**PATRONES DE DISEÑO**

**¿QUE ES UN PATRON DE DISEÑO?**

Es una solución general y reutilizable para un problema común en el diseño de software. Es una forma de abordar un problema especifico de manera eficiente y efectiva.

**CLASIFICACION PATRONES DE DISEÑO:**

Estos se clasifican en tres categorías:

* PATRONES CREACIONALES: Se enfocan en la creación de objetos.
* PATRONES ESTRUCTURALES: Se enfocan en la organización de objetos y clases.
* PATRONES COMPORTAMENTALES: Se enfocan en la interacción entre objetos.

**VENTAJAS:**

 Mejoran la legibilidad y mantenimiento del código.

 Reducen la complejidad del código.

 Aumentan la reutilización de código.

**¿QUIEN LOS INVENTO?**

Los patrones de diseño fueron popularizados por el libro "Design Patterns: Elements of Reusable Object-Oriented Software" de Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, publicado en 1994.

**¿Se puede utilizar los patrones fuera del desarrollo de Software, cómo se hace?**

Sí, los patrones de diseño se pueden aplicar en otros campos, como la arquitectura, la ingeniería y la gestión de proyectos. Se pueden utilizar para resolver problemas comunes y mejorar la eficiencia en diferentes contextos.

**Patrón creacional:**

Un patrón creacional es un patrón de diseño que se enfoca en la creación de objetos. Se clasifican en:

* Patrón Singleton: garantiza que solo se cree una instancia de una clase.
* Patrón Factory Method: proporciona una forma de crear objetos sin especificar la clase exacta.
* Patrón Abstract Factory: proporciona una forma de crear familias de objetos relacionados.
* Patrón Builder: separa la construcción de un objeto de su representación.

**Patrón de Método de Fábrica (Factory Method)**

Un patrón de Método de Fábrica es un patrón creacional que proporciona una forma de crear objetos sin especificar la clase exacta. Su propósito es encapsular la creación de objetos y permitir la extensión de la creación de objetos sin modificar la clase base.

**Problema:** La creación de objetos es compleja y depende de la clase exacta.

**Solución:** Utilizar un método de fábrica que crea objetos sin especificar la clase exacta.

**Estructura:**

* La clase base define el método de fábrica.
* Las clases derivadas implementan el método de fábrica.

**Aplicabilidad:** Se utiliza cuando se necesita crear objetos sin especificar la clase exacta.

**Método Abstracto(Factory Abstract):**

Un patrón de Fábrica Abstracto es un patrón creacional que proporciona una forma de crear familias de objetos relacionados. Su propósito es encapsular la creación de objetos y permitir la extensión de la creación de objetos sin modificar la clase base.

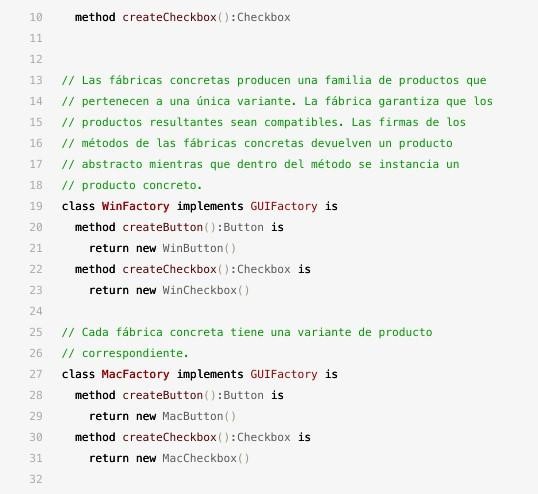
Problema: La creación de objetos es compleja y depende de la clase exacta.

Solución: Utilizar una fábrica abstracta que crea objetos sin especificar la clase exacta.

Estructura:

* La clase base define la fábrica abstracta.
* Las clases derivadas implementan la fábrica abstracta.

Aplicabilidad: Se utiliza cuando se necesita crear familias de objetos relacionados.



**PATRON BUILDER:**

Un patrón Builder es un patrón creacional que separa la construcción de un objeto de su representación. Su propósito es permitir la creación de objetos complejos paso a paso.

Problema: La creación de objetos complejos es difícil y depende de la clase exacta.

Solución: Utilizar un constructor que crea objetos paso a paso.

Estructura:

* La clase base define el constructor.
* Las clases derivadas implementan el constructor.

Aplicabilidad: Se utiliza cuando se necesita crear objetos complejos paso a paso.

// El uso del patrón Builder sólo tiene sentido cuando tus

// productos son bastante complejos y requieren una

// configuración extensiva. Los dos siguientes productos están

// relacionados, aunque no tienen una interfaz común. class Car is

// Un coche puede tener un GPS, una computadora de

// navegación y cierto número de asientos. Los distintos

// modelos de coches (deportivo, SUV, descapotable) pueden

// tener distintas características instaladas o habilitadas. class Manual is

// Cada coche debe contar con un manual de usuario que se

// corresponda con la configuración del coche y explique // todas sus características.

// La interfaz constructora especifica métodos para crear las

// distintas partes de los objetos del producto. interface Builder is method reset() method setSeats(...) method setEngine(...) method setTripComputer(...) method setGPS(...)

// Las clases constructoras concretas siguen la interfaz

// constructora y proporcionan implementaciones específicas de

// los pasos de construcción. Tu programa puede tener multitud

// de variaciones de objetos constructores, cada una de ellas

// implementada de forma diferente. class CarBuilder implements Builder is private field car:Car

// Una nueva instancia de la clase constructora debe // contener un objeto de producto en blanco que utiliza en

// el montaje posterior. constructor CarBuilder() is this.reset()

// El método reset despeja el objeto en construcción.

method reset() is this.car = new Car()

// Todos los pasos de producción funcionan con la misma

// instancia de producto. method setSeats(...) is

// Establece la cantidad de asientos del coche.

method setEngine(...) is // Instala un motor específico. method setTripComputer(...) is // Instala una computadora de navegación.

method setGPS(...) is

// Instala un GPS.

// Los constructores concretos deben proporcionar sus

// propios métodos para obtener resultados. Esto se debe a

// que varios tipos de objetos constructores pueden crear

// productos completamente diferentes de los cuales no todos

// siguen la misma interfaz. Por lo tanto, dichos métodos no

// pueden declararse en la interfaz constructora (al menos

// no en un lenguaje de programación de tipado estático).

//

// Normalmente, tras devolver el resultado final al cliente,

// una instancia constructora debe estar lista para empezar

// a generar otro producto. Ese es el motivo por el que es

// práctica común invocar el método reset al final del

// cuerpo del método `getProduct`. Sin embargo, este

// comportamiento no es obligatorio y puedes hacer que tu // objeto constructor espere una llamada reset explícita del

// código cliente antes de desechar el resultado anterior. method getProduct():Car is product = this.car this.reset() return product

// Al contrario que otros patrones creacionales, Builder te // permite construir productos que no siguen una interfaz común. class CarManualBuilder implements Builder is private field manual:Manual constructor CarManualBuilder() is this.reset() method reset() is this.manual = new Manual() method setSeats(...) is

// Documenta las características del asiento del coche

**PATRON SINGLETON:**

Un patrón Singleton es un patrón creacional que garantiza que solo se cree una instancia de una clase. Su propósito es proporcionar un acceso global a una instancia única.

Problema: La creación de objetos es compleja y depende de la clase exacta.

Solución: Utilizar un patrón Singleton que garantiza que solo se cree una instancia de una clase.

Estructura:

* La clase Singleton define la instancia única.

Aplicabilidad: Se utiliza cuando se necesita proporcionar un acceso global a una instancia única.

**PATRON ESTRUCTURAL:**

Un patrón estructural es un patrón de diseño que se enfoca en la organización de objetos y clases. Se clasifican en:

* Patrón Adapter: permite que objetos incompatibles trabajen juntos.
* Patrón Bridge: separa la abstracción de la implementación.
* Patrón Composite: permite que objetos se compongan de otros objetos.
* Patrón Decorator: permite que objetos se decoren con nuevos comportamientos.
* Patrón Facade: proporciona una interfaz simplificada para un sistema complejo.
* Patrón Flyweight: reduce la cantidad de memoria utilizada por objetos.
* Patrón Proxy: proporciona un acceso controlado a un objeto.

**PATRON ADAPTER:**

Un patrón Adapter es un patrón estructural que permite que objetos incompatibles trabajen juntos. Su propósito es adaptar la interfaz de un objeto para que sea compatible con otra interfaz.

Problema: Los objetos tienen interfaces incompatibles.

Solución: Utilizar un adaptador que adapte la interfaz de un objeto para que sea compatible con otra interfaz.

Estructura:

* La clase Adapter adapta la interfaz de un objeto.

Aplicabilidad: Se utiliza cuando se necesita que objetos incompatibles trabajen juntos.

// Digamos que tienes dos clases con interfaces compatibles:

// RoundHole (HoyoRedondo) y RoundPeg (PiezaRedonda).

class RoundHole is constructor RoundHole(radius) { ... } method getRadius() is

// Devuelve el radio del agujero. method fits(peg: RoundPeg) is return this.getRadius() >= peg.getRadius() class RoundPeg is constructor RoundPeg(radius) { ... } method getRadius() is

// Devuelve el radio de la pieza.

// Pero hay una clase incompatible: SquarePeg (PiezaCuadrada). class SquarePeg is constructor SquarePeg(width) { ... } method getWidth() is

// Devuelve la anchura de la pieza cuadrada.

// Una clase adaptadora te permite encajar piezas cuadradas en

// hoyos redondos. Extiende la clase RoundPeg para permitir a

// los objetos adaptadores actuar como piezas redondas. class SquarePegAdapter extends RoundPeg is // En realidad, el adaptador contiene una instancia de la

// clase SquarePeg. private field peg: SquarePeg constructor SquarePegAdapter(peg: SquarePeg) is this.peg = peg method getRadius() is

// El adaptador simula que es una pieza redonda con un

// radio que pueda albergar la pieza cuadrada que el

// adaptador envuelve.

return peg.getWidth() \* Math.sqrt(2) / 2

// En algún punto del código cliente.

hole = new RoundHole(5) rpeg = new RoundPeg(5) hole.fits(rpeg) // verdadero small\_sqpeg = new SquarePeg(5) large\_sqpeg = new SquarePeg(10) hole.fits(small\_sqpeg) // esto no compila (tipos incompatibles) small\_sqpeg\_adapter = new SquarePegAdapter(small\_sqpeg) large\_sqpeg\_adapter = new SquarePegAdapter(large\_sqpeg) hole.fits(small\_sqpeg\_adapter) // verdadero hole.fits(large\_sqpeg\_adapter) // falso

**Patrón Bridge**

Propósito: desacoplar dos abstracciones: una de su implementación y permitir que cada una pueda evolucionar independientemente.

Problema: a medida que una aplicación crece, puede tener una combinatoria de clases de implementación y abstracción, lo que significa dificultades de mantenimiento.

Solución: separar la interfaz de la abstracción y la implementación para que ambas puedan evolucionar de forma independiente.

Estructura:

Abstracción: define la abstracción para el cliente; Implementador: define la interfaz para todas las implementaciones; Refinamiento de la abstracción: extiende la abstracción; Implementaciones concretas: clases que implementan la interfaz de implementador.

Aplicabilidad: cuando hay varias implementaciones de una sola interfaz y desea evitar que existan muchos subtipos de subclases.

Patrón Composite

Propósito: componer objetos de estructuras de árbol para representar jerarquías de parte: entera.

Problema: necesidad de tratar de la misma manera a los objetos indiferenciadamente simples y compuestos.

Solución: definir una interfaz común para operar tanto sobre los objetos simples como compuestos; entonces el cliente no necesitará saber la diferencia.

Estructura:

Componente: interfaz común para todos los objetos: simples y compuestos.

Hoja: objetos simples que no tienen hijos.

Composite: objetos compuestos que tienen hijos que a su vez pueden ser hojas o composites.

Aplicabilidad: Útil cuando el sistema podía representarse como una jerarquía, por ej., gráficos, menús.

// La "abstracción" define la interfaz para la parte de

// "control" de las dos jerarquías de clase. Mantiene una // referencia a un objeto de la jerarquía de "implementación" y

// delega todo el trabajo real a este objeto.

class RemoteControl is protected field device: Device constructor RemoteControl(device: Device) is this.device = device method togglePower() is if (device.isEnabled()) then device.disable() else device.enable() method volumeDown() is device.setVolume(device.getVolume()- 10) method volumeUp() is device.setVolume(device.getVolume() + 10) method channelDown() is device.setChannel(device.getChannel()- 1) method channelUp() is device.setChannel(device.getChannel() + 1) // Puedes extender clases de la jerarquía de abstracción // independientemente de las clases de dispositivo. class AdvancedRemoteControl extends RemoteControl is method mute() is device.setVolume(0) interface Device is method isEnabled() method enable() method disable() method getVolume() method setVolume(percent) method getChannel() method setChannel(channel)

// Todos los dispositivos siguen la misma interfaz.

class Tv implements Device is

// ...

class Radio implements Device is

// ...

// En algún lugar del código cliente. tv = new Tv() remote = new RemoteControl(tv)

**Patrón Composite**

**Propósito:** Componer objetos en estructuras de árbol para representar jerarquías parte-todo.

**Problema:** Necesidad de tratar objetos individuales y compuestos de manera uniforme.

**Solución:** Crear una interfaz común que permita operaciones en objetos simples y compuestos.

**Estructura:**

* **Componente**: interfaz común.
* **Hoja**: representa objetos simples.
* **Composite**: representa objetos compuestos.

**Aplicabilidad:** Útil en sistemas que representan jerarquías como gráficos o menús.

**Ejemplo de Código:** interface Graphic { void print();

}

class Dot implements Graphic {

public void print() {

System.out.println("Dibujo de un punto.");

}

}

class Circle implements Graphic {

List<Graphic> shapes = new ArrayList<>(); public void add(Graphic graphic) { shapes.add(graphic);

}

public void print() { for (Graphic shape : shapes) { shape.print();

}

}

}

public class CompositePatternDemo { public static void main(String[] args) {

Graphic dot = new Dot(); Circle circle = new Circle(); circle.add(dot); circle.print();

}

}

**Patrón Decorator**

**Propósito:** Añadir comportamiento a objetos de manera dinámica.

**Problema:** Aumentar las subclases puede llevar a una explosión de clases.

**Solución:** Utilizar una clase de envoltura que pueda añadir funcionalidades a un objeto.

**Estructura:**

* **Componente**: interfaz común.
* **Componente Concreto**: implementa la interfaz.
* **Decorator**: mantiene una referencia al componente.
* **Decoradores Concretos**: añaden funcionalidades.

**Aplicabilidad:** Útil para añadir responsabilidades a objetos sin alterar sus clases.

interface Coffee { double cost();

}

class SimpleCoffee implements Coffee { public double cost() {

return 1.0;

}

}

abstract class CoffeeDecorator implements Coffee { protected Coffee coffee; public CoffeeDecorator(Coffee coffee) { this.coffee = coffee;

}

}

class MilkDecorator extends CoffeeDecorator { public MilkDecorator(Coffee coffee) { super(coffee);

}

public double cost() { return coffee.cost() + 0.5;

}

}

public class DecoratorPatternDemo { public static void main(String[] args) { Coffee coffee = new SimpleCoffee(); coffee = new MilkDecorator(coffee);

System.out.println("Costo total: " + coffee.cost());

}

}

**Patrón Facade**

**Propósito:** Proporcionar una interfaz simplificada a un conjunto de interfaces en un subsistema.

**Problema:** Interactuar con múltiples clases complejas puede ser complicado.

**Solución:** Crear una clase fachada que encapsule la complejidad.

**Estructura:**

* **Facade**: proporciona métodos simplificados.
* **Subsystem Classes**: clases que contienen la lógica compleja. **Aplicabilidad:** Útil para simplificar la interacción con sistemas complejos.

class SubsystemA { void operationA() {

System.out.println("Operación A");

}

}

class SubsystemB { void operationB() {

System.out.println("Operación B");

}

class Facade { private SubsystemA a = new SubsystemA(); private SubsystemB b = new SubsystemB();

public void simpleOperation() {

a.operationA();

b.operationB();

}

}

public class FacadePatternDemo { public static void main(String[] args) { Facade facade = new Facade(); facade.simpleOperation();

}

}

**5. Patrón Flyweight**

**Propósito:** Minimizar el uso de memoria mediante el uso compartido de objetos.

**Problema:** Crear demasiados objetos puede consumir mucha memoria.

**Solución:** Utilizar objetos compartidos para representar estados comunes.

**Estructura:**

* **Flyweight**: interfaz que define métodos.
* **ConcreteFlyweight**: implementa el Flyweight.
* **FlyweightFactory**: crea y gestiona Flyweights.

**Aplicabilidad:** Útil en sistemas que manejan grandes cantidades de objetos similares.

**Ejemplo de Código:** interface Shape { void draw(); class Circle implements Shape { private String color; public Circle(String color) { this.color = color;

}

public void draw() {

System.out.println("Círculo de color: " + color);

}

}

class ShapeFactory { private HashMap<String, Shape> shapes = new HashMap<>(); public Shape getCircle(String color) { if (!shapes.containsKey(color)) { shapes.put(color, new Circle(color));

}

return shapes.get(color);

}

}

public class FlyweightPatternDemo { public static void main(String[] args) {

ShapeFactory factory = new ShapeFactory();

Shape redCirce = factory.getCircle("Rojo");

Shape greenCircle = factory.getCircle("Verde"); redCircle.draw(); greenCircle.draw();

}

}

**Patrón Proxy**

**Propósito:** Proveer un sustituto o representante para otro objeto para controlar el acceso a este.

**Problema:** Acceder a objetos remotos o costosos puede ser ineficiente.

**Solución:** Usar un objeto proxy que maneje el acceso al objeto real.

**Estructura:**

* **Subject**: interfaz común.
* **RealSubject**: objeto real.
* **Proxy**: controla el acceso al RealSubject.

**Aplicabilidad:** Útil para manejar accesos remotos o costosos.

interface Image { void display();

}

class RealImage implements Image { private String filename;

public RealImage(String filename) { this.filename = filename; loadImageFromDisk();

}

private void loadImageFromDisk() {

System.out.println("Cargando " + filename);

}

public void display() {

System.out.println("Mostrando " + filename);

}

}

class ProxyImage implements Image { private RealImage realImage; private String filename;

public ProxyImage(String filename) { this.filename = filename;

}

public void display() { if (realImage == null) { realImage = new RealImage(filename);

}

realImage.display();

}

}

public class ProxyPatternDemo { public static void main(String[] args) { Image image = new ProxyImage("foto.jpg"); image.display(); // Carga image.display(); // Muestra sin cargar de nuevo

}

}

**Patrón Chain of Responsibility**

**Propósito:** Pasar solicitudes a lo largo de una cadena de manejadores.

**Problema:** Necesidad de desacoplar el remitente de la respuesta.

**Solución:** Crear una cadena de objetos que manejan la solicitud.

**Estructura:**

* **Handler**: interfaz o clase abstracta para manejar solicitudes.
* **ConcreteHandlers**: implementaciones que manejan la solicitud.

**Aplicabilidad:** Útil en sistemas donde se requieren múltiples objetos para manejar solicitudes. abstract class Handler { protected Handler next; public void setNext(Handler next) { this.next = next;

}

public abstract void handleRequest(int request);

}

class ConcreteHandlerA extends Handler { public void handleRequest(int request) { if (request < 10) {

System.out.println("Handler A manejó la solicitud: " + request);

} else if (next != null) { next.handleRequest(request);

}

}

}

class ConcreteHandlerB extends Handler { public void handleRequest(int request) { if (request < 20) {

System.out.println("Handler B manejó la solicitud: " + request);

} else if (next != null) { next.handleRequest(request);

}

}

public class ChainOfResponsibilityDemo { public static void main(String[] args) {

Handler handlerA = new ConcreteHandlerA();

Handler handlerB = new ConcreteHandlerB();

handlerA.setNext(handlerB);

handlerA.handleRequest(15); handlerA.handleRequest(5);

}

}

**Patrón Command**

**Propósito:** Encapsular una solicitud como un objeto, permitiendo parametrizar clientes.

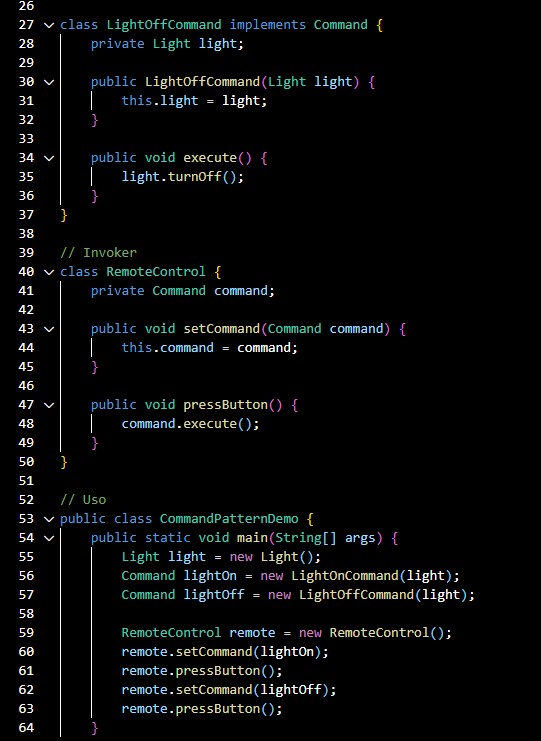
**Problema:** Dificultad para manejar operaciones que necesitan deshacer o rehacer.

**Solución:** Crear comandos que pueden ser ejecutados, desechados o almacenados.

**Estructura:**

* **Command**: interfaz para ejecutar operaciones.
* **ConcreteCommand**: implementación del comando.
* **Invoker**: llama al comando.
* **Receiver**: sabe cómo realizar las acciones.

**Aplicabilidad:** Útil en sistemas que requieren deshacer/rehacer o colas de operaciones.



**Patrón Iterator**

**Propósito:** Proveer un medio para acceder a los elementos de un agregado sin exponer su representación interna.

**Problema:** Dificultad para iterar sobre colecciones sin exponer su implementación.

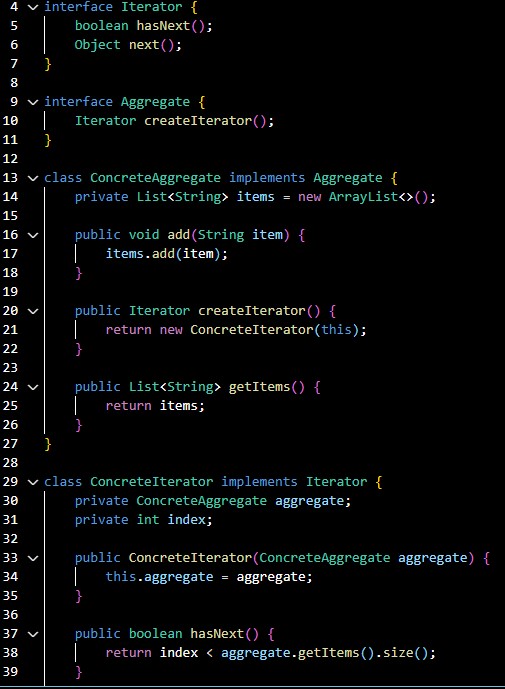
**Solución:** Crear un objeto iterador que maneje la iteración.

**Estructura:**

* **Iterator**: interfaz para acceder a elementos.
* **ConcreteIterator**: implementa la iteración. • **Aggregate**: interfaz para crear un iterador.
* **ConcreteAggregate**: implementa la creación de un iterador.

**Aplicabilidad:** Útil para colecciones de datos.

**Ejemplo de Código:**



**10. Patrón Mediator**

**Propósito:** Definir un objeto que encapsula la forma en que interactúan un conjunto de objetos.

**Problema:** Las interacciones complejas entre objetos pueden hacer que el código sea difícil de mantener.

**Solución:** Usar un mediador para manejar la comunicación entre objetos.

**Estructura:**

* **Mediator**: interfaz para la comunicación.
* **ConcreteMediator**: implementación que gestiona la interacción.
* **Colleagues**: objetos que interactúan a través del mediador.

**Aplicabilidad:** Útil en sistemas con muchas interacciones entre objetos.

**Ejemplo de Código:**



**Patrón Memento**

**Propósito:** Capturar y externalizar el estado interno de un objeto sin violar su encapsulamiento.

**Problema:** Necesidad de restaurar el estado anterior de un objeto.

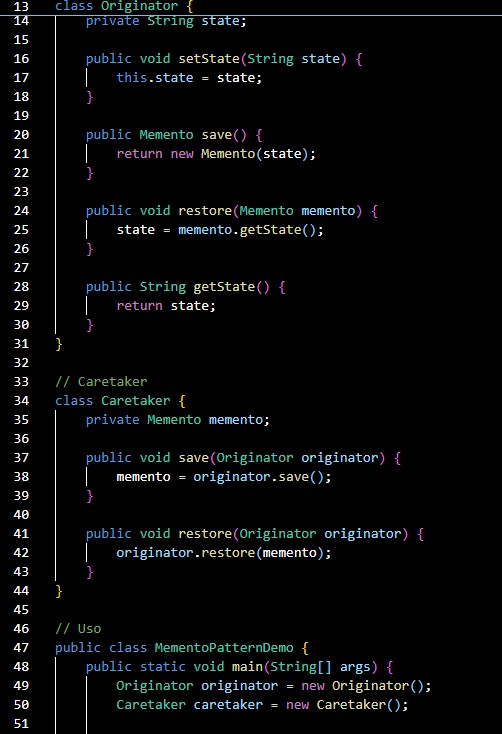
**Solución:** Usar un objeto memento para almacenar el estado del objeto.

**Estructura:**

* **Memento**: objeto que almacena el estado.
* **Originator**: objeto cuyo estado se guarda.
* **Caretaker**: responsable de almacenar y restaurar el memento.

**Aplicabilidad:** Útil en sistemas donde se requiere deshacer acciones.

**Ejemplo de Código:**



**12. Patrón Observer**

**Propósito:** Definir una dependencia uno a muchos entre objetos para que cuando uno cambie de estado, todos sus dependientes sean notificados.

**Problema:** Necesidad de notificar múltiples objetos sobre cambios en el estado.

**Solución:** Usar un patrón que permita registrar objetos interesados en ser notificados.

**Estructura:**

* **Subject**: objeto que tiene estado y notifica a los observadores.
* **Observer**: interfaz que define el método de actualización.
* **ConcreteSubject**: implementación del sujeto que mantiene una lista de observadores.
* **ConcreteObserver**: implementación del observador.

**Aplicabilidad:** Útil en sistemas donde hay cambios de estado que deben ser comunicados.

**Ejemplo de Código:**



**Patrón State**

**Propósito:** Permitir que un objeto altere su comportamiento cuando su estado interno cambie.

**Problema:** La lógica de estado puede volverse compleja.

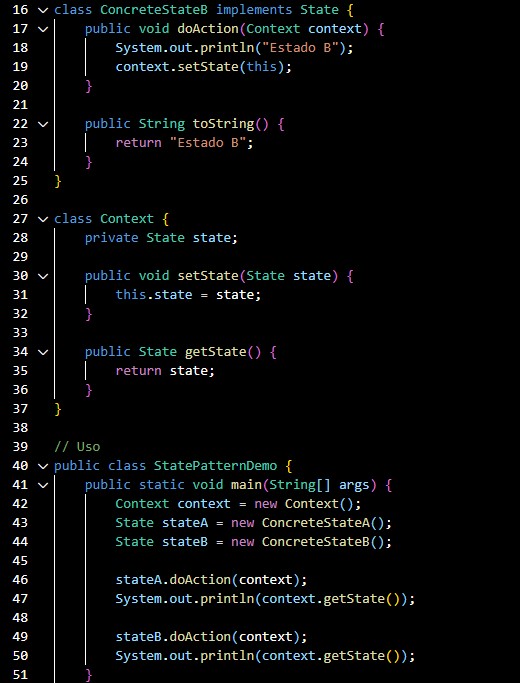
**Solución:** Usar objetos de estado que encapsulen el comportamiento.

**Estructura:**

* **Context**: mantiene una referencia al estado.
* **State**: interfaz para estados.
* **ConcreteStates**: implementaciones de los estados.

**Aplicabilidad:** Útil en sistemas donde un objeto tiene estados distintos que cambian su comportamiento.

**Ejemplo de Código:**



**Patrón Strategy**

**Propósito:** Definir una familia de algoritmos, encapsular cada uno y hacerlos intercambiables.

**Problema:** Dificultad para manejar múltiples algoritmos que pueden variar.

**Solución:** Crear una interfaz para los algoritmos y encapsular cada uno en una clase.

**Estructura:**

* **Strategy**: interfaz común para los algoritmos.
* **ConcreteStrategies**: implementaciones de los algoritmos.
* **Context**: referencia a la estrategia.

**Aplicabilidad:** Útil cuando se pueden definir múltiples formas de realizar una operación.

**Ejemplo de Código:**



**Patrón Template Method**

**Propósito:** Definir el esqueleto de un algoritmo en una operación, permitiendo que las subclases redefinan ciertos pasos sin cambiar la estructura del algoritmo.

**Problema:** Las subclases pueden tener diferentes implementaciones de ciertos pasos.

**Solución:** Crear un método plantilla en la clase base que llame a métodos que las subclases pueden implementar.

**Estructura:**

* **AbstractClass**: define el método plantilla.
* **ConcreteClass**: implementa los pasos específicos.

**Aplicabilidad:** Útil en algoritmos que tienen pasos fijos pero permiten variaciones en algunos.

**Ejemplo de Código:**



**Patrón Visitor**

**Propósito:** Permitir añadir nuevas operaciones a objetos sin modificar su estructura.

**Problema:** Necesidad de realizar operaciones sobre objetos de diferentes clases.

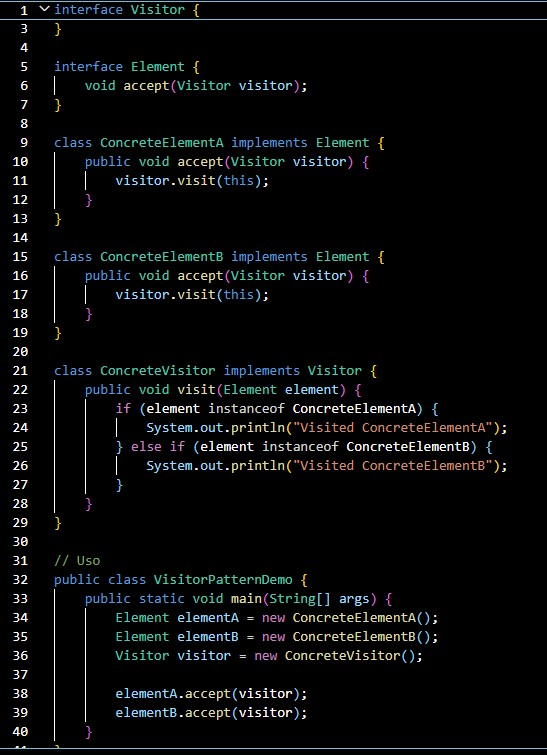
**Solución:** Crear un visitante que implementa las operaciones.

**Estructura:**

* **Visitor**: interfaz que define un método para cada tipo de elemento.
* **ConcreteVisitor**: implementa la lógica de las operaciones.
* **Element**: interfaz para los elementos que aceptan el visitante.
* **ConcreteElements**: implementaciones que aceptan visitantes.

**Aplicabilidad:** Útil en sistemas donde se necesitan realizar múltiples operaciones sobre una estructura de objetos.

**Ejemplo de Código:**



**Patrón MVC (Modelo – Vista - Controlador)**

**Propósito:** Separar la representación de información de la interacción del usuario.

**Problema:** Dificultades para manejar la lógica de la aplicación y la interfaz de usuario juntas.

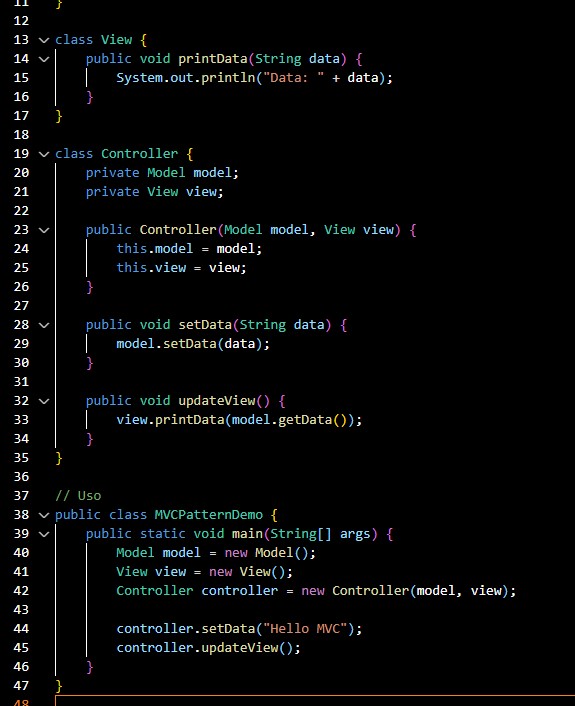
**Solución:** Dividir la aplicación en modelos, vistas y controladores.

**Estructura:**

* **Modelo**: maneja los datos y la lógica de negocio.
* **Vista**: presenta los datos al usuario.
* **Controlador**: maneja la entrada del usuario y actualiza el modelo.

**Aplicabilidad:** Útil en aplicaciones de interfaz gráfica y web.

**Ejemplo de Código:**



**Patrón DAO (Data – Access - Object)**

**Propósito:** Proveer una abstracción de acceso a datos, desacoplando la lógica de negocio de la lógica de persistencia.

**Problema:** Necesidad de cambiar la lógica de acceso a datos sin afectar la lógica de negocio.

**Solución:** Usar un objeto DAO para encapsular las operaciones de acceso a datos.

**Estructura:**

* **DAO**: interfaz para las operaciones CRUD.
* **ConcreteDAO**: implementación específica de acceso a datos.
* **Model**: objeto que representa la entidad.

**Aplicabilidad:** Útil en aplicaciones que requieren operaciones de persistencia.

**Ejemplo de Código:**

