**Primer Tarea de Programación lll**

**CURSO:** Programación III

**CICLO:** II Ciclo 2021

**CARRERA:** Ingeniería en Sistemas

**PROFESOR:** Rubén Mora Vargas

**TEMA:** Los 5 principios SOLID

**FECHA:** 04 / 09 / 2021

**GRUPO:** 89

**ESTUDIANTES**

Anthony Mora Valverde

Kervin Muñoz Mejías

**CORREO ELECTRÓNICO**

[kervin.munoz.mejias@est.una.ac.cr](mailto:kervin.munoz.mejias@est.una.ac.cr)

[anthony.mora.valverde@est.una.ac.cr](mailto:anthony.mora.valverde@est.una.ac.cr)

**Índice**

## **Introducción**

# 

# **Contenido**

¿Qué son los principios SOLID?

SOLID es un acrónimo que reúne los cinco principios básicos de la programación orientada a objetos definidos por **Robert C. Martin**, si estos se usan correctamente, serán de gran ayuda para desarrollar software de calidad en cualquier lenguaje de programación orientada a objetos.

**Los cinco principios SOLID** en el diseño de aplicaciones de software son:

* Principio de Responsabilidad Única (Single Responsibility Principle).
* Principio de Abierto/Cerrado (Open/Closed Principle).
* Principio de Sustitución de Liskov (Liskov Substitution Principle).
* Principio de Segregación de la Interfaz (Interface Segregation Principle).
* Principio de Inversión de Dependencias (Dependency Inversion Principle).

Estos principios no dejan de ser una manera más de hacer las cosas, que a largo plazo hará que tus proyectos sean más robustos, estables, además de que te permiten crear un código limpio y flexible ante la posibilidad de cualquier cambio, esto hace que se pueda modificar de una manera más sencilla dependiendo de la necesidad, permitiendo que sea mantenible, reutilizable y escalables a futuro, aceptando ser ampliado con funcionalidades nuevas agilizando dicho proceso.

A continuación, se explicará de una manera más detallada en qué consisten cada una de estas bases:

**Single Responsability Principle (SRP)**

*“Una clase debe tener solo una razón para cambiar” (Robert C. Martin).*

Lo que este principio propone es destinar una única responsabilidad a cada clase perteneciente a nuestro software, esta responsabilidad debe ser concreta y estar definida. Cada uno de los métodos implementados deben estar relacionados con la finalidad de la clase.

Al encontrarse con una clase que se encuentra conformada por una serie de métodos que realizan funciones con diferentes finalidades, es conveniente reestructurar el código y crear nuevas clases que se encarguen de cumplir con los objetivos del método.

No es una tarea sencilla definir una única responsabilidad a cada clase, para esto será necesario analizar previamente las funciones que se necesitan implementar y de qué manera se encontrara estructurada la aplicación a desarrollar.

Este principio es de suma importancia para evitar que el cambio en alguna responsabilidad provoque la modificación en otra y que esta situación genere errores en nuestra aplicación.

Algunas maneras de darse cuenta si no se está cumpliendo con este principio podrían ser las siguientes:

* Una clase es demasiado extensa, posee muchas líneas de código.
* Al realizar una nueva funcionalidad o modificación, es necesario hacer el cambio en muchos sitios distintos.
* Existen muchas relaciones para una sola clase.
* Se mezclan funciones de distintas capas del proyecto

**Open/Closed Principle (OCP)**

En este principio se establece que: *“Las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas para extensión, pero cerradas para modificación” (Bertrand Meyer).*

A esto se refiere a que una entidad de software debe ser capaz de adaptarse a cambios y nuevas necesidades según la aplicación lo requiera, pero lo adaptabilidad de dicha entidad no debe surgir como resultado de la modificación, sino como resultado de un diseño que permita la extensión sin necesidad de hacer modificaciones.

Algunas ventajas de diseñar el código aplicando este principio, son las siguientes:

* Un software fácil de mantener, al minimizar la cantidad de cambios en la base del código de la aplicación.
* Permite ampliación de funcionalidades sin modificar partes básicas que fueron probadas.
* Evita la generación de nuevos errores en funcionalidades que ya se encontraban desarrolladas, probadas y funcionando en correcto estado.

Teniendo en cuenta esta regla, si el día de mañana el cliente solicita un requisito nuevo para algo que ya se encuentra desarrollado, el comportamiento de esa clase debería ser extendido, nunca modificado. En caso de no ser así la probabilidad de encontrarse con inconvenientes es muy alta.

**Liskov Substitution Principle (LSP)**

El tercer principio de la programación orientada a objetos recibe su nombre gracias a **Barbara Liskov,** reconocida ingeniera de software, fue la primera mujer de los Estados Unidos en conseguir un doctorado en Ciencias de la Computación, además ganadora de un premio Turing.

Barbara Liskov y Jeannette Wing de manera conjunta formularon lo siguiente:

*“Sea ϕ(x) una propiedad comprobable acerca de los objetos x de tipo T. Entonces ϕ(y) debe ser verdad para los objetos y del tipo S donde S, es un subtipo de T.”*

El principio de Liskov brinda una serie de pasos a seguir al realizar la herencia entre clases. Lo principal que se debe cumplir para realizar una herencia de manera correcta es que cada clase que hereda de otra tenga la posibilidad de usarse como su padre.

De tal manera que si al sobrescribir un método en una clase hija y se necesita lanzar una excepción o esta función no realiza nada, entonces es probable que se esté violando el principio LSP.

**Interface Segregation Principle (ISP)**

*“Los clientes no deben verse obligados a depender de métodos que no utilicen.” (Robert C. Martin)*

Este principio puede definirse como la priorización de crear múltiples interfaces específicas, en lugar de tener una sola interfaz de uso general. Es mejor crear una nueva interfaz y que las clases puedan implementar las interfaces que necesiten a utilizarlas de métodos que sus clientes no necesitaran. De esta manera se vuelve más sencillo realizar reestructuraciones y modificaciones.

Las interfaces brindan una capa de abstracción que ayudan a desvincular de otros módulos. De tal forma, que la interfaz defina el comportamiento que nuestro código desea para lograr comunicarse con otros módulos, por medio de métodos y propiedades.

Al crear interfaces más pequeñas, se obtiene la ventaja de que al ser necesario crear nuevas implementaciones de la interfaz para un caso de uso en específico, no será necesario implementar el resto de los métodos que no se encuentran relacionados con ese caso de uso.

Se puede concluir de este principio que ninguna clase debería depender de métodos que no utiliza. Por lo tanto, al crear interfaces con comportamientos definidos, es de suma importancia asegurarse de que todas las clases que implementen dichas interfaces vayan a necesitar y tengan la capacidad de agregar acciones a todos los métodos. De no ser así, lo mejor es contar con otras interfaces de menor tamaño, ya que cuando un código depende involuntariamente de elementos o implementa métodos que no se necesitan, puede traer problemas inesperados.

**Dependency Inversion Principle (DIP)**

En este principio se especifica de que manera deben ser las relaciones entre los componentes, para evitar la unión entre los distintos módulos de un software.

La definición dada por Robert C. Martin esta compuesta por dos argumentos:

* *Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.*
* *Las abstracciones no deben depender de detalles (implementaciones concretas). Los detalles deben depender de abstracciones.*

En el desarrollo de grandes aplicaciones llega un momento en el que se cuenta con una gran multitud de módulos. Para evitar que se produzca un fuerte acoplamiento entre los distintos componentes y se desarrolle una gran dependencia de las librerías empleadas, es recomendable aplicar el principio DIP.

Al poner en práctica estas reglas, el resultado es una arquitectura donde las dependencias se encuentran invertidas con respecto a la manera tradicional de pensar en la programación orientada a objetos, se reduce la unión de entre los módulos de alto nivel y los módulos de bajo nivel al agregar una clase abstracta entre ambos.

La razón por la cual se invierten las dependencias utilizando clases abstractas e interfaces es porque estas son más estables que implementaciones concretas, de tal forma que, si tuviésemos que realizar un cambio en la lógica de la aplicación, se aplicarían los cambios solo en la clase afectada, evitando de esta manera realizar modificaciones en otros módulos.

Aplicando el principio de inversión de dependencias se logra que las clases de alto nivel no realicen su trabajo directamente con las clases de bajo nivel, para ello se utiliza interfaces que crean una capa abstracta en medio de ambos componentes, dando a la arquitectura de la aplicación una mayor flexibilidad y menor acoplamiento, características que a largo plazo pueden ser indispensables para la mantenibilidad del software.