AWAKELAB

#programmingbootcamp

PRINCIPIOS SOLID

PRINCIPIOS SOLID

¿QUÉ VAMOS A VER?

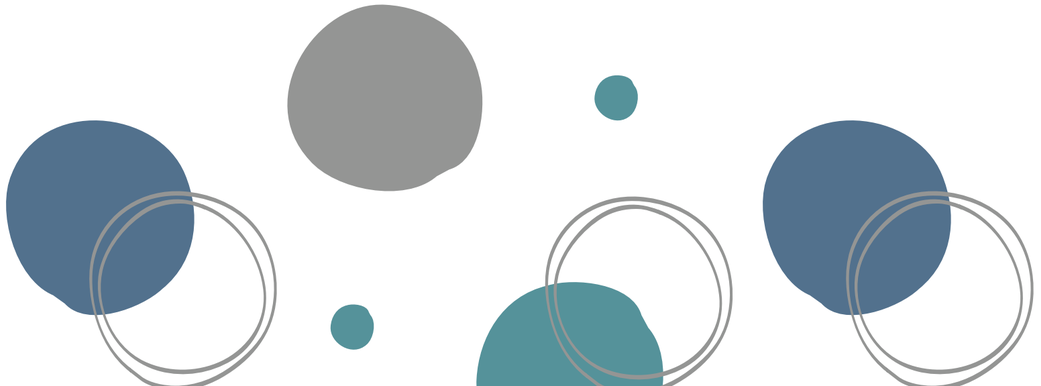
- Introducción.
- Cohesión y acoplamiento.
- Principio de Responsabilidad Única.
- Principio Abierto/Cerrado.
- Principio de Sustitución de Liskov.
- Principio de Segregación de Interfaces.
- Principio de Inversión de Dependencias.

PRINCIPIOS SOLID

- Son un conjunto de principios que nos ayudarán a escribir código de calidad en cualquier lenguaje de programación orientado a objetos.

- Si se aplican correctamente, será más fácil leer, testear y mantener el código.

- Los cinco principios son:
  - **S**ingle Responsibility Principle *(Principio de Responsabilidad Única).*
  - **O**pen/Closed Principle *(Principio Abierto/Cerrado).*
  - **L**iskov Substitution Principle *(Principio de Sustitución de Liskov).*
  - **I**nterface Segregation Principle *(Principio de Segregación de Interfaces).*
  - **D**ependency Inversion Principle *(Principio de Inversión de Dependencias).*



COHESIÓN

- Se refiere al grado en que los elementos de un módulo permanecen juntos.

- Se podría definir como lo estrecha que es la relación entre dos componentes. Por ejemplo, tendríamos una alta cohesión en caso de que las diferentes clases de un paquete están muy relacionadas entre sí, con un objetivo bastante claro.

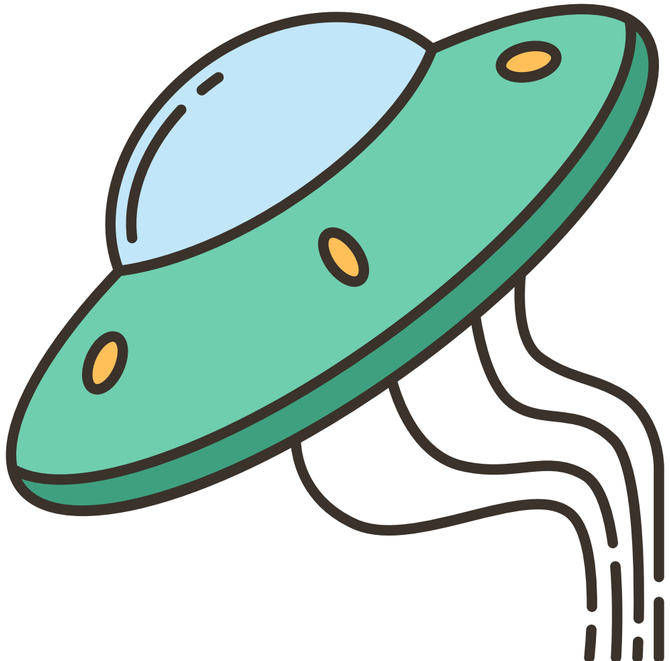
- Una alta cohesión se traduce en, robustez, fiabilidad, reutilización y alta comprensión del código, mientras que baja cohesión hace referencia a un software difícil de mantener, probar o reutilizar.

COHESIÓN

- Se considera una alta cohesión si:
  - Las funcionalidades integradas en una clase tienen mucho en común.
  - Los métodos realizan un pequeño número de actividades relacionadas.

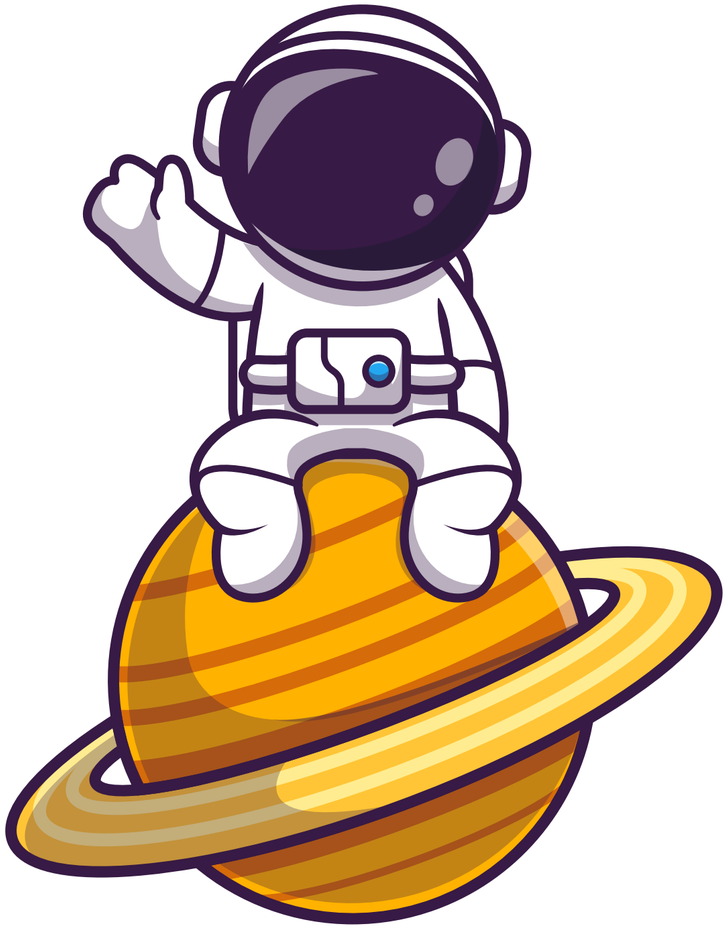
- Las ventajas de la cohesión alta son:
  - Menor número de operaciones.
  - Mejor posibilidad de aplicar mantención al sistema.
  - Aumento de la reutilización de módulos.

- En resumen, un código, módulo o proyecto con un alto nivel de cohesión es mucho más fácil de entender, mantener y por, sobre todo, comprender.



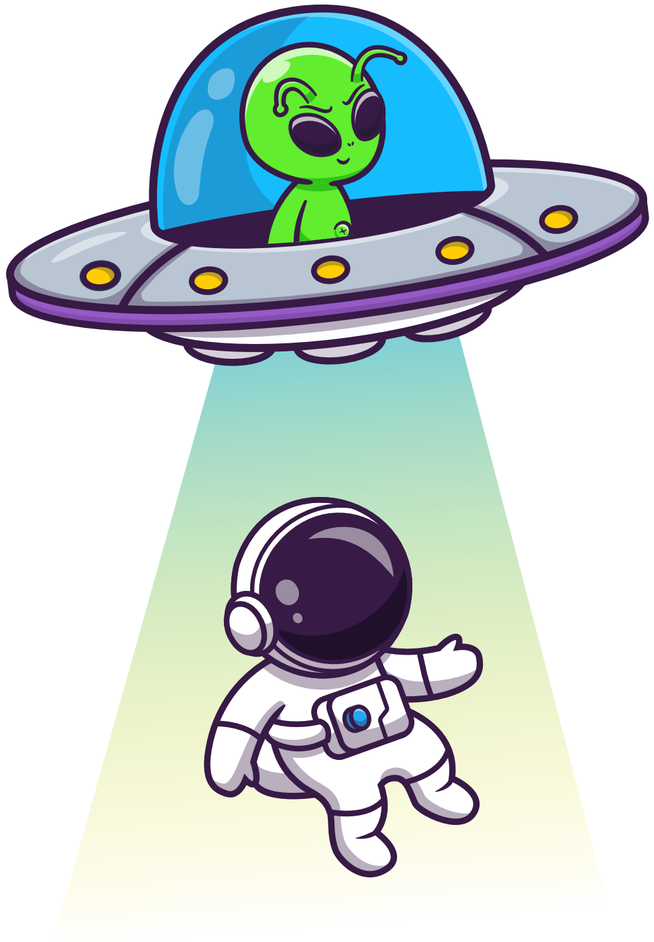
ACOPLAMIENTO

- Es la manera en que se relacionan los componentes entre ellos.
- Muchas relaciones y dependencias, se traduce en un grado de acoplamiento alto.
- Si tenemos componentes independientes, el grado de acoplamiento será bajo.
- Favorecer el bajo acoplamiento nos puede servir para tener un código mucho más fácil de entender, modificar o utilizar por separado. Con ello, favorecemos el desacoplamiento del código.



ACOPLAMIENTO

- El bajo acoplamiento nos permite:
  - Minimizar el riesgo de tener que cambiar múltiples archivos cuando debamos alterar algún dato.
  - Mejorar la capacidad de mantención del código.
  - Evita que el error de un módulo o bloque de código afecte a otro, siendo más fácil aislar el problema y solucionarlo.
  - Aumenta la reutilización del código.



PRINCIPIO DE RESPONSABILIDAD ÚNICA

- Señala que un objeto debe realizar una única cosa.

- No debemos caer en la mala práctica de cargar más de una responsabilidad lógica a las clases.

- Podemos darnos cuenta de que este principio está mal aplicado cuando una clase tiene demasiados métodos que no tienen ninguna relación entre ellos.

- El principio es simple, una clase solo debería ser modificada por una única razón, y no estar involucrada en cosas que le competen.

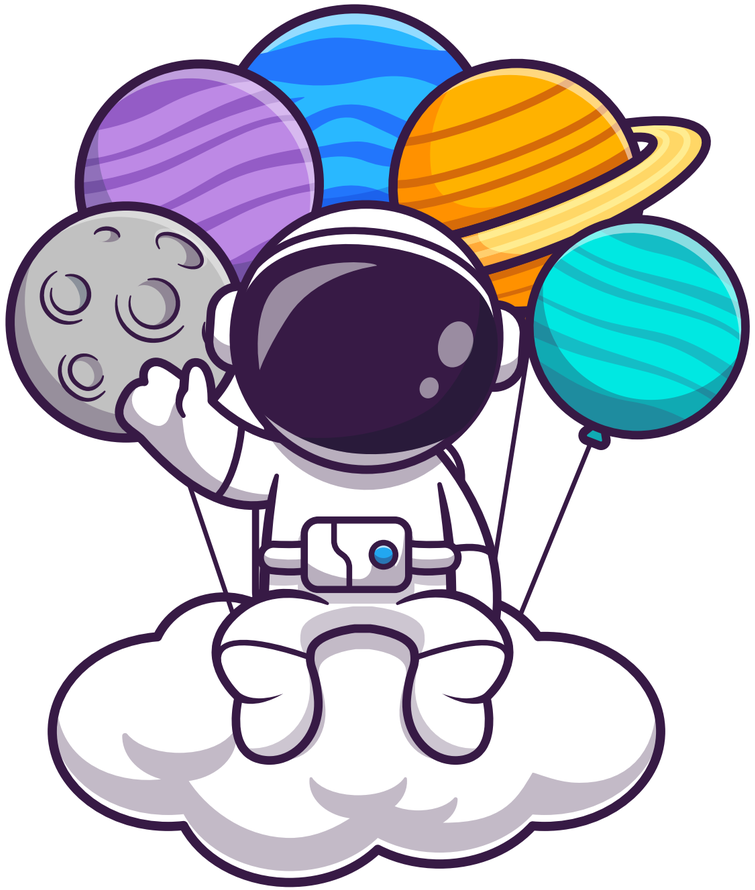
PRINCIPIO DE ABIERTO/CERRADO

- Este principio expone que una clase debe estar siempre abierta a extensiones, más no a ser modificada.

- Hace referencia a que se puede extender el comportamiento de una clase, pero no modificar su código para cambiarlo.

- Esto ayuda a seguir utilizando el código o darle otro enfoque, pero siempre por medio de nuevas clases o métodos que no modifiquen al previamente escrito.

- Si cada vez que necesitamos hacer un cambio, modificamos una clase, afectando al código que ya tenemos, nos podemos dar cuenta que este principio no se está cumpliendo.



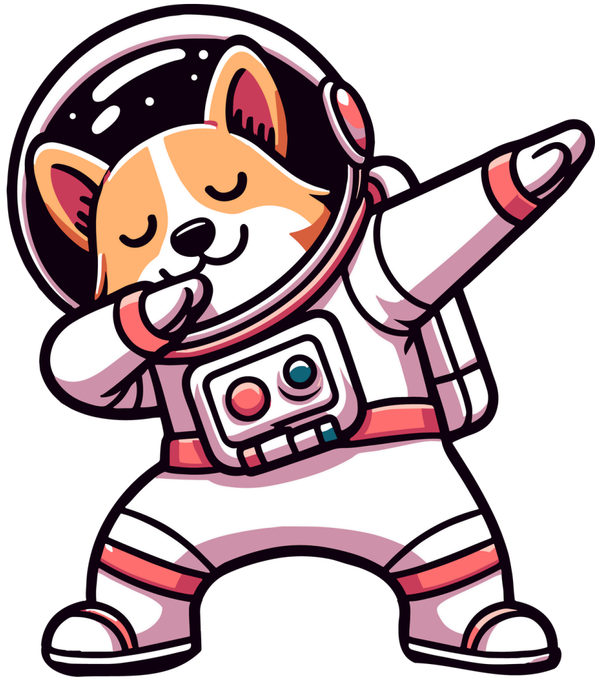
PRINCIPIO DE ABIERTO/CERRADO

- Este principio expone que una clase debe estar siempre abierta a extensiones, más no a ser modificada.

- Hace referencia a que se puede extender el comportamiento de una clase, pero no modificar su código para cambiarlo.

- Esto ayuda a seguir utilizando el código o darle otro enfoque, pero siempre por medio de nuevas clases o métodos que no modifiquen al previamente escrito.

- Si cada vez que necesitamos hacer un cambio, modificamos una clase, afectando al código que ya tenemos, nos podemos dar cuenta que este principio no se está cumpliendo.

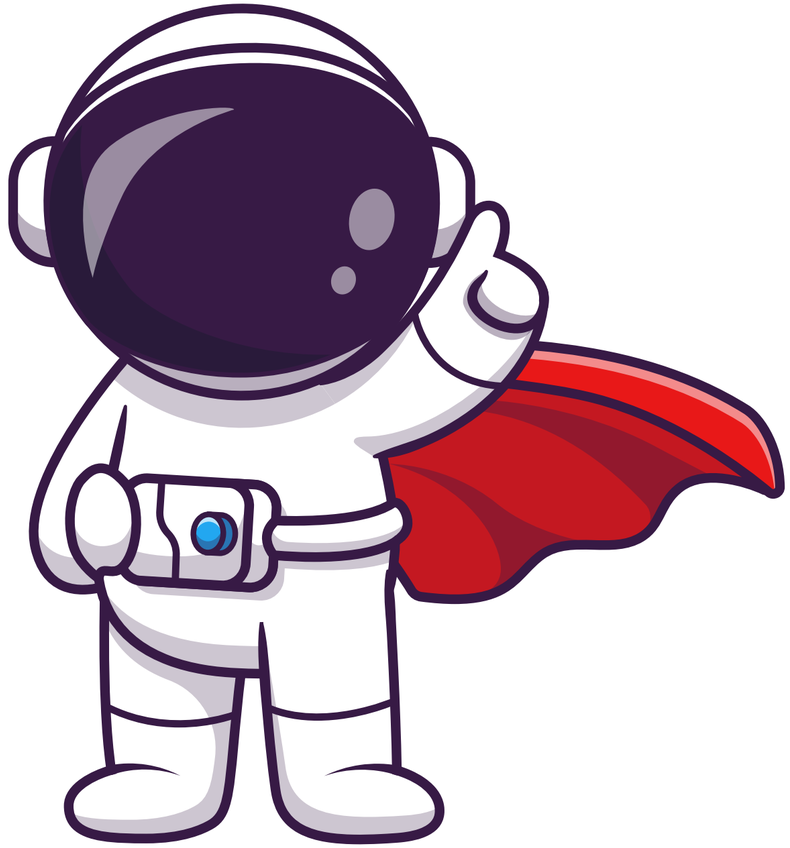


PRINCIPIO DE SUSTITUCIÓN DE LISKOV

- Este principio nos dice que, si una clase está siendo extendida, tenemos que poder utilizar cualquiera de sus clases hijas y que nuestro código funcione sin problemas.

- No debemos alterar el comportamiento de la clase padre cuando extendemos a una clase hija.

- Sus beneficios:
  - Código reutilizable.
  - Facilidad de entendimiento de jerarquías de clases.
  - Validar que las abstracciones del código sean correctas.



PRINCIPIO DE SEGREGACIÓN DE INTERFACES

- Ninguna clase debería depender de métodos que no esté utilizando.

- A la hora de asignar una interfaz, es importante tomar en cuenta que las clases que vayan a implementarla, realmente le estén dando un uso a todos sus métodos abstractos.

- En caso de que alguna clase o varias no utilicen métodos de una interfaz, deberíamos considerar segmentarlas para crear nuevas.

- Este método ayuda a que ninguna clase deba implementar métodos que no va a utilizar.

PRINCIPIO DE INVERSIÓN DE DEPENDENCIAS

- Establece que las clases de alto nivel no deberían depender de clases de bajo nivel. En realidad, ambos tipos de clases deberían depender de las abstracciones (interfaces o clases abstractas).

- Cuando los módulos dependen de abstracciones, es mucho más fácil modificar estas dependencias e implementarlas en el código, así también favorecemos la modularidad del mismo.

AWAKELAB

#programmingbootcamp