Java-POO II

13.11.2025

Enrique Krause Buedo

1 Principios generales

13.11.2025

Enrique Krause Buedo

Principios de la POO

Es una nueva forma de pensar el proceso de descomposición del problema. Supone un paso a lenguajes de más alto nivel; es decir, más cercanos a la comunicación humana.
El proceso se centra en simular los elementos de la realidad asociados al problema.

así pues, los **objetos** similares se abstraen en **clases**. Y se dice que un objeto es una **instancia** de una clase.

Una clase contiene la información necesaria con la que abstraer un concepto mediante: los datos, llamados **atributos** y las operaciones, llamadas **métodos**.

Como todo tiene sus ventajas e inconvenientes, las

ventajas son más:

- Ocultación de la información
- Encapsulado de datos y procedimientos
- Herencia de clases
- Reusabilidad de clases ya definida y probadas
- Fiabilidad (se pueden aislar los errores con mas facilidad)

Inconvenientes:

- Cambia la forma de abordar el diseño de la aplicación
- La implementación es más compleja: no facilita el desarrollo en pequeñas aplicaciones

La ocultación de la información

Con la ocultación, se consigue proteger la información; aislarla. El términos generales, se suele realizar dando acceso a la modificación solamente mediante los métodos de la clase

Los niveles de ocultación, en general de atributos, de métodos o de clase, la mayoría de los lenguajes los definen con las palabras:

Público: accesible desde cualquier parte

Privado: Solamente desde la propia clase

Protegido: Desde la propia clase o desde clases derivadas de ella

Al definir la visibilidad de la clase, debe tenerse en cuenta:

1- Los atributos de una clase deberían ser privados, para que solo sean modificados mediante métodos de la propia clase.

2 -Los métodos de la clase deberían ser públicos.

3- Los métodos que definen las operaciones que ayudan a implementar parte de la funcionalidad deberían ser privados (si no se utilizan desde clases derivadas) **o protegidos** (si se utilizan desde clases derivadas).

2 Principios SOLID

13.11.2025

Enrique Krause Buedo

Uso correcto de la herencia

Más que el uso de la ocultación, a la hora de hacer un buen diseño de clases, la dificultad real está en la aplicación de la herencia. ¿Cuándo conviene heredar de clases abstractas y cuándo extender interfaces?

La aplicación de los **principios SOLID** permite realizar un diseño más óptimo, lo que facilita su posterior mantenimiento y ampliación. SOLID es un acrónimo mnemotécnico que representa **cinco principios básicos del diseño y la programación orientada a objetos.**

S: Single Responsibility Principle

O: Open Closed Principle

L: Liskov Substitution Principle

I: Interface Segregation Principle

D: Dependency Inversion Principle

(ya se que no todos saben inglés, por ellos está detallado más abajo)

Single Responsibility Principle (Principio de Responsabilidad Única): cada clase debe tener una sola responsabilidad y las operaciones que pueda realizar deben tener como objetivo el llevarla a cabo.
Open Closed Principle (Principio Abierto-Cerrado): una clase debe estar abierta a su extensión, pero cerrada a su modificación, ya que debe ser posible extender su funcionalidad sin necesidad de modificar su código fuente.

Liskov Substitution Principle (Principio de Sustitución de Liskov): toda subclase podría ser sustituida por su superclase; es decir, que el cliente de una superclase podría seguir funcionando si se cambia esa superclase por una de sus subclases.

Interface Segregation Principle (Principio de Segregación de Interfaces): es recomendable tener interfaces específicas para cada cliente y no tener una sola de propósito general, ya que ello obligaría a los clientes a depender de métodos que no utilizan.

Dependency Inversion Principle (Principio de Inversión de Dependencia): tiene como objetivo conseguir desacoplar las clases, haciendo que una clase interactúe con otras sin que las conozca directamente. Se utilizarán clases abstractas en niveles superiores, que no conocerán los detalles de la implementación en las clases de nivel inferior.

Este tema merece ser un poco más desarrollado, de modo que se entienda con palabras más sencillas que las del manual

Principio de única responsabilidad (S)

Si más de un agente puede solicitar cambios en una de nuestras clases, eso es que la clase tiene muchas razones para cambiar y, por lo tanto, mantiene muchas responsabilidades. En consecuencia, será necesario repartir esas responsabilidades entre clases.

Principio de única responsabilidad (S)

```
class Libro {
	String nombre;
	String nombreAutor;
	int anyo;
	int precio;
	String isbn;

	public Libro(String nombre, String nombreAutor, int anyo, int precio, String isbn) {
		this.nombre = nombre;
		this.nombreAutor = nombreAutor;
		this.anyo = anyo;
	  this.precio = precio;
		this.isbn = isbn;
	}
}
public class Factura {
	private Libro libro;
	private int cantidad;
	private double tasaDescuento;
	private double tasaImpuesto;
	private double total;
	public Factura(Libro libro, int cantidad, double tasaDescuento, double tasaImpuesto) {
		this.libro = libro;
		this.cantidad = cantidad;
		this.tasaDescuento = tasaDescuento;
		this.tasaImpuesto = tasaImpuesto;
		this.total = this.calculaTotal();
	}
```

```
    public double calculaTotal() {
            double precio = ((libro.precio - libro.precio * tasaDescuento) * this.cantidad);

           double precioConImpuestos = precio * (1 + tasaImpuesto);

           return precioConImpuestos;
    }
    public void imprimeFactura() {
          System.out.println(cantidad + "x " + libro.nombre + " " +          libro.precio + "$");
          System.out.println("Tasa de Descuento: " + tasaDescuento);
          System.out.println("Tasa de Impuesto: " + tasaImpuesto);
          System.out.println("Total: " + total);
    }
     public void guardarArchivo(String nombreArchivo) {
    // Crea un archivo con el nombre dado y escribe la factura.
    }
}
// imprime factura y guarda archivo no cumplen el principio (S)
```

```
public class FacturaImpresion {
    private Factura factura;

    public FacturaImpresion(Factura factura) {
        this.factura = factura;
    }
    public void imprimir() {
        System.out.println(factura.cantidad + "x " + factura.libro.nombre + " " + factura.libro.precio + " $");
        System.out.println("Tasa de Descuento: " + factura.tasaDescuento);
        System.out.println("Tasa de Impuesto: " + factura.tasaImpuesto);
        System.out.println("Total: " + factura.total + " $");
    }
}
```

```
public class FacturaPersistencia {
    Factura factura;

    public FacturaPersistencia(Factura factura) {
        this.factura = factura;
    }

    public void guardarArchivo(String nombreArchivo) {
        // Crea un archivo con el nombre dado y escribe la factura.
    }
}
```

Principio abierto/cerrado

No siempre es posible reutilizar el código directamente, porque las necesidades o las tecnologías subyacentes van cambiando, y nos vemos tentados a modificar ese código para adaptarlo a la nueva situación.

El principio abierto/cerrado nos dice que debemos evitar justamente eso y no tocar el código de las clases que ya está terminado.

Cuando creamos nuevas clases es importante tener en cuenta este principio para facilitar su extensión en un futuro.

ANÁLISIS Y PROGRAMACIÓN EN

Principio abierto cerrado (O)

```
public class FacturaPersistencia {
    Factura factura;
    public FacturaPersistencia(Factura factura) {
        this.factura = factura;
    }
    public void guardarArchivo(String nombreArchivo) {
        // Crea un archivo con el nombre dado y escribe la factura.
    }
    public void guardarEnBaseDatos() {
        // Guarda la factura en la base de datos
    }}
//refactorizamos
interface FacturaPersistencia {
    public void guardar(Factura factura);
}
public class BaseDeDatosPersistencia implements FacturaPersistencia {
    @Override
    public void guardar(Factura factura) {
        // Guardar en la base de datos
    }}
public class ArchivoPersistencia implements FacturaPersistencia {
    @Override
    public void guardar(Factura factura) {
        // Guardar en archivo
    }}
//ampliamos aplicación
public class AdministradorPersistencia {
    FacturaPersistencia facturaPersistencia;
    LibroPersistencia libroPersistencia;

    public AdministradorPersistencia(FacturaPersistencia facturaPersistencia, LibroPersistencia libroPersistencia) {
        this.facturaPersistencia = facturaPersistencia;
        this.libroPersistencia = libroPersistencia;
    }
}
```

Principio de sustitución de Liskov (L)

En una jerarquía de clases, las clases base y las subclases deben poder intercambiarse sin tener que alterar el código que las utiliza. Esto no quiere decir que tengan que hacer exactamente lo mismo, sino que han de poder reemplazarse.

El reverso de este principio es que no debemos extender clases mediante herencia por el hecho de aprovechar código de las clases bases o por conseguir forzar que una clase sea una “hija de” y superar un ***type hinting*** si no existe una relación que justifique la herencia (ser clases con el mismo tipo de comportamiento, pero que lo realizan de manera diferente). En ese caso, es preferible basar el polimorfismo en una interfaz (ver el Principio de segregación de interfaces)

Principio de sustitución de Liskov (L)

```
class Rectangulo {
	protected int ancho, alto;
	public Rectangulo() {
	}
	public Rectangulo(int ancho, int alto) {
		this.ancho = ancho;
		this.alto = alto;
	}
	public int getAncho() {
		return ancho;
	}
	public void setAncho(int ancho) {
		this.ancho = ancho;
	}
	public int getAlto() {
		return alto;
	}
	public void setAlto(int alto) {
		this.alto = alto;
	}
	public int getArea() {
		return ancho * alto;
	}}
class Cuadrado extends Rectangulo {
	public Cuadrado() {}
	public Cuadrado(int talla) {
		ancho = alto = talla;}
	@Override
	public void setAncho(int ancho) {
		super.setAncho(ancho);
		super.setAlto(ancho);}
	@Override
	public void setAlto(int alto) {
		super.setAlto(alto);
		super.setAncho(alto);}
}
```

```
class Test {

   static void getAreaTest(Rectangulo r) {
      int ancho = r.getAncho();
      r.setAlto(10);
      System.out.println("Area esperada de " + (ancho * 10) + ", tiene
" + r.getArea());
   }

   public static void main(String[] args) {
      Rectangulo rc = new Rectangulo(2, 3);
      getAreaTest(rc);

      Rectangulo sq = new Cuadrado();
      sq.setAncho(5);
      getAreaTest(sq);
   }
}
```

creamos un rectángulo donde el ancho es 2 y la altura es 3 y llamamos a **getAreaTest**. La salida es 20 como se esperaba, pero las cosas salen mal cuando pasamos en la plaza. Esto se debe a que la llamada a la función **setAlto** en la prueba también establece el ancho y da como resultado un resultado inesperado.

Principio de segregación de interfaces

El principio de segregación de interfaces puede definirse diciendo que una clase no debería verse obligada a depender de métodos o propiedades que no necesita.
Una forma sencilla de verlo es decir que las interfaces se han de definir a partir de las necesidades de la clase cliente.

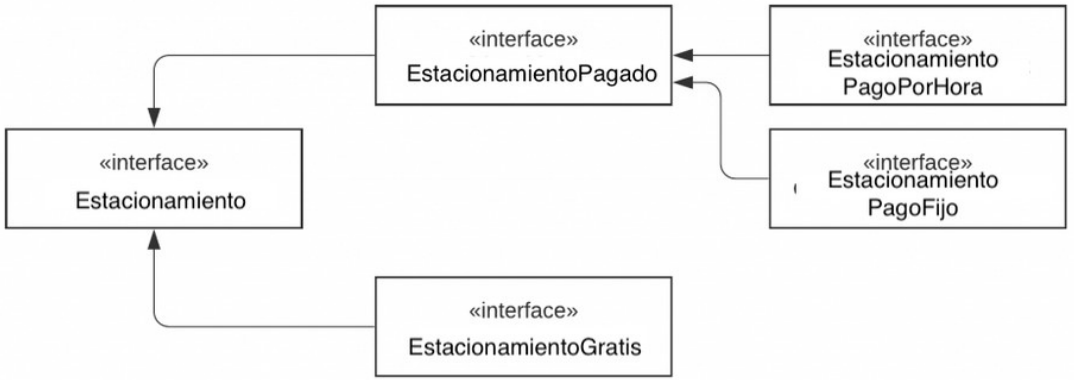
El principio establece que muchas interfaces específicas del cliente son mejores que una interfaz de propósito general. No se debe obligar a los clientes a implementar una función que no necesitan.

13.11.2025
Enrique Krause Buedo

Principio de segregación de interfaces (I)

```
public interface Estacionamiento {

	void aparcarCoche(); // Reducir el recuento de puntos
vacíos en 1

	void sacarCoche(); // Aumenta los espacios vacíos en 1

	void getCapacidad(); // Devuelve la capacidad del
coche

	double calcularTarifa(Coche coche); // Devuelve el
precio en función del número de horas.

	void hacerPago(Coche coche);

}

class Coche {

}

public class EstacionamientoGratis implements
Estacionamiento {

	@Override

	public void aparcarCoche() {

	}

	@Override

	public void sacarCoche() {

	}

	@Override

	public void getCapacidad() {

	}

	@Override

	public double calcularTarifa(Coche coche) {

		return 0;

	}

	@Override

	public void hacerPago(Coche coche) {

		throw new Exception("Estacionamiento es
gratis");

	}

}}
```

EstacionamientoGratis se vio obligada a implementar métodos relacionados con el pago que son irrelevantes. Separemos o segreguemos las interfaces:



Principio de inversión de dependencia (D)

El principio de inversión de dependencia dice que:

• Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.

• Las abstracciones no deben depender de detalles, son los detalles los que deben depender de abstracciones.

Las abstracciones definen conceptos que son estables en el tiempo, mientras que los detalles de implementación pueden cambiar con frecuencia. Una interfaz es una abstracción, pero una clase que la implemente de forma concreta es un detalle. Por tanto, cuando una clase necesita usar otra, debemos establecer la dependencia de una interfaz, o lo que es lo mismo, el type hinting indica una interfaz no una clase concreta. De ese modo, podremos cambiar la implementación (el detalle)

Principio de inversión de dependencia (D)

```
class EmailSender {
    public void send(String message) {
        System.out.println("Enviando correo con: " + message);
    }
}
class User {
    private EmailSender emailSender; // Dependencia directa
    public User() {
        this.emailSender = new EmailSender(); // Creación de la instancia
    }
    public void sendMessage(String message) {
        this.emailSender.send(message);
    }}}
```

Aplicando el principio (dependencia de abstracciones)

Ahora, se crea una interfaz MessageSender y User depende de ella. La implementación concreta se inyecta, haciendo el código desacoplado y flexible.

```
// Abstracción (interfaz)
interface MessageSender {
    void sendMessage(String message);
}
// Módulo de bajo nivel (concreción)
class EmailSender implements MessageSender {
    @Override
    public void sendMessage(String message) {
        System.out.println("Enviando correo con: " + message);
    }
}
// Módulo de bajo nivel (otra concreción)
class SMSSender implements MessageSender {
    @Override
    public void sendMessage(String message) {
        System.out.println("Enviando SMS con: " + message);
    }
}
```

Principio de inversión de dependencia (D)

```
// Módulo de alto nivel (depende de la abstracción)
class User {
    private MessageSender messageSender; // Dependencia de la interfaz

    // Inyección de dependencias a través del constructor
    public User(MessageSender messageSender) {
        this.messageSender = messageSender;
    }

    public void performSendMessage(String message) {
        this.messageSender.sendMessage(message);
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        // Caso 1: Inyectando EmailSender
        MessageSender emailSender = new EmailSender();
        User user1 = new User(emailSender);
        user1.performSendMessage("Hola, este es un correo.");

        // Caso 2: Inyectando SMSSender
        MessageSender smsSender = new SMSSender();
        User user2 = new User(smsSender);
        user2.performSendMessage("Hola, este es un SMS.");
    }
}
```

3 Algunos patrones de diseño

Patrones de diseño

En 1994, con la publicación del libro ***Design Patterns: Elements of Reusable Object-Oriented Software,*** con la “banda de los cuatro” como autores, se definieron los patrones de diseño como su título indica: como elementos de software reutilizables en orientación a objetos.
Se trata de una suerte de plantillas utilizables en cualquier lenguaje orientado a objetos, con efectividad y reusabilidad muy comprobadas.
Aportan soluciones a problemas frecuentes en ingeniería de software, y ayudan a estandarizar el Código. No deben usarse de modo forzado, pero si es recommendable siempre que se presentan los problemas que éstos resuelven.

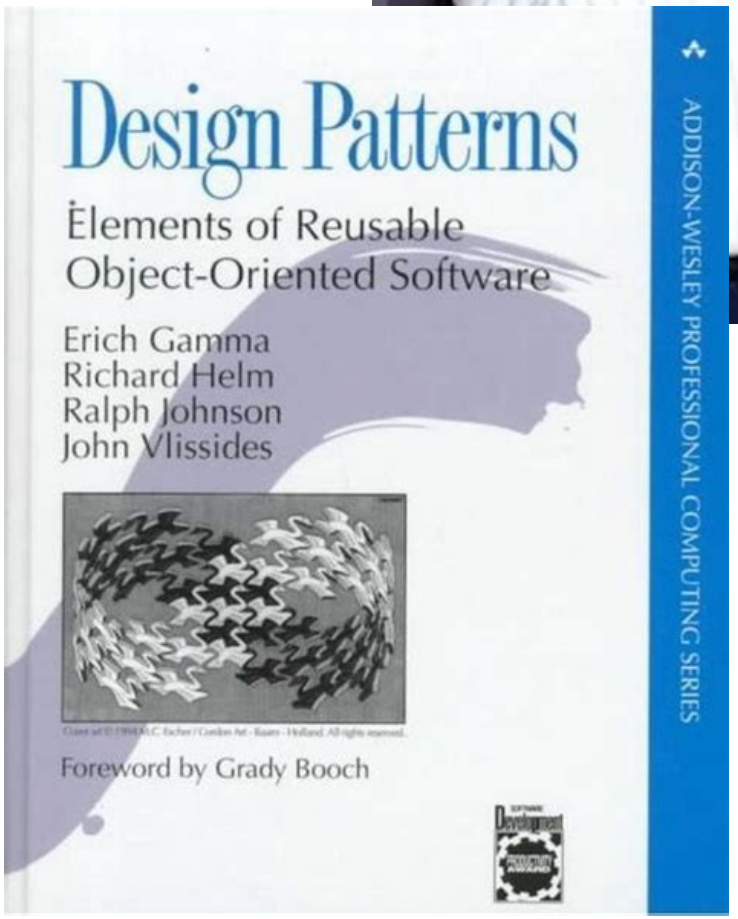
Existen patrones de tres tipos:

1- de creación o construcción.

2- de estructuración o estructurales.

3- Patrones de comportamiento

Patrones de diseño



By Purpose		Creational	Structural	Behavioral
By Scope	Class	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

Patrones de creación o construcción

Factory Method

El objetivo del patrón de diseño Factory Method es proveer un método abstracto de creación de un objeto delegando en las subclases concretas su creación efectiva. Dicho de otro modo: es un patrón de diseño de creación que proporciona una interfaz para crear objetos en una superclase, mientras permite a las subclases alterar el tipo de objetos que se crearán.

13.11.2025

Enrique Krause Buedo

Patrones de creación o construcción

Factory Method

Los participantes del patrón de diseño Factory Method, en un ejemplo Clientes y Pedidos, serían los siguientes:

CreadorAbstracto (AbstractCliente) es una clase abstracta que contiene la firma del método de fabricación y la implementación de métodos que invocan a este método de fabricación.

CreadorConcreto (ClienteContado, ClienteCredito) es una clase concreta que implementa el método de fabricación. Pueden existir varios creadores concretos.

Producto (AbstractPedido) es una clase abstracta que describe las propiedades comunes de los productos.

ProductoConcreto (PedidoContado, PedidoCredito) es una clase concreta que describe completamente un producto.

Patrones de creación o construcción

Factory Method

El patrón de diseño Factory Method se utiliza en los casos siguientes:

Una clase que solo conoce las clases abstractas de los objetos con los que tiene relaciones.

Una clase quiere delegar en sus subclases las elecciones de instanciación apoyándose en el mecanismo del polimorfismo.

Es el método más utilizado cuando no se conocen de antemano las dependencias y los tipos exactos de los objetos con los que deba funcionar tu código.

Con esto resulta más fácil extender el código de construcción de producto de forma independiente al resto del código.

Patrones de creación o construcción

Factory Method

Cuando trabajamos con frameworks que implementan menos clases y métodos de los que necesitamos, esta solución reduce el código que construye componentes en todo el framework a un único patrón Factory Method y permitir que cualquiera sobrescriba este método además de extender el propio componente. A menudo experimentas esta necesidad cuando trabajas con objetos grandes y que consumen muchos recursos, como conexiones de bases de datos, sistemas de archivos y recursos de red.

Todos los objetos siguen la misma interfaz. Esta interfaz deberá declarar métodos que tengan sentido en todos los productos.

Al añadir un patrón Factory Method vacío dentro de la clase creadora. El tipo de retorno del método deberá coincidir con la interfaz común de los productos.

Encuentra todas las referencias a constructores de producto en el código

Patrones de creación o construcción **Factory Method**

Ejemplos de uso: El patrón Factory Method se utiliza mucho en el código Java. Resulta muy útil cuando necesitas proporcionar un alto nivel de flexibilidad a tu código. El patrón está presente en las principales bibliotecas de Java:

java.util.Calendar#getInstance()

java.util.ResourceBundle#getBundle()

java.text.NumberFormat#getInstance()

java.nio.charset.Charset#forName()

java.net.URLStreamHandlerFactory#createURLStreamHandler(String) (Devuelve distintos objetos singleton, dependiendo de un protocolo).

java.util.EnumSet#of()

javax.xml.bind.JAXBContext#createMarshaller()

Factory Method

```
// Si nos fijamos la clase lo único que hace es instanciar un objeto u otro dependiendo

// del tipo que le solicitemos.Eso en un principio parece poco práctico. Pero vamos a ver

}}

// Nos podemos dar cuenta que el programador ya solo tiene que tratar con el concepto de
```

Patrones de creación o construcción

Singleton

Es un patrón de diseño de creación que garantiza que tan solo exista un objeto de su tipo y proporciona un único punto de acceso a él para cualquier otro código.

El patrón de diseño Singleton tiene como objetivo asegurar que una clase solo posee una instancia en memoria y proporcionar un método de clase que devuelva esta instancia única.

13.11.2025

Enrique Krause Buedo

Patrones de creación o construcción

Singleton

En ciertos casos puede ser útil gestionar clases que posean una única instancia. En el marco de los patrones de diseño de construcción, podemos citar el caso de la fábrica de productos (patrón de diseño Abstract Factory) de la que solo es necesario crear una instancia.

El patrón Singleton se puede reconocer por un método de creación estático, que devuelve el mismo objeto guardado en caché.

El patrón **Singleton** es famoso por limitar la reutilización de código y complicar las pruebas de unidad. No obstante, sigue resultando muy útil en algunos casos. En particular, viene bien cuando debes controlar recursos compartidos. Hacer uso de él, es inevitable, cuando deseas acceso a un archivo de registros que

Patrones de creación o construcción

Singleton

Cada cliente de la clase Singleton accede a la instancia única mediante el método de clase getInstance. No puede crear nuevas instancias utilizando el operador habitual de instanciación (operador new), al que ya no se puede acceder porque ahora su visibilidad es privada. Así mismo, el método mágico __clone también puede tener un modo de acceso privado para impedir que el código cliente esquive el bloqueo establecido por el constructor intentando crear clones del singleton. Aquí, hemos optado por dejarlo público, pero generando una excepción para indicar que este método no puede invocarse.

Factory Method

```
public final class Singleton {
    private static Singleton instance;
    public String value;

    private Singleton(String value) {
        // The following code emulates slow initialization.
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        this.value = value;
    }

    public static Singleton getInstance(String value) {
        if (instance == null) {
            instance = new Singleton(value);
        }
        return instance;
    }
```

```
public class DemoMultiThread {
public static void main(String[] args) {
System.out.println("If you see the same value, then
singleton was reused (yay!)" + "\n" +
"If you see different values, then 2 singletons were created
(booo!!)" + "\n\n" +
"RESULT:" + "\n");
Thread threadFoo = new Thread(new ThreadFoo());
Thread threadBar = new Thread(new ThreadBar());
threadFoo.start();
threadBar.start();
}
static class ThreadFoo implements Runnable {
@Override
public void run() {
Singleton singleton = Singleton.getInstance("FOO");
System.out.println(singleton.value);
}
}
static class ThreadBar implements Runnable {
@Override
public void run() {
Singleton singleton = Singleton.getInstance("BAR");
System.out.println(singleton.value);
}
}
}
```

Al tratarse de diferentes hilos la segunda llamada a getInstance se hace sin poder conocer si en instance hay un null o no, con lo que crea otra instancia.
En la página siguiente se ve una posible solución a este problema

Patrones de creación o construcción

Factory Method

```
public final class Singleton1 {
        // El campo debe declararse como volátil para que el bloqueo
de doble verificación funcione
    // correctamente. modificar un atributo con la palabra clave
volatile significa que se está
        //indicando a la máquina virtual de Java (JVM) y al
compilador que el valor de esta variable
        //será compartido entre diferentes hilos y que su valor
puede cambiar en cualquier momento
        // lectura o escritura de esa variable se realice
directamente en la memoria principal, evitando
        // la caché del CPU de cada hilo
    private static volatile Singleton1 instance;
    public String value;
    private Singleton1(String value) {
        this.value = value;
    }
    public static Singleton1 getInstance(String value) {
          // El enfoque adoptado aquí se denomina bloqueo de doble
verificación (DCL). Su
        // finalidad es evitar la condición de carrera entre varios
subprocesos que pueden
        // intentar obtener una instancia única al mismo tiempo,
creando así
        // instancias separadas.
        // Puede parecer que tener la variable `result` aquí es
completamente
        // inútil. Sin embargo, hay una advertencia muy importante
al
        // implementar el bloqueo de doble verificación en Java, que
se resuelve
        // introduciendo esta variable local.
        Singleton1 result = instance;
        if (result != null) {
            return result;
        }
        synchronized(Singleton1.class) {
            if (instance == null) {
                instance = new Singleton1(value);
            }
            return instance;
        }
    }
```

13.11.2025

Enrique Krause Buedo

Patrones de estructura

Proxy

El patrón de diseño Proxy tiene como objetivo el diseño de un objeto que sustituye a otro y controla su acceso.

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

Muy utilizado, por ejemplo, cuando la clase es parte de una biblioteca cerrada de un tercero (componentes).

Patrones de estructura **Proxy**

El patrón Proxy sugiere que crees una nueva clase proxy con la misma interfaz que un objeto de servicio original. Después actualizas tu aplicación para que pase el objeto proxy a todos los clientes del objeto original. Al recibir una solicitud de un cliente, el proxy crea un objeto de servicio real y le delega todo el trabajo.

Ya que implementa la misma interfaz que la clase original, puede pasarse a cualquier cliente que espere un objeto de servicio real. De este modo, por ejemplo, podríamos reemplazar un objeto pesado, en una visualización que no necesite todos los detalles, por uno ligero

Patrones de estructura Proxy

```
private void connectToServer(String server) {
        System.out.print("Connecting to " + server + "... ");
        experienceNetworkLatency();
        System.out.print("Connected!" + "\n");
    }
    private HashMap<String, Video> getRandomVideos() {
        System.out.print("Downloading populars... ");

        experienceNetworkLatency();
        HashMap<String, Video> hmap = new HashMap<String, Video>();
        hmap.put("catzzzzzzzzz", new Video("sadgahasgdas", "Catzzzz.avi"));
        hmap.put("mkafksangasj", new Video("mkafksangasj", "Dog play with
ball.mp4"));
        hmap.put("dancesvideoo", new Video("asdfas3ffasd", "Dancing
video.mpq"));
        hmap.put("dlsdk5jfslaf", new Video("dlsdk5jfslaf", "Barcelona vs
RealM.mov"));
        hmap.put("3sdfgsd1j333", new Video("3sdfgsd1j333", "Programing
lesson#1.avi"));

        System.out.print("Done!" + "\n");
        return hmap;
    }
    private Video getSomeVideo(String videoId) {
        System.out.print("Downloading video... ");
        experienceNetworkLatency();
        Video video = new Video(videoId, "Some video title");
        System.out.print("Done!" + "\n");
        return video;
    }
}
```

En este ejemplo, el patrón Proxy ayuda a implementar la inicialización diferida y el almacenamiento en caché a una ineficiente biblioteca de integración de YouTube de un tercero.
Proxy también es muy valioso cuando tienes que añadir comportamientos a una clase cuyo código no puedes cambiar

Patrones de estructura Proxy

```
public class Video {
    public String id;
    public String title;
    public String data;
    Video(String id, String title) {
        this.id = id;
        this.title = title;
        this.data = "Random video.";
    }
}
-------------------------------------------------------
public class YouTubeCacheProxy implements ThirdPartyYouTubeLib {
    private ThirdPartyYouTubeLib youtubeService;
    private HashMap<String, Video> cachePopular = new HashMap<String, Video>();
    private HashMap<String, Video> cacheAll = new HashMap<String, Video>();

    public YouTubeCacheProxy() {
        this.youtubeService = new ThirdPartyYouTubeClass();
    }
    @Override
    public HashMap<String, Video> popularVideos() {
        if (cachePopular.isEmpty()) {
            cachePopular = youtubeService.popularVideos();
        } else {
            System.out.println("Retrieved list from cache.");
        }
        return cachePopular;
    }
    @Override
    public Video getVideo(String videoId) {
        Video video = cacheAll.get(videoId);
        if (video == null) {
            video = youtubeService.getVideo(videoId);
            cacheAll.put(videoId, video);
        } else {
            System.out.println("Retrieved video '" + videoId + "'
from cache.");
        }
        return video;
    }
    public void reset() {
        cachePopular.clear();
        cacheAll.clear();
    }
}
```

Patrones de estructura Proxy

```
import java.util.HashMap;
public class YouTubeDownloader {
    private ThirdPartyYouTubeLib api;

    public YouTubeDownloader(ThirdPartyYouTubeLib api) {
        this.api = api;
    }
    public void renderVideoPage(String videoId) {
        Video video = api.getVideo(videoId);
        System.out.println("\n-------------------------------");
        System.out.println("Video page (imagine fancy HTML)");
        System.out.println("ID: " + video.id);
        System.out.println("Title: " + video.title);
        System.out.println("Video: " + video.data);
        System.out.println("-------------------------------\n");
    }
    public void renderPopularVideos() {
        HashMap<String, Video> list = api.popularVideos();
        System.out.println("\n-------------------------------");
        System.out.println("Most popular videos on YouTube (imagine fancy HTML)");
        for (Video video : list.values()) {
            System.out.println("ID: " + video.id + " / Title: " + video.title);
        }
        System.out.println("-------------------------------\n");
    }
}
```

```
import java.util.HashMap;

public class YouTubeDownloader {
    private ThirdPartyYouTubeLib api;

    public YouTubeDownloader(ThirdPartyYouTubeLib api) {
        this.api = api;
    }
    public void renderVideoPage(String videoId) {
        Video video = api.getVideo(videoId);
        System.out.println("\n-------------------------------");
        System.out.println("Video page (imagine fancy HTML)");
        System.out.println("ID: " + video.id);
        System.out.println("Title: " + video.title);
        System.out.println("Video: " + video.data);
        System.out.println("-------------------------------\n");
    }
    public void renderPopularVideos() {
        HashMap<String, Video> list = api.popularVideos();
        System.out.println("\n-------------------------------");
        System.out.println("Most popular videos on YouTube (imagine
fancy HTML)");
        for (Video video : list.values()) {
            System.out.println("ID: " + video.id + " / Title: " +
video.title);
        }
        System.out.println("-------------------------------\n");
    }
}
```

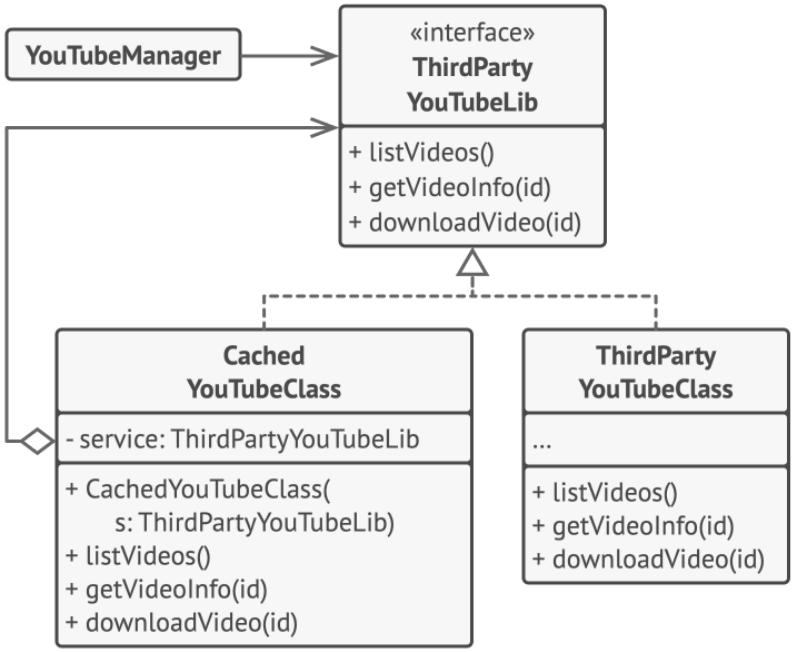
Patrones de estructura Proxy

```
import java.util.HashMap;
public class YouTubeDownloader {
    private ThirdPartyYouTubeLib api;

    public YouTubeDownloader(ThirdPartyYouTubeLib api) {
        this.api = api;
    }
    public void renderVideoPage(String videoId) {
        Video video = api.getVideo(videoId);
        System.out.println("\n-------------------------------");
        System.out.println("Video page (imagine fancy HTML)");
        System.out.println("ID: " + video.id);
        System.out.println("Title: " + video.title);
        System.out.println("Video: " + video.data);
        System.out.println("-------------------------------\n");
    }
    public void renderPopularVideos() {
        HashMap<String, Video> list = api.popularVideos();
        System.out.println("\n-------------------------------");
        System.out.println("Most popular videos on YouTube (imagine fancy HTML)");
        for (Video video : list.values()) {
            System.out.println("ID: " + video.id + " / Title: " + video.title);
        }
        System.out.println("-------------------------------\n");
    }
}
```

```
import java.util.HashMap;

public class YouTubeDownloader {
    private ThirdPartyYouTubeLib api;

    public YouTubeDownloader(ThirdPartyYouTubeLib api) {
        this.api = api;
    }
    public void renderVideoPage(String videoId) {
        Video video = api.getVideo(videoId);
        System.out.println("\n-------------------------------");
        System.out.println("Video page (imagine fancy HTML)");
        System.out.println("ID: " + video.id);
        System.out.println("Title: " + video.title);
        System.out.println("Video: " + video.data);
        System.out.println("-------------------------------\n");
    }
    public void renderPopularVideos() {
        HashMap<String, Video> list = api.popularVideos();
        System.out.println("\n-------------------------------");
        System.out.println("Most popular videos on YouTube (imagine
fancy HTML)");
        for (Video video : list.values()) {
            System.out.println("ID: " + video.id + " / Title: " +
video.title);
        }
        System.out.println("-------------------------------\n");
    }
}
```

Patrones de estructura Proxy

```
public class Demo {

    public static void main(String[] args) {

        YouTubeDownloader naiveDownloader = new YouTubeDownloader(new
ThirdPartyYouTubeClass());

        YouTubeDownloader smartDownloader = new YouTubeDownloader(new YouTubeCacheProxy());

        long naive = test(naiveDownloader);

        long smart = test(smartDownloader);

        System.out.print("Time saved by caching proxy: " + (naive - smart) + "ms");


    }

    private static long test(YouTubeDownloader downloader) {

        long startTime = System.currentTimeMillis();


        // User behavior in our app:

        downloader.renderPopularVideos();

        downloader.renderVideoPage("catzzzzzzzzz");

        downloader.renderPopularVideos();

        downloader.renderVideoPage("dancesvideoo");

        // Users might visit the same page quite often.

        downloader.renderVideoPage("catzzzzzzzzz");

        downloader.renderVideoPage("someothervid");


        long estimatedTime = System.currentTimeMillis() - startTime;

        System.out.print("Time elapsed: " + estimatedTime + "ms\n");

        return estimatedTime;

    }
}
```



Patrones de estructura Facade

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

El objetivo del patrón de diseño **Facade** es agrupar las interfaces de un conjunto de objetos en una interfaz unificada volviendo a este conjunto más fácil de usar.

Imagina que debes lograr que tu código trabaje con un amplio grupo de objetos que pertenecen a una sofisticada biblioteca o *framework*.

Patrones de estructura **Facade**

Una fachada es una clase que proporciona una interfaz simple a un subsistema complejo que contiene muchas partes móviles

Tener una fachada resulta útil cuando tienes que integrar tu aplicación con una biblioteca sofisticada con decenas de funciones, de la cual sólo necesitas una pequeña parte. Puedes en él, inicializar objetos en el orden correcto y suministrarles datos en el formato adecuado.

Las clases del subsistema no conocen la existencia de la fachada. Operan dentro del sistema y trabajan entre sí directamente.

El **Cliente** utiliza la fachada en lugar de invocar directamente los objetos del subsistema.

Patrones de estructura Facade

```
public class Lighting {

    public void on() {
        System.out.println("Lights are on");
    }
    public void off() {
        System.out.println("Lights are off");
    }}
public class MusicSystem {
    public void playMusic() {
        System.out.println("Music is playing");
    }
    public void stopMusic() {
        System.out.println("Music is stopped");
    }}
public class ClimateControl {
    public void setTemperature(int temp) {
        System.out.println("Temperature set to " + temp + " degrees");
    }}
public class Demo {
public static void main(String[] args) {
        Lighting light= new Lighting();
        MusicSystem ms= new MusicSystem();
        ClimateControl cc= new ClimateControl();
        SmartHomeFacade shf= new SmartHomeFacade(light, ms, cc);
        shf.startEveningRoutine();
        System.out.println("_________________");
        shf.endEveningRoutine();
}}
```

```
public class SmartHomeFacade {

    private Lighting lighting;

    private MusicSystem musicSystem;

    private ClimateControl climateControl;


    public SmartHomeFacade(Lighting lighting, MusicSystem
musicSystem, ClimateControl climateControl) {
        this.lighting = lighting;

        this.musicSystem = musicSystem;

        this.climateControl = climateControl;

    }


    public void startEveningRoutine() {
        lighting.on();

        musicSystem.playMusic();

        climateControl.setTemperature(22);

    }


    public void endEveningRoutine() {
        lighting.off();

        musicSystem.stopMusic();

    }

}
```

Patrones de comportamiento Iterator

El patrón de diseño Iterator proporciona un acceso secuencial a una colección de objetos a los clientes, sin que estos tengan que preocuparse de la implementación de esta colección.

Dicho de otro modo; es un patrón de diseño de comportamiento que te permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.).

La misma colección puede recorrerse de varias formas diferentes. Añadir más y más algoritmos de recorrido a la colección puede nublar gradualmente su responsabilidad principal, que es el almacenamiento eficiente de la información.

Patrones de comportamiento **Iterator**

La idea central del patrón Iterator es extraer el comportamiento de recorrido de una colección y colocarlo en un objeto independiente llamado *iterador*.

Independientemente de cómo se estructure una colección, debe aportar una forma de acceder a sus elementos de modo que otro código pueda utilizarlos. Debe haber una forma de recorrer cada elemento de la colección sin acceder a los mismos elementos una y otra vez.

Además de implementar el propio algoritmo, un objeto iterador encapsula todos los detalles del recorrido, como la posición actual y cuántos elementos quedan hasta el final. Debido a esto, varios iteradores pueden recorrer la misma colección al mismo tiempo, independientemente los unos de los otros.

Patrones de comportamiento

Iterator

```
class Book {
    private String title;
    public Book(String title) {
        this.title = title;
    }
    public String getTitle() {
        return title;
    }}
public interface BookCollection {
    Iterator<Book> createIterator();
}
class BookIterator implements Iterator<Book> {
    private List<Book> books;
    private int position = 0;
public BookIterator(List<Book> books) {
        this.books = books;
    }
    @Override
    public boolean hasNext() {
        return position < books.size();
    }
    @Override
    public Book next() {
        return books.get(position++);
    }}
```

```
public interface Iterator<T> {
         boolean hasNext();
         T next();
       }
import java.util.ArrayList;
import java.util.List;

class Library implements BookCollection {
    private List<Book> books = new ArrayList<>();

    public void addBook(Book book) {
        books.add(book);
    }
    @Override
    public Iterator<Book> createIterator() {
        return new BookIterator(books);
    }}
public class IteratorPatternDemo {
        public static void main(String[] args) {
                Library library = new Library();
                library.addBook(new Book("Design Patterns"));
                library.addBook(new Book("Clean Code"));
                library.addBook(new Book("Effective Java"));
                Iterator<Book> iterator = library.createIterator();
                System.out.println("Books in the library:");
                while (iterator.hasNext()) {
                    System.out.println(iterator.next().getTitle());
                } }}
```

Patrones de comportamiento **Observer**

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

Es muy utilizado en el modelo MVC

Explicado mejor: El patrón de diseño Observer tiene como objetivo modelizar una dependencia entre un sujeto y los observadores de modo que cada modificación del estado interno del sujeto sea notificada a sus observadores para que puedan actualizar su estado.

Un cliente puede visitar la tienda cada día para comprobar la disponibilidad del producto. Pero, mientras el producto está en camino, la mayoría de estos viajes serán en vano.

Patrones de comportamiento **Observer**

El objeto que tiene un estado interesante suele denominarse *sujeto*, pero, como también va a notificar a otros objetos los cambios en su estado, le llamaremos *notificador*.

Cuando le sucede un evento importante al notificador, recorre sus suscriptores y llama al método de notificación específico de sus objetos.

Utiliza el patrón Observer cuando los cambios en el estado de un objeto puedan necesitar cambiar otros objetos y el grupo de objetos sea desconocido de antemano o cambie dinámicamente.

Puedes experimentar este problema a menudo al trabajar con clases de la interfaz gráfica de usuario.

Patrones de comportamiento **Observer**

Utiliza el patrón cuando algunos objetos de tu aplicación deban observar a otros, pero sólo durante un tiempo limitado o en casos específicos. La lista de suscripción es dinámica, por lo que los suscriptores pueden unirse o abandonar la lista cuando lo deseen.

La meta del patrón *Observer* es establecer conexiones dinámicas de un único sentido entre objetos, donde algunos objetos actúan como subordinados de otros.

Patrones de comportamiento

Observer

```
import java.util.*;
//Subject (Observable)
class WeatherStation {
 private List<Observer> observers = new ArrayList<>();
 private float temperature;
 public void addObserver(Observer observer) {
     observers.add(observer);
 }
 public void removeObserver(Observer observer) {
     observers.remove(observer);
 }
 public void setTemperature(float temperature) {
     this.temperature = temperature;
     notifyObservers();
 }
 private void notifyObservers() {
     for (Observer observer : observers) {
         observer.update(temperature);
     }}}

 //Observer
interface Observer {
 void update(float temperature);
}
```

```
//Concrete Observer
class CurrentConditionsDisplay implements Observer {
 private float temperature;
 @Override
 public void update(float temperature) {
     this.temperature = temperature;
     display();
 }
 private void display() {
     System.out.println("Current Conditions Display: Temperature = "
+ temperature);
 }}
//Concrete Observer
class StatisticsDisplay implements Observer {
 private float temperature;
 @Override
 public void update(float temperature) {
     this.temperature = temperature;
     display();
 }
 private void display() {
     System.out.println("Statistics Display: Temperature = " +
temperature);
 }
}
```

Patrones de comportamiento

Observer

```
//Main class
public class ObserverPatternExample {
 public static void main(String[] args) {
     WeatherStation weatherStation = new WeatherStation();
     CurrentConditionsDisplay currentConditionsDisplay = new CurrentConditionsDisplay();
     StatisticsDisplay statisticsDisplay = new StatisticsDisplay();
     weatherStation.addObserver(currentConditionsDisplay);
     weatherStation.addObserver(statisticsDisplay);
     // Simulate a change in temperature
     weatherStation.setTemperature(25.5f);
     // Output:
     // Current Conditions Display: Temperature = 25.5
     // Statistics Display: Temperature = 25.5
     // Simulate another change in temperature
     weatherStation.setTemperature(30.0f);
     // Output:
     // Current Conditions Display: Temperature = 30.0
     // Statistics Display: Temperature = 30.0
     // Remove an observer
     weatherStation.removeObserver(currentConditionsDisplay);
     // Simulate another change in temperature
     weatherStation.setTemperature(28.0f);
     // Output:
     // Statistics Display: Temperature = 28.0
 }
}
```

```
Current Conditions Display: Temperature = 25.5
Statistics Display: Temperature = 25.5
Current Conditions Display: Temperature = 30.0
Statistics Display: Temperature = 30.0
Statistics Display: Temperature = 28.0
```

Gracias

13.11.2025

Enrique Krause Buedo