RAJALAKSHMI ENGINEERING COLLEGE
RAJALAKSHMI NAGAR, THANDALAM – 602 105

RAJALAKSHMI
ENGINEERING COLLEGE

CB23332
SOFTWARE ENGINEERING LAB

Laboratory Record Note Book

Name : . . . . . .

Year / Branch / Section : . . . . . .

Register No. : . . . . . .

Semester : . . . . . .

Academic Year : . . . . . .

RAJALAKSHMI ENGINEERING COLLEGE (AUTONOMOUS)
RAJALAKSHMI NAGAR, THANDALAM – 602-105

BONAFIDE CERTIFICATE

NAME:_____REGISTER NO.: _____

ACADEMIC YEAR: 2024-25 **SEMESTER:** III **BRANCH:_____**B.E/B.Tech

This Certification is the bonafide record of work done by the above student in the

CB23332-SOFTWARE ENGINEERING - Laboratory during the year 2024 – 2025.

Signature of Faculty -in – Charge

Submitted for the Practical Examination held on _____

Internal Examiner External Examiner

INDEX

S.No.	Name of the Experiment	Expt. Date	Faculty Sign
1.	Preparing Problem Statement		
2.	Software Requirement Specification (SRS)		
3.	Entity-Relational Diagram		
4.	Data Flow Diagram		
5.	Use Case Diagram		
6.	Activity Diagram		
7.	State Chart Diagram		
8.	Sequence Diagram		
9.	Collaboration Diagramt		
10.	Class Diagram		

EX NO:1 Problem Statement

AIM:

The agricultural supply chain is often marred by inefficiencies, with multiple intermediaries driving up costs and reducing the quality of produce by the time it reaches the consumer. Farmers face challenges in accessing markets for their produce, while consumers struggle with high prices and lack of transparency about the origins of their food. The aim is to develop an integrated platform that directly connects farmers with consumers, eliminating intermediaries, reducing costs, ensuring fair pricing, and increasing transparency. This system will also allow farmers to gain better control over pricing and access to a larger consumer base while enabling consumers to purchase fresh, quality produce at competitive rates.

ALGORITHM:

1. **Initialize System**:
    - Load necessary data (farmers, consumers, produce details, payment systems).
    - Set up user roles (farmer, consumer, admin).
2. **User Authentication**:
    - Prompt user to log in (farmer, consumer, or admin).
    - Validate credentials and direct user to appropriate dashboard.
3. **Farmer Profile Management**:
    - Allow farmers to create and update profiles (personal details, farm location, available produce).
    - Add information about farming practices (organic, conventional, etc.) to increase consumer trust.
4. **Product Listing**:
    - Farmers list available produce, including details such as:
        - **Type of product** (vegetables, fruits, grains, etc.).
        - **Quantity** available.
        - **Price** per unit (fixed or negotiable).
        - **Delivery terms** (pickup or home delivery).
        - **Quality certification** (if applicable).
5. **Consumer Search and Browse**:
    - Consumers can search for produce by:
        - **Type of product**.
        - **Price range**.
        - **Delivery availability**.
        - **Farm type** (e.g., organic, local).
        - **Location proximity** (for local produce).

- Display available products with details.

6. **Product Purchase**:
   - Consumers can add products to their cart.
   - They can select payment options (credit/debit card, online payment gateway).
   - Process payment and confirm order.
   - Send order details to the respective farmer.
7. **Order Management (Farmer)**:
   - Farmers can view incoming orders, accept or reject them.
   - Update order status (pending, processing, shipped, delivered).
   - Communicate directly with the consumer for any clarification.
8. **Delivery Management**:
   - System helps facilitate delivery logistics (either via the farmer or a third-party delivery service).
   - Track delivery status and notify both farmer and consumer when delivery is complete.
9. **Feedback and Reviews**:
   - After receiving the produce, consumers can provide feedback and rate the farmer.
   - Farmers can also rate consumers, ensuring mutual trust and accountability.
10. **Payment Processing**:
    - Payment is held in escrow until delivery is confirmed.
    - Farmers are paid once the order is successfully delivered and confirmed by the consumer.
11. **Communication System**:
    - Direct messaging between farmers and consumers for clarification on product details, delivery, etc.
    - Notifications for new products, price changes, and order updates.
12. **Admin Dashboard**:
    - Admins can monitor platform activity, ensure compliance, and resolve disputes.
    - Generate reports on market trends, popular products, consumer satisfaction, etc.
13. **User Logout**:
    - Secure log-out process.

INPUTS:

1. **User Authentication:**
   - **Username**
   - **Password**
2. **Farmer Profile Management:**

- **Farm name**
- **Farm location**
- **Contact details** (phone number, email)
- **Produce details** (type, quantity, price, quality certification)

3. **Product Listing:**
   - **Type of product** (vegetables, fruits, etc.)
   - **Quantity** available
   - **Price** per unit
   - **Quality certification** (if applicable)
4. **Consumer Search:**
   - **Product type**
   - **Price range**
   - **Delivery terms**
   - **Location proximity**
5. **Payment Processing:**
   - **Payment method** (credit/debit card, online payment gateway)
   - **Billing address**
   - **Card details** (if using card)
6. **Order Management (Farmer):**
   - **Order details** (consumer, product, quantity)
   - **Order status** (pending, shipped, delivered)
7. **Feedback/Reviews:**
   - **Farmer ID** or **Consumer ID**
   - **Rating** (1 to 5 stars)
   - **Review content**
8. **Search and Reporting:**
   - **Date range** for order tracking
   - **Product type or category** for reporting

STAKEHOLDER PROBLEM STATEMENT:

Problem

The agricultural supply chain faces several inefficiencies, such as:

- **Middlemen**: Intermediaries that increase the cost of produce, leading to higher prices for consumers and lower earnings for farmers.
- **Lack of transparency**: Consumers are unaware of where their food comes from and may not trust the quality of the produce.
- **Limited market access for farmers**: Farmers often struggle to reach consumers directly, especially small-scale producers.
- **Inconsistent pricing**: Due to fluctuating demand and supply, farmers often face challenges in pricing their produce appropriately, leading to either surplus or shortage.

Background

Agriculture is the backbone of many economies, yet farmers face difficulties in marketing their products directly to consumers due to a lack of platforms that bridge the gap. As a result, farmers depend on intermediaries, which lowers their profit margins. Consumers, on the other hand, often end up paying higher prices and lack information about the origin and quality of the produce they purchase.

In addition, rising concerns over food safety, quality, and environmental sustainability have made consumers more conscious about where their food comes from and how it is produced. An effective system that allows consumers to buy fresh, local, and possibly organic produce directly from the farmer can significantly improve both the economic conditions for farmers and the food security of consumers.

OBJECTIVES

- **Reduce Middlemen**: Remove intermediaries in the agricultural supply chain to increase profits for farmers and reduce costs for consumers.
- **Provide Fresh Produce**: Allow consumers to directly purchase fresh produce from farmers, ensuring better quality and taste.
- **Transparent Pricing**: Enable clear pricing mechanisms that are fair to both farmers and consumers.
- **Increase Farmer Revenue**: Help farmers gain better control over pricing and reach a broader market.
- **Consumer Empowerment**: Allow consumers to make informed purchasing decisions, providing insights into the origin and quality of the food they purchase.
- **Ensure Security and Trust**: Build a secure platform that ensures consumer data protection, payment security, and fair transactions.
- **Foster Sustainability**: Support sustainable farming practices and create a market for environmentally conscious consumers.

RESULTS

In conclusion, the direct access platform between farmers and consumers aims to build a fairer, more efficient food distribution system that benefits all stakeholders—farmers, consumers, and the environment.

EX NO:2 SOFTWARE REQUIREMENT SPECIFICATION (SRS)

AIM:

The aim of this Software Requirement Specification (SRS) document is to define the functional and non-functional requirements for a **Direct Access between Farmer and Consumer** platform. This system is designed to establish a direct connection between farmers and consumers, allowing them to buy and sell fresh produce efficiently. It removes the need for intermediaries, ensuring better prices for both parties and promoting local farming.

ALGORITHM:

1. **Initialize System**
    - Load configuration settings and user data (farmers, consumers, products, orders, payments).
    - Define user roles (admin, farmer, consumer).
2. **User Authentication**
    - Prompt user to enter username and password.
    - Validate credentials.
    - If valid, direct user to their respective dashboard; if invalid, display an error message.
3. **Farmer's Product Management**
    - Add Products: Farmers can add products, providing details such as name, description, price, quantity, and photos.
    - Update Products: Farmers can update product details.
    - View Product Listings: Consumers can search and browse products by category, price, or location.
    - Delete Products: Farmers can delete products when they are out of stock or no longer available.
4. **Consumer's Order Management**
    - Browse and Search Products: Consumers can search for available produce based on different criteria (type of product, location, price).
    - Create Orders: Consumers can select products, add them to their cart, and place orders.
    - Track Orders: Consumers can track the status of their orders (e.g., processing, shipped, delivered).
    - Order History: Consumers can view their past orders.
5. **Payment Processing**
    - Capture payment details (method, card info, digital wallet).
    - Validate payment and update order status.
    - Notify consumer of payment status (successful, failed).
    - Provide transaction history for consumers.
6. **Delivery Management**
    - Assign Deliveries: Farmers or system staff can assign deliveries to logistics teams.
    - Track Deliveries: Consumers and farmers can track the delivery status.

- Confirm Delivery: Consumers can confirm receipt of the product.

7. **Communication System**
   - Send Messages: Consumers can contact farmers for inquiries, feedback, or negotiation.
   - View Messages: Both consumers and farmers can view and reply to messages.
   - Notifications: System sends notifications to consumers and farmers about new orders, product updates, and deliveries.
8. **Review and Rating**
   - Rate Products: Consumers can rate products based on quality and satisfaction.
   - Rate Sellers: Consumers can rate farmers based on their service (delivery time, product quality).
   - View Reviews: Consumers can view reviews and ratings before purchasing.
9. **Reporting and Analytics**
   - Generate reports on sales, payment transactions, and consumer feedback.
   - Provide insights into best-selling products and consumer preferences.
10. **User Logout**

- Securely log out the user and clear session data.

TABLE OF CONTENT:

- 3.5 Delivery Management
- 3.6 Communication System
- 3.7 Review and Rating System
- 3.8 Reporting and Analytics

4. **Non-Functional Requirements**
   - 4.1 Performance
   - 4.2 Security
   - 4.3 Usability
   - 4.4 Reliability
   - 4.5 Compatibility
5. **System Architecture**
   - 5.1 Overview
   - 5.2 Component Diagram
6. **Use Cases**
   - 6.1 Use Case Diagram
   - 6.2 Key Use Cases
7. **User Interface Requirements**
   - 7.1 Design Principles
   - 7.2 Mockups
8. **Testing Requirements**
   - 8.1 Test Strategy
   - 8.2 Test Cases Overview
9. **Implementation Plan**
   - 9.1 Development Methodology
10. **Appendices**

- 10.1 Glossary
- 10.2 Change Log

SAMPLE OUTPUT:

1. Introduction

1.1 Purpose: The purpose of this Software Requirements Specification (SRS) document is to outline the functional and non-functional requirements for a **Direct Access between Farmer and Consumer** platform. This system enables farmers to list and sell their products directly to consumers, enhancing transparency and eliminating intermediaries. It also provides a seamless communication and payment system, ensuring both parties benefit from fair pricing.

1.2 Scope: The scope of this document includes the core functionalities such as product management, order management, payment processing, and communication tools. The platform will allow farmers to manage their listings, and consumers to browse and purchase fresh produce, facilitating a direct farmer-to-consumer exchange.

1.3 Audience: This document is intended for:

- **Developers:** To understand the system's requirements and implement the system accordingly.
- **Project Managers:** To ensure alignment with the project goals and ensure timelines are met.
- **Quality Assurance Teams:** To design test cases based on these requirements.
- **Farmers and Consumers:** To provide insights into the system's functionality.

1.4 Definitions, Acronyms, and Abbreviations:

- **Farmers:** Sellers of produce on the platform.
- **Consumers:** Buyers of fresh produce.
- **Admin:** The system administrator who oversees the platform.
- **API:** Application Programming Interface.

2. Overall Description

2.1 Product Perspective: This platform will be a web and mobile-based application that facilitates a marketplace between farmers and consumers. The system will allow farmers to list their products, manage orders, and handle payments, while consumers will be able to browse products, place orders, and communicate with farmers.

2.2 User Classes and Characteristics:

- **Farmers:** Farmers can create, update, and manage their product listings. They can also communicate with consumers and manage orders.
- **Consumers:** Consumers can browse products, place orders, and make payments. They can also provide reviews and ratings.
- **Admin:** The system administrator who manages user roles, resolves disputes, and monitors the overall system performance.

2.3 Operating Environment:

- Web-based application accessible through major browsers (Chrome, Firefox, Safari).
- Mobile applications for iOS and Android for users to browse and order on-the-go.

2.4 Design and Implementation Constraints:

- The system must adhere to data protection regulations (e.g., GDPR).
- Integration with various payment gateways is necessary.
- A scalable architecture to accommodate the growing number of farmers and consumers.

2.5 Assumptions and Dependencies:

- Internet access is required for both farmers and consumers.
- Integration with external logistics platforms for delivery management.

3. Functional Requirements

3.1 User Authentication:

- The system should allow users to register and log in using secure credentials.

- Users should be able to recover forgotten passwords.

3.2 Farmer's Product Management:

- Farmers can list products with descriptions, images, price, and stock quantity.
- Farmers can update product details as necessary.

3.3 Consumer's Order Management:

- Consumers can add products to their cart, modify quantities, and place orders.

3.4 Payment Processing:

- Secure payment gateway integration (e.g., Stripe, PayPal).
- Payment status notifications to the consumer upon successful transaction.

3.5 Delivery Management:

- Integration with third-party delivery systems for shipping tracking.

3.6 Communication System:

- Messaging system for consumers and farmers to communicate directly.

3.7 Review and Rating System:

- Consumers can rate products and farmers after each transaction.

3.8 Reporting and Analytics:

- Admins can generate reports on sales, payments, consumer demographics, and product popularity.

4. Non-Functional Requirements

4.1 Performance:

- The system should handle up to 10,000 concurrent users with minimal latency.

4.2 Security:

- Implement SSL encryption for all sensitive data.
- Secure payment transactions with PCI-DSS compliance.

4.3 Usability:

- The user interface should be intuitive and easy to navigate.
- The system should be mobile-responsive.

4.4 Reliability:

- The platform should maintain 99.9% uptime with scheduled maintenance windows.

4.5 Compatibility:

- The system should work across major browsers and platforms (web and mobile).

Result:

The SRS document for the **Direct Access between Farmer and Consumer** system has been successfully created.

EX NO:3 ENTITY RELATIONAL MODEL

AIM:

To draw an Entity Relational Model (ERM) for a **Direct Access Between Farmer and Consumer** system.

ALGORITHM:

Step 1: Mapping of Regular Entity Types

- **Entities**: Identify the main entities in the system.
    - **Farmer**: Attributes: FarmerID (PK), Name, Email, Phone, Address, FarmType.
    - **Consumer**: Attributes: ConsumerID (PK), Name, Email, Phone, Address.
    - **Product**: Attributes: ProductID (PK), Name, Category, Price, StockQuantity.
    - **Order**: Attributes: OrderID (PK), ConsumerID (FK), OrderDate, Status.
    - **Payment**: Attributes: PaymentID (PK), OrderID (FK), Amount, PaymentDate, PaymentStatus.

Step 2: Mapping of Weak Entity Types

- **Weak Entities**: Identify any weak entities that depend on a strong entity for their existence.
    - **OrderItem**: Attributes: OrderItemID (PK), OrderID (FK), ProductID (FK), Quantity, SubTotal.
        - **Dependency**: The OrderItem entity depends on the Order entity.

Step 3: Mapping of Binary 1:1 Relation Types

- **Relations**: Identify relationships where one entity is related to one and only one instance of another entity.
    - **Farmer to Farm**: Each farmer owns one farm.
        - Relationship: **Farmer (FarmerID)** 1:1 **Farm (FarmID)**

Step 4: Mapping of Binary 1

Relationship Types

- **Relationships**: Identify relationships where one entity is related to many instances of another entity.
    - **Farmer to Product**: One farmer can grow multiple products.
        - Relationship: **Farmer (FarmerID)** 1
          Product (ProductID)
    - **Consumer to Order**: A consumer can place multiple orders.
        - Relationship: **Consumer (ConsumerID)** 1
          Order (OrderID)
    - **Product to OrderItem**: One product can appear in multiple orders (order items).
        - Relationship: **Product (ProductID)** 1
          OrderItem (OrderItemID)

Step 5: Mapping of Binary M

Relationship Types

- **Relationships**: Identify many-to-many relationships.
    -
    - **Consumer to Product**: Consumers can buy multiple products, and each product can be bought by many consumers.
        - Mapping: Create a junction table.
            - **Junction Table**: ConsumerProduct: Attributes: ConsumerID (FK), ProductID (FK).
            - Relationship: **Consumer (ConsumerID)** M **Product (ProductID)**.

Step 6: Mapping of Multivalued Attributes

- **Attributes**: Identify attributes that can hold multiple values for a single entity.
    - **Product Categories**: A product can belong to multiple categories (e.g., Organic, Fresh, Seasonal).

INPUTS:

Step 1: Mapping of Regular Entity Types

- **Farmer**: A farmer has a unique ID, name, email, phone, address, and farm type.
- **Consumer**: A consumer has a unique ID, name, email, phone, and address.
- **Product**: A product has a unique ID, name, category, price, and stock quantity.
- **Order**: An order has a unique ID, references a consumer, and includes the order date and status.
- **Payment**: A payment is linked to a specific order and includes the amount and payment date.

Step 2: Mapping of Weak Entity Types

- **OrderItem**: Each order can contain multiple items, and each item references a specific product.

Step 3: Mapping of Binary 1:1 Relationship Types

- **Farmer to Farm**: Each farmer owns one farm.

Step 4: Mapping of Binary 1

Relationship Types

- **Farmer to Product**: One farmer can grow multiple products.
- **Consumer to Order**: One consumer can place multiple orders.
- **Product to OrderItem**: A product can appear in many order items.

Step 5: Mapping of Binary M

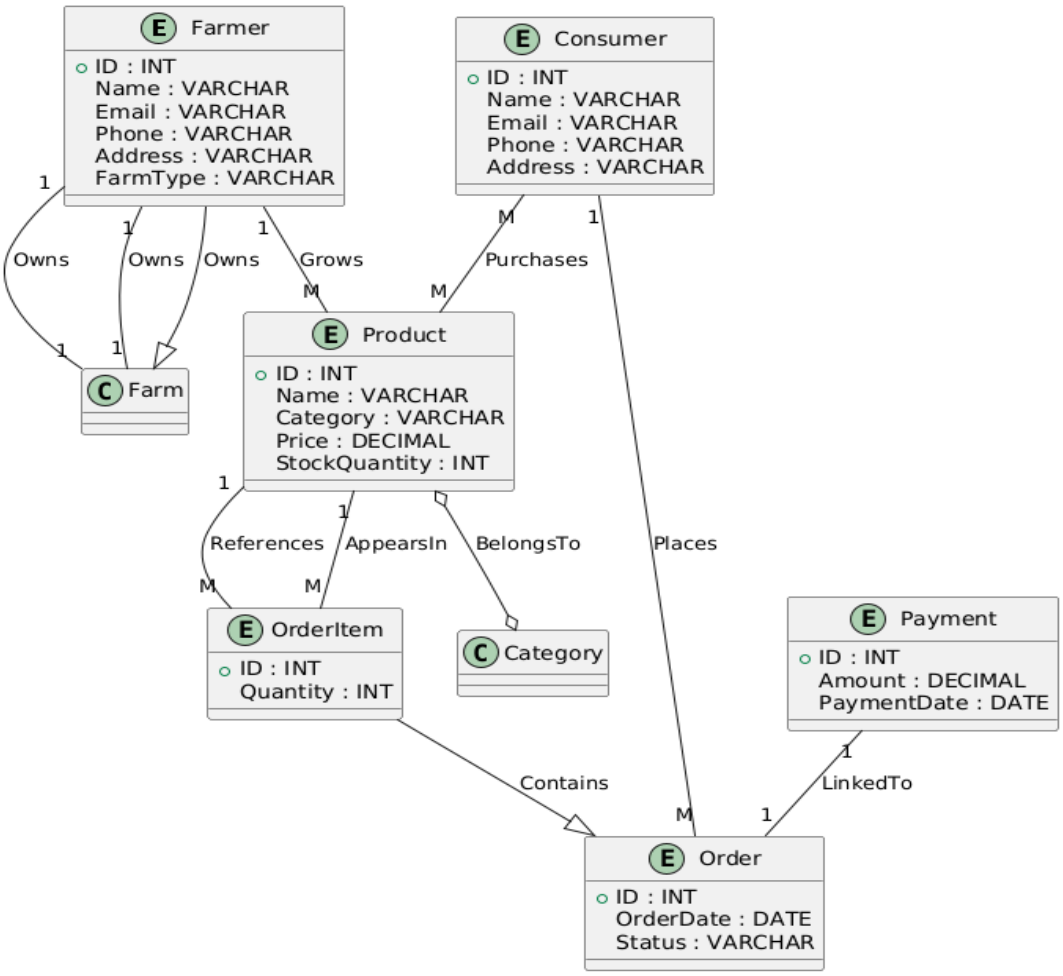
Relationship Types

- **Consumer to Product**: A consumer can purchase multiple products, and each product can be bought by many consumers.

Step 6: Mapping of Multivalued Attributes

- **Product Categories**: A product can belong to multiple categories.

OUPUT DIAGRAM:



Result:
The entity relationship diagram was made successfully by following the steps described above.

EX NO :4 DATA FLOW DIAGRAM

AIM

To draw the Data Flow Diagram for a **Direct Farmer-to-Consumer System** and list the application modules.

ALGORITHM

1. **Open the Tool**: Use a tool like Visual Paradigm or Lucidchart and select a DFD template.
2. **Name the Diagram**: Title it "Direct Farmer-to-Consumer System DFD."
3. **Add External Entities**: Add entities like *Farmer* and *Consumer*.
4. **Add Processes**: Define processes such as *Product Listing*, *Order Management*, and *Payment Processing*.
5. **Add Data Stores**: Include data stores like *Product Inventory*, *Order Database*, and *Payment Records*.
6. **Add Data Flows**: Connect entities, processes, and data stores with arrows showing data flows.
7. **Name the Data Flows**: Label each data flow to represent information like *Product Details*, *Order Details*, and *Payment Information*.
8. **Customize**: Adjust colors, fonts, and labels to distinguish each part of the DFD.
9. **Title and Share**: Finalize by adding a title and sharing the DFD.

INPUTS

1. External Entities:

- **Farmer**: Provides *Farmer ID*, *Product Listings* (like Product Name, Description, Quantity, Price).
- **Consumer**: Provides *Consumer ID*, *Order Requests*, and *Payment Information*.

2. Processes:

- **Product Listing**: Adds and updates product details from farmers.
- **Order Management**: Manages orders from consumers, including selection and quantity.
- **Payment Processing**: Handles payments, confirming transactions.

3. Data Stores:

- **Product Inventory**: Stores *Product Details* (Product ID, Name, Quantity, Price).
- **Order Database**: Stores *Order Information* (Order ID, Consumer ID, Product ID, Quantity, Status).
- **Payment Records**: Stores *Payment Details* (Order ID, Amount, Payment Method, Status).

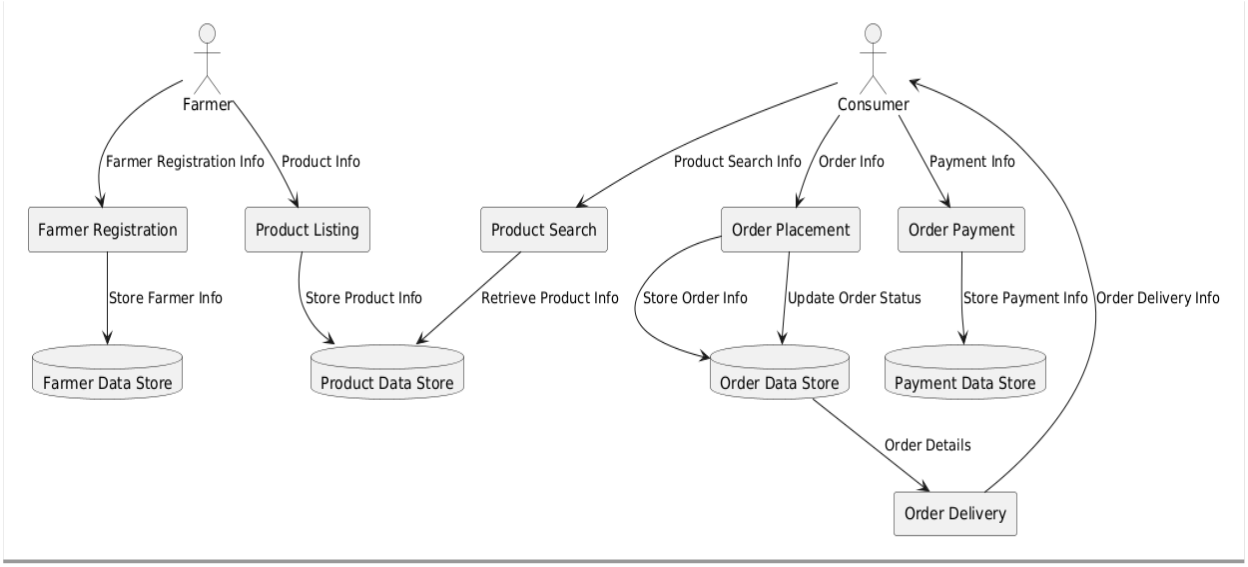
4. Data Flows:

- **Farmer → Product Listing**: Sends *Product Information* (name, price, description).
- **Product Listing → Product Inventory**: Updates *Inventory Data*.
- **Consumer → Order Management**: Sends *Order Details* (product selection, quantity).
- **Order Management → Order Database**: Stores *Order Data* (Order ID, details).
- **Consumer → Payment Processing**: Sends *Payment Information*.
- **Payment Processing → Payment Records**: Updates *Payment Confirmation* for each transaction.

5. Customization:

- **Title**: "Direct Farmer-to-Consumer System DFD"
- **Colors/Fonts**: Use unique colors and shapes to represent processes, data stores, and entities clearly.

OUTPUT DIAGRAM:



Result:

The Data Flow diagram was made successfully by following the steps described above.

EX NO:5 Use Case Diagram

AIM:
To draw the Use Case Diagram for a system that facilitates direct access between consumers and farmers.

ALGORITHM:

Step 1: Identify Actors

- **Actors**: Identify the key external users interacting with the system.
    - **Farmer**: The entity who grows and sells products.
    - **Consumer**: The entity who buys products directly from farmers.

Step 2: Identify Use Cases

- **Use Cases**: Define the key functionalities the system should provide.
    - **For Farmer**:
        - Register Account
        - List Products
        - Update Product Information
        - Receive Order
        - Deliver Product
    - **For Consumer**:
        - Browse Products
        - Place Order
        - Make Payment
        - View Order Status

Step 3: Connect Actors and Use Cases

- **Farmer**: Connected to "Register Account," "List Products," "Update Product Information," "Receive Order," and "Deliver Product."
- **Consumer**: Connected to "Browse Products," "Place Order," "Make Payment," and "View Order Status."

Step 4: Add System Boundary

- Draw a boundary box around the system to represent the scope.
    - The system boundary defines which use cases belong to the **Farmer-Consumer Direct Access System**.

Step 5: Define Relationships

- Define relationships such as Include or Extend if applicable.
    - **Make Payment** may include **Payment Confirmation**.
    - **Place Order** may extend **Receive Order** (for confirmation or tracking).

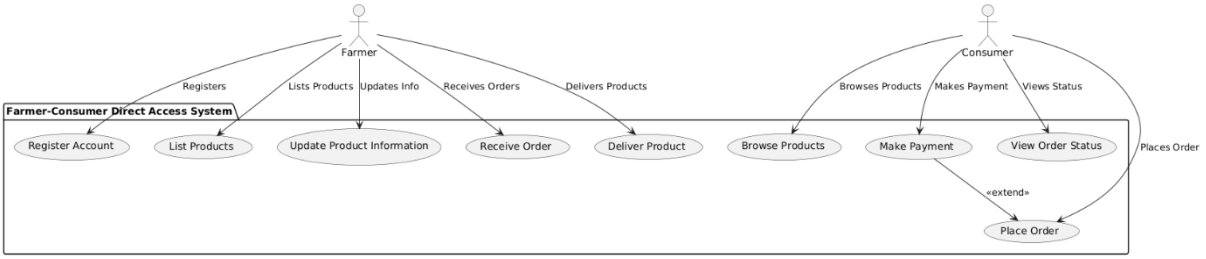
Step 6: Review and Refine

- Review the diagram to ensure that all major actors and use cases are captured.
    - Refine use case names and relationships as needed.

Step 7: Validate

- Validate the diagram with stakeholders (farmers, consumers) to ensure the system covers all required functionality.
    - Ensure that the actors' needs and system's use cases align.

OUTPUT:



Result :

The use case diagram has been created successfully by following the steps given.

EX NO:6 ACTIVITY DIAGRAM

AIM:

To draw the Activity Diagram for a Farmer and Customer Management System.

ALGORITHM:

1. **Identify the Initial and Final States**:
    - **Initial State**: System starts in an "Idle" state.
    - **Final State**: The system ends in the "Order Completed" or "Payment Confirmed" state.
2. **Identify Intermediate Activities**:
    - **Product Selection**: Customer browses and selects products from the list.
    - **Order Placement**: Customer places an order.
    - **Payment Processing**: Customer enters payment details and completes payment.
    - **Delivery Assignment**: Farmer or system assigns a delivery method.
3. **Identify Conditions or Constraints**:
    - **Condition**: Customer can only complete an order if payment is successful.
    - **Constraint**: Order can only proceed if products are available in stock.
4. **Draw the Diagram with Appropriate Notations**:
    - Draw states as rectangles with rounded corners.
    - Use arrows to show transitions between states based on conditions or actions (e.g., from "Order Placement" to "Payment Processing").
    - Label transitions with conditions or events triggering them (e.g., "Payment Success").

INPUT DETAILS FOR THE DIAGRAM:

1. **Activities**:
    - **Product Selection**: Customer selects products and provides order details.
    - **Making Payment**: Customer enters payment details and completes payment.

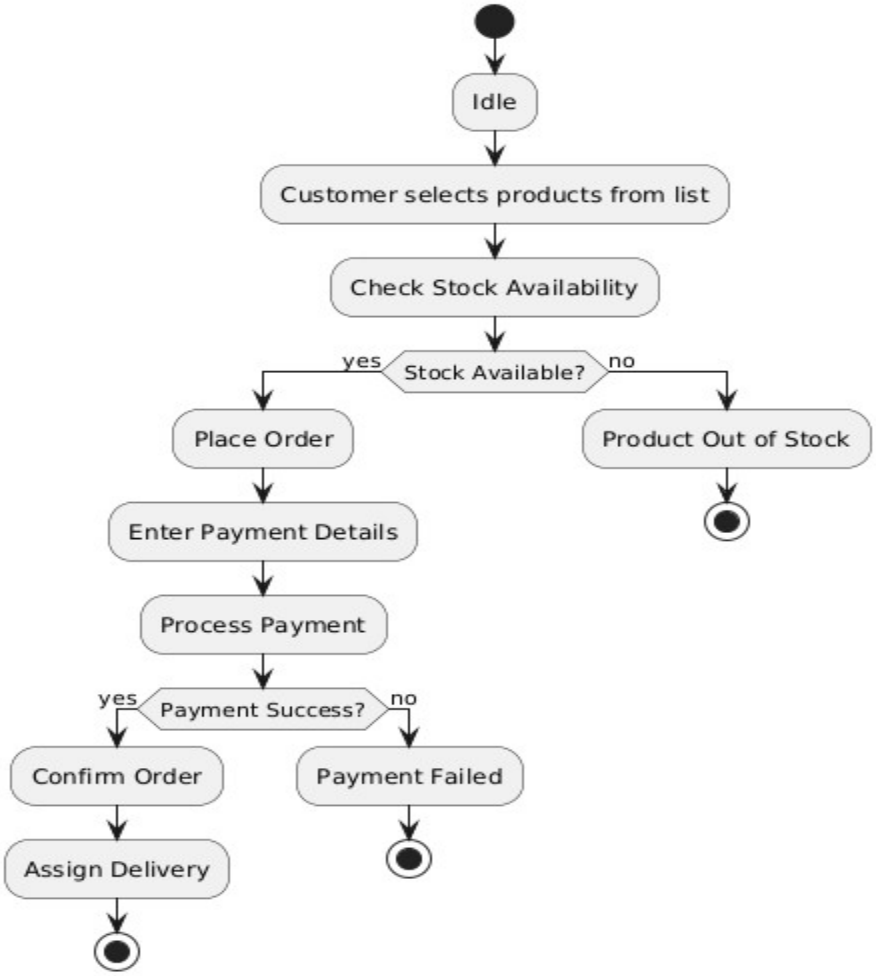
- **Order Confirmation**: System confirms the order if payment is successful.
- **Delivery Assignment**: System assigns delivery for the order.

2. **Decision Points**:
   - **Payment Success?**: System checks if payment was successful or failed.
   - **Stock Availability?**: System checks product availability before confirming the order.
3. **Guards**:
   - **Payment Validity**: Ensures payment is successful before confirming the order.
   - **Stock Availability**: Ensures products are in stock before proceeding with the order.
4. **Parallel Activities**:
   - **Order Placement and Payment**: Order confirmation and payment can proceed in parallel (e.g., stock availability check can happen while the customer is making payment).
   - **Delivery Assignment and Order Confirmation**: Delivery assignment can proceed while customer receives order confirmation after successful payment.
5. **Conditions**:
   - **Successful Payment**: Transition from Order Placement to Order Confirmation happens only if the payment is successful.
   - **Product Availability**: Order can only proceed if the selected products are in stock.

SAMPLE OUTPUT:

- **Initial State**: System starts as idle.
- **Activity Flow**:
  - Customer selects products from the list.
  - System checks if products are available.
  - If available, customer proceeds with payment.
  - System checks payment success.
  - If payment is successful, system confirms the order and assigns delivery.

SAMPLE OUTPUT:



Result :

The Activity diagram has been created successful following the steps given.

EXNO 7 STATE CHART DIAGRAM

AIM:

To draw the State Chart Diagram for a Direct Access between Farmer and Consumer System.

ALGORITHM:

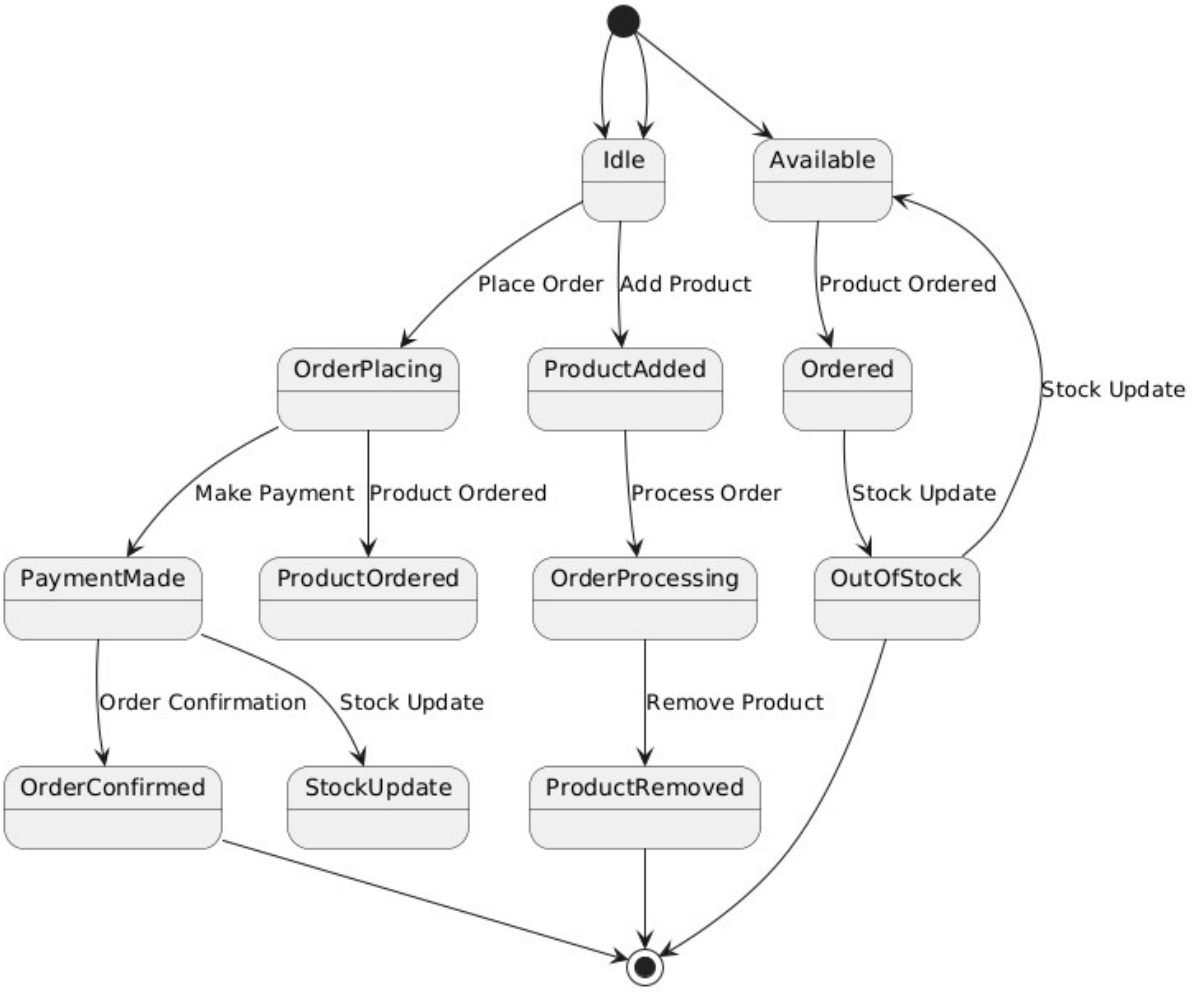
1. **Identify Important Objects to Be Analyzed**:
    - **Consumer**: Tracks the consumer's order and payment status.
    - **Product**: Tracks the availability and order status of the product.
    - **Farmer**: Tracks the status of farmer activities (e.g., adding products, handling orders).
2. **Identify the States**:
    - **Consumer States**:
        - **Idle**: Consumer has not placed an order.
        - **Order Placed**: Consumer has selected and placed an order.
        - **Payment Made**: Consumer has completed payment.
        - **Order Confirmed**: Consumer's order is confirmed, ready for delivery.
    - **Product States**:
        - **Available**: Product is listed and ready for purchase.
        - **Ordered**: Product has been ordered by a consumer.
        - **Out of Stock**: Product is unavailable for order until restocked.
    - **Farmer States**:
        - **Idle**: Farmer has no current orders to process.
        - **Product Added**: Farmer lists a product in the system.
        - **Order Processing**: Farmer is preparing or fulfilling an order.
        - **Product Removed**: Farmer removes a product when it's no longer available.

3. **Identify the Events**:
    - For **Consumer**:
        - **Place Order**: Consumer selects products and places an order.
        - **Make Payment**: Consumer completes payment.
        - **Order Confirmation**: Order is confirmed after payment and farmer's acceptance.
    - For **Product**:
        - **Product Ordered**: Product is ordered by a consumer.
        - **Stock Update**: Product availability is updated by the farmer.
    - For **Farmer**:
        - **Add Product**: Farmer lists a product.
        - **Process Order**: Farmer prepares and confirms order.
        - **Remove Product**: Farmer removes product when out of stock.

INPUTS:

1. **Objects**:
    - Consumer
    - Product
    - Farmer
2. **States**:
    - **Consumer**: Idle, Order Placed, Payment Made, Order Confirmed
    - **Product**: Available, Ordered, Out of Stock
    - **Farmer**: Idle, Product Added, Order Processing, Product Removed
3. **Events**:
    - **Consumer**: Place Order, Make Payment, Order Confirmation
    - **Product**: Product Ordered, Stock Update
    - **Farmer**: Add Product, Process Order, Remove Product

OUPUT DIAGRAM:



Result :

The State Chart diagram has been created successfully by following the steps given.

NO:8 SEQUENCE DIAGRAM

Aim:

To draw the Sequence Diagram for a Direct Access between Farmer and Consumer System.

ALGORITHM:

1. **Identify the Scenario**:
    - Define a specific scenario, such as **Consumer Purchasing a Product** or **Farmer Adding a Product**.
2. **List the Participants**:
    - **Consumer**: The buyer browsing and purchasing products.
    - **System**: The platform that manages product listings, orders, and communication.
    - **Farmer**: The seller adding products and handling orders.
    - **Payment Gateway**: The service handling payment processing.
3. **Define Lifelines**:
    - Create vertical dashed lines for each participant: **Consumer**, **System**, **Farmer**, and **Payment Gateway**.
4. **Arrange Lifelines**:
    - Place participants in the order of interaction: **Consumer**, **System**, **Farmer**, and **Payment Gateway**.
5. **Add Activation Bars**:
    - Draw activation bars (rectangles) to show when each object is actively participating in the process.
6. **Draw Messages**:
    - Add horizontal arrows to show the interactions between participants, such as **View Products**, **Place Order**, **Process Payment**, etc.
7. **Include Return Messages**:
    - Add dashed arrows for return messages like **Order Confirmation**, **Payment Success**, and **Order Shipped**.

8. **Indicate Timing and Order**:
    - Arrange interactions in the chronological sequence from top to bottom.
9. **Include Conditions and Loops**:
    - Use if conditions or loops, for example, "If payment is successful, confirm order."
10. **Consider Parallel Execution**:
    - If certain actions occur in parallel, use separate lifelines to represent this.
11. **Review and Refine**:
    - Ensure all interactions are accurately captured and arranged in correct sequence.
12. **Add Annotations and Comments**:
    - Include comments to clarify actions as needed.
13. **Document Assumptions and Constraints**:
    - List any assumptions like "Payment must be confirmed before order fulfillment".
14. **Use a Tool to Create a Neat Sequence Diagram**:
    - Use a diagramming tool to visually organize and refine the diagram.

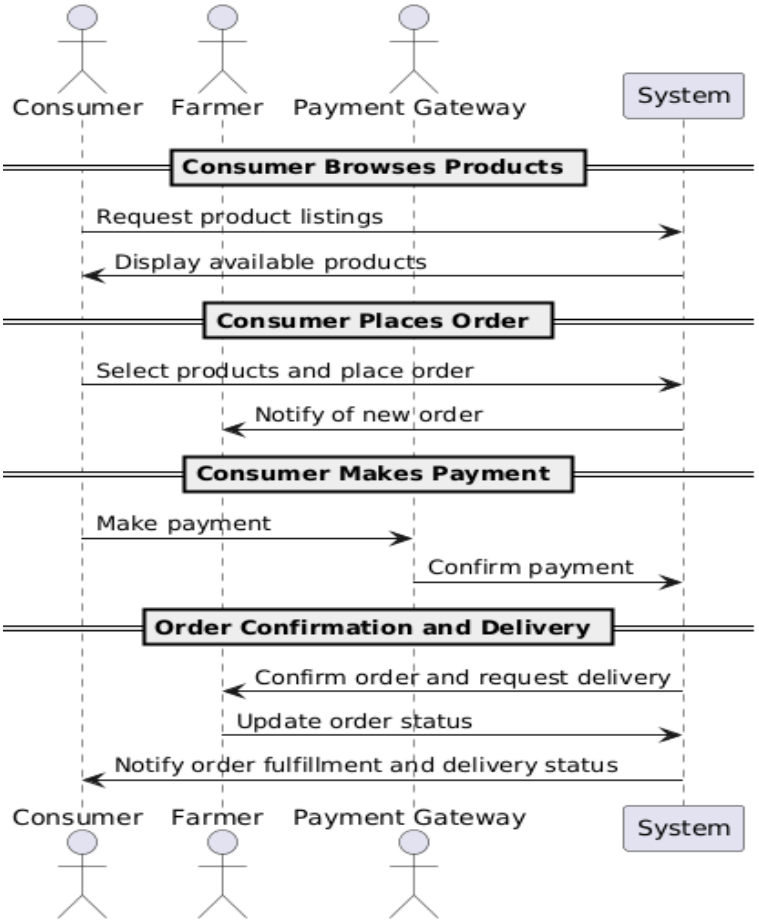
INPUTS:

1. **Objects Taking Part in the Interaction**:
    - **Consumer**: Interacts with the system to view, purchase, and confirm orders.
    - **System**: Manages product availability, order processing, payment, and communication.
    - **Farmer**: Lists products, confirms orders, and manages delivery.
    - **Payment Gateway**: Processes consumer payments securely.
2. **Message Flows Among the Objects**:
    - **Consumer → System**: View Product Listings.
    - **System → Consumer**: Display Products.
    - **Consumer → System**: Place Order.
    - **System → Farmer**: Notify of New Order.

- **Consumer → Payment Gateway**: Make Payment.
- **Payment Gateway → System**: Confirm Payment.
- **System → Farmer**: Confirm Order and Request Delivery.
- **Farmer → System → Consumer**: Confirm Order Fulfillment and Delivery Status.

3. **The Sequence in Which the Messages Are Flowing**:
   - 1st: Consumer → System: View product listings.
   - 2nd: System → Consumer: Display available products.
   - 3rd: Consumer → System: Place an order.
   - 4th: System → Farmer: Notify farmer of new order.
   - 5th: Consumer → Payment Gateway: Make payment.
   - 6th: Payment Gateway → System: Confirm payment.
   - 7th: System → Farmer: Confirm order and initiate delivery process.
   - 8th: Farmer → System → Consumer: Notify consumer of order fulfillment and delivery status.
4. **Object Organization**:
   - **Left to Right Arrangement**: Consumer (leftmost), System, Farmer, Payment Gateway (rightmost).
   - **Lifelines**: Vertical dashed lines with activation bars showing each object's active involvement.

OUTPUT DIAGRAM:



Result :

The Sequence diagram has been created successfully by following the steps given

EX NO:9 Collaboration Diagaram

AIM:

To Draw the Collaboration Diagram for Farmer and Consumer Management System

ALGORITHM:

Step 1: Identify Objects/Participants

- Consumer: The user who interacts with the system to place orders and make payments.
- System: The central platform that manages the interaction between the consumer and the farmer.
- Farmer: The seller providing the products and fulfilling the consumer's order.
- Payment Gateway: A third-party service that processes payments made by the consumer.

Step 2: Define Interactions

- Consumer → System: Request Product Listings
- System → Consumer: Return Available Products
- Consumer → System: Select Products and Place Order
- System → Farmer: Notify Farmer of New Order
- Consumer → Payment Gateway: Make Payment
- Payment Gateway → System: Payment Confirmation
- System → Farmer: Confirm Order and Request Delivery
- Farmer → System: Update Order Status
- System → Consumer: Notify Order Fulfillment and Delivery Status

Step 3: Add Messages

Assign numbered messages to each interaction:

1. Message 1: Consumer → System: Request Product Listings
2. Message 2: System → Consumer: Return Available Products
3. Message 3: Consumer → System: Select Products and Place Order
4. Message 4: System → Farmer: Notify Farmer of New Order

5. Message 5: Consumer → Payment Gateway: Make Payment
6. Message 6: Payment Gateway → System: Payment Confirmation
7. Message 7: System → Farmer: Confirm Order and Request Delivery
8. Message 8: Farmer → System: Update Order Status
9. Message 9: System → Consumer: Notify Order Fulfillment and Delivery Status

Step 4: Consider Relationships

- Draw lines connecting the objects (e.g., Consumer → System or System → Farmer).
- Indicate message flow with numbered arrows between the objects.

Step 5: Document the Collaboration Diagram with Explanations

- Message 1: The Consumer sends a request to the System for available products.
- Message 2: The System returns the list of available products to the Consumer.
- Message 3: The Consumer selects the products and places the order.
- Message 4: The System notifies the Farmer about the new order.
- Message 5: The Consumer makes a payment via the Payment Gateway.
- Message 6: The Payment Gateway confirms the payment to the System.
- Message 7: The System confirms the order to the Farmer and requests delivery.
- Message 8: The Farmer updates the System with the delivery status of the order.
- Message 9: The System notifies the Consumer about the order fulfillment and delivery status.

INPUTS:

1. Objects Taking Part in the Interaction:

- Consumer: The user who places the order, makes payment, and receives products.
- System: The central platform that manages orders, payments, and interactions.
- Farmer: The seller who provides the products and fulfills the orders.
- Payment Gateway: The service that processes the payment transactions.

2. Message Flows Among the Objects:

- Consumer → System: Request Product Listings
- System → Consumer: Return Available Products
- Consumer → System: Select Products and Place Order

- System → Farmer: Notify Farmer of New Order
- Consumer → Payment Gateway: Make Payment
- Payment Gateway → System: Payment Confirmation
- System → Farmer: Confirm Order and Request Delivery
- Farmer → System: Update Order Status
- System → Consumer: Notify Order Fulfillment and Delivery Status

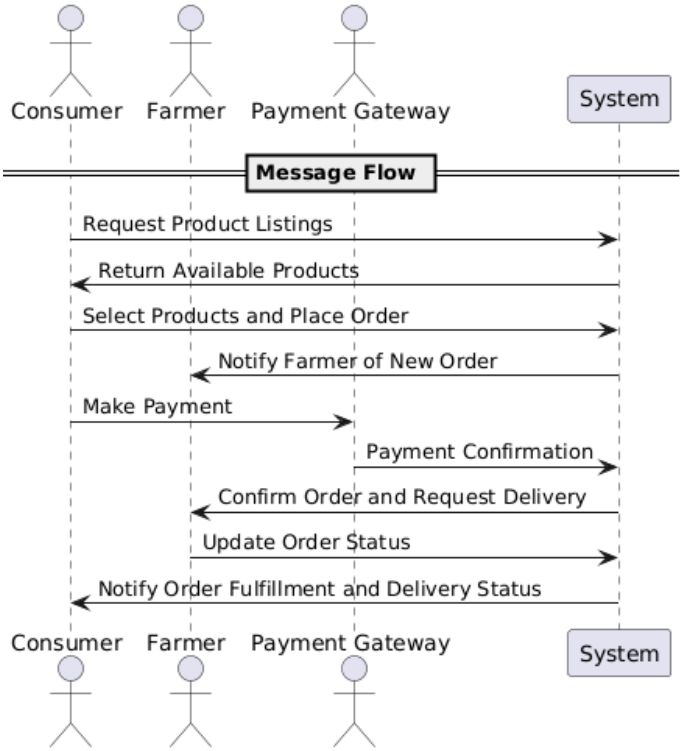
3. The Sequence in Which the Messages Are Flowing:

1. Message 1: Consumer → System: Request Product Listings
2. Message 2: System → Consumer: Return Available Products
3. Message 3: Consumer → System: Select Products and Place Order
4. Message 4: System → Farmer: Notify Farmer of New Order
5. Message 5: Consumer → Payment Gateway: Make Payment
6. Message 6: Payment Gateway → System: Payment Confirmation
7. Message 8: Farmer → System: Update Order Status
8. Message 9: System → Consumer: Notify Order Fulfillment and Delivery Status

4. Object Organization:

- Left to Right Arrangement:
  - Consumer (leftmost), followed by System, then Farmer, and Payment Gateway (rightmost).
- Lifelines: Vertical dashed lines represent the timeline for each object.
- Message Flow: Draw horizontal arrows between objects to show the flow of messages.
- Numbering: Number the messages to indicate the order of interaction.

OUTPUT:



Result :

The Collaboration diagram has been created successfully by following the steps given.

EX NO:10 Class Diagram

AIM:

To Draw the **Class Diagram** for the **Farmer and Consumer Management System**

ALGORITHM:

Step 1: Identify Classes

- **Consumer**
- **Farmer**
- **Product**
- **Order**
- **Payment**

Step 2: List Attributes and Methods

- **Consumer**:
  - **Attributes**: consumerID, name, contact, orderHistory
  - **Methods**: viewProducts(), placeOrder(), makePayment()
- **Farmer**:
  - **Attributes**: farmerID, name, contact, productList
  - **Methods**: addProduct(), updateProduct(), fulfillOrder()
- **Product**:
  - **Attributes**: productID, name, description, price, stockQuantity
  - **Methods**: updateStock(), checkAvailability()
- **Order**:
  - **Attributes**: orderID, consumerID, productID, orderDate, quantity, status
  - **Methods**: createOrder(), cancelOrder(), updateStatus()
- **Payment**:
  - **Attributes**: paymentID, orderID, amount, paymentDate, paymentStatus
  - **Methods**: processPayment(), confirmPayment()

Step 3: Identify Relationships

- **Consumer → Order**: A consumer places an order.
- **Order → Product**: An order contains a product.
- **Farmer → Product**: A farmer manages and sells products.
- **Order → Payment**: A payment is linked to an order.

Step 4: Create Class Boxes

- Draw rectangular boxes for each class, label them accordingly (Consumer, Farmer, Product, Order, Payment).

Step 5: Add Attributes and Methods

- Inside each class box, list the attributes and methods.

Step 6: Draw Relationships

- Use lines to connect classes that have relationships (e.g., **Consumer → Order**, **Order → Product**, etc.).

Step 7: Label Relationships

- Add appropriate relationship labels (e.g., "places", "contains", "manages", "linked to").

Step 8: Review and Refine

- Ensure all relevant classes, attributes, methods, and relationships are captured correctly.

Step 9: Use Tools for Digital Drawing

- Use diagramming tools like Lucidchart, Visual Paradigm, or UMLet to create a clean, professional class diagram.

INPUTS:

1. Class Names:

- **Consumer**
- **Farmer**
- **Product**
- **Order**
- **Payment**

2. Attributes:

- **Consumer**:
  - **consumerID** (String)
  - **name** (String)
  - **contact** (String)
  - **orderHistory** (List of Orders)
- **Farmer**:
  - **farmerID** (String)
  - **name** (String)
  - **contact** (String)
  - **productList** (List of Products)
- **Product**:
  - **productID** (String)
  - **name** (String)
  - **description** (String)
  - **price** (Float)
  - **stockQuantity** (Integer)
- **Order**:
  - **orderID** (String)
  - **consumerID** (String)
  - **productID** (String)
  - **orderDate** (Date)
  - **quantity** (Integer)
  - **status** (String)
- **Payment**:
  - **paymentID** (String)
  - **orderID** (String)
  - **amount** (Float)
  - **paymentDate** (Date)

- **paymentStatus** (String)

3. Methods:

- **Consumer**:
  - **viewProducts()**
  - **placeOrder()**
  - **makePayment()**
- **Farmer**:
  - **addProduct()**
  - **updateProduct()**
  - **fulfillOrder()**
- **Product**:
  - **updateStock()**
  - **checkAvailability()**
- **Order**:
  - **createOrder()**
  - **cancelOrder()**
  - **updateStatus()**
- **Payment**:
  - **processPayment()**
  - **confirmPayment()**

4. Visibility Notation:

- **Public**: +
  - Accessible from any other class.
  - Example: viewProducts(), processPayment()
- **Private**: -
  - Only accessible within the class itself.
  - Example: paymentStatus (in Payment class)
- **Protected**: #

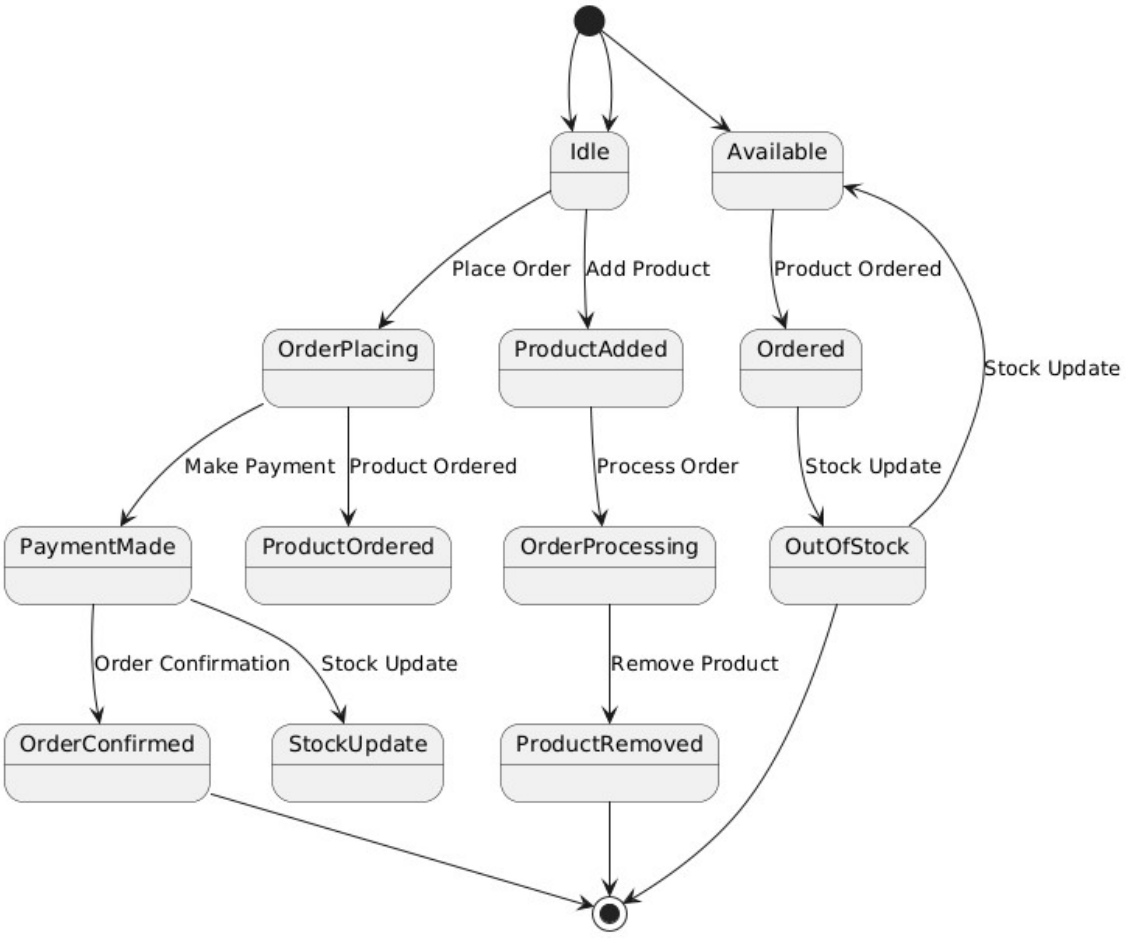
- Accessible within the class and by subclasses.
- Example: consumerID (in Consumer class)

- **Package-Private (default)**: No symbol
  - Accessible within the same package.
  - Example: productList (in Farmer class)

SAMPLE OUTPUT:

1. **Classes**:
   - **Consumer**
   - **Farmer**
   - **Product**
   - **Order**
   - **Payment**
2. **Attributes & Methods**:
   - Listed inside the class boxes as described above.
3. **Relationships**:
   - **Consumer → Order**: A consumer places an order.
   - **Order → Product**: An order contains a product.
   - **Farmer → Product**: A farmer manages and sells products.
   - **Order → Payment**: A payment is linked to an order.

OUTPUT:



Result:

The Class diagram has been created successfully by following the steps given.

IMPLEMENTATION CODE:

```
from datetime import datetime

# Class to represent a Consumer
class Consumer:
    def __init__(self, consumer_id, name, contact):
        self.consumer_id = consumer_id
        self.name = name
        self.contact = contact
        self.order_history = []

    def view_products(self, products):
        print(f"{self.name} is viewing products:")
        for product in products:
            print(f"ID: {product.product_id}, Name: {product.name}, Price:
{product.price}, Stock: {product.stock_quantity}")

    def place_order(self, product, quantity):
        if product.check_availability(quantity):
            order = Order(self, product, quantity)
            self.order_history.append(order)
            return order
        else:
            print(f"Product {product.name} is not available in the requested quantity.")
            return None

    def make_payment(self, order):
        payment = Payment(self, order)
        payment.process_payment()
```

```
        return payment
    # Class to represent a Farmer
    class Farmer:
        def __init__(self, farmer_id, name, contact):
            self.farmer_id = farmer_id
            self.name = name
            self.contact = contact
            self.product_list = []
        def add_product(self, product):
            self.product_list.append(product)
        def update_product(self, product, new_price, new_stock):
            product.update_stock(new_price, new_stock)

    # Class to represent a Product
    class Product:
        def __init__(self, product_id, name, price, stock_quantity):
            self.product_id = product_id
            self.name = name
            self.price = price
            self.stock_quantity = stock_quantity

        def update_stock(self, new_price, new_stock):
            self.price = new_price
            self.stock_quantity = new_stock
            print(f"Product {self.name} updated. New price: {self.price}, New stock:
{self.stock_quantity}")

        def check_availability(self, quantity):
            return self.stock_quantity >= quantity
```

```
# Class to represent an Order
class Order:
    def __init__(self, consumer, product, quantity):
        self.order_id = f"ORD{datetime.now().strftime('%Y%m%d%H%M%S')}"
        self.consumer = consumer
        self.product = product
        self.quantity = quantity
        self.order_date = datetime.now()
        self.status = "Pending"

    def cancel_order(self):
        self.status = "Cancelled"
        print(f"Order {self.order_id} has been cancelled.")

# Class to represent a Payment
class Payment:
    def __init__(self, consumer, order):
        self.payment_id = f"PAY{datetime.now().strftime('%Y%m%d%H%M%S')}"
        self.consumer = consumer
        self.order = order
        self.amount = order.product.price * order.quantity
        self.payment_date = datetime.now()
        self.payment_status = "Pending"

    def process_payment(self):
        # Simulate payment processing (In real-world scenario, this would interact with a payment gateway)
        self.payment_status = "Successful"
```

```
            print(f"Payment of {self.amount} for Order {self.order.order_id} processed
successfully.")
            return self.payment_status

        # Main flow (example usage)
        if __name__ == "__main__":
            # Create products
            apple = Product("P001", "Apple", 1.5, 100)
            banana = Product("P002", "Banana", 0.8, 50)

            # Create farmers
            farmer1 = Farmer("F001", "John Smith", "9876543210")
            farmer1.add_product(apple)
            farmer1.add_product(banana)

            # Create a consumer
            consumer1 = Consumer("C001", "Jane Doe", "1234567890")

            # Consumer views available products
            consumer1.view_products(farmer1.product_list)

            # Consumer places an order
            print(f"\n{consumer1.name} places an order for 3 Apples...")
            order1 = consumer1.place_order(apple, 3)

            if order1:
                # Consumer makes payment for the order
                payment1 = consumer1.make_payment(order1)
```

```
            # Update stock after order
            apple.update_stock(apple.price, apple.stock_quantity - 3)

        # Farmer updates product price and stock
        print(f"\nFarmer {farmer1.name} updates product details...")
        farmer1.update_product(apple, 2.0, 150)

        # Consumer places another order after product update
        print(f"\n{consumer1.name} places another order for 5 Apples after price update...")
        order2 = consumer1.place_order(apple, 5)
        if order2:
            payment2 = consumer1.make_payment(order2)
            apple.update_stock(apple.price, apple.stock_quantity - 5)
```

OUTPUT:

```
Jane Doe is viewing products:
ID: P001, Name: Apple, Price: 1.5, Stock: 100
ID: P002, Name: Banana, Price: 0.8, Stock: 50

Jane Doe places an order for 3 Apples...
Payment of 4.5 for Order ORD202411211235987 processed successfully.

Farmer John Smith updates product details...
Product Apple updated. New price: 2.0, New stock: 150

Jane Doe places another order for 5 Apples after price update...
Payment of 10.0 for Order ORD202411211236987 processed successfully.
Product Apple updated. New price: 2.0, New stock: 145
```

EX