**Clase 10 de Mayo**

Texto, Carta

Descripción generada automáticamente

Grupos de proyecto, tres personas

Interfaz de usuario gráfica, Diagrama

Descripción generada automáticamente

El MVC o Modelo-Vista-Controlador es un patrón de arquitectura de software que, utilizando 3 componentes (Vistas, Models y Controladores) separa la lógica de la aplicación de la lógica de la vista en una aplicación. Es una arquitectura importante puesto que se utiliza tanto en componentes gráficos básicos hasta sistemas empresariales; la mayoría de los frameworks modernos utilizan MVC (o alguna adaptación del MVC) para la arquitectura, entre ellos podemos mencionar a [Ruby on Rails](https://codigofacilito.com/cursos/rails), [Django](https://codigofacilito.com/cursos/django), [AngularJS](https://codigofacilito.com/cursos/angularjs) y muchos otros más. En este pequeño artículo intentamos introducirte a los conceptos del MVC.

Usamos Android Studio

**Principios SOLID**

Publicado por [Carlos Macías Martín](https://enmilocalfunciona.io/author/carlos-macias-martin/) el 03 April 2019

[Best Practices](https://enmilocalfunciona.io/tag/best-practices/)[Programacion](https://enmilocalfunciona.io/tag/programacion/)[SOLID](https://enmilocalfunciona.io/tag/solid/)

Todos sabemos que la POO (Programación Orientada a Objetos) nos permite agrupar entidades con funcionalidades parecidas o relacionadas entre sí, pero esto no implica que los programas no se vuelvan confusos o difíciles de mantener.

Un dibujo de un perro

Descripción generada automáticamente con confianza media

De hecho, muchos programas acaban volviéndose un monstruo al que se va alimentando según se añaden nuevas funcionalidades, se realiza mantenimiento, etc…

Viendo este problema, Robert C. Martin estableció cinco directrices o principios para facilitarnos a los desarrolladores la labor de crear programas legibles y mantenibles.

Estos principios se llamaron **S.O.L.I.D.** por sus siglas en inglés:

* **S: Single responsibility principle** o Principio de responsabilidad única
* **O: Open/closed principle** o Principio de abierto/cerrado
* **L: Liskov substitution principle** o Principio de sustitución de Liskov
* **I: Interface segregation principle** o Principio de segregación de la interfaz
* **D: Dependency inversion principle** o Principio de inversión de dependencia

Aplicar estos principios facilitará mucho el trabajo, tanto propio como ajeno (es muy probable que tu código lo acabe leyendo muchos otros desarrolladores a lo largo de su ciclo de vida). Algunas de las ventajas de aplicarlo son:

* Mantenimiento del código más fácil y rápido
* Permite añadir nuevas funcionalidades de forma más sencilla
* Favorece una mayor reusabilidad y calidad del código, así como la encapsulación

Vamos a ver en detalle cada uno de estos principios, junto a ejemplos básicos, que, a pesar de no ser aplicables en el mundo real, espero que aporten la suficiente claridad para que seas capaz de entender y aplicar estos principios en tus desarrollos.

**S: Principio de responsabilidad única**

Como su propio nombre indica, establece que una clase, componente o microservicio debe ser responsable de una sola cosa (el tan aclamado término “decoupled” en inglés). Si por el contrario, una clase tiene varias responsabilidades, esto implica que el cambio en una responsabilidad provocará la modificación en otra responsabilidad.

Considera este ejemplo:

class Coche {

String marca;

Coche(String marca){ this.marca = marca; }

String getMarcaCoche(){ return marca; }

void guardarCocheDB(Coche coche){ ... }

}

¿Por qué este código viola el principio de responsabilidad única? Para un minuto y piensa un poco ;)

Como podemos observar, la clase Coche permite tanto el acceso a las propiedades de la clase como a realizar operaciones sobre la BBDD, por lo que la clase ya tiene más de una responsabilidad.

Supongamos que debemos realizar cambios en los métodos que realizan las operaciones a la BBDD. En este caso, además de estos cambios, probablemente tendríamos que tocar los nombres o tipos de las propiedades, métodos, etc, cosa que no parece muy eficiente porque solo estamos modificando cosas que tienen que ver con la BBDD, ¿verdad?

Para evitar esto, debemos separar las responsabilidades de la clase, por lo que podemos crear otra clase que se encargue de las operaciones a la BBDD:

class Coche {

String marca;

Coche(String marca){ this.marca = marca; }

String getMarcaCoche(){ return marca; }

}

class CocheDB{

void guardarCocheDB(Coche coche){ ... }

void eliminarCocheDB(Coche coche){ ... }

}

Nuestro programa será mucho más cohesivo y estará más encapsulado aplicando este principio.

**O: Principio abierto/cerrado**

Establece que las entidades software (clases, módulos y funciones) deberían estar abiertos para su extensión, pero cerrados para su modificación.

Si seguimos con la clase Coche:

class Coche {

String marca;

Coche(String marca){ this.marca = marca; }

String getMarcaCoche(){ return marca; }

}

Si quisiéramos iterar a través de una lista de coches e imprimir sus marcas por pantalla:

public static void main(String[] args) {

Coche[] arrayCoches = {

new Coche("Renault"),

new Coche("Audi")

};

imprimirPrecioMedioCoche(arrayCoches);

}

public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){

for (Coche coche : arrayCoches) {

if(coche.marca.equals("Renault")) System.out.println(18000);

if(coche.marca.equals("Audi")) System.out.println(25000);

}

}

Esto no cumpliría el principio abierto/cerrado, ya que si decidimos añadir un nuevo coche de otra marca:

Coche[] arrayCoches = {

new Coche("Renault"),

new Coche("Audi"),

new Coche("Mercedes")

};

También tendríamos que modificar el método que hemos creado anteriormente:

public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){

for (Coche coche : arrayCoches) {

if(coche.marca.equals("Renault")) System.out.println(18000);

if(coche.marca.equals("Audi")) System.out.println(25000);

if(coche.marca.equals("Mercedes")) System.out.println(27000);

}

}

Como podemos ver, para cada nuevo coche habría que añadir nueva lógica al método precioMedioCoche(). Esto es un ejemplo sencillo, pero imagina que tu aplicación crece y crece… ¿cuántas modificaciones tendríamos que hacer? Mejor evitarnos esta pérdida de tiempo y dolor de cabeza, ¿verdad?

Para que cumpla con este principio podríamos hacer lo siguiente:

abstract class *Coche* {

// ...

abstract int precioMedioCoche();

}

class Renault extends *Coche* {

@Override

int precioMedioCoche() { return 18000; }

}

class Audi extends *Coche* {

@Override

int precioMedioCoche() { return 25000; }

}

class Mercedes extends *Coche* {

@Override

int precioMedioCoche() { return 27000; }

}

public static void main(String[] args) {

Coche[] arrayCoches = {

new Renault(),

new Audi(),

new Mercedes()

};

imprimirPrecioMedioCoche(arrayCoches);

}

public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){

for (Coche coche : arrayCoches) {

System.out.println(coche.precioMedioCoche());

}

}

Cada coche extiende la clase abstracta Coche e implementa el método abstracto precioMedioCoche().

Así, cada coche tiene su propia implementación del método precioMedioCoche(), por lo que el método imprimirPrecioMedioCoche() itera el array de coches y solo llama al método precioMedioCoche().

Ahora, si añadimos un nuevo coche, precioMedioCoche() no tendrá que ser modificado. Solo tendremos que añadir el nuevo coche al array, cumpliendo así el principio abierto/cerrado.

**L: Principio de substitución de Liskov**

Declara que una subclase debe ser sustituible por su superclase, y si al hacer esto, el programa falla, estaremos violando este principio.

Cumpliendo con este principio se confirmará que nuestro programa tiene una jerarquía de clases fácil de entender y un código reutilizable.

Veamos un ejemplo:

// ...

public static void imprimirNumAsientos(Coche[] arrayCoches){

for (Coche coche : arrayCoches) {

if(coche instanceof Renault)

System.out.println(numAsientosRenault(coche));

if(coche instanceof Audi)

System.out.println(numAsientosAudi(coche));

if(coche instanceof Mercedes)

System.out.println(numAsientosMercedes(coche));

}

}

imprimirNumAsientos(arrayCoches);

Esto viola tanto el principio de substitución de Liskov como el de abierto/cerrado. El programa debe conocer cada tipo de Coche y llamar a su método numAsientos() asociado.

Así, si añadimos un nuevo coche, el método debe modificarse para aceptarlo.

// ...

Coche[] arrayCoches = {

new Renault(),

new Audi(),

new Mercedes(),

new Ford()

};

public static void imprimirNumAsientos(Coche[] arrayCoches){

for (Coche coche : arrayCoches) {

if(coche instanceof Renault)

System.out.println(numAsientosRenault(coche));

if(coche instanceof Audi)

System.out.println(numAsientosAudi(coche));

if(coche instanceof Mercedes)

System.out.println(numAsientosMercedes(coche));

if(coche instanceof Ford)

System.out.println(numAsientosFord(coche));

}

}

imprimirNumAsientos(arrayCoches);

Para que este método cumpla con el principio, seguiremos estos principios:

* Si la superclase (Coche) tiene un método que acepta un parámetro del tipo de la superclase (Coche), entonces su subclase (Renault) debería aceptar como argumento un tipo de la superclase (Coche) o un tipo de la subclase (Renault).
* Si la superclase devuelve un tipo de ella misma (Coche), entonces su subclase (Renault) debería devolver un tipo de la superclase (Coche) o un tipo de la subclase (Renault).

Si volvemos a implementar el método anterior:

public static void imprimirNumAsientos(Coche[] arrayCoches){

for (Coche coche : arrayCoches) {

System.out.println(coche.numAsientos());

}

}

imprimirNumAsientos(arrayCoches);

Ahora al método no le importa el tipo de la clase, simplemente llama al método numAsientos() de la superclase. Solo sabe que el parámetro es de tipo coche, ya sea Coche o alguna de las subclases.

Para esto, ahora la clase Coche debe definir el nuevo método:

abstract class Coche {

// ...

abstract int numAsientos();

}

Y las subclases deben implementar dicho método:

class Renault extends *Coche* {

// ...

@Override

int numAsientos() {

return 5;

}

}

// ...

Como podemos ver, ahora el método imprimirNumAsientos() no necesita saber con qué tipo de coche va a realizar su lógica, simplemente llama al método numAsientos() del tipo Coche, ya que por contrato, una subclase de Coche debe implementar dicho método.

**I: Principio de segregación de interfaz**

Este principio establece que los clientes no deberían verse forzados a depender de interfaces que no usan.

Dicho de otra manera, cuando un cliente depende de una clase que implementa una interfaz cuya funcionalidad este cliente no usa, pero que otros clientes sí usan, este cliente estará siendo afectado por los cambios que fuercen otros clientes en dicha interfaz.

Imaginemos que queremos definir las clases necesarias para albergar algunos tipos de aves. Por ejemplo, tendríamos loros, tucanes y halcones:

interface *IAve* {

void volar();

void comer();

}

class Loro implements *IAve*{

@Override

public void volar() {

//...

}

@Override

public void comer() {

//..

}

}

class Tucan implements *IAve*{

@Override

public void volar() {

//...

}

@Override

public void comer() {

//..

}

}

Hasta aquí todo bien. Pero ahora imaginemos que queremos añadir a los pingüinos. Estos son aves, pero además tienen la habilidad de nadar. Podríamos hacer esto:

interface *IAve* {

void volar();

void comer();

void nadar();

}

class Loro implements *IAve*{

@Override

public void volar() {

//...

}

@Override

public void comer() {

//...

}

@Override

public void nadar() {

//...

}

}

class Pinguino implements *IAve*{

@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 tendríamos que añadir una excepción o aviso si se intenta llamar a estos métodos. Además, si quisiéramos añadir otro método a la interfaz IAve, 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 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 podríamos hacer lo siguiente:

interface *IAve* {

void comer();

}

interface *IAveVoladora* {

void volar();

}

interface *IAveNadadora* {

void nadar();

}

class Loro implements IAve, *IAveVoladora*{

@Override

public void volar() {

//...

}

@Override

public void comer() {

//...

}

}

class Pinguino implements IAve, *IAveNadadora*{

@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 nos ahorrará bastante tiempo, y además, cumplimos con el primer principio (Responsabilidad Única).

**D: Principio de inversión de dependencias**

Establece que las dependencias deben estar en las abstracciones, no en las concreciones. Es decir:

* Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
* Las abstracciones no deberían depender de detalles. Los detalles deberían depender de abstracciones.

En algún momento nuestro programa o aplicación llegará a estar formado por muchos módulos. Cuando esto pase, es cuando debemos usar inyección de dependencias, lo que nos permitirá controlar las funcionalidades desde un sitio concreto en vez de tenerlas esparcidas por todo el programa. Además, este aislamiento nos permitirá realizar testing mucho más fácilmente.

Supongamos que tenemos una clase para realizar el acceso a datos, y lo hacemos a través de una BBDD:

class DatabaseService{

//...

void getDatos(){ //... }

}

class AccesoADatos {

private DatabaseService databaseService;

public AccesoADatos(DatabaseService databaseService){

this.databaseService = databaseService;

}

Dato getDatos(){

databaseService.getDatos();

//...

}

}

Imaginemos que en el futuro queremos cambiar el servicio de BBDD por un servicio que conecta con una API. Para un minuto a pensar qué habría que hacer... ¿Ves el problema? Tendríamos que ir modificando todas las instancias de la clase AccesoADatos, una por una.

Esto es debido a que nuestro módulo de alto nivel (AccesoADatos) depende de un módulo de más bajo nivel (DatabaseService), violando así el principio de inversión de dependencias. El módulo de alto nivel debería depender de abstracciones.

Para arreglar esto, podemos 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();

}

}

Así, sin importar el tipo de conexión que se le pase al módulo AccesoADatos, ni este ni sus instancias tendrán que cambiar, por lo que nos ahorraremos mucho trabajo.

Ahora, cada servicio que queramos 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 cumplimos el principio de inversión de dependencias. Además, esto nos forzará a cumplir el principio de Liskov, ya que los tipos derivados de Conexion (DatabaseService y APIService) son sustituibles por su abstracción (interfaz Conexion).

**Conclusión**

Aplicar estos cinco principios puede parecer algo tedioso, pero a la larga, mediante la práctica y echarles un vistazo de vez en cuando, se volverán parte de nuestra forma de programar.

Nuestro programa será más sencillo de mantener, pero no solo para nosotros, si no más aún para los desarrolladores que vengan después, ya que verán un programa con una estructura bien definida y clara.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente