

**Tema:** Principios SOLID

**Curso:** Programacion I.

**Profesor:** Gustavo Coronel Castillo.

**Autor:**

- Ayala Villavicencio, Alvaro.

Fecha de entrega:

06/10/2017

**Indice:**

1. Resumen…………………………………………………………..….3

2. Antecedentes…………………………………………………….......3

3. Principios SOLID……………………………………………………..4

3.1. Principio de Responsabilidad Única…………………………..4

3.2. Principio Open/Closed……………………..............................5

3.3. Principio de Sustitución de Liskov………………………….....6

3.4. Principio de Segregación de Interfaces……………………....6

3.5. Principio de Inversión de Dependencias…………………......7

4. Conclusiones………………………………………………………….9

5. Recomendaciones……………………………………………………9

6. Bibliografía………………………………………………………….....9

**Resumen:**

Son cinco principios fundamentales, uno por cada letra, que hablan del diseño orientado a objetos en términos de la gestión de dependencias. Las dependencias entre unas clases y otras son las que hacen al código más frágil o más robusto y reutilizable. El problema con el modelado tradicional es que no se ocupa en profundidad de la gestión de dependencias entre clases sino de la conceptualización. Quién decidió resaltar estos principios y darles nombre a algunos de ellos fueRobert C. Martin, allá por el año 1995.  
SOLID es un acrónimo en inglés de:

**S: Principio de Responsabilidad Única** (Single Responsability Principle)  
"Un objeto debe tener una única responsabilidad"

**O: Principio de Abierto / Cerrado** (Open / Closed Principle)  
"las entidades de software deben ser abiertas a la extensión y cerradas a la modificación"

**L: Principio de Sustitución de Liskov** (Liskov Substitution Principle)  
"los objetos de un programa deben poder reemplazarse por instancias de sus subtipos sin alterar la correctitud del programa"

**I: Principio de Segregación de Interfaces** (Interface Segregation Principle)  
"es preferible muchas interfaces específicas de cliente que una interfaz de uso general"

**D: Principio de Inversión de Dependencias** (Dependency Inversion Principle)  
"debemos depender de las abstracciones y no de las concreciones"

**Antecedentes:**

Me intereso investigar sobre este tema ya que al entrar en el curso de Programacion I y ver una nueva manera de enseñanza y que esta integraba los principios SOLID, que hacen que el código sea mas limpio y legible ante uno, ya que en la practica nos dimos cuenta que podiamos mejorar de muchas maneras los códigos que planteabamos tomando en cuenta estos cinco principios SOLID.

**Principios Solid:**

Solides un acrónimo inventado por RobertC. Martin para establecer los cinco principios básicos de la programación orientada a objetos y diseño. Este acrónimo tiene bastante relación con los patrones de diseño, en especial, con la alta cohesión y el bajo acoplamiento.

El objetivo de tener un buen diseño de programación es abarcar la fase de mantenimiento de una manera más legible y sencilla, así como conseguir crear nuevas funcionalidades sin tener que modificar en gran medida código antiguo. Los costes de mantenimiento pueden abarcar el 80% de un proyecto de software por lo que hay que valorar un buen diseño.

Son 5 principios sólidos:

- Principio de Responsabilidad Única.

- Principio Open/Closed.

- Principio de Sustitución de Liskov.

- Principio de Segregación de Interfaces.

- Principio de Inversión de Dependencias.

A continuación, explicaremos cada uno de estos principios tomando en cuenta sus objetivos, errores recurrentes y posibles soluciones.

**1. Principio de Responsabilidad Única:**

El Principio de responsabilidad única es el primero de los cinco que componen SOLID. El principio de Responsabilidad Única nos viene a decir que un objeto debe realizar una única cosa. Es muy habitual, si no prestamos atención a esto, lo más probable es que acabemos teniendo clases que tienen varias responsabilidades lógicas a la vez.

Pero surge la pregunta ¿Cómo podemos saber si estamos violando el principio de Responsabilidad Única?

Existen muchos casos como: que en una misma clase están involucradas dos capas, que tengas demasiados métodos públicos, los métodos que usan cada uno de los campos de la clase, por el número de imports que tenemos, se nos hace difícil testear o probar la clase, cada vez que escribes una nueva función esa clase se ve afectada, por el número de líneas que tiene, etc.

En conclusión, el principio de Responsabilidad Única es una herramienta indispensable para proteger nuestro código frente a cambios, ya que implica que sólo debería haber un motivo por el que modificar una clase.

**2. Principio Open/Closed:**

El principio Open/Closed fue nombrado por primera vez por Bertrand Mayer, un programador francés, quien lo incluyó en su libro *Object Oriented Software Construction* en 1988.

El principio Open/Closed nos dice que una entidad de software debería estar abierta a extensión, pero cerrada a modificación. Es decir que tenemos que ser capaces de extender el comportamiento de nuestras clases sin necesidad de modificar su código. Esto nos ayuda a seguir añadiendo funcionalidad con la seguridad de que no afectará al código existente. Nuevas funcionalidades implicarán añadir nuevas clases y métodos, pero en general no debería suponer modificar lo que ya ha sido escrito.  
La forma de llegar a ello está muy relacionada con el principio anterior. Si las clases sólo tienen una responsabilidad, podremos añadir nuevas características que no les afectarán. Esto no quiere decir que cumpliendo el primer principio se cumpla automáticamente el segundo, ni viceversa.   
Muchas veces se suele resolver utilizando polimorfismo. En vez de obligar a la clase principal a saber cómo realizar una operación, delega esta a los objetos que utiliza, de tal forma que no necesita saber explícitamente cómo llevarla a cabo. Estos objetos tendrán una interfaz común que implementarán de forma específica según sus requerimientos.

Pero ¿Cómo nos damos cuenta de que estamos violando este principio?

Una de las maneras más sencillas es de detectarlo es darnos cuenta de qué clases modificamos más a menudo. Si cada vez que hay un nuevo requisito o una modificación de los existentes, las mismas clases se ven afectadas, podemos empezar a entender que estamos violando este principio.

En conclusión, el principio Open/Closed es una herramienta indispensable para protegernos frente a cambios en módulos o partes de código en los que esas modificaciones son frecuentes. Tener código cerrado a modificación y abierto a extensión nos da la máxima flexibilidad con el mínimo impacto.

**3. Principio de Sustitución de Liskov:**

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

Pero ¿Cómo nos damos cuenta de que estamos violando el principio de Sustitución de Liskov?

Si un método sobrescrito no hace nada o lanza una excepción, es muy probable que estés violando el principio de sustitución de Liskov. Otra herramienta que te avisará fácilmente son los tests. Si los tests de la clase padre no funcionan para la hija, también estarás violando este principio.

En conclusión, el principio de Liskov nos ayuda a utilizar la herencia de forma correcta, y a tener mucho más cuidado a la hora de extender clases. En la práctica nos ahorrará muchos errores derivados de nuestro afán por modelar lo que vemos en la vida real en clases siguiendo la misma lógica. No siempre hay una modelización exacta, por lo que este principio nos ayudará a descubrir la mejor forma de hacerlo.

**4. Principio de Segregación de Interfaces:**

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.

Pero llegamos a tener un problema y surge cuando esas interfaces intentan definir más cosas de las debidas, lo que se denominan *fat interfaces*. Probablemente ocurrirá que las clases hijas acabarán por no usar muchos de esos métodos, y habrá que darles una implementación.

Pero ¿Cómo nos damos cuenta de que estamos violando el principio de Segregación de Interfaces?

Si al implementar una interfaz ves que uno o varios de los métodos no tienen sentido y te hace falta dejarlos vacíos o lanzar excepciones, es muy probable que estés violando este principio. Si la interfaz forma parte de tu código, divídela en varias interfaces que definan comportamientos más específicos.

Pero si ya tienes un coddigo que utiliza fat interfaces, la solución puede ser utilizar el patrón de diseño Adapter. El patrón Adapter nos permite convertir unas interfaces en otras, por lo que puedes usar adaptadores que conviertan la interfaz antigua en las nuevas.

En conclusión, el principio de segregación de interfaces nos ayuda a no obligar a ninguna clase a implementar métodos que no utiliza. Esto nos evitará problemas que nos pueden llevar a errores inesperados y a dependencias no deseadas. Además, nos ayuda a reutilizar código de forma más inteligente.

**5. Principio de Inversión de Dependencias:**

También fue definido por Robert C. Martin. 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. El objetivo de este principio conseguir desacoplar las clases. En todo diseño siempre debe existir un acoplamiento, pero hay que evitarlo en la medida de lo posible. 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 interactue 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 lainyección de dependencias o service locator que nos permiten invertir el control.

Los problemas más comunes que se suscitan son que: las partes más genéricas de nuestro código dependerá por todas partes de detalles de implementación, que las dependencias no queden claras y que es muy complicado el realizar pruebas de estos.

Pero ¿Cómo nos damos cuenta de que estamos violando el principio de Inversión de Dependencias?

Cualquier instanciación de clases complejas o módulos es una violación de este principio. Además, si haces tests te darás cuenta muy rápido, en cuanto no puedas probar esa clase con facilidad porque dependan del código de otra clase. Y para resolverlo tendrás que utilizar alguna de las alternativas que existen para suministrarle esas dependencias. Aunque hay varias, las que más se suelen utilizar son mediante constructor y mediante setters (funciones que lo único que hacen es asignar un valor).

En conclusión, este principio nos obliga a organizar nuestro código de una manera muy distinta a como estamos acostumbrados, y en contra de lo que la lógica dicta inicialmente, pero a la larga compensa por la flexibilidad que otorga a la arquitectura de nuestra aplicación.

**Conclusiones:**

En conclusión, con lo investigado se hace hincapié en el uso de los principios SOLID al momento de programar, de tomar en cuenta todos los principios para así poder generar códigos, fáciles de interpretar y de adaptarse o modificarse.

Estos cinco principios contribuyen mucho ya que nos ayudan a saber cuándo un objeto debe tener responsabilidad única, que las entidades de software sean abiertas a la extensión y cerradas a la modificación, la fácil modificación de objetos por instancias sin alterar el correcto funcionamiento del programa, el uso de varias interfaces específicas en contraste de una general y que debemos depender de las abstracciones y no de las concreciones.   
Para mí el lograr entender y aplicar estos principios y lograr que mejore mi manera de programar es de mucha ayuda y considero que todos deberían de tomar en cuenta siempre estos principios a la hora de hacer programación orientada a objetos.

**Recomendaciones:**

Se recomienda el constante uso y práctica en programación de los principios SOLID ya que son una base muy importante para obtener y generar códigos limpios, fáciles de entender y que se puedan modificar sin alterar el funcionamiento del programa.

**Video en YouTube:** https://youtu.be/lV5ozXqBxiI

**Bibliografía:**

De Seta, L. (2013). “Los principios SOLID para diseño de objetos”. 4 de febrero del 2013. Recuperado: https://dosideas.com/noticias/desarrollo-de-software/968-los-principios-solid-para-diseno-de-objetos

Leiva, A. (2015). “Principio de responsabilidad única (SOLID 1ª parte)”. 31 de diciembre del 2015. Recuperado: https://devexperto.com/principio-responsabilidad-unica/

Leiva, A. (2016). “Principio Open/Closed (SOLID 2ª parte)”. 7 de enero del 2016. Recuperado:   
https://devexperto.com/principio-open-closed/

Leiva, A. (2016). “Principio de sustitución de Liskov (SOLID 3ª parte)”. 14 de enero del 2016. Recuperado: https://devexperto.com/principio-de-sustitucion-de-liskov/

Leiva, A. (2016). “Principio de Segregación de Interfaces (SOLID 4ª parte)”. 21 de enero del 2016. Recuperado: https://devexperto.com/principio-de-segregacion-de-interfaces/

Leiva, A. (2016). “Principio de Inversión de Dependencias (SOLID 5ª parte)”. 28 de enero del 2016. Recuperado: https://devexperto.com/principio-de-inversion-de-dependencias/

Rubira, J. (2011). “Solid, cinco principios básicos de diseño de clases”. 14 de julio del 2011. Recuperado: https://www.genbetadev.com/paradigmas-de-programacion/solid-cinco-principios-basicos-de-diseno-de-clases