Código Limpio

Created	@March 15, 2022 9:51 AM
Tags	2doTrimestre
Formato	
Materia	
Tema	

Introducción

3 principios básicos

- Legibilidad de la Fuente.
- Organización de Elementos
- Certeza de Funcionamiento

Razones para escribir con código limpio

- Limpieza
- Buenas prácticas
- Mantenimiento
- Fácil de testear
- Simplicidad
- Consistencia
- Ahorro de costes

PRINCIPIOS A SEGUIR PARA ESCRIBIR CÓDIGO LIMPIO

Haz unidades de código pequeñas

- Pocas líneas por clases.
- Pocas líneas en cada método

- Pocos niveles de anidamiento (ej: bucle for, if/else)

Usar nombres que revelen las intenciones

```
//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 mas 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;
}
```

El uso de switch es sospechoso

```
//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();
  }
}
```

Las funciones deben tener un número limitado de argumentos

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 utilizar comentarios siempre que sea posible.

```
//Incorrecto

// Esta variable define el color del coche
int c;
```

```
//Correcto

int color;
```

Utiliza un formato único en tu código

Abstrae (encapsula) tus datos

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

No uses getters y setters indiscriminadamente.

Legibilidad del código

```
producto = Producto.crearProducto("Nombre", 34);
conexion = GestorBD.abrirBD("Usuario", "Pass", "NombreBD");
conexion.insertar(producto);
conexion.cerrarBD();
```

STUPID

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.

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.

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.

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

DRY Principle (Don’t Repeat Yourself).

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

```
//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

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.

```
//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

Deja las cosas mejor de como te las encontraste.
Si encuentra un código sucio, incluso si no lo has escrito tú, mejóralo.

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.
- **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.
- **Repeatable**: Los test deben poder ejecutarse en cualquier entorno (desarrollo, pre-producción, producción…).
- **Self-Validating**: Los test deben devolver una respuesta booleana. Pasan o no pasan.
- **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

S- Single Responsibility Principle (SRP)

SOLID

Una clase debe tener una única responsabilidad.

```
//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)

SOLID

Una clase debe estar abierta a extensiones pero cerrada a modificaciones.

```
//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) {...}}
}
```

```
//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 () {...}
}
```

El principio de sustitución de Liskov (LSP)

SOLID

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

```
//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{
  ...
}
```

El principio de segregación de interfaces

SOLID

Una clase que implementa una interfaz no debe depender de métodos que no utiliza.

```
//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() {}
}
```

El principio de inversión de dependencias, DIP (Dependency Inversion Principle)

Principio SOLI**D**

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.

```
//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();
  }
}
```

```
//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);
  }
}
```

```
//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);
  }
}
```

```
//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
      }
}
```

Ley de Demeter

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.

El principio de Hollywood

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.

Lanza excepciones en lugar de devolver códigos de retorno

- 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.

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.

Separa los asuntos

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

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

El principio YAGNI

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.

Navaja de Occam

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.