Lucid Vibes

**1. Creational Patterns**

**Singleton Pattern:**

* **Purpose**: Ensures that there is only one instance of Configuration class throughout the application.
* **Implementation**:

java

Copy code

public class Configuration {

private static Configuration instance;

private String config;

private Configuration() {

config = "Default Configuration";

}

public static Configuration getInstance() {

if (instance == null) {

instance = new Configuration();

}

return instance;

}

public String getConfig() {

return config;

}

}

* + **Usage**: Provides global access to configuration settings (config) and ensures that changes made to it are consistent across the application.

**Factory Method Pattern:**

* **Purpose**: Allows dynamic creation of different types of User objects (Admin, Seller, Buyer) without exposing the instantiation logic.
* **Implementation**:

java

Copy code

abstract class User {

public abstract void displayRole();

}

class Admin extends User {

public void displayRole() {

System.out.println("I am an Admin.");

}

}

class Seller extends User {

public void displayRole() {

System.out.println("I am a Seller.");

}

}

class Buyer extends User {

public void displayRole() {

System.out.println("I am a Buyer.");

}

}

class UserFactory {

public User createUser(String type) {

if (type.equals("Admin")) {

return new Admin();

} else if (type.equals("Seller")) {

return new Seller();

} else if (type.equals("Buyer")) {

return new Buyer();

}

return null;

}

}

* + **Usage**: Enables creation of different types of users based on input (Admin, Seller, Buyer), allowing flexibility in user management.

**2. Behavioral Patterns**

**Observer Pattern:**

* **Purpose**: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
* **Implementation**:

java

Copy code

interface Observer {

void update(String message);

}

class UserObserver implements Observer {

private String name;

public UserObserver(String name) {

this.name = name;

}

@Override

public void update(String message) {

System.out.println(name + " received notification: " + message);

}

}

class NotificationService {

private List<Observer> observers = new ArrayList<>();

public void addObserver(Observer observer) {

observers.add(observer);

}

public void removeObserver(Observer observer) {

observers.remove(observer);

}

public void notifyObservers(String message) {

for (Observer observer : observers) {

observer.update(message);

}

}

}

* + **Usage**: Allows users (UserObserver) to subscribe to notifications (NotificationService) and receive updates when events occur (e.g., new product availability).

**Interpreter Pattern (Command):**

* **Purpose**: Defines a grammatical representation for a language and provides an interpreter to interpret sentences in that language.
* **Implementation**:

java

Copy code

interface Expression {

void interpret(OrderContext context);

}

class BuyExpression implements Expression {

private String product;

public BuyExpression(String product) {

this.product = product;

}

@Override

public void interpret(OrderContext context) {

context.addOrder("Buying " + product);

}

}

class SellExpression implements Expression {

private String product;

public SellExpression(String product) {

this.product = product;

}

@Override

public void interpret(OrderContext context) {

context.addOrder("Selling " + product);

}

}

class OrderContext {

private List<String> orders = new ArrayList<>();

public void addOrder(String order) {

orders.add(order);

}

public List<String> getOrders() {

return orders;

}

}

class OrderInterpreter {

public void interpret(String command, OrderContext context) {

String[] parts = command.split(" ");

String action = parts[0];

String product = parts[1];

Expression expression = null;

if (action.equalsIgnoreCase("buy")) {

expression = new BuyExpression(product);

} else if (action.equalsIgnoreCase("sell")) {

expression = new SellExpression(product);

}

if (expression != null) {

expression.interpret(context);

}

}

}

* + **Usage**: Parses commands (buy Laptop, sell Phone) and interprets them into actionable orders (OrderContext), allowing flexible handling of user commands.

**3. Structural Patterns**

**Decorator Pattern:**

* **Purpose**: Allows behavior to be added to individual objects, dynamically, without affecting the behavior of other objects from the same class.
* **Implementation**:

java

Copy code

interface Product {

String getDescription();

double getCost();

}

class BasicProduct implements Product {

private String name;

private double price;

public BasicProduct(String name, double price) {

this.name = name;

this.price = price;

}

@Override

public String getDescription() {

return name;

}

@Override

public double getCost() {

return price;

}

}

abstract class ProductDecorator implements Product {

protected Product decoratedProduct;

public ProductDecorator(Product decoratedProduct) {

this.decoratedProduct = decoratedProduct;

}

public String getDescription() {

return decoratedProduct.getDescription();

}

public double getCost() {

return decoratedProduct.getCost();

}

}

class GiftWrap extends ProductDecorator {

public GiftWrap(Product decoratedProduct) {

super(decoratedProduct);

}

public String getDescription() {

return decoratedProduct.getDescription() + ", with Gift Wrap";

}

public double getCost() {

return decoratedProduct.getCost() + 5.00;

}

}

class ExtendedWarranty extends ProductDecorator {

public ExtendedWarranty(Product decoratedProduct) {

super(decoratedProduct);

}

public String getDescription() {

return decoratedProduct.getDescription() + ", with Extended Warranty";

}

public double getCost() {

return decoratedProduct.getCost() + 20.00;

}

}

* + **Usage**: Adds additional features (GiftWrap, ExtendedWarranty) to products (BasicProduct) dynamically, enhancing product descriptions and pricing.

**Facade Pattern:**

* **Purpose**: Provides a unified interface to a set of interfaces in a subsystem, making it easier to use.
* **Implementation**:

java

Copy code

class ProductInventory {

public void checkInventory(String product) {

System.out.println("Checking inventory for " + product);

}

}

class PaymentGateway {

public void processPayment(String paymentType) {

System.out.println("Processing " + paymentType + " payment");

}

}

class ShippingService {

public void shipProduct(String product) {

System.out.println("Shipping " + product);

}

}

class ECommerceFacade {

private ProductInventory inventory;

private PaymentGateway payment;

private ShippingService shipping;

public ECommerceFacade() {

inventory = new ProductInventory();

payment = new PaymentGateway();

shipping = new ShippingService();

}

public void placeOrder(String product, String paymentType) {

inventory.checkInventory(product);

payment.processPayment(paymentType);

shipping.shipProduct(product);

}

}

* + **Usage**: Provides a simplified interface (ECommerceFacade) to interact with complex subsystems (ProductInventory, PaymentGateway, ShippingService), encapsulating their functionality.

**Project Working Explanation**

1. **Singleton Pattern**:
   * The Configuration class ensures there is only one instance of configuration settings (config). It provides global access to these settings via getInstance() method.
2. **Factory Method Pattern**:
   * The UserFactory allows the creation of different types of users (Admin, Seller, Buyer) based on input (type). Each user type (Admin, Seller, Buyer) extends User and implements displayRole() to specify their role.
3. **Observer Pattern**:
   * The NotificationService allows UserObserver objects to subscribe (addObserver()) and unsubscribe (removeObserver()) for notifications. When a notification (notifyObservers()) is triggered, all registered observers are notified (update() method).
4. **Interpreter Pattern (Command)**:
   * The OrderInterpreter parses user commands (buy Laptop, sell Phone) using interpret() method. It splits the command into action (buy or sell) and product, then interprets them into specific orders (BuyExpression, SellExpression) added to OrderContext.
5. **Decorator Pattern**:
   * The Product interface defines basic product operations (getDescription(), getCost()). BasicProduct implements Product, providing base functionality.
   * ProductDecorator abstract class extends Product, allowing dynamic addition (GiftWrap, ExtendedWarranty) of functionalities (getDescription(), getCost()).
6. **Facade Pattern**:
   * The ECommerceFacade simplifies interaction with subsystems (ProductInventory, PaymentGateway, ShippingService). It provides a unified placeOrder() method that internally invokes subsystem methods (checkInventory(), processPayment(), shipProduct()).

**Integration and Usage**

* **Main Class (ECommerceApp)**:
  + Demonstrates integration of patterns:
    - **Singleton**: Retrieves configuration instance (Configuration.getInstance()).
    - **Factory Method**: Creates different users (UserFactory.createUser()).
    - **Observer**: Registers observers (UserObserver) and triggers notifications (NotificationService.notifyObservers()).
    - **Interpreter**: Interprets commands (OrderInterpreter.interpret()) and displays orders (OrderContext.getOrders()).
    - **Decorator**: Wraps products (GiftWrap, ExtendedWarranty) and displays enhanced product details.
    - **Facade**: Places orders (ECommerceFacade.placeOrder()) and manages subsystem interactions (ProductInventory, PaymentGateway, ShippingService).