Código limpio

Ángel González M.

Índice

Introducción

Según Robert C. Martin. Hay que seguir 3 principios básicos

- **Legibilidad de la Fuente**, o capacidad del código de explicar su propósito sin necesidad de recurrir a nada más que el código.
- **Organización de Elementos**, es decir, la estructuración correcta de los elementos de un sistema donde uno espera que estén.
- **Certeza de Funcionamiento**, que se obtiene a través de la implementación de diferentes pruebas que verifiquen nuestros sistemas. Robert C. Martin lo lleva un paso más allá asegurando que “los verdaderos ingenieros de software prueban su código y la mejor manera de garantizar esto es implementando TDD“.

Razones para escribir con código limpio

Ángel González M.

Limpieza

Nuestro código puede ser leída muchas veces por otro humanos (normalmente compañeros de trabajo). Los desarrolladores son como escritores de libros, los mejores son aquellos que escriben libros fáciles de leer, una historia irresistible. Utilizan capítulos, títulos y parágrafos para organizar de una manera mas sencilla sus pensamientos y guiar al lector a través del libro. Los desarrolladores trabajan de la misma manera, simplemente utilizan clases, métodos… etc.

Buenas prácticas

Estos últimos años, mejores pràcticas de software como las pruebas unitarias, TDD, BDD, CI, etc están teniendo cada vez mas adopción. Estas prácticas elevan la calidad del código y su facilidad de mantenimiento.

Mantenimiento

Si alguien te pregunta sobre la calidad de tu código, has de aportar una justificación racional. Si nunca has considerado la calidad de tu estilo de codigo de una manera metódica, hay una gran oportunidad de mejora. Aquellos que escriben un código limpio, siguiendo unos ciertos patrones y técnicas.

Mantenimiento

Escribir un código fácil de leer te permite optimizar el 90% del tiempo invertido en ello.

Por lo que, al final, escribiendo un código fácil de mantener, estas optimizando el 70% del tiempo y coste invertido en mantenerlo, en vez del 30% que cuesta escribirlo.

Fácil de testear

Escribiendo un código limpio, se aconseja automáticamente testearlo.

Se refiere al desarrollo guiado por pruebas, la cual es la mejor manera de mejorar la calidad de tu codigo, velocidad del equipo y reducir el número de defectos en el software.

Todos estos factores tienen un gran peso en el ROI (beneficio de la inversión) del software.

Simplicidad

Mantén tu código tan simple y fácil de leer como sea posible.

Manteniéndolo simple puedes producir codigo de gran calidad, solventar problemas más rápido y en general trabajar mejor en los grupos.

Consistencia

Imagina que vas a una tienda y no hay consistencia sobre los productos expuestos. Sería muy complicado encontrar aquello que buscas. El arreglo en el código es más o menos como en el supermercado.

Cuando tu código está debidamente utilizado, es más fácil leer y buscar aquello que estás buscando.

Mantener un debido orden en los nombres es extremadamente importante en futuras ediciones del código.

Ahorro de costes

Escribiendo un código limpio, consigues todas esas ventajas anteriormente descritas que enfocan hacia un ahorro en costes.

Con lo cual tus jefes/jefas deberían obligar a escribir con código limpio a toda la plantilla.

PRINCIPIOS A SEGUIR PARA ESCRIBIR CÓDIGO LIMPIO

Ángel González M.

Haz unidades de código pequeñas

Haz unidades de código pequeñas

Las funciones o clases que son muy largos se vuelven casi imposibles de entender. Por el contrario, si son cortos y, combinado con el punto anterior, les damos mucho significado, la lectura será directa.

- Pocas líneas por clases.
- Pocas líneas en cada método
- Pocos niveles de anidamiento (ej bucle for, if/else)

Usar nombres que revelen las intenciones

Usar nombres que revelen las intenciones

Poner un nombre en programación es un proceso que hay que tomarse seriamente y cualquier ayuda, punto de vista o idea es bienvenida para seleccionar el más adecuado.

Si una variable o función requiere un comentario, entonces ese nombre no revela su intención.

Error:

```
int d; // tiempo transcurrido en días
```

Correcto:

```
int tiempoTranscurridoEnDias;
```

Tampoco te pases con nombres muy largos

Error:

```
void EstablecerCentroCosteImputacionParaSolicitudesQueDebenSerImputadasContraCentroCosteDelSolicitante();
```

Ejemplo

Incorrecto

```
public List getThem() {
   List<int[]> list1 = new ArrayList();
   for (int[] x : theList)
     if (x[0] == 4)
       list1.add(x);
   return list1;
}
```

Correcto

```
public List getFlaggedCells() {
    List flaggedCells = new ArrayList();
    for (int[] cell : gameBoard)
      if (cell[STATUS_VALUE] == FLAGGED)
        flaggedCells.add(cell);
    return flaggedCells;
}
```

Mucho más correcto

```
class Cell{
        boolean flagged;
        boolean isFlagged(){ return flaged; }
}
public List getFlaggedCells() {
    List flaggedCells = new ArrayList();
    for (Cell cell : gameBoard)
      if (cell.isFlagged())
        flaggedCells.add(cell);
    return flaggedCells;
}
```

Ángel González M.

El uso de switch es sospechoso

Switch está hecho para realizar N cosas y recuerda que las funciones deberían hacer una sola cosa.

La solución:

- Intentar aplicar polimorfismo.
- Usar algún patrón de diseño como (Factory method)
- Lleva los switch al nivel de detalles más bajo y aíslalo lo máximo que puedas

Ejemplo

En el siguiente ejemplo a simple vista no lo parece, pero la función tiene varios problemas:

- Aunque ahora no es muy larga, a medida que vayan apareciendo nuevos tipos de pagos la función crecer
- La función realiza más de una sola cosa
- Hay más de una razón por la que podría ser modificada, por tanto viola el Single Responsability Principle (SRP)
- Necesitará ser modificada cada vez que se añada un nuevo tipo de pago, por tanto también viola el Open Closed Principle (OCP)
- Y el peor de todos: vas a repetir este mismo switch en diferentes funciones, y con cada nuevo switch el código será más rígido.

Con la corrección los posibles efectos dañinos han quedado reducidos al máximo.

- Será una función que crezca en función de los nuevos tipos de pagos que aparezcan. Este punto no mejora :(
- La función sólo realiza un cosa, crear Empleado según su tipo. Punto mejorado :)
- Sólo hay una razón por la que podría ser modificada, por tanto cumple el Single Responsability Principle (SRP). Punto mejorado :)
- Necesitará ser modificada cada vez que se añada un nuevo tipo de pago, por tanto también viola el Open Closed Principle (OCP). Volvemos a fallar :(
- El punto más importante: no repetiremos esta misma lógica en ninguna función más del código. Punto mejorado :)

Ejemplo

Incorrecto

```
public void calcularGolpe(Jugador atacante, Jugador atacado) {
        int golpe;
        switch(atacante.tipo) {
        case "gerrero":
                golpe=10 * atacante.experiencia * Math.random();
                break;
        case "mago":
                golpe=8 * atacante.experiencia * Math.random();
                break;
        case "ladron":
                golpe=4 * atacante.experiencia * Math.random();
                break;
        }
}
```

Correcto

```
abstract class Personaje{
        float experiencia = 1;
        abstract double calcularGolpe(Personaje atacado);
}
class Guerrero extends Personaje{
        double calcularGolpe(Personaje atacado) {
                return 10 * experiencia * Math.random();
        }
}
class Mago extends Personaje{
        double calcularGolpe(Personaje atacado) {
                return 8 * experiencia * Math.random();
        }
}
class Ladron extends Personaje{
        double calcularGolpe(Personaje atacado) {
                return 4 * experiencia * Math.random();
        }
}
```

Ángel González M.

Las funciones deben tener un número limitado de argumentos

Las funciones deben tener un número limitado de argumentos

- Los parámetros añaden una gran cantidad de carga conceptual que dificulta la lectura.
- Además dificultan el testing.
- Más de tres parámetros deberían estar muy justificados, y es mejor evitarlos.
- Una alternativa sería sustituir todos (o algunos de) esos parámetros por un objeto que modele la entrada.

Evita usar comentarios si es posible

Evita utilizar comentarios siempre que sea posible.

Un comentario es un síntoma de no haber conseguido escribir un código claro.

Siempre que te veas en la necesidad de escribir un comentario, intenta reescribir el código o nombrar las cosas de otra forma, de tal forma que el comentario se vuelva irrelevante.

Muchas veces modificamos el código pero nos olvidamos de hacer lo propio con los comentarios

Los comentarios de javadoc si son importantes.

Ejemplo

Incorrecto

```
// Esta variable define el color del coche

int c;
```

Correcto

```
int color;
```

Utiliza un formato único en tu código

Utiliza un formato único en tu código

Dará coherencia y estructura al código, y unas pautas a seguir por todos los desarrolladores.

- Densidad vertical: los conceptos que están relacionados deberían aparecer juntos verticalmente. Necesitarás reglas que decidan cuándo se debe dejar una línea en blanco y cuándo no.
- Ubicación de los componentes: normalmente los campos de una clase se suelen poner al principio de la misma. También hay que decidir dónde situar las clases internas o las interfaces, en el caso de que permitáis usarlas.
- El número de caracteres por línea: Normalmente los editores vienen marcados con una línea vertical que suele rondar los 120 caracteres. Es una buena regla a seguir.

Ejemplo

Incorrecto

```
public class MiClase{

	int atributo1 = 4;

	public static void miMetodo(int parametro1,
			int parametro2) {

		for(int indice = 0; indice < 40; indice++)
		{
			int numero_unico = 6;
		}

		for(int i=0;i<10;i++) {
			int Numero=6;
		}

	}
	public static void OtroMetodo(int param1, int parametro2)
	{
	//...
	}

	int atributo2 = 6;
}
```

correcto

```
public class MiClase {

	int atributo1 = 4;
	int atributo2 = 6;

	public static void miMetodo(int parametro1, int parametro2) {
		//
		for (int indice = 0; indice < 40; indice++) {
			int numeroUnico = 6;
		}

		for (int i = 0; i < 10; i++) {
			int numero = 6;
		}

	}
	public static void otroMetodo(int parametro1, int parametro2) {
		//...
	}
}
```

Ángel González M.

Abstrae (encapsula) tus datos

Ángel González M.

No uses siempre public. También deber usar private y protected.

No uses getters y setters indiscriminadamente

Nuestro código está hecho para ser leído.

Legibilidad del código

El código debe funcionar correctamente; un código sin funcionalidad es inútil, pero en el mundo del desarrollo profesional también debemos considerar de forma consciente la Legibilidad del Código.

El código tiene que poder leerse como si fuera un libro.

Ejemplo

Correcto:

```
producto = Producto.crearProducto("Nombre", 34);

conexion = GestorBD.abrirBD("Usuario", "Pass", "NombreBD");

conexion.insertar(producto);

conexion.cerrarBD();
```

STUPID

Ángel González M.

S - Singleton

El uso del patrón singleton:

- Crea instancias única
- Se pueden reutilizar en cualquier parte del programa sin instanciar los objetos.
- Interesante para hacer caches de memoria

El problema del uso de singleton es el **acoplamiento**.

Lo recomendable es huir del patrón singleton.

T- Tight coupling

Código fuertemente acoplado debes de evitarlo.

Por ejemplo, tu aplicación está construida para conectarse a mysql, o usas un proveedor de envío de emails determinado (ej mandril).

El días de mañana, cuando decidas cambiar la base de datos o el proveedor de email por otro… tu aplicación está tan acoplada que vas a tener muchos problemas. Tu código no es tolerante a cambios.

Solución: Aplica técnicas de programación o patrones de diseño para evitar el acoplamiento. Ej patrón de inversión de dependencias.

U- Untestability

Has programado un código que no puedes testear.

Siempre que programes código, este tiene que poder ser testeado.

Una buena herramienta para evitar la **no testeabilidad** es hacer TDD (hacer primero los test y luego escribir el código)

P- Premature optimización

Decidir implementar cosas que no son necesarias (ahora mismo) en tu aplicación.

Ej Tienes que hacer una web… y decides montar un servidor en clúster, o usar microservicios en kubernetes, pues en el futuro tu aplicación seguramente tendrá una alta carga de demanda de querys.

Solución: Cíñete a lo necesario para que la funcionalidad pedida por el cliente funcione, y con el tiempo puedes ir añadiendo complejidad.

I- Indescriptible naming

Usas nombre de métodos, atributos, clases que no dan información. Esto es un code smell.

Ejem a= a * c

Solución: Poner nombres adecuados

precio = precio * cantidad

D- Duplicación

Repetir código fuente en muchas partes del código.

Solución: Usar abstracción, herencias, polimorfismo …

NOTA: Más vale un código duplicado, que una mala abstracción

Veamos una serie de **PRINCIPIOS** del código limpio

DRY Principle (Don’t Repeat Yourself).

Ángel González M.

DRY Principle (Don’t Repeat Yourself).

No te repitas.

El **código duplicado:**

- hace que nuestro código sea más **difícil de mantener** y **comprender** además de generar posibles inconsistencias.
- Obliga a **testear** lo mismo **varias veces**
- Al **modificar** código, tenemos que recordar hacer los cambios en el código repetido.
- **Complica** el mantenimiento

Hay que evitar el código duplicado SIEMPRE.

Para ello, dependiendo del caso concreto, aplicaremos:

- la refactorización
- la abstracción
- o el uso de patrones de diseño

Ejemplo

Incorrecto

```
public static void main(String[] args) {
        int numeros[] = {1,4,6};

        // calcular el promedio
        float media = 0;
        for(int i=0;i< numeros.length;i++) {
                media = media + numeros[i];
        }
        media = media / numeros.length;
        System.out.println(media);

        numeros = new int[] {7,3,2};

        // calcular el promedio
        media = 0;
        for(int i=0;i< numeros.length;i++) {
                media = media + numeros[i];
        }
        media = media / numeros.length;
        System.out.println(media);

}
```

correcto

```
public static float promedio(int[] numeros) {
        float media = 0;
        for(int i=0;i< numeros.length;i++) {
                media = media + numeros[i];
        }
        return media / numeros.length;
}

public static void main(String[] args) {
        int numeros[] = {1,4,6};
        System.out.println(  promedio(numeros)  );

        numeros = new int[] {7,3,2};
        System.out.println(  promedio(numeros)  );
}
```

The Principle of Least Surprise

Ángel González M.

The Principle of Least Surprise

También conocido como The Principle of Least Astonishment, nos dice que: las funciones o clases deben hacer lo que (razonablemente) se espera de ellas.

Es decir, una función o una clase debe tener, en función de su nombre, un comportamiento obvio para cualquier programador, **sin** que este tenga la necesidad de **sumergirse en su código**.

Ejemplo

Day day = DayDate.StringToDay(String dayName);

Cualquier programador al ver esta función (método) esperará que, si le pasa la cadena de caracteres “Monday”, la respuesta sea Day.MONDAY. Incluso podríamos esperar que diese igual enviar la cadena con mayúsculas o minúsculas.

Si esta función no hiciese esto su comportamiento no sería obvio y no estaría cumpliendo con el principio.

Ejemplo

Incorrecto

```
public String concatena2Cadenas(String cadena1, String cadena2) {
        String resultado = cadena1.trim() + cadena2.trim();
        return resultado.toLowerCase();
}
```

correcto

```
public String concatena2Cadenas(String cadena1, String cadena2)
{
        return cadena1 + cadena2;
}

public String eliminaEspaciosExtremos(String cadena) {
        return cadena.trim();
}

public String convertirAMinusculas(String cadena) {
        return cadena.toLowerCase();
}
```

The Boy Scout Rule

Ángel González M.

The Boy Scout Rule

Deja las cosas mejor de como te las encontraste.

Si encuentra un código sucio, incluso si no lo has escrito tú, mejóralo.

Ejemplo

Incorrecto

```
public static void calcularMedia(int[] numeros) {
                float media = 0;
                for(int i=0;i< numeros.length; i++) {
                        media = media + numeros[i];
                }
                media = media / numeros.length;
                System.out.println(media);
}
```

correcto

```
        /**
         * Calcula el valor medio de un conjuntode números
         * @param numeros Array con los numeros a calcular
         * @return el valor medio de todos los números pasados como parámetro
         */
        public static float calcularMedia(int[] numeros) {
                float suma = 0;
                for(int n: numeros) {
                        suma += n;
                }
                return suma / numeros.length;
        }
```

F.I.R.S.T. (testing)

Ángel González M.

F.I.R.S.T. (testing)

Es crucial crear los test, pero los test también se escriben con código limpio. Deben cumplir una serie de reglas.

- **Fast**: los test deben correr rápido. Deben tardar poco en ejecutarse. De no ser así, es probable que nos de pereza ejecutarlos y, por tanto, que no lo hagamos con la frecuencia deseada.
- **Independent**: los test deben ser independientes unos de otros. El resultado de un test no debe condicionar el de los siguientes. Deben poder ejecutarse en el orden que se quiera. De lo contrario, un fallo en el primer test, puede desencadenar un fallo en cascada de los demás, haciendo complejo el diagnóstico del sistema.
- **Repeatable**: los test deben poder ejecutarse en cualquier entorno (desarrollo, pre-producción, producción…). De no ser así, siempre tendremos una excusa para cuando los test fallen.
- **Self-Validating**: los test deben devolver una respuesta booleana. Pasan o no pasan. No deben dejar una cadena de caracteres en un fichero de log que tengamos que comprobar nosotros mismos, o dejar dos ficheros de un tamaño determinado que, igualmente tengamos que comprobar. De lo contrario, requerirán una alta evaluación manual que nos hará perder tiempo y precisión.
- **Timely**: los test deben ser escritos antes que el código de producción. De no ser así, el código de producción será difícil de testear.

Principios SOLID

Ángel González M.

S- Single Responsibility Principle (SRP)

Ángel González M.

Introducción

El principio de Responsabilidad Única (la S de los principios SOLID), hace referencia al diseño de nuestras clases. Dice que una clase debe tener una única responsabilidad.

- Esta clase será muy propensa a cambios: Aumentamos el número de razones por el que necesitaremos modificar esa clase, y por ello es más fácil que introduzcamos errores y que se vuelva excesivamente compleja.
- Nos cuesta más testearla: al no tener bien definido lo que hace, los tests serán muy largos y seguramente poco eficaces. Además nos tocará cambiarlos muy a menudo cuando esa clase se modifique.
- El código crecerá de forma descontrolada, y nos será muy difícil añadir nueva funcionalidad.

Qué es

Una clase = Un concepto y una responsabilidad

Una clase debería tener una única razón por la que cambiar.

Ejemplo

Incorrecto

```
class Vehiculo{
        String getMarca() { … }
        String getMatricula() { … }
        Date[] getFechasMultas(){ … }
        float[] getImportesMulta(){ … }
}
```

Correcto

```
class Vehiculo{
        String getMarca() { … }
        String getMatricula() { … }

        Multa[] multas;

}

class Multa{
        Date getFecha() { … };
        float getImporte(){ … };
        float getTipo(){ ... };
}
```

Open Closed Principle.(OCP)

Ángel González M.

Open Closed Principle.

Dice que: “una clase debe estar abierta a extensiones pero cerrada a modificaciones”. O lo que es lo mismo, el comportamiento de dicha clase debe ser alterado sin tener que modificar su código fuente. De lo contrario podría desencadenar efectos colaterales.

Si cambiasen los requisitos, el comportamiento de la clase debe ser extendido, no modificado. La inyección de dependencias también nos puede ayudar en esta tarea.

Podemos solucionar este problema usando polimorfismo.

Nos dice que el código está mejor diseñado si se puede modificar su comportamiento sin cambiar su código fuente

Si necesitáramos añadir una nueva operación (ej. update) tendríamos que modificar la clase

Ejemplo

Incorrecto

```
public class Sql {
        public Sql (String table, Column[] columns) {...}
        public String insert (Object[] fields) {...}
        public String findByKey (String keyColumn, String keyValue) {...}
        public String select (Criteria criteria) {...}}
}
```

Ahora si necesitáramos añadir una nueva operación (ej. update) solo tenemos que crear una nueva clase que herede de Sql

Correcto

```
abstract class Sql {
        public Sql (String table, Column[] columns) {...}
        public abstract String generate();
}

public class InsertSql extends Sql {
        public InsertSql (String table, Column[] columns, Object[] fields) {...}
        public String generate () {...}
}

public class FindByKeySql extends Sql {
        public FindByKeySql (String table, Column[] columns, String keyColumn, String keyValue) {...}
        public String generate () {...}
}
```

Ángel González M.

El principio de sustitución de Liskov LSP

Ángel González M.

Este principio dice que una clase derivada no debe modificar el comportamiento de la clase base.

En líneas generales el LSP nos viene a decir que, si parece un pato y grazna como un pato pero usa pilas, probablemente mi abstracción no sea la correcta.

Ejemplo

Incorrecto

```
abstract class Ave{
        abstract void volar();
}

class Aguila extends Ave{
        @Override
        void volar() {
                ...
        }
}
class Pinguino extends Ave{
        @Override
        void volar() {
                System.out.println("No puedo volar");
        }
}
```

correcto

```
abstract class Ave{

}

interface Voladora{
        void volar();
}

class Aguila extends Ave implements Voladora{
        @Override
        public void volar() {

        }
}

class Pinguino extends Ave{

}
```

correcto

```
abstract class Ave{
        ...
}

abstract class AveVoladora extends Ave{
        abstract void volar();
}

class Aguila extends AveVoladora{
        @Override
        void volar() {
                ...
        }
}

class Pinguino extends Ave{
        ...
}
```

Otro ejemplo

```
class Rectangle {

        private int width;
        private int height;

        public int getWidth() {
                return width;
        }

        public void setWidth(int width) {
                this.width = width;
        }

        public int getHeight() {
                return height;
        }

        public void setHeight(int height) {
                this.height = height;
        }

        public int calculateArea() {
                return width * height;
        }
}
```

```
class Square extends Rectangle {

        @Override
        public void setWidth(int width) {
                super.setWidth(width);
                super.setHeight(width);
        }

        @Override
        public void setHeight(int height) {
                super.setHeight(height);
                super.setWidth(height);
        }
}
```

```
@Test
public void testAreaRectangulo() {
        Rectangle r = new Rectangle();
        r.setWidth(5);
        r.setHeight(4);
        assertEquals(20, r.calculateArea());
}
@Test
public void testAreaCuadrado() {
        Square r = new Square();
        r.setWidth(5);
        r.setHeight(4);
        assertEquals(20, r.calculateArea());
}
```

El principio de segregación de interfaces

Ángel González M.

Este principio, conocido por ISP (Interface segregation principle), dice que el una clase que implementa una interfaz no debe depender de métodos que no utiliza. Esto como norma general implica que nuestras interfaces deben ser sencillas y tener pocos métodos. Es preferible tener varias interfaces pequeñas, a tener una interfaz grande. Así no obligamos a los clientes a depender de métodos que no necesitan implementar.

Ejemplo

Incorrecto

```
interface Ave{
        void comer();
        void nadar();
        void volar();
        void correr();
}

class Avestruz implements Ave{
        @Override public void comer() {}
        @Override public void nadar() {}
        @Override public void volar() {}
        @Override public void correr() {}
}

class Loro implements Ave{
        @Override public void comer() {}
        @Override public void nadar() {}
        @Override public void volar() {}
        @Override public void correr() {}
}
class Pinguino implements Ave{
        @Override public void comer() {}
        @Override public void nadar() {}
        @Override public void volar() {}
        @Override public void correr() {}
}
```

correcto

```
abstract class Ave{
        abstract void comer();
}
interface iCorredor{
        void correr();
}

interface iNadador{
        void nadar();
}
interface iVolador{
        void volar();
}

class Avestruz extends Ave implements iCorredor{
        @Override public void comer() {}
        @Override public void correr() {}
}

class Gaviota extends Ave implements iVolador, iNadador{
        @Override public void comer() {}
        @Override public void nadar() {}
        @Override public void volar() {}
}
class Pinguino extends Ave implements iNadador{
        @Override public void comer() {}
        @Override public void nadar() {}
}
```

Ángel González M.

El principio de inversión de dependencias

Ángel González M.

En inglés DIP (Dependency Inversion Principle), que viene a decir que las clases de alto nivel, no deben depender de clases de bajo nivel. Ambos deben depender de abstracciones.

Además las abstracciones no deben depender de los detalles, si no que los detalles deben depender de las abstracciones.

En definitiva, lo que tenemos que hacer es invertir las dependencias.

Además si lo hacemos bién, será mucho más fácil testear nuestras clases.

Ejemplo 1

Incorrecto

```
class LimpiarPantalla{
        public void limpiar() throws IOException  {
                String tipoSistemaOperativo = System.getProperty("os.name");
                if(tipoSistemaOperativo.equals("Mac Os X")
                || tipoSistemaOperativo.equals("Linux") ) {
                        Runtime.getRuntime().exec("clear");
                }else if(tipoSistemaOperativo.equals("Windows")) {
                        Runtime.getRuntime().exec("cls");
                }
        }
}
```

correcto

```
interface iLimpiable {
        void limpiar() throws IOException;
}

class WindowsSSOO implements iLimpiable{
        @Override
        public void limpiar() throws IOException{
                Runtime.getRuntime().exec("cls");
        }
}

class LinuxSSOO implements iLimpiable{
        @Override
        public void limpiar() throws IOException{
                Runtime.getRuntime().exec("clear");
        }
}

class LimpiarPantalla{
        public void limpiar(iLimpiable sistemaOperativo)  throws IOException {
                sistemaOperativo.limpiar();
        }
}
```

Ejemplo

Incorrecto

```
public class BasicEmployeeSalaryCalculator {
        public float getSalary (Employee employee) {
                // calcula el salario del empleado
        }
}

public class Employee {
        public float calculateSalary (BasicEmployeeSalaryCalculator
employeeSalaryCalculator) {
                return employeeSalaryCalculator.getSalary(this);
        }
}
```

Vemos que la clase Employee está fuertemente acoplada a la implementación BasicEmployeeSalaryCalculator. Ahora bien, ¿qué pasaría si hubiese otra forma de calcular el salario de un empleado? Por ejemplo, en función del mes (imaginemos que tiene paga extra).

correcto

```
public interface EmployeeSalaryCalculator {
        public float getSalary (Employee employee);
}

public class BasicEmployeeSalaryCalculator implements EmployeeSalaryCalculator {
        public float getSalary (Employee employee) {
                // calcula el salario del empleado
        }
}

public class ExtraPayEmployeeSalaryCalculator implements EmployeeSalaryCalculator {
        public float getSalary (Employee employee) {
                // calcula el salario del empleado en función de su paga extra
        }
}

public class Employee {
        public float calculateSalary (EmployeeSalaryCalculator employeeSalaryCalculator) {
                return employeeSalaryCalculator.getSalary(this);
        }
}
```

Vemos que ahora, la clase Employee está desacoplada del detalle concreto (BasicEmployeeSalaryCalculator) y, por el contrario se vincula a la abstracción (la interface). Por tanto, ahora bastará con llamar al método calculateSalary de la clase Employee pasándole la calculadora concreta que necesitemos en cada momento (incluso pueden surgir nuevas calculadoras de salario).

Ejemplo 2

Incorrecto

```
class Compra{
        // ...
}
 class CarritoCompra {

    public void buy(Compra compra) {

        SqlDatabase db = new SqlDatabase();
        db.insert(compra);

        TarjetaCredito creditCard = new TarjetaCredito();
        creditCard.pagar(compra);
    }
}

 class SqlDatabase {
    public void insert(Compra compra){
        // inserta los datos en sql
    }
}

 class TarjetaCredito {
    public void pagar(Compra compra){
        // realiza el pago usando la tarjeta de credito
    }
}
```

correcto

```
class Compra{
            //...
}
 interface Persistence {
    void insert(Compra shopping);
}

 class SqlDatabase implements Persistence {

    @Override
    public void insert(Compra shopping){
        // inserta los datos en sql
    }
}

 interface MetodoDePago {
    void pagar(Compra shopping);
}

 class TarjetaCredito implements MetodoDePago {

    @Override
    public void pagar(Compra shopping){
        // realiza el pago usando la tarjeta de credito
    }
}
```

```
class CarritoCompra {

    private final Persistence persistence;
    private final MetodoDePago paymentMethod;

    public CarritoCompra(Persistence persistence,
            MetodoDePago paymentMethod) {
        this.persistence = persistence;
        this.paymentMethod = paymentMethod;
    }

    public void comprar(Compra compra) {
        persistence.insert(compra);
        paymentMethod.pagar(compra);
    }
}


class Mysql implements Persistence {

            @Override
            public void insert(Compra compra) {
                // inserta los datos en BD mysql
            }
}

class Paypal implements MetodoDePago {

            @Override
            public void pagar(Compra compra) {
                // Realiza el pago usando paypal
            }
}
```

Ángel González M.

Ley de Demeter

Ángel González M.

La Ley de Demeter es un mecanismo de detección de acoplamiento, y nos viene a decir que nuestro objeto no debería conocer las entrañas de otros objetos con los que interactúa. Si queremos que haga algo, debemos pedírselo directamente en vez de navegar por su estructura.

Según este principio, una unidad solo debe tener conocimiento limitado de otras unidades, y solo conocer aquellas que están relacionadas. La una unidad solo debe hablar con amigos y nunca con extraños. Además la unidad solo debe hablar con amigos inmediatos.

Simplificando mucho, tenemos que tratar de evitar utilizar métodos de un objeto que ha sido devuelto por otro método. En este caso es útil seguir la regla de nunca usar más de un punto cuando accedemos a métodos de un objeto.

Por ejemplo no usar clienteActual.ObtenerDireccion.calle.CambiarNombreDeCalle.

El principio de Hollywood

Ángel González M.

Basado en la típica respuesta que se les da a los actores que hacen una prueba para una película: "No nos llame, nosotros le llamaremos". Este principio está relacionado con el principio de inversión de dependencias de SOLID.

Un ejemplo del principio de Hollywood es la inversión de control (IoC), que hace que una clase obtenga las referencias a objetos que necesita para funcionar, a través de una entidad externa.

Lanza excepciones en lugar de devolver códigos de retorno

Ángel González M.

- Por un lado, no tienes que recordar operar con ese código de error y decidir qué camino tomar, pues las excepciones lo harán por ti.
- Por otro lado, se separa fácilmente la lógica del "camino feliz" de la de errores, ya que los errores serán manejados en sus catch correspondientes

Lo idóneo es crearse excepciones propias que den un significado más semántico al error de un solo vistazo, y proveerlas de mensajes de error claros y descriptivos.

```
public static float dividirSucio(float a, float b) {
        if(b==0) {
                System.out.println("ERROR AL DIVIDIR POR CERO");
                return 0;
        }else {
                return a / b;
        }
}
```

```
public static float dividirLimpio(float a, float b) throws
DivisionPorCeroException{
```

Ángel González M.

Establece fronteras

Existen muchos casos en nuestros software en los que no tenemos control sobre el código que ejecutamos, ya sea por ejemplo cuando utilizamos librerías de terceros o frameworks.

Las fronteras nos permiten establecer límites entre nuestro código y ese código que no controlamos, de tal forma que podamos acotar la interacción con él, y que a su vez nos permita sustituirlo sin problemas en caso de necesidad.

Las fronteras también nos pueden ayudar en los casos en los que tenemos que trabajar con código que aún no existe. Podemos declarar una serie de interfaces que inicialmente implementaremos con datos falsos que nos sirvan de sustituto hasta que el código definitivo esté listo.

Ejemplo correcto

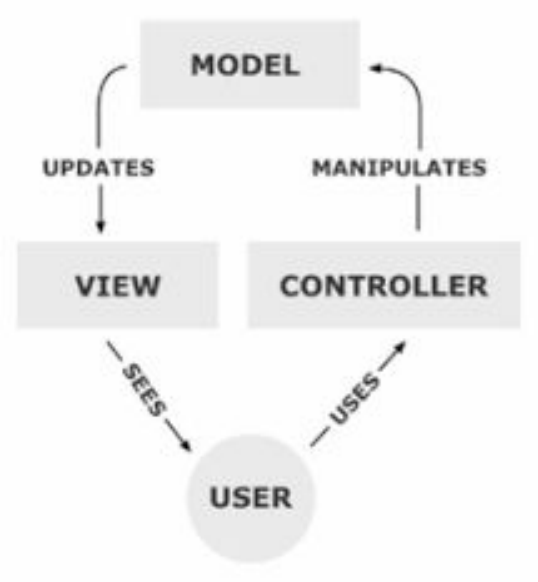
```
interface iRobot{
	void moverPieDer();
	void moverPieIzq();
	boolean detectarObstaculo();
}

class C3PO implements iRobot{

	@Override
	public void moverPieDer() { }

	@Override
	public void moverPieIzq() {  }

	@Override
	public boolean detectarObstaculo() {   return false; }

}


public class Manejo {
	public void moverRobot( iRobot robot ) {
		robot.moverPieDer();
		robot.moverPieIzq();
		robot.detectarObstaculo();
	}
}
```

Ángel González M.

Separa los asuntos

Ángel González M.

Principio SoC (separation of concerns). Los asuntos, son los diferentes aspectos de la funcionalidad de nuestra aplicación. Por ejemplo la capa de negocio, la capa de datos etc. Un buen ejemplo de separación es el patrón MVC.



Principio KISS

Ángel González M.

KISS significa “Keep It Simple, Stupid” (Mantenlo simple, estúpido) . Es uno de los principios más antiguos de diseño de software, aunque lo olvidamos a menudo.

Los sistemas más eficaces son los que mantienen la simplicidad, evitando la complejidad innecesaria. El objetivo es que el diseño del software sea lo más simple posible.

Los desarrolladores somos humanos, por lo que tenemos capacidades limitadas. Tanto a la hora de escribir código como de depurarlo, aún más cuando nos enfrentamos a código complejo sin motivo.

Un diseño de software simple es el que se centra en los requisitos actuales, pero no olvida necesidad futuras como:

- la mantenibilidad
- extensibilidad
- o la reutilización

Ejemplo

Incorrecto

```
public static String getNombreSemana(int day) {
        switch (day) {
            case 1:
                return "Monday";
            case 2:
                return "Tuesday";
            case 3:
                return "Wednesday";
            case 4:
                return "Thursday";
            case 5:
                return "Friday";
            case 6:
                return "Saturday";
            case 7:
                return "Sunday";
            default:
                System.out.println("Dias válidos de 1 a 7");
                return "";
        }
    }
```

correcto

```
public String getNombreSemana(int day) {
    if ((day < 1) || (day > 7))
        throw new InvalidParameterException("ERR: Dias válidos de 1 a 7");
    String[] days = {
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday"
    };
    return days[day - 1];
}
```

El principio YAGNI

Ángel González M.

Muchas veces, para evitar problemas posteriores, los programadores tendemos a desarrollar funcionalidades que no estamos seguros de necesitar. Simplemente lo hacemos "por si acaso". El principio YAGNI (You ain't gonna need it) (No vas a necesitarlo), viene a decir que no debemos implementar algo si no estamos seguros de necesitarlo. Así ahorramos tiempo y esfuerzo.

"Aplicar siempre las cosas cuando realmente los necesita, no cuando lo que prevén que los necesita." - Ron Jeffries

Ej ERROR: en la base de datos vamos a necesitar hacer selects e inserts, pero vamos a implementar las actualizaciones y borrados para el futuro aunque no las necesitemos ahora.

Navaja de Occam

Ángel González M.

La Navaja de Occam se remonta a un principio metodológico y filosófico, perfectamente aplicable al desarrollo de software, según el cual, “en igualdad de condiciones, la explicación más sencilla suele ser la correcta”.

Este principio lo podemos usar tanto en el momento de implementar una solución como a la hora de encontrar el causante a un bug. ¿Cuántas veces nos habrá pasado que la metedura de pata más tonta es la causante del problema aunque hayamos estado comprobando lo más complejo?

MÁS EJERCICIOS

Libro: Clean code Robert C. Martin
https://drive.google.com/file/d/0B_rRtttpELcPSVZsVmtBTGQ4eEE/view

https://www.genbeta.com/desarrollo/doce-principios-de-diseno-que-todo-desarrollador-deberia-conocer