

# Patrones de diseño

Juan Diego Solís Martínez - 23720

Nils Muralles Morales - 23727

Víctor Manuel Pérez Chávez - 23731

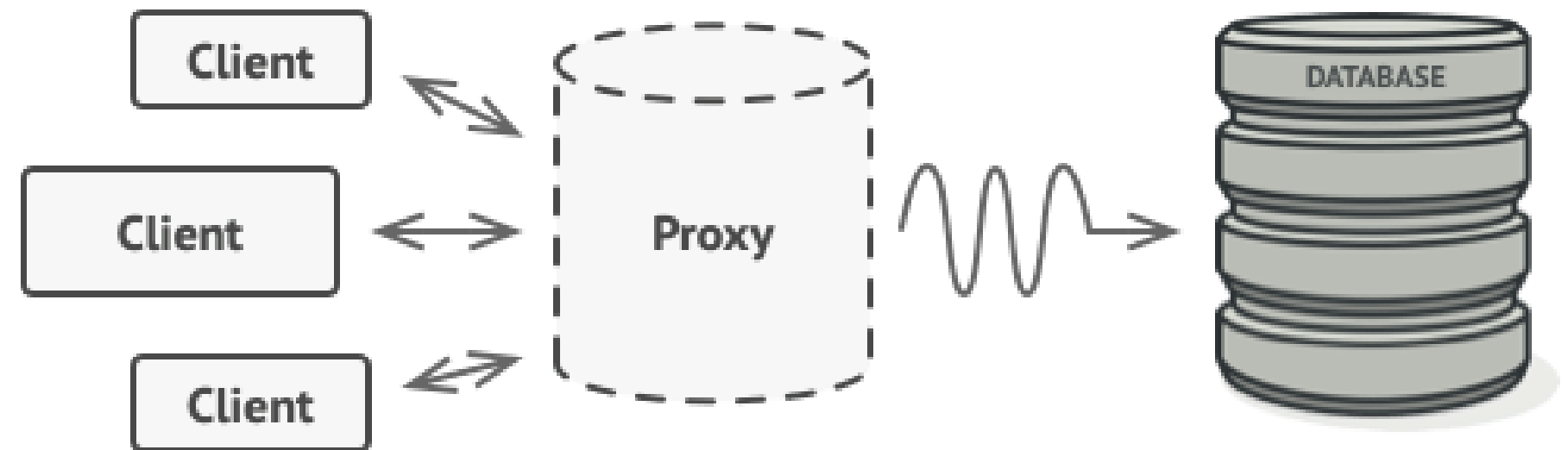
Diego Oswaldo Flores Rivas - 23714

Isabella Recinos Rodríguez- 23003

# Proxy

# Intención

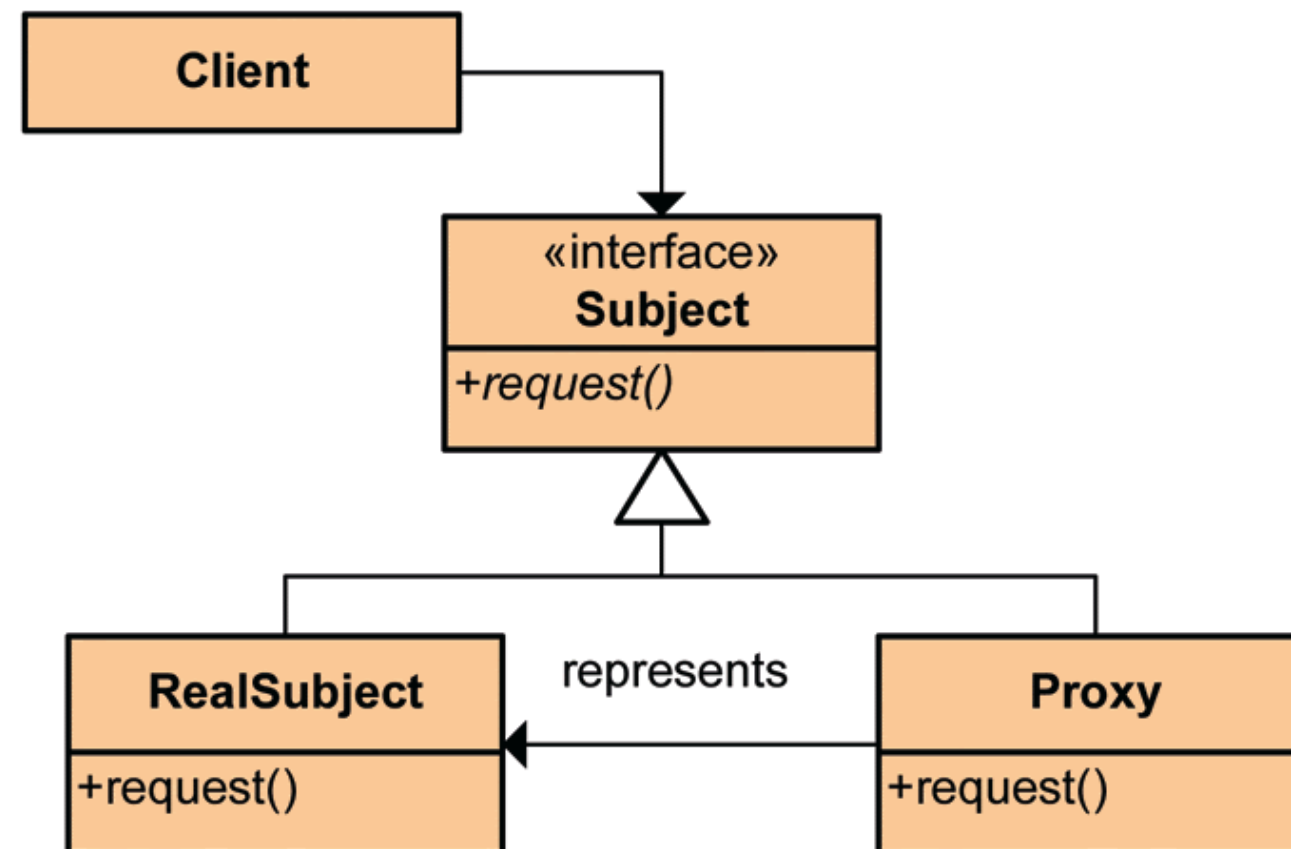
Proveer un servicio intermediario que controla el acceso al objeto en sí, el cual permite agregar lógica de programación en función de las necesidades del programa.



# Motivo

## Proxy

Type: Structural



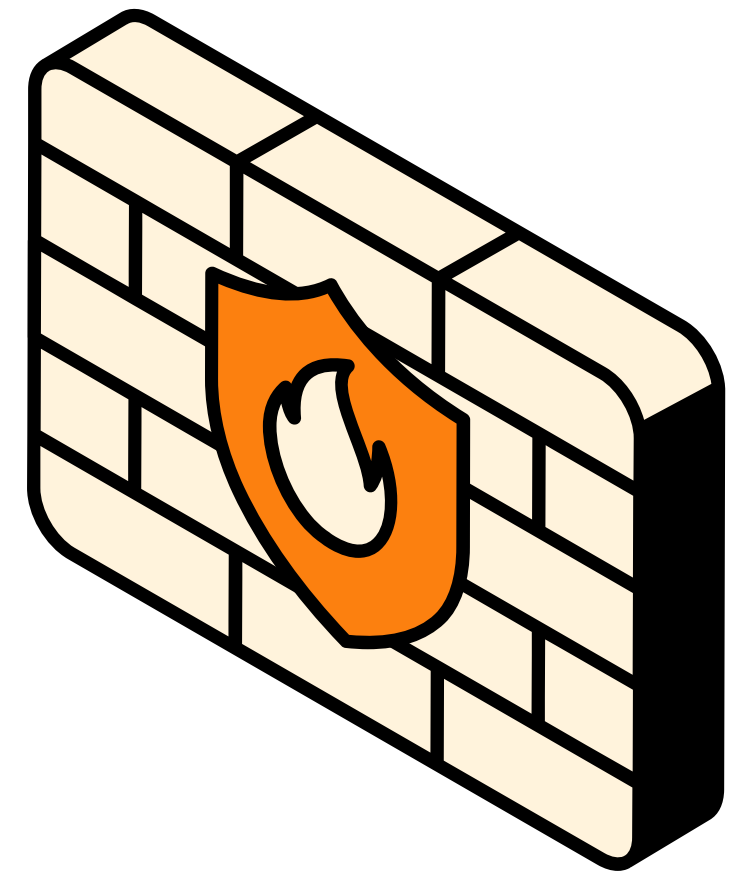
### What it is:

Provide a surrogate or placeholder for another object to control access to it

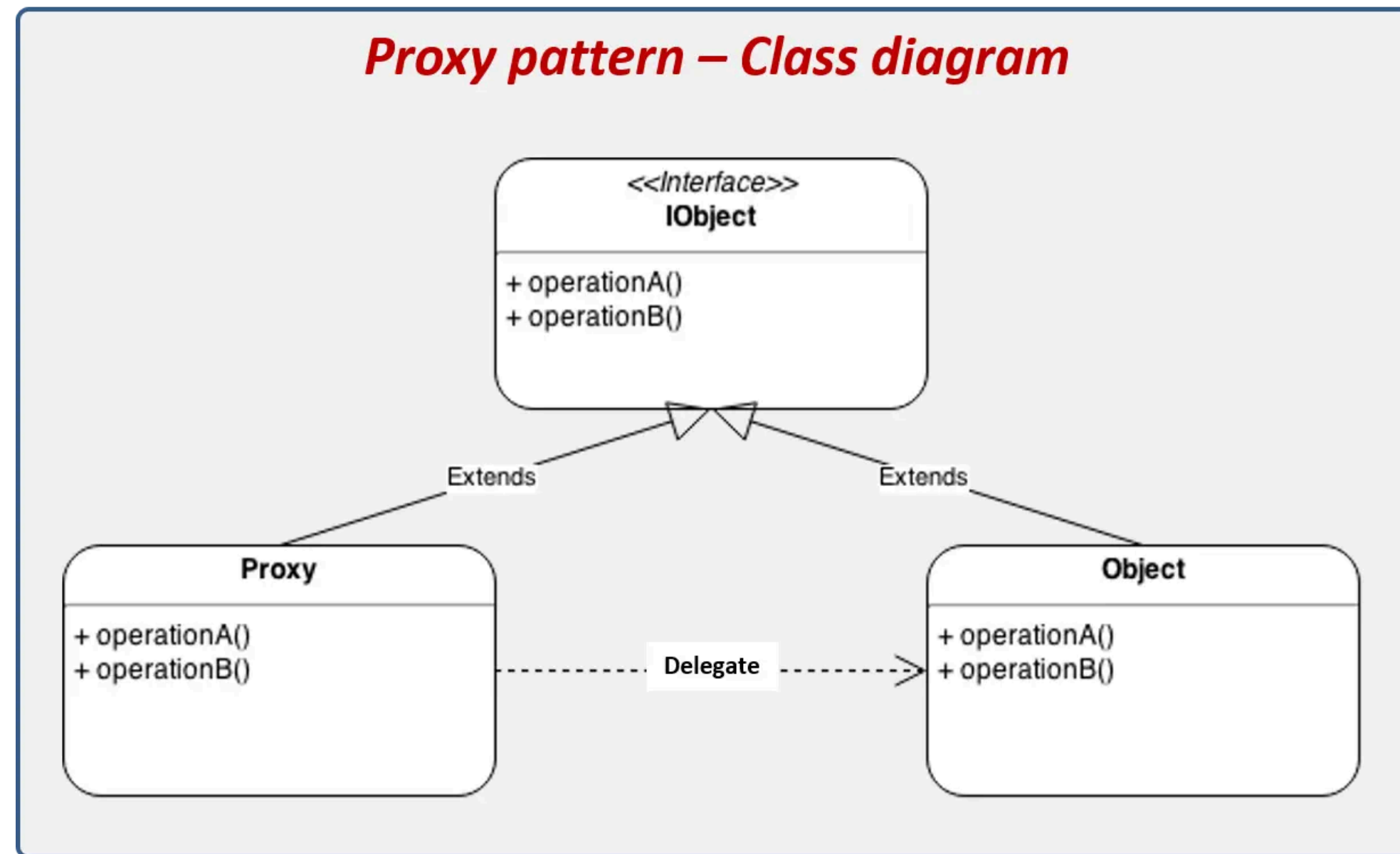
- Guía para el control de acceso a objetos y clases.
- Mantenibilidad y escalabilidad
- Cumple con el principio de responsabilidad única.

# Aplicaciones

1. **Proxy de autenticación:** Verificar los datos de los usuarios antes de permitir el acceso.
2. **Proxy de optimización:** Permitir el acceso a los objetos únicamente cuando es necesario.
3. **Proxy remoto:** Implementación del proxy en una máquina externa.



# Estructura



# Participantes

1. **IObject**: Interfaz que define las funcionalidades que tanto el proxy como el objeto implementarán.
2. **Object**: Proporciona la implementación de la interfaz IObject y realiza las tareas de manera predeterminada.
3. **Proxy**: También implementa la interfaz IObject, pero a su vez delega las tareas y controla el acceso al objeto Object en sí. Simplemente, hace referencia al mismo y añade lógica por encima.

# Colaboraciones

## **IObject (Interfaz Común)**

1. Define las funcionalidades que tanto el Proxy como el Object deben implementar.

## **Object (Objeto Real)**

1. Proporciona la implementación concreta de la interfaz IObject.
2. Ejecuta las tareas de manera predeterminada.

## **Proxy**

1. Implementa la interfaz IObject, actuando como intermediario entre el cliente y el objeto real.
2. Puede realizar tareas adicionales antes y después de llamar al Object.



# Consecuencias

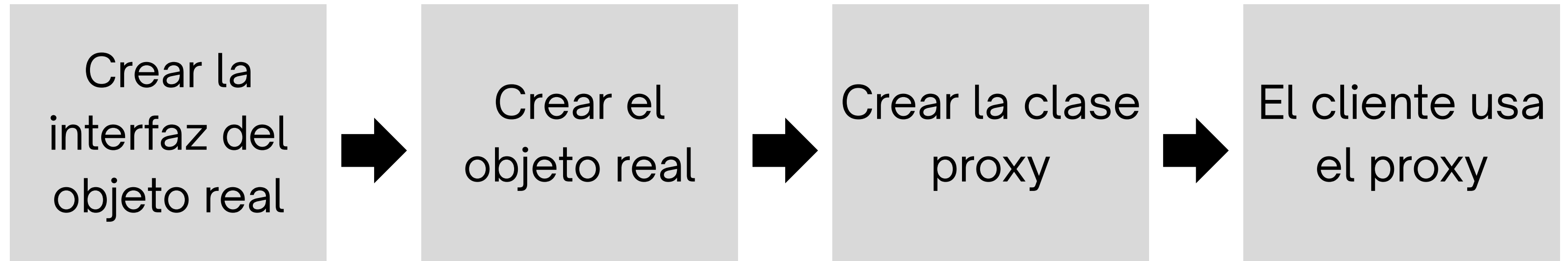
## Beneficios

- 1.Carga diferida
- 2.Control de acceso y seguridad
- 3.Caché y optimización de rendimiento.
- 4.Registro y monitoreo de acciones.

## Riesgos y posibles inconvenientes

- 1.Sobrecarga en operaciones simples
- 2.Complejidad innecesaria en el diseño
- 3.Impacto negativo en el rendimiento

# Implementación



# Código de ejemplo

```
1 // Interfaz IObject (Subject)
2 interface IObject {
3     void operationA();
4     void operationB();
5 }
```

```
7 // Clase Object (Real Subject)
8 class Object implements IObject {
9     private String name;
10
11     public Object(String name) {
12         this.name = name;
13         loadFromDisk();
14     }
15
16     private void loadFromDisk() {
17         System.out.println("Cargando objeto: " + name);
18     }
19
20     public void operationA() {
21         System.out.println("Ejecutando operación A en: " + name);
22     }
23
24     public void operationB() {
25         System.out.println("Ejecutando operación B en: " + name);
26     }
27 }
```

# Código de ejemplo

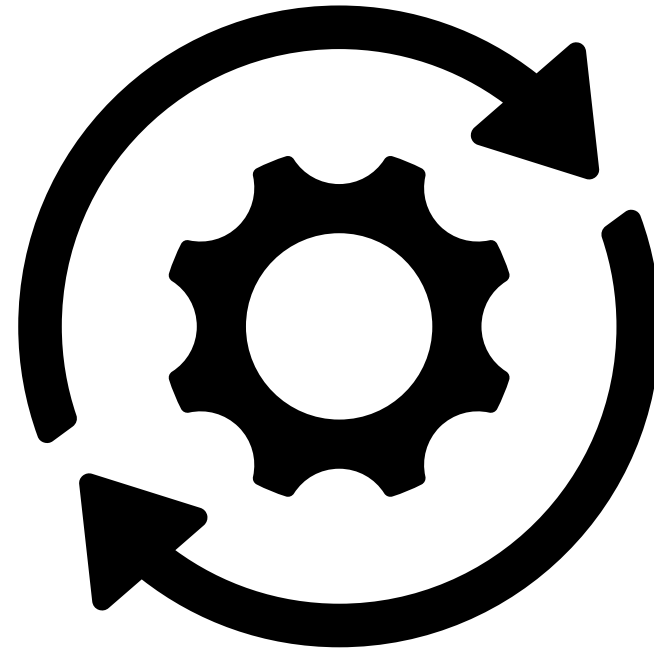
```
29 // Clase Proxy (Proxy)
30 class Proxy implements IObject {
31     private Object realObject;
32     private String name;
33
34     public Proxy(String name) {
35         this.name = name;
36     }
37
38     public void operationA() {
39         if (realObject == null) {
40             System.out.println("Creando instancia de Object en Proxy...");
41             realObject = new Object(name);
42         } else {
43             System.out.println("Usando objeto en caché en Proxy...");
44         }
45         realObject.operationA();
46     }
47
48     public void operationB() {
49         if (realObject == null) {
50             System.out.println("Creando instancia de Object en Proxy...");
51             realObject = new Object(name);
52         } else {
53             System.out.println("Usando objeto en caché en Proxy...");
54         }
55         realObject.operationB();
56     }
57 }
```

```
59 // Cliente
60 public class ProxyPatternExample {
61     public static void main(String[] args) {
62         IObject proxy = new Proxy("Ejemplo");
63
64         System.out.println("Primera ejecución de operationA():");
65         proxy.operationA(); // Crea y ejecuta
66
67         System.out.println("\nSegunda ejecución de operationA():");
68         proxy.operationA(); // Usa caché
69
70         System.out.println("\nEjecución de operationB():");
71         proxy.operationB(); // Usa caché
72     }
73 }
```

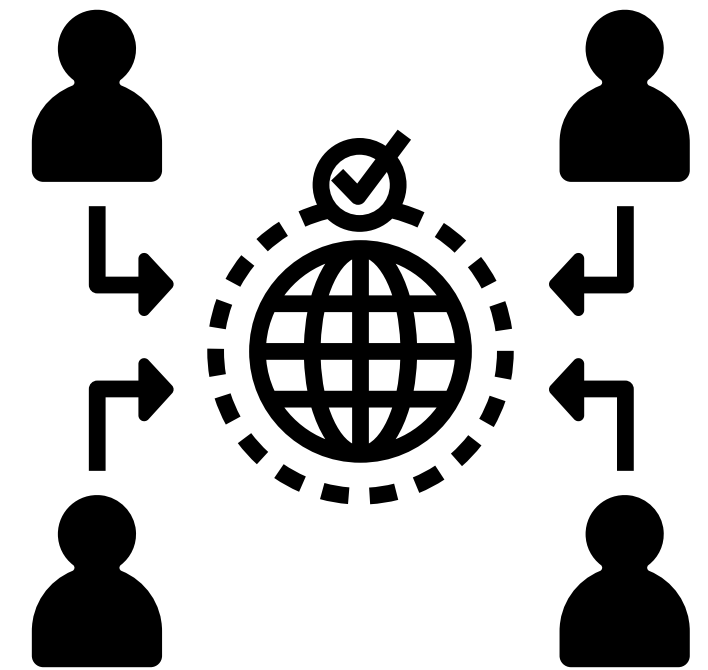
# Usos conocidos



Sistemas de  
seguridad

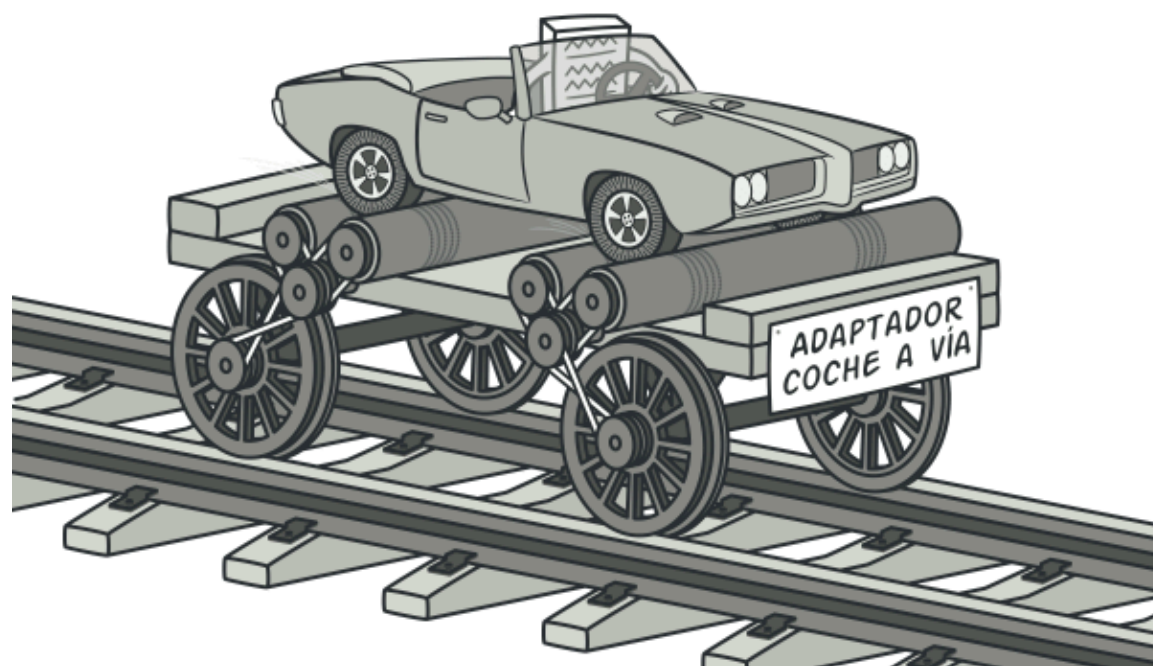


Optimización de  
recursos

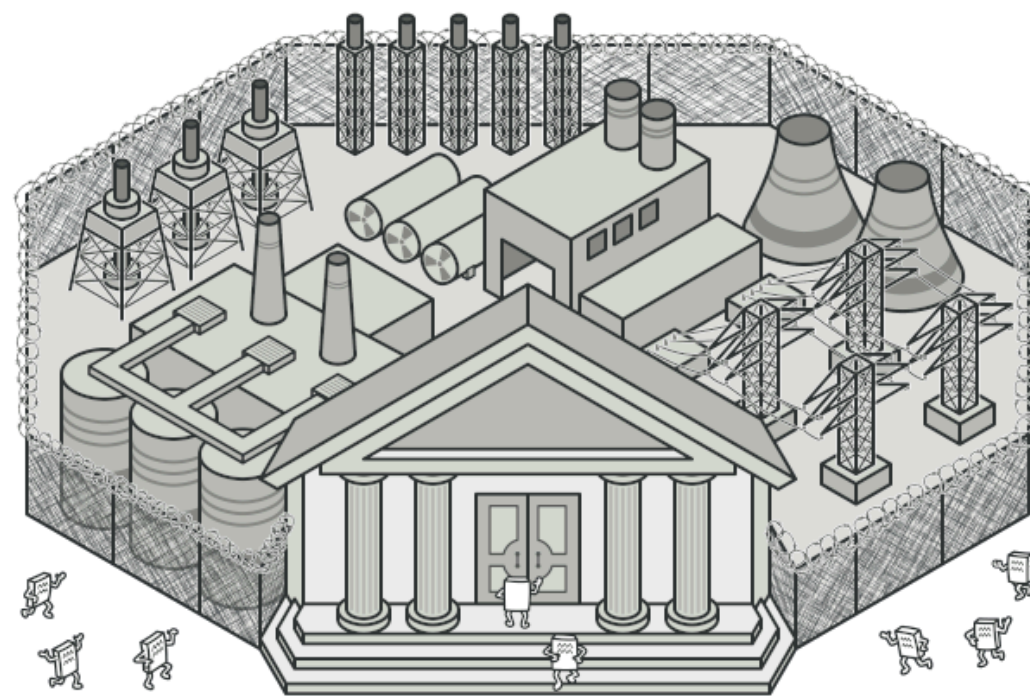


Control de acceso  
en redes

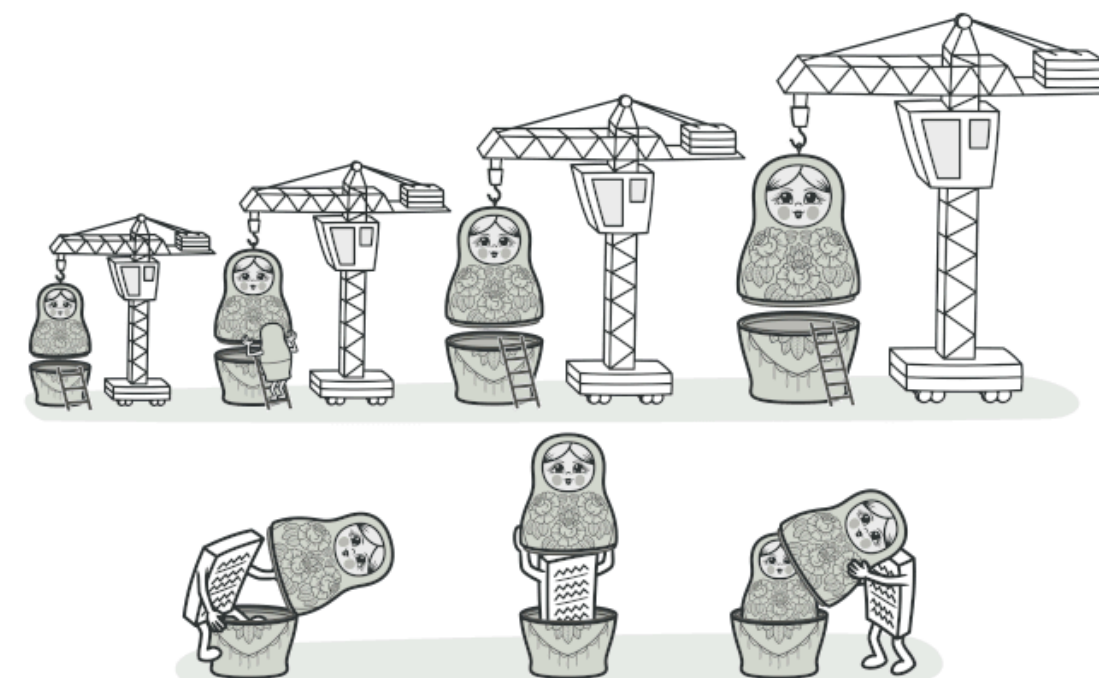
# Patrones relacionados



Adapter



Facade



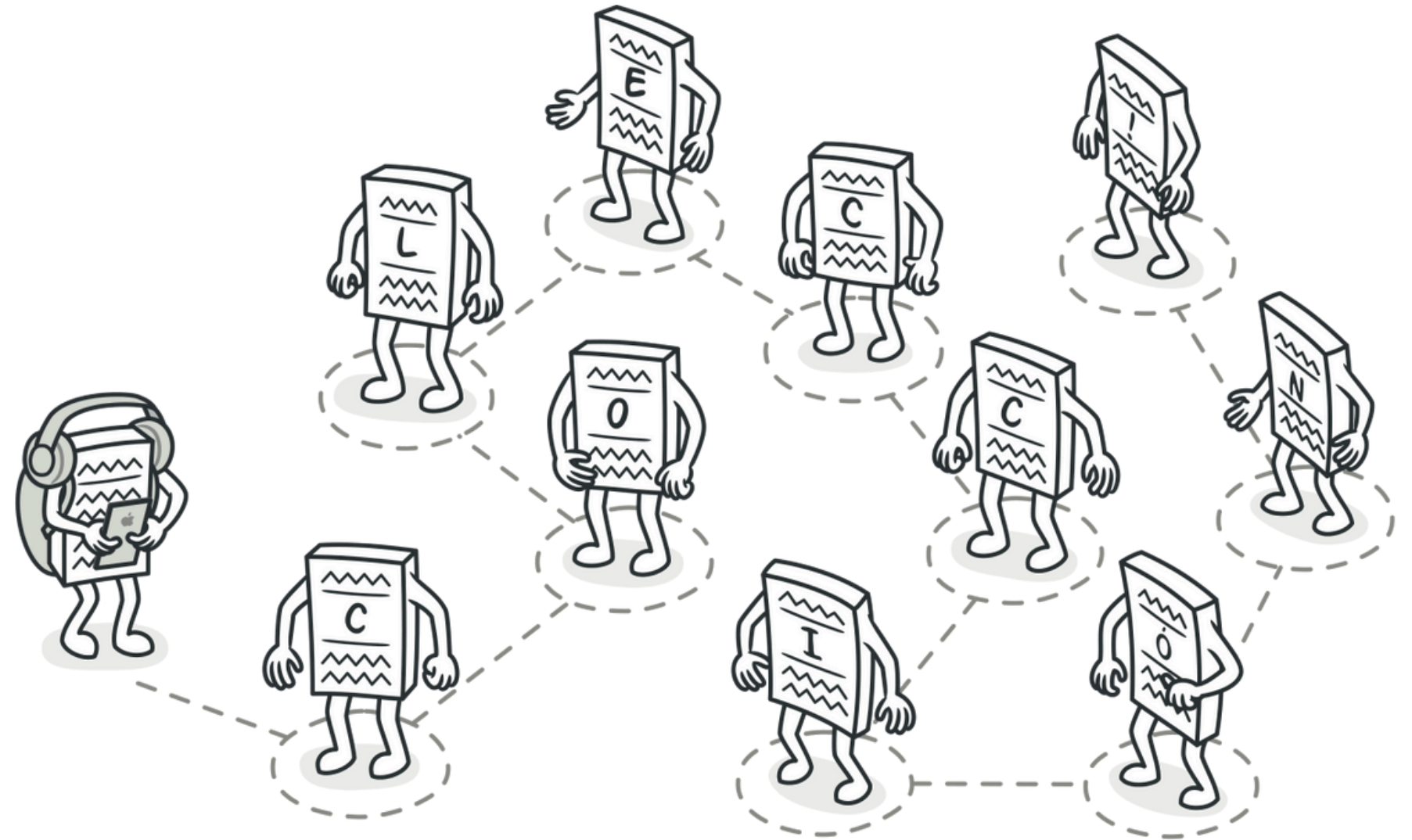
Decorator

# Iterator



# Intención

Permite recorrer los elementos de una colección de manera secuencial sin exponer su estructura interna.

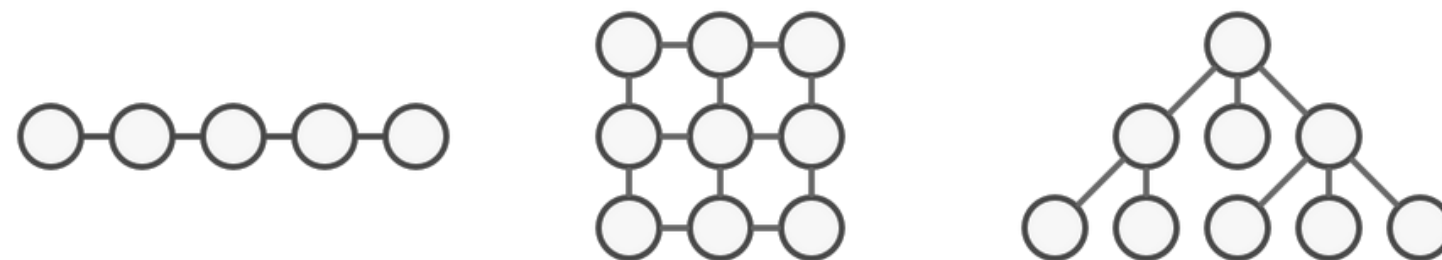




# Motivos

- Resuelve el recorrido de estructuras de datos sin exponer su implementación interna.
- Encapsula la lógica de iteración, permitiendo un acceso uniforme a diferentes colecciones.

- Ayuda a separar la estructura de los datos de la forma en que se recorren.
- Mejora la reutilización y el mantenimiento del código.

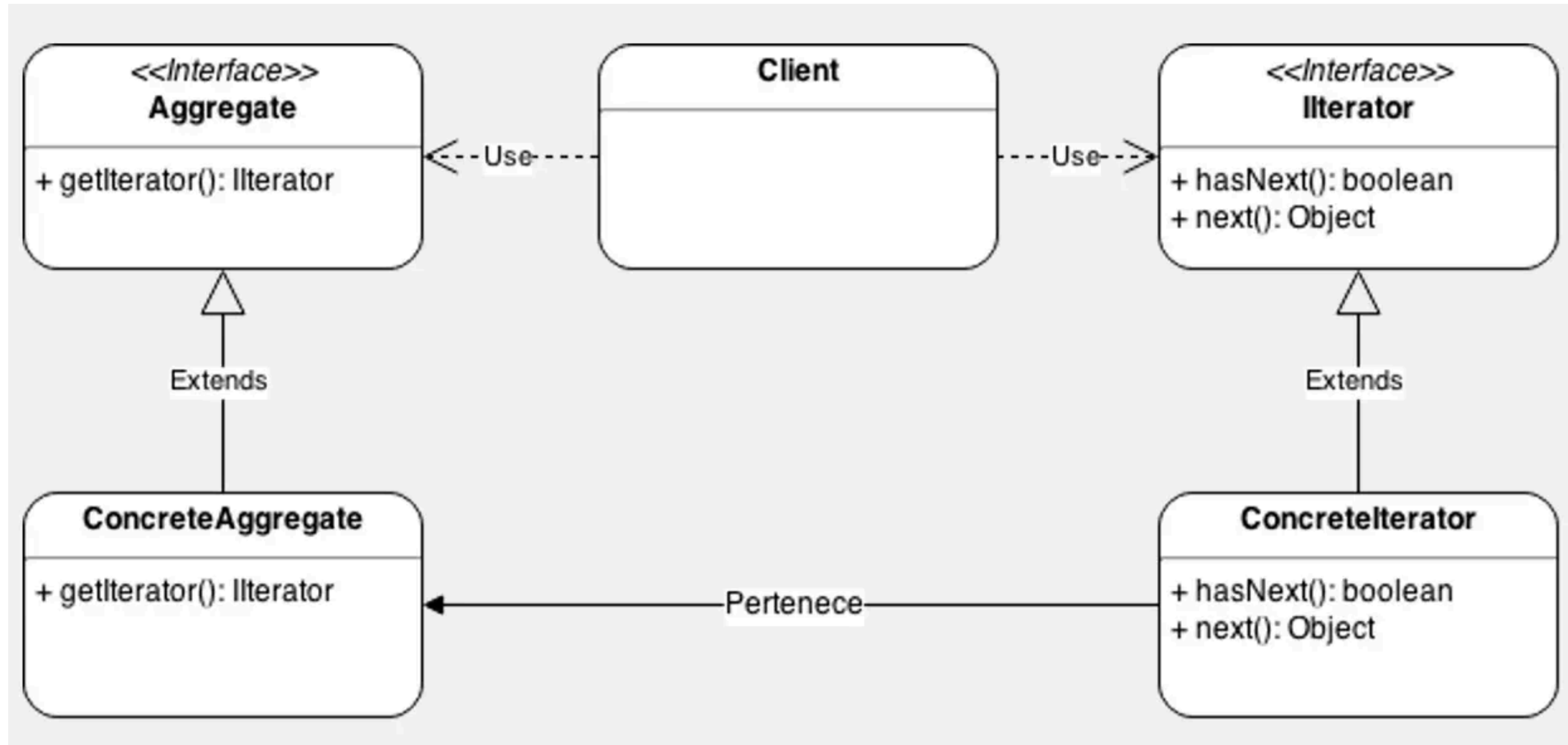


# Aplicaciones

1. **Bibliotecas de colecciones:** como Iterator en Java, foreach en C#, e iter() en Python.
2. **Bases de datos:** para recorrer registros sin cargar todo en memoria.
3. **Frameworks de UI:** para recorrer elementos de una interfaz.
4. **Procesamiento de archivos:** para leer líneas sin cargar el archivo completo.
5. **Inteligencia Artificial:** para recorrer árboles de decisión o grafos.



# Estructura



# Participantes

## **Client**

Actor que utiliza al Iterator para recorrer una colección.

## **Aggregate**

Es la interfaz que define las estructuras que pueden ser recorridas.

## **ConcreteAggregate**

Es la clase que contiene la estructura de datos que se desea recorrer.

## **Iterator**

Es la interfaz que define los métodos necesarios para recorrer una colección

## **ConcreteIterator**

Es la implementación específica del iterador, que sabe cómo recorrer una ConcreteAggregate.

# Ejemplo

```
1 // Colección concreta sobre la cual se itera
2 class Biblioteca implements Iterable<Libro> {
3     private List<Libro> libros = new ArrayList<>();
4
5     public void agregarLibro(Libro libro) {
6         libros.add(libro);
7     }
8
9     // Conexión con su iterador
10    @Override
11    public Iterator<Libro> iterator() {
12        return new LibroIterator(libros);
13    }
14 }
```

```
1 // Iterador personalizado
2 class LibroIterator implements Iterator<Libro> {
3     private List<Libro> libros;
4     private int indice = 0;
5
6     public LibroIterator(List<Libro> libros) {
7         this.libros = libros;
8     }
9
10    @Override
11    public boolean hasNext() {
12        return indice < libros.size();
13    }
14
15    @Override
16    public Libro next() {
17        return libros.get(indice++);
18    }
19 }
```

# Ejemplo



```
1  public class Main {  
2      public static void main(String[] args) {  
3          Biblioteca biblioteca = new Biblioteca();  
4          biblioteca.agregarLibro(new Libro("El Hobbit"));  
5          biblioteca.agregarLibro(new Libro("1984"));  
6          biblioteca.agregarLibro(new Libro("Cien años de soledad"));  
7  
8          // Utilización del patrón  
9          for (Libro libro : biblioteca) {  
10             System.out.println(libro.getTitulo());  
11         }  
12     }  
13 }
```

# Patrones relacionados

## Factory

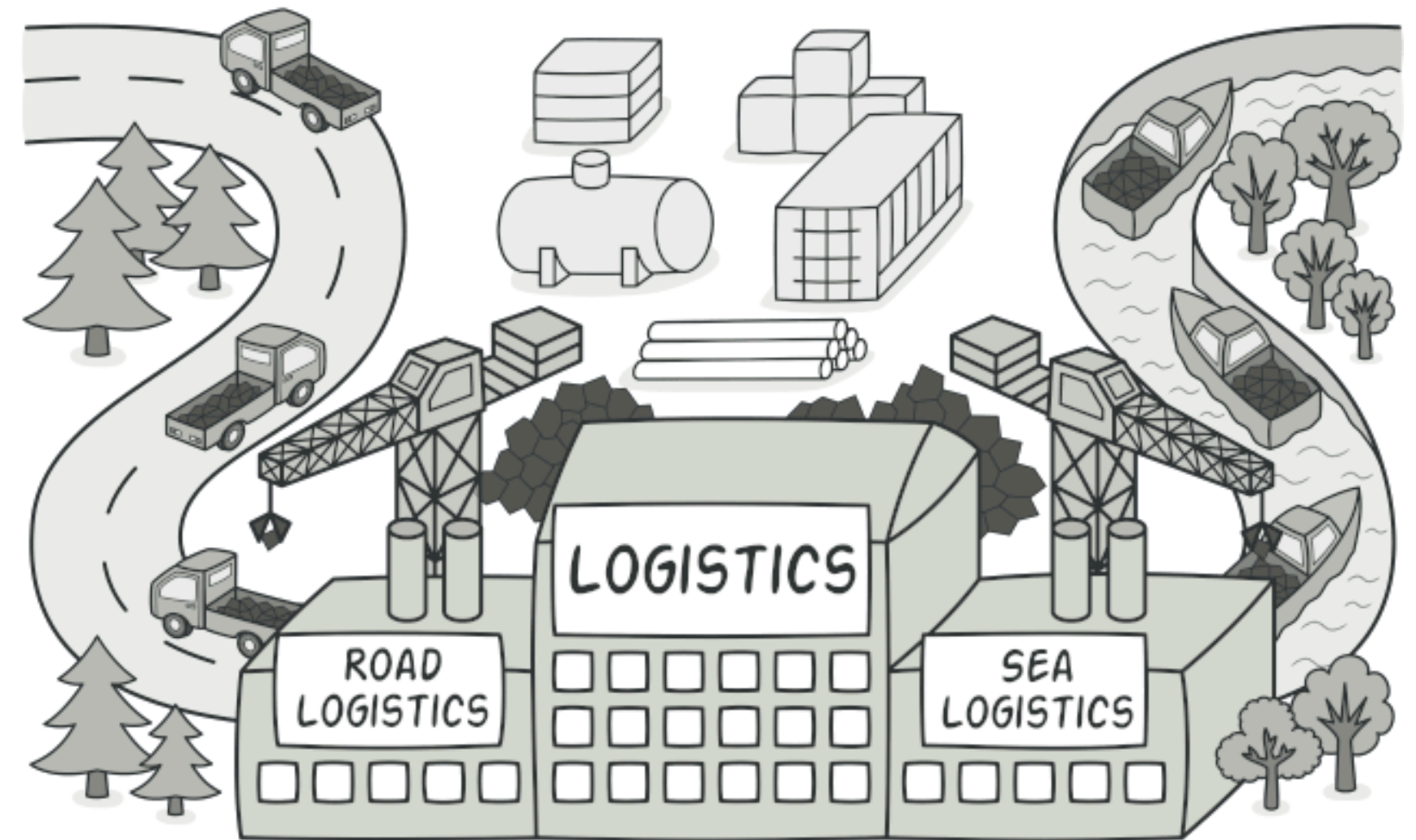
Intercambio de las implementaciones a utilizar

## Memento

Snapshots de un momento concreto de la iteración

## Visitor

Operar sobre cada elemento de la iteración

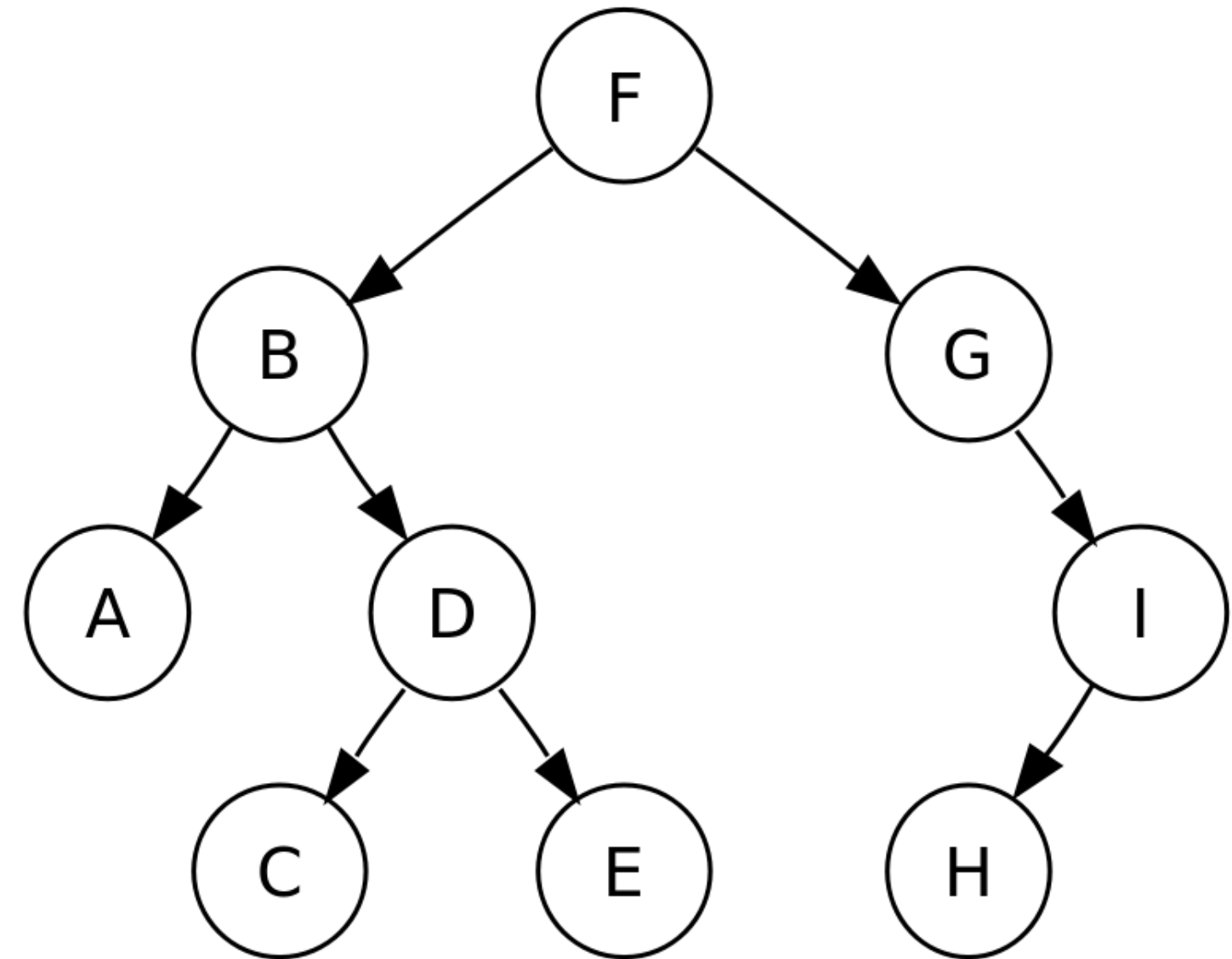


# Strategy



# Intención

Establecer una familia de algoritmos, haciendo que sean intercambiables y encapsulables cada uno de ellos. Esto logra que el algoritmo sea independiente del cliente que lo usa.



# Motivos

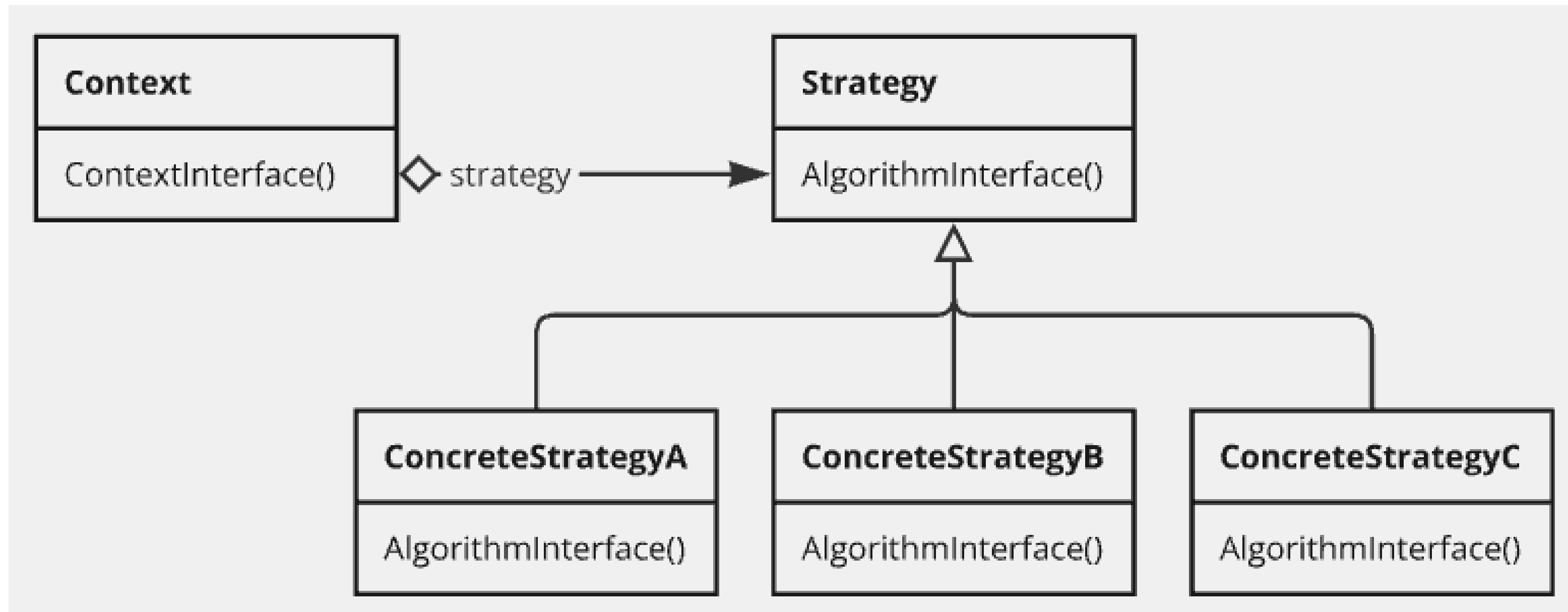
- Código menos rígido, altamente escalable y que principalmente pueda cambiar el comportamiento de un componente o módulo en tiempo real.
- Uso de clases separadas para la implementación de los algoritmos que se desean usar.



# Aplicaciones

1. Utilizar diferentes variantes de un algoritmo sin la necesidad de recurrir a estructuras condicionales y se desee cambiar su comportamiento en tiempo de ejecución.
2. Clases similares entre sí, que solo se diferencian en la forma que ejecutan algún comportamiento.
3. Cuando un algoritmo usa información o "data" que el cliente no debería saber.
4. Código que tenga una cantidad masiva de estructuras condicionales, ya que la encapsulación de algoritmos en clases reduce su uso.

# Estructura



# Participantes

## Strategy

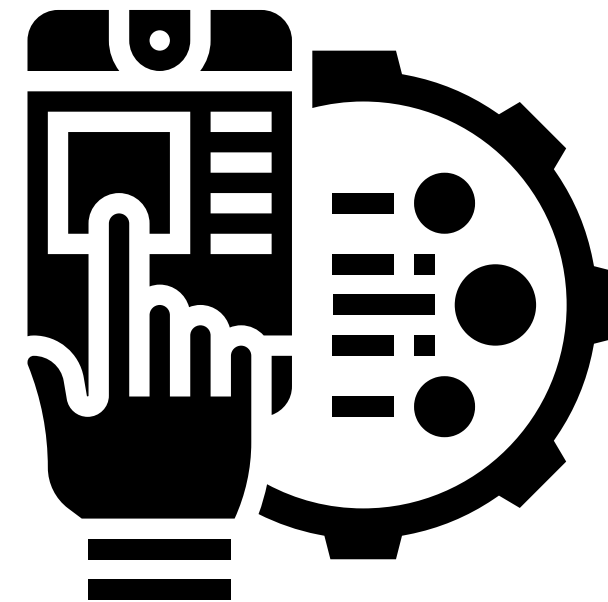
Es la interfaz que contiene al algoritmo que se desea tenga un diferente comportamiento.

## ConcreteStrategy

Es la clase o clases que pueden implementar el algoritmo definido en Strategy.

## Context

Es la clase que utiliza las implementaciones del algoritmo definido en Strategy y lleva a cabo el polimorfismo.



# Ejemplo

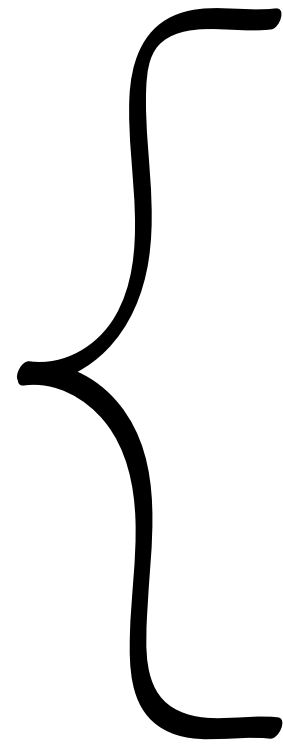
## Strategy



```
1 public interface DiscountStrategy {  
2     double applyDiscount(double price); //Algoritmo con comportamiento variable  
3 }
```

# Ejemplo

ConcreteStrategy




```
1 // ConcreteStrategyA: Sin descuento (clientes normales)
2 public class ConcreteStrategyNoDiscount implements DiscountStrategy {
3     @Override
4     public double applyDiscount(double price) {
5         return price;
6     }
7 }
```

```
1 // ConcreteStrategyB: Descuento del 30% para clientes VIP
2 public class ConcreteStrategyVipDiscount implements DiscountStrategy{
3     @Override
4     public double applyDiscount(double price) {
5         return price * 0.7;
6     }
7 }
```

# Ejemplo

## Context



```
1  public class ContextShoppingCart {
2      private DiscountStrategy discountStrategy;
3
4      public ContextShoppingCart(DiscountStrategy discountStrategy) {
5          this.discountStrategy = discountStrategy;
6      }
7
8      public void setDiscountStrategy(DiscountStrategy discountStrategy) {
9          this.discountStrategy = discountStrategy;
10     }
11
12     public double calculateTotal(double price) {
13         return discountStrategy.applyDiscount(price);
14     }
15 }
```



# Ejemplo

## Main

```
1 public class Main {  
2     public static void main(String[] args) {  
3         ContextShoppingCart cart = new ContextShoppingCart(new ConcreteStrategyNoDiscount());  
4  
5         double price = 100.0;  
6  
7         System.out.println("Precio sin descuento: " + cart.calculateTotal(price));  
8  
9         cart.setDiscountStrategy(new ConcreteStrategyVipDiscount()); // Se aplica descuento VIP  
10        System.out.println("Precio con descuento VIP: " + cart.calculateTotal(price));  
11  
12        cart.setDiscountStrategy(new ConcreteStrategySeasonalDiscount()); // Se aplica descuento por temporada  
13        System.out.println("Precio con descuento por temporada: " + cart.calculateTotal(price));  
14    }  
15 }
```

# Patrones relacionados

## Template

Utiliza herencia para permitir la personalización de partes específicas del algoritmo dentro de la estructura.

## State

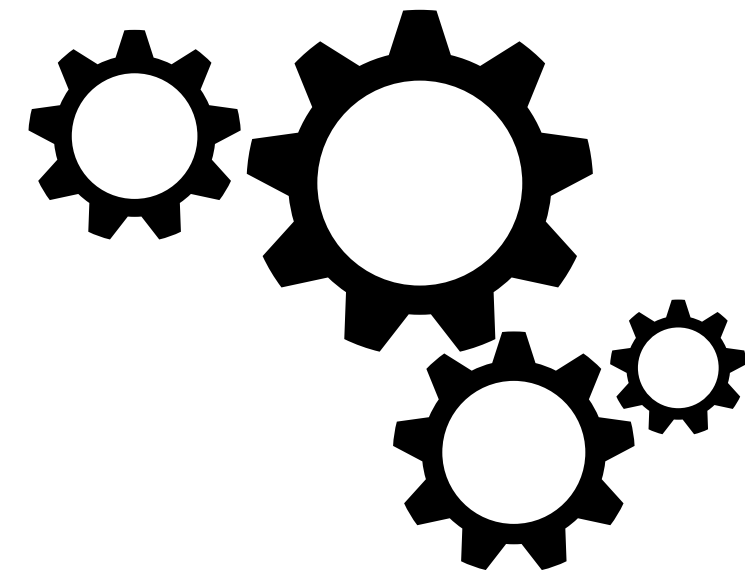
Se centra en cambiar el comportamiento de un objeto en función de su estado interno

## Command

Command encapsula acciones, permitiendo operaciones como comandos.

## Decorator

Añade responsabilidades adicionales a un objeto, mientras que Strategy cambia el comportamiento completo al seleccionar diferentes algoritmos.



**GRACIAS**

**POR VER**