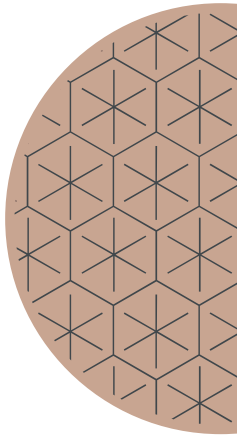
Principios Programación

Principios Programación

Es fácil escribir código. Es un reto escribir buen código.

El código malo se presenta de muchas formas: código desordenado, cadenas masivas de if-else, programas que se rompen con un solo ajuste, variables que no tienen sentido, multitud de bucles anidados en el mismo método.

Si quieres convertirte en programador intenta escribir un código que sea fácil de mantener tanto para ti como para cualquier otro desarrollador de tu equipo.



¿Qué es SOLID?

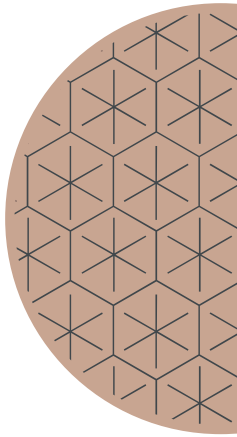
Los principios SOLID fueron introducidos por Robert C. Martin en su documento del año 2000 "Design Principles and Design Patterns". Estos conceptos fueron desarrollados posteriormente por Michael Feathers, quien nos presentó el acrónimo SOLID.

En los últimos 20 años, estos cinco principios han revolucionado el mundo de la programación orientada a objetos, cambiando la forma en que escribimos el software.

¿Qué es SOLID?

Entonces, ¿qué es SOLID y cómo nos ayuda a escribir mejor código? En pocas palabras, los principios de diseño de Martin y Feathers nos animan a crear un software más mantenible, comprensible y flexible.

En consecuencia, a medida que nuestras aplicaciones aumentan de tamaño, podemos reducir su complejidad y ahorrarnos muchos dolores de cabeza en el futuro.

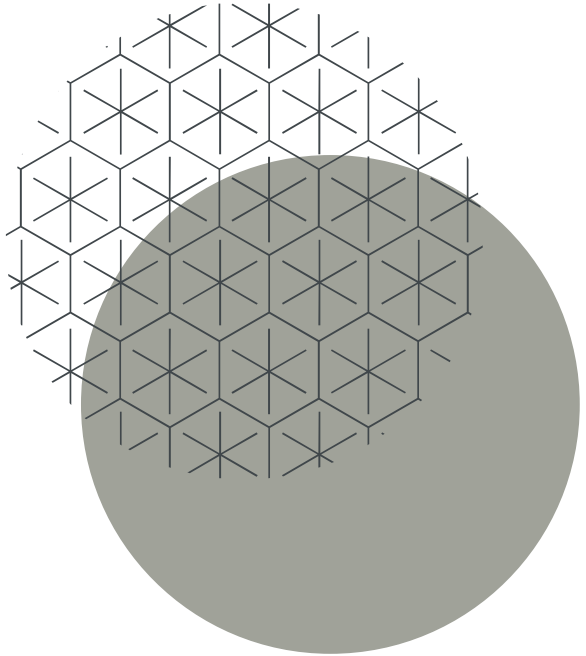


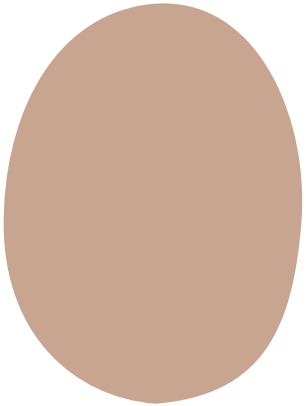
¿Qué es SOLID?

Los siguientes cinco conceptos conforman nuestros principios SOLID:

- Single Responsibility
- Open/Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

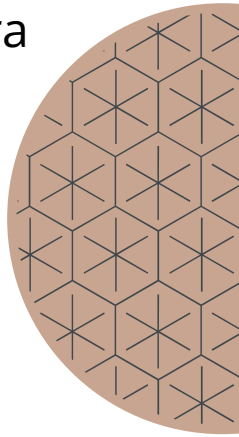
Single Responsibility





Single Responsibility

Comencemos con el principio de Single Responsibility. Como es de esperar, este principio establece que una clase sólo debe tener una responsabilidad. Además, sólo debe tener una razón para cambiar.



Single Responsibility

¿Cómo nos ayuda este principio a construir un mejor software? Veamos algunos de sus beneficios:

- **Pruebas**: Una clase con una sola responsabilidad tendrá muchos menos casos de prueba.
- **Menor acoplamiento:** Una clase con una sola funcionalidad tendrá menos dependencias.
- **Organización:** Las clases más pequeñas y bien organizadas son más fáciles de buscar que las monolíticas.

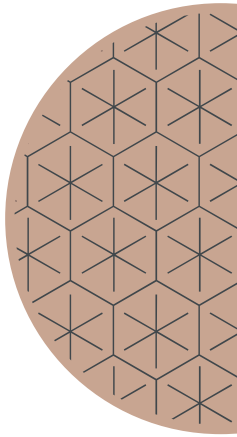
Single Responsibility

Por ejemplo, veamos una clase para representar un simple libro:

```
public class Book {

    private String name;
    private String author;
    private String text;

    //constructor, getters and setters
}
```

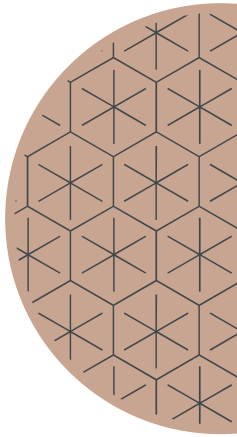
Single Responsibility

En este código, almacenamos el nombre, el autor y el texto asociados a una instancia de un Libro. Añadamos ahora un par de métodos para consultar el texto.



Single Responsibility

```
public class Book {

    private String name;
    private String author;
    private String text;
    // methods that directly relate to the book properties
    public String replaceWordInText(String word){
        return text.replaceAll(word, text);
    }

    public boolean isWordInText(String word){
        return text.contains(word);
    }
}
```



Single Responsibility

Sin embargo, este código viola el principio de responsabilidad única que hemos señalado antes. Para arreglar nuestro desorden, debemos implementar una clase separada que se ocupe sólo de imprimir nuestros textos:

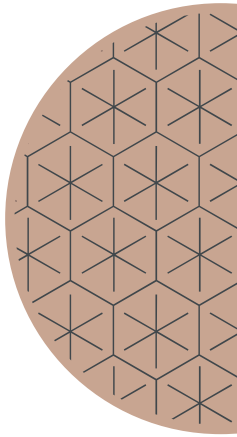
Single Responsibility

```
public class BookPrinter {

    // methods for outputting text
    void printTextToConsole(String text){
        //our code for formatting and printing the text
    }

    void printTextToAnotherMedium(String text){
        // code for writing to any other location..
    }
}
```

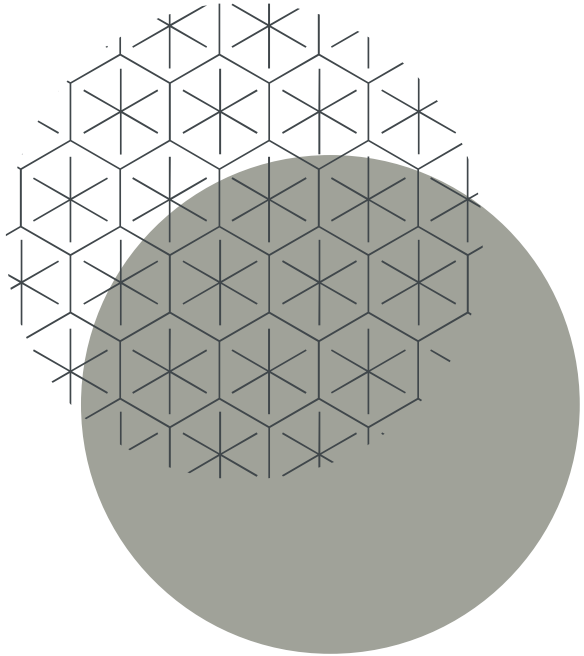
Single Responsibility

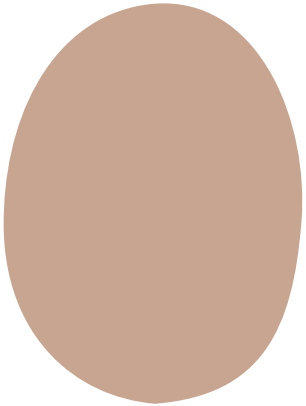
No sólo hemos desarrollado una clase que libera al Libro de sus tareas de impresión, sino que también podemos aprovechar nuestra clase BookPrinter para enviar nuestro texto a otros medios.

Ya sea correo electrónico, registro, o cualquier otra cosa, tenemos una clase separada dedicada a esta preocupación.



Open/Closed

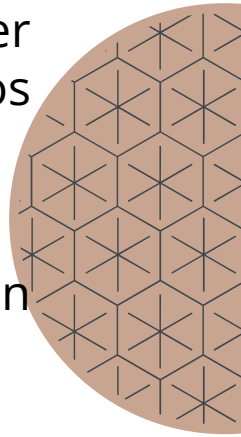




Open/Closed

Ahora es el momento de la O de SOLID, conocida como el principio abierto-cerrado. En pocas palabras, las clases deben estar abiertas para su ampliación, pero cerradas para su modificación. Al hacer esto, evitamos modificar el código existente y causar nuevos errores potenciales en una aplicación.

Por supuesto, la única excepción a la regla es cuando se arreglan errores en el código existente.



Open/Closed

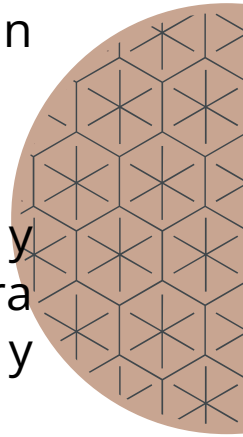
Exploremos el concepto con un rápido ejemplo de código. Como parte de un nuevo proyecto, imaginemos que hemos implementado una clase de guitarra.

Open/Closed

```
public class Guitar {

    private String make;
    private String model;
    private int volume;

    //Constructors, getters & setters
}
```

Open/Closed

Lanzamos la aplicación, y a todo el mundo le encanta. Pero después de unos meses, decidimos que la Guitarra necesitaba cambiarla y que le vendría bien un patrón de llamas para darle un aspecto más rockero.

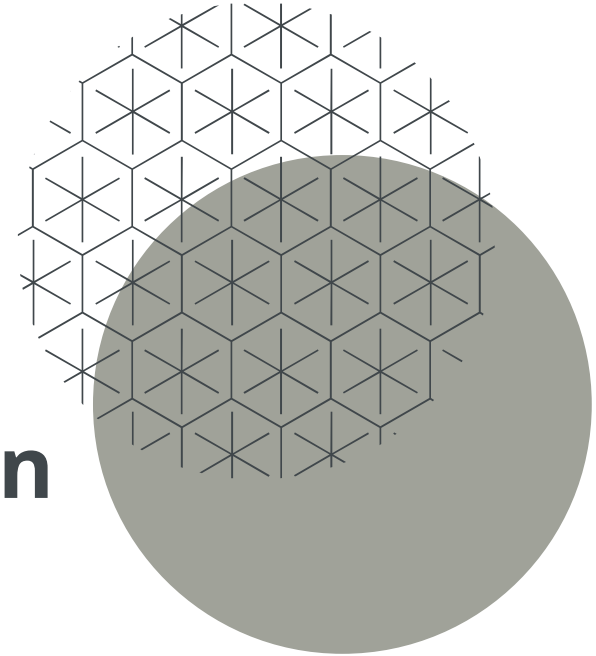


En este punto, podría ser tentador abrir la clase Guitarra y modificarla pero quién sabe qué errores podría arrojar en nuestra aplicación. En su lugar, vamos a seguir el principio de abrir-cerrar y simplemente extender nuestra clase Guitarra:

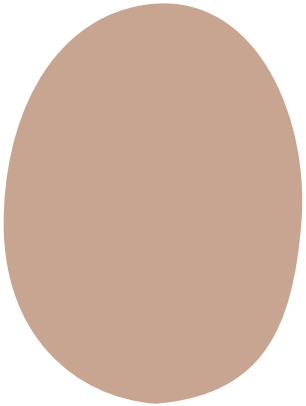
Open/Closed

```
public class SuperCoolGuitarWithFlames extends Guitar {

    private String flameColor;

    //constructor, getters + setters
}
```

Al extender la clase Guitarra, podemos estar seguros de que nuestra aplicación existente no se verá afectada.



Liskov Substitution



Liskov Substitution

El siguiente en nuestra lista es la sustitución de Liskov, que es posiblemente el más complejo de los cinco principios. En pocas palabras, si la clase A es un subtipo de la clase B, deberíamos poder sustituir B por A sin alterar el comportamiento de nuestro programa.

Pasemos directamente al código para ayudarnos a entender este concepto:

Liskov Substitution

```
public interface Car {

    void turnOnEngine();
    void accelerate();
}
```

Arriba, definimos una simple interfaz de Coche con un par de métodos que todos los coches deberían poder cumplir: encender el motor y acelerar hacia adelante.

Implementemos nuestra interfaz y proporcionemos algo de código para los métodos:

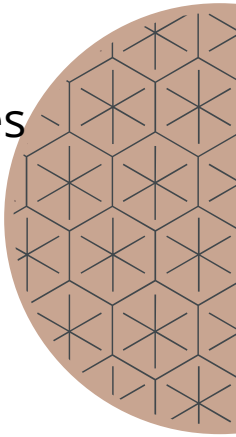
Liskov Substitution

```
public class MotorCar implements Car {

    private Engine engine;

    public void turnOnEngine() {
        //turn on the engine!
        engine.on();
    }

    public void accelerate() {
        //move forward!
        engine.powerOn(1000);
    }
}
```

Liskov Substitution

Como describe nuestro código, tenemos un motor que podemos encender, y podemos aumentar la potencia.

Pero espera, ahora estamos viviendo en la era de los coches eléctricos:

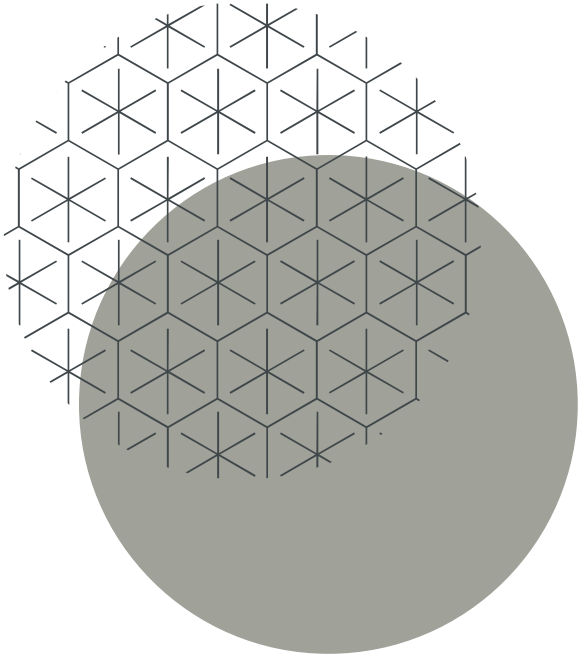


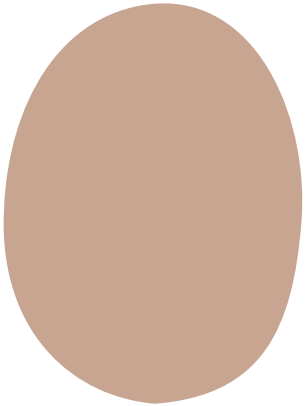
Liskov Substitution

Al lanzar un coche sin motor en la mezcla, estamos cambiando inherentemente el comportamiento de nuestro programa. Esto es una violación flagrante de la sustitución de Liskov y es un poco más difícil de arreglar que nuestros dos principios anteriores.

Una posible solución sería rehacer nuestro modelo en interfaces que tengan en cuenta el estado sin motor de nuestro coche.

Interface Segregation





Interface Segregation

La I de SOLID significa segregación de interfaces, y significa simplemente que las interfaces más grandes deben dividirse en otras más pequeñas. De este modo, nos aseguramos de que las clases implementadoras sólo tengan que preocuparse por los métodos que les interesan.

Para este ejemplo, vamos a probar nuestras manos como guardianes del zoo. Y más concretamente, trabajaremos en el recinto de los osos.

Interface Segregation

Empecemos con una interfaz que describe nuestras funciones como cuidadores de osos:

```
public interface BearKeeper {
    void washTheBear();
    void feedTheBear();
    void petTheBear();
}
```

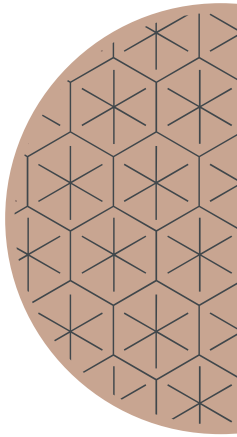
Interface Segregation

Como ávidos cuidadores de zoológicos, estamos más que contentos de lavar y alimentar a nuestros queridos osos. Pero somos muy conscientes de los peligros de acariciarlos. Por desgracia, nuestra interfaz es bastante grande, y no tenemos más remedio que implementar el código para acariciar al oso.

Vamos a solucionarlo dividiendo nuestra gran interfaz en tres independientes:

Interface Segregation

```
public interface BearCleaner {
    void washTheBear();
}

public interface BearFeeder {
    void feedTheBear();
}

public interface BearPetter {
    void petTheBear();
}
```



Interface Segregation

Ahora, gracias a la segregación de interfaces, somos libres de implementar sólo los métodos que nos interesan:

```
public class BearCarer implements BearCleaner, BearFeeder {

    public void washTheBear() {
        //I think we missed a spot...
    }

    public void feedTheBear() {
        //Tuna Tuesdays...
    }
}
```

Interface Segregation

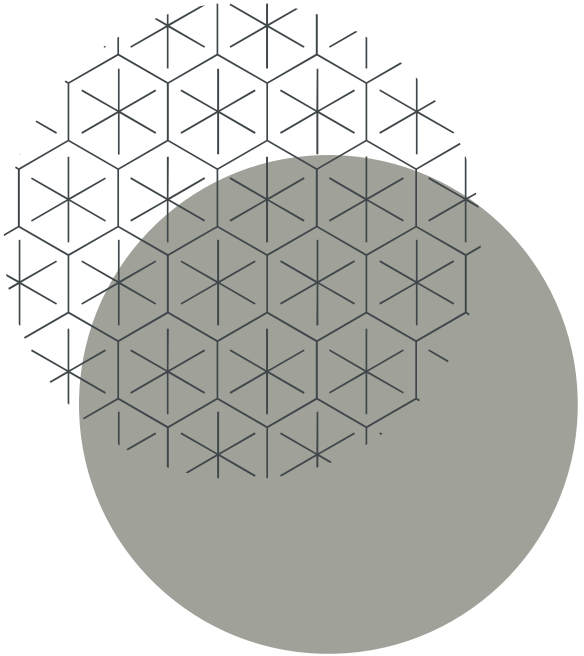
Y por último, podemos dejar las cosas peligrosas para los temerarios:

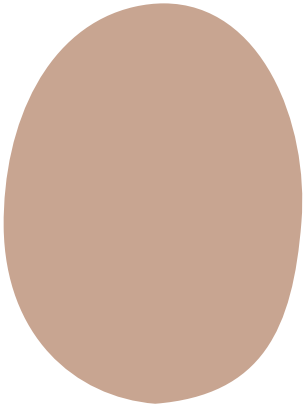
```
public class CrazyPerson implements BearPetter {

    public void petTheBear() {
        //Good luck with that!
    }
}
```

Interface Segregation

Yendo más allá, podríamos incluso dividir nuestra clase BookPrinter de nuestro ejemplo anterior para utilizar la segregación de interfaces de la misma manera. Implementando una interfaz de impresora con un único método de impresión, podríamos instanciar clases ConsoleBookPrinter y OtherMediaBookPrinter por separado.

Dependency Inversion

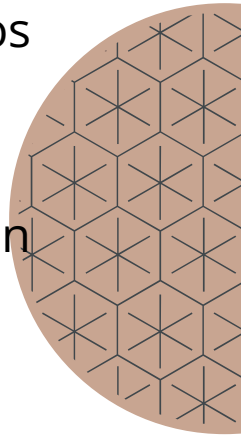




Dependency Inversion

El principio de inversión de la dependencia se refiere al desacoplamiento de los módulos de software. Así, en lugar de que los módulos de alto nivel dependan de los de bajo nivel, ambos dependerán de las abstracciones.

Para demostrarlo, vayamos a la vieja escuela y demos vida a un ordenador Windows 98 con código



```
public class Windows98Machine {}
```

Dependency Inversion

Añadamos uno de cada uno a nuestro constructor para que cada Windows98Computer que instanciemos venga preempacado con un Monitor y un Teclado estándar:

```
public class Windows98Machine {
    private final StandardKeyboard keyboard;
    private final Monitor monitor;

    public Windows98Machine() {
        monitor = new Monitor();
        keyboard = new StandardKeyboard();
    }

}
```

Dependency Inversion

Este código funcionará, y podremos utilizar el StandardKeyboard y el Monitor libremente dentro de nuestra clase Windows98Computer.

¿Problema resuelto? No del todo. Al declarar el StandardKeyboard y el Monitor con la palabra clave new, hemos unido fuertemente estas tres clases.

Dependency Inversion

Esto no sólo hace que nuestro Windows98Computer difícil de probar, pero también hemos perdido la capacidad de cambiar nuestra clase StandardKeyboard con una diferente en caso de necesidad. Y también estamos atascados con nuestra clase Monitor.

Vamos a desacoplar nuestra máquina del StandardKeyboard añadiendo una interfaz de teclado más general y utilizando esto en nuestra clase:

Dependency Inversion

```
public interface Keyboard { }
public interface Monitor{ }
public class Windows98Machine{

    private final Keyboard keyboard;
    private final Monitor monitor;

    public Windows98Machine(Keyboard keyboard, Monitor monitor)
{
        this.keyboard = keyboard;
        this.monitor = monitor;
    }
}
```

Dependency Inversion

Aquí, estamos utilizando el patrón de inyección de dependencia para facilitar la adición de la dependencia del teclado en la clase Windows98Machine.

Modifiquemos también nuestra clase StandardKeyboard para que implemente la interfaz Keyboard de forma que sea adecuada para inyectarla en la clase Windows98Machine:

```
public class StandardKeyboard implements Keyboard { }
```

Dependency Inversion

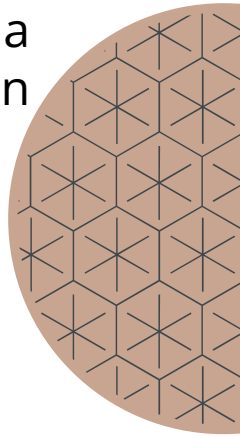
Ahora nuestras clases están desacopladas y se comunican a través de la abstracción Keyboard. Si queremos, podemos cambiar fácilmente el tipo de teclado de nuestra máquina con una implementación diferente de la interfaz. Podemos seguir el mismo principio para la clase Monitor.

Hemos desacoplado las dependencias y somos libres de probar nuestra Windows98Machine con cualquier marco de pruebas que elijamos.

Otros principios

DRY

DRY son las siglas de "Don't Repeat Yourself" (no te repitas), un principio básico del desarrollo de software cuyo objetivo es reducir la repetición de información. El principio DRY se enuncia así: "Cada pieza de conocimiento o lógica debe tener una representación única e inequívoca dentro de un sistema".



KISS

El principio KISS (Keep It Simple, Stupid) es descriptivo para mantener el código simple y claro, haciéndolo fácil de entender.

Después de todo, los lenguajes de programación son para que los humanos los entiendan así que mantén la codificación simple y directa. Mantén tus métodos pequeños. Cada método no debería tener más de 10-20 líneas ni complejidad lógica: Bucles anidados de for, anidamiento de if else, etc.

YAGNI

Este principio significa que nunca debes codificar para la funcionalidad en la posibilidad de que puedas necesitar algo en el futuro. Uno de los principios más importantes de la programación informática que hay que aprender es que no hay que intentar resolver un problema que no existe.

A menudo, los programadores inexpertos tratan de escribir el código más abstracto y genérico que puedan. Sin embargo, demasiada abstracción provoca un código hinchado que es imposible de mantener.

FIN