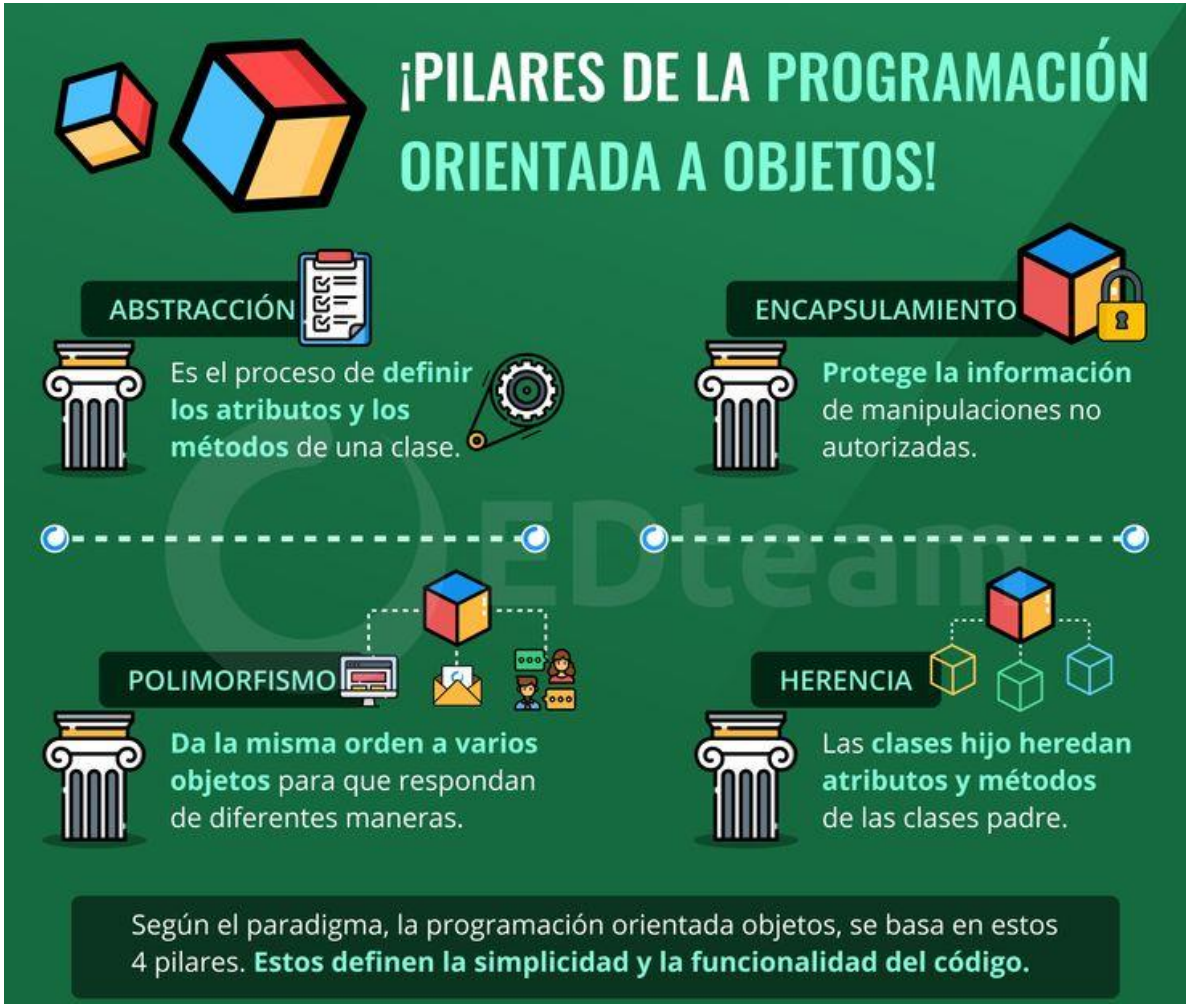
FIP

Carrera Programador full-stack

SOLID


¡PILARES DE LA PROGRAMACIÓN ORIENTADA A OBJETOS!
ABSTRACCIÓN
Es el proceso de definir los atributos y los métodos de una clase.
ENCAPSULAMIENTO
Protege la información de manipulaciones no autorizadas.
POLIMORFISMO
Da la misma orden a varios objetos para que respondan de diferentes maneras.
HERENCIA
Las clases hijo heredan atributos y métodos de las clases padre.
Según el paradigma, la programación orientada objetos, se basa en estos 4 pilares. Estos definen la simplicidad y la funcionalidad del código.

PRINCIPIOS SOLID DE DISEÑO

SOLID es un acrónimo de los primeros cinco principios del diseño orientado a objetos (OOD) de Robert C. Martin (también conocido como el Tío Bob).

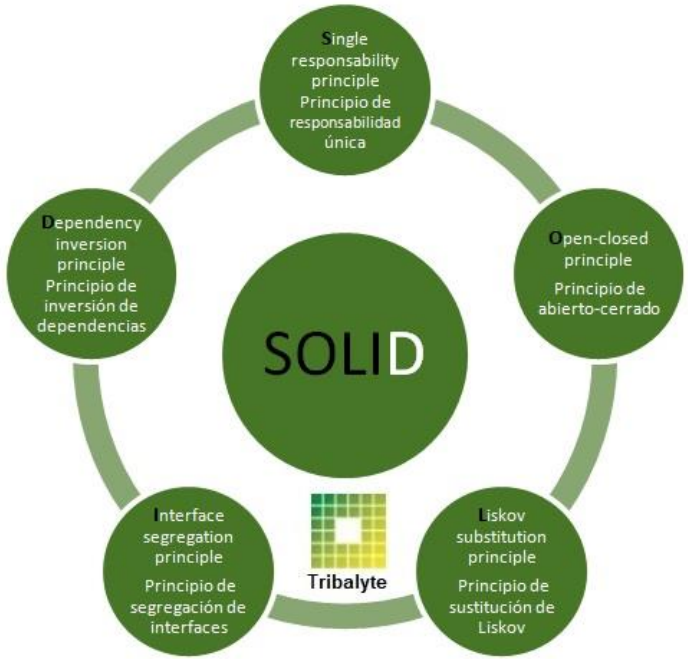
Estos principios establecen prácticas que se prestan al desarrollo de software con consideraciones para su mantenimiento y expansión a medida que el proyecto se amplía. Adoptar estas prácticas también puede ayudar a evitar los aromas de código, refactorizar el código y aprender sobre el desarrollo ágil y adaptativo de software.

S: (Single) Principio de responsabilidad única
O: (Open) Principio abierto-cerrado
L: (Liskov) Principio de sustitución de Liskov
I: (Interface) Principio de segregación de interfaz
D: (Dependency) Principio de inversión de dependencia

Objetivos

Entre los objetivos de tener en cuenta estos 5 principios a la hora de escribir código encontramos:

- Crear un **software eficaz**: que cumpla con su cometido y que sea **robusto y estable**.
- Escribir un **código limpio y flexible** ante los cambios: que se pueda modificar fácilmente según necesidad, que sea **reutilizable** y **mantenible**.
- Permitir **escalabilidad**: que acepte ser ampliado con nuevas funcionalidades de manera ágil.



1.Single responsability

- Single Responsability Principle (SRP)
- Principio de responsabilidad única
- Una clase debe tener una única responsabilidad
- Se entiende por responsabilidad un motivo de cambio

EJEMPLO

Nuestra casa posee tres tipos de aves diferentes: gorriones, loros y pinguinos. Se debe permitir setear la altitud de vuelo para el caso de aves que vuelan)
El canto de los loros es grabado habitualmente por sus dueños en distintos formatos: MP3 y WMA.
Se espera llevar registro del tiempo total que las aves que vuelan han permanecido volando

Si el AVE y el VUELO son incluidos en la misma clase, en lugar de dos clases distintas, se estaría infringiendo este principio

EJEMPLO II

Pensemos en un módulo que compila e imprime un reporte.
Este módulo puede cambiarse por dos razones: primero, puede cambiar el contenido del reporte, y segundo, puede cambiar el formato del reporte
Estos son cambios muy distintos: uno de contenido y otro de formato
Estos aspectos son dos problemas distintos y deben existir en clases separadas

2.Open/closed

- Open Closed Principle (OCP)
- Las entidades de software deben estar abiertas para ser extendidas, cerradas para ser modificadas

El segundo principio de SOLID lo formuló Bertrand Meyer en 1988 en su libro “Object Oriented Software Construction” y dice: “Deberías ser capaz de extender el comportamiento de una clase, sin modificarla”. En otras palabras: las clases que usas deberían estar **abiertas para poder extenderse y cerradas para modificarse**.

Ejemplo

Nuestra casa posee tres tipos de aves diferentes: gorriones, loros y pinguinos. Se debe permitir setear la altitud de vuelo para el caso de aves que vuelan)
El canto de los loros es grabado habitualmente por sus dueños en distintos formatos: MP3 y WMA.
Se espera llevar registro del tiempo total que las aves que vuelan han permanecido volando

Si una hacemos una clase Audio tiene los métodos “record_mp3” y “record_wma”, cada nuevo formato requiere modificar la clase. Mejor hacer una clase con cada formato (clase MP3, clase WMA, ambas subclases de Audio)

3.Liskov substitution

- Liskov Substitution Principle (LSP)

Los objetos deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del sistema o lo que es lo mismo: si en un programa utilizamos cierta clase, **deberíamos poder usar cualquiera de sus subclases** sin interferir en la funcionalidad del programa.

La L de SOLID alude al apellido de quien lo creó, Barbara Liskov, y dice que **“las clases derivadas deben poder sustituirse por sus clases base”.**

Si nada como un pato, vuela como un pato, pero necesita batería probablemente haya un problema de abstracción

Ejemplo

Violaciones:
El cuadrado si bien tiene características comunes a los rectángulos, podría tener características o comportamientos diferentes (por ejemplo, la forma de calcular algún atributo)

Solución:
Clase paralelogramo y sus subclases rectángulo y cuadrado

4.Interface segregation

- Interface Segregation Principle (ISP)
- Los clientes no deben ser forzados a depender de métodos que no usan
- El principio separa las interfaces que son muy grandes en interfaces más pequeñas y específicas, de manera que los clientes sólo tengan que conocer los métodos que les interesan.

En el cuarto principio de SOLID, *el tío Bob* sugiere: “Haz interfaces que sean específicas para un tipo de cliente”, es decir, para **una finalidad concreta**.
En este sentido, según el **Interface Segregation Principle (ISP),** es preferible contar con muchas interfaces que definan pocos métodos que tener una interface forzada a implementar muchos métodos a los que no dará uso.

Ejemplo

Violaciones:
Suponiendo que el registro de cliente no necesite enviar email está obligado a implementar el método de envío de email (aunque sin funcionalidad, poniendo por ejemplo comentarios)

Solución:
Armar diversas interfaces para cada registro

Dependency inversion

- Dependency Inversion Principle (DIP)
- 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. Los detalles deben depender de abstracciones.
- El objetivo es desacoplar a los componentes de alto nivel de los componentes de bajo nivel, de manera que sea posible la reutilización de los componentes de alto nivel al usar componentes de bajo nivel distintos

Llegamos al último principio: “**Depende de abstracciones**, no de clases concretas”.

Conclusión

Los principios SOLID son eso: principios, es decir, **buenas prácticas** que pueden ayudar a escribir un mejor código: más limpio, mantenible y escalable.
Como indica el propio Robert C. Martin en su artículo “Getting a SOLID start” **no se trata de reglas, ni leyes, ni verdades absolutas**, sino más bien soluciones de sentido común a problemas comunes. **Son heurísticos, basados en la experiencia**: “se ha observado que funcionan en muchos casos; pero no hay pruebas de que siempre funcionen, ni de que siempre se deban seguir.”