PRINCIPIOS SOLID

Para una mejor calidad de software.

Los principios SOLID son un conjunto de cinco principios de diseño que ayudan a crear código más legible, reutilizable, flexible y fácil de mantener. Estos principios se basan en la idea de que el buen diseño del software debería permitir que el código sea adaptable a los cambios sin causar demasiadas alteraciones.

SRP - Single Responsability Principle.

El Principio de Responsabilidad Única dice que **una clase debe hacer una cosa y, por lo tanto, debe tener una sola razón para cambiar.**

Para enunciar este principio más técnicamente: Solo un cambio potencial (lógica de base de datos, lógica de registro, etc.) en la especificación del software debería poder afectar la especificación de la clase.

Código que no sique SRP

```
class Empleado:
    def __init__(self, nombre, cargo, salario):
        self.nombre = nombre
        self.cargo = cargo
        self.salario = salario

    def calcular_salario(self):
```

```
        # Lógica para calcular el salario
        return self.salario * 1.1  # Ejemplo simple: 10% de aumento

    def guardar_en_base_de_datos(self):
        # Lógica para guardar el empleado en la base de datos
        print(f"Guardando empleado {self.nombre} en la base de datos")

    def generar_reporte(self):
        # Lógica para generar un reporte del empleado
        return f"Reporte de {self.nombre}: Cargo: {self.cargo}, Salario:
{self.salario}"

# Uso
empleado = Empleado("Juan", "Desarrollador", 50000)
nuevo_salario = empleado.calcular_salario()
empleado.guardar_en_base_de_datos()
reporte = empleado.generar_reporte()
```

Código que sigue SRP

```
class Empleado:
    def __init__(self, nombre, cargo, salario):
        self.nombre = nombre
        self.cargo = cargo
        self.salario = salario

class CalculadoraSalario:
    def calcular_salario(self, empleado):
        # Lógica para calcular el salario
        return empleado.salario * 1.1  # Ejemplo simple: 10% de aumento

class AlmacenadorEmpleado:
    def guardar_en_base_de_datos(self, empleado):
        # Lógica para guardar el empleado en la base de datos
        print(f"Guardando empleado {empleado.nombre} en la base de datos")

class GeneradorReporteEmpleado:
    def generar_reporte(self, empleado):
        # Lógica para generar un reporte del empleado
        print(f"Reporte de {empleado.nombre}: Cargo: {empleado.cargo},
Salario: {empleado.salario}")

# Uso
empleado = Empleado("Juan", "Desarrollador", 50000)

calculadora = CalculadoraSalario()
nuevo_salario = calculadora.calcular_salario(empleado)

almacenador = AlmacenadorEmpleado()
almacenador.guardar_en_base_de_datos(empleado)

generador_reporte = GeneradorReporteEmpleado()
reporte = generador_reporte.generar_reporte(empleado)
```

OCP - (Open Closed Principle)

El segundo principio de SOLID lo formuló Bertrand Meyer en 1988 en su libro “Object Oriented Software Construction” y dice: “Deberías ser capaz de extender el comportamiento de una clase, sin modificarla”. En otras palabras: las clases que usas deberían estar **abiertas para poder extenderse y cerradas para modificarse** .

En su blog Robert C. Martin defendió este principio que a priori puede parecer una paradoja. Es importante tener en cuenta el **Open/Closed Principle (OCP)** a la hora de desarrollar **clases, librerías o frameworks.**

Codigo que no sigue OCP.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

class AreaCalculator:
    def calculate_area(self, rectangle):
        return rectangle.width * rectangle.height

# Uso
rectangle = Rectangle(5, 4)
calculator = AreaCalculator()
area = calculator.calculate_area(rectangle)
print(f"El área del rectángulo es: {area}")
```

Código que cumple con OCP

```
from abc import ABC, abstractmethod
import math

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
```

```
    def area(self):
        return self.width * self.height

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

class AreaCalculator:
    def calculate_area(self, shape):
        return shape.area()

# Uso
rectangle = Rectangle(5, 4)
circle = Circle(3)

calculator = AreaCalculator()

rectangle_area = calculator.calculate_area(rectangle)
circle_area = calculator.calculate_area(circle)

print(f"El área del rectángulo es: {rectangle_area}")
print(f"El área del círculo es: {circle_area:.2f}")


# class Triangle(Shape):
#     def __init__(self, base, height):
#         self.base = base
#         self.height = height

#     def area(self):
#         return 0.5 * self.base * self.height

# # Uso
# triangle = Triangle(6, 4)
# triangle_area = calculator.calculate_area(triangle)
# print(f"El área del triángulo es: {triangle_area}")
```

LSP - (Liskov Sustitution Principle)

La L de SOLID alude al apellido de quien lo creó, Barbara Liskov, y dice que **“las clases derivadas deben poder sustituirse por sus clases base”.**

Esto significa que los objetos deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del sistema o lo que es lo mismo: si en un programa utilizamos cierta clase, **deberíamos poder usar cualquiera de sus subclases** sin interferir en la funcionalidad del programa.

Según Robert C. Martin incumplir el **Liskov Substitution Principle (LSP)** implica violar también el principio de Abierto/Cerrado.

Código que no sigue LSP

```
class Ave:
    def volar(self):
        print("Esta ave está volando")

class Pinguino(Ave):
    def volar(self):
        raise Exception("Los pingüinos no pueden volar")

def hacer_volar_ave(ave: Ave):
    ave.volar()

# Uso
golondrina = Ave()
pinguino = Pinguino()

hacer_volar_ave(golondrina)
try:
    hacer_volar_ave(pinguino)
 # Funciona bien
except Exception as e:
    print(f"Error->{e}")   # Lanza una excepción
```

Código que cumple con LSP

```
from abc import ABC, abstractmethod

class Ave(ABC):
    @abstractmethod
    def mover(self):
        pass

class AveVoladora(Ave):
    def mover(self):
        print("Esta ave está volando")

class AveNoVoladora(Ave):
    def mover(self):
        print("Esta ave está caminando")

class Golondrina(AveVoladora):
    pass

class Pinguino(AveNoVoladora):
    pass
```

```
def hacer_mover_ave(ave: Ave):
    ave.mover()

# Uso
golondrina = Golondrina()
pinguino = Pinguino()

hacer_mover_ave(golondrina)  # Imprime: Esta ave está volando
hacer_mover_ave(pinguino)    # Imprime: Esta ave está caminando
```

ISP - (Interface Segregation Principle)

La segregación significa mantener las cosas separadas, y el Principio de Segregación de Interfaces se trata de separar las interfaces.

El principio establece que muchas interfaces específicas del cliente son mejores que una interfaz de propósito general. No se debe obligar a los clientes a implementar una función que no necesitan.

Código que no sigue ISP

```
from abc import ABC, abstractmethod

class MultifunctionPrinter(ABC):
    @abstractmethod
    def print(self, document):
        pass

    @abstractmethod
    def scan(self, document):
        pass

    @abstractmethod
    def fax(self, document):
        pass

class AllInOnePrinter(MultifunctionPrinter):
    def print(self, document):
        print(f"Printing: {document}")

    def scan(self, document):
        print(f"Scanning: {document}")

    def fax(self, document):
        print(f"Faxing: {document}")

class OldPrinter(MultifunctionPrinter):
    def print(self, document):
```

```
        print(f"Printing: {document}")

    def scan(self, document):
        raise NotImplementedError("This printer cannot scan.")

    def fax(self, document):
        raise NotImplementedError("This printer cannot fax.")
```

Código que cumple con ISP

```
from abc import ABC, abstractmethod

class Printer(ABC):
    @abstractmethod
    def print(self, document):
        pass

class Scanner(ABC):
    @abstractmethod
    def scan(self, document):
        pass

class Fax(ABC):
    @abstractmethod
    def fax(self, document):
        pass

class AllInOnePrinter(Printer, Scanner, Fax):
    def print(self, document):
        print(f"Printing: {document}")

    def scan(self, document):
        print(f"Scanning: {document}")

    def fax(self, document):
        print(f"Faxing: {document}")

class OldPrinter(Printer):
    def print(self, document):
        print(f"Printing: {document}")

class NewPrinter(Printer, Scanner):
    def print(self, document):
        print(f"Printing: {document}")

    def scan(self, document):
        print(f"Scanning: {document}")


def print_document(printer: Printer, document):
```

```
    printer.print(document)

def scan_document(scanner: Scanner, document):
    scanner.scan(document)

# Uso
all_in_one = AllInOnePrinter()
old_printer = OldPrinter()
new_printer = NewPrinter()

print_document(all_in_one, "all_in_one->Informe anual")
print_document(old_printer, "old_printer->Carta")
print_document(new_printer, "new_printer->Foto")

scan_document(all_in_one, "all_in_one->Contrato")
scan_document(new_printer, "new_printer->ID")

# Esto generaría un error en tiempo de ejecución, ya que OldPrinter no
implementa Scanner
#scan_document(old_printer, "old_printer->Documento")
```

DIP - (Dependency Inversion Principe)

Depende de abstracciones , no de clases concretas:

1. Los módulos de alto nivel **no deberían depender de módulos de bajo nivel** . Ambos deberían depender de abstracciones.
2. **Las abstracciones no deberían depender de los detalles** . Los detalles deberían depender de las abstracciones.

Códifo que no sigue DIP

```
class Database:
    def connect(self):
        print("Conectando a la base de datos...")

class Application:
    def __init__(self):
        self.db = Database()

    def run(self):
        self.db.connect()
        print("Aplicación ejecutándose...")
```

```
app = Application()
app.run()
```

Código que cumple con DIP

```
from abc import ABC, abstractmethod

# Abstracción
class DatabaseInterface(ABC):
    @abstractmethod
    def connect(self) -> None:
        pass

# Implementación concreta de la abstracción
class MySQLDatabase(DatabaseInterface):
    def connect(self) -> None:
        print("Conectando a la base de datos MySQL...")

# Otra implementación concreta (puede ser un mock para pruebas)
class TestDatabase(DatabaseInterface):
    def connect(self) -> None:
        print("Simulando conexión a la base de datos para pruebas...")

# Aplicación que depende de la abstracción
class Application:
    def __init__(self, db: DatabaseInterface) -> None:
        self.db = db

    def run(self) -> None:
        self.db.connect()
        print("Aplicación ejecutándose...")

# Uso del código refactorizado
if __name__ == "__main__":
    db = MySQLDatabase()  # O TestDatabase() para pruebas
    app = Application(db)
    app.run()

    # Pruebas
    print("\nPruebas:\n")
    db2=TestDatabase()
    app2=Application(db2)
    app2.run()
```