E-Commerce System Project Documentation

Introduction:

The E-Commerce System is a desktop application designed to manage product listings, customer carts, orders, and payments. This project incorporates five design patterns: Singleton, Factory, Prototype, Builder, and Decorator. Each pattern is utilized to solve specific problems and ensure a modular, scalable, and maintainable codebase.

Class Descriptions and Design Patterns:

1. Singleton Pattern

Purpose:

Ensures that certain classes (e.g., CartManager, DatabaseConnection, and PaymentGateway) have a single instance across the application lifecycle.

Classes:

- **DatabaseConnection:** Manages the database connection to prevent redundant instances and ensure efficient resource management.

  Justification:

  The application frequently interacts with the database, and a Singleton ensures a single, consistent connection.

- **CartManager:** Handles cart operations, such as adding products and calculating totals, while maintaining a single instance for the user session.

  Justification: Cart data must remain consistent across multiple views and operations.

- **PaymentGateway:** Processes payments (e.g., credit card, PayPal) with a single instance for consistent behavior.

- **Justification:** Ensures all payment transactions go through the same gateway instance.

Example Code:

```
public class CartManager {
    private static CartManager instance;
    private List<ProductComponent> cart ;

    private CartManager() {cart = new ArrayList<>();}

    public static CartManager getInstance() {
        if (instance == null) {
            instance = new CartManager();
        }
        return instance;
    }

    public void addProduct(Product product) {
        cart.add(product);
    }

    public List<Product> getCart() {
        return cart;
    }
}
```

Purpose: Creates objects based on specified criteria, promoting code reusability and scalability.

Classes:

- **ProductFactory:** Creates products based on their category (e.g., Electronics, Clothing).

  Justification: Simplifies object creation and separates product instantiation logic.
- **OrderFactory:** Creates order types (e.g., StandardOrder, ExpressOrder) based on customer preferences.

  Justification: Encapsulates the logic for order processing into a single, manageable interface.

Example Code:

```
public class ProductFactory {
    public static Product createProduct(String name, String category, double
price) {
        switch (category.toLowerCase()) {
            case "electronics":
                return new Product(name, "electronics", price);
            case "clothing":
                return new Product(name, "clothing", price);
            case "homeappliances":
                return new Product(name, "home appliances", price);
            default:
                throw new IllegalArgumentException("Unknown category");
        }
    }
}
```

3. Prototype Pattern

Purpose: Allows cloning of objects to create a new instance with the same properties.

Classes:

- **Product:** Implements the `Cloneable` interface to support cloning.
  Justification: Enables duplication of products, e.g., when adding a similar item to

Example Code:

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

    public Product clone(){
        return new Product (this.name, this.category, this.price);
    }
}
```

4. Builder Pattern

Purpose: Constructs complex objects step by step, ensuring clarity and customization.

Classes:

- **OrderBuilder:** Builds customized orders with specific attributes (e.g., delivery type, payment method).
  Justification: Simplifies the construction of complex order objects with optional attributes.

Example Code:

```
public class OrderBuilder {
    private String location;
    private String totalPrice;

    private String type;

    private String customerName;



    public OrderBuilder setLocation(String location) {
        this.location = location;
        return this;
    }

    public OrderBuilder setTotalPrice(double totalPrice) {
        this. totalPrice = totalPrice;
        return this;
    }

public OrderBuilder setType (String type) {
        this. type = type;
        return this;
    }

public OrderBuilder setCustomerName(String customerName) {
        this. customerName = customerName;
        return this;
    }
 public Order build() {
        return new Order(deliveryType, paymentMethod);
    }
}
```

5. Decorator Pattern

Purpose: Dynamically adds new functionalities to objects without modifying their structure.

Classes:

- **Product Decorator:** Adds additional attributes to the product price, such as extended warranty or discount.
  Justification: Allows flexible price modification without altering the core `Product` class.

Example Code:

```java
public class WarrantyDecorator extends ProductDecorator {
    private static final double WARRANTY_COST = 20.0;

    public WarrantyDecorator(ProductComponent product) {
        super(product);
    }

    @Override
    public double getPrice() {
        return super.getPrice() + WARRANTY_COST;
    }

    @Override
    public String getName() {
        return super.getName() + " (With Extended Warranty)";
    }
}
```

Integration of Patterns

1. **Singleton** ensures consistency in shared resources (e.g., CartManager, DatabaseConnection).

2. **Factory** simplifies the creation of objects (products, orders) and encapsulates instantiation logic.
3. **Prototype** allows cloning of products for flexibility in duplication tasks.
4. **Builder** manages the creation of complex orders with optional attributes.
5. **Decorator** enhances functionality dynamically, such as adding extended warranties.

Implementation Details

Database Integration

- **DatabaseConnection Singleton:** Ensures only one connection instance is active.

Example:

Example:

```
public class DatabaseConnection {
    private static DatabaseConnection instance;
    private Connection connection;

    private DatabaseConnection() {
        try {
            connection =
DriverManager.getConnection("jdbc:sqlserver://localhost;databaseName=EComm
erce", "user", "password");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    public static DatabaseConnection getInstance() {
        if (instance == null) {
            instance = new DatabaseConnection();
        }
        return instance;
    }

    public Connection getConnection() {
        return connection;
    }
}

    public Connection getConnection() {
        return connection;
    }}
```

Cart Functionality

- **CartManager Singleton:** Manages the products added to the cart.
- **Decorator for Price Adjustments:** Allows price modifications such as adding warranty.

Conclusion

This project demonstrates the effective use of design patterns to achieve a modular, maintainable, and scalable application. Each pattern was chosen to address specific challenges, ensuring clarity and flexibility in design. The integration of these patterns ensures the application is robust and adaptable to future requirements.