17. Inventory Management System

1. Low-level design (LLD) for an inventory management system:

- **Product catalog**: The system will need to store information about the products in the inventory, such as the product name, SKU, description, and current quantity. This will likely involve creating a database or data structure to store this information.
- **Order processing**: The system will need to allow users to place orders for products and update the inventory accordingly. This will involve creating forms and handling the associated server-side logic to process the orders and update the inventory.
- **Stock management**: The system will need to track the current stock levels and alert users when the stock is running low or has run out. This will involve implementing logic to monitor the inventory and generate alerts when necessary.
- **Reporting and analytics**: The system will need to provide reports and analytics on the inventory, such as sales data and product popularity. This will involve implementing functions to query and analyze the data, as well as creating views to display the results.
- **User accounts and authentication**: The system will need to allow users to create accounts, log in, and log out. We will also need to implement measures to ensure the security of user accounts, such as password hashing and potentially two-factor authentication.
- **User interface**: Finally, the system will need to design the user interface for the service, including the layout, navigation, and any necessary graphics or styling. This will involve creating wireframes and mockups, as well as implementing the front-end code to bring the design to life.

Code

```
import java.time.LocalDateTime;
import java.util.*;

// Enum representing Order Status
public enum OrderStatus {
    PENDING,
    PROCESSING,
    SHIPPED,
    DELIVERED,
    CANCELLED
}

// Abstract Factory to create Products and Orders
interface InventoryFactory {
    Product createProduct(String name, String description, double price, int quantity);
    Order createOrder(Customer customer, Map<Product, Integer> products);
}

// Concrete Factory for Inventory Management
public class ConcreteInventoryFactory implements InventoryFactory {
    @Override
    public Product createProduct(String name, String description, double price, int quantity) {
        return new Product(name, description, price, quantity);
    }

    @Override
    public Order createOrder(Customer customer, Map<Product, Integer> products) {
        return new Order(customer, products);
    }
}

// Product Class representing inventory items
public class Product {
    private long id;
    private String name;
    private String description;
    private double price;
    private int quantity;

    // Observers for product (e.g., low inventory notification)
    private List<Observer> observers = new ArrayList<>();

    public Product(String name, String description, double price, int quantity) {
        this.id = IdGenerator.generateId();
        this.name = name;
        this.description = description;
        this.price = price;
        this.quantity = quantity;
    }
    public long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
        notifyObservers();  // Notify when quantity changes (e.g., low stock alert)
    }

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    private void notifyObservers() {
        for (Observer observer : observers) {
            observer.update("Product stock updated for " + name);
        }
    }
}

// Order Class representing customer orders
public class Order {
    private long id;
    private Customer customer;
    private Map<Product, Integer> products;
    private LocalDateTime createdAt;
    private OrderStatus status;
    private List<Observer> observers = new ArrayList<>();

    public Order(Customer customer, Map<Product, Integer> products) {
        this.id = IdGenerator.generateId();
        this.customer = customer;
        this.products = products;
        this.createdAt = LocalDateTime.now();
        this.status = OrderStatus.PENDING;
    }

    public long getId() {
        return id;
    }

    public Customer getCustomer() {
        return customer;
    }

    public Map<Product, Integer> getProducts() {
        return products;
    }

    public LocalDateTime getCreatedAt() {
        return createdAt;
    }

    public OrderStatus getStatus() {
        return status;
    }

    public void setStatus(OrderStatus status) {
        this.status = status;
        notifyObservers();  // Notify observers when status changes
    }

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    private void notifyObservers() {
        for (Observer observer : observers) {
            observer.update("Order " + id + " status changed to " + status);
        }
    }
}

// Customer Class representing customers
public class Customer {
    private long id;
    private String name;
    private String email;
    private String phone;
    private String address;
    public Customer(String name, String email, String phone, String address) {
        this.id = IdGenerator.generateId();
        this.name = name;
        this.email = email;
        this.phone = phone;
        this.address = address;
    }

    public long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getEmail() {
        return email;
    }

    public String getPhone() {
        return phone;
    }

    public String getAddress() {
        return address;
    }
}

// Interface for Observer (e.g., Notifiers for product or order changes)
interface Observer {
    void update(String message);
}

// Command Pattern for handling various operations
interface Command {
    void execute();
}

// Command for adding a Product
class AddProductCommand implements Command {
    private InventoryManagementSystem system;
    private Product product;

    public AddProductCommand(InventoryManagementSystem system, Product product) {
        this.system = system;
        this.product = product;
    }

    @Override
    public void execute() {
        system.addProduct(product);
    }
}

// Command for placing an Order
class PlaceOrderCommand implements Command {
    private InventoryManagementSystem system;
    private Order order;

    public PlaceOrderCommand(InventoryManagementSystem system, Order order) {
        this.system = system;
        this.order = order;
    }

    @Override
    public void execute() {
        system.placeOrder(order);
    }
}

// Strategy Pattern for pricing and discount strategies
interface PricingStrategy {
    double calculatePrice(Product product, int quantity);
}

// Regular pricing strategy
class RegularPricingStrategy implements PricingStrategy {
    @Override
    public double calculatePrice(Product product, int quantity) {
        return product.getPrice() * quantity;
    }
}

// Discounted pricing strategy
class DiscountPricingStrategy implements PricingStrategy {
    private double discountRate;

    public DiscountPricingStrategy(double discountRate) {
        this.discountRate = discountRate;
    }

    @Override
    public double calculatePrice(Product product, int quantity) {
        return (product.getPrice() * quantity) * (1 - discountRate);
    }
}

// InventoryManagementSystem with product, customer, and order management
public class InventoryManagementSystem {
    private Map<Long, Product> products = new HashMap<>();
    private Map<Long, Order> orders = new HashMap<>();
    private Map<Long, Customer> customers = new HashMap<>();

    // Add product to the system
    public void addProduct(Product product) {
        products.put(product.getId(), product);
        System.out.println("Product added: " + product.getName());
    }

    // Get product by ID
    public Product getProduct(long id) {
        return products.get(id);
    }

    // Update product quantity
    public void updateProductQuantity(long productId, int quantity) {
        Product product = products.get(productId);
        if (product != null) {
            product.setQuantity(quantity);
            System.out.println("Product quantity updated for: " + product.getName());
        }
    }

    // Place order in the system
    public void placeOrder(Order order) {
        orders.put(order.getId(), order);
        System.out.println("Order placed: " + order.getId());
    }

    // Get order by ID
    public Order getOrder(long id) {
        return orders.get(id);
    }

    // Update order status
    public void updateOrderStatus(long orderId, OrderStatus status) {
        Order order = orders.get(orderId);
        if (order != null) {
            order.setStatus(status);
        }
    }

    // Add customer to the system
    public void addCustomer(Customer customer) {
        customers.put(customer.getId(), customer);
        System.out.println("Customer added: " + customer.getName());
    }

    // Get customer by ID
    public Customer getCustomer(long id) {
        return customers.get(id);
    }
```

```
}

// ID Generator class for unique ID creation
class IdGenerator {
   private static long currentId = 0;

   public static long generateId() {
      return ++currentId;
   }
}

// Main class to demonstrate the Inventory Management System
public class Main {
   public static void main(String[] args) {
      // Create the factory
      InventoryFactory factory = new ConcreteInventoryFactory();

      // Create products
      Product product1 = factory.createProduct("Laptop", "High-performance laptop", 1200.0, 10);
      Product product2 = factory.createProduct("Smartphone", "Latest smartphone", 800.0, 20);

      // Create customers
      Customer customer1 = new Customer("Alice", "alice@example.com", "1234567890", "123
Main St");
      Customer customer2 = new Customer("Bob", "bob@example.com", "0987654321", "456 Elm
St");

      // Create the inventory system
      InventoryManagementSystem inventorySystem = new InventoryManagementSystem();

      // Add customers to the system
      inventorySystem.addCustomer(customer1);
      inventorySystem.addCustomer(customer2);

      // Add products using command pattern
      Command addProduct1 = new AddProductCommand(inventorySystem, product1);
      Command addProduct2 = new AddProductCommand(inventorySystem, product2);
      addProduct1.execute();
      addProduct2.execute();

      // Create and place an order
      Map<Product, Integer> orderProducts = new HashMap<>();
      orderProducts.put(product1, 2);
      orderProducts.put(product2, 1);

      Order order1 = factory.createOrder(customer1, orderProducts);

      Command placeOrder = new PlaceOrderCommand(inventorySystem, order1);
      placeOrder.execute();
```

```
        // Update order status
        inventorySystem.updateOrderStatus(order1.getId(), OrderStatus.PROCESSING);
    }
}
```

2. In this design, we'll implement the Inventory Management System using various design patterns like Singleton, Factory, Strategy, and Observer. We'll also provide a class diagram, database schema, and Java code for implementation.

Step 1: Design Patterns Used

1. Singleton Pattern:

Used for the Inventory class to ensure only one instance of the inventory system exists.

2. Factory Pattern:

Used to create different types of products (e.g., electronics, clothing, groceries).

3. Observer Pattern:

Used to notify subscribers (e.g., suppliers) when stock goes below a certain threshold.

4. Strategy Pattern:

Used to dynamically change the stock replenishment strategy.

Step 2: Class Diagram

Class Name	Type	Attributes	Methods	Design Pattern
Inventory	Singleton	products: List<Product> observers: List<Observer>	getInstance(): Inventory addProduct(product: Product): void updateProduct(productId: int, qty: int): void removeProduct(productId: int): void checkStock(productId: int): int notifyObservers(productId: int): void addObserver(observer: Observer): void	Singleton, Observer
ProductFactory	Factory	-	createProduct(type: String): Product	Factory
Product	Abstract	id: int name: String price: float quantity: int	getId(): int getName(): String getQuantity(): int setQuantity(qty: int): void restock(qty: int): void	-
ElectronicsProduct	Concrete	warrantyPeriod: int	getWarrantyPeriod(): int	-
GroceryProduct	Concrete	expiryDate: Date	getExpiryDate(): Date	-
ClothingProduct	Concrete	size: String material: String	getSize(): String getMaterial(): String	-
StockReplenisher	Strategy	-	replenish(product: Product): void	Strategy
ThresholdReplenisher	Concrete	-	replenish(product: Product): void	-
DemandBasedReplenisher	Concrete	-	replenish(product: Product): void	-
Observer		-	update(productId: int): void	Observer
Supplier	Concrete	-	update(productId: int): void	-

Step 3: Java Code Implementation

```
Inventory (Singleton)
public class Inventory {
    private static Inventory instance;
    private List<Product> products = new ArrayList<>();
    private List<Observer> observers = new ArrayList<>();

    private Inventory() {}
    public static synchronized Inventory getInstance() {
        if (instance == null) {
            instance = new Inventory();
        }
        return instance;
    }
    public void addProduct(Product product) {
        products.add(product);
    }
    public void updateProduct(int productId, int quantity) {
        for (Product product : products) {
            if (product.getId() == productId) {
                product.setQuantity(quantity);
                notifyObservers(productId);
            }
        }
    }
    public void removeProduct(int productId) {
        products.removeIf(product -> product.getId() == productId);
    }
    public void checkStock(int productId) {
        for (Product product : products) {
            if (product.getId() == productId) {
                System.out.println("Stock level for product: " + product.getQuantity());
            }
        }
    }
    public void addObserver(Observer observer) {
        observers.add(observer);
    }
    private void notifyObservers(int productId) {
        for (Observer observer : observers) {
            observer.update(productId);
        }
    }
}
Product and Product Types
public abstract class Product {
    protected int id;
    protected String name;
    protected float price;
    protected int quantity;
    public int getId() { return id; }
    public String getName() { return name; }
    public int getQuantity() { return quantity; }
    public void setQuantity(int qty) { this.quantity = qty; }
}
public class ElectronicsProduct extends Product {
    private int warrantyPeriod;
    public ElectronicsProduct(int id, String name, float price, int warrantyPeriod) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.warrantyPeriod = warrantyPeriod;
    }
    public int getWarrantyPeriod() {
        return warrantyPeriod;
    }
}
public class GroceryProduct extends Product {
    private Date expiryDate;
    public GroceryProduct(int id, String name, float price, Date expiryDate) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.expiryDate = expiryDate;
    }
    public Date getExpiryDate() {
        return expiryDate;
    }
}
public class ClothingProduct extends Product {
    private String size;
    private String material;
    public ClothingProduct(int id, String name, float price, String size, String material) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.size = size;
        this.material = material;
    }
    public String getSize() {
        return size;
    }
    public String getMaterial() {
        return material;
    }
}
ProductFactory (Factory Pattern)
public class ProductFactory {
    public static Product createProduct(String type, int id, String name, float price) {
        switch (type.toLowerCase()) {
            case "electronics":
                return new ElectronicsProduct(id, name, price, 12); // 12 months warranty
        case "grocery":
          return new GroceryProduct(id, name, price, new Date()); // today's date for now
        case "clothing":
          return new ClothingProduct(id, name, price, "M", "Cotton");
        default:
          throw new IllegalArgumentException("Unknown product type");
      }
  }
}
Observer Pattern
public interface Observer {
  void update(int productId);
}
public class Supplier implements Observer {
  @Override
  public void update(int productId) {
    System.out.println("Supplier notified for product ID: " + productId);
  }
}
Strategy Pattern
public interface StockReplenisher {
  void replenish(Product product);
}
public class ThresholdReplenisher implements StockReplenisher {
  @Override
  public void replenish(Product product) {
    if (product.getQuantity() < 10) {
      System.out.println("Replenishing stock for product: " + product.getName());
    }
  }
}
public class DemandBasedReplenisher implements StockReplenisher {
  @Override
  public void replenish(Product product) {
        System.out.println("Demand-based replenishment for product: " + product.getName());
    }
}
```

This design provides a solid foundation for an Inventory Management System that is scalable and flexible with the use of design patterns like Singleton, Factory, and Strategy