**SOLID** es un acrónimo acuñado por Robert C. Martin, en el cual se establecen los 5 principios básicos de la programación orientada a objetos (POO), estos principios tienen gran relación con los patrones de diseño ya que prevén una alta cohesión y un bajo acoplamiento.

Su objetivo es tener un buen diseño de software para que en fases de mantenimiento el código sea más legible y sencillo, y así con ello lograr crear nuevas funcionalidades sin tener que modificar a gran escala el código que ya se tiene.

**S- Una sola responsabilidad:** El objetivo de este principio es repartir las responsabilidades, haciendo que cada clase cumple la responsabilidad que debe cumplir y no agregar métodos que no pertenecen a esta clase. Por ejemplo, si requiero un método desde otra clase, la mejor manera de hacer el diseño seria tener una clase con ese método que pueda se llamados desde las demás clases que lo requieran, así evitamos que las clases realicen tareas que no son su responsabilidad.

**O- Abierto/Cerrado:** En este principio, las entidades de software (Clases, módulos, funciones, etc.) deben estar abiertas para extenderse, pero cerradas para su modificación.

Bajo esta premisa una clase debe ser fácilmente extensible, ósea abierta a nuevas funcionalidades, pero sin que se modifique su actual funcionamiento, por lo tanto, cerrado a modificaciones.

Una forma de lograr este principio es usando abstracción y polimorfismo, o con interfaces bien definidas de manera que se pueda ejecutar clases que implementen dicha interface.

**L- Sustitución de Liskov:** “Los objetos en un programa deben ser remplazables con instancias”.

Este principio nos indica que, si extendemos una clase, deberíamos poder usar cualquier subclase sin afectar el funcionamiento del programa, así se asegura qué al extender una clase, no modificamos el comportamiento del padre.

**I-Segregación de interfaces:** “Muchas interfaces especificas del cliente son mejor que una interfaz de propósito general”.

La idea es que una clase no implemente métodos que no va a usar, por ejemplo, aves, con una interface volar y comer. Pero no todas las aves vuelan y sería una implementación innecesaria para algunas.

**D-Inversión de dependencias:** A. Las clases de alto nivel no deben depender de las de bajo nivel, deberían depender de las abstracciones.

B. Las abstracciones no deberían depender del detalle, Los detalles deberían depender de las abstracciones.

Un ejemplo de su implementación es el patrón DTO. Teniendo una interface que nos indique los métodos para almacenar los datos, esto nos facilita que, si en algún momento ya no guardamos en una base de datos si no en otra, no debemos tocar el código si no abstraer esa interface en el nuevo servicio de almacenamiento.