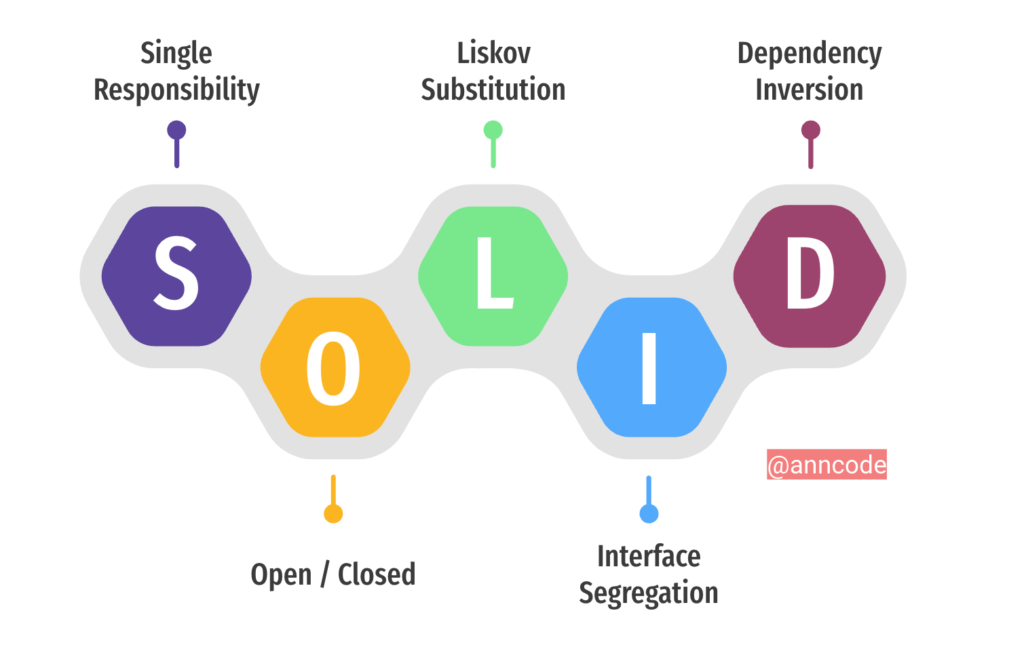
**PRINCIPIOS SOLID**



**Original article:** [The SOLID Principles of Object-Oriented Programming Explained in Plain English](https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/)

Los Principios SOLID tienen cinco principios de diseño de clases Orientado a Objetos. Son un conjunto de reglas y mejores prácticas a seguir al diseñar una estructura de clase.

Estos cinco principios nos ayudan a comprender la necesidad de ciertos patrones de diseño y arquitectura de software en general. Así que creo que es un tema que todo desarrollador debería aprender.

Este artículo te enseñará todo lo que necesitas saber para aplicar los principios SOLID a tus proyectos.

Empezaremos echando un vistazo a la historia de este término. Luego vamos a entrar en los detalles esenciales, los por qué y cómo de cada principio, creando un diseño de clase y mejorándolo paso a paso.

¡Así que toma una taza de café, mate o té y empecemos!

**Antecedentes**

Los principios SOLID fueron introducidos por primera vez por el famoso científico informático Robert J. Martin (también conocido como el tío Bob) en uno de sus artículos en 2000. Pero el acrónimo SOLID fue introducido más tarde por Michael Feathers.

El tío Bob también es el autor de los libros best-sellers *Clean Code* y *Clean Architecture*, y es uno de los participantes de la "[Alianza Agile](https://agilemanifesto.org/history.html)".

Por lo tanto, no sorprende que todos estos conceptos de codificación limpia, arquitectura orientada a objetos y patrones de diseño estén de alguna manera conectados y se complementen entre sí.

Todos tienen el mismo propósito:

"Para crear código comprensible, legible y comprobable en el que muchos desarrolladores puedan trabajar en colaboración."

Veamos cada principio uno por uno. Siguiendo el acrónimo inglés SOLID, son:

* El Principio de responsabilidad única (**S**ingle Responsibility Principle)
* El Principio Abierto-Cerrado (**O**pen-Closed Principle)
* El Principio de sustitución de Liskov (**L**iskov Substitution Principle)
* El Principio de segregación de interfaz (**I**nterface Segregation Principle)
* El Principio de inversión de dependencia (**D**ependency Inversion Principle)

**Principio de responsabilidad única**

El Principio de Responsabilidad Única dice que **una clase debe hacer una cosa y, por lo tanto, debe tener una sola razón para cambiar.**

Para enunciar este principio más técnicamente: Solo un cambio potencial (lógica de base de datos, lógica de registro, etc.) en la especificación del software debería poder afectar la especificación de la clase.

Esto significa que si una clase es un contenedor de datos, como una clase Libro o una clase Estudiante, y tiene algunos campos relacionados con esa entidad, debería cambiar solo cuando cambiamos el modelo de datos.

Es importante seguir el principio de responsabilidad única. En primer lugar, debido a que muchos equipos diferentes pueden trabajar en el mismo proyecto y editar la misma clase por diferentes motivos, esto podría dar lugar a módulos incompatibles.

En segundo lugar, facilita el control de versiones. Por ejemplo, digamos que tenemos una clase de persistencia que maneja las operaciones de la base de datos y vemos un cambio en ese archivo en las confirmaciones de GitHub. Al seguir el PRU, sabremos que está relacionado con el almacenamiento o con cosas relacionadas con la base de datos.

Los conflictos de fusión son otro ejemplo. Aparecen cuando diferentes equipos modifican el mismo archivo. Pero si se sigue el PRU, aparecerán menos conflictos: los archivos tendrán una sola razón para cambiar y los conflictos que existen serán más fáciles de resolver.

**Trampas comunes y Anti-patrones**

En esta sección, veremos algunos errores comunes que violan el Principio de Responsabilidad Única. Luego hablaremos sobre algunas formas de solucionarlos.

Veremos el código de un programa simple de facturación de librería como ejemplo. Comencemos definiendo una clase de libro para usar en nuestra factura.

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;

}

}

Esta es una clase de libro simple con algunos campos. Nada sofisticado. No estoy haciendo que los campos sean privados para que no tengamos que lidiar con getters y setters y podamos centrarnos en la lógica.

Ahora vamos a crear la clase de factura que contendrá la lógica para crear la factura y calcular el precio total. Por ahora, suponga que nuestra librería solo vende libros y nada más.

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.

}

}

Aquí está nuestra clase de Factura. También contiene algunos campos sobre facturación y 3 métodos:

* **calculaTotal**método que calcula el precio total,
* **imprimeFactura** método que debería imprimir la factura por consola, y
* **guardaArchivo**método responsable de escribir la factura en un archivo.

Debe darse un segundo para pensar en lo que está mal con este diseño de clase antes de leer el siguiente párrafo.

Bien, entonces, ¿qué está pasando aquí? Nuestra clase viola el Principio de Responsabilidad Única de múltiples maneras.

La primera violación es el método **imprimeFactura**, el cual contiene nuestra lógica de impressión. El PRU establece que nuestra clase solo debería tener una única razón para cambiar, y esa razón debería ser un cambio en el cálculo de la factura para nuestra clase.

Pero en esta arquitectura, si queremos cambiar el formato de impresión, necesitaríamos cambiar la clase. Esta es la razón por la que no deberíamos tener lógica de impresión mezclada con lógica de negocios en la misma clase.

Hay otro método que viola el PRU en nuestra clase: el método **guardarArchivo**. También es un error extremadamente común mezclar la lógica de persistencia con la lógica de negocios.

No piense solo en términos de escribir en un archivo, podría ser guardarlo en una base de datos, hacer una llamada a la API u otras cosas relacionadas con la persistencia.

Entonces, ¿cómo podemos arreglar esta función de impresión?, puede preguntar.

Podemos crear nuevas clases para nuestra lógica de impresión y persistencia, por lo que ya no necesitaremos modificar la clase de factura para esos fines.

Creamos 2 clases, **FacturaImpresion** y **FacturaPersistencia,** y movemos los métodos.

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.

}

}

Ahora nuestra estructura de clases obedece al principio de responsabilidad única y cada clase es responsable de un aspecto de nuestra aplicación. ¡Excelente!

**Principio de apertura y cierre**

El principio de apertura y cierre exige que **las clases deban estar abiertas a la extensión y cerradas a la modificación.**

Modificación significa cambiar el código de una clase existente y extensión significa agregar una nueva funcionalidad.

Entonces, lo que este principio quiere decir es: Deberíamos poder agregar nuevas funciones sin tocar el código existente para la clase. Esto se debe a que cada vez que modificamos el código existente, corremos el riesgo de crear errores potenciales. Por lo tanto, debemos evitar tocar el código de producción probado y confiable (en su mayoría) si es posible.

Pero, ¿cómo vamos a agregar una nueva funcionalidad sin tocar la clase?, puede preguntarse. Por lo general, se hace con la ayuda de interfaces y clases abstractas.

Ahora que hemos cubierto los conceptos básicos del principio, apliquémoslo a nuestra aplicación Factura.

Digamos que nuestro jefe vino a nosotros y dijo que quiere que las facturas se guarden en una base de datos para que podamos buscarlas fácilmente. Creemos que está bien, esto es fácil jefe, ¡solo dame un segundo!

Creamos la base de datos, nos conectamos a ella y agregamos un método de guardado a nuestra clase **FacturaPersistencia**:

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

}

}

Lamentablemente, nosotros, como desarrolladores perezosos de la librería, no diseñamos las clases para que fueran fácilmente ampliables en el futuro. Entonces, para agregar esta función, hemos modificado la clase FacturaPersistencia.

Si nuestro diseño de clase obedeciera al principio Abierto-Cerrado, no necesitaríamos cambiar esta clase.

Entonces, como el desarrollador perezoso pero inteligente de la librería, vemos el problema de diseño y decidimos refactorizar el código para obedecer el principio.

interface FacturaPersistencia {

public void guardar(Factura factura);

}

Cambiamos el tipo de **FacturaPersistencia** a Interface y agregamos un método de guardado. Cada clase de persistencia implementará este método de guardado.

public class BaseDeDatosPersistencia implements FacturaPersistencia {

@Override

public void guardar(Factura factura) {

// Guardar en la base de datos

}

}

public class ArchivoPersistencia implements FacturaPersistencia {

@Override

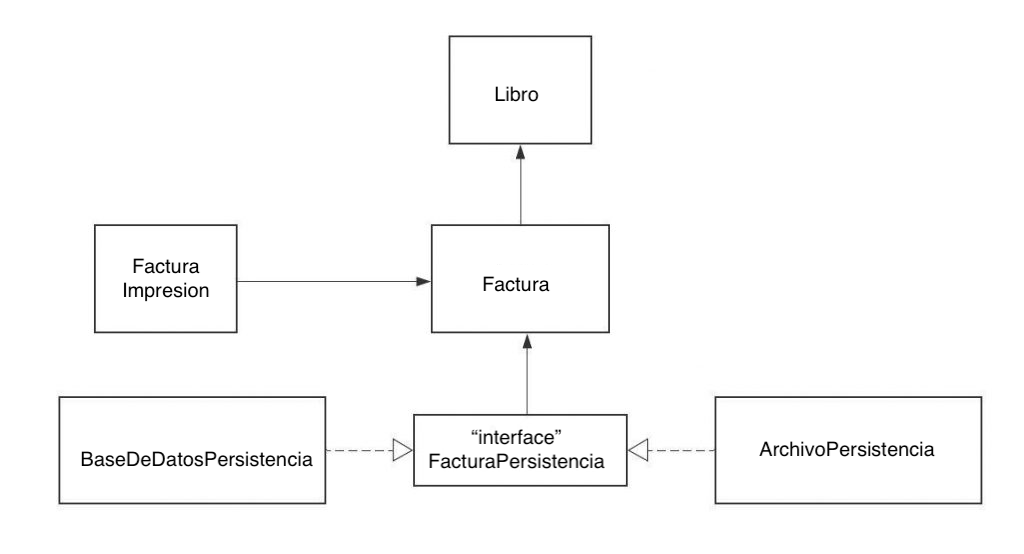
public void guardar(Factura factura) {

// Guardar en archivo

}

}

Así que nuestra estructura de clases ahora se ve así:



Ahora nuestra lógica de persistencia es fácilmente extensible. Si nuestro jefe nos pide que agreguemos otra base de datos y tengamos 2 tipos diferentes de bases de datos como MySQL y MongoDB, podemos hacerlo fácilmente.

Puede pensar que podríamos simplemente crear múltiples clases sin una interfaz y agregar un método de guardado para todas ellas.

Pero supongamos que ampliamos nuestra aplicación y tenemos varias clases de persistencia como **FacturaPersistencia**, **LibroPersistencia** y creamos una clase **AdministradorPersistencia** que administra todas las clases de persistencia:

public class AdministradorPersistencia {

FacturaPersistencia facturaPersistencia;

LibroPersistencia libroPersistencia;

public AdministradorPersistencia(FacturaPersistencia facturaPersistencia, LibroPersistencia libroPersistencia) {

this.facturaPersistencia = facturaPersistencia;

this.libroPersistencia = libroPersistencia;

}

}

Ahora podemos pasar cualquier clase que implemente la interfaz **FacturaPersistencia** a esta clase con la ayuda del polimorfismo. Esta es la flexibilidad que proporcionan las interfaces.

**Principio de sustitución de Liskov**

El Principio de Sustitución de Liskov establece que las subclases deben ser sustituibles por sus clases base.

Esto significa que, dado que la clase B es una subclase de la clase A, deberíamos poder pasar un objeto de la clase B a cualquier método que espere un objeto de la clase A y el método no debería dar ningún resultado extraño en ese caso.

Este es el comportamiento esperado, porque cuando usamos la herencia asumimos que la clase secundaria hereda todo lo que tiene la superclase. La clase secundaria extiende el comportamiento pero nunca lo reduce.

Por lo tanto, cuando una clase no obedece este principio, genera algunos errores desagradables que son difíciles de detectar.

El principio de Liskov es fácil de entender, pero difícil de detectar en el código. Así que veamos un ejemplo.

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;

}

}

Tenemos una clase Rectángulo simple y una función **getArea** que devuelve el área del rectángulo.

Ahora decidimos crear otra clase para Cuadrados. Como sabrás, un cuadrado es solo un tipo especial de rectángulo donde el ancho es igual a la altura.

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);

}

}

Nuestra clase Cuadrado amplía la clase Rectángulo. Establecemos alto y ancho en el mismo valor en el constructor, pero no queremos que ningún cliente (alguien que use nuestra clase en su código) cambie el alto o el peso de una manera que pueda violar la propiedad cuadrada.

Por lo tanto, anulamos los setters para establecer ambas propiedades cada vez que se cambia una de ellas. Pero al hacerlo acabamos de violar el principio de sustitución de Liskov.

Vamos a crear una clase principal para realizar pruebas en la función **getArea**.

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);

}

}

Al probador de su equipo se le ocurrió la función de prueba **getAreaTest** y le dice que su función **getArea** no pasa la prueba de objetos cuadrados.

En la primera prueba, 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**

La segregación significa mantener las cosas separadas, y el Principio de Segregación de Interfaces se trata de separar las interfaces.

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.

Este es un principio simple de entender y aplicar, así que veamos un ejemplo.

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 {

}

Modelamos un estacionamiento muy simplificado. Es el tipo de estacionamiento donde pagas una tarifa por hora. Ahora considere que queremos implementar un estacionamiento que sea gratuito.

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) {

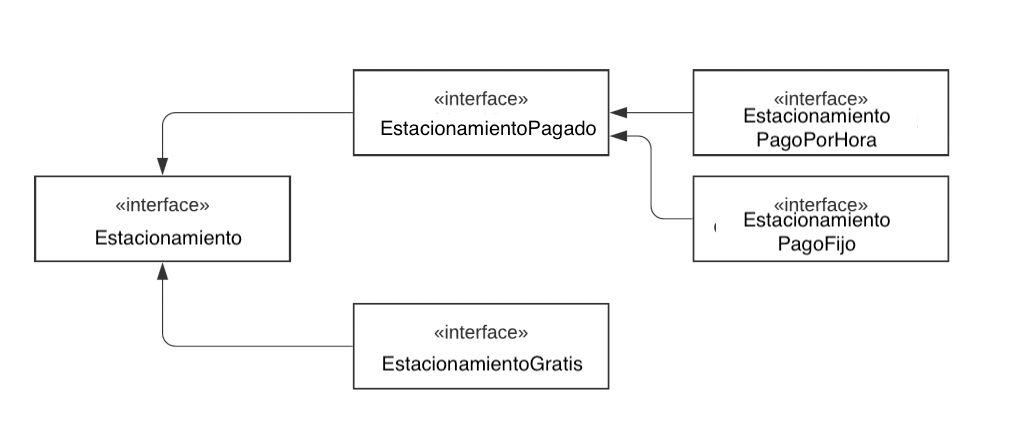
throw new Exception("Estacionamiento es gratis");

}

}

Nuestra interfaz de estacionamiento se componía de 2 cosas: lógica relacionada con el estacionamiento (aparcar coche, sacar automóvil, obtener capacidad) y lógica relacionada con el pago.

Pero es demasiado específico. Debido a eso, nuestra clase EstacionamientoGratis se vio obligada a implementar métodos relacionados con el pago que son irrelevantes. Separemos o segreguemos las interfaces.



Ahora hemos separado el estacionamiento. Con este nuevo modelo, incluso podemos ir más allá y dividir **EstacionamientoPagado** para admitir diferentes tipos de pago.

Ahora nuestro modelo es mucho más flexible, extensible y los clientes no necesitan implementar ninguna lógica irrelevante porque solo proporcionamos funcionalidad relacionada con el estacionamiento en la interfaz del estacionamiento.

**Principio de inversión de dependencia**

El principio de inversión de dependencia establece que nuestras clases deben depender de interfaces o clases abstractas en lugar de clases y funciones concretas.

En su [artículo](https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf) (2000), el tío Bob resume este principio de la siguiente manera:

"Si el PAC establece el objetivo de la arquitectura OO, el PID establece el mecanismo principal".

Estos dos principios están realmente relacionados y hemos aplicado este patrón antes mientras discutíamos el Principio Abierto-Cerrado.

Queremos que nuestras clases estén abiertas a la extensión, por lo que hemos reorganizado nuestras dependencias para que dependan de interfaces en lugar de clases concretas. Nuestra clase **AdministradorPersistencia** depende de **FacturaPersistencia** en lugar de las clases que implementan esa interfaz.