Patrones de diseño

Patrones creacionales

Abstract factory: Es un patrón de diseño creacional que nos permite producir familias de objetos relacionados sin especificar sus clases concretas.

```
=====================================================
interface GUIFactory { Button createButton(); }
class WinFactory implements GUIFactory {
    public Button createButton() { return new WinButton(); }
}
interface Button { void paint(); }
class WinButton implements Button {
    public void paint() { System.out.println("Windows Button"); }
}
public class Main {
    public static void main(String[] args) {
        GUIFactory factory = new WinFactory();
        factory.createButton().paint();
    }
}
=====================================================
```

GUIFactory define un método para crear un Button, y WinFactory lo implementa para crear un WinButton.

Builder: Es un patrón de diseño creacional que nos permite construir objetos complejos paso a paso. El patrón nos permite producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.

```
=====================================================
class Pizza {
    private String topping;
    public void setTopping(String topping) { this.topping = topping; }
    public String getDetails() { return "Pizza with " + topping; }
}
class PizzaBuilder {
    private Pizza pizza = new Pizza();
    public PizzaBuilder addTopping(String topping) {
        pizza.setTopping(topping);
        return this;
    }
    public Pizza build() { return pizza; }
}
public class Main {
    public static void main(String[] args) {
        Pizza pizza = new PizzaBuilder().addTopping("cheese").build();
        System.out.println(pizza.getDetails());
    }
}
=====================================================
```

El PizzaBuilder construye una Pizza paso a paso.

Factory Method: Es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

```
=====================================================
interface Transport { void deliver(); }
class Truck implements Transport {
    public void deliver() { System.out.println("Deliver by Truck"); }
}
abstract class Logistics {
    public void planDelivery() {
        Transport transport = createTransport();
        transport.deliver();
    }
    protected abstract Transport createTransport();
}
class RoadLogistics extends Logistics {
    protected Transport createTransport() { return new Truck(); }
}
public class Main {
    public static void main(String[] args) {
        new RoadLogistics().planDelivery();
    }
}
=====================================================
```

RoadLogistics decide crear un Truck usando el método createTransport.

Prototype: Es un patrón de diseño creacional que nos permite copiar objetos existentes sin que el código dependa de sus clases.

```
=====================================================
class Circle implements Cloneable {
    public Circle clone() {
        try { return (Circle) super.clone(); }
        catch (CloneNotSupportedException e) { return null; }
    }
}
public class Main {
    public static void main(String[] args) {
        Circle c1 = new Circle();
        Circle c2 = c1.clone();
        System.out.println("Cloned Circle");
    }
}
=====================================================
```

Circle se clona a sí mismo utilizando el método clone.

Singleton: Es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

```
=======================================================
class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) { instance = new Singleton(); }
        return instance;
    }
}
public class Main {
    public static void main(String[] args) {
        Singleton s = Singleton.getInstance();
        System.out.println("Singleton instance");
    }
}
=======================================================
```

Singleton asegura que solo haya una instancia mediante getInstance.

Patrones estructurales

Adapter: Es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.

```
=====================================================
interface Target {
    void request();
}

class Adaptee {
    void specificRequest() {
        System.out.println("Specific request");
    }
}

class Adapter implements Target {
    private Adaptee adaptee = new Adaptee();
    public void request() {
        adaptee.specificRequest();
    }
}

public class Main {
    public static void main(String[] args) {
        Target target = new Adapter();
        target.request();
    }
}
=====================================================
```

Adapter convierte la interfaz de Adaptee en una interfaz Target que el cliente espera.

Bridge: Es un patrón de diseño estructural que te permite dividir una clase grande, o un grupo de clases estrechamente relacionadas, en dos jerarquías separadas (abstracción e implementación) que pueden desarrollarse independientemente la una de la otra.

```
=====================================================
interface Implementor {
    void operation();
}

class ConcreteImplementorA implements Implementor {
    public void operation() {
        System.out.println("ConcreteImplementorA operation");
    }
}

abstract class Abstraction {
    protected Implementor implementor;
    public Abstraction(Implementor implementor) {
        this.implementor = implementor;
    }
    abstract void operation();
}

class RefinedAbstraction extends Abstraction {
    public RefinedAbstraction(Implementor implementor) {
        super(implementor);
    }
    public void operation() {
        implementor.operation();
    }
}

public class Main {
    public static void main(String[] args) {
        Abstraction abstraction = new RefinedAbstraction(new ConcreteImplementorA());
        abstraction.operation();
    }
}
=====================================================
```

Abstraction y Implementor pueden variar independientemente, permitiendo mayor flexibilidad.

Composite: Es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales.

==========

```
interface Component {
    void operation();
}

class Leaf implements Component {
    public void operation() {
        System.out.println("Leaf operation");
    }
}

class Composite implements Component {
    private List<Component> children = new ArrayList<>();
    public void add(Component component) {
        children.add(component);
    }
    public void operation() {
        for (Component child : children) {
            child.operation();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Composite composite = new Composite();
        composite.add(new Leaf());
        composite.add(new Leaf());
        composite.operation();
    }
}
```

==========

Composite permite tratar objetos individuales (Leaf) y compuestos (Composite) de manera uniforme.

Decorator: es un patrón de diseño estructural que te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.

==========

```
interface Component {
    void operation();
}

class ConcreteComponent implements Component {
    public void operation() {
        System.out.println("ConcreteComponent operation");
    }
}

class Decorator implements Component {
    private Component component;
    public Decorator(Component component) {
        this.component = component;
    }
    public void operation() {
        component.operation();
        System.out.println("Decorator operation");
    }
}

public class Main {
    public static void main(String[] args) {
        Component component = new Decorator(new ConcreteComponent());
        component.operation();
    }
}
```

==========

Composite permite tratar objetos individuales (Leaf) y compuestos (Composite) de manera uniforme.

Facade: es un patrón de diseño estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases

==========

```
class SubsystemA {
    void operationA() {
        System.out.println("SubsystemA operation");
    }
}

class SubsystemB {
    void operationB() {
        System.out.println("SubsystemB operation");
    }
}

class Facade {
    private SubsystemA subsystemA = new SubsystemA();
    private SubsystemB subsystemB = new SubsystemB();

    public void operation() {
        subsystemA.operationA();
        subsystemB.operationB();
    }
}

public class Main {
    public static void main(String[] args) {
        Facade facade = new Facade();
        facade.operation();
    }
}
```

==========

Facade simplifica las interacciones con varios subsistemas (SubsystemA, SubsystemB) a través de una única interfaz.

Flyweight: Es un patrón de diseño estructural que te permite mantener más objetos dentro de la cantidad disponible de RAM compartiendo las partes comunes del estado entre varios objetos en lugar de mantener toda la información en cada objeto.

==========

```
import java.util.HashMap;
import java.util.Map;

interface Flyweight {
    void operation(String extrinsicState);
}
class ConcreteFlyweight implements Flyweight {
    private String intrinsicState;
    public ConcreteFlyweight(String intrinsicState) {
        this.intrinsicState = intrinsicState;
    }
    public void operation(String extrinsicState) {
        System.out.println("Intrinsic: " + intrinsicState + ", Extrinsic: " + extrinsicState);
    }
}
class FlyweightFactory {
    private Map<String, Flyweight> flyweights = new HashMap<>();

    public Flyweight getFlyweight(String key) {
        if (!flyweights.containsKey(key)) {
            flyweights.put(key, new ConcreteFlyweight(key));
        }
        return flyweights.get(key);
    }
}
public class Main {
    public static void main(String[] args) {
        FlyweightFactory factory = new FlyweightFactory();
        Flyweight fw1 = factory.getFlyweight("state1");
        Flyweight fw2 = factory.getFlyweight("state1");
        fw1.operation("extrinsic1");
        fw2.operation("extrinsic2");
    }
}
```

==========

FlyweightFactory asegura que se reutilicen instancias (ConcreteFlyweight) para minimizar el consumo de memoria

Proxy: Es un patrón de diseño estructural que te permite proporcionar un sustituto o marcador de posición para otro objeto. Un proxy controla el acceso al objeto original, permitiéndole hacer algo antes o después de que la solicitud llegue al objeto original

==========

```
interface Subject {
    void request();
}

class RealSubject implements Subject {
    public void request() {
        System.out.println("RealSubject request");
    }
}

class Proxy implements Subject {
    private RealSubject realSubject;
    public void request() {
        if (realSubject == null) {
            realSubject = new RealSubject();
        }
        realSubject.request();
    }
}

public class Main {
    public static void main(String[] args) {
        Subject proxy = new Proxy();
        proxy.request();
    }
}
```

==========

Proxy controla el acceso a RealSubject, instanciándolo solo cuando es necesario.

Patrones de comportamiento

Chain of responsibility: Es un patrón de diseño de comportamiento que te permite pasar solicitudes a lo largo de una cadena de manejadores. Al recibir una solicitud, cada manejador decide si la procesa o si la pasa al siguiente manejador de la cadena.

```
=====================================================
abstract class Handler {
    protected Handler next;
    public void setNext(Handler next) {
        this.next = next;
    }
    public void handleRequest(String request) {
        if (next != null) next.handleRequest(request);
    }
}
class ConcreteHandler1 extends Handler {
    public void handleRequest(String request) {
        if (request.equals("one")) {
            System.out.println("Handled by ConcreteHandler1");
        } else {
            super.handleRequest(request);
        }
    }
}
class ConcreteHandler2 extends Handler {
    public void handleRequest(String request) {
        if (request.equals("two")) {
            System.out.println("Handled by ConcreteHandler2");
        } else {
            super.handleRequest(request);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Handler handler1 = new ConcreteHandler1();
        Handler handler2 = new ConcreteHandler2();
        handler1.setNext(handler2);
        handler1.handleRequest("two");
    }
}
=====================================================
```

La solicitud se pasa a través de una cadena de manejadores hasta que uno de ellos la procesa..

Command: Es un patrón de diseño de comportamiento que convierte una solicitud en un objeto independiente que contiene toda la información sobre la solicitud. Esta transformación te permite parametrizar los métodos con diferentes solicitudes, retrasar o poner en cola la ejecución de una solicitud y soportar operaciones que no se pueden realizar.

```
=====================================================
interface Command {
    void execute();
}

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

class LightOnCommand implements Command {
    private Light light;
    public LightOnCommand(Light light) {
        this.light = light;
    }
    public void execute() {
        light.on();
    }
}
class RemoteControl {
    private Command command;
    public void setCommand(Command command) {
        this.command = command;
    }
    public void pressButton() {
        command.execute();
    }
}
public class Main {
    public static void main(String[] args) {
        Light light = new Light();
        Command lightOn = new LightOnCommand(light);
        RemoteControl remote = new RemoteControl();
        remote.setCommand(lightOn);
        remote.pressButton();
    }
}
=====================================================
```

Command encapsula la acción (LightOnCommand) y el RemoteControl invoca la ejecución de esa acción.

Iterator: Es un patrón de diseño de comportamiento que te permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).

==========

```
import java.util.Iterator;
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");
        list.add("C");

        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

==========

El patrón Iterator proporciona una manera uniforme de recorrer una colección sin exponer los detalles internos de la misma.

Mediator: Es un patrón de diseño de comportamiento que te permite reducir las dependencias caóticas entre objetos. El patrón restringe las comunicaciones directas entre los objetos, forzándolos a colaborar únicamente a través de un objeto mediador.

==========

```
class Mediator {
    private Colleague1 colleague1;
    private Colleague2 colleague2;

    public void registerColleague1(Colleague1 colleague) {
        this.colleague1 = colleague;
    }

    public void registerColleague2(Colleague2 colleague) {
        this.colleague2 = colleague;
    }

    public void send(String message, Colleague sender) {
        if (sender == colleague1) {
            colleague2.notify(message);
        } else {
            colleague1.notify(message);
        }
    }
}

abstract class Colleague {
    protected Mediator mediator;
    public Colleague(Mediator mediator) {
        this.mediator = mediator;
    }
}

class Colleague1 extends Colleague {
    public Colleague1(Mediator mediator) {
        super(mediator);
    }
    public void send(String message) {
        mediator.send(message, this);
    }
    public void notify(String message) {
        System.out.println("Colleague1 gets message: " + message);
    }
}

class Colleague2 extends Colleague {
    public Colleague2(Mediator mediator) {
        super(mediator);
```

```
    }
    public void send(String message) {
        mediator.send(message, this);
    }
    public void notify(String message) {
        System.out.println("Colleague2 gets message: " + message);
    }
}

public class Main {
    public static void main(String[] args) {
        Mediator mediator = new Mediator();
        Colleague1 colleague1 = new Colleague1(mediator);
        Colleague2 colleague2 = new Colleague2(mediator);

        mediator.registerColleague1(colleague1);
        mediator.registerColleague2(colleague2);

        colleague1.send("Hello");
        colleague2.send("Hi");
    }
}
```

==========

Mediator facilita la comunicación entre Colleague1 y Colleague2 sin que interactúen directamente.

Memento: Es un patrón de diseño de comportamiento que te permite guardar y restaurar el estado previo de un objeto sin revelar los detalles de su implementación.

==========

```
class Memento {
    private String state;
    public Memento(String state) {
        this.state = state;
    }
    public String getState() {
        return state;
    }
}

class Originator {
    private String state;
    public void setState(String state) {
        this.state = state;
    }
    public Memento saveStateToMemento() {
        return new Memento(state);
    }
    public void getStateFromMemento(Memento memento) {
        state = memento.getState();
    }
    public String getState() {
        return state;
    }
}

class Caretaker {
    private Memento memento;
    public void saveState(Memento memento) {
        this.memento = memento;
    }
    public Memento restoreState() {
        return memento;
    }
}

public class Main {
    public static void main(String[] args) {
        Originator originator = new Originator();
        Caretaker caretaker = new Caretaker();

        originator.setState("State1");
        caretaker.saveState(originator.saveStateToMemento());
```

```
        originator.setState("State2");
        originator.getStateFromMemento(caretaker.restoreState());

        System.out.println("Restored State: " + originator.getState());
    }
}
```
==========

Memento guarda el estado interno de Originator, permitiendo que Caretaker lo restaure más tarde.

Observer: Es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

==========

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

interface Observer {
    void update(String state);
}

class Subject {
    private List<Observer> observers = new ArrayList<>();
    private String state;

    public void setState(String state) {
        this.state = state;
        notifyObservers();
    }
    public void addObserver(Observer observer) {
        observers.add(observer);
    }
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(state);
        }
    }
}
class ConcreteObserver implements Observer {
    public void update(String state) {
        System.out.println("Observer state updated to: " + state);
    }
}
public class Main {
    public static void main(String[] args) {
        Subject subject = new Subject();
        Observer observer = new ConcreteObserver();
        subject.addObserver(observer);

        subject.setState("New State");
    }
}
```

==========

Subject notifica a todos sus Observer cuando cambia su estado, manteniéndolos sincronizados.

State: Es un patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiara su clase.

==========

```
interface State {
    void handle(Context context);
}

class ConcreteStateA implements State {
    public void handle(Context context) {
        System.out.println("State A");
        context.setState(new ConcreteStateB());
    }
}

class ConcreteStateB implements State {
    public void handle(Context context) {
        System.out.println("State B");
        context.setState(new ConcreteStateA());
    }
}

class Context {
    private State state;
    public Context(State state) {
        this.state = state;
    }
    public void setState(State state) {
        this.state = state;
    }
    public void request() {
        state.handle(this);
    }
}

public class Main {
    public static void main(String[] args) {
        Context context = new Context(new ConcreteStateA());
        context.request();
        context.request();
    }
}
```

==========

Context cambia su comportamiento basado en su estado interno (ConcreteStateA, ConcreteStateB).

Strategy: Es un patrón de diseño de comportamiento que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

==========

```
interface Strategy {
    void execute();
}

class ConcreteStrategyA implements Strategy {
    public void execute() {
        System.out.println("Strategy A");
    }
}

class ConcreteStrategyB implements Strategy {
    public void execute() {
        System.out.println("Strategy B");
    }
}

class Context {
    private Strategy strategy;
    public Context(Strategy strategy) {
        this.strategy = strategy;
    }
    public void setStrategy(Strategy strategy) {
        this.strategy = strategy;
    }
    public void executeStrategy() {
        strategy.execute();
    }
}

public class Main {
    public static void main(String[] args) {
        Context context = new Context(new ConcreteStrategyA());
        context.executeStrategy();
        context.setStrategy(new ConcreteStrategyB());
        context.executeStrategy();
    }
}
```

==========

Context permite cambiar el Strategy (algoritmo) en tiempo de ejecución sin modificar el código del cliente.

Template method: Es un patrón de diseño de comportamiento que define el esqueleto de un algoritmo en la superclase pero permite que las subclases sobreescriban pasos del algoritmo sin cambiar su estructura.

```
==========================================================
abstract class AbstractClass {
    public final void templateMethod() {
        stepOne();
        stepTwo();
    }
    abstract void stepOne();
    abstract void stepTwo();
}

class ConcreteClass extends AbstractClass {
    void stepOne() {
        System.out.println("Step One");
    }
    void stepTwo() {
        System.out.println("Step Two");
    }
}

public class Main {
    public static void main(String[] args) {
        AbstractClass instance = new ConcreteClass();
        instance.templateMethod();
    }
}
==========================================================
```

ConcreteClass proporciona las implementaciones específicas de los métodos, mientras que AbstractClass define el esqueleto del algoritmo.

Visitor: Es un patrón de diseño de comportamiento que te permite separar algoritmos de los objetos sobre los que operan.

==========

```
interface Visitor {
    void visit(ElementA element);
    void visit(ElementB element);
}

interface Element {
    void accept(Visitor visitor);
}

class ElementA implements Element {
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

class ElementB implements Element {
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

class ConcreteVisitor implements Visitor {
    public void visit(ElementA element) {
        System.out.println("Visited ElementA");
    }
    public void visit(ElementB element) {
        System.out.println("Visited ElementB");
    }
}

public class Main {
    public static void main(String[] args) {
        Element[] elements = { new ElementA(), new ElementB() };
        Visitor visitor = new ConcreteVisitor();
        for (Element element : elements) {
            element.accept(visitor);
        }
    }
}
```

==========

Visitor permite añadir nuevas operaciones (ConcreteVisitor) sobre los elementos (ElementA, ElementB) sin cambiar su estructura.

Referencias

Refactoring Guru()*Patrones de diseño*
https://refactoring.guru/es/design-patterns/catalog