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.

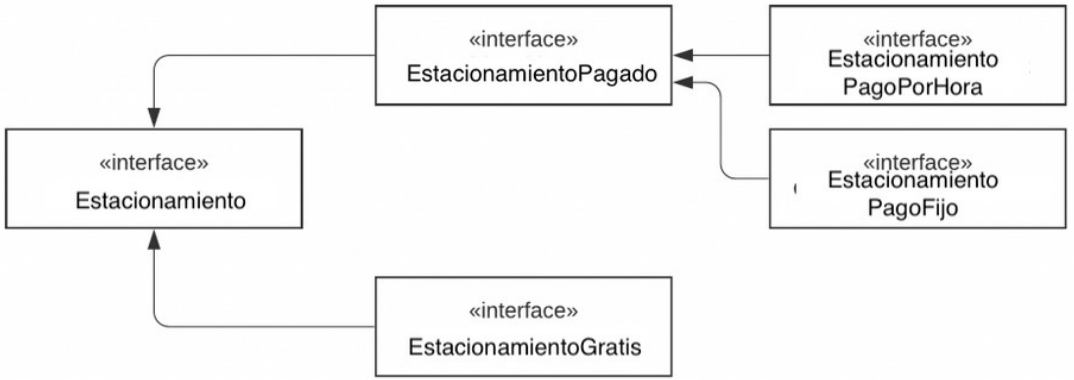
El principio establece que muchas interfaces específicas del cliente son mejores que una interfaz de propósito general. No se debe obligar a los clientes a implementar una función que no necesitan.

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.");
    }
}
```