Módulo 1. Principios SOLID en Java

CÓRDOBA CREA FUTURO

Los principios SOLID son un conjunto de cinco directrices fundamentales en la programación orientada a objetos (POO) que, aplicados en lenguajes como Java, ayudan a crear software más mantenible, flexible y escalable. El acrónimo SOLID representa cada uno de los cinco principios.

1. **S** por *single responsibility principle* (SRP) o principio de responsabilidad única.
2. **O** por *open/closed principle* (OCP) o principio abierto/cerrado.
3. **L** por *Liskov substitution principle* (LSP) o principio de sustitución de Liskov.
4. **I** por *interface segregation principle* (ISP) o principio de segregación de interfaces.
5. **D** por *dependency inversion principle* (DIP) o principio de inversión de dependencias.

Beneficios de aplicar SOLID

- Código más limpio y legible: facilita la comprensión.

- Mayor mantenibilidad: los cambios son menos propensos a generar errores.
- Mayor flexibilidad y escalabilidad: permite añadir funcionalidades sin rehacer todo el sistema.
- Menor acoplamiento y mayor cohesión: las partes del sistema están más independientes.

1. Interfaces

Las interfaces están muy relacionadas con el polimorfismo. En relación con este aspecto, cabe recordar que los objetos de clases diferentes pueden tener el mismo comportamiento (o sea, los mismos métodos), pero se pueden aplicar de diferentes maneras. Por ejemplo, cuando decimos bailar, cada persona podrá bailar, pero de diferentes formas. Dicho de otra manera, son varias formas de hacer lo mismo.

Sin embargo, este polimorfismo trae el riesgo de tener código inconsistente: la forma de bailar de una persona con una estructura de código, nombre y argumentos es diferente a la de otra. Además, si luego creamos otro objeto de persona que también baila, crearemos otra estructura para que baile, lo que complicaría cada vez más el código.

Para evitarlo, usamos las interfaces, que son contratos que definen el nombre de los métodos y los argumentos que

reciben; sin embargo, estas no especifican cómo implementar el método. Ese es el problema de cada clase. Un ejemplo sencillo es el volante de un auto. Si se gira a la derecha, el auto gira a la derecha. Cada fabricante de autos puede tener su propia tecnología de conducción, pero todos deben cumplir el contrato de darle al conductor un volante.

Tabla 1: Principios SOLID y rol de las interfaces

Principio SOLID	Rol de las interfaces
SRP	Separan responsabilidades.
OCP	Permiten extender sin modificar.
LSP	Definen contratos sustituibles.
ISP	Deben ser pequeñas y específicas.
DIP	Invierten dependencias hacia abstracciones.

Fuente: elaboración propia.

En el caso de los bailarines, creamos una interfaz Bailador que tanto el que baila cuarteto como el que baila vals deben

respetar. Como en el ejemplo del volante del auto, a la interfaz no le interesa cómo baila cuarteto ni cómo baila vals, solo que se respete el contrato.

Cuatro de estos principios fueron desarrollados por Robert C. Martin, ingeniero de software que recibió el apodo cariñoso de «tío Bob» (uncle Bob) por parte de la comunidad de desarrollo de software, por su figura de mentor y guía. Por otra parte, el tercer principio fue desarrollado por Bárbara Liskov, una reconocida ingeniera de software norteamericana.

Figura 1: Robert Martin y Bárbara Liskov

Fuente: [imagen sin título sobre Robert Martin y Bárbara Liskov], s. f.,

https://bit.ly/3MRXM6T; https://bit.ly/4js65CV.

En Java, la relación entre los principios SOLID y el uso de interfaces es estructural y fundamental. Las interfaces no son solo una característica del lenguaje: son uno de los principales mecanismos técnicos para aplicar SOLID de forma correcta y mantenible.

CONTINUAR

2. Patrones comunes que combinan SOLID e interfaces

Los principios SOLID pueden combinarse, y esas combinaciones —que se detallarán en otro apartado más adelante— reciben los nombres que se describen en los próximos párrafos.

Strategy pattern —

ISP + OCP es un patrón de diseño de comportamiento que convierte un grupo de comportamientos en objetos y los hace intercambiables dentro del objeto de contexto original. El objeto original, llamado contexto, contiene una referencia a un objeto de estrategia y le delega la ejecución del comportamiento.

Adapter pattern —

DIP + OCP es un patrón de diseño estructural que permite que dos interfaces incompatibles colaboren y que actúen como un «traductor» o «puente» entre ellas sin modificar el código original de

ninguna, es decir, el patrón Adapter actúa como envoltorio entre dos objetos, atrapa las llamadas a un objeto y las transforma a un formato y una interfaz reconocible para el segundo objeto, similar a la forma en la que un cargador móvil adapta el voltaje de la pared a tu dispositivo; crea una clase intermedia (el adaptador) que envuelve una clase existente, y expone una interfaz que el cliente espera, lo que hace que funcionen juntas, ideal para integrar sistemas heredados o librerías de terceros.

Decorator pattern —

OCP + SRP es un patrón de diseño estructural que permite estructurar la lógica de negocio en capas, crear un decorador para cada capa y componer objetos con varias combinaciones de esta lógica durante el tiempo de ejecución, es decir, permite añadir dinámicamente nuevas responsabilidades o funcionalidades a un objeto y lo envuelve con objetos «decoradores» sin modificar el código de la clase original, lo que promueve la composición sobre la herencia y evita la «explosión de clases» que resulta de muchas subclases.

Factory pattern —

DIP + OCP es un patrón de creación que define una interfaz para crear un objeto, pero permite que las subclases modifiquen el tipo de objetos que se crearán. Este patrón promueve la flexibilidad y la escalabilidad del código base.

Observer pattern

DIP + ISP es un patrón de diseño de comportamiento que permite a un objeto notificar a otros objetos sobre cambios en su estado. El patrón Observer proporciona una forma de suscribirse y cancelar la subscripción a estos eventos para cualquier objeto que aplica una interfaz suscriptora, es decir, es un patrón de diseño de comportamiento que define una dependencia de uno a muchos, lo que permite que un objeto (el sujeto o publisher) notifique automáticamente a otros objetos (los observadores o subscriptores) cuando su estado cambia, sin que el sujeto necesite conocer los detalles de los observadores. Esto promueve la modularidad y el desacoplamiento. Los observadores se registran (se suscriben) al sujeto y este les informa (métodos attach/detach/notify), lo cual se actualiza de forma reactiva.

Las interfaces en Java son la herramienta principal para aplicar SOLID por los siguientes motivos:

- definen contratos claros (LSP).
- Permiten segregar responsabilidades (ISP, SRP).
- Facilitan la extensión (OCP).
- Invierten dependencias (DIP).

- Promueven el bajo acoplamiento y alta cohesión.

Las interfaces no son solo para definir tipos, sino que son el mecanismo fundamental para lograr código flexible, mantenible y escalable siguiendo principios SOLID.

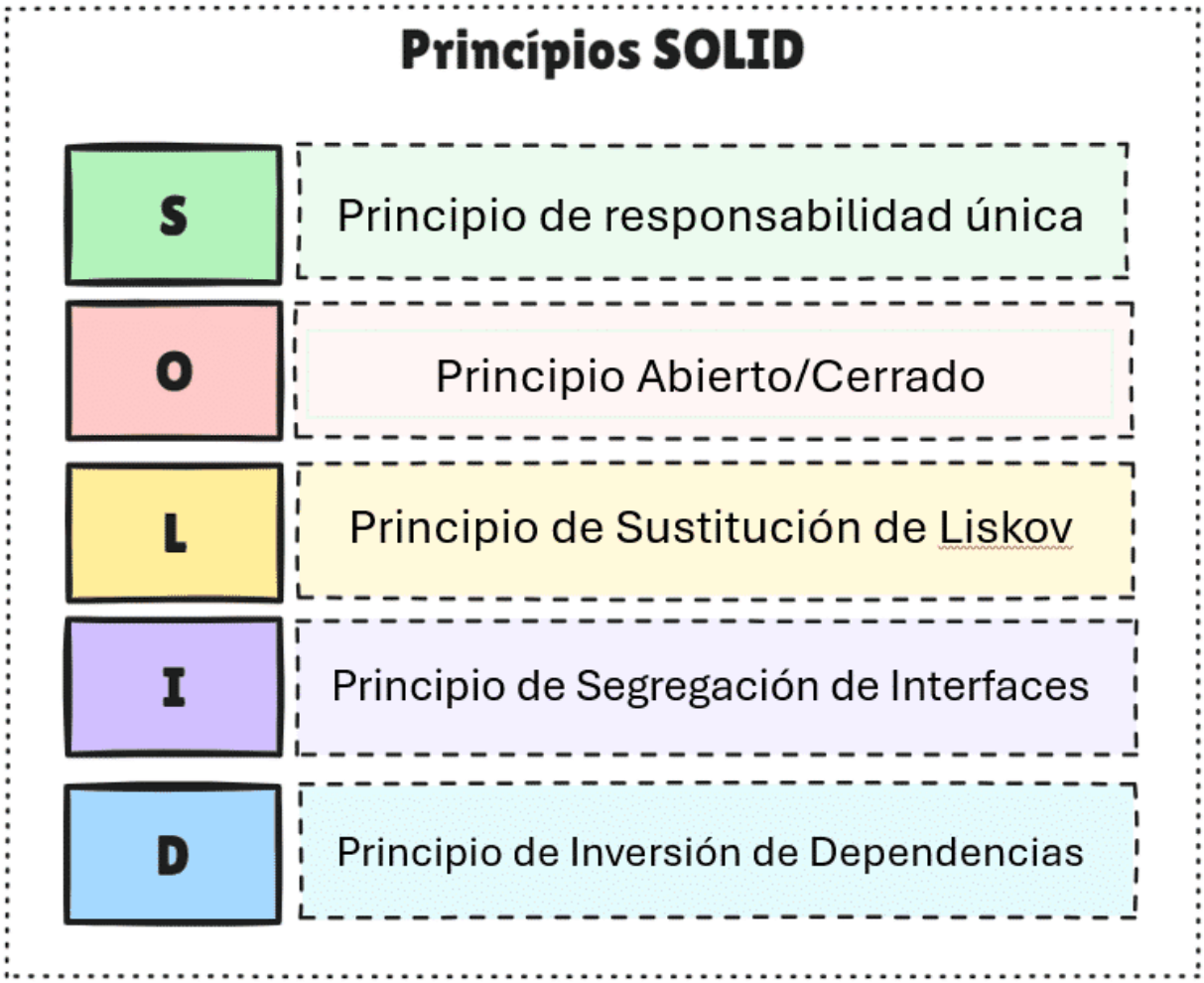
CONTINUAR

3. Descripción de cada principio

Se describe seguidamente cada uno de los principios mencionados con ejemplos claros que le ayudarán al estudiante a comprender y aplicar estos conceptos. En el mundo profesional, los desarrolladores de software emplean como un pilar estos principios que tienen incorporados como algo habitual en sus desarrollos.

En la siguiente imagen, se resumen los principios que se desarrollarán.

Figura 2: Principios SOLID



Fuente: elaboración propia.

Principio de responsabilidad única (S)

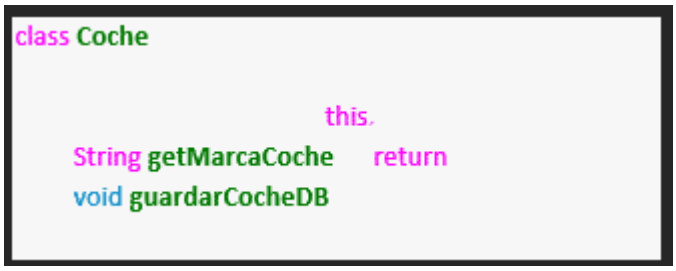
El primer principio, que en inglés es single responsibility principle (SRP) o principio de responsabilidad única en español, establece que una clase debe tener una, y solamente una, razón para cambiar (según Robert C. Martin). Esto significa que una clase debe tener una única responsabilidad o tarea específica que llevar a cabo. Esto no significa lo siguiente:

- «una sola función».
- «Una sola línea de código».
- «Una sola responsabilidad técnica».

Explicación

Como su propio nombre indica, establece que una clase, componente o microservicio debe ser responsable de una sola cosa (el tan aclamado término decoupled en inglés). Si, por el contrario, una clase tiene varias responsabilidades, esto implica que el cambio en una responsabilidad provocará la modificación en otra responsabilidad.

Considera este ejemplo:

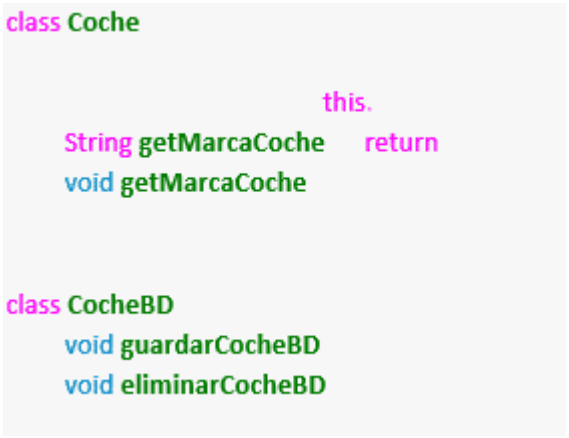


¿Por qué este código viola el principio de responsabilidad única?

Como podemos observar, la clase Coche permite tanto acceder a las propiedades de la clase como llevar a cabo operaciones sobre la BBDD (base de datos), por lo que la clase ya tiene más de una responsabilidad.

Supongamos que debemos hacer cambios en los métodos que llevan a cabo las operaciones a la BBDD. En este caso, además de estos cambios, probablemente tendríamos que tocar los nombres o tipos de las propiedades, métodos, etcétera, cosa que no parece muy eficiente porque solo estaríamos modificando cosas que tienen que ver con la BBDD, ¿verdad?

Para evitar esto, debemos separar las responsabilidades de la clase, por lo que podemos crear otra clase que se encargue de las operaciones a la BBDD.



Refactorización hacia SRP

Seguidamente, se deja un ejemplo completo de refactorización, es decir, transformar cambiando el código que no cumple con SRP y a uno que sí mediante los pasos que se describen a continuación.

Tabla 2: Código que viola el principio SRP. Pequeño código que simula la validación de ingreso de datos

class UsuarioManager {	// Responsabilidad 3: Notificación
private Usuario usuario;	public void enviarEmailBienvenida() {

```
  // Responsabilidad 1: Validación
  public boolean validarDatos() {
      if (usuario.getEmail() == null) return false;
      if (usuario.getPassword().length() < 8) return false;
      return true;
  }

  // Responsabilidad 2: Persistencia
  public void guardarEnBD() {
      Connection conn = DriverManager.getConnection(...);
      PreparedStatement stmt = conn.prepareStatement(...);
      EmailService.send(usuario.getEmail(), "Bienvenido!");
  }

  // Responsabilidad 4: Logging
  public void registrarActividad() {
      Logger.log("Usuario creado: " + usuario.getEmail());
  }
}
```

// ... código JDBC }	

Fuente: elaboración propia.

Tabla 3: Código que cumple el principio SRP. Refactorizando

// Responsabilidad 1: Entidad de negocio
 	
class Usuario {
 	
private String email;
 	
private String password;
 	
// Getters y setters
 	
}
 	
// Responsabilidad 2: Validación
 	
class ValidadorUsuario {
	/// Responsabilidad 5: Logging
	
	interface LoggerActividad {
	
	void registrar(String actividad, Usuario usuario);
	
	}
	
	class LoggerActividadArchivo implements LoggerActividad {
	
	public void registrar(String actividad, Usuario usuario) {
	
	// Lógica de logging a archivo
	
	}

```
        public boolean validar(Usuario usuario) {

        if (usuario.getEmail() == null) return false;

        if (usuario.getPassword() == null || usuario.getPassword().length() < 8) {

            return false;

        }

        return true;

    }

}

// Responsabilidad 3: Persistencia

interface UsuarioRepositorio {

        void guardar(Usuario usuario);

}

// Responsabilidad 6: Coordinación (Orquestador)

class UsuarioService {

    private ValidadorUsuario validador;

    private UsuarioRepositorio repositorio;

    private Notificador notificador;

    private LoggerActividad logger;



    public UsuarioService(ValidadorUsuario v, UsuarioRepositorio r,

                          Notificador n, LoggerActividad l) {

        this.validador = v;

        this.repositorio = r;

        this.notificador = n;

        this.logger = l;
```

```
class UsuarioRepositorioMySQL implements UsuarioRepositorio {

    public void guardar(Usuario usuario) {

        // Lógica específica de MySQL

    }

}

// Responsabilidad 4: Notificación

interface Notificador {

    void enviarBienvenida(Usuario usuario);

}

class EmailNotificador implements Notificador {

    public void enviarBienvenida(Usuario usuario) {

    }

}

class RegistroUsuario {

    public void crearUsuario(Usuario usuario) {

        if (!validador.validar(usuario)) {

            throw new IllegalArgumentException("Usuario inválido");

        }

        repositorio.guardar(usuario);

        notificador.enviarBienvenida(usuario);

        logger.registrar("Usuario creado", usuario);

    }

}
```

```
usuario) {

      // Lógica de envío de email

  }

}
```

Fuente: elaboración propia.

Formas de identificar violaciones de SRP en código existente

Señales de alarma

1. Clases muy largas (más de quinientas líneas).
2. Muchos imports de diferentes dominios.
3. Métodos con muchos parámetros no relacionados.
4. Cambios frecuentes en la misma clase por diferentes razones.

5 Dificultad para nombrar la clase claramente.

Conclusión

El principio de responsabilidad única es el más importante de SOLID debido a los siguientes motivos:

1. es la base para los otros principios (especialmente OCP).
2. Reduce el acoplamiento entre componentes.
3. Aumenta la cohesión dentro de cada componente.
4. Facilita las pruebas unitarias.
5. Mejora la mantenibilidad a largo plazo.

SRP no significa «una clase por función», sino «una responsabilidad por clase». Una responsabilidad es un eje de cambio si algo puede cambiar por múltiples razones independientes, necesita separación.

La clave está en encontrar el balance correcto entre los siguientes aspectos:

- cohesión interna (elementos relacionados juntos).
- Separación de preocupaciones (responsabilidades distintas separadas).

Cuando SRP se aplica correctamente, el código se vuelve más flexible, más comprensible y más resistente al cambio.

Principio abierto/cerrado (O)

Las entidades de software (clases, módulos, funciones, etcétera) deben estar abiertas para su extensión, pero cerradas para su modificación. Esto se logra mediante el uso de abstracciones (interfaces o clases abstractas) que permiten añadir nuevas funcionalidades sin alterar el código existente. Esto significa lo siguiente:

- abiertas para extensión. Se puede agregar comportamiento nuevo.
- Cerradas para modificación: el código ya probado no debe tocarse.

El objetivo es evitar introducir errores en código estable al incorporar nuevas funcionalidades.

Este principio existe por los siguientes motivos:

- el software cambia constantemente.
- El código existente suele estar en producción.

- Modificar código probado es riesgoso y costoso.

Explicación

Establece que las entidades software (clases, módulos y funciones) deberían estar abiertas para su extensión, pero cerradas para su modificación.

Si seguimos con la clase Coche, podemos observar lo siguiente:

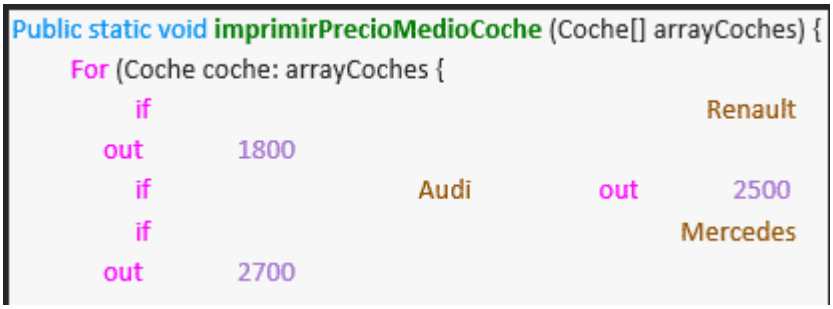
Si quisiéramos iterar a través de una lista de coches e imprimir sus marcas por pantalla, observaríamos lo siguiente:

```
Public static void main(String[] args) {
   Coche[] arrayCoches = {
              new Coche(“Renault”),
              new Coche(“Audi”),
   };
  imprimirPrecioMedioCoche(arrayCoches);
}
Public static void imprimirPrecioMedioCoche (Coche[] arrayCoches)
{
      For (Coche coche: arrayCoches {
            if                                                Renault
         out          1800
            if                        Audi            out          2500
```

Esto no cumple el principio abierto/cerrado, dado que, si decidimos añadir un nuevo coche de otra marca, se da lo siguiente:

```
Coche[] arrayCoches = {
            new Coche(“Renault”),
            new Coche(“Audi”),
            new Coche(“Mercedes”),
};
```

También tendríamos que modificar el método que hemos creado anteriormente.



Como podemos ver, para cada nuevo coche habría que añadir nueva lógica al método precioMedioCoche(). Esto es un ejemplo sencillo, pero imagina que tu aplicación crece y crece... ¿Cuántas modificaciones tendríamos que hacer? Mejor evitarnos esta pérdida de tiempo y dolor de cabeza, ¿verdad?

Para que cumpla con este principio, podríamos hacer lo siguiente:

```
Abstract class Coche {
   // ...
  abstract int precioMedioCoche();
}

class Renault extends Coche {
   @Override
   int precioMedioCoche() {  return 18000}
}

class Audi extends Coche {
   @Override
   int precioMedioCoche() {  return 25000}
}

class Mercedes extends Coche {
   @Override
   int precioMedioCoche() {  return 27000}
}

Public static void main(String[] args) {
     Coche[] arrayCoches = {
                new Renault(),
                new Audi(),
     };
    imprimirPrecioMedioCoche(arrayCoches);
}

Public static void imprimirPrecioMedioCoche (Coche[] arrayCoches)
{
       For (Coche coche: arrayCoches {
                        out          coche.precioMedioCoche()
```

Cada coche extiende la clase abstracta Coche e implementa el método abstracto precioMedioCoche().

Así, cada coche tiene su propia implementación del método precioMedioCoche(), por lo que el método

imprimirPrecioMedioCoche() itera el array de coches y solo llama al método precioMedioCoche().

Ahora, si añadimos un nuevo coche, precioMedioCoche() no tendrá que ser modificado. Solo tendremos que añadir el nuevo coche al array, lo que hace que se cumpla el principio abierto/cerrado.

Ejemplo del mundo real

Tabla 4: Procesamiento de pagos

Sin principio abierto/cerrado (mala práctica)	Con principio abierto/cerrado (buena práctica)

class ProcesadorPago {
 	
public void procesar(String tipo, double cantidad) {
 	
if (tipo.equals("TARJETA_CREDITO")) {
 	
// Lógica específica para tarjeta
 	
validarTarjeta();
	// Interfaz común
	
	interface MetodoPago {
	
	void procesarPago(double cantidad);
	
	boolean validar();
	
	}

```
        cargarTarjeta(cantidad);

        generarRecibo();

    } else if (tipo.equals("PAYPAL")) {

        // Lógica específica para PayPal

        autenticarPayPal();

        procesarPayPal(cantidad);

        notificarPayPal();

    } else if (tipo.equals("BITCOIN")) {

        // Lógica específica para Bitcoin

        verificarWallet();

        procesarTransaccionBTC(cantidad);

        confirmarBloque();

    }

        // ¡Cada nuevo método requiere MODIFICAR esta clase!

  }
```

```
// Implementaciones

class TarjetaCredito implements MetodoPago {

  private String numeroTarjeta;



   public boolean validar() {    // Validar tarjeta

       return numeroTarjeta.length() == 16;

  }



    public void procesarPago(double cantidad) {

     if (validar()) {

         System.out.println("Procesando tarjeta: $" + cantidad);

        // Lógica específica

     }
```

```
}
```

```
  }

}

class PayPal implements MetodoPago
{

  private String email;



  public boolean validar() {

     return email.contains("@");

  }



     public void procesarPago(double
cantidad) {

         System.out.println("Procesando
PayPal: $" + cantidad);

     // Lógica específica PayPal

  }

}
```

```
class Bitcoin implements MetodoPago {

  private String walletAddress;



  public boolean validar() {

                                  return walletAddress.startsWith("1") || walletAddress.startsWith("3");

  }



    public void procesarPago(double cantidad) {

        System.out.println("Procesando Bitcoin: $" + cantidad);

    // Lógica específica Bitcoin

  }

}
```

```
// Clase procesadora (CERRADA para modificación)

class ProcesadorPagoOCP {

    public void procesar(MetodoPago metodo, double cantidad) {

        if (metodo.validar()) {

            metodo.procesarPago(cantidad);

        }

    }

}
```

Usando el principio abierto/cerrado de buenas prácticas, para añadir nuevo método de pago, se puede hacer lo siguiente:

1. crear nueva clase que implemente MetodoPago.
2. ¡Esta clase no necesita cambiar!

Cuándo y cómo aplicar OCP

Usando el principio abierto/cerrado de buenas prácticas, para añadir nuevo método de pago, se puede hacer lo siguiente:

1. Frecuentes modificaciones en la misma clase para nuevas funcionalidades.
2. Muchos if/else o switch para manejar diferentes tipos.
3. Alta probabilidad de que el sistema necesite extensión futura.
4. Múltiples razones para cambiar en una misma clase.
5. SRP (principio de responsabilidad única) es prerrequisito del OCP (principio de abierto/cerrado).

Conclusión

El principio abierto/cerrado es fundamental para crear sistemas mantenibles y escalables. Las interfaces en Java son la herramienta principal para lograrlo por los siguientes motivos:

1. 1.desacoplan la especificación de la implementación.
2. Permiten añadir nuevas funcionalidades sin tocar código existente.
3. Facilitan la extensión mediante herencia de interfaz.
4. Habilitan patrones de diseño como Strategy, Decorator, Template Method.

OCP no significa «nunca modificar», sino «diseñar para que las extensiones sean más probables que las modificaciones». Es una inversión en flexibilidad futura que paga dividendos en mantenimiento a largo plazo. En Java, se implementa principalmente mediante los siguientes aspectos:

- interfaces.
- Polimorfismo.
- Inyección de dependencias.

Un sistema bien diseñado crece agregando clases, no modificando las existentes.

Liskov substitution principle (LSP): principio de sustitución de Liskov

Este principio indica que los objetos de una superclase deben poder ser sustituidos por objetos de sus subclases sin alterar la corrección del programa (Barbara Liskov, 1987). Simplemente exige que, siempre que se requiera una instancia de una clase padre, cualquier instancia de una subclase sea suficiente.

Formalmente: si S es un subtipo de T, entonces los objetos de tipo T pueden ser reemplazados por objetos de tipo S sin alterar las propiedades deseables del programa.

Explicación

Declara que una subclase debe ser sustituible por su superclase, y, si al hacer esto el programa falla, estaremos violando este principio.

Cumpliendo con este principio, se confirmará que nuestro programa tiene una jerarquía de clases fácil de entender y un código reutilizable.

Veamos un ejemplo.

```
Public static void imprimirNumAsientos(Coche[] arrayCoches) {
    For (Coche coche: arrayCoches {
            If
                out         numAsientosRenault(coche)
            If
                out         numAsientosAudi(coche)
            If
                out         numAsientosMercedes(coche)
```

Esto viola tanto el principio de substitución de Liskov como el de abierto/cerrado. El programa debe conocer cada tipo de Coche y llamar a su método numAsientos() asociado.

Así, si añadimos un nuevo coche, el método debe modificarse para aceptarlo.

```
Coche[] arrayCoches = {
            new Renaulut(),
            new Audi(),
            new Mercedes(),
            new Ford(),
};
public static void imprimirNumAsientos(Coche[] arrayCoches) {
    For (Coche coche: arrayCoches {
            If
                    out         numAsientosRenault(coche)
            If
                    out         numAsientosAudi(coche)
            If
                    out         numAsientosMercedes(coche)
            If
                    out         numAsientosFord(coche)
```

Para que este método cumpla con el principio, seguiremos los siguientes principios:

- si la superclase (Coche) tiene un método que acepta un parámetro del tipo de la superclase (Coche), entonces su subclase (Renault) debería aceptar como argumento un tipo de la superclase (Coche) o un tipo de la subclase (Renault).

- Si la superclase devuelve un tipo de ella misma (Coche), entonces su subclase (Renault) debería devolver un tipo de la

superclase (Coche) o un tipo de la subclase (Renault).

Si volvemos a implementar el método anterior, se da lo siguiente:

```
Public static void imprimirNumAsientos (Coche[] arrayCoches) {
    For (Coche coche: arrayCoches {
            out        coche.numAsientos()
```

Ahora, al método no le importa el tipo de la clase, simplemente llama al método numAsientos() de la superclase. Solo sabe que el parámetro es de tipo coche, ya sea Coche o alguna de las subclases.

Para esto, ahora la clase Coche debe definir el nuevo método:

```
Abstract class Coche {
  // ...
 abstract int numAsientos();
}
```

Y las subclases deben implementar dicho método:

```
class Renault extends Coche {
  @Override
  int numAsientos() {
      return 5
      }
}
```

Como podemos ver, ahora el método imprimirNumAsientos() no necesita saber con qué tipo de coche va a hacer su lógica, simplemente llama al método numAsientos() del tipo Coche, dado que, por contrato, una subclase de Coche debe implementar dicho método.

Conclusión final

El principio de sustitución de Liskov es crucial debido a los siguientes motivos:

1. garantiza la corrección del sistema cuando se usan subtipos.
2. Permite el polimorfismo seguro y predecible.
3. Facilita las pruebas al permitir mock objects que se comportan como los reales.
4. Habilita la extensibilidad sin romper el código existente.

LSP no es solo sobre herencia, sino sobre comportamiento substitutable. Una clase puede ser estructuralmente un subtipo (mismos métodos), pero no ser un subtipo comportamental.

La clave es pensar en términos de «contratos» y «expectativas de comportamiento» en lugar de solo «relaciones de tipo». Cuando un cliente usa una clase base, tiene ciertas expectativas sobre cómo se comportará; todos los subtipos deben cumplir esas expectativas.

Violar LSP introduce bugs sutiles y difíciles de detectar que aparecen solo en tiempo de ejecución y en contextos específicos. Respetar LSP lleva a un código más robusto, mantenible y predecible.

Interface segregation principle (ISP): principio de segregación de interfaces

Es mejor tener muchas interfaces específicas para un cliente que una interfaz de propósito general. Los clientes no deben verse obligados a depender de interfaces que no utilizan.

Explicación

Este principio establece que los clientes no deberían verse forzados a depender de interfaces que no usan.

Dicho de otra manera, cuando un cliente depende de una clase que implementa una interfaz cuya funcionalidad este cliente no usa, pero que otros clientes sí usan, este cliente estará siendo afectado por los cambios que fuercen otros clientes en dicha interfaz.

Imaginemos que queremos definir las clases necesarias para albergar algunos tipos de aves. Por ejemplo, tendríamos

loros, tucanes y halcones:

```
Interface  IAve {
	void volar();
	void comer();
}

Class Loro implements  IAve {
	@Override
	public void volar(){
		// ...
	}
	@Override
	public void comer(){
		// ...
	}
}

Class Tucan implements  IAve {
	@Override
	public void volar(){
		// ...
	}
	@Override
	public void comer(){
		// ...
	}
}
```

Hasta aquí todo bien. Sin embargo, ahora imaginemos que queremos añadir a los pingüinos. Estos son aves, pero además tienen la habilidad de nadar. Podríamos hacer esto:

```
interface  IAve {
        void volar();
        void comer();
}

Class Loro implements  IAve {
        @Override
        public void volar(){
                // ...
        }
        @Override
        public void comer(){
                // ...
        }
        @Override
        public void nadar(){
                // ...
        }
}


Class Pinguino implements  IAve {
        @Override
        public void volar(){
                // ...
        }
        @Override
        public void comer(){
                // ...
        }
        @Override
        public void nadar(){                    // ...
        }
}
```

El problema es que el loro no nada, y el pingüino no vuela, por lo que tendríamos que añadir una excepción o aviso si se intenta llamar a estos métodos. Además, si quisiéramos añadir otro método a la interfaz IAve, tendríamos que

recorrer cada una de las clases que la implementa e ir añadiendo la implementación de dicho método en todas ellas. Esto viola el principio de segregación de interfaz, dado que estas clases (los clientes) no tienen por qué depender de métodos que no usan.

Lo más correcto sería segregar más las interfaces, tanto como sea necesario. En este caso, podríamos hacer lo siguiente:

```
interface  IAve {
        void comer();
}

interface  IAveVoladora {
        void volar();
}

interface  IAveNadadora {
        void nadar();
}

Class Loro implements  IAve, IAveVoladora {
        @Override
        public void volar(){
                // ...
        }
        @Override
        public void comer(){
                // ...
        }
}

Class Pinguino implements  IAve, IAveNadadora {
        @Override
        public void comer(){
                // ...
        }
        @Override
        public void nadar(){                    // ...
        }
}
```

Así, cada clase implementa las interfaces cuyos métodos realmente necesita implementar. A la hora de añadir nuevas funcionalidades, esto nos ahorrará bastante tiempo, y, además, cumplimos con el primer principio (responsabilidad única).

Dependency inversion principle (DIP): principio de inversión de dependencias

Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones. Además, las abstracciones no deben depender de los detalles, sino que los detalles deben depender de las abstracciones. Esto promueve el desacoplamiento.

Seguir estos principios ayuda a evitar problemas comunes en el desarrollo de software, como la rigidez del código o la acumulación de errores, lo que da como resultado aplicaciones más robustas y fáciles de trabajar en equipo.

Explicación

En algún momento, nuestro programa o aplicación llegará a estar formado por muchos módulos. Cuando esto pase, deberemos usar inyección de dependencias, lo que nos permitirá controlar las funcionalidades desde un sitio concreto en vez de tenerlas esparcidas por todo el programa. Además, este aislamiento nos permitirá hacer testing mucho más fácilmente.

Supongamos que tenemos una clase para acceder a datos, y que lo hacemos a través de una BBDD:

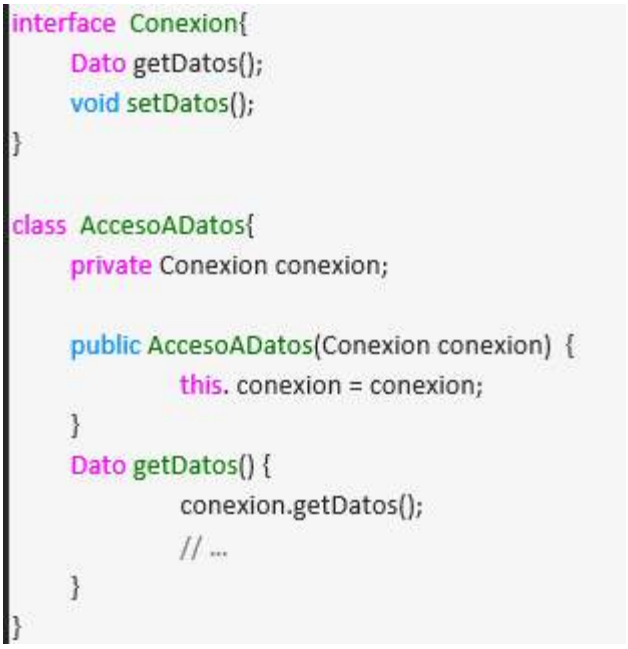
```
class  DatabaseService{
      @// ...
      void getDatos()  {  // ... }
}
class  AccesoADatos{
      private DataService dataService;
      public AccesoADatos(DataService dataService)  {
                  this.databaseService = dataService;
      }
      Dato getDatos() {
                  databaseService.getDatos();
                  // ...
      }
}
```

Imaginemos que en el futuro queremos cambiar el servicio de BBDD por un servicio que conecta con una API.

Tendríamos que ir modificando todas las instancias de la clase AccesoADatos, una por una.

Esto se debe a que nuestro módulo de alto nivel (AccesoADatos) depende de un módulo de más bajo nivel (DatabaseService), lo que viola el principio de inversión de dependencias. El módulo de alto nivel debería depender de abstracciones.

Para arreglar esto, podemos hacer que el módulo AccesoADatos dependa de una abstracción más genérica:



Así, sin importar el tipo de conexión que se le pase al módulo AccesoADatos, ni este ni sus instancias tendrán que cambiar,

por lo que nos ahorraremos mucho trabajo.

Ahora, cada servicio que queramos pasar a AccesoADatos deberá implementar la interfaz Conexion:

```
Class DatabaseService implements Conexion {
    @Overrride
    Public Dato getDatos() { // ... }

    @Overrride
    Public void  setDatos() { // ... }
}

Class APIService implements Conexion {
    @Overrride
    Public Dato getDatos() { // ... }

    @Overrride
    Public void  setDatos() { // ... }
}
```

Así, tanto el módulo de alto nivel como el de bajo nivel dependen de abstracciones, por lo que cumplimos el principio de inversión de dependencias. Además, esto nos forzará a cumplir el principio de Liskov, dado que los tipos derivados de Conexion (DatabaseService y APIService) son sustituibles por su abstracción (interfaz Conexion).

Conclusión

Aplicar estos cinco principios puede parecer algo tedioso, pero, a la larga, mediante la práctica y echándoles un vistazo de vez en cuando, estos se volverán parte de nuestra forma de programar.

Nuestro programa será más sencillo de mantener, pero no solo para nosotros, sino para los desarrolladores que vengan después, dado que verán un programa con una estructura bien definida y clara.

CONTINUAR

Referencias

[Imagen sin título sobre Robert Martin y Bárbara Liskov]. (s. f.). https://www.ecured.cu/Barbara_Liskov; https://www.wikiwand.com/es/articles/Robert_C._Martin.

CONTINUAR

Descarga en PDF