**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** 3](#_Toc84176160)

[**Contenido** 4](#_Toc84176161)

[**Conclusiones** 13](#_Toc84176162)

[**Recomendaciones** 13](#_Toc84176163)

[**Referencias** 14](#_Toc84176164)

## **Introducción**

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): en este principio establece que un componente o clase debe tener una responsabilidad única, sencilla y concreta.
* Principio de Abierto/Cerrado (Open/Closed Principle): para el segundo se establece que los componentes del software deben estar abiertos para extender a partir de ellos, pero cerrados para evitar que se modifiquen.
* Principio de Sustitución de Liskov (Liskov Substitution Principle): bajo este principio se establece que una subclase puede ser sustituida por su superclase. Es decir, es posible crear una subclase llamada Auto, la cual deriva de la superclase Vehículo. Si al usar la superclase el programa falla, este principio no se cumple.
* Principio de Segregación de la Interfaz (Interface Segregation Principle): este principio establece que los clientes no deben ser forzados a depender de interfaces que no utilizan. Es importante que cada clase implemente las interfaces que va a utilizar. De este modo, agregar nuevas funcionalidades o modificar las existentes será más fácil.
* Principio de Inversión de Dependencias (Dependency Inversion Principle): para el último principio se establece que los módulos de alto nivel no deben de depender de los de bajo nivel. En ambos casos deben depender de las abstracciones.

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.

Para todo aspirante a ingeniero en software, es necesario aprender nuevos conceptos durante toda su carrera. En la actualidad las tecnologías avanzan a un nivel increíble, lo cual causa que los centros académicos no actualicen sus enseñanzas al mismo ritmo, por este motivo la enseñanza de las instituciones ha cambiado de enseñar nuevas tecnologías a enseñar principios, ya que con los principios es posible crear un programar capaz de trabajar casi en cualquier lenguaje existente.

# **Contenido**

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

Considera el siguiente ejemplo:

class Empresa {  
   
 String nombre;  
 Empresa(String nombre){this.nombre = nombre;}  
 String getNombreEmpresa(){return nombre;}  
 void guardarEmpresaDB(Empresa empresa){//Guarda empresa en base de datos}  
}

En el anterior ejemplo se viola el principio de responsabilidad única, ya que la clase Empresa permite tanto el acceso a las propiedades de dicha clase, como a realizar operaciones dentro de la base de datos, por lo tanto, esta clase ya tiene más de una responsabilidad.

Para evitar esto, se debe separar las responsabilidades de la clase creando otra clase que se encargue de realizar las operaciones a la base de datos:

class Empresa {  
  
 String nombre;  
 Empresa(String nombre){this.nombre = nombre;}  
 String getNombreEmpresa(){return nombre;}  
}  
  
class EmpresaDB{

void guardarEmpresaDB(Empresa empresa){//Guarda empresa en base de datos}  
 void eliminarEmpresaDB(Empresa empresa){//Elimina empresa en base de datos}  
}

**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.

Siguiendo con el ejemplo de empresas:

class Empresa {  
  
 String nombre;  
  
 Empresa(String nombre){this.nombre = nombre;}  
  
 String getNombreEmpresa(){return nombre;}  
}

Si se quisiera recorrer una lista de empresas e imprimir sus nombres en pantalla:

public static void imprimirGananciasPromedioEmpresa(Empresa[] arrayEmpresas){  
 for (Empresa empresa : arrayEmpresas) {  
 if(empresa.nombre.equals("McDonalds")) System.out.println(18000);  
 if(empresa.nombre.equals("CocaCola")) System.out.println(25000);  
 }

}

public static void main(String[] args) {  
 Empresa[] arrayEmpresas = {  
 new Empresa("McDonalds"),  
 new Empresa("CocaCola")  
 };  
 imprimirGananciasPromedioEmpresa(arrayEmpresas);  
}

Esto no cumpliría el principio, ya que si se agregar una empresa nueva con otro nombre:

public static void main(String[] args) {  
 Empresa[] arrayEmpresas = {  
 new Empresa("McDonalds"),  
 new Empresa("CocaCola")  
 new Empresa("Waltmart")  
 };  
 imprimirGananciasPromedioEmpresa(arrayEmpresas);  
}

También seria necesario modificar el método creado anteriormente:

public static void imprimirGananciasPromedioEmpresa(Empresa[] arrayEmpresas){  
 for (Empresa empresa : arrayEmpresas) {  
 if(empresa.nombre.equals("McDonalds")) System.out.println(18000);  
 if(empresa.nombre.equals("CocaCola")) System.out.println(25000);  
 if(empresa.nombre.equals("Waltmart")) System.out.println(22000);  
 }  
}

Para lograr cumplir con este principio se podría hacer lo siguiente:

abstract class Empresa {  
   
 abstract int gananciasPromedioEmpresas();  
}  
  
class McDonalds extends Empresa {  
 @Override  
 int gananciasPromedioEmpresas() { return 18000; }  
}  
  
class CocaCola extends Empresa {  
 @Override  
 int gananciasPromedioEmpresas() { return 25000; }  
}  
  
class Waltmart extends Empresa {  
 @Override  
 int gananciasPromedioEmpresas() { return 22000; }  
}  
  
public static void main(String[] args) {  
  
 Empresa[] arrayEmpresas = {new McDonalds(),new CocaCola(),new Waltmart()};  
 imprimirGananciasPromedioEmpresas(arrayEmpresas);  
}  
  
public static void imprimirGananciasPromedioEmpresas(Empresa[] arrayCoches){  
 for (Empresa empresa : arrayEmpresas) {  
 System.out.println(empresa.gananciasPromedioEmpresas());  
 }  
}

Cada empresa extiende de la clase abstracta Empresa e implementa el método abstracto gananciasPromedioEmpresas ().

Ahora si se agrega una nueva empresa, el método gananciasPromedioEmpresas () no tendrá que ser modificado. Solo se añade la nueva empresa al array, cumpliendo de esta manera el principio abierto/cerrado.

**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.

Supongamos que tenemos la clase pájaro:

class Pajaro{  
  
 void volar(){}  
 void comer(){}  
}

Ahora se agrega a la aplicación un pato y un pingüino:

class Pato extends Parajo{}  
class Pinguino extends Pajaro{}

El problema que se evidencia es que, si bien el pingüino es un pájaro, no posee la habilidad de volar como el pato. Por ende, se tendría un método en pingüino que se podrá implementar. Una posible solución a esta violación del principio LSP sería:

class Ave{  
  
 void comer(){}  
  
}  
  
class AveVolador extends Ave{  
  
 void volar(){}  
  
}  
  
class Pato extends AveVolador {}  
class Pinguino extends Ave {}

**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.

Se desea crear las clases necesarias para albergar algunos tipos de aves. Por ejemplo, se tendrían loros y pingüinos:

interface Ave {  
 void volar();  
 void comer();  
 void nadar();  
}  
  
class Loro implements Ave{  
  
 @Override  
 public void volar() {//...}  
 @Override  
 public void comer() {//...}  
 @Override  
 public void nadar() {//...}  
}  
  
class Pinguino implements Ave{  
  
 @Override  
 public void volar() {//...}  
 @Override  
 public void comer() {//...}  
 @Override  
 public void nadar() {//...}  
}

El problema es que el loro no nada, y el pingüino no vuela, por lo que se tendría que añadir una excepción o aviso si se intenta llamar a estos métodos. Además, si se quisiera añadir otro método a la interfaz Ave, tendríamos que recorrer cada una de las clases que la implementa e ir añadiendo la implementación de dicho método en todas ellas. Esto viola el principio de segregación de interfaz, ya que en estas clases los clientes no tienen por qué depender de métodos que no usan.

Lo más correcto sería segregar más las interfaces, tanto como sea necesario. En este caso se podría hacer lo siguiente:

interface Ave {  
 void comer();  
}  
interface AveVoladora {  
 void volar();  
}  
  
interface AveNadadora {  
 void nadar();  
}  
  
class Loro implements Ave, AveVoladora{  
  
 @Override  
 public void volar() {//...}  
 @Override  
 public void comer() {//...}  
}  
  
class Pinguino implements Ave, AveNadadora{  
  
 @Override  
 public void nadar() {//...}  
 @Override  
 public void comer() {//...}  
}

Así, cada clase implementa las interfaces de la que realmente necesita implementar sus métodos. A la hora de añadir nuevas funcionalidades, esto ahorrará bastante tiempo.

**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.

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.

Se tiene una clase para realizar el acceso a datos, y lo hacemos a través de una base de datos:

class DatabaseService{  
  
 void getDatos(){}  
}  
  
class AccesoADatos {  
  
 private DatabaseService databaseService;  
   
 public AccesoADatos(DatabaseService databaseService){this.databaseService = databaseService;}  
   
 Dato getDatos(){databaseService.getDatos();}  
}

Si en el futuro queremos cambiar el servicio de base de datos por un servicio que conecta con una API, sería necesario modificar todas las instancias de la clase de la clase AccesoADatos.

Para arreglar esto, se puede hacer que el módulo AccesoADatos dependa de una abstracción más genérica:

interface Conexion {  
 Dato getDatos();  
 void setDatos();  
}  
  
class AccesoADatos {  
  
 private Conexion conexion;  
  
 public AccesoADatos(Conexion conexion){this.conexion = conexion;}  
 Dato getDatos(){conexion.getDatos();}  
}

Ahora, cada servicio que sea necesario pasar a AccesoADatos deberá implementar la interfaz Conexion:

class DatabaseService implements Conexion {  
  
 @Override  
 public Dato getDatos() {}  
  
 @Override  
 public void setDatos() {}  
}  
  
class APIService implements Conexion{  
  
 @Override  
 public Dato getDatos() {}  
  
 @Override  
 public void setDatos() {}  
}

Así, tanto el módulo de alto nivel como el de bajo nivel dependen de abstracciones, por lo que se cumple con el principio de inversión de dependencias.

# **Conclusiones**

Los principios S.O.L.I.D le dan a conocer al programador un forma distinta por la cual desarrollar sus programas, mostrándole que a través de la descentralización de responsabilidades en clases, expansión de módulos, aplicación de herencia de una manera controlada, creación de pequeñas interfaces que permiten realizar implementaciones flexibles de métodos, y la implementación de dependencias de interfaces en vez de objetos concretos, sus soluciones tendrán una mejor calidad y esto ayudara a reducir los costes a la hora de desarrollar nuevas implementaciones al programa o aplicación.

Actualmente, estas reglas se consideran buenas prácticas para un programador profesional, ya que con sus principios ayudan a tener un código: más limpio, mantenible y escalable, las cuales son características de un código de alta calidad, para un correcto código se necesitan correctos principios. Como indica el propio Robert C. Martin: *“No se trata de reglas, ni leyes, ni verdades absolutas, sino más bien de soluciones de sentido común a problemas comunes”.*

Finalmente, se puede decir que S.O.L.I.D es indispensables para la correcta programación, en la actualidad las tecnologías avanzan de forma exponencial lo cual no permite que los centros académicos estén al orden con las nuevas tecnologías, pero enseñando principios fundamentales como los mencionados anteriormente, ayudarán a la formación de un programador capaz de enfrentar cualquier problema y haciendo de este mismo una solución de alta calidad. Un programador con correctos principios puede desenvolverse mejor en un ambiente laboral que un programador sin principios.

# **Recomendaciones**

Es recomendable que los programadores tengan conocimiento de estos principios de programación y los pongan en práctica, ya que contribuirá a un mejor desarrollo como programador, además de tener una alta calidad en el código para una solución, esta implementación genera una correcta posición a la hora de estar en un ambiente laboral, desarrollando un código limpio, mantenible y escalable.

Antes de aplicar estos principios es importante tener un análisis previo de las funciones que se desean implementar e identificar de qué manera estará estructurada la aplicación y así lograr reconocer como y donde aplicar dichas reglas, además de identificar si en alguno de los casos se está violando uno de ellos.

# **Referencias**

C.M., M. (04 de 04 de 2019). *Principios SOLID con ejemplos*. Obtenido de https://enmilocalfunciona.io/principios-solid/

G., F. (05 de 09 de 2019). *Los 5 principios SOLID - Gerardo Fernández*. Obtenido de https://latteandcode.medium.com/los-5-principios-solid-68d697984abd

Marcos, N. A. (2018). *Principios SOLID en programación*. Obtenido de https://formiik.com/publicacion/principios-solid-en-programacion

Rebordinos, M. (24 de 08 de 2021). *¿Cuáles son los principios S.O.L.I.D. - "Single Responsability"*. Obtenido de https://tech.tribalyte.eu/blog-solid-single-responsability