# Principios SOLID

Acrónimo acuñado por Michael Feathers basándose en los principios de POO que Robert C. Martin recopilo a principios de la década del 2000 en su documento “Design Principles and Design Pattterns”

SOLID representa a 5 principios de la POO.

* S -> Single Responsability
* O -> Open-Closed
* L -> Liskov Substitution
* I -> Interface Segregation
* D -> Dependency Inversion

## Principio de Responsabilidad Única “Single Responsability”.

*Una clase debe tener una y solo una razón para cambiar, “Razón para Cambiar” es la responsabilidad.*

Una clase debería ser destinada a una única responsabilidad y no mezclas otras que no le imponen a su dominio (Su razón de ser).

Para lograr esto, debemos de reunir las cosas que cambian por las mismas razones y separa aquellas que cambiar por razones diferentes, “si voy a cambiar algo en mi clase y eso afecta a ciertos métodos, todo ese código va junto en una unida de código”.

El único problema que tendríamos es la gran cantidad de archivos, pero tenemos, un proyecto modular.

Ejemplo de un Caso de Uso:

***Crear una clase que nos permita crear un producto y registrar esa actividad para llevar un historial de acciones.***

*Tenemos dos actividades, “Crear un Producto” y “Registrar la Actividad”. Según el principio una clase solo debe hacer una sola responsabilidad, por ende, creamos una clase “LogService”, para el registro de la acción y otra “ProductService” para la creación de producto y la implementación de la función de “LogService”.*

*Ejemplo Codificado: Apps/SingleResponsability*

*Nota:*

*Si tenemos una clase con más de 10 métodos, se debe pensar ¿Cuánta responsabilidad tiene esta clase?, no será mejor sepáralo.*

## Principio de Abierto/Cerrado “Open-Closed”.

***Cada entidad de software debe estar abierta para extensión, pero cerrará para modificación.***

Nuestras clases que diseñemos deben estar abiertas para poder extenderse y cerradas para modificarse, *“Nuestras clases deben poder hacer cada vez más cosas” (Hoy tenemos una clase y mañana podrá hacer más funciones que mañana), el detalle de esto es que se debe extender la funcionalidad de la clase, pero sin modificarla.*

**Analogía:**

**“Tenemos una app en un servidor, mismo que tenemos de expandir ya sea horizontal (más servidores) o verticalmente (más recursos), pero en nuestro caso será horizontal, porque no se debe modificar la clase, pero si extender”**

Las clases deberán hacer más cosas, pero relacionadas con la responsabilidad, ejemplo: “Mi clase hoy registra un producto en Base Datos, mañana en un formato JSON”, se debe extender su única responsabilidad, no agregar más responsabilidades.

Observación:

El que una clase solo tenga una responsabilidad no significa que solo tendrá un método, en ocasiones lo habrá, como ejemplo tenemos el siguiente caso de uso:

“*Diseñar una clase que pueda registrar actividades hacia la consola, hacia un archivo y un futuro hacia otros destinos, ejemplo: el log de Windows o a través de correo electrónico.”*

La responsabilidad es “*pueda registrar actividades*”, entonces pondremos un método público que representara la responsabilidad, pero en este caso nos conviene tener los métodos de registros juntos por que forma parte de la misma responsabilidad, además estos serán usados por el método principal.

## Principio de sustitución de Liskov.

***Se puede definir como: “Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias”.***

*Si “S” es un subtipo “T”, entonces los objetos de “T” en un programa de computadora pueden ser sustituidos por los objetos de tipo S.*