier** prepares the items for shipment.
3. An **alt block** checks if the items are ready for shipment:
    - If the items are ready, the **Shipping Service** ships the items, and the **Customer** is notified of the shipment status.
    - If the items are not ready, the **Customer** is notified of the delay.

5. Transaction and Invoice Generation

- **Actors:** Customer, System, Database
- **Flow:**
    1. Once an order is shipped, the **System** records the transaction details in the **Database**.
    2. The **System** generates an invoice for the **Customer** and sends the invoice via email.
    3. An **alt block** checks the payment status:
        - If the payment is successful, the order is marked as complete.
        - If the payment fails, the **Customer** is notified to retry or use another payment method.

6. Inventory Report Request

- **Actors:** Admin, Supplier, Database
- **Flow:**
    1. The **Admin** or **Supplier** requests an inventory report through the **System**.
    2. The **Database** generates the inventory report data.
    3. The **System** sends the report to the **Admin** or **Supplier**.

Artifact 07
Sequence Diagram

Sequence diagram

A **sequence diagram** is a type of UML (Unified Modeling Language) diagram that shows how objects interact in a particular scenario of a system. It visually represents the sequence of messages or interactions between objects over time, making it useful for understanding the dynamic behaviour of a system.

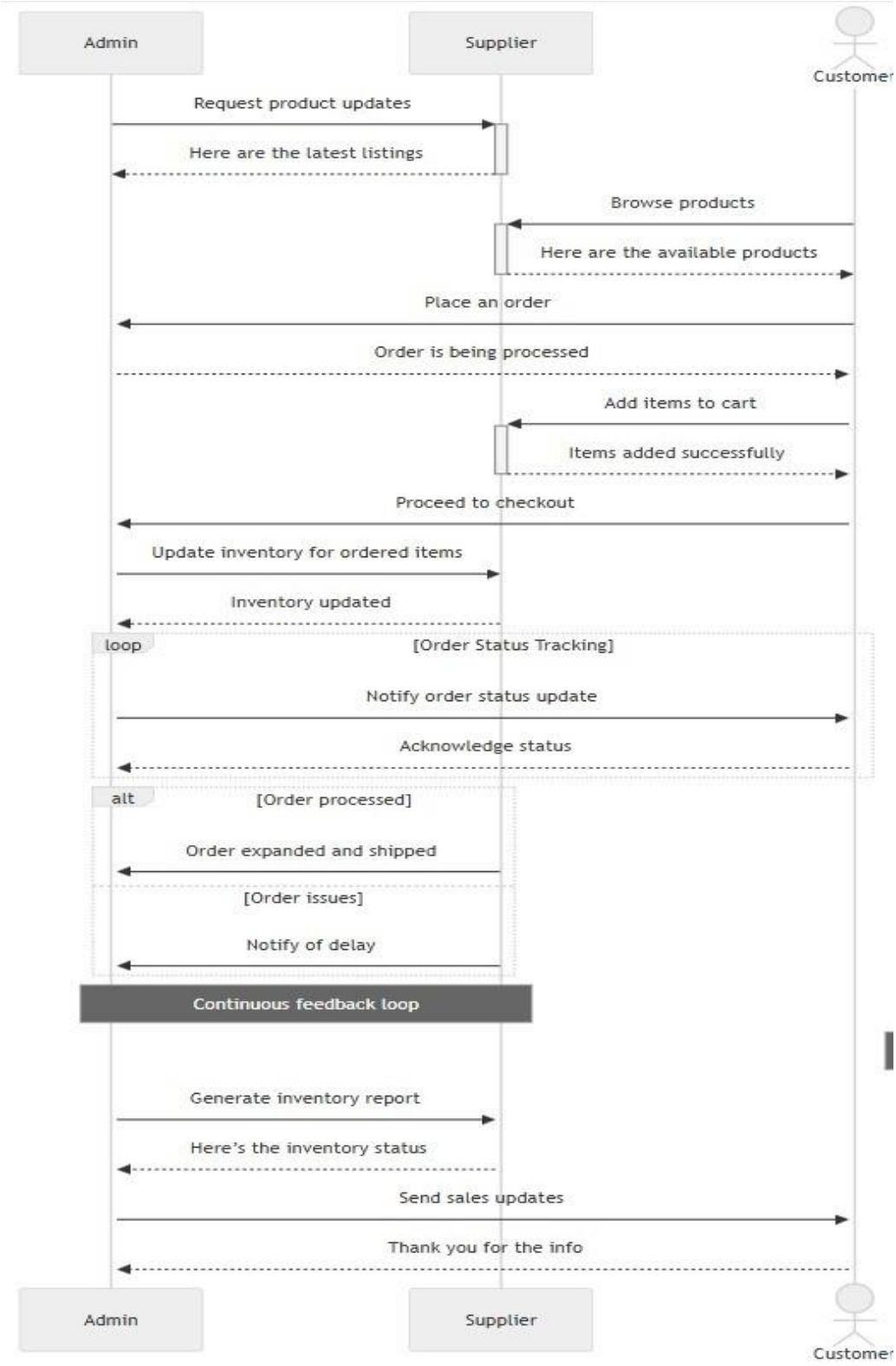


Figure 5 Sequence Diagram

1. User Login Process

- **Actors:** Customer, System, Authentication Service
- **Flow:**
    1. The **Customer** sends a login request to the **System**.
    2. The **System** forwards the login information to the **Authentication Service**.
    3. An **alt** block checks the login credentials:
        - If the password is correct, the **System** grants access, and the **Authentication Service** indicates that the user is authenticated.
        - If the password is incorrect, access is denied, and authentication fails.

2. Product Addition to Inventory

- **Actors:** Supplier, System, Database
- **Flow:**
    1. The **Supplier** initiates a request to add a new product to the **System**.
    2. The **System** validates the product details.
    3. The **System** then saves the product details to the **Database**.
    4. The **System** confirms the successful addition of the product to the inventory.

3. Order Placement

- **Actors:** Customer, System, Inventory Service, Order Service
- **Flow:**
    1. The **Customer** selects products and places an order through the **System**.
    2. The **System** checks product availability by querying the **Inventory Service**.
    3. An **alt** block checks if the product is in stock:

- If the product is available, the **Order Service** processes the order, and the **System** confirms the successful order placement.
- If the product is out of stock, the **Customer** is notified of unavailability.

4. Order Shipping and Status Update

- **Actors:** Supplier, System, Shipping Service
- **Flow:**

1. The **System** notifies the **Supplier** of the new order.
2. The **Supplier** prepares the product for shipment.
3. An **alt** block checks if the product is ready for shipment:

- If the product is ready, the **Shipping Service** ships the product, and the **Customer** is notified of the shipment status.
- If the product is not ready, the **Customer** is notified about the delay.

5. Order Payment and Transaction Completion

- **Actors:** Customer, System, Payment Gateway, Database
- **Flow:**
  1. The **Customer** enters payment details through the **System**.
  2. The **System** forwards the payment details to the **Payment Gateway** for processing.
  3. An **alt** block checks the payment status:

- If the payment is successful, the **System** logs the transaction details in the **Database** and notifies the **Customer** of a successful purchase.
- If the payment fails, the **Customer** is notified to retry or use a different payment method.

6. Inventory and Sales Report Request

- **Actors:** Admin, Supplier, Database
- **Flow:**
  1. The **Admin** or **Supplier** requests a sales or inventory report via the **System**.
  2. The **System** queries the **Database** to gather report data.
  3. The **System** generates the requested report and sends it to the **Admin** or **Supplier**.

Artifact 08
State Transition Diagram

State Diagram:

A state diagram (also known as a state machine diagram or statechart) is a type of UML diagram that depicts the different states an object can be in throughout its lifecycle and the transitions between those states. It is particularly useful for modeling the behavior of objects that change state in response to external events.

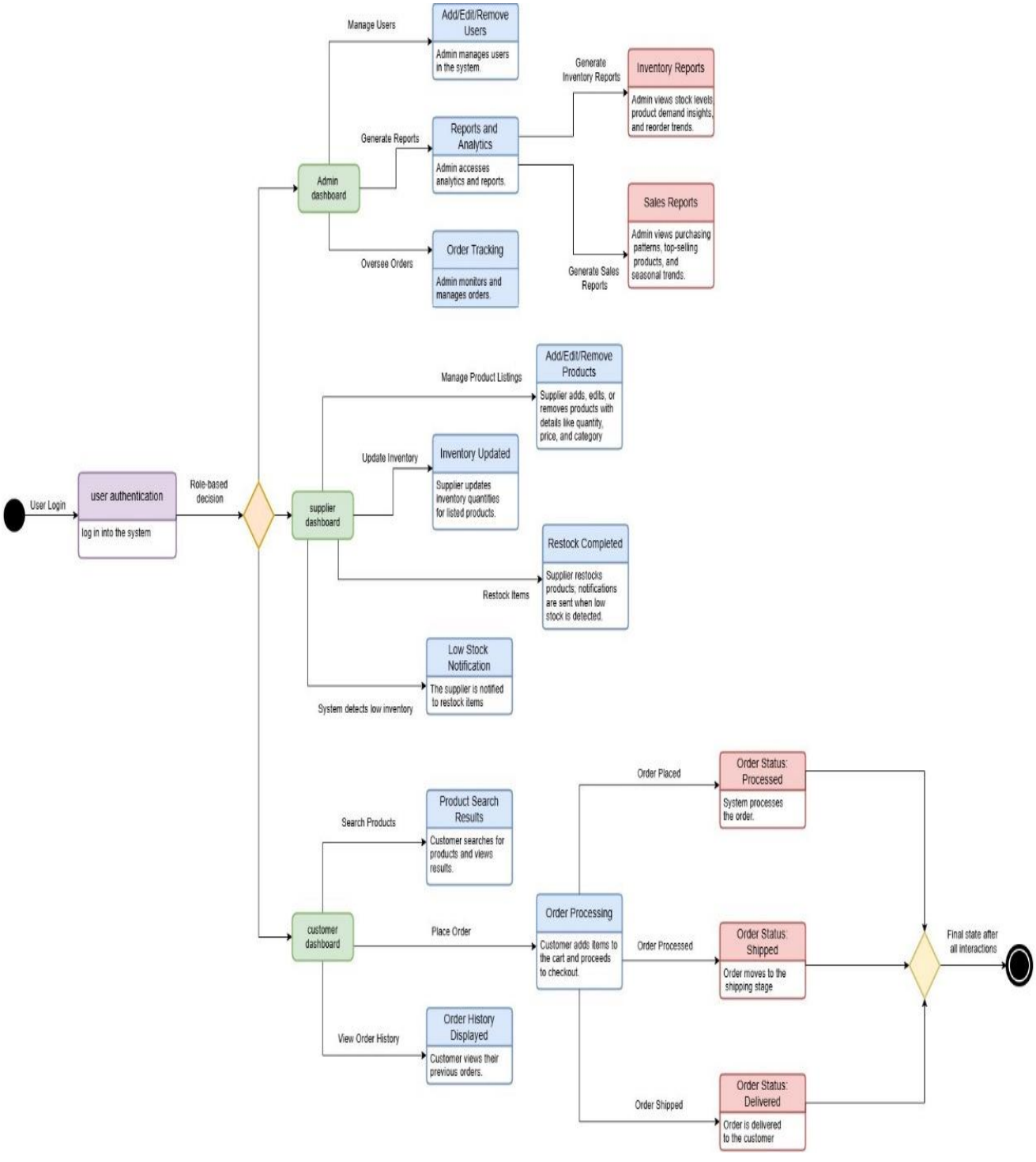


Figure 6 States Diagram)

State Diagram for SIOMS

1. Notification

- States:
  - Pending: A notification is created but not yet sent.
  - Sent: The notification is sent to the user.
  - Read: The user has viewed the notification.
  - Archived: The notification is archived after being read or dismissed.
- Transitions:
  - Create → Pending → Sent → Read → Archived
  - Create → Pending → Sent → Archived (if dismissed by the user)

2. User

- States:
  - Logged Out: The user is not logged into the system.
  - Logged In: The user has successfully logged into the system.
  - Profile Updated: The user has updated their profile.
  - Password Changed: The user has changed their password.
- Transitions:
  - Logged Out → Login → Logged In
  - Logged In → Update Profile → Profile Updated
  - Logged In → Change Password → Password Changed
  - Logged In → Logout → Logged Out

3. Admin (Inherits from User)

- States:
  - Idle: The admin is logged in but not performing any action.
  - Managing Users: The admin is managing user accounts.
  - Managing Properties: The admin is managing property listings.

    - Managing Transactions: The admin is overseeing transactions.
- Transitions:
    - Idle → Manage Users → Managing Users
    - Idle → Manage Properties → Managing Properties
    - Idle → Manage Transactions → Managing Transactions

4. Client (Inherits from User)

- States:
    - Searching Properties: The client is browsing properties.
    - Requesting Viewing: The client requests a property viewing.
    - Submitting Offer: The client submits an offer for a property.
    - Tracking Offer: The client is tracking the status of submitted offers.
- Transitions:
    - Logged In → Searching Properties → Searching Properties
    - Searching Properties → Requesting Viewing → Requesting Viewing
    - Requesting Viewing → Submitting Offer → Submitting Offer
    - Submitting Offer → Tracking Offer → Tracking Offer

5. Agent (Inherits from User)

- States:
    - Idle: The agent is logged in but not managing any properties.
    - Managing Properties: The agent is managing property listings.
    - Scheduling Viewings: The agent is scheduling property viewings.
    - Approving Offers: The agent is reviewing and approving offers.
    - Uploading Virtual Tour: The agent is uploading media content for properties.
- Transitions:
    - Idle → Managing Properties → Managing Properties
    - Managing Properties → Scheduling Viewings → Scheduling Viewings
    - Scheduling Viewings → Approving Offers → Approving Offers
    - Approving Offers → Uploading Virtual Tour → Uploading Virtual Tour

6. Property

- States:

    - Listed: The property is available for viewing.
    - Under Offer: The property has an active offer.
    - Sold: The property has been sold.
    - Removed: The property is no longer listed (removed by agent or admin).
- Transitions:
    - Listed → Under Offer → Sold
    - Listed → Removed → Removed

7. Viewing

- States:
    - Requested: The client has requested a viewing for the property.
    - Scheduled: The viewing has been confirmed and scheduled.
    - Completed: The viewing has been completed.
    - Cancelled: The viewing was canceled, either by the client or agent.
- Transitions:
    - Requested → Scheduled → Completed
    - Requested → Cancelled → Cancelled

8. Offer

- States:
    - Submitted: The client has submitted an offer for the property.
    - Under Review: The offer is under review by the agent.
    - Accepted: The offer has been accepted, and a transaction is created.
    - Rejected: The offer has been rejected by the agent.
- Transitions:
    - Submitted → Under Review → Accepted
    - Submitted → Under Review → Rejected
    - Accepted → Transaction Created → Completed
    - Rejected → End

9. Transaction

- States:
    - Pending: The transaction is being processed.
    - Completed: The transaction has been successfully completed.

    - Canceled: The transaction was canceled due to various reasons (e.g., rejected offer, failed payment).
- Transitions:
    - Pending → Completed → Completed
    - Pending → Canceled → Canceled

Relationships:

- **Inheritance:**
    - Admin, Client, and Agent all inherit from User and have additional state management behaviors.
- **Associations:**
    - Notification is linked to User for notifying them about important events.
    - Admin manages Property listings, Client requests Viewing, and Agent manages Property and Viewing.
    - Viewing involves Client and Agent for scheduling and managing appointments.
    - Offer transitions to Transaction when accepted.

Artifact 09
MVC Framework

Model-View-Controller (MVC) Architecture

The **Model-View-Controller (MVC)** is a software architectural pattern that divides an application into three main components: **Model**, **View**, and **Controller**. This separation of concerns provides a modular structure, enabling each component to be developed, maintained, and tested independently, ultimately leading to cleaner and more maintainable code. MVC is especially useful for applications where data needs to be managed and displayed interactively, as it isolates business logic from the user interface.

Components of MVC

1. Model

The **Model** component is responsible for the application's data and business logic. It represents the core functionality of the application by managing data storage, retrieval, and any operations or rules associated with data processing. The Model communicates directly with the data source or database, performing operations like create, read, update, and delete (CRUD).

- **Responsibilities**:
    - Defines and manages application data and business logic.
    - Encapsulates all data-related operations, including CRUD operations.
    - Notifies the View of data changes when necessary to keep the user interface updated.
- **Model Components in This Application**:
    - **User**: Represents system users, containing information such as username, password, and role (Admin, Agent, or Client). The User model provides methods for checking user roles and managing user details.
    - Renders data from the Model for the user to view. ○ Collects user inputs and relays them to the Controller.
    - Updates the user interface based on interactions managed by the Controller.
    - Receives and processes user inputs from the View. ○ Interacts with the Model to execute data operations and handle business logic.
    - Updates the View to reflect any data changes or responses from the Model.

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:

```
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:

```
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":
                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:

```
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:

```
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());
   }
}
```

Artifact 10
GRASP Patterns

GRASP

GRASP (General Responsibility Assignment Software Patterns) is a set of guidelines for assigning responsibilities to classes and objects in object-oriented design. It helps designers create maintainable, understandable, and robust systems by promoting good object-oriented practices. These patterns were introduced by Craig Larman in his book "*Applying UML and 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.

```
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.

```
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!