Принципи SOLID в ООП

GENIUS space

Принцип єдиного обов'язку (Single Responsibility Principle - SRP)

Принцип SRP стверджує, що кожен клас повинен мати лише одну причину для зміни. Це означає, що клас повинен виконувати лише одну конкретну функцію або обов'язок. Розділення функціональності між класами допомагає зробити код більш зрозумілим та підтримуваним.
Уявімо, що ми створюємо клас **Task**, який представляє завдання. Він має відповідати тільки за представлення завдання, а не за логіку видалення завдань.

```
class Task:
    def __init__(self, description):
        self.description = description

class TaskManager:
    def delete_task(self, task):
        # Логіка видалення завдання збереження історії
```

У цьому прикладі **Task** відповідає тільки за представлення завдання, а логіка видалення та збереження історії розміщена в окремому класі **TaskManager**.

Принцип відкритості/закритості (Open/Closed Principle - OCP)

```python
from abc import ABC, abstractmethod

class OrderProcessor(ABC):
    @abstractmethod
    def process_order(self, order):
        pass

class RegularOrderProcessor(OrderProcessor):
    def process_order(self, order):
        # Логіка обробки звичайного замовлення

class VIPOrderProcessor(OrderProcessor):
    def process_order(self, order):
        # Логіка обробки VIP-замовлення

# Додавання нового типу замовлення без зміни існуючого коду.
class CustomOrderProcessor(OrderProcessor):
    def process_order(self, order):
        # Логіка обробки спеціального замовлення
```

Принцип OCP стверджує, що програмні сутності (класи, модулі тощо) повинні бути відкритими для розширення, але закритими для зміни.
Це означає, що ви можете додавати новий функціонал до існуючих класів, не модифікуючи їхний вихідний код

Розглянемо систему обробки замовлень, де різні типи замовлень можуть мати різну обробку без зміни коду.

У цьому прикладі, для обробки різних типів замовлень ми створюємо базовий клас **OrderProcessor**, а потім створюємо підкласи для конкретних типів замовлень, які реалізують метод **process_order**.
Додавання нового типу замовлення не вимагає зміни існуючого коду.

Принцип підстановки Ліскова (Liskov Substitution Principle - LSP)

```python
from abc import ABC, abstractmethod

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

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

    def area(self):
        return 3.14 * self.radius * self.radius

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

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

Принцип LSP стверджує, що об'єкт базового класу повинен бути замінений об'єктом підкласу без порушення функціональності програми.
Це означає, що всі методи підкласів повинні бути сумісними з методами базового класу.

Уявімо ієрархію класів для геометричних фігур, де кожен підклас може замінити базовий клас без порушення функціональності.

У цьому прикладі **Circle** та **Rectangle** є підкласами **Shape**, і вони замінюють базовий клас, не порушуючи контракту.

Принцип інтерфейсу користувача (Interface Segregation Principle - ISP)

```python
from abc import ABC, abstractmethod

# Інтерфейси
class Printable(ABC):
    @abstractmethod
    def print(self):
        pass

class Scanable(ABC):
    @abstractmethod
    def scan(self):
        pass

# Класи пристроїв
class Printer(Printable):
    def print(self):
        # Логіка друку

class Scanner(Scanable):
    def scan(self):
        # Логіка сканування

class MultifunctionDevice(Printable, Scanable):
    def print(self):
        # Логіка друку для багатофункціонального пристрою

    def scan(self):
        # Логіка сканування для багатофункціонального пристрою
```

Принцип ISP стверджує, що клієнти не повинні залежати від інтерфейсів, які вони не використовують.
Інтерфейси повинні бути достатньо специфічними та маленькими, щоб не нав'язувати зайвих обов'язків клієнтам.

Уявімо систему управління друком, де різні пристрої мають різні можливості.
Принцип ISP допомагає розділити функціональність на окремі інтерфейси.

У цьому прикладі **Printer** та **Scanner** реалізують конкретні інтерфейси, які вони підтримують, і **MultifunctionDevice** реалізує обидва інтерфейси для багатофункціонального пристрою.

GENIUS space

Принцип залежностей (Dependency Inversion Principle - DIP)

```python
from abc import ABC, abstractmethod

# Абстракція для сервісу доставки
class DeliveryService(ABC):
    @abstractmethod
    def deliver(self, order):
        pass

# Конкретні сервіси доставки
class FedEx(DeliveryService):
    def deliver(self, order):
        # Логіка доставки через FedEx

class UPS(DeliveryService):
    def deliver(self, order):
        # Логіка доставки через UPS

# Клас обробки замовлення
class OrderProcessor:
    def __init__(self, delivery_service):
        self.delivery_service = delivery_service

    def process_order(self, order):
        # Логіка обробки замовлення і доставки
        self.delivery_service.deliver(order)
```

Принцип DIP стверджує, що класи вищого рівня не повинні залежати від класів нижчого рівня, а обидва типи класів повинні залежати від абстракцій.
Це допомагає вирішити проблему керованості залежностями і робить код більш гнучким та легким для розширення.

Розглянемо систему доставки товарів, де можна легко змінювати сервіс доставки без зміни класу OrderProcessor.

У цьому прикладі OrderProcessor залежить від абстракції DeliveryService, і ви можете легко змінювати конкретний сервіс доставки (наприклад, з FedEx на UPS) без модифікації класу OrderProcessor.