SOLID 원칙

단일 책임 원칙

Single Responsibility Principle(SRP)

소프트웨어 컴포넌트(클래스) 가 단 하나의 책임을 져야 한다는 원칙이다.

클래스가 유일한 책임이 있다⇒하나의 구체적인 일을 담당한다는 것을 의미한다.

오직 해당 도메인의 문제가 변경되는 경우에만 클래스를 업데이트 해야 한다.

어떠한 경우에도 여러 책임을 가진 객체를 만들어서는 안된다. → 클래스는 작을수록 좋다.

하나의 클래스에 있는 메서드 중에 상호 배타적이며 관련이 없는 것이 있다면, 서로 다른 책임을 가지고 있는 것이므로 더 작은 클래슬 분리할 수 있어야한다.

너무 많은 책임을 가진 클래스

```
Class CustomerManager:
    def __init__(self):
        self.account_balance
        self.order_detail
```

```
    def get_account_balance(self):
        pass
    def get_order_detail(self):
        pass
```

각각의 동작은 나머지 부분과 독립적으로 수행할 수 있다→유지보수를 어렵게 하여 클래스가 오류가 쉽게 나게 된다. → 각각의 책임마다 수정 사유가 발생한다.

책임 분산

솔루션을 관리하기 쉽도록 모든 메서드를 다른 클래스로 분리하여 각 클래스마다 단일 책임을 가지게 하자

```
Class AccountManager:
    def __init__self:
        self.account_balance
    def get_account_balance(self):
        pass

Class OrderManager:
    def __init__(self):
        self.order_detail
    def get_order_detail(self):
        pass
```

담당해야 할 로직이 맞다면 하나의 클래스가 여러 메서드를 가질 수 있다.

개방/폐쇄의 원칙

Open/Close Principle

모듈이 개방되어 있으면서도 폐쇄되어야 한다는 원칙

확장 가능하고 새로운 요구사항이나 도메인 변화에 잘 적응하는 코드를 작성해야 한다는 뜻이다.

이상적으로는 요구사항이 변경되면 새로운 기능을 구현하기 위해 기존 모듈을 확장하되 기존 코드는 수정하면 안된다.

원칙을 따르지 않을 경우 유지보수의 어려움

```
@dataclass
Class AreaCalculator:
    shapes: list

    def total_area(self):
        total=0
        for shape in shapes:
            if shape.type =='circle':
                total+=shape.radius**2
            elif shape.type=='rectangle':
                total+=shape.width*shape.height
            elif shape.type=='triangle':
                total+=shape.width*shape.height//2

Class Shape:
    def __init__(self,name,content):
        self.name=name
        if(self.name=='radius'):
            self.radius=content[0]
        elif self.name=='triangle'
            self.width,self.height=content
        elif sellf.name=='retangle'
            self.width,self.height=content
```

도형의 종류가 추가될때마다 AreaCalculator 와 shape 클래스를 수정해줘야 한다.

확장성을 가진 시스템으로 리팩토링

```
from abc import ABC, abstractmethod
from dataclasses import dataclass
import math

@dataclass
class AreaCalculator:
    shapes: list

    def total_area(self):
        total = 0
        for shape in self.shapes:
            total += shape.area()
        return total

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

@dataclass
class Rectangle(Shape):
    width: int
    height: int

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

@dataclass
class Triangle(Shape):
    width: int
    height: int

    def area(self):
        return (self.width * self.height) / 2

@dataclass
class Circle(Shape):
    radius: int
```

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

도형이 추가될때마다 Shape 클래스를 상속받은 클래스를 상속받아 만들어(기존 모듈을 확장) 기존의 코드가 수정되지 않도록 코드를 구성할 수 있다.

다형성을 따르는 형태의 계약을 만들고 모델을 쉽게 확장할 수 있는 일반적인 구조로 디자인 하는 것이다.

리스코프 치환 원칙

Liscov substitution principle

클라이언트가 특별한 주의를 기울이지 않고도 부모 클래스를 대신하여 하위 클래스를 그대로 사용할 수 있어야 한다는 것이다.

- 매개변수의 타입이 부모/자식 클래스가 같아야 한다.
- 반환되는 타입이 같아야 한다.

```
from abc import ABC, abstractmethod
from dataclasses import dataclass

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

@dataclass
class Rectangle(Shape):
    width: int
    height: int

    def area(self):
```

```
        return self.width * self.height

@dataclass
class Square(Rectangle):
    side: int

    def __post_init__(self):
        self.width = self.side
        self.height = self.side

    def area(self):
        if self.width != self.height:
            raise ValueError("Width and height must be equal
        return self.side * self.side

# AreaCalculator 클래스 정의 (LSP 위반 테스트용)
@dataclass
class AreaCalculator:
    shapes: list

    def total_area(self):
        total = 0
        for shape in self.shapes:
            total += shape.area()
        return total

# 예제 실행
rectangle = Rectangle(width=5, height=10)
square = Square(side=5)

shapes = [rectangle, square]
calculator = AreaCalculator(shapes)
print("Total Area:", calculator.total_area())
```

—> 하위 클래스는 부모 클래스에 정의된 것보다 사전조건을 엄격하게 만들면 안된다.

—> 하위 클래스는 부모 클래스에 정의된 것보다 약한 사후조건을 만들면 안된다.

인터페이스 분리 원칙

Interface Segregation Principle

클라이언트가 자신이 사용하지 않는 인터페이스에 의존하지 않도록 설계해야 한다.

인터페이스는 객체가 노출하는 메서드의 집합이다. 인터페이스는 클래스의 정의와 구현을 분리한다.

파이썬의 인터페이스는 메서드의 형태를 보고 암시적으로 정의된다.

덕 타이핑을 통해 인터페이스를 정의하는 유일한 방법이었지만, 추상 기본 클래스 개념을 통해 특정 중요 메서드가 실제로 재정의되었는지 확인이 필요할 때 유용하다

```
from abc import ABC,abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak():
        pass
    @abstractmethod
    def walk():
        pass

class Human(Animal):
    def speak():
        print("안뇽")
    def walk():
        print("뚜벅뚜벅")
class Dog(Animal):
    def speak():
        print("멍멍")
    def walk():
        print("총총")
class Fish(Animal):
    def speak():
        pass
```

```
    def walk():
        print("어푸어푸")
```

여기서 Fish 클래스는 speak 메서드가 필요없지만 추상 메서드를 다 구현해야 하는 점에서 인터페이스 분리를 통해 구현해야 한다.

```
from abc import ABC,abstractmethod

class Move(ABC):
    @abstractmethod
    def walk():
        pass
class Talk(ABC)
    @abstractmethod
    def speak():
        pass

class Human(Move,Talk)
    def speak():
        print("안뇽")
    def walk():
        print("뚜벅뚜벅")

class Dog(Move,Talk):
    def speak():
        print("멍멍")
    def walk():
        print("총총")
class Fish(Move):
    def walk():
        print("어푸어푸")
```

의존성 역전 원칙

Dependency Inversion Principle

어떤 클래스를 참조해서 사용해야 하는 상황이 생긴다면, 그 클래스를 직접 참조하는 것이 아니라 상위 요소(추상 클래스 또는 인터페이스)로 참조하라는 원칙이다.

```
class CreditCardProcessor:
    def process_payment(self, amount):
        print(f"Processing credit card payment of {amount}.")

class OrderService:
    def __init__(self, payment_processor: CreditCardProcessor
        self.payment_processor = payment_processor

    def place_order(self, amount):
        self.payment_processor.process_payment(amount)


payment_processor = CreditCardProcessor()
order_service = OrderService(payment_processor)
order_service.place_order(100)
```

OrderService가 CreditCardProcessor와 직접 연결되어 있어 결제 방식이 변경되면 OrderService 코드도 수정해야 한다.

의존성 주입을 통한 의존성 역전 원칙 실현

구체 클래스가 아닌 인터페이스와 대화하도록 하는 것이 좋다. 생성자 주입을 통해 의존성 주입을 구현할 수 있다.

```
from abc import ABC, abstractmethod

# 추상화된 PaymentProcessor 인터페이스
class PaymentProcessor(ABC):
    @abstractmethod
    def process_payment(self, amount):
        pass

# 저수준 모듈: CreditCardProcessor
```

```
class CreditCardProcessor(PaymentProcessor):
    def process_payment(self, amount):
        print(f"Processing credit card payment of {amount}.")

# 저수준 모듈: PayPalProcessor
class PayPalProcessor(PaymentProcessor):
    def process_payment(self, amount):
        print(f"Processing PayPal payment of {amount}.")

# 고수준 모듈: OrderService
class OrderService:
    def __init__(self, payment_processor: PaymentProcessor):
        self.payment_processor = payment_processor

    def place_order(self, amount):
        self.payment_processor.process_payment(amount)

# 실행 예제
credit_card_processor = CreditCardProcessor()
paypal_processor = PayPalProcessor()

order_service1 = OrderService(credit_card_processor)
order_service2 = OrderService(paypal_processor)

order_service1.place_order(100)
order_service2.place_order(200)
```