**Behavioural Design Patterns**

**1. Observer Pattern**

// Stock.java

public class Stock {

private String name;

private double price;

private List<Investor> investors = new ArrayList<>();

public Stock(String name, double price) {

this.name = name;

this.price = price;

}

public void addObserver(Investor investor) {

investors.add(investor);

}

public void removeObserver(Investor investor) {

investors.remove(investor);

}

public void notifyObservers() {

for (Investor investor : investors) {

investor.update(this);

}

}

public void setPrice(double price) {

this.price = price;

notifyObservers();

}

public String getName() {

return name;

}

public double getPrice() {

return price;

}

}

// Investor.java

public interface Investor {

void update(Stock stock);

}

// ConcreteInvestor.java

public class ConcreteInvestor implements Investor {

private String name;

public ConcreteInvestor(String name) {

this.name = name;

}

@Override

public void update(Stock stock) {

System.out.println("Investor " + name + " notified. New price of " + stock.getName() + " is $" + stock.getPrice());

}

}

// Main.java

public class Main {

public static void main(String[] args) {

Stock appleStock = new Stock("Apple", 150.00);

Investor investor1 = new ConcreteInvestor("Investor 1");

Investor investor2 = new ConcreteInvestor("Investor 2");

appleStock.addObserver(investor1);

appleStock.addObserver(investor2);

appleStock.setPrice(155.00);

appleStock.setPrice(160.00);

}

}

**2. Strategy Pattern**

// PaymentStrategy.java

public interface PaymentStrategy {

void pay(int amount);

}

// CreditCardPayment.java

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

public CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

@Override

public void pay(int amount) {

System.out.println("Paid " + amount + " using Credit Card: " + cardNumber);

}

}

// PayPalPayment.java

public class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public void pay(int amount) {

System.out.println("Paid " + amount + " using PayPal: " + email);

}

}

// ShoppingCart.java

public class ShoppingCart {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void checkout(int amount) {

paymentStrategy.pay(amount);

}

}

// Main.java

public class Main {

public static void main(String[] args) {

ShoppingCart cart = new ShoppingCart();

cart.setPaymentStrategy(new CreditCardPayment("1234-5678-9876-5432"));

cart.checkout(100);

cart.setPaymentStrategy(new PayPalPayment("user@example.com"));

cart.checkout(200);

}

}

**Creational Design Patterns**

**1. Singleton Pattern**

// ConfigurationManager.java

public class ConfigurationManager {

private static ConfigurationManager instance;

private Properties properties;

private ConfigurationManager() {

properties = new Properties();

// Load properties from a file or any other source

}

public static ConfigurationManager getInstance() {

if (instance == null) {

synchronized (ConfigurationManager.class) {

if (instance == null) {

instance = new ConfigurationManager();

}

}

}

return instance;

}

public String getProperty(String key) {

return properties.getProperty(key);

}

}

// Main.java

public class Main {

public static void main(String[] args) {

ConfigurationManager configManager1 = ConfigurationManager.getInstance();

ConfigurationManager configManager2 = ConfigurationManager.getInstance();

System.out.println(configManager1 == configManager2); // Output: true

}

}

**2. Factory Pattern**

// Shape.java

public interface Shape {

void draw();

}

// Circle.java

public class Circle implements Shape {

@Override

public void draw() {

System.out.println("Drawing a Circle");

}

}

// Square.java

public class Square implements Shape {

@Override

public void draw() {

System.out.println("Drawing a Square");

}

}

// ShapeFactory.java

public class ShapeFactory {

public static Shape getShape(String shapeType) {

if (shapeType == null) {

return null;

}

if (shapeType.equalsIgnoreCase("CIRCLE")) {

return new Circle();

} else if (shapeType.equalsIgnoreCase("SQUARE")) {

return new Square();

}

return null;

}

}

// Main.java

public class Main {

public static void main(String[] args) {

Shape circle = ShapeFactory.getShape("CIRCLE");

circle.draw();

Shape square = ShapeFactory.getShape("SQUARE");

square.draw();

}

}

**Structural Design Patterns**

**1. Adapter Pattern**

// NewLogger.java

public interface NewLogger {

void logMessage(String message);

}

// LegacyLogger.java

public class LegacyLogger {

public void log(String msg) {

System.out.println("Legacy Logger: " + msg);

}

}

// LoggerAdapter.java

public class LoggerAdapter implements NewLogger {

private LegacyLogger legacyLogger;

public LoggerAdapter(LegacyLogger legacyLogger) {

this.legacyLogger = legacyLogger;

}

@Override

public void logMessage(String message) {

legacyLogger.log(message);

}

}

// Main.java

public class Main {

public static void main(String[] args) {

LegacyLogger legacyLogger = new LegacyLogger();

NewLogger logger = new LoggerAdapter(legacyLogger);

logger.logMessage("This is a log message.");

}

}

**2. Decorator Pattern**

// TextEditor.java

public interface TextEditor {

void type(String words);

}

// BasicTextEditor.java

public class BasicTextEditor implements TextEditor {

@Override

public void type(String words) {

System.out.println(words);

}

}

// TextEditorDecorator.java

public abstract class TextEditorDecorator implements TextEditor {

protected TextEditor textEditor;

public TextEditorDecorator(TextEditor textEditor) {

this.textEditor = textEditor;

}

public void type(String words) {

textEditor.type(words);

}

}

// SpellCheckDecorator.java

public class SpellCheckDecorator extends TextEditorDecorator {

public SpellCheckDecorator(TextEditor textEditor) {

super(textEditor);

}

@Override

public void type(String words) {

super.type(words);

System.out.println("Spell checking applied.");

}

}

// AutoCorrectDecorator.java

public class AutoCorrectDecorator extends TextEditorDecorator {

public AutoCorrectDecorator(TextEditor textEditor) {

super(textEditor);

}

@Override

public void type(String words) {

super.type(words);

System.out.println("Auto-correct applied.");

}

}

// Main.java

public class Main {

public static void main(String[] args) {

TextEditor editor = new BasicTextEditor();

TextEditor spellCheckEditor = new SpellCheckDecorator(editor);

TextEditor autoCorrectEditor = new AutoCorrectDecorator(spellCheckEditor);

autoCorrectEditor.type("Helo Wrd");

}

}