

## Programación con principios

<b>Programación con principios</b>	<b>1</b>
¿Qué aprenderás?	2
Introducción	2
Principio de modularización	3
Principio DRY (Don't Repeat Yourself)	4
Principio KISS (Keep It Simple Stupid)	5
Principio YAGNI (You Aren't Gonna Need It)	5
Principio de Acoplamiento y Cohesión (Tight & Loose)	6
Principios SOLID	7
Single Responsibility Principle (Principio de responsabilidad única)	7
Open Closed Principle (Principio Abierto - Cerrado)	8
Liskov Substitution Principle (Principio de sustitución de Liskov )	9
Interface Segregation Principle (Principio de segregación de interfaces)	10
Dependency Inversion Principle (Principio de inversión de dependencias)	11



**¡Comencemos!**

## ¿Qué aprenderás?

- Comprender los principios que se aplican en la POO.
- Aplicar los diversos principios para entender la POO.

## Introducción

A continuación, veremos algunos de los lineamientos más importantes en la Orientación a Objetos: los famosos **principios de la programación** o “técnicas para el desarrollo más acertado”. Veremos ejemplos básicos para comprender lo que significa cada uno de ellos, con el fin de lograr que nuestros desarrollos en el futuro sean bien calificados.

**¡Vamos con todo!**



## Principio de modularización

El principio de modularización dicta que se deben separar las funcionalidades de un software en módulos y cada uno de ellos debe estar encargado de una parte del sistema. Esto ayuda a que cada módulo sea independiente y, por consecuencia, si se desea modificar uno de los módulos, el impacto en los otros no sea importante.

Para lograr la modularidad se debe atomizar un problema y obtener sub-problemas donde cada módulo tienda a dar solución. Estos módulos pueden estar separados en métodos, clases, paquetes, colecciones de paquetes e incluso proyectos. La escala de modularidad va a depender del tamaño del software en cuestión.



Imagen 1. Ejemplo de modularización con piezas de puzzle.  
Fuente: Desafío Latam

## Principio DRY (Don't Repeat Yourself)

El principio DRY, como su nombre lo indica, se refiere a no repetir el código en ninguna instancia. Por ejemplo, si vamos a usar un `if` idéntico más de una vez, estamos violando el principio y, como solución, deberíamos guardar ese `if` dentro de un método y reutilizarlo donde sea necesario.

Hay otros casos en que dos porciones de código hacen casi lo mismo, por ejemplo:

```
int indiceNombre;  
int indiceApellido;  
for(int i = 0; i <= listaNombres; i++){  
    if(listaNombres.get(i).equals("Juan")){  
        indiceNombre = i;  
    }  
    for(int i = 0; i <= listaApellidos; i++){  
        if(listaApellidos.get(i).equals("Perez")){  
            indiceApellido = i;  
        }  
    }  
}
```

En este caso, tenemos dos ciclos que hacen casi lo mismo y podríamos reemplazarlos creando el siguiente método:

```
public int retornarIndice(String elementoBuscado, List<String> lista){  
    for(int i = 0; i <= lista; i++){  
        if(lista.get(i).equals(elementoBuscado)){  
            return i;  
        }  
    }  
}
```

Entonces el primer código quedaría así:

```
int indiceNombre = retornarIndice("Juan", listaNombres);  
int indiceApellido = retornarIndice("Perez", listaApellidos);
```

De esta forma, el código queda más ordenado y se cumple el principio DRY.

## Principio KISS (Keep It Simple Stupid)

Este principio se refiere a crear un software sin necesidad de hacerlo más complejo. De esta forma es más fácil de entender y utilizar. La simplicidad es bien aceptada en el diseño de todo ámbito y qué mejor ejemplo que el de Apple que creó un teléfono con un solo botón para manejarlo.



Imagen 2. Principio KISS.  
Fuente: Desafío Latam

## Principio YAGNI (You Aren't Gonna Need It)

Este principio indica que no se debe agregar piezas que no se utilizarán. Por ejemplo, al pensar en lo que se hará en el futuro, se agregan piezas de software que aún no van a ser utilizadas, pero que podrían ser necesarias más adelante cuando la aplicación crezca. En este caso, se está violando el principio porque estaríamos agregando código que no necesitamos en el momento, es mejor enfocarse en lo realmente necesario y no perder tiempo en funcionalidades extras que no se han pedido... **tu tiempo como desarrollador/a vale oro.**

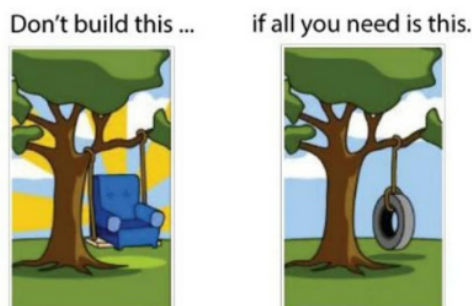


Imagen 3. Principio YAGNI.  
Fuente: Desafío Latam

Estos últimos tres principios se refieren a la programación en general, hay algunos principios que se refieren a la programación modularizada y a la cohesión del software y diseño.

## Principio de Acoplamiento y Cohesión (Tight & Loose)

La cohesión es el concepto que mide la "fuerza" con que las partes o piezas de un software están conectadas dentro de un módulo. La cohesión puede medirse como alta (fuerte) o baja (débil). Se prefiere una cohesión alta debido a que esto significa que el software es más robusto, escalable y fiable. Además el código facilita el entendimiento de los desarrolladores debido a su grado de reutilización del mismo.

Este concepto y sus métricas fueron primero diseñadas por Larry Constantine en el diseño estructural (o diseño para programación estructurada). El concepto de acoplamiento, se podría decir que es lo contrario de la cohesión del software, ya que un código acoplado es difícil de entender y mejorar debido a que muchas cosas dependen de otras muchas cosas dentro del código y si algo se modifica podrían dejar de funcionar correctamente, ya sea durante su compilación o durante la ejecución del software.

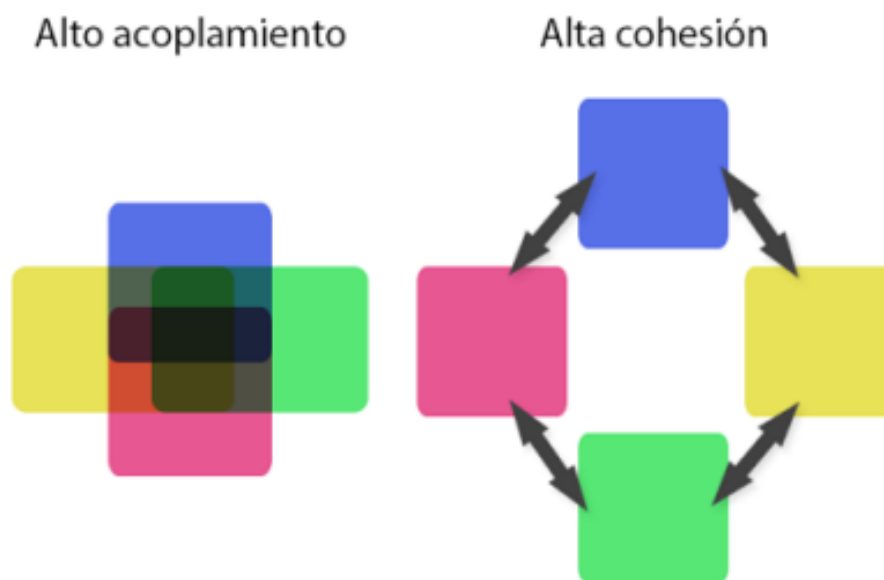


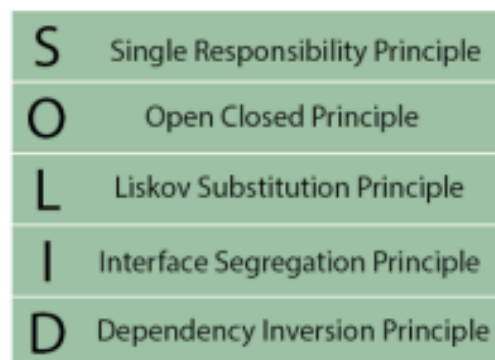
Imagen 4. Acoplamiento vs Cohesión.

Fuente: Desafío Latam

A continuación, veremos la segunda parte de los principios de desarrollo. Es el turno del conjunto de principios SOLID, una mezcla de principios un tanto complejos de entender, pero que si se llevan a la práctica el código creado será de una calidad excelente.

## Principios SOLID

Los principios SOLID le deben su nombre a dos cosas, la primera y la más evidente es su traducción de la palabra en inglés: "ROBUSTO". La segunda razón es debido a los cinco principios de Programación Orientada a Objetos que lo conforman, que no son más que buenas prácticas para realizar un software de buena calidad. Los cinco principios que conforman SOLID son:

Una imagen que muestra los cinco principios SOLID en una lista vertical. Cada principio está en una caja verde con un borde blanco. Las letras S, O, L, I, D están grandes y en negrita, seguidas por el nombre del principio en un tamaño de fuente menor.

<b>S</b>	Single Responsibility Principle
<b>O</b>	Open Closed Principle
<b>L</b>	Liskov Substitution Principle
<b>I</b>	Interface Segregation Principle
<b>D</b>	Dependency Inversion Principle

Imagen 5. Principios SOLID.  
Fuente: Desafío Latam

A continuación, conoceremos la importancia y el significado de cada uno:

### Single Responsibility Principle (Principio de responsabilidad única)

Este principio, aunque es fácil de explicar, es difícil de implementar y no es más que lo que su mismo nombre indica: cada objeto debe tener una única responsabilidad dentro del software.

Para este principio, la responsabilidad es la razón por la cual cambia el estado de una clase, es decir, darle a una clase una responsabilidad es darle una razón para cambiar su estado. Robert C. Martin es el creador de este principio y lo dio a conocer en su obra "Agile Principles, Patterns, and Practices in C#".

Si, por ejemplo, tenemos una clase PDF representando un archivo .pdf:

```
public class PDF{  
    int paginas;  
    String titulo;  
}
```

Y quisiéramos hacer que el archivo se imprima, deberíamos crear otra clase que lo imprima y no hacer un método `imprimir()` dentro de la misma clase.

Ahora, teniendo una clase que imprima PDF, podríamos tener una clase Docx y no tendríamos que copiar y pegar el método imprimir, sino que podríamos usar la clase que imprime PDF para imprimir Docx.

Este principio ayudará a que las clases puedan ser reutilizadas fácilmente, gracias a que se transforman en algo más genérico al tener una sola responsabilidad dentro del software. Al regirnos por este principio ayuda a la cohesión del software, ya que cada clase está encargada de una función en específico y, por ende, puede ser modificada fácilmente.

## Open Closed Principle (Principio Abierto - Cerrado)

Este principio dice que un objeto dentro del software, sea este una clase, módulo, función, etc., debe estar disponible para ser extendido (Abierto), pero no estarlo para modificaciones directas de su código actual (Cerrado).

Esto quiere decir que si, por ejemplo, tenemos un objeto con los atributos:

```
int valor;  
String nombre;
```

Deberíamos agregar nuevos atributos sin modificar valor ni nombre, que son los que ya existen. La razón de esto es simple, puesto que el atributo podría estar referenciado en otras partes del software y, al cambiar su nombre, la referencia se perdería.

Por ejemplo, si en el main llamamos al método getter de valor:

```
public int getValor(){  
    return valor;  
}
```



Y cambiamos la variable de la siguiente forma:

```
int valorActual;
```

El getter dejaría de funcionar, debido a que no podrá encontrar `int valor` dentro de la clase, ya que esta ahora es `int valorActual`. Por ende, tendríamos que cambiar el getter:

```
public int getValorActual() {  
    return valorActual;  
}
```

Y ahora deberíamos cambiar todas las referencias que hay hacia `getValor()` para que sean `getValorActual()`. En el fondo, es un problema cambiar la variable de nombre y el ejemplo anterior es el menor de los problemas que podría ocasionar el cambio del nombre de una variable. Es por eso que se debe hacer un análisis y al menos un diagrama de clases de lo que será el proyecto que se vaya a realizar en cualquier caso.

Aplicando el principio “Abierto-Cerrado” conseguirás una mayor cohesión, mejorarás la lectura y reducirás el riesgo de romper alguna funcionalidad ya existente.

## Liskov Substitution Principle (Principio de sustitución de Liskov )

Este principio nos indica que cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas. Este principio fue creado por Barbara Liskov y dicta la forma correcta de utilizar la herencia.



Imagen 6. Liskov Substitution Principle.  
Fuente: Desafío Latam

Por ejemplo, si tenemos una clase `Gato` que extiende de `Animal`, y la clase `Animal` tiene el método `volar()`, la herencia deja de ser válida. Es bastante evidente que los gatos no pueden volar, por ende, siempre que se haga una herencia se debe pensar que todas las subclases de una clase realmente utilicen los métodos y atributos que están heredando.

## Interface Segregation Principle (Principio de segregación de interfaces)

Este principio dicta que las clases que implementen una interface deberían utilizar todos y cada uno de los métodos que tiene la interface. Si no es así, la mejor opción es dividir la interface en varias, hasta lograr que las clases solo implementen métodos que utilizan. Los clientes no deberían verse forzados a depender de interfaces que no usan.

Imagina que haces una interface llamada `VehiculoMotorizado` y tres clases que le hereden: `Auto`, `Lancha` y `Avión`. Le agregamos el método `despegar()` y `encenderMotor()`. Sin embargo, al hacer esto estaríamos violando el principio de segregación de clases, ya que un auto no debería ocupar el método `despegar`. La solución a esto sería crear más interfaces hasta lograr que todos los que heredan de estas necesiten hacerlo, tal como se muestra en la imagen 8.

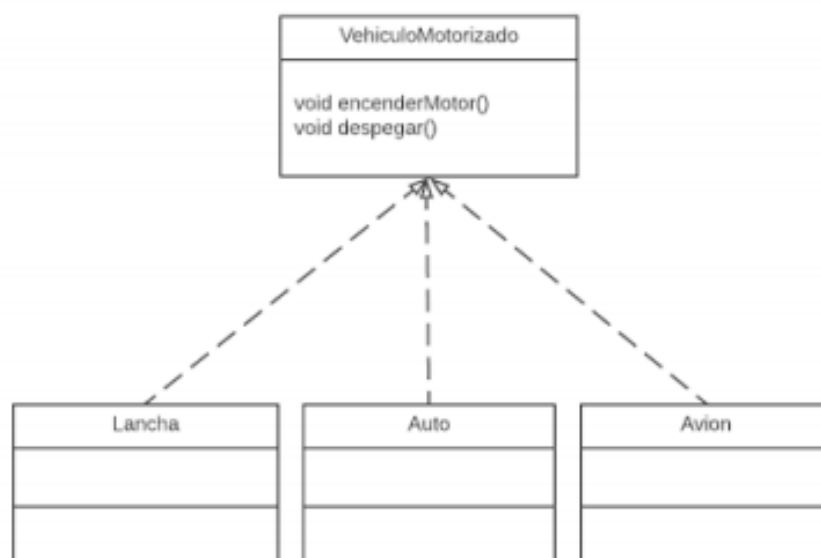


Imagen 7. Interface Segregation Principle.

Fuente: Desafío Latam

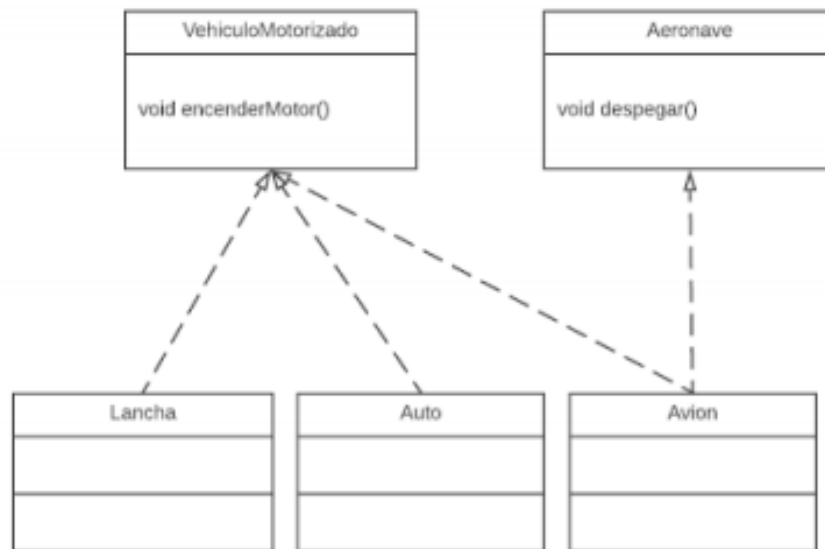


Imagen 8. Aplicando el principio de segregación de clases.  
Fuente: Desafío Latam

## Dependency Inversion Principle (Principio de inversión de dependencias)

Este principio indica que los módulos de alto nivel no deben depender de módulos de bajo nivel. Las abstracciones no deberían depender de los detalles, sino que los detalles deberían depender de las abstracciones.

Cuando hay una fuente externa de datos, como por ejemplo, un usuario ingresando datos, se debe hacer que el módulo del software donde se reciben los datos ingresados por el usuario (módulo de bajo nivel) no sea parte del módulo donde los datos se procesan (módulo de alto nivel). Esto permite que el módulo donde se reciben los datos pueda ser reemplazado fácilmente por uno diferente, manteniendo el módulo que los procesa intacto y haciéndolo reutilizable.

Esta reutilización se hace incluso sin necesidad de reemplazar el módulo de ingreso de datos, recibiendo estos datos paralelamente de diferentes fuentes sin estar ligado a ninguna de ellas. Por ejemplo, si tenemos un botón que enciende y apaga una lámpara, y lo hiciéramos sin aplicar el principio de inversión de dependencias, se vería algo así:

```
class Boton{
    Lampara lamp;
    void presionarBoton(Boolean presionado){
        this.lamp.encenderApagar(presionado);
    }
}

class Lampara{
    boolean encendido;
    void encenderApagar(Boolean presionado){
        this.encendido = presionado;
    }
}
```

Analicemos lo anterior. Si tenemos una "interface de usuario" llamada **Boton** que es un código de bajo nivel (ya que recibe datos del usuario) y que se comunica con una lámpara en concreto, encendiéndose o apagándose gracias a su método **encenderApagar()**; la lógica para apagar cuando está encendida y encender cuando está apagada, se considera de alto nivel, ya que es la que procesa la información.



Imagen 9. Flujo de Dependencias.  
Fuente: Desafío Latam

En este caso, estamos haciendo que **Lampara**, que es el código de alto nivel, dependa del **Boton**, por ende estamos violando el principio Inversión de Dependencias. Para solucionar esto, debemos crear una interfaz de **Boton**... te preguntarás, ¿para qué necesitas una interfaz de **Boton** si puedes dejarlo como una clase y usarlo cuando sea necesario instanciarlo?

La respuesta es que haciendo una interface podrás usar el mismo botón para encender y apagar cualquier otro aparato, y esto se rige por la segunda parte que dicta el principio de inversión de dependencias: las abstracciones no deberían depender de los detalles, sino que los detalles deberían depender de las abstracciones.

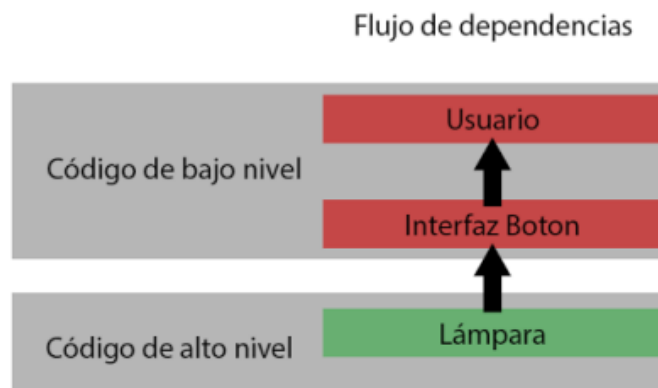


Imagen 10. Flujo de Dependencias para Lampara.  
Fuente: Desafío Latam

Haremos que la lámpara dependa de la abstracción de botón, abstracción que servirá para cualquier otro aparato que pueda necesitar un botón que lo encienda o apague.

```
interface Boton{
    void presionarBoton(Boolean presionado);
}
class Lampara implements Boton{
    public boolean encendido;
    @Override
    void presionarBoton(Boolean presionado){
        this.encendido = presionado;
    }
}
```

Aunque es un ejemplo simple debido a su reducido nivel de complejidad, este principio es muy importante, al igual que los otros cuatro, debido a que aumenta muchísimo la cohesión del código, haciendo que solo dependa de abstracciones y no existan clases acopladas como el `Boton` y la `Lampara` de la primera versión del código.