



Linux
Málaga

@linux_malaga

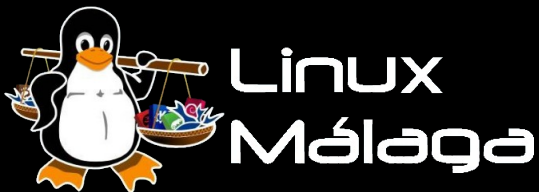
www.linux-malaga.org

INTRODUCCIÓN A LOS PRINCIPIOS DE DISEÑO Y PATRONES DE DESARROLLO

¿QUIEN SOY?

Juan Pablo Guerrero Durán (Bart) – 1977
@bartola_

- I.T. I de Gestión por la UMA.
- Desde 2003 hasta 2014 dedicado a la programación, implantación y diseño de aplicaciones informáticas para el sector sanitario.
- Actualmente trabajo en Centrologic como desarrollador PHP.
<http://www.centrologic.com>
@centrologic_es
- Desarrollando un proyecto personal bajo Python/Django.

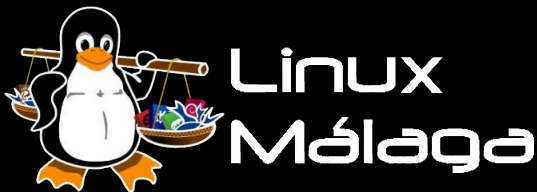


¿QUE ES EL DISEÑO?

Proceso de aplicar distintas técnicas y principios con el propósito de definir un dispositivo, proceso o sistema con los suficientes detalles como para permitir su realización física.

Es necesario para conseguir un software bien acabado.

Un error en el proceso de diseño puede arruinar un proyecto y llevarlo a una dinámica de cambios constantes



¿DE QUE SE COMPONE UN DISEÑO?

- Creatividad
- Intuición
- Experiencia
- Guías
- Métodos
- Heurísticas
- Criterios de calidad
- Proceso iterativo

ALGUNAS REGLAS QUE DEBE SEGUIR UN BUEN DISEÑO

- Reusabilidad
- Extensibilidad
- Funcionalidad
- Orden
- Trazabilidad
- Seguro

Para ayudarnos a conseguirlo, además del sentido común podemos apoyarnos en una serie de principios que otros programadores ya enunciaron antes que nosotros.

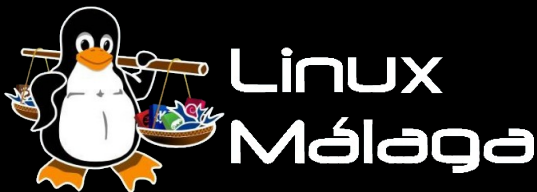
PRINCIPIO DRY

DON'T REPEAT YOURSELF

Uno de los mas simples y a la vez mas importantes.

No se debe escribir código duplicado.

El código repetido debería extraerse a una función para encapsularlo.



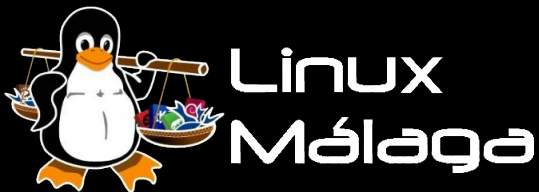
PRINCIPIO KISS

Keep it simple stupid!

Otras variantes, Keep is short and simple o keep it small and simple.

La simplicidad debe ser un objetivo clave en el diseño.

Cuanto mas simple es un sistema, mas eficaz es.

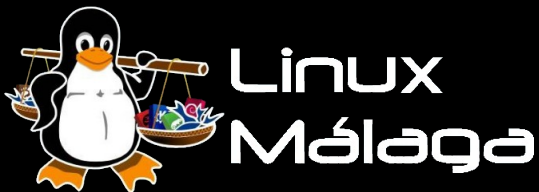


PRINCIPIO YAGNI

You aren't gonna need it.

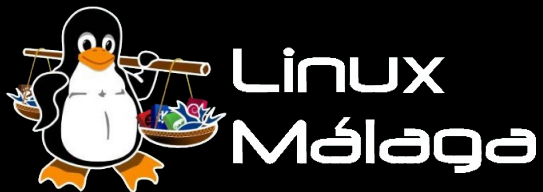
No lo vas a necesitar.

No se deben agregar funcionalidades extras hasta que no sea necesario.



LA REGLA DEL BOY SCOUT

Un Boy Scout siempre deja el campo mas limpio que cuando llegó.



Principios SOLID

Conjunto de principios básicos orientado a conseguir un buen software.

Los principios que incluye son los siguientes:

- Single responsibility.
- Open-closed
- Liskov substitution
- Interface segregation principle
- Dependency inversion.

Single Responsibility Principle (SRP)

"There should never be more than one reason for a class to change."

Una clase o módulo debe tener una única responsabilidad.

Debe ser la única clase o módulo con dicha responsabilidad



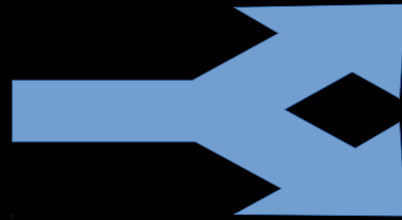
Single Responsibility Principle (SRP)

```
Class Empleado {  
  
    RepostarGasolina();  
    CobrarCliente();  
  
}
```

Single Responsibility Principle (SRP)

Class Empleado {

RepostarGasolina();
CobrarCliente();
}



Class Cajero {
CobrarCliente();
}

Class Repostador {
RepostarGasolina();
}

Open-Closed principle (OCP)

"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

Un código está mejor diseñado si puede modificarse su comportamiento sin necesidad de cambiar su código fuente ya existente, simplemente añadiendo código en vez de modificaciones.

La manera mas usual de seguir este principio es usar interfaces o clases abstractas

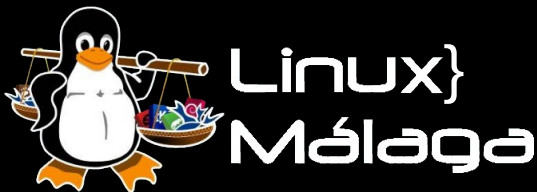


Open-Closed principle (OCP)

```
public interface IForma {  
    int GetTipo();  
}
```

```
public class Rectangulo implements  
IForma {  
    public int GetTipo() {  
        return 1;  
    }  
}
```

```
public class Circulo implements  
IForma {  
    public int GetTipo() {  
        return 2;  
    }  
}
```



Open-Closed principle (OCP)

```
public class EditorGrafico {  
    public void DibujaUnCirculo(Circulo c) {  
        //dibuja un círculo...  
    }  
    public void DibujaUnRectangulo(Rectangulo r) {  
        //dibuja un rectángulo  
    }  
    public void DibujarForma(IForma forma) {  
        switch (forma.GetTipo()) {  
            case 1:  
                DibujaUnRectangulo((Rectangulo)forma);  
                break;  
            case 2:  
                DibujaUnCirculo((Circulo)forma);  
                break;  
        }  
    }  
}
```

Open-Closed principle (OCP)

```
public abstract class Forma {  
    public abstract void Dibujar();  
}
```

```
public class Rectangulo : Forma {  
    public override void Dibujar() {  
        // Dibuja un Rectangulo  
    }  
}
```

```
public class Circulo : Forma {  
    public override void Dibujar() {  
        // Dibuja un círculo  
    }  
}
```

```
public class EditorGrafico {  
    public void DibujarForma(Forma forma) {  
        forma.Dibujar();  
    }  
}
```

Liskov substitution principle (LSP)

"Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."

Una clase derivada no debe modificar el comportamiento de la clase base.

Los objetos de un software deberían poder reemplazarse por instancias de sus clases derivadas sin alterar el correcto funcionamiento del programa.

Liskov substitution principle (LSP)

```
class Rectangulo {  
    public alto;  
    public ancho;  
    setAlto (altura) {  
        alto = altura;  
    }  
    setAncho (anchura) {  
        ancho = anchura;  
    }  
    public int GetArea() {  
        return alto * ancho;  
    }  
}
```

```
class Cuadrado extends  
Rectangulo {  
    setAlto (altura) {  
        alto = altura;  
        ancho = altura;  
    }  
    setAncho (anchura) {  
        alto = anchura;  
        ancho = anchura;  
    }  
}
```



Liskov substitution principle (LSP)

```
class Main {  
    function comprobarArea (Rectangulo rect) {  
        r = GetNewRectangulo();  
  
        rectangulo.setAlto(3);  
        rectangulo.setAncho (4);  
        area = rectangulo.GetArea();  
  
        return area;  
    }  
}
```

Liskov substitution principle (LSP)

```
Interface IRectangular {  
    setAlto (alto);  
    setAncho (ancho);  
    getArea();  
}
```

```
public class Rectangulo implements  
Irectangular {  
    alto;  
    ancho;  
    setAlto (altura) {  
        alto = altura;  
    }  
    setAncho (anchura) {  
        ancho = anchura;  
    }  
    getArea () {  
        return alto * ancho;  
    }  
}
```

```
public class Cuadrado  
implements Irectangular {  
    lado;  
    setAlto (altura) {  
        lado = altura;  
    }  
    setAncho (anchura) {  
        lado = anchura;  
    }  
    getArea () {  
        return lado * lado;  
    }  
}
```



Linux
Málaga



Interface segregation principle (ISP)

"Clients should not be forced to depend upon interfaces that they do not use."

Las interfaces han de ser simples y proveer pocos métodos.

Es preferible disponer de muchos interfaces pequeños que pocos interfaces muy cargados. Esto permite en gran medida reaprovechar esos interfaces en otras clases.

Dependency inversion principle (DIP)

"A. High level modules should not depend upon low level modules. Both should depend upon abstractions."

B. Abstractions should not depend upon details. Details should depend upon abstractions."

Depender de abstracciones, no de concreciones.



Dependency inversion principle (DIP)

```
public class Motor {  
    arrancar();  
    parar();  
}  
  
public class Coche {  
    private Motor motor;  
    public Coche() {  
        motor = new Motor();  
    }  
}
```

Dependency inversion principle (DIP)

```
Interfaz IMotor {  
    arrancar();  
    parar();  
}  
public class Coche {  
    private IMotor motor;  
    public coche (IMotor m) {  
        motor = m;  
    }  
}
```

Patrón DAO

El patrón de desarrollo DAO se apoya en el principio al ISP al separar en interfaces las acciones propias de persistencia en la base de datos para una clase de lo que es su lógica de negocio.

Igualmente cumple con patrón SRP al separar las responsabilidades.

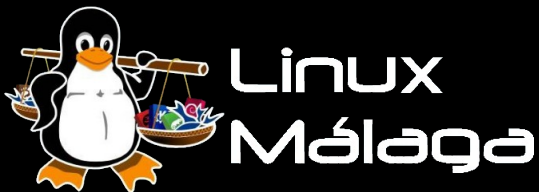
Patrón GenericDAO

Si el patrón DAO no se implementa correctamente romperemos el principio DRY.

El patrón genericDAO lo amplia y hace que además de los principios ISP y SRP cumplamos con el patrón DRY.

Patrón Abstract Factory

El patrón de desarrollo Abstract Factory hace uso del principio OCP y además cumple con DRY ya que permite que la extensión de la funcionalidad y evita el código repetido.



Patrón de inyección de dependencias

Se basa en el principio DIP de inyección de dependencias.

Evitar las sentencias new.

Todas las clases deben crearse mediante factorías.

Antipatrones

Parálisis del análisis
Diseñar por diseñar
Clase gorda
Punto de vista ambiguo
Ancla de barco
Hard code
Confianza ciega
Retorno positivo

Programar por casos especiales
Infierno de dependencias
Codificación brutal
Contenedor mágico
Máquina de Rube Goldberg
Matrimonio para siempre
Teoría de la plastilina

Resumiendo

Aplicar estos principios y patrones en la medida de lo posible ayudarán a hacer mejor y mas mantenible nuestro código.

Por encima de todos ellos está siempre el sentido común.

Aplicarlos supone un esfuerzo inicial mayor para recuperarlo con creces.

Si aún no estás convencido piensa en lo siguiente:



Resumiendo

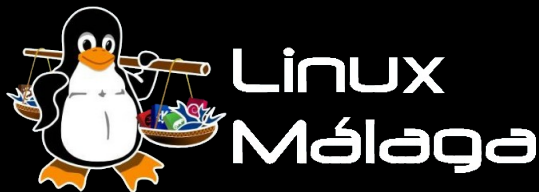
Aplicar estos principios y patrones en la medida de lo posible ayudarán a hacer mejor y mas mantenible nuestro código.

Por encima de todos ellos está siempre el sentido común.

Aplicarlos supone un esfuerzo inicial mayor para recuperarlo con creces.

Si aún no estás convencido piensa en lo siguiente:

“Lo mas probable es que el siguiente que va a tener que mantener el código que tu has creado seas tu mismo”. Robert C. Martin





Linux
Málaga

@linux_malaga
www.linux-malaga.org

Muchas gracias por
vuestra atención.

Juan Pablo Guerrero Durán

@bartola_

