Project Documentation: E-Commerce System

Overview

This project is an E-Commerce system that manages inventory, customer orders, and cart operations. It is designed to demonstrate the integration of a database with Java and the application of various design patterns for maintainable and scalable software. The system supports features like adding items to the inventory, placing customer orders, managing a shopping cart, applying pricing strategies, and simulating payment processing.

The project uses Java as the programming language and SQL Server as the database backend. The design is modular, and the use of design patterns such as Strategy, Singleton, and Factory ensures the code is extensible and easy to maintain.

Features

1. **Database Operations**: Enables adding, updating, and retrieving inventory items and customer orders.
2. **Cart Management**: Provides a shopping cart system for managing items and calculating the total cost dynamically.
3. **Pricing Strategy**: Implements a flexible pricing mechanism through the Strategy pattern.
4. **Payment Processing**: Simulates payment handling with support for multiple payment methods.
5. **Discount System**: Supports the application of discounts dynamically using the Strategy pattern.

Architecture

The project is structured with clearly defined classes and components, all placed in the default package for simplicity.

Database

The database consists of the following tables: - **`inventory`**: - Fields: - `itemId`: Auto-increment primary key. - `itemName`: Name of the product. - `itemCategory`: Category of the product (e.g., electronics, furniture). - `itemPrice`: Price of the product.

- `Orders`:
  - Fields:
    - `orderId`: Auto-increment primary key.
    - `deliveryAddress`: Customer's address.

* `totalAmount`: Total order value.
* `orderType`: Type of order (e.g., regular, express).
* `customerName`: Name of the customer.

Classes and Components

`DBHandler` **Purpose**: Manages database connectivity and operations, such as adding items and orders to the database.

Code:

```
public class DBHandler {
    public static void addItemToDatabase(Connection connection, Item item) {
        String sql = "INSERT INTO inventory (itemName, itemCategory, itemPrice) VALUES (?, ?
        try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
            pstmt.setString(1, item.getName());
            pstmt.setString(2, item.getCategory());
            pstmt.setDouble(3, item.getPrice());
            pstmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Justification: Centralizes database logic, reducing redundancy and improving code maintainability.

`Item` **Purpose**: Represents individual products in the inventory.

Code:

```
public class Item {
    private int itemId;
    private String itemName;
    private String itemCategory;
    private double itemPrice;

    public Item(String itemName, String itemCategory, double itemPrice) {
        this.itemName = itemName;
        this.itemCategory = itemCategory;
        this.itemPrice = itemPrice;
    }

    public String getName() {
        return itemName;
    }
```

```
    public String getCategory() {
        return itemCategory;
    }

    public double getPrice() {
        return itemPrice;
    }
}
```

Justification: Encapsulates product-related data and methods, promoting better object-oriented design.

`CustomerOrder` **Purpose**: Represents customer orders, encapsulating order details like delivery address, total amount, and order type.

Code:

```
public class CustomerOrder {
    private String deliveryAddress;
    private double totalAmount;
    private String orderType;
    private String customerName;

    public CustomerOrder(String deliveryAddress, double totalAmount, String orderType, Strin
        this.deliveryAddress = deliveryAddress;
        this.totalAmount = totalAmount;
        this.orderType = orderType;
        this.customerName = customerName;
    }

    public String getDeliveryAddress() {
        return deliveryAddress;
    }

    public double getTotalAmount() {
        return totalAmount;
    }

    public String getOrderType() {
        return orderType;
    }
}
```

Justification: Organizes order-related data for easier management and retrieval.

`PricingStrategy` **Purpose**: Defines an interface for applying different pricing strategies.

Code:

```
public interface PricingStrategy {
    double calculatePrice(Item item);
}
```

Justification: Allows dynamic implementation of various pricing strategies without modifying the core logic.

`StandardPricing and DiscountPricing` **Purpose**: Implement specific pricing strategies (e.g., standard pricing, discounted pricing).

Code:

```
public class DiscountPricing implements PricingStrategy {
    private double discountPercentage;

    public DiscountPricing(double discountPercentage) {
        this.discountPercentage = discountPercentage;
    }

    @Override
    public double calculatePrice(Item item) {
        return item.getPrice() * (1 - discountPercentage / 100);
    }
}
```

Justification: Encapsulates distinct pricing logic, making it reusable and maintainable.

`CartManager` **Purpose**: Manages the shopping cart operations, including adding items and calculating totals.

Code:

```
public class CartManager {
    private List<Item> cart = new ArrayList<>();

    public void addItemToCart(Item item) {
        cart.add(item);
    }
```

```
    public double calculateTotal() {
        return cart.stream().mapToDouble(Item::getPrice).sum();
    }
}
```

Justification: Simplifies cart management and ensures data consistency.

`PaymentGateway` **Purpose**: Simulates payment processing with support for multiple payment methods.

Code:

```
public class PaymentGateway {
    public boolean processPayment(String paymentMethod, double amount) {
        switch (paymentMethod.toLowerCase()) {
            case "credit card":
                System.out.println("Processing credit card payment of " + amount);
                return true;
            case "paypal":
                System.out.println("Processing PayPal payment of " + amount);
                return true;
            default:
                System.out.println("Unsupported payment method.");
                return false;
        }
    }
}
```

Justification: Handles payment logic independently, making it easy to integrate additional payment methods in the future.

Design Patterns Used

1. **Strategy Pattern**: Enables dynamic selection of pricing strategies.
2. **Factory Pattern**: Simplifies object creation for `Item` and `CustomerOrder`.
3. **Singleton Pattern**: Ensures a single instance for managing the cart and payment gateway.

How to Run

1. Set up the database using the provided SQL script.
2. Update the database connection credentials in the `Main` class.

3. Compile and run the project.

Conclusion

This project showcases an extensible architecture for an E-Commerce system. By leveraging design patterns, it ensures maintainability and scalability, making it a solid foundation for future enhancements.