U.A.N.L.

Facultad de Ciencias Físico Matemático

Materia: Cultura de Calidad

Maestro: Miguel Ángel Salazar Santillán

“S.O.L.I.D”

Alumno: José Israel Guerrero Ibarra

Matrícula: 1734152

San Nicolás de los Garza, Mty., Nuevo León

31/03/2017

Los Principios básicos de la programación y el diseño Orientado a Objetos

S.O.L.I.D

S.O.L.I.D es un acrónimo mnemónico introducido por Robert C. Martin a comienzos de la década del 2003 que representa los cinco principios básicos de la programación orientada a objetos y el diseño. Cuando estos principios se aplican en conjunto es más probable que un desarrollador cree un sistema que sea fácil de mantener y ampliar con el tiempo.

A continuación veremos cada uno de los acrónimos:

S – Principio de responsabilidad única (*Single Responsability Principle*)

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

Todos sus servicios deben estar estrechamente alineados con esa responsabilidad, y Robert C. Martin expresa el principio como: *"Una clase debe tener solo una razón para cambiar."*

O – Principio de Abierto/Cerrado (*Open/Closed Principle*)

Principio atribuido a Bertrand Meyer que habla de crear clases extensibles sin necesidad de entrar al código fuente a modificarlo. Es decir, el diseño debe ser abierto para poderse extender pero cerrado para poderse modificar. Aunque dicho parece fácil, lo complicado es predecir por donde se debe extender y que no tengamos que 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.

L – Principio de sustitución de Liskov (*Liskov substitution principle*)

El principio de sustitución de Liskov nos dice que si en alguna parte de nuestro código estamos usando una clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de las clases hijas y que el programa siga siendo válido. Esto nos obliga a asegurarnos de que cuando extendemos una clase no estamos alterando el comportamiento de la padre.

Este principio viene a desmentir la idea preconcebida de que las clases son una forma directa de modelar la realidad. Esto no siempre es así, y el ejemplo más típico es el de un rectángulo y un cuadrado. En breve veremos por qué.

La primera en hablar de él fue Bárbara Liskov (de ahí el nombre), una reconocida ingeniera de software americana.

Un claro ejemplo de cómo NO aplicamos este principio sería algo como crear una clase que extiende de otra, pero de repente uno de los métodos te sobra, y no sabes qué hacer con él. Las opciones más rápidas son bien dejarlo vacío, bien lanzar una excepción cuando se use, asegurándote de que nadie llama incorrectamente a un método que no se puede utilizar. Si un método sobrescrito no hace nada o lanza una excepción, es muy probable que estés violando el principio de sustitución de Liskov.

I – Principio de segregación de la interfaz (*Interface segregation principle*)

El principio de segregación de interfaces viene a decir que ninguna clase debería depender de métodos que no usa. Por tanto, cuando creemos interfaces que definan comportamientos, es importante estar seguros de que todas las clases que implementen esas interfaces vayan a necesitar y ser capaces de agregar comportamientos a todos los métodos. En caso contrario, es mejor tener varias interfaces más pequeñas.

Las interfaces nos ayudan a desacoplar módulos entre sí. Esto es así porque si tenemos una interfaz que explica el comportamiento que el módulo espera para comunicarse con otros módulos, nosotros siempre podremos crear una clase que lo implemente de modo que cumpla las condiciones. El módulo que describe la interfaz no tiene que saber nada sobre nuestro código y, sin embargo, nosotros podemos trabajar con él sin problemas.

Dentro del diseño orientado a objetos, las interfaces constituyen capas de abstracción que facilitan la descripción conceptual del código y que crean una barrera que impide dependencias.

Según muchos expertos de software que han firmado el Manifiesto para la Artesanía del Software, el hecho de que el software quede bien terminado y sea auto-explicativo es casi tan importante como hacer que funcione.4 En este sentido, usar interfaces para describir la funcionalidad del software suele ser buena idea.

Un sistema puede llegar a estar tan acoplado en múltiples niveles, que resulte imposible realizar una modificación de forma aislada, pues ello conllevaría muchas otras modificaciones colaterales.1 Utilizar una interfaz o clase abstracta puede impedir este "arrastre" de dependencias.

D – Principio de inversión de la dependencia (*Dependency inversion principle*)

Este principio es una técnica básica, y será el que más presente tengas en tu día a día si quieres hacer que tu código sea testeable y mantenible. Gracias al principio de inversión de dependencias, podemos hacer que el código que es el núcleo de nuestra aplicación no dependa de los detalles de implementación, como pueden ser el framework que utilices, la base de datos, cómo te conectes a tu servidor… Todos estos aspectos se especificarán mediante interfaces, y el núcleo no tendrá que conocer cuál es la implementación real para funcionar.

La definición que se suele dar es:

> A. Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.
>
> B. Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

¿Cómo detectar que estamos violando el Principio de inversión de dependencias?

Este es muy fácil: cualquier instanciación de clases complejas o módulos es una violación de este principio. Además, si escribes tests te darás cuenta muy rápido, en cuanto no puedas probar esa clase con facilidad porque dependan del código de otra clase.