.UBAfiuba
FACULTAD DE INGENIERÍA

Algoritmos y Programación III

Minicatálogo de
Patrones de Diseño

Cátedra Corsi - Essaya - Maraggi

1. Singleton (Único) — Patrón de creación (ámbito: objeto)

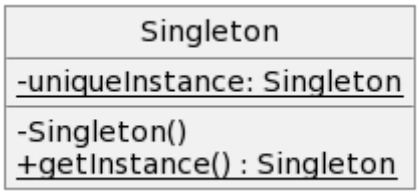
Propósito

Garantiza que una clase solo tenga una instancia, y proporciona un punto de acceso global a ella.

Aplicabilidad

Cuando deba haber exactamente una instancia de una clase, y esta deba ser accesible a los clientes desde un punto de acceso conocido. Por ejemplo, en el caso de requerirse una única cola de impresión para las múltiples impresoras de un sistema, esta podría seguir el patrón Singleton.

Estructura



Ejemplo

```java
package algo3;

public class ColaDeImpresion {

    private static ColaDeImpresion uniqueInstance = null;

    // Aqui irian otros atributos

    private ColaDeImpresion() {
    }

    public static ColaDeImpresion getInstance() {
        if (uniqueInstance == null) {
             uniqueInstance = new ColaDeImpresion();
        }
        return uniqueInstance;
    }

    // Aqui irian otros metodos
}
```

```java
package algo3;

public class Main {

  public static void main(String[] args) {

    Object o1 = new Object();
    Object o2 = new Object();
    Object o3 = new Object();
```

```
    System.out.println("Hashcode de o1: " + o1.hashCode());
    System.out.println("Hashcode de o2: " + o2.hashCode());
    System.out.println("Hashcode de o3: " + o3.hashCode());

    if (o1 == o2 && o2 == o3) {
      System.out.println("Las variables o1, o2 y o3 se refieren al mismo objeto.");
    } else {
      System.out.println("Las variables o1, o2 y o3 NO se refieren al mismo objeto.");
    }

    System.out.println();

    ColaDeImpresion s1 = ColaDeImpresion.getInstance();
    ColaDeImpresion s2 = ColaDeImpresion.getInstance();
    ColaDeImpresion s3 = ColaDeImpresion.getInstance();

    System.out.println("Hashcode de s1: " + s1.hashCode());
    System.out.println("Hashcode de s2: " + s2.hashCode());
    System.out.println("Hashcode de s3: " + s3.hashCode());

    if (s1 == s2 && s2 == s3) {
      System.out.println("Las variables s1, s2 y s3 se refieren al mismo objeto.");
    } else {
      System.out.println("Las variables s1, s2 y s3 NO se refieren al mismo objeto.");
    }
  }
}
```

Salida:

```
Hashcode de o1: 1365202186
Hashcode de o2: 212628335
Hashcode de o3: 1579572132
Las variables o1, o2 y o3 NO se refieren al mismo objeto.

Hashcode de s1: 305808283
Hashcode de s2: 305808283
Hashcode de s3: 305808283
Las variables s1, s2 y s3 se refieren al mismo objeto.
```

Usos conocidos

Muchas clases básicas de Java siguen este patrón, por ejemplo `java.lang.Runtime`: https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Runtime.html

Consecuencias

Efectos deseados: Acceso controlado a la única instancia. El patrón es una mejora sobre las variables globales. Se puede crear una subclase de la clase `Singleton`. El patrón hace que sea fácil cambiar de opinión y permitir más de una instancia de la clase `Singleton`. El patrón es más flexible que los métodos estáticos (de clase).

Efectos no deseados: El patrón viola el *Principio de Responsabilidad Única*. El patrón requiere de un tratamiento especial en un entorno con múltiples hilos de ejecución. Puede resultar complicado realizar pruebas unitarias del código cliente de la clase `Singleton`.

2. Factory Method (Método de Fabricación) — Patrón de creación (ámbito: clase)

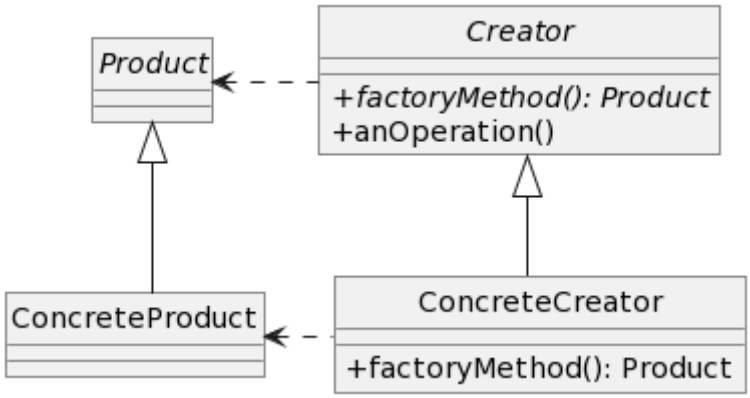
Propósito

Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.

Aplicabilidad

Cuando desde una clase se desee crear y usar objetos sin que la misma quede acoplada a las clases de estos. Cuando una clase no pueda anticipar la clase de objetos que debe crear. Cuando una clase quiera que sean sus subclases las que especifiquen los objetos que crea. Por ejemplo, en una aplicación de mensajería, la clase que genere los avisos puede ser abstracta y delegar a sus subclases concretas la creación de los distintos tipos de avisos. Así, se podrá utilizar estas clases desde otra eliminando el acoplamiento de esta con las clases de los avisos concretos.

Estructura



Ejemplo

```java
package algo3;

public abstract class Aviso {

  public abstract void avisar();
}
```

```java
package algo3;

public class LlamadaDeLinea extends Aviso {

  @Override
  public void avisar() {
    System.out.println("Se hace una llamada de linea.");
  }
}
```

```java
package algo3;

public class LlamadaDeWhatsapp extends Aviso {

  @Override
  public void avisar() {
    System.out.println("Se hace una llamada de Whatsapp.");
  }
}
```

```java
package algo3;

public class CartaDocumento extends Aviso {

  @Override
  public void avisar() {
    System.out.println("Se envia una carta documento.");
  }
}
```

```java
package algo3;

public abstract class FabricaDeAvisos {

  public abstract Aviso crearAviso(String tipoDeAviso);
}
```

```java
package algo3;

public class FabricaDeAvisosTelefonicos extends FabricaDeAvisos {

  @Override
  public Aviso crearAviso(String tipoDeAviso) {
    switch (tipoDeAviso) {
    case "Llamada de linea":
      return new LlamadaDeLinea();
    case "Llamada de Whatsapp":
      return new LlamadaDeWhatsapp();
    default:
      return null;
    }
  }
}
```

```java
package algo3;

public class FabricaDeAvisosPorCorreo extends FabricaDeAvisos {

  @Override
  public Aviso crearAviso(String tipoDeAviso) {
    switch (tipoDeAviso) {
    case "Carta documento":
      return new CartaDocumento();
    default:
      return null;
    }
  }
}
```

```java
package algo3;

public class Main {

  public static void main(String[] args) {
    FabricaDeAvisos fabricaDeAvisos = new FabricaDeAvisosTelefonicos();
    Aviso aviso1 = fabricaDeAvisos.crearAviso("Llamada de linea");
    aviso1.avisar();
    Aviso aviso2 = fabricaDeAvisos.crearAviso("Llamada de Whatsapp");
    aviso2.avisar();
    fabricaDeAvisos = new FabricaDeAvisosPorCorreo();
    Aviso aviso3 = fabricaDeAvisos.crearAviso("Carta documento");
    aviso3.avisar();
  }
}
```

Salida:

```
Se hace una llamada de linea.
Se hace una llamada de Whatsapp.
Se envia una carta documento.
```

Usos conocidos

Muchas clases básicas de Java siguen este patrón, por ejemplo `java.text.NumberFormat`: https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/text/NumberFormat.html

Consecuencias

Efectos deseados: El patrón sigue el *Principio de Responsabilidad Única* (la creación de los objetos se concentra en las fábricas). Se disminuye el acoplamiento entre las clases.

Efectos no deseados: Aumenta la complejidad del código, al haber más clases. Un inconveniente potencial de los métodos de fabricación es que los clientes pueden tener que heredar de la clase `Creator` simplemente para crear un determinado objeto `ConcreteProduct`. La herencia está bien cuando el cliente tiene que heredar de todos modos de la clase `Creator`, pero si no es así estaríamos introduciendo una nueva vía de futuros cambios.

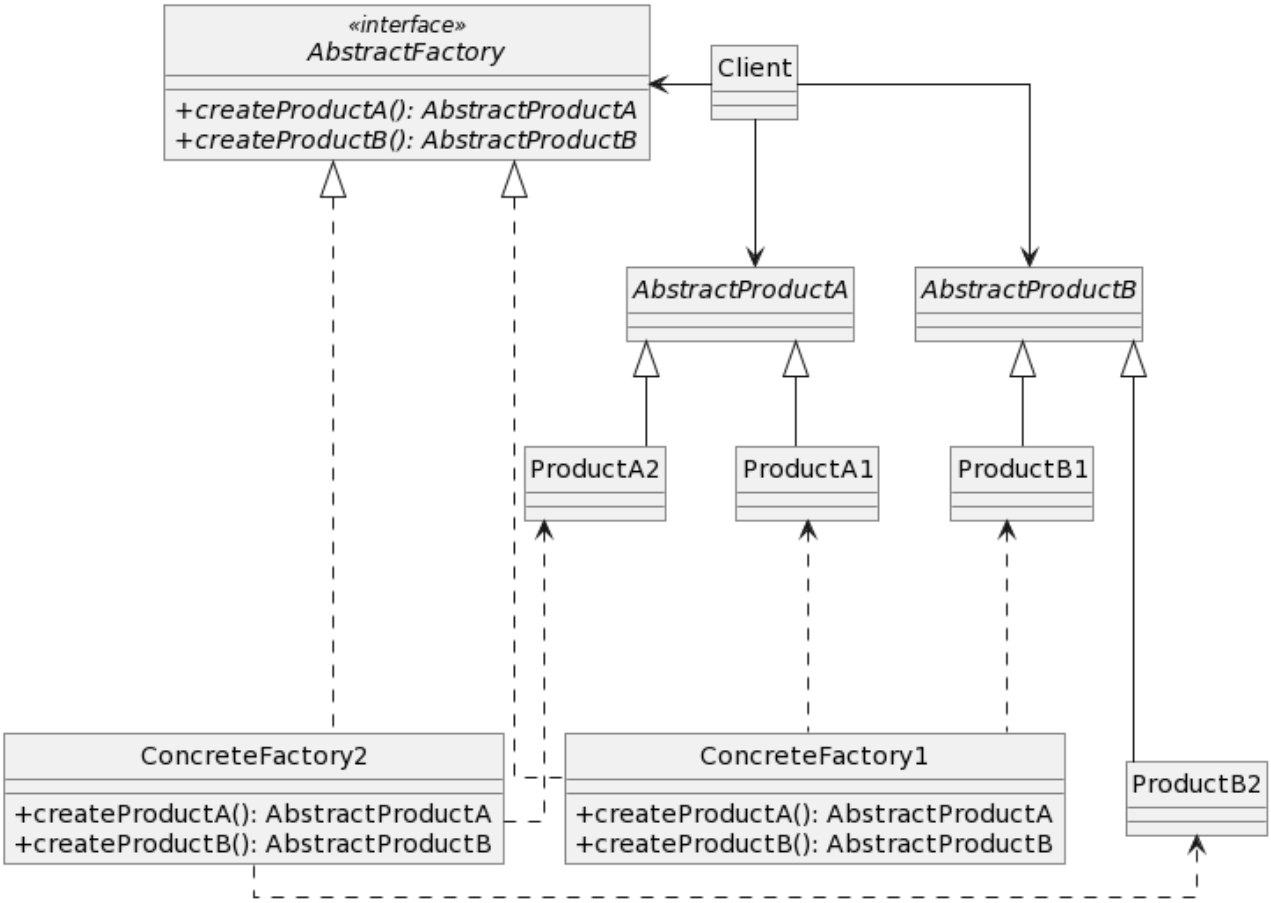
Propósito

Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas.

Aplicabilidad

Cuando un sistema deba ser independiente de cómo se crean, componen y representan sus productos. Cuando un sistema deba ser configurado con una familia de productos de entre varias. Cuando una familia de objetos producto relacionados esté diseñada para ser usada conjuntamente, y sea necesario hacer cumplir esta restricción. Cuando se quiera proporcionar una biblioteca de clases de productos, y solo se quiera revelar sus interfaces, no sus implementaciones. Por ejemplo, un catálogo de vehículos puede utilizar fábricas concretas para instanciar los distintos tipos de vehículos (autos, motos) que pertenezcan a diversas familias (nafteros, eléctricos, etc.).

Estructura



Ejemplo

```java
package algo3;

public abstract class Auto {
  private String modelo;
  private double potencia;
  private double capacidadDelBaul;

  public Auto(String modelo, double potencia, double capacidadDelBaul) {
    this.modelo = modelo;
    this.potencia = potencia;
    this.capacidadDelBaul = capacidadDelBaul;
  }

  public String getModelo() {
    return modelo;
  }

  public double getPotencia() {
    return potencia;
  }

  public double getCapacidadDelBaul() {
    return capacidadDelBaul;
  }

  public abstract void mostrarCaracteristicas();
}
```

```java
package algo3;

public class AutoNaftero extends Auto {

  public AutoNaftero(String modelo, double potencia, double capacidadDelBaul) {
    super(modelo, potencia, capacidadDelBaul);
  }

  @Override
  public void mostrarCaracteristicas() {
    System.out.println("Auto NAFTERO modelo: " + getModelo() + ". "
        + "Potencia: " + getPotencia() + " HP. "
        + "Capacidad del baul: " + getCapacidadDelBaul() + " litros.");
  }
}
```

```java
package algo3;

public class AutoElectrico extends Auto {

  public AutoElectrico(String modelo, double potencia, double capacidadDelBaul) {
    super(modelo, potencia, capacidadDelBaul);
  }

  @Override
  public void mostrarCaracteristicas() {
    System.out.println("Auto ELECTRICO modelo: " + getModelo() + ". "
      + "Potencia: " + getPotencia() + " HP. "
      + "Capacidad del baul: " + getCapacidadDelBaul() + " litros.");
  }
}
```

```java
package algo3;

public abstract class Moto {
  private String modelo;
  private double potencia;

  public Moto(String modelo, double potencia) {
    this.modelo = modelo;
    this.potencia = potencia;
  }

  public String getModelo() {
    return modelo;
  }

  public double getPotencia() {
    return potencia;
  }

  public abstract void mostrarCaracteristicas();
}
```

```java
package algo3;

public class MotoNaftera extends Moto {

  public MotoNaftera(String modelo, double potencia) {
    super(modelo, potencia);
  }

  @Override
  public void mostrarCaracteristicas() {
    System.out.println("Moto NAFTERA modelo: " + getModelo() + ". "
    + "Potencia: " + getPotencia() + " HP.");
  }
}
```

```java
package algo3;

public class MotoElectrica extends Moto {

  public MotoElectrica(String modelo, double potencia) {
    super(modelo, potencia);
  }

  @Override
  public void mostrarCaracteristicas() {
    System.out.println("Moto ELECTRICA modelo: " + getModelo() + ". "
    + "Potencia: " + getPotencia() + " HP.");
  }
}
```

```java
package algo3;

public interface FabricaDeVehiculos {
    Auto crearAuto(String modelo, double potencia, double capacidadDelBaul);
    Moto crearMoto(String modelo, double potencia);
}
```

```java
package algo3;

public class FabricaDeVehiculosNafteros implements FabricaDeVehiculos {

  public Auto crearAuto(String modelo, double potencia, double capacidadDelBaul) {
    return new AutoNaftero(modelo, potencia, capacidadDelBaul);
  }

  public Moto crearMoto(String modelo, double potencia) {
    return new MotoNaftera(modelo, potencia);
  }
}
```

```java
package algo3;

public class FabricaDeVehiculosElectricos implements FabricaDeVehiculos {

  public Auto crearAuto(String modelo, double potencia, double capacidadDelBaul) {
    return new AutoElectrico(modelo, potencia, capacidadDelBaul);
  }

  public Moto crearMoto(String modelo, double potencia) {
    return new MotoElectrica(modelo, potencia);
  }
}
```

```java
package algo3;

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

public class Catalogo {

  private FabricaDeVehiculos fabricaDeVehiculos;
  private List<Auto> catalogoDeAutos;
  private List<Moto> catalogoDeMotos;

  public Catalogo(FabricaDeVehiculos fabricaDeVehiculos) {
    this.fabricaDeVehiculos = fabricaDeVehiculos;
    catalogoDeAutos = new ArrayList<>();
    catalogoDeMotos = new ArrayList<>();
  }

  public void agregarAuto(String modelo, double potencia, double capacidadDelBaul) {
    catalogoDeAutos.add(fabricaDeVehiculos.crearAuto(modelo, potencia, capacidadDelBaul));
  }

  public void agregarMoto(String modelo, double potencia) {
    catalogoDeMotos.add(fabricaDeVehiculos.crearMoto(modelo, potencia));
  }

  public void mostrar(String titulo) {
    System.out.println(titulo);
    for (Auto auto : catalogoDeAutos)
      auto.mostrarCaracteristicas();
    for (Moto moto : catalogoDeMotos)
      moto.mostrarCaracteristicas();
  }
}
```

```java
package algo3;

public class Main {

  public static void main(String[] args) {

    FabricaDeVehiculos fabricaDeNafteros = new FabricaDeVehiculosNafteros();
    FabricaDeVehiculos fabricaDeElectricos = new FabricaDeVehiculosElectricos();

    Catalogo catalogoDeNafteros = new Catalogo(fabricaDeNafteros);
    Catalogo catalogoDeElectricos = new Catalogo(fabricaDeElectricos);

    catalogoDeNafteros.agregarAuto("Honda Pilot", 280, 827);
    catalogoDeNafteros.agregarAuto("Toyota SW4 SRX", 204, 900);
    catalogoDeNafteros.agregarMoto("Ducati 1100 Tribute PRO 2022", 86);

    catalogoDeElectricos.agregarAuto("Coradir Tito", 6, 80);
    catalogoDeElectricos.agregarMoto("KTM SX-E 5", 6.7);
    catalogoDeElectricos.agregarMoto("Zanella E-Styler", 1.6);

    catalogoDeNafteros.mostrar("CATALOGO DE VEHICULOS NAFTEROS");
    catalogoDeElectricos.mostrar("CATALOGO DE VEHICULOS ELECTRICOS");
  }
}
```

Salida:

```
CATALOGO DE VEHICULOS NAFTEROS
Auto NAFTERO modelo: Honda Pilot. Potencia: 280.0 HP. Capacidad del baul: 827.0 litros.
Auto NAFTERO modelo: Toyota SW4 SRX. Potencia: 204.0 HP. Capacidad del baul: 900.0 litros.
Moto NAFTERA modelo: Ducati 1100 Tribute PRO 2022. Potencia: 86.0 HP.
CATALOGO DE VEHICULOS ELECTRICOS
Auto ELECTRICO modelo: Coradir Tito. Potencia: 6.0 HP. Capacidad del baul: 80.0 litros.
Moto ELECTRICA modelo: KTM SX-E 5. Potencia: 6.7 HP.
Moto ELECTRICA modelo: Zanella E-Styler. Potencia: 1.6 HP.
```

Usos conocidos

Muchas clases básicas de Java siguen este patrón, por ejemplo `javax.xml.parsers.SAXParserFactory`: https://docs.oracle.com/en/java/javase/19/docs/api/java.xml/javax/xml/parsers/SAXParserFactory.html

Consecuencias

Efectos deseados: El patrón sigue el *Principio de Responsabilidad Única* (la creación de los objetos se concentra en las fábricas) y el *Principio Abierto/Cerrado* (es posible incorporar nuevos tipos de productos en el programa sin modificar estructuras condicionales). Los productos que se obtienen de una fábrica son compatibles entre sí. Se disminuye el acoplamiento entre las clases.

Efectos no deseados: Aumenta la complejidad del código, al haber más clases e interfaces.

4. Façade (Fachada) — Patrón estructural (ámbito: objeto)

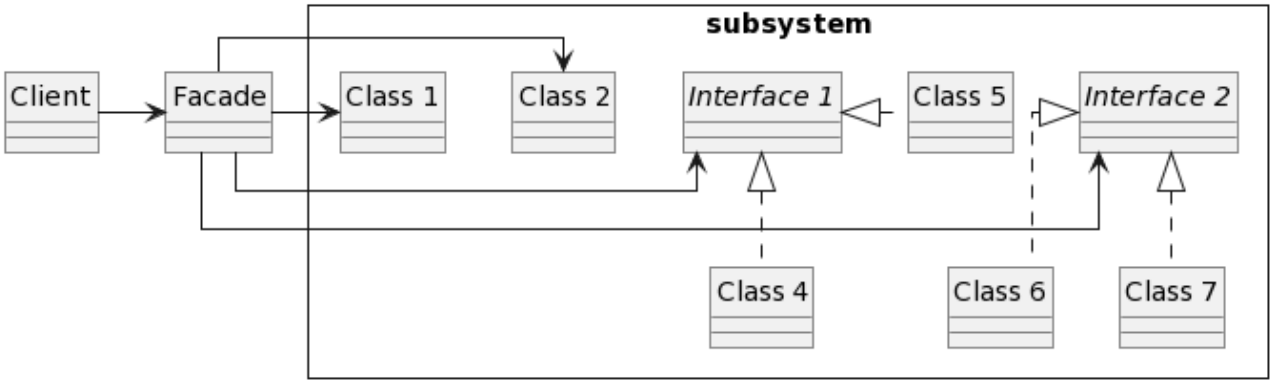
Propósito

Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.

Aplicabilidad

Cuando se desee proporcionar una interfaz simple para un subsistema complejo. Cuando haya muchas dependencias entre los clientes y las clases que implementan una abstracción. Cuando se desee dividir en capas los subsistemas. Por ejemplo, siguiendo el patrón Façade podría proporcionarse una única clase con un único método para gestionar toda la complejidad de comunicarse con las clases encargadas de comprimir un archivo de imagen al formato JPEG.

Estructura



Ejemplo

```java
package algo3;

import java.io.File;

public class BlockPreparation {

  public File prepare(String fileName) {
    System.out.println("Efectuando la preparacion de bloques...");
    File file = new File(fileName);
    return file;
  }
}
```

```java
package algo3;

import java.io.File;

public class DiscreteCosineTransform {

  public File transform(File file) {
    System.out.println("Efectuando la transformacion por coseno discreto...");
    return file;
  }
}
```

```java
package algo3;

import java.io.File;

public class Quantization {

  public File quantize(File file) {
    System.out.println("Efectuando la cuantizacion...");
    return file;
  }
}
```

```java
package algo3;

import java.io.File;

public class DifferentialQuantization {

  public File quantize(File file) {
    System.out.println("Efectuando la cuantizacion diferencial...");
    return file;
  }
}
```

```java
package algo3;

import java.io.File;

public class RunLengthEncoding {

  public File encode(File file) {
    System.out.println("Efectuando la codificacion por longitud de serie...");
    return file;
  }
}
```

```java
package algo3;

import java.io.File;

public class StatisticalOutputEncoding {

  public File encode(File file) {
    System.out.println("Efectuando la codificación estadistica de salida...");
    return file;
  }
}
```

```java
package algo3;

import java.io.File;

public class EncoderJPEG {

  private BlockPreparation blockPreparation;
  private DiscreteCosineTransform discreteCosineTransform;
  private Quantization quantization;
  private DifferentialQuantization differentialQuantization;
  private RunLengthEncoding runLengthEncoding;
  private StatisticalOutputEncoding statisticalOutputEncoding;

  public EncoderJPEG() {
    blockPreparation = new BlockPreparation();
    discreteCosineTransform = new DiscreteCosineTransform();
    quantization = new Quantization();
```

```
    differentialQuantization = new DifferentialQuantization();
    runLengthEncoding = new RunLengthEncoding();
    statisticalOutputEncoding = new StatisticalOutputEncoding();
  }

  public File encode(String fileName) {
    System.out.println("EncoderJPEG: iniciando la compresion de " + fileName + ".");
    File file1 = blockPreparation.prepare(fileName);
    File file2 = discreteCosineTransform.transform(file1);
    File file3 = quantization.quantize(file2);
    File file4 = differentialQuantization.quantize(file3);
    File file5 = runLengthEncoding.encode(file4);
    File file6 = statisticalOutputEncoding.encode(file5);
    System.out.print("EncoderJPEG: compresion finalizada. Se ha generado: ");
    System.out.println(fileName.substring(0, fileName.indexOf('.'))+".jpg");
    return file6;
  }
}
```

```
package algo3;

import java.io.File;

public class Main {

  public static void main(String[] args) {
    EncoderJPEG encoder = new EncoderJPEG();
    File jPEGfile = encoder.encode("photo.raw");
    // Aqui ya esta disponible photo.jpg

  }
}
```

Salida:

```
EncoderJPEG: iniciando la compresion de photo.raw.
Efectuando la preparacion de bloques...
Efectuando la transformacion por coseno discreto...
Efectuando la cuantizacion...
Efectuando la cuantizacion diferencial...
Efectuando la codificacion por longitud de serie...
Efectuando la codificación estadistica de salida...
EncoderJPEG: compresion finalizada. Se ha generado: photo.jpg
```

Usos conocidos

Muchas clases de Java siguen este patrón, por ejemplo `javax.faces.context.ExternalContext`: https://docs.oracle.com/javaee/7/api/javax/faces/context/ExternalContext.html

Consecuencias

Efectos deseados: El patrón permite que el código del cliente quede aislado de la complejidad de un subsistema. Promueve un débil acoplamiento entre el subsistema y sus clientes.

Efectos no deseados: Una fachada puede convertirse en un *objeto todopoderoso* acoplado a todas las clases de una aplicación.

5. Decorator (Decorador) — Patrón estructural (ámbito: objeto)

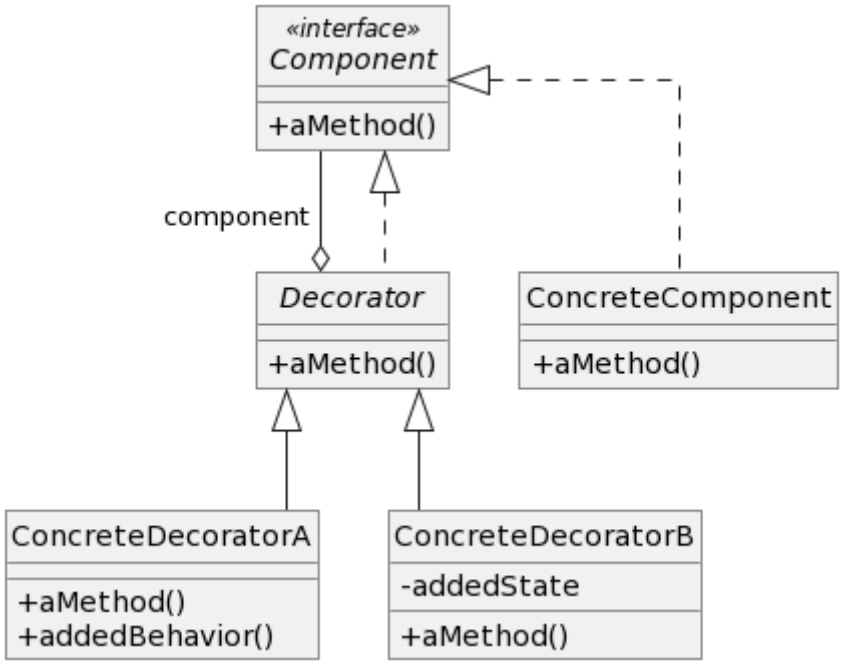
Propósito

Añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

Aplicabilidad

Cuando se desee añadir responsabilidades a objetos individuales de forma dinámica y transparente, es decir, sin afectar a otros objetos. Cuando se desee que las responsabilidades puedan ser retiradas. Cuando la extensión mediante la herencia no es viable (a veces es posible tener un gran número de extensiones independientes, produciéndose una explosión de subclases para permitir todas las combinaciones). Por ejemplo, un procesador de textos podría utilizar este patrón para aplicarles diferentes combinaciones de estilos a las distintas partes de un texto sin tener que disponer de una clase específica para cada una de las combinaciones.

Estructura



Ejemplo

```
package algo3;

public interface MensajeAbstracto {
  String getTexto();
}
```

```
package algo3;

public class Mensaje implements MensajeAbstracto {

  private String texto;

  public Mensaje(String texto) {
```

```
    this.texto = texto;
  }

  public String getTexto() {
    return texto;
  }
}
```

```
package algo3;

public abstract class Decorador implements MensajeAbstracto {

  private MensajeAbstracto mensaje;

  public Decorador(MensajeAbstracto mensaje) {
    this.mensaje = mensaje;
  }

  public String getTexto() {
    return mensaje.getTexto();
  }
}
```

```
package algo3;

public class Expanded extends Decorador {

  public Expanded(MensajeAbstracto mensaje) {
    super(mensaje);
  }

  @Override
  public String getTexto() {
    String[] vectorCaracteres = super.getTexto().split("");
      StringBuilder resultado = new StringBuilder();
      for(String caracter : vectorCaracteres){
          resultado.append(caracter);
          resultado.append(" ");
      }
    return resultado.deleteCharAt(resultado.length()-1).toString();
  }
}
```

```
package algo3;

public class CamelCase extends Decorador {

  public CamelCase(MensajeAbstracto mensaje) {
    super(mensaje);
  }

  @Override
  public String getTexto() {
    String[] vectorPalabras = super.getTexto().trim().split("\\P{L}+");
      StringBuilder resultado = new StringBuilder();
      for(String palabra : vectorPalabras){
          resultado.append(palabra.substring(0,1).toUpperCase());
          resultado.append(palabra.substring(1));
      }
    return resultado.toString();
  }
}
```

```java
package algo3;

public class Main {

  public static void main(String[] args) {

    MensajeAbstracto mensaje = new Mensaje("el mensaje es Hola, mundo!");
    System.out.print("Mensaje original:\n    ");
    System.out.println(mensaje.getTexto());

    MensajeAbstracto expanded = new Expanded(mensaje);
    System.out.print("Mensaje expandido:\n    ");
    System.out.println(expanded.getTexto());

    MensajeAbstracto camelCase = new CamelCase(mensaje);
    System.out.print("Mensaje en camel case:\n    ");
    System.out.println(camelCase.getTexto());

    MensajeAbstracto expandedCamelCase = new Expanded(new CamelCase(mensaje));
    System.out.print("Mensaje en camel case y expandido:\n    ");
    System.out.println(expandedCamelCase.getTexto());

    MensajeAbstracto camelCaseExpanded = new CamelCase(new Expanded(mensaje));
    System.out.print("Mensaje expandido y en camel case:\n    ");
    System.out.println(camelCaseExpanded.getTexto());
  }
}
```

Salida:

```
Mensaje original:
    el mensaje es Hola, mundo!
Mensaje expandido:
    e l   m e n s a j e   e s   H o l a ,   m u n d o !
Mensaje en camel case:
    ElMensajeEsHolaMundo
Mensaje en camel case y expandido:
    E l M e n s a j e E s H o l a M u n d o
Mensaje expandido y en camel case:
    ELMENSAJEESHOLAMUNDO
```

Usos conocidos

Muchas clases de Java siguen este patrón, por ejemplo `java.io.Reader` y `java.io.Writer`:
https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/io/Reader.html
https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/io/Writer.html

Consecuencias

Efectos deseados: El patrón ofrece más flexibilidad que la herencia estática (ofrece un enfoque para añadir responsabilidades que consiste en pagar solo por aquello que se necesita). El patrón permite seguir el *Principio de Responsabilidad Única*.

Efectos no deseados: Un diseño que usa este patrón suele dar como resultado sistemas formados por muchos objetos pequeños muy parecidos (los objetos solo se diferencian en la forma en que están interconectados, y no en su clase o en el valor de sus variables. Aunque dichos sistemas son fáciles de adaptar por quienes los comprenden bien, pueden ser difíciles de aprender).

6. Template Method (Método Plantilla) Patrón de comportamiento (ámbito: clase)

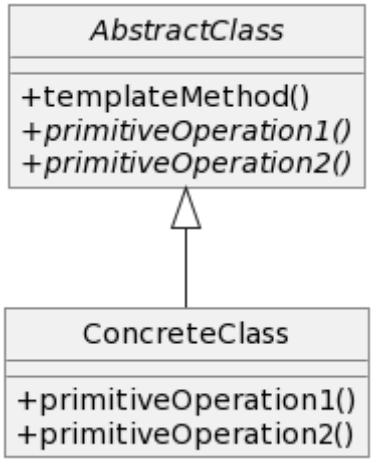
Propósito

Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.

Aplicabilidad

Cuando se desee implementar en una clase las partes de un algoritmo que no cambian y dejar que sean las subclases quienes implementen el comportamiento que puede variar. Cuando, para evitar el código duplicado, se desee localizar en una clase común el comportamiento repetido de varias subclases. Por ejemplo, al generar un informe de compras con una tarjeta de crédito, el cálculo de los cargos por los impuestos aplicables varía según el tipo de compra (nacional o internacional), pero, una vez sumados estos, el algoritmo para calcular el total a pagar no varía, por lo cual podría seguirse el patrón Template Method al desarrollar las clases.

Estructura



Ejemplo

```
package algo3;

public abstract class Compra {

  private double monto;
  private double impuestos;

  public Compra(double monto) {
    this.monto = monto;
  }

  public double getMonto() {
    return monto;
  }

  public double getImpuestos() {
    return impuestos;
  }
```

```
  public void setImpuestos(double impuestos) {
    this.impuestos = impuestos;
  }

  public void mostrarInforme() {
    mostrarTitulo();
    calcularImpuestos();
    mostrarImpuestos();
    sumarImpuestos();
    mostrarTotal();
  }

  public abstract void mostrarTitulo();

  public abstract void calcularImpuestos();

  public abstract void sumarImpuestos();

  public abstract void mostrarImpuestos();

  private void mostrarTotal() {
    System.out.println("Valor a pagar: " + (monto + impuestos));
  }
}
```

```
package algo3;

public class CompraNacionalConTarjetaCABA extends Compra {

  private double impuestoDeSellos;

  public CompraNacionalConTarjetaCABA(double monto) {
    super(monto);
  }

  @Override
  public void mostrarTitulo() {
    System.out.println("Compra nacional con tarjeta de credito CABA: " + getMonto());
  }

  @Override
  public void calcularImpuestos() {
    impuestoDeSellos = getMonto() * 0.012;
  }

  @Override
  public void mostrarImpuestos() {
    System.out.println("Impuesto de sellos (1.2%): " + impuestoDeSellos);
  }

  @Override
  public void sumarImpuestos() {
    setImpuestos(impuestoDeSellos);
  }
}
```

```
package algo3;

public class CompraInternacionalConTarjetaCABA extends Compra {

  private double impuestoPAIS;
  private double percepcionRG4815;
  private double impuestoDeSellos;
```

```
  public CompraInternacionalConTarjetaCABA(double monto) {
    super(monto);
  }

  @Override
  public void mostrarTitulo() {
    System.out.println("Compra internacional con tarjeta de credito CABA: " + getMonto());
  }

  @Override
  public void calcularImpuestos() {
    impuestoPAIS = getMonto() * 0.3;
    percepcionRG4815 = getMonto() * 0.45;
    impuestoDeSellos = getMonto() * 0.012;
  }

  @Override
  public void mostrarImpuestos() {
    System.out.println("Impuesto PAIS (30%): " + impuestoPAIS);
    System.out.println("Percepcion RG 4815 (45%): " + percepcionRG4815);
    System.out.println("Impuesto de sellos (1.2%): " + impuestoDeSellos);
  }

  @Override
  public void sumarImpuestos() {
    setImpuestos(impuestoPAIS + percepcionRG4815 + impuestoDeSellos);
  }
}
```

```
package algo3;

public class Main {

  public static void main(String[] args) {

    double monto = 10000;

    Compra compraNacionalConTarjetaCABA = new CompraNacionalConTarjetaCABA(monto);
    compraNacionalConTarjetaCABA.mostrarInforme();

    System.out.println();

    Compra compraInternacionalConTarjetaCABA = new CompraInternacionalConTarjetaCABA(monto);
    compraInternacionalConTarjetaCABA.mostrarInforme();
  }
}
```

Salida:

```
Compra nacional con tarjeta de credito CABA: 10000.0
Impuesto de sellos (1.2%): 120.0
Valor a pagar: 10120.0

Compra internacional con tarjeta de credito CABA: 10000.0
Impuesto PAIS (30%): 3000.0
Percepcion RG 4815 (45%): 4500.0
Impuesto de sellos (1.2%): 120.0
Valor a pagar: 17620.0
```

Usos conocidos

Muchas clases de Java siguen este patrón, por ejemplo `java.util.AbstractList`: https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/AbstractList.html

Consecuencias

Efectos deseados: El patrón permite a los clientes que sobrescriban tan solo ciertas partes de un algoritmo grande, para que les afecten menos los cambios que tienen lugar en otras partes del algoritmo, y sigue el *Principio DRY* al colocar el código duplicado dentro de una superclase. Los métodos plantilla son una técnica fundamental de reutilización de código. Son particularmente importantes en las bibliotecas de clases, ya que son el modo de factorizar y extraer el comportamiento común de las clases de la biblioteca. Los métodos plantilla llevan a una estructura de control invertido que a veces se denomina el *Principio de Hollywood*, es decir, "No nos llame, nosotros la/lo llamaremos". Esto se refiere a cómo una clase padre llama a las operaciones de una subclase y no al revés.

Efectos no deseados: Algunos clientes pueden verse limitados por el esqueleto proporcionado de un algoritmo. Podría violarse el *Principio de Sustitución de Liskov* si se suprime una implementación por defecto de un paso a través de una subclase. Los métodos plantilla tienden a ser más difíciles de mantener cuantos más pasos tengan.

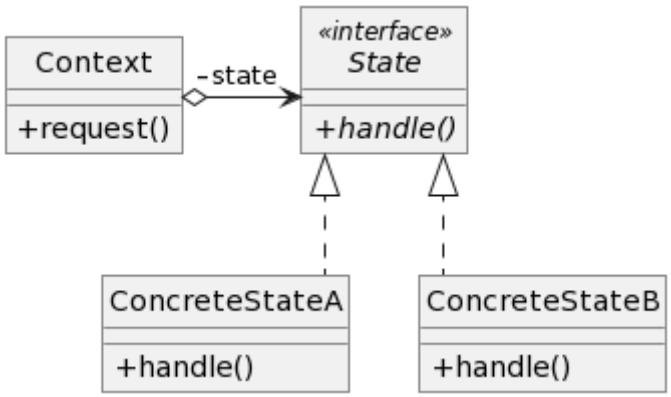
Propósito

Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.

Aplicabilidad

Cuando el comportamiento de un objeto dependa de su estado y deba cambiar en tiempo de ejecución dependiendo de ese estado. Cuando las operaciones tengan largas sentencias condicionales con múltiples ramas que dependen del estado del objeto. Este estado se suele representar por una o más constantes enumeradas. Muchas veces son varias las operaciones que contienen esta misma estructura condicional. El patrón State pone cada rama de la condición en una clase aparte. Esto permite tratar al estado del objeto como un objeto de pleno derecho que puede variar independientemente de otros objetos. Por ejemplo, un televisor tiene estados que pueden cambiarse desde el control remoto, por lo que el software de este podría programarse siguiendo este patrón, y se evitarían así las sentencias condicionales.

Estructura



Ejemplo

```
package algo3;

public interface PossibleState {
  void pressOnButton(Television tvContext);
  void pressOffButton(Television tvContext);
  void pressMuteButton(Television tvContext);
}
```

```
package algo3;

public class Television {
  private PossibleState currentState;

  public PossibleState getCurrentState() {
    return currentState;
  }
```

```
  public void setCurrentState(PossibleState currentState) {
    this.currentState = currentState;
  }

  public Television() {
    this.currentState = new OffState();
  }

  public void executeOffButton() {
    System.out.println("Ud. ha presionado Off.");
    currentState.pressOffButton(this);
  }

  public void executeOnButton() {
    System.out.println("Ud. ha presionado On.");
    currentState.pressOnButton(this);
  }

  public void executeMuteButton() {
    System.out.println("Ud. ha presionado Mute.");
    currentState.pressMuteButton(this);
  }
}
```

```
package algo3;

public class OffState implements PossibleState {

  public String toString() {
    return "Televisor apagado.";
  }

  public void pressOffButton(Television tvContext) {
    System.out.println("El televisor sigue apagado.");
  }

  public void pressMuteButton(Television tvContext) {
    System.out.println("El televisor va a seguir apagado.");
  }

  public void pressOnButton(Television tvContext) {
    System.out.println("El televisor estaba apagado. Se va a encender con audio.");
    tvContext.setCurrentState(new OnState());
  }
}
```

```
package algo3;

public class OnState implements PossibleState {

  public String toString() {
    return "Televisor encendido con audio.";
  }

  public void pressOnButton(Television tvContext) {
    System.out.println("El televisor sigue encendido con audio.");
  }
```

```
  public void pressMuteButton(Television tvContext) {
    System.out.println("El televisor estaba encendido con audio. Se va a desactivar el audio.");
    tvContext.setCurrentState(new MuteState());
  }

  public void pressOffButton(Television tvContext) {
    System.out.println("El televisor estaba encendido con audio. Se va a apagar.");
    tvContext.setCurrentState(new OffState());
  }
}
```

```
package algo3;

public class MuteState implements PossibleState {

  public String toString() {
    return "Televisor encendido sin audio.";
  }

  public void pressOnButton(Television tvContext) {
    System.out.println("El televisor sigue encendido sin audio.");
  }

  public void pressMuteButton(Television tvContext) {
    System.out.println("El televisor estaba encendido sin audio. Se va a activar el audio.");
    tvContext.setCurrentState(new OnState());
  }

  public void pressOffButton(Television tvContext) {
    System.out.println("El televisor estaba encendido sin audio. Se va a apagar.");
    tvContext.setCurrentState(new OffState());
  }
}
```

```
package algo3;

public class Main {

  public static void main(String[] args) {

    Television tv = new Television();

    System.out.println("Secuencia de teclas del control remoto de TV presionadas:");
    System.out.println("Off -> Mute -> On -> On -> Mute -> Mute -> On -> Off");
    System.out.println();

    tv.executeOffButton(); System.out.println("Estado: " + tv.getCurrentState());
    tv.executeMuteButton(); System.out.println("Estado: " + tv.getCurrentState());
    tv.executeOnButton(); System.out.println("Estado: " + tv.getCurrentState());
    tv.executeOnButton(); System.out.println("Estado: " + tv.getCurrentState());
    tv.executeMuteButton(); System.out.println("Estado: " + tv.getCurrentState());
    tv.executeMuteButton(); System.out.println("Estado: " + tv.getCurrentState());
    tv.executeOnButton(); System.out.println("Estado: " + tv.getCurrentState());
    tv.executeOffButton(); System.out.println("Estado: " + tv.getCurrentState());
  }
}
```

Salida:

```
Secuencia de teclas del control remoto de TV presionadas:
Off -> Mute -> On -> On -> Mute -> Mute -> On -> Off

Ud. ha presionado Off.
El televisor sigue apagado.
Estado: Televisor apagado.
Ud. ha presionado Mute.
El televisor va a seguir apagado.
Estado: Televisor apagado.
Ud. ha presionado On.
El televisor estaba apagado. Se va a encender con audio.
Estado: Televisor encendido con audio.
Ud. ha presionado On.
El televisor sigue encendido con audio.
Estado: Televisor encendido con audio.
Ud. ha presionado Mute.
El televisor estaba encendido con audio. Se va a desactivar el audio.
Estado: Televisor encendido sin audio.
Ud. ha presionado Mute.
El televisor estaba encendido sin audio. Se va a activar el audio.
Estado: Televisor encendido con audio.
Ud. ha presionado On.
El televisor sigue encendido con audio.
Estado: Televisor encendido con audio.
Ud. ha presionado Off.
El televisor estaba encendido con audio. Se va a apagar.
Estado: Televisor apagado.
```

Usos conocidos

Muchas clases de Java siguen este patrón, por ejemplo `javax.faces.lifecycle.Lifecycle`: https://docs.oracle.com/javaee/7/api/javax/faces/lifecycle/Lifecycle.html

Consecuencias

Efectos deseados: El patrón sigue el *Principio de Responsabilidad Única* (organiza en clases separadas el código relacionado con estados particulares) y el *Principio Abierto/Cerrado* (es posible introducir nuevos estados sin cambiar las clases de estado existentes o la clase contexto).
Efectos no deseados: Aplicar este patrón puede resultar excesivo si una máquina de estados solo tiene unos pocos estados o raramente cambia.

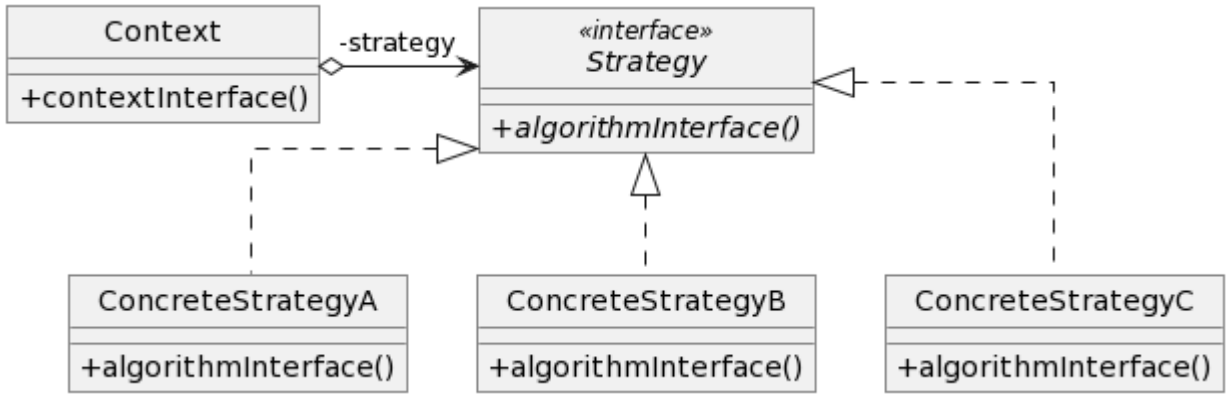
Propósito

Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.

Aplicabilidad

Cuando muchas clases relacionadas difieran solo en su comportamiento (las estrategias permiten configurar una clase con un determinado comportamiento de entre muchos posibles). Cuando una clase define muchos comportamientos, y estos se representan como múltiples sentencias condicionales en sus operaciones (en vez de tener muchos condicionales, se pueden mover las ramas de estos a su propia clase de estrategia). Por ejemplo, un sistema de cálculo de áreas de polígonos que tenga una interfaz común y distintas implementaciones según el tipo de polígono podría implementarse siguiendo este patrón.

Estructura



Ejemplo

```java
package algo3;

public interface EstrategiaParaCalcularArea {

  double calcularArea(int cantidadDeLados, double lado);
}
```

```java
package algo3;

public class AreaConMultiplicacion implements EstrategiaParaCalcularArea {

  @Override
  public double calcularArea(int cantidadDeLados, double lado) {
    return lado * lado;
  }
}
```

```java
package algo3;

public class AreaConRaizCuadrada implements EstrategiaParaCalcularArea {

  @Override
  public double calcularArea(int cantidadDeLados, double lado) {
    return lado * lado * Math.sqrt(3) / 4;
  }
}
```

```java
package algo3;

public class AreaConTrigonometria implements EstrategiaParaCalcularArea {

  @Override
  public double calcularArea(int cantidadDeLados, double lado) {
    return cantidadDeLados * lado * lado / (4 * Math.tan(Math.PI / cantidadDeLados));
  }
}
```

```java
package algo3;

public abstract class PoligonoRegular {

  private double lado;
  private String nombre;
  private int cantidadDeLados;
  private EstrategiaParaCalcularArea estrategiaParaCalcularArea;

  public PoligonoRegular(double lado, String nombre, int cantidadDeLados,
      EstrategiaParaCalcularArea estrategiaParaCalcularArea) {
    this.lado = lado;
    this.nombre = nombre;
    this.cantidadDeLados = cantidadDeLados;
    this.estrategiaParaCalcularArea = estrategiaParaCalcularArea;
  }

  public void setEstrategiaParaCalcularArea(EstrategiaParaCalcularArea estrategia) {
    this.estrategiaParaCalcularArea = estrategia;
  }

  public void mostrarArea() {
    System.out.print("Area del " + nombre + ": ");
    System.out.println(estrategiaParaCalcularArea.calcularArea(cantidadDeLados, lado));
  }
}
```

```java
package algo3;

public class TrianguloEquilatero extends PoligonoRegular {

  public TrianguloEquilatero(double lado, EstrategiaParaCalcularArea estrategiaParaCalcularArea) {
    super(lado, "Triangulo Equilatero", 3, estrategiaParaCalcularArea);
  }
}
```

```java
package algo3;
public class Cuadrado extends PoligonoRegular {
  public Cuadrado(double lado, EstrategiaParaCalcularArea estrategiaParaCalcularArea) {
    super(lado, "Cuadrado", 4, estrategiaParaCalcularArea);
  }
}
```

```java
package algo3;
public class PentagonoRegular extends PoligonoRegular {
  public PentagonoRegular(double lado, EstrategiaParaCalcularArea estrategiaParaCalcularArea) {
    super(lado, "Pentagono Regular", 5, estrategiaParaCalcularArea);
  }
}
```

```java
package algo3;
import java.util.ArrayList;
public class Main {
  public static void main(String[] args) {
    ArrayList<PoligonoRegular> coleccionPolimorfica = new ArrayList<>();
    coleccionPolimorfica.add(new TrianguloEquilatero(5, new AreaConRaizCuadrada()));
    coleccionPolimorfica.add(new TrianguloEquilatero(5, new AreaConTrigonometria()));
    coleccionPolimorfica.add(new Cuadrado(5, new AreaConMultiplicacion()));
    coleccionPolimorfica.add(new Cuadrado(5, new AreaConTrigonometria()));
    coleccionPolimorfica.add(new PentagonoRegular(5, new AreaConTrigonometria()));
    for (PoligonoRegular poligono : coleccionPolimorfica) {
      poligono.mostrarArea();
    }
    coleccionPolimorfica.get(3).setEstrategiaParaCalcularArea(new AreaConMultiplicacion());
    System.out.println();
    System.out.println("Despues de cambiar la estrategia del segundo cuadrado...");
    for (PoligonoRegular poligono : coleccionPolimorfica) {
      poligono.mostrarArea();
    }
  }
}
```

Salida:

```
Area del Triangulo Equilatero: 10.825317547305483
Area del Triangulo Equilatero: 10.825317547305486
Area del Cuadrado: 25.0
Area del Cuadrado: 25.000000000000004
Area del Pentagono Regular: 43.01193501472417

Despues de cambiar la estrategia del segundo cuadrado...
Area del Triangulo Equilatero: 10.825317547305483
Area del Triangulo Equilatero: 10.825317547305486
Area del Cuadrado: 25.0
Area del Cuadrado: 25.0
Area del Pentagono Regular: 43.01193501472417
```

Usos conocidos

Muchas clases de Java siguen este patrón, por ejemplo `java.text.Collator`:
https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/text/Collator.html

Consecuencias

Efectos deseados: El patrón sigue el *Principio de Responsabilidad Única* (organiza en clases separadas el código relacionado con algoritmos particulares) y el *Principio Abierto/Cerrado* (es posible introducir nuevos comportamientos sin cambiar las clases de estrategia existentes o la clase contexto). Permite aislar los detalles de implementación de un algoritmo del código que lo utiliza y sustituir la herencia por composición.

Efectos no deseados: Aplicar este patrón puede resultar excesivo si solo hay unos pocos algoritmos involucrados o si estos raramente cambian (además, muchos lenguajes de programación modernos tienen un soporte de tipo funcional que permite implementar distintas versiones de un algoritmo dentro de un grupo de funciones anónimas, sin saturar el código con clases e interfaces adicionales). Los clientes deben conocer las diferencias entre las estrategias para poder seleccionar la adecuada.

9. Observer (Observador) — Patrón de comportamiento (ámbito: objeto)

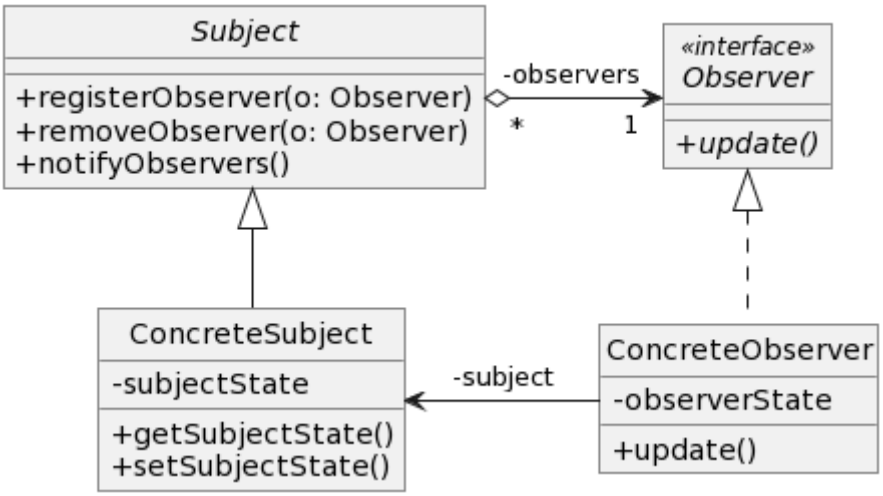
Propósito

Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifica y se actualizan automáticamente todos los objetos que dependen de él.

Aplicabilidad

Cuando una abstracción tiene dos aspectos y uno depende del otro (encapsular estos aspectos en objetos separados permite modificarlos y reutilizarlos de forma independiente). Cuando un cambio en un objeto requiere cambiar otros, y no se sabe cuántos objetos necesitan cambiarse. Cuando un objeto debería ser capaz de notificar a otros sin hacer suposiciones sobre quiénes son dichos objetos (en otras palabras, cuando no se quiere que estos objetos estén fuertemente acoplados). Por ejemplo, en un sistema de comunicación mediante difusión, donde las notificaciones se envía automáticamente a todos los objetos observadores interesados que se hayan suscrito a ella, puede aplicarse este patrón de diseño.

Estructura



Ejemplo

```
package algo3;

import java.util.*;

// 1) Knows its observers. Any number of Observer objects may observe a subject.
// 2) Provides an interface for attaching and detaching Observer objects.

public abstract class ObservableSubject {

  private List<Observer> observers = new ArrayList<>();

  public void registrarObserver(Observer observer) {
    observers.add(observer);
  }
```

```
  public void removerObserver(Observer observer) {
    observers.remove(observer);
  }

  public void notificarObservers() {
    for (Observer observer : observers)
      observer.actualizar();
  }
}
```

```
package algo3;

// 1) Stores state of interest to ConcreteObserver objects.
// 2) Sends a notification to its observers when its state changes.

public class ConcreteSubjectSpam extends ObservableSubject {
  private String mensajeSpam;

  public String getMensajeSpam() {
    return mensajeSpam;
  }

  public void setMensajeSpam(String mensajeSpam) {
    this.mensajeSpam = mensajeSpam;
    notificarObservers();
  }
}
```

```
package algo3;

// Defines an updating interface for objects that should be notified of changes in a
subject.

public interface Observer {

  void actualizar();
}
```

```
package algo3;

// 1) Maintains a reference to a ConcreteSubject object.
// 2) Stores state that should stay consistent with the subject's.
// 3) Implements an updating interface to keep its state consistent with the subject's.

public class ConcreteObserverUsuario implements Observer {
  private ConcreteSubjectSpam spam;
  private String mensajeSpamRecibido;
  private String nombre;

  public ConcreteObserverUsuario(String nombre, ConcreteSubjectSpam spam) {
    this.nombre = nombre;
    this.spam = spam;
    actualizarMensajeSpamRecibido();
  }

  public void actualizar() {
    actualizarMensajeSpamRecibido();
  }
```

```
  private void actualizarMensajeSpamRecibido() {
    mensajeSpamRecibido = spam.getMensajeSpam();
  }

  public void mostrar() {
    System.out.println("Estado de " + nombre + ": " + mensajeSpamRecibido);
  }
}
```

```
package algo3;

public class Main {

  private static void mostrarEstadoDeUsuarios(Observer u1, Observer u2, Observer u3) {
    ((ConcreteObserverUsuario) u1).mostrar();
    ((ConcreteObserverUsuario) u2).mostrar();
    ((ConcreteObserverUsuario) u3).mostrar();
    System.out.println();
  }

  public static void main(String[] args) {

    ConcreteSubjectSpam s = new ConcreteSubjectSpam();
    ConcreteObserverUsuario u1 = new ConcreteObserverUsuario("Matias", s);
    ConcreteObserverUsuario u2 = new ConcreteObserverUsuario("Diego", s);
    ConcreteObserverUsuario u3 = new ConcreteObserverUsuario("Esteban", s);

    s.registrarObserver(u1);
    s.registrarObserver(u2);
    s.setMensajeSpam("Solo por hoy! Laptops 30% Off");
    mostrarEstadoDeUsuarios(u1, u2, u3);
    s.registrarObserver(u3);
    s.setMensajeSpam("Oferta imperdible! Cartuchos HP 3x2");
    mostrarEstadoDeUsuarios(u1, u2, u3);

    s.removerObserver(u2);
    s.setMensajeSpam("Ultima oportunidad! Impresoras Brother 10% Off");
    mostrarEstadoDeUsuarios(u1, u2, u3);
  }
}
```

Salida:

```
Estado de Matias: Solo por hoy! Laptops 30% Off
Estado de Diego: Solo por hoy! Laptops 30% Off
Estado de Esteban: null

Estado de Matias: Oferta imperdible! Cartuchos HP 3x2
Estado de Diego: Oferta imperdible! Cartuchos HP 3x2
Estado de Esteban: Oferta imperdible! Cartuchos HP 3x2

Estado de Matias: Ultima oportunidad! Impresoras Brother 10% Off
Estado de Diego: Oferta imperdible! Cartuchos HP 3x2
Estado de Esteban: Ultima oportunidad! Impresoras Brother 10% Off
```

Usos conocidos

Muchas clases de Java siguen este patrón, por ejemplo, las que implementan la interfaz `java.util.EventListener`:
https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/EventListener.html

Consecuencias

Efectos deseados: El patrón sigue el *Principio Abierto/Cerrado* (el patrón permite agregar clases notificadoras *Subject* y clases suscriptoras *Observer* de forma independiente, siendo posible reutilizar las primeras sin reutilizar las segundas, y viceversa).

Efectos no deseados: Aplicar este patrón incorrectamente puede provocar actualizaciones inesperadas (una operación aparentemente inofensiva sobre la clase notificadora puede dar lugar a una serie de actualizaciones en cascada de los observadores y sus objetos dependientes, provocando falsas actualizaciones, que pueden ser muy difíciles de localizar).

10. Visitor (Visitante) — Patrón de comportamiento (ámbito: objeto)

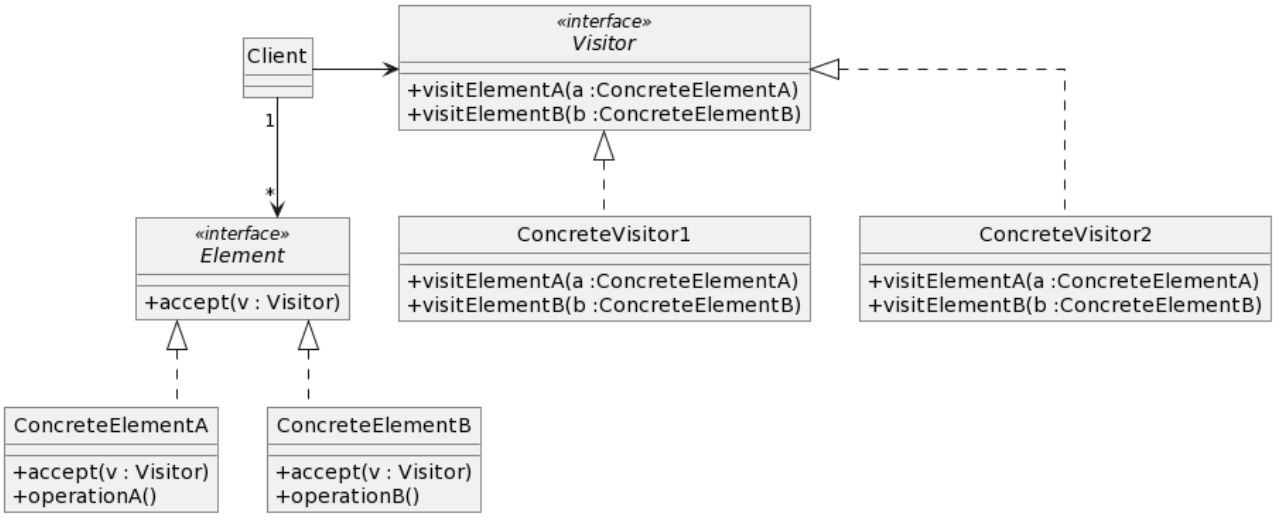
Propósito

Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

Aplicabilidad

Cuando una estructura de objetos contiene muchas clases de objetos con diferentes interfaces, y se quiere realizar operaciones sobre esos elementos que dependen de su clase concreta. Cuando se necesita realizar muchas operaciones distintas y no relacionadas sobre objetos de una estructura de objetos, y se quiere evitar "contaminar" sus clases con dichas operaciones. Cuando las clases que definen la estructura de objetos rara vez cambian, pero se desea definir nuevas operaciones sobre la estructura. Por ejemplo, en un sistema de facturación, los distintos productos pueden tener algoritmos diferentes para calcular sus precios (la venta de algunos podría ser por unidad y la de otros por peso), y en tal caso se puede seguir este patrón de diseño.

Estructura



Ejemplo

```
package algo3;

public interface VisitanteCalculoDeCosto {

  double visitar(ProductoPorUnidad producto);

  double visitar(ProductoPorPeso producto);
}
```

```
package algo3;

public class VisitanteConcretoCalculoDeCosto implements VisitanteCalculoDeCosto {
```

```
  public double visitar(ProductoPorUnidad producto) {
    double costo = producto.getPrecioUnitario();
    System.out.printf("1 %s: $%.2f (precio por unidad)\n",
      producto.getDenominacion(), costo);
    return costo;
  }

  public double visitar(ProductoPorPeso producto) {
    double costo = producto.getPeso() * producto.getPrecioPorKg();
    System.out.printf("%.2f Kg de %s a $%.2f/Kg: $%.2f\n",
      producto.getPeso(),
      producto.getDenominacion(),
      producto.getPrecioPorKg(), costo);
    return costo;
  }
}
```

```
package algo3;

public interface Elemento {

  public double accept(VisitanteCalculoDeCosto visitante);
}
```

```
package algo3;

public class ProductoPorPeso implements Elemento {

  private double precioPorKg;
  private double peso;
  private String denominacion;

  public ProductoPorPeso(double peso, String denominacion, double precioPorKg) {
    this.precioPorKg = precioPorKg;
    this.peso = peso;
    this.denominacion = denominacion;
  }

  public double getPrecioPorKg() {
    return precioPorKg;
  }

  public double getPeso() {
    return peso;
  }

  public String getDenominacion() {
    return denominacion;
  }

  public double accept(VisitanteCalculoDeCosto visitante) {
    return visitante.visitar(this);
  }
}
```

```
package algo3;

public class ProductoPorUnidad implements Elemento {
```

```
  private double precioUnitario;
  private String denominacion;

  public ProductoPorUnidad(String denominacion, double precioUnitario) {
    this.precioUnitario = precioUnitario;
    this.denominacion = denominacion;
  }

  public double getPrecioUnitario() {
    return precioUnitario;
  }

  public String getDenominacion() {
    return denominacion;
  }

  public double accept(VisitanteCalculoDeCosto visitante) {
    return visitante.visitar(this);
  }
}
```

```
package algo3;

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

public class Main {

  public static void main(String[] args) {
    List<Elemento> elementos = new ArrayList<>();
    elementos.add(new ProductoPorUnidad("Mango", 349));
    elementos.add(new ProductoPorPeso(1.4, "Banana", 259));
    elementos.add(new ProductoPorUnidad("Palta", 239));
    elementos.add(new ProductoPorPeso(1.8, "Naranja", 119));

    VisitanteCalculoDeCosto visitante = new VisitanteConcretoCalculoDeCosto();
    double total = 0;
    for (Elemento elemento : elementos) {
      total += elemento.accept(visitante);
    }
    System.out.printf("Total: $%.2f", total);
  }
}
```

Salida:

```
1 Mango: $349,00 (precio por unidad)
1,40 Kg de Banana a $259,00/Kg: $362,60
1 Palta: $239,00 (precio por unidad)
1,80 Kg de Naranja a $119,00/Kg: $214,20
Total: $1164,80
```

Usos conocidos

Muchas clases de Java siguen este patrón, por ejemplo, la mayoría de las incluidas en el paquete `javax.lang.model.util`: https://docs.oracle.com/en/java/javase/19/docs/api/java.compiler/javax/lang/model/util/package-summary.html

Consecuencias

Efectos deseados: El patrón sigue el *Principio de Responsabilidad Única* (es posible tomar varias versiones del mismo comportamiento y ponerlas en la misma clase) y el *Principio Abierto/Cerrado* (introducir un nuevo comportamiento que puede funcionar con objetos de clases diferentes sin cambiar esas clases). Un objeto visitante puede acumular cierta información útil mientras trabaja con varios objetos (esto puede resultar útil cuando se desee atravesar una compleja estructura de objetos, como un árbol de objetos, y aplicar el visitante a cada objeto de esa estructura).

Efectos no deseados: Es necesario actualizar todos los visitantes cada vez que una clase se añada o elimine de la jerarquía de elementos. Los visitantes pueden carecer del acceso necesario a los campos y métodos privados de los elementos con los que se supone que deben trabajar. El patrón suele obligar a proporcionar operaciones públicas que accedan al estado interno de un elemento, lo que puede comprometer su encapsulamiento.

11. Otros patrones para investigar

En este minicatálogo han quedado muchos patrones de diseño conocidos sin abarcar. Se recomienda investigarlos para tener un panorama más completo sobre este tema. Los principales son los siguientes:

GoF

Adapter (Adaptador): Convierte la interfaz de una clase en otra distinta que es la que esperan los clientes. Permite que cooperen clases que de otra manera no podrían por tener interfaces incompatibles.

Bridge (Puente): Desacopla una abstracción de su implementación, de manera que ambas puedan variar de forma independiente.

Builder (Constructor): Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.

Chain of Responsibility (Cadena de Responsabilidad): Evita acoplar el emisor de una petición a su receptor, al dar a más de un objeto la posibilidad de responder a la petición. Crea una cadena con los objetos receptores y pasa la petición a través de la cadena hasta que ésta sea tratada por algún objeto.

Command (Orden): Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con distintas peticiones, encolar o llevar un registro de las peticiones y poder deshacer las operaciones.

Composite (Compuesto): Combina objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.

Flyweight (Peso Ligero): Usa el compartimiento para permitir un gran número de objetos de grano fino de forma eficiente.

Interpreter (Intérprete): Dado un lenguaje, define una representación de su gramática junto con un intérprete que usa dicha representación para interpretar sentencias del lenguaje.

Iterator (Iterador): Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

Mediator (Mediador): Define un objeto que encapsula cómo interactúan un conjunto de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.

Memento (Recuerdo): Representa y externaliza el estado interno de un objeto sin violar la encapsulación, de forma que éste puede volver a dicho estado más tarde.

Prototype (Prototipo): Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando de este prototipo.

Proxy (Apoderado): Proporciona un sustituto o representante de otro objeto para controlar el acceso a éste.

Otros

Null object (objeto nulo): Provee un sustituto para otro objeto que comparte la misma interfaz pero en realidad no hace nada. El Objeto Nulo encapsula las decisiones de implementación de cómo efectivamente "no hacer nada" y esconde esos detalles de sus colaboradores.

12. Bibliografía y enlaces de interés

- **Java Design Patterns: A Hands-On Experience with Real-World Examples, 3ra. ed. (2022)**
  Vaskaran Sarcar
  ISBN-13: 978-1-4842-7970-0

- **Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software, 2da. ed. (2021)**
  Eric Freeman, Elisabeth Robson
  ISBN-13: 9781492078005

- **Design Patterns: Elements of Reusable Object-Oriented Software (1994)**
  Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
  ISBN-13: 9780201633610

- **https://refactoring.guru/es**
  Sitio con numerosas explicaciones y ejemplos sobre patrones de diseño y refactorización.

- **https://sourcemaking.com/design_patterns**
  Otro sitio con explicaciones y ejemplos sobre patrones de diseño y refactorización.

- **https://deviq.com/**
  Referencia resumida de principios y patrones de diseño, buenas prácticas, valores, antipatrones, etc.

Código fuente de los ejemplos de este minicatálogo:

- https://github.com/dcorsi/algo3/tree/main/catalogo-abstract-factory
- https://github.com/dcorsi/algo3/tree/main/catalogo-decorator
- https://github.com/dcorsi/algo3/tree/main/catalogo-facade
- https://github.com/dcorsi/algo3/tree/main/catalogo-factory-method
- https://github.com/dcorsi/algo3/tree/main/catalogo-observer
- https://github.com/dcorsi/algo3/tree/main/catalogo-singleton
- https://github.com/dcorsi/algo3/tree/main/catalogo-state
- https://github.com/dcorsi/algo3/tree/main/catalogo-strategy
- https://github.com/dcorsi/algo3/tree/main/catalogo-template-method
- https://github.com/dcorsi/algo3/tree/main/catalogo-visitor