أنماط التصميم
Design Patterns

يوتيوب
Youtube
@Nullexia

تيليجرام
Telegram
@Nullexia

الفهرس
Index

أنماط التصميم الإبداعي
Creational Design Patterns

التعريف:	هى أنماط تصميم في البرمجة الكائنية يركز على توفير طرق مرنة وفعّالة لإنشاء الكائنات، بحيث يتم فصل عملية الإنشاء عن المنطق الرئيسي للتطبيق لتحسين القابلية للتوسع والصيانة.

نمط سينجلتون	Singleton Pattern
نمط طريقة المصنع	Factory Method Pattern
نمط المصنع المجرد	Abstract Factory Pattern
نمط المنشئ	Builder Pattern
نمط النموذج الأولي	Prototype Pattern

أنماط التصميم الهيكلي
Structural Design Patterns

التعريف:	هي أنماط تصميم تهدف إلى تسهيل تصميم وتنظيم العلاقات بين الكائنات أو المكونات لجعل النظام أكثر مرونة وقابلية للصيانة.

نمط المحول	Adapter Pattern
نمط الجسر	Bridge Pattern
نمط المركب	Composite Pattern
نمط المُزيّن	Decorator Pattern
نمط الواجهة	Facade Pattern
نمط الوكيل	Proxy Pattern
نمط وزن الذبابة	Flyweight Pattern

أنماط التصميم السلوكي
Behavioral Design Patterns

التعريف:	هي أنماط تصميم تركز على كيفية تفاعل الكائنات مع بعضها البعض وتحديد المسؤوليات بينها لتحسين مرونة وكفاءة النظام.

نمط المراقب	Observer Pattern
نمط الاستراتيجية	Strategy Pattern
نمط الحالة	State Pattern
نمط الأمر	Command Pattern
نمط سلسلة المسؤولية	Chain of Responsibility Pattern
نمط القالب	Template Pattern
نمط المترجم	Interpreter Pattern
نمط الزائر	Visitor Pattern
نمط الوسيط	Mediator Pattern
نمط التذكار	Memento Pattern
نمط التكرار	Iterator Pattern

أنماط مضادة
Anti-Patterns

التعريف:	هى أنماط تصميم لإيجاد حلول شائعة لمشكلة متكررة لكنه يؤدي إلى نتائج سلبية أو غير فعالة على المدى الطويل.

Spaghetti Code Pattern	نمط الكود السباغيتي
Golden Hammer Pattern	نمط المطرقة الذهبية
Boat Anchor Pattern	نمط مرساة القارب
Dead Code Pattern	نمط الكود الميت
Mastermind Class and Mastermind Object Pattern	نمط فئة العقل المدبر وكائن العقل المدبر
Copy and Paste Programming Pattern	نمط البرمجة بالنسخ واللصق

أنماط التصميم الإبداعي
Creational Design Patterns

نمط سينجلتون
Singleton Pattern

التعريف:

هو نمط تصميم هدفه وجود كائن واحد فقط من الفئة المحددة طوال دورة حياة التطبيق مع توفير نقطة وصول عالمية له.

Singleton.java

```java
public class Singleton {
    private static volatile Singleton instance;
    private String data;

    private Singleton(String data) {
        this.data = data;
    }

    static Singleton getInstance(String data) {
        Singleton result = instance;
        if (result == null) {
            synchronized (Singleton.class) {
                result = instance;
                if (result == null) {
                    instance = result = new Singleton(data);
                }
            }
        }
        return result;
    }

    String getData() {
        return data;
    }

    void setData(String data) {
        this.data = data;
    }
}
```

Main.java

```java
public class Main {
    public static void main(final String[] PARAMETERS) {
        System.out.println(Singleton.getInstance("Java").getData());
        final Singleton SINGLETON = Singleton.getInstance("Kotlin");
        System.out.println(SINGLETON.getData());
        SINGLETON.setData("Kotlin");
        System.out.print(SINGLETON.getData());
    }
}
```

Output

```
Java
Java
Kotlin
```

نمط طريقة المصنع
Factory Method Pattern

التعريف: هو نمط تصميم يوفر واجهة لإنشاء الكائنات في فئة عليا مع السماح للفئات الفرعية بتحديد أنواع الكائنات التي تنشئها.

factories

Restaurant.java

```java
import products.Rice;

public abstract class Restaurant {
    public void orderRice() {
        System.out.println("Ordering Rice ... ");
        Rice rice = makeRice();
        rice.prepare();
    }
    protected abstract Rice makeRice();
}
```

BrownRiceRestaurant.java

```java
import products.BrownRice;
import products.Rice;

public class BrownRiceRestaurant extends Restaurant {
    @Override
    protected Rice makeRice() {
        System.out.println("Making Brown Rice ... ");
        return new BrownRice();
    }
}
```

WhiteRiceRestaurant.java

```java
import products.Rice;
import products.WhiteRice;

public class WhiteRiceRestaurant extends Restaurant {
    @Override
    protected Rice makeRice() {
        System.out.println("Making White Rice ... ");
        return new WhiteRice();
    }
}
```

products

Rice.java

```java
public interface Rice {
    void prepare();
}
```

BrownRice.java

```java
public class BrownRice implements Rice {
    @Override
    public void prepare() {
        System.out.println("Preparing Brown Rice ... ");
    }
}
```

WhiteRice.java

```java
public class WhiteRice implements Rice {
    @Override
    public void prepare() {
        System.out.println("Preparing White Rice ... ");
    }
}
```

Main.java

```java
import factories.BrownRiceRestaurant;
import factories.Restaurant;
import factories.WhiteRiceRestaurant;

public class Main {
    public static void main(final String[] PARAMETERS) {
        Restaurant brownRiceRestaurant = new BrownRiceRestaurant();
        brownRiceRestaurant.orderRice();

        System.out.println("=".repeat(30));

        Restaurant whiteRiceRestaurant = new WhiteRiceRestaurant();
        whiteRiceRestaurant.orderRice();
    }
}
```

Output

```
Ordering Rice ...
Making Brown Rice ...
Preparing Brown Rice ...
==============================
Ordering Rice ...
Making White Rice ...
Preparing White Rice ...
```

نمط المصنع المجرد
Abstract Factory Pattern

التعريف:

هو نمط تصميم يُستخدم لإنشاء عائلات من الكائنات المرتبطة أو المتوافقة دون الحاجة إلى تحديد الفئات الفعلية لهذه الكائنات.

factories

Restaurant.java

```java
import products.burger.Burger;
import products.pizza.Pizza;

public abstract class Restaurant {
    public abstract Burger makeBurger();
    public abstract Pizza makePizza();
}
```

OrientalRestaurant.java

```java
import products.burger.Burger;
import products.burger.OrientalBurger;
import products.pizza.OrientalPizza;
import products.pizza.Pizza;

public class OrientalRestaurant extends Restaurant {
    @Override
    public Burger makeBurger() {
        return new OrientalBurger();
    }
    @Override
    public Pizza makePizza() {
        return new OrientalPizza();
    }
}
```

ClassicRestaurant.java

```java
import products.burger.Burger;
import products.burger.ClassicBurger;
import products.pizza.ClassicPizza;
import products.pizza.Pizza;

public class ClassicRestaurant extends Restaurant {
    @Override
    public Burger makeBurger() {
        return new ClassicBurger();
    }
    @Override
    public Pizza makePizza() {
        return new ClassicPizza();
    }
}
```

products

burger

Burger.java

```java
public interface Burger {
    void prepare();
}
```

OrientalBurger.java

```java
public class OrientalBurger implements Burger {
    @Override
    public void prepare() {
        System.out.println("Preparing Oriental Burger ... ");
    }
}
```

ClassicBurger.java

```java
public class ClassicBurger implements Burger {
    @Override
    public void prepare() {
        System.out.println("Preparing Classic Burger ... ");
    }
}
```

pizza

Pizza.java

```java
public interface Pizza {
    void bake();
}
```

OrientalPizza.java

```java
public class OrientalPizza implements Pizza {
    @Override
    public void bake() {
        System.out.println("Baking Oriental Pizza ... ");
    }
}
```

ClassicPizza.java

```java
public class ClassicPizza implements Pizza {
    @Override
    public void bake() {
        System.out.println("Baking Classic Pizza ... ");
    }
}
```

Main.java

```java
import factories.ClassicRestaurant;
import factories.OrientalRestaurant;
import factories.Restaurant;
import products.burger.Burger;
import products.pizza.Pizza;

public class Main {
    public static void main(final String[] PARAMETERS) {
        Restaurant orientalRestaurant = new OrientalRestaurant();
        Pizza orientalPizza = orientalRestaurant.makePizza();
        Burger orientalBurger = orientalRestaurant.makeBurger();
        orientalPizza.bake();
        orientalBurger.prepare();

        System.out.println("=".repeat(30));

        Restaurant classicRestaurant = new ClassicRestaurant();
        Pizza classicPizza = classicRestaurant.makePizza();
        Burger classicBurger = classicRestaurant.makeBurger();
        classicPizza.bake();
        classicBurger.prepare();
    }
}
```

Output

```
Baking Oriental Pizza ...
Preparing Oriental Burger ...
==============================
Baking Classic Pizza ...
Preparing Classic Burger ...
```

نمط المنشئ
Builder Pattern

التعريف: هو نمط تصميم يُستخدم لإنشاء كائنات معقدة خطوة بخطوة بطريقة مرنة وقابلة للتخصيص دون الحاجة إلى استخدام مُنشئ (constructor) كبير أو معقّد.

model

car

Car.java

```java
public class Car {
    private final CarData CAR_DATA = new CarData();

    Car(
            final int ID,
            final String BRAND,
            final String MODEL,
            final String COLOR,
            final int NUMBER_OF_DOORS,
            final String GLASS_TYPE,
            final double WEIGHT,
            final double HEIGHT
    ) {
        CAR_DATA.id = ID;
        CAR_DATA.brand = BRAND;
        CAR_DATA.model = MODEL;
        CAR_DATA.color = COLOR;
        CAR_DATA.numberOfDoors = NUMBER_OF_DOORS;
        CAR_DATA.glassType = GLASS_TYPE;
        CAR_DATA.weight = WEIGHT;
        CAR_DATA.height = HEIGHT;
    }

    public void printCarInformation() {
        System.out.printf("""
                          ID: %d
                          Brand: %s
                          Model: %s
                          Color: %s
                          Number of Doors: %d
                          Glass Type: %s
                          Weight: %f
                          Height: %f
                          """,
                  CAR_DATA.id,
                  CAR_DATA.brand,
                  CAR_DATA.model,
                  CAR_DATA.color,
                  CAR_DATA.numberOfDoors,
                  CAR_DATA.glassType,
                  CAR_DATA.weight,
                  CAR_DATA.height
          );
    }
}
```

CarData.java

```java
public class CarData {
    int id;
    String brand;
    String model;
    String color;
    int numberOfDoors;
    String glassType;
    double weight;
    double height;
}
```

Director.java

```java
public class Director {
    public void buildSeat(CarBuilder builder) {
        builder.brand("Seat");
    }

    public void buildToyota(CarBuilder builder) {
        builder.brand("Toyota");
    }
}
```

Builder.java

```java
public interface Builder {
    Builder id(final int ID);

    Builder brand(final String BRAND);

    Builder model(final String MODEL);

    Builder color(final String COLOR);

    Builder numberOfDoors(final int NUMBER_OF_DOORS);

    Builder glassType(final String GLASS_TYPE);

    Builder weight(final double WEIGHT);

    Builder height(final double HEIGHT);
}
```

CarBuilder.java

```java
public class CarBuilder implements Builder {
    private final CarData CAR_DATA = new CarData();

    @Override
    public CarBuilder id(final int ID) {
        CAR_DATA.id = ID;
        return this;
    }

    @Override
    public CarBuilder brand(final String BRAND) {
        CAR_DATA.brand = BRAND;
        return this;
    }

    @Override
    public CarBuilder model(final String MODEL) {
        CAR_DATA.model = MODEL;
        return this;
    }

    @Override
    public CarBuilder color(final String COLOR) {
        CAR_DATA.color = COLOR;
        return this;
    }

    @Override
    public CarBuilder numberOfDoors(final int NUMBER_OF_DOORS) {
        CAR_DATA.numberOfDoors = NUMBER_OF_DOORS;
        return this;
    }

    @Override
    public CarBuilder glassType(final String GLASS_TYPE) {
        CAR_DATA.glassType = GLASS_TYPE;
        return this;
    }

    @Override
    public CarBuilder weight(final double WEIGHT) {
        CAR_DATA.weight = WEIGHT;
        return this;
    }

    @Override
    public CarBuilder height(final double HEIGHT) {
        CAR_DATA.height = HEIGHT;
        return this;
    }

    public Car build() {
        return new Car(
                CAR_DATA.id,
                CAR_DATA.brand,
                CAR_DATA.model,
                CAR_DATA.color,
                CAR_DATA.numberOfDoors,
                CAR_DATA.glassType,
                CAR_DATA.weight,
                CAR_DATA.height
        );
    }
}
```

Main.java

```java
import model.car.Car;
import model.car.CarBuilder;
import model.car.Director;

public class Main {
    public static void main(final String[] PARAMETERS) {
        CarBuilder carBuilder = new CarBuilder();
        carBuilder.id(101)
                .model("Corolla")
                .color("Red")
                .numberOfDoors(4)
                .glassType("Tempered")
                .weight(1300.5F)
                .height(1.45F);
        Director director = new Director();
        director.buildToyota(carBuilder);
        Car car = carBuilder.build();
        car.printCarInformation();
    }
}
```

Output

```
ID: 101
Brand: Toyota
Model: Corolla
Color: Red
Number of Doors: 4
Glass Type: Tempered
Weight: 1300.500000
Height: 1.450000
```

نمط النموذج الأولي
Prototype Pattern

التعريف:

هو نمط تصميم يسمح بإنشاء كائنات جديدة عن طريق نسخ كائنات موجودة (نسخ عميقة أو سطحية)، مما يوفر طريقة مرنة وسريعة لإنشاء الكائنات دون الحاجة لإعادة تهيئتها أو بنائها من الصفر.

Shape.java

```java
abstract class Shape {
    String color;

    public Shape() {
    }

    public Shape(Shape source) {
        if (source ≠ null) {
            this.color = source.color;
        }
    }

    public abstract Shape clone();

    public void printInformation() {
        System.out.printf("""
                          Color: %s
                          """,
                color
        );
    }
}
```

Circle.java

```java
class Circle extends Shape {
    double radius;

    public Circle() {
    }

    private Circle(Circle source) {
        super(source);
        if (source ≠ null) {
            this.radius = source.radius;
        }
    }

    @Override
    public Circle clone() {
        return new Circle(this);
    }

    @Override
    public void printInformation() {
        super.printInformation();
        System.out.printf("""
                          Radius: %f
                          """,
                radius
        );
    }
}
```

Rectangle.java

```java
class Rectangle extends Shape {
    double width;
    double height;

    public Rectangle() {
    }

    private Rectangle(Rectangle source) {
        super(source);
        if (source ≠ null) {
            this.width = source.width;
            this.height = source.height;
        }
    }

    @Override
    public Rectangle clone() {
        return new Rectangle(this);
    }

    @Override
    public void printInformation() {
        super.printInformation();
        System.out.printf("""
                          Width: %f
                          Height: %f
                          """,
                width,
                height
        );
    }
}
```

Main.java

```java
class Main {
    public static void main(final String[] PARAMETERS) {
        Circle circle = new Circle();
        circle.radius = 20;
        circle.color = "Red";
        Circle anotherCircle = circle.clone();

        System.out.println("Original Circle:");
        circle.printInformation();

        System.out.println("=".repeat(30));

        System.out.println("Copy of Circle:");
        anotherCircle.printInformation();
    }
}
```

Output

```
Original Circle:
Color: Red
Radius: 20.000000
==============================
Copy of Circle:
Color: Red
Radius: 20.000000
```

أنماط التصميم الهيكلي
Structural Design Patterns

نمط المحول
Adapter Pattern

التعريف:

هو نمط تصميم هيكلي يُستخدم لتوفير واجهة متوافقة بين كائنين غير متوافقين عن طريق تغليف أحدهما داخل كائن وسيط (Adapter).

Printer.java

```java
public class Printer {
    void printDocument() {
        System.out.println("Printer is printing a document.");
    }
}
```

PrinterServices.java

```java
public interface PrinterServices {
    void print();
}
```

PrinterAdapter.java

```java
public class PrinterAdapter implements PrinterServices {
    private final Printer Printer;

    public PrinterAdapter() {
        this.Printer = new Printer();
    }

    @Override
    public void print() {
        Printer.printDocument();
    }
}
```

Main.java

```java
public class Main {
    public static void main(final String[] PARAMETERS) {
        final PrinterAdapter PRINTER_ADAPTER = new PrinterAdapter();
        PRINTER_ADAPTER.print();
    }
}
```

Output

```
Printer is printing a document.
```

نمط الجسر
Bridge Pattern

التعريف:

هو نمط تصميم هيكلي يفصل بين التجريد (Abstraction) والتنفيذ (Implementation) لجعل كل منهما قابل للتطوير والتغيير بشكل مستقل.

abstractions

Restaurant.java

```java
import implementations.Pizza;

public abstract class Restaurant {
    protected Pizza pizza;

    protected Restaurant(Pizza pizza) {
        this.pizza = pizza;
    }

    abstract void addSauce();

    abstract void addToppings();

    abstract void makeCrust();

    public void deliver() {
        makeCrust();
        addSauce();
        addToppings();
        pizza.assemble();
        pizza.qualityCheck();
        System.out.println("Order in Progress!");
    }
}
```

EgyptianRestaurant.java

```java
import implementations.Pizza;

public class EgyptianRestaurant extends Restaurant {
    public EgyptianRestaurant(Pizza pizza) {
        super(pizza);
    }

    @Override
    public void addToppings() {
        pizza.setToppings("Cheese and Black Olives");
    }

    @Override
    public void addSauce() {
        pizza.setSauce("Tomato Sauce with Garlic");
    }

    @Override
    public void makeCrust() {
        pizza.setCrust("Thick and Crispy");
    }
}
```

SyrianRestaurant.java

```java
import implementations.Pizza;

public class SyrianRestaurant extends Restaurant {
    public SyrianRestaurant(Pizza pizza) {
        super(pizza);
    }

    @Override
    public void addToppings() {
        pizza.setToppings("Lamb and Pine Nuts");
    }

    @Override
    public void addSauce() {
        pizza.setSauce("Yogurt and Tahini Sauce");
    }

    @Override
    public void makeCrust() {
        pizza.setCrust("Soft and Thin");
    }
}
```

implementaions

Pizza.java

```java
public abstract class Pizza {
    protected String sauce;
    protected String toppings;
    protected String crust;

    public String getSauce() {
        return sauce;
    }

    public void setSauce(String sauce) {
        this.sauce = sauce;
    }

    public String getToppings() {
        return toppings;
    }

    public void setToppings(String toppings) {
        this.toppings = toppings;
    }

    public String getCrust() {
        return crust;
    }

    public void setCrust(String crust) {
        this.crust = crust;
    }

    public abstract void assemble();

    public abstract void qualityCheck();
}
```

PepperoniPizza.java

```java
public class PepperoniPizza extends Pizza {
    @Override
    public void assemble() {
        System.out.println("Adding Sauce: " + sauce);
        System.out.println("Adding Toppings: " + toppings);
        System.out.println("Adding Pepperoni");
    }

    @Override
    public void qualityCheck() {
        System.out.println("Crust is: " + crust);
    }
}
```

VeggiePizza.java

```java
public class VeggiePizza extends Pizza {
    @Override
    public void assemble() {
        System.out.println("Adding Sauce: " + sauce);
        System.out.println("Adding Toppings: " + toppings);
        System.out.println("Adding Cheese");
    }

    @Override
    public void qualityCheck() {
        System.out.println("Crust is: " + crust);
    }
}
```

Main.java

```java
import abstractions.AmericanRestaurant;
import abstractions.ItalianRestaurant;
import abstractions.Restaurant;
import implementations.PepperoniPizza;
import implementations.VeggiePizza;

public class Main {
    public static void main(final String[] PARAMETERS) {
        Restaurant americanRestaurant = new AmericanRestaurant(new PepperoniPizza());
        americanRestaurant.deliver();

        System.out.println("=".repeat(30));

        Restaurant italianRestaurant = new ItalianRestaurant(new VeggiePizza());
        italianRestaurant.deliver();
    }
}
```

Output

```
Adding Sauce: Tomato Sauce with Garlic
Adding Toppings: Cheese and Black Olives
Adding Pepperoni
Crust is: Thick and Crispy
Order in Progress!
==============================
Adding Sauce: Yogurt and Tahini Sauce
Adding Toppings: Lamb and Pine Nuts
Adding Cheese
Crust is: Soft and Thin
Order in Progress!
```

نمط المركب
Composite Pattern

التعريف:

هو نمط تصميم هيكلي يتيح التعامل مع مجموعة من الكائنات ككائن واحد، حيث يمكن للأشجار الهيكلية (tree structures) أن تمثل التسلسلات الهرمية للأجزاء والكُلّ بطريقة متجانسة.

FileSystemComponents.java

```java
interface FileSystemComponents {
    int COUNT_OF_SPACES = 4;

    void showDetails();
}
```

File.java

```java
class File implements FileSystemComponents {
    private String name;

    public File(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public void showDetails() {
        System.out.println(" ".repeat(COUNT_OF_SPACES) + "File: " + name);
    }
}
```

Folder.java

```java
import java.util.ArrayList;
import java.util.List;

class Folder implements FileSystemComponents {
    private String name;
    private final List<FileSystemComponents> components = new ArrayList<>();

    public Folder(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void addComponent(FileSystemComponents... component) {
        components.addAll(List.of(component));
    }

    @Override
    public void showDetails() {
        System.out.println("Folder: " + name);
        for (FileSystemComponents component : components) {
            System.out.print(" ".repeat(COUNT_OF_SPACES));
            component.showDetails();
        }
    }
}
```

Main.java

```java
public class Main {
    public static void main(final String[] PARAMETERS) {
        FileSystemComponents[] documents = new FileSystemComponents[]{
                new File("Document1.txt"),
                new File("Document2.txt"),
                new File("Document3.txt")
        };
        FileSystemComponents[] images = new FileSystemComponents[]{
                new File("Image1.mp3"),
                new File("Image2.mp3"),
                new File("Image3.mp3")
        };
        FileSystemComponents[] videos = new FileSystemComponents[]{
                new File("Video1.mp4"),
                new File("Video2.mp4"),
                new File("Video3.mp4")
        };

        Folder folderOfDocuments = new Folder("My Documents");
        folderOfDocuments.addComponent(documents);

        Folder folderOfImages = new Folder("My Images");
        folderOfImages.addComponent(images);

        Folder folderOfVidoes = new Folder("My Videos");
        folderOfVidoes.addComponent(videos);

        Folder mainFolder = new Folder("Main Folder");
        mainFolder.addComponent(
                folderOfDocuments,
                folderOfImages,
                folderOfVidoes
        );

        mainFolder.showDetails();
    }
}
```

Output

```
Folder: Main Folder
    Folder: My Documents
        File: Document1.txt
        File: Document2.txt
        File: Document3.txt
    Folder: My Images
        File: Image1.mp3
        File: Image2.mp3
        File: Image3.mp3
    Folder: My Videos
        File: Video1.mp4
        File: Video2.mp4
        File: Video3.mp4
```

نمط المُزيّن
Decorator Pattern

التعريف:

هو نمط تصميم يسمح بإضافة ميزات أو وظائف جديدة للكائنات ديناميكيًا دون تعديل بنيتها الأساسية.

CoffeeService.java

```java
interface CoffeeService {
    String getDescription();

    double getCost();
}
```

PlainCoffee.java

```java
class PlainCoffee implements CoffeeService {
    @Override
    public String getDescription() {
        return "Plain Coffee";
    }

    @Override
    public double getCost() {
        return 2.0;
    }
}
```

CoffeeDecorator.java

```java
abstract class CoffeeDecorator implements CoffeeService {
    protected CoffeeService decoratedCoffee;

    public CoffeeDecorator(CoffeeService decoratedCoffee) {
        this.decoratedCoffee = decoratedCoffee;
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }
}
```

SugarDecorator.java

```java
class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(CoffeeService decoratedCoffee) {
        super(decoratedCoffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Sugar";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.2;
    }
}
```

MilkDecorator.java

```java
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(CoffeeService decoratedCoffee) {
        super(decoratedCoffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Milk";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.5;
    }
}
```

Main.java

```java
public class Main {
    public static void main(final String[] PARAMETERS) {
        CoffeeService coffee = new PlainCoffee();
        System.out.println("Description: " + coffee.getDescription());
        System.out.println("Cost: £" + coffee.getCost());

        CoffeeService sugarCoffee = new SugarDecorator(new PlainCoffee());
        System.out.println("\nDescription: " + sugarCoffee.getDescription());
        System.out.println("Cost: £" + sugarCoffee.getCost());

        CoffeeService milkCoffee = new MilkDecorator(new PlainCoffee());
        System.out.println("\nDescription: " + milkCoffee.getDescription());
        System.out.println("Cost: £" + milkCoffee.getCost());

        CoffeeService sugarMilkCoffee = new SugarDecorator(new MilkDecorator(new PlainCoffee()));
        System.out.println("\nDescription: " + sugarMilkCoffee.getDescription());
        System.out.println("Cost: £" + sugarMilkCoffee.getCost());
    }
}
```

Output

```
Description: Plain Coffee
Cost: £2.0

Description: Plain Coffee, Sugar
Cost: £2.2

Description: Plain Coffee, Milk
Cost: £2.5

Description: Plain Coffee, Milk, Sugar
Cost: £2.7
```

نمط الواجهة
Facade Pattern

التعريف:

هو نمط تصميم يوفر واجهة مبسطة وعالية المستوى لتسهيل التعامل مع نظام معقد أو مجموعة من الأنظمة الفرعية.

DeviceControlService.java

```java
interface DeviceControlService {
    void turnOn();

    void turnOff();
}
```

Light.java

```java
class Light implements DeviceControlService {
    @Override
    public void turnOn() {
        System.out.println("Lights are ON");
    }

    @Override
    public void turnOff() {
        System.out.println("Lights are OFF");
    }
}
```

AirConditioner.java

```java
class AirConditioner implements DeviceControlService {
    @Override
    public void turnOn() {
        System.out.println("Air Conditioner is ON");
    }

    @Override
    public void turnOff() {
        System.out.println("Air Conditioner is OFF");
    }
}
```

HomeAutomationSystem.java

```java
class HomeAutomationSystem {
    private final Light LIGHT;
    private final AirConditioner AIR_CONDITIONER;

    public HomeAutomationSystem() {
        this.LIGHT = new Light();
        this.AIR_CONDITIONER = new AirConditioner();
    }

    public void startSleepMode() {
        System.out.println("Starting Sleep Mode ... ");
        LIGHT.turnOff();
        AIR_CONDITIONER.turnOn();
    }

    public void stopSleepMode() {
        System.out.println("Stopping Sleep Mode ... ");
        LIGHT.turnOn();
        AIR_CONDITIONER.turnOff();
    }
}
```

Main.java

```java
public class Main {
    public static void main(final String[] PARAMETERS) {
        HomeAutomationSystem homeSystem = new HomeAutomationSystem();
        homeSystem.startSleepMode();
        System.out.println("=".repeat(30));
        homeSystem.stopSleepMode();
    }
}
```

Output

```
Starting Sleep Mode ...
Lights are OFF
Air Conditioner is ON
==============================
Stopping Sleep Mode ...
Lights are ON
Air Conditioner is OFF
```

نمط الوكيل
Proxy Pattern

التعريف:

هي نمط تصميم هيكلي يستخدم كوسيط للتحكم في الوصول إلى كائن آخر، حيث يقوم بالتحكم في العمليات التي تتم عليه أو توجيهها.

BankAccountService.java

```java
interface BankAccountService {
    void deposit(double amount);

    void withdraw(double amount);

    void displayBalance();
}
```

BankAccount.java

```java
class BankAccount implements BankAccountService {
    private double balance;

    public BankAccount(double balance) {
        this.balance = balance;
    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }

    @Override
    public void deposit(double amount) {
        if (amount > 0) {
            setBalance(getBalance() + amount);
            System.out.print("Deposited: " + amount + "£ , ");
            displayBalance();
        } else
            System.out.println("Enter amonut in the positive!");
    }

    @Override
    public void withdraw(double amount) {
        if (amount > 0)
            if (amount <= balance) {
                setBalance(balance - amount);
                System.out.print("Withdrawn: " + amount + "£ , ");
                displayBalance();
            } else
                System.out.println("Insufficient balance!");
        else
            System.out.println("Enter amonut in the positive!");
    }

    @Override
    public void displayBalance() {
        System.out.println("Current Balance: " + getBalance() + "£");
    }
}
```

BankAccountProxy.java

```java
class BankAccountProxy implements BankAccountService {
    private final BankAccount REAL_ACCOUNT;
    private final boolean IS_AUTHORIZE;
    private final String MESSAGE_OF_ACCESS_DENIED_UNAUTHORIZED_USER = "Access denied: Unauthorized user!";

    public BankAccountProxy(BankAccount realAccount, boolean authorizedUser) {
        this.REAL_ACCOUNT = realAccount;
        this.IS_AUTHORIZE = authorizedUser;
    }

    @Override
    public void deposit(double amount) {
        if (IS_AUTHORIZE)
            REAL_ACCOUNT.deposit(amount);
        else
            System.out.println(MESSAGE_OF_ACCESS_DENIED_UNAUTHORIZED_USER);
    }

    @Override
    public void withdraw(double amount) {
        if (IS_AUTHORIZE)
            REAL_ACCOUNT.withdraw(amount);
        else
            System.out.println(MESSAGE_OF_ACCESS_DENIED_UNAUTHORIZED_USER);
    }

    @Override
    public void displayBalance() {
        if (IS_AUTHORIZE)
            REAL_ACCOUNT.displayBalance();
        else
            System.out.println(MESSAGE_OF_ACCESS_DENIED_UNAUTHORIZED_USER);
    }
}
```

Main.java

```java
public class Main {
    public static void main(final String[] PARAMETERS) {
        BankAccount realAccount = new BankAccount(5000);

        BankAccountService proxy = new BankAccountProxy(
                realAccount,
                true
        );
        proxy.displayBalance();
        proxy.withdraw(1000);

        proxy = new BankAccountProxy(
                realAccount,
                false
        );
        proxy.deposit(500);
    }
}
```

Output

```
Current Balance: 5000.0£
Withdrawn: 1000.0£ , Current Balance: 4000.0£
Access denied: Unauthorized user!
```

نمط وزن الذبابة
Flyweight Pattern

التعريف:

هو نمط تصميم هيكلي يهدف إلى تقليل استهلاك الذاكرة من خلال مشاركة الكائنات المتشابهة بدلاً من إنشاء نسخ جديدة لكل كائن.

Rectangle.java

```java
class Rectangle {
    private final String color;

    public Rectangle(String color) {
        this.color = color;
    }

    public void draw(int width, int height) {
        System.out.println("Drawing Rectangle [Color: " + color + ", x: " + width + ", y: " + height + "]");
    }
}
```

RectangleFactory.java

```java
import java.util.HashMap;

class RectangleFactory {
    private static final HashMap<String, Rectangle> RECTANGLES = new HashMap<>();

    public static Rectangle getRectangle(String color) {
        RECTANGLES.putIfAbsent(color, new Rectangle(color));
        return RECTANGLES.get(color);
    }
}
```

Main.java

```java
public class Main {
    public static void main(final String[] PARAMETERS) {
        Rectangle redRectangle = RectangleFactory.getRectangle("Red");
        redRectangle.draw(10, 20);

        Rectangle blueRectangle = RectangleFactory.getRectangle("Blue");
        blueRectangle.draw(25, 35);

        Rectangle anotherRedRectangle = RectangleFactory.getRectangle("Red");
        anotherRedRectangle.draw(50, 60);
    }
}
```

Output

```
Drawing Rectangle [Color: Red, x: 10, y: 20]
Drawing Rectangle [Color: Blue, x: 25, y: 35]
Drawing Rectangle [Color: Red, x: 50, y: 60]
```

أنماط التصميم السلوكي
Behavioral Design Patterns

نمط المراقب
Observer Pattern

التعريف:

هو نمط تصميم يسمح لك باشعار مجموعة من الكائنات تلقائيًا عند حدوث تغيير في حالة كائن آخر.

ObserverService.java

```java
interface ObserverService {
    void update(float temperature);
}
```

WeatherDisplay.java

```java
class WeatherDisplay implements ObserverService {
    private String displayName;

    WeatherDisplay(String name) {
        this.displayName = name;
    }

    public String getDisplayName() {
        return displayName;
    }

    public void setDisplayName(String displayName) {
        this.displayName = displayName;
    }

    @Override
    public void update(float temperature) {
        System.out.println(displayName + " updated: Temperature is now " + temperature + "°C");
    }
}
```

WeatherStation.java

```java
import java.util.ArrayList;
import java.util.List;

class WeatherStation {
    private final List<ObserverService> OBSERVERS = new ArrayList<>();
    private float temperature;

    void addObserver(ObserverService observer) {
        OBSERVERS.add(observer);
    }

    void removeObserver(ObserverService observer) {
        OBSERVERS.remove(observer);
    }

    void setTemperature(float newTemperature) {
        this.temperature = newTemperature;
        System.out.println("Weather Station: New temperature recorded: " + temperature + "°C");
        notifyObservers();
    }

    private void notifyObservers() {
        for (ObserverService observer : OBSERVERS)
            observer.update(temperature);
    }
}
```

Main.java

```java
public class Main {
    public static void main(final String[] PARAMETERS) {
        WeatherStation station = new WeatherStation();

        ObserverService phoneDisplay = new WeatherDisplay("Phone Display");
        ObserverService lcdScreen = new WeatherDisplay("LCD Screen");

        station.addObserver(phoneDisplay);
        station.addObserver(lcdScreen);

        station.setTemperature(25.5f);
        System.out.println("=".repeat(50));
        station.setTemperature(30.0f);
    }
}
```

Output

```
Weather Station: New temperature recorded: 25.5°C
Phone Display updated: Temperature is now 25.5°C
LCD Screen updated: Temperature is now 25.5°C
==================================================
Weather Station: New temperature recorded: 30.0°C
Phone Display updated: Temperature is now 30.0°C
LCD Screen updated: Temperature is now 30.0°C
```

نمط الاستراتيجية
Strategy Pattern

التعريف:

هو نمط تصميم يسمح بتحديد مجموعة من الخوارزميات القابلة للتبديل ديناميكيًا دون تغيير كود العميل.

---

strategy

sort

SortingStrategy.java

```java
interface SortingStrategy {
    void sort(int[] array);
}
```

BubbleSortStrategy.java

```java
class BubbleSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] array) {
        System.out.println("Sorting using Bubble Sort");
    }
}
```

MergeSortStrategy.java

```java
class MergeSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] array) {
        System.out.println("Sorting using Merge Sort");
    }
}
```

QuickSortStrategy.java

```java
class QuickSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] array) {
        System.out.println("Sorting using Quick Sort");
    }
}
```

SortingContext.java

```java
class SortingContext {
    private SortingStrategy sortingStrategy;

    public SortingContext(SortingStrategy sortingStrategy) {
        this.sortingStrategy = sortingStrategy;
    }

    public void setSortingStrategy(SortingStrategy sortingStrategy) {
        this.sortingStrategy = sortingStrategy;
    }

    public void performSort(int[] array) {
        sortingStrategy.sort(array);
    }
}
```

Main.java

```java
public class Main {
    public static void main(final String[] PARAMETERS) {
        final int[] NUMBERS = {5, 2, 9, 1, 5};

        SortingContext sortingContext = new SortingContext(new BubbleSortStrategy());
        sortingContext.performSort(NUMBERS);

        sortingContext.setSortingStrategy(new MergeSortStrategy());
        sortingContext.performSort(NUMBERS);

        sortingContext.setSortingStrategy(new QuickSortStrategy());
        sortingContext.performSort(NUMBERS);
    }
}
```

Output

```
Sorting using Bubble Sort
Sorting using Merge Sort
Sorting using Quick Sort
```

نمط الحالة
State Pattern

التعريف:

هو نمط تصميم يسمح لك بتغيير سلوك الكائن ديناميكيًا وفقًا لحالته الداخلية دون استخدام شروط متفرعة.

---

state

State.java

```java
import Fan;

public interface State {
    void turnUp(Fan fan);
    void turnDown(Fan fan);
}
```

OffState.java

```java
import Fan;

public class OffState implements State {
    public void turnUp(Fan fan) {
        fan.setState(new LowState());
        System.out.println("Fan is on Low Speed");
    }
    public void turnDown(Fan fan) {
        System.out.println("Fan is already Off");
    }
}
```

LowState.java

```java
import Fan;

class LowState implements State {
    public void turnUp(Fan fan) {
        fan.setState(new HighState());
        System.out.println("Fan is on High Speed");
    }
    public void turnDown(Fan fan) {
        fan.setState(new OffState());
        System.out.println("Fan is turned Off");
    }
}
```

HighState.java

```java
import Fan;

class HighState implements State {
    public void turnUp(Fan fan) {
        System.out.println("Fan is already on High Speed");
    }
    public void turnDown(Fan fan) {
        fan.setState(new LowState());
        System.out.println("Fan is on Low Speed");
    }
}
```

Fan.java

```java
import state.OffState;
import state.State;

public class Fan {
    private State state = new OffState();

    public void setState(State state) {
        this.state = state;
    }

    public void turnUp() {
        state.turnUp(this);
    }

    public void turnDown() {
        state.turnDown(this);
    }
}
```

Main.java

```java
public class Main {
    public static void main(final String[] PARAMETERS) {
        Fan fan = new Fan();
        fan.turnUp();
        fan.turnUp();
        fan.turnDown();
        fan.turnDown();
    }
}
```

Output

```
Fan is on Low Speed
Fan is on High Speed
Fan is on Low Speed
Fan is turned Off
```

نمط الحالة
Command Pattern

التعريف:

هو نمط تصميم يفصل بين الجهة التي تطلب تنفيذ الأمر والجهة التي تنفذه، مما يسهل التراجع وجدولة الأوامر.

---

components

RemoteControl.java

```java
import commands.Command;

public class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}
```

Light.java

```java
public class Light {
    public void turnOn() {
        System.out.println("Light is ON");
    }

    public void turnOff() {
        System.out.println("Light is OFF");
    }
}
```

commands

Command.java

```java
public interface Command {
    void execute();
}
```

TurnOffCommand.java

```java
import components.Light;

public class TurnOffCommand implements Command {
    private Light light;

    public TurnOffCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOff();
    }
}
```

TurnOnCommand.java

```java
import components.Light;

public class TurnOnCommand implements Command {
    private Light light;

    public TurnOnCommand(Light light) {
        this.light = light;
    }

    public void execute() {
        light.turnOn();
    }
}
```

Main.java

```java
import commands.Command;
import commands.TurnOffCommand;
import commands.TurnOnCommand;
import components.Light;
import components.RemoteControl;

public class Main {
    public static void main(final String[] PARAMETERS) {
        Light light = new Light();
        Command turnOn = new TurnOnCommand(light);
        Command turnOff = new TurnOffCommand(light);

        RemoteControl remoteControl = new RemoteControl();

        remoteControl.setCommand(turnOn);
        remoteControl.pressButton();

        remoteControl.setCommand(turnOff);
        remoteControl.pressButton();
    }
}
```

Output

```
Light is ON
Light is OFF
```

نمط سلسلة المسؤولية
Chain of Responsibility Pattern

التعريف:

هو نمط تصميم يسمح بتمرير الطلب عبر سلسلة من المعالجات حتى يتم التعامل معه دون أن يعرف المرسل المستقبل الفعلي للطلب.

Request.java

```java
public class Request {
    private final Priority PRIORITY;

    public Request(Priority priority) {
        this.PRIORITY = priority;
    }

    public Priority getPriority() {
        return PRIORITY;
    }
}
```

Priority.java

```java
public enum Priority {
    BASIC,
    INTERMEDIATE,
    CRITICAL,
    THING
}
```

handlers

SupportHandler.java

```java
import Request;

public interface SupportHandler {
    void handleRequest(Request request);

    void setNextHandler(SupportHandler nextHandler);
}
```

Level1SupportHandler.java

```java
import Priority;
import Request;

public class Level1SupportHandler implements SupportHandler {
    private SupportHandler nextHandler;

    public void setNextHandler(SupportHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    public void handleRequest(Request request) {
        if (request.getPriority() == Priority.BASIC) {
            System.out.println("Level 1 Support handled the request.");
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        }
    }
}
```

Level2SupportHandler.java

```java
import Priority;
import Request;

public class Level2SupportHandler implements SupportHandler {
    private SupportHandler nextHandler;

    public void setNextHandler(SupportHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    public void handleRequest(Request request) {
        if (request.getPriority() == Priority.INTERMEDIATE) {
            System.out.println("Level 2 Support handled the request.");
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        }
    }
}
```

Level3SupportHandler.java

```java
import Priority;
import Request;

public class Level3SupportHandler implements SupportHandler {
    public void setNextHandler(SupportHandler nextHandler) {
    }

    public void handleRequest(Request request) {
        if (request.getPriority() == Priority.CRITICAL) {
            System.out.println("Level 3 Support handled the request.");
        } else {
            System.out.println("Request cannot be handled.");
        }
    }
}
```

Main.java

```java
import handlers.Level1SupportHandler;
import handlers.Level2SupportHandler;
import handlers.Level3SupportHandler;
import handlers.SupportHandler;

public class Main {
    public static void main(final String[] PARAMETERS) {
        SupportHandler level1Handler = new Level1SupportHandler();
        SupportHandler level2Handler = new Level2SupportHandler();
        SupportHandler level3Handler = new Level3SupportHandler();

        level1Handler.setNextHandler(level2Handler);
        level2Handler.setNextHandler(level3Handler);

        Request request1 = new Request(Priority.BASIC);
        Request request2 = new Request(Priority.INTERMEDIATE);
        Request request3 = new Request(Priority.CRITICAL);
        Request request4 = new Request(Priority.THING);

        level1Handler.handleRequest(request1);
        level1Handler.handleRequest(request2);
        level1Handler.handleRequest(request3);
        level1Handler.handleRequest(request4);
    }
}
```

Output

```
Level 1 Support handled the request.
Level 2 Support handled the request.
Level 3 Support handled the request.
Request cannot be handled.
```

نمط القالب
Template Pattern

التعريف:

هو نمط تصميم في البرمجة الكائنية يحدد الهيكل العام لخوارزمية في فئة أساسية (abstract class) ويترك تنفيذ بعض الخطوات للفئات الفرعية (subclasses) دون تغيير ترتيب الخطوات.

BeverageMaker.java

```java
abstract class BeverageMaker {
    public final void makeBeverage() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

CoffeeMaker.java

```java
class CoffeeMaker extends BeverageMaker {
    @Override
    void brew() {
        System.out.println("Dripping coffee through filter");
    }

    @Override
    void addCondiments() {
        System.out.println("Adding sugar and milk");
    }
}
```

TeaMaker.java

```java
class TeaMaker extends BeverageMaker {
    @Override
    void brew() {
        System.out.println("Steeping the tea");
    }

    @Override
    void addCondiments() {
        System.out.println("Adding lemon");
    }
}
```

Main.java

```java
public class Main {
    public static void main(final String[] PARAMETERS) {
        System.out.println("Making tea:");
        BeverageMaker teaMaker = new TeaMaker();
        teaMaker.makeBeverage();

        System.out.println("=".repeat(30));

        System.out.println("Making coffee:");
        BeverageMaker coffeeMaker = new CoffeeMaker();
        coffeeMaker.makeBeverage();
    }
}
```

Output

```
Making tea:
Boiling water
Steeping the tea
Pouring into cup
Adding lemon
==============================
Making coffee:
Boiling water
Dripping coffee through filter
Pouring into cup
Adding sugar and milk
```

نمط المترجم
Interpreter Pattern

التعريف:

هو نمط تصميم يُستخدم لتفسير الجمل في لغة معينة عبر تمثيل قواعدها النحوية ككائنات قابلة للتقييم.

Expression.java

```java
public interface Expression {
    int interpret();
}
```

Number.java

```java
class Number implements Expression {
    private final int value;

    public Number(int value) {
        this.value = value;
    }

    public int interpret() {
        return value;
    }
}
```

operators

Add.java

```java
import Expression;

public class Add implements Expression {
    private final Expression left, right;

    public Add(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    public int interpret() {
        return left.interpret() + right.interpret();
    }
}
```

Subtract.java

```java
import Expression;

public class Subtract implements Expression {
    private final Expression left, right;

    public Subtract(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    public int interpret() {
        return left.interpret() - right.interpret();
    }
}
```

Main.java

```java
import operators.Add;
import operators.Subtract;

public class Main {
    public static void main(final String[] PARAMETERS) {
        Expression expression = new Add(
                new Number(5),
                new Subtract(new Number(10),
                        new Number(3)
                )
        );
        System.out.print(expression.interpret());
    }
}
```

Output

```
12
```

نمط الزائر
Visitor Pattern

التعريف:

هو نمط تصميم يسمح بإضافة عمليات جديدة على كائنات من دون تعديل هياكلها، عبر فصل الخوارزميات عن الكائنات التي تعمل عليها.

ElementVisitor.java

```java
public interface ElementVisitor {
    void visit(Pen pen);

    void visit(Pencil pencil);

    void visit(PencilSharpener pencilSharpener);
}
```

TotalPricesCalculator.java

```java
import models.ElementVisitor;
import models.Pen;
import models.Pencil;
import models.PencilSharpener;

public class TotalPricesCalculator implements ElementVisitor {
    private final double PEN_PRICE = 5,
            PENCIL_PRICE = 3,
            PENCIL_SHARPENER_PRICE = 2;
    private double totalPrices = 0;

    public double getPenPrice() {
        return PEN_PRICE;
    }

    public double getPencilPrice() {
        return PENCIL_PRICE;
    }

    public double getPencilSharpenerPrice() {
        return PENCIL_SHARPENER_PRICE;
    }

    public double getTotalPrices() {
        return totalPrices;
    }

    @Override
    public void visit(Pen pen) {
        totalPrices += PEN_PRICE;
    }

    @Override
    public void visit(Pencil pencil) {
        totalPrices += PENCIL_PRICE;
    }

    @Override
    public void visit(PencilSharpener pencilSharpener) {
        totalPrices += PENCIL_SHARPENER_PRICE;
    }
}
```

models

Element.java

```java
interface Element {
    void accept(ElementVisitor elementVisitor);
}
```

Pen.java

```java
public class Pen implements Element {
    @Override
    public void accept(ElementVisitor elementVisitor) {
        elementVisitor.visit(this);
    }
}
```

Pencil.java

```java
public class Pencil implements Element {
    @Override
    public void accept(ElementVisitor elementVisitor) {
        elementVisitor.visit(this);
    }
}
```

PencilSharpener.java

```java
public class PencilSharpener implements Element {
    @Override
    public void accept(ElementVisitor elementVisitor) {
        elementVisitor.visit(this);
    }
}
```

Main.java

```java
import models.Element;
import models.Pen;
import models.Pencil;
import models.PencilSharpener;

import java.util.List;

public class Main {
    public static void main(final String[] parameters) {
        final List<Element> elements = List.of(
                new Pen(),
                new Pencil(),
                new Pencil(),
                new PencilSharpener()
        );
        printTotalPrice(elements);
    }

    private static void printTotalPrice(final List<Element> elements) {
        final TotalPricesCalculator TOTAL_PRICE_CALCULATOR = new TotalPricesCalculator();

        final StringBuilder OUTPUT = new StringBuilder();
        for (Element element : elements) {
            element.accept(TOTAL_PRICE_CALCULATOR);
            OUTPUT.append(getElementPrice(element, TOTAL_PRICE_CALCULATOR)).append("£ + ");
        }

        OUTPUT.setLength(OUTPUT.length() - 3);
        OUTPUT.append(" = ")
                .append(TOTAL_PRICE_CALCULATOR.getTotalPrices())
                .append("£");

        System.out.print(OUTPUT);
    }

    private static double getElementPrice(
            final Element ELEMENT,
            final TotalPricesCalculator TOTAL_PRICE_CALCULATOR
    ) {
        return switch (ELEMENT) {
            case Pen _ -> TOTAL_PRICE_CALCULATOR.getPenPrice();
            case Pencil _ -> TOTAL_PRICE_CALCULATOR.getPencilPrice();
            case PencilSharpener _ -> TOTAL_PRICE_CALCULATOR.getPencilSharpenerPrice();
            default -> 0;
        };
    }
}
```

Output

```
5.0£ + 3.0£ + 3.0£ + 2.0£ = 13.0£
```

نمط الوسيط
Mediator Pattern

التعريف:

هو نمط تصميم يُستخدم لتقليل التعقيد بين الكائنات عن طريق تقديم كائن وسيط يتولى عملية التواصل بينها بدلاً من أن تتواصل مباشرة.

ChatMediator.java

```java
interface ChatMediator {
    void sendMessage(User user, String message);
}
```

User.java

```java
abstract class User {
    protected ChatMediator mediator;
    protected String name;

    public User(ChatMediator mediator, String name) {
        this.mediator = mediator;
        this.name = name;
    }

    public abstract void send(String message);

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

ChatRoom.java

```java
class ChatRoom implements ChatMediator {
    @Override
    public void sendMessage(User user, String message) {
        System.out.println(user.getName() + ": " + message);
    }
}
```

ChatUser.java

```java
class ChatUser extends User {
    public ChatUser(ChatMediator mediator, String name) {
        super(mediator, name);
    }

    @Override
    public void send(String message) {
        mediator.sendMessage(this, message);
    }
}
```

Main.java

```java
public class Main {
    public static void main(final String[] PARAMETERS) {
        ChatMediator chat = new ChatRoom();
        User mohamed = new ChatUser(chat, "Mohamed");
        User ahmed = new ChatUser(chat, "Ahmed");

        mohamed.send("Salam, Ahmed!");
        ahmed.send("Salam, Mohamed!");
    }
}
```

Output

```
Mohamed: Salam, Ahmed!
Ahmed: Salam, Mohamed!
```

نمط التذكار
Memento Pattern

التعريف:

هو نمط تصميم سلوكي يُستخدم لحفظ واستعادة الحالة السابقة لكائن دون انتهاك مبدأ الكبسولة (Encapsulation).

Memento.java

```java
record Memento(String state) {
}
```

Memento.java

```java
class TextEditor {
    private String text = "";

    public void write(String newText) {
        text += newText;
    }

    public Memento save() {
        return new Memento(text);
    }

    public void restore(Memento memento) {
        text = memento.state();
    }

    public void show() {
        System.out.println(text);
    }
}
```

Main.java

```java
public class Main {
    public static void main(final String[] PARAMETERS) {
        TextEditor editor = new TextEditor();

        editor.write("Hello");
        Memento savedState = editor.save();

        editor.write(", World!");
        editor.show();

        editor.restore(savedState);
        editor.show();
    }
}
```

Output

```
Hello, World!
Hello
```

نمط التكرار
Iterator Pattern

التعريف:

هو نمط تصميم سلوكي يتيح الوصول إلى عناصر مجموعة ما بشكل متسلسل دون الحاجة إلى كشف تفاصيل تنفيذها الداخلي.

Aggregate.java

```java
interface Aggregate<T> {
    Iterator<T> createIterator();
}
```

Iterator.java

```java
interface Iterator<T> {
    boolean hasNext();

    T next();
}
```

Company.java

```java
import java.util.List;

class Company implements Aggregate<Employee> {
    private final List<Employee> employees;

    public Company(List<Employee> employees) {
        this.employees = employees;
    }

    @Override
    public Iterator<Employee> createIterator() {
        return new EmployeeIterator(employees);
    }
}
```

EmployeeIterator.java

```java
import java.util.List;
import java.util.NoSuchElementException;

class EmployeeIterator implements Iterator<Employee> {
    private int currentIndex = 0;
    private final List<Employee> employees;

    public EmployeeIterator(List<Employee> employees) {
        this.employees = employees;
    }

    @Override
    public boolean hasNext() {
        return currentIndex < employees.size();
    }

    @Override
    public Employee next() {
        if (!hasNext())
            throw new NoSuchElementException();
        else
            return employees.get(currentIndex++);
    }
}
```

Main.java

```java
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(final String[] PARAMETERS) {
        List<Employee> employees = new ArrayList<>();
        employees.add(new Employee("Ahmed", 50_000));
        employees.add(new Employee("Mahmoud", 60_000));
        employees.add(new Employee("Hamada", 70_000));

        Company company = new Company(employees);
        Iterator<Employee> iterator = company.createIterator();

        double totalSalary = 0;
        while (iterator.hasNext())
            totalSalary += iterator.next()
                    .salary();
        System.out.println("Total salary: " + totalSalary);
    }
}
```

Output

```
Total salary: 180000.0
```

أنماط مضادة

Anti-Patterns

نمط الكود السباغيتي
Spaghetti Code Pattern

التعريف: هو نمط برمجي يتميز بالكود الفوضوي غير المنظم، حيث تتداخل التدفقات المنطقية بشكل معقد، مما يجعل فهمه وصيانته صعبين.

نمط المطرقة الذهبية
Golden Hammer Pattern

التعريف: هو نمط يشير إلى استخدام أداة أو تقنية مألوفة لحل جميع المشكلات، حتى عندما لا تكون الأداة الأنسب.

نمط مرساة القارب
Boat Anchor Pattern

التعريف: هو نمط يشير إلى الاحتفاظ بكود غير مستخدم أو غير ضروري في النظام، مما يزيد التعقيد دون فائدة فعلية.

نمط الكود الميت
Dead Code Pattern

التعريف: هو نمط يشير إلى أجزاء من الشيفرة المصدرية التي لا تُستخدم أو لا تؤثر على النتيجة النهائية للبرنامج ويمكن إزالتها دون تغيير سلوكه.

نمط فئة العقل المدبر وكائن العقل المدبر
Mastermind Class and Mastermind Object Pattern

التعريف: هو نمط يركز على فصل منطق التحكم (العقل المدبر) في كائن منفصل عن البيانات والسلوك، مما يعزز الفصل بين المسؤوليات وقابلية إعادة الاستخدام.

نمط البرمجة بالنسخ واللصق
Copy and Paste Programming Pattern

التعريف: هو نمط يشير إلى تكرار الكود عن طريق نسخه ولصقه في أماكن متعددة بدلاً من إنشاء حلول مرنة وقابلة لإعادة الاستخدام.