



SOLID

Prof: Lic. Rosales Pablo (prosales@unpata.edu.ar)
Materia: Programación Orientada a Objetos
Año: 2024





Principios SOLID

1. **S**: Single responsibility
2. **O**: Open-closed
3. **L**: Liskov substitution
4. **I**: Interface segregation
5. **D**: Dependency Inversion

5 principios desarrollados por Robert C. Martin



Single responsibility

Una clase solo debe ser responsable por una cosa. Si una clase tiene más de una responsabilidad se acopla. Un cambio en una responsabilidad resulta en cambios para la otra responsabilidad.

aplica no solo a clases sino también a SW, componentes y servicios.

la clase usuario maneja las propiedades y el almacenamiento en base de datos

bien aplicado la aplicación se vuelve altamente cohesiva.

```
class Usuario:
    def __init__(self, nombre, edad):
        self.__nombre = nombre
        self.__edad = edad
    def save_usuario(self):
        pass

    @property
    def nombre(self):
        return self.__nombre

    @nombre.setter
    def nombre(self, nombre):
        self.__nombre = nombre
```

Al diseñar clases, tenemos que apuntar a unir características relacionadas, de modo que cuando cambien, lo hagan por la misma razón. Y deberíamos separar las características si cambian por diferentes razones.

Steve Fenton



Open - Closed

Las entidades de SW (Clases módulos funciones) deben ser abiertas para extensión y no para modificación.


```
animales = [Animal("Perro"), Animal("Gato")]

for animal in animales:
    animal.imprimir_sonidos()
```

```
class Animal:
    def __init__(self, nombre):
        self.__nombre = nombre

    def imprimir_sonidos(self):
        if self.__nombre == "Perro":
            print("Guau!")
        elif self.__nombre == "Gato":
            print("Miau!")

    @property
    def nombre(self):
        return self.__nombre
```



La función imprimirSonidos está cerrada para nuevos animales, si se nos suma uno más es necesario agregar otro if.

```
class Animal:
    def __init__(self, nombre):
        self.__nombre = nombre

    def imprimir_sonidos(self):
        if self.__nombre == "Perro":
            print("Guau!")
        elif self.__nombre == "Gato":
            print("Miau!")

    @property
    def nombre(self):
        return self.__nombre
```

Solución

```
from abc import ABC, abstractmethod

class Animal(ABC):
    def __init__(self, nombre):
        self.__nombre = nombre

    @abstractmethod
    def imprimir_sonidos(self):
        pass

    @property
    def nombre(self):
        return self.__nombre

class Perro(Animal):
    def imprimir_sonidos(self):
        print("Guau!")

class Gato(Animal):
    def imprimir_sonidos(self):
        print("Miau!")

animales = [Perro("roberto"), Gato("michifus")]

for animal in animales:
    animal.imprimir_sonidos()
```

Liskov Substitution

Una subclase debe ser sustituible por su superclase.

El objetivo es comprobar que una subclase puede asumir el lugar del padre sin errores, si es necesario tener que andar haciendo comprobación de tipos, no se cumple el principio de Liskov Substitution.

```
animales = [Perro("roberto"), Gato("michifus"), Anaconda("juana")]

for animal in animales:
    if type(animal) == Perro:
        print("cantidad de patas de perro: " + str(animal.cantidad_patas()))
    elif type(animal) == Gato:
        print("cantidad de patas de gato: " + str(animal.cantidad_patas()))
    elif type(animal) == Anaconda:
        print("cantidad de patas de anaconda: " + str(animal.cantidad_patas()))
```

```
class Animal():
    def __init__(self, nombre):
        self.__nombre = nombre

    def imprimir_sonidos(self):
        pass
    def cantidad_patas(self):
        pass

    @property
    def nombre(self):
        return self.__nombre
```

```
class Perro(Animal):
    def imprimir_sonidos(self):
        print("Guau!")

    def cantidad_patas(self):
        return 4
```

```
class Gato(Animal):
    def imprimir_sonidos(self):
        print("Miau!")

    def cantidad_patas(self):
        return 4
```

```
class Anaconda(Animal):
    def imprimir_sonidos(self):
        print("?")
```



```

animales = [Perro("roberto"), Gato("michifus"), Anaconda("juana")]

for animal in animales:
    if type(animal) == Perro:
        print("cantidad de patas de perro: " + str(animal.cantidad_patas()))
    elif type(animal) == Gato:
        print("cantidad de patas de gato: " + str(animal.cantidad_patas()))
    elif type(animal) == Anaconda:
        print("cantidad de patas de anaconda: " + str(animal.cantidad_patas()))

```

0.0s

Python

```

-----
peError                                Traceback (most recent call last)
ome/pablo/Desktop/notebooks/notebook_solid.ipynb Cell 9 in 3
    34     def imprimir_sonidos(self):
    35         print("?")
-> 38 animales = [Perro("roberto"), Gato("michifus"), Anaconda("juana")]
    40 for animal in animales:
    41     if type(animal) == Perro:

peError: Can't instantiate abstract class Anaconda with abstract method cantidad_p

```

```

from abc import ABC, abstractmethod

```

```

class Animal(ABC):
    def __init__(self, nombre):
        self.__nombre = nombre

    @abstractmethod
    def imprimir_sonidos(self):
        pass

    @abstractmethod
    def cantidad_patas(self):
        pass

    @property
    def nombre(self):
        return self.__nombre

```

```

class Perro(Animal):
    def imprimir_sonidos(self):
        print("Guau!")

    def cantidad_patas(self):
        return 4

```

```

class Gato(Animal):
    def imprimir_sonidos(self):
        print("Miau!")

    def cantidad_patas(self):
        return 4

```

```

class Anaconda(Animal):
    def imprimir_sonidos(self):
        print("?")

```



Interface Segregation

Las interfaces tienen que ser específicas. Las clases que implementan interfaces no deben verse obligadas a implementar métodos que no necesitan.

```
from abc import ABC, abstractmethod

class Animal(ABC):

    @abstractmethod
    def emitir_sonido(self):
        pass

    @abstractmethod
    def comer_plantas(self):
        pass

    @abstractmethod
    def comer_carne(self):
        pass

    @abstractmethod
    def comer_insectos(self):
        pass

class Gato (Animal):
    def comer_carne(self):
        print("si come carne..")

un_gato = Gato()
un_gato.comer_carne()
```



Dependency Inversion

Las dependencias deben ser sobre abstracciones no sobre concreciones.

1. Módulos de alto nivel no deben de depender de los de bajo nivel. Ambas deben de depender de abstracciones
2. Abstracciones no deben de depender de los detalles, los detalles deben depender de las abstracciones.

```
class BaseDeDatos(Postgresql, Mysql, Oracle):  
    pass
```

```
class ConexionPostgres (Conexion):  
    pass
```



Profundizar en:

“Agile Software Development, Principles, Patterns, and Practices” Robert C. Martin
capítulos 8 al 12.