

## Tarea

Investigar que es S.O.L.I.D.

31/03/2017

Brenda Abigail Bautista Cano

Salón: 409

Maestro: Miguel Salazar

Materia: Diseño Orientado a Objetos

## S.O.L.I.D

Es un acrónimo inventado por Robert C. Martin para establecer los 5 principios básicos de la programación orientada a objetos, tiene relación con los patrones de diseño, en especial, con la alta cohesión y el bajo acoplamiento.

### S → Single responsibility (Responsabilidad simple)

Este principio trata de destinar cada clase a una finalidad sencilla y concreta.

En muchas ocasiones queremos poner un método reutilizable que no tienen nada que ver con la clase simplemente ese momento pensamos "Para que voy a crear una clase para realizar esto, directamente lo pongo aquí".

El problema surge cuando tenemos la necesidad de utilizar ese mismo método desde otra clase. Si no se refactoriza en ese momento y se crea una clase destinada para la finalidad del método, nos toparemos a largo plazo con que las clases realizan tareas que no deberían ser de su responsabilidad.

Con la anterior mentalidad nos encontraremos, por ejemplo, con un algoritmo de formateo de números en una clase destinada a leer de la base de datos porque fue el primer sitio donde se empezó a utilizar. Esto conlleva a tener métodos difíciles de detectar y encontrar de manera que el código hay que tenerlo memorizado en la cabeza.

## O→Open/Closed (Abierto/Cerrado)

Principio atribuido a Bertrand Meyer que habla de crear clases extensibles sin necesidad de entrar al código fuente a modificarlo. Para conseguir este principio hay que tener muy claro cómo va a funcionar la aplicación, por donde se puede extender y cómo van a interactuar las clases.

El uso más común de extensión es mediante la herencia y la re implementación de métodos. Existe otra alternativa que consiste en utilizar métodos que acepten una interface de manera que podemos ejecutar cualquier clase que implemente ese interface. En todos los casos, el comportamiento de la clase cambia sin que hayamos tenido que tocar código interno.

Como ya he comentado llega un momento en que las necesidades pueden llegar a ser tan imprevisibles que nos topemos que con los métodos definidos en el interface o en los métodos extensibles, no sean suficientes para cubrir las necesidades. En este caso no habrá más remedio que romper este principio y refactorizar.

## L→Liskov substitution (Sustitución Liskov)

Este principio fue creado por **Barbara Liskov** y habla de la importancia de crear todas las clases derivadas para que también puedan ser tratadas como la propia clase base. Cuando creamos clases derivadas debemos asegurarnos de no re implementar métodos que hagan que los métodos de la clase base no funcionasen si se tratasen como un objeto de esa clase base.

## I→Interface segregation (Segregación del interface)

Este principio fue formulado por **Robert C. Martin** y trata de algo parecido al primer principio. Cuando se definen interfaces estos deben ser específicos a una finalidad concreta. Por ello, si tenemos que definir una serie de métodos abstractos que debe utilizar una clase a través de interfaces, es preferible tener muchos interfaces que definan pocos métodos que tener un interface con muchos métodos.

El objetivo de este principio es principalmente poder reaprovechar los interfaces en otras clases. Si tenemos un interface que compara y clona en el mismo interface, de manera más complicada se podrá utilizar en una clase que solo debe comparar o en otra que solo debe clonar.

## D→Dependency inversión (Inversión de dependencias)

También fue definido por **Robert C. Martin**. El objetivo de este principio conseguir desacoplar las clases. Un sistema no acoplado no hace nada pero un sistema altamente acoplado es muy difícil de mantener.

El objetivo de este principio es el uso de abstracciones para conseguir que una clase interactúe con otras clases sin que las conozca directamente. Es decir, las clases de nivel superior no deben conocer las clases de nivel inferior. Dicho de otro modo, no debe conocer los detalles. Existen diferentes patrones como la inyección **de** dependencias o service locator que nos permiten invertir el control.