Programmation Orientée Objet en Python

#4 POO Avancée

par David Albert

2023

Table des matières

01

SOLID principles *

*cette section reprend les exemples de @dmmeteo

Single responsability

Explication

Chaque classe / composant logiciel doit avoir une responsable unique.

✓ GOOD

```python
class Animal:
    def __init__(self, name: str):
            self.name = name

    def get_name(self):
        pass


class AnimalDB:
    def get_animal(self) -> Animal:
        pass

    def save(self, animal: Animal):
        pass
```

✕ BAD

```python
class Animal:
    def __init__(self, name: str):
        self.name = name

    def get_name(self) -> str:
        pass

    def save(self, animal: Animal):
        pass
```

↑ Ici, la classe `Animal` a deux responsabilités: gérer la base de données et gérer les attributs.

Open-Closed

Explication

Chaque entité logiciel (classe, module, fonction) doit être ouverte à l'extension mais fermée à la modification.

✓ GOOD

```python
class Discount:
    def __init__(self, customer, price):
        self.customer = customer
        self.price = price

    def get_discount(self):
            return self.price * 0.2


class VIPDiscount(Discount):
    def get_discount(self):
        return super().get_discount() * 2
```

✕ BAD

```python
class Discount:
    def __init__(self, customer, price):
        self.customer = customer
        self.price = price

    def give_discount(self):
            if self.customer == 'fav':
                return self.price * 0.2
            if self.customer == 'vip':
                return self.price * 0.4
```

↑ Ici, au moindre souhait d'ajouter un nouveau type de remise, on devrai toucher le code existant !

Substitution de Liskov

Explication

Une sous-classe doit être substituable à sa super-classe. Le but est de s'assurer qu'une sous-classe peut prendre la place de sa super-classe sans erreur.

✓ GOOD

```
def animal_leg_count(animals: list):
    for animal in animals:
        print(animal.leg_count())

animal_leg_count(animals)
```

✕ BAD

```
def animal_leg_count(animals: list):
    for animal in animals:
        if isinstance(animal, Lion):
            print(lion_leg_count(animal))
        elif isinstance(animal, Mouse):
            print(mouse_leg_count(animal))
        elif isinstance(animal, Pigeon):
            print(pigeon_leg_count(animal))

animal_leg_count(animals)
```

↑ Ici, le type de classe est vérifié, le principe de substitution de Liskov est donc violé !

Séparation des Interfaces

Explication

Fournir des interfaces simples et spécifiques. Ne pas contraindre la personne qui voudra étendre notre logiciel de dépendre d'interfaces qu'il n'utilise pas.

✓ GOOD

```
class IShape:
    def draw(self):
        raise NotImplementedError

class Circle(IShape):
    def draw(self):
        pass

class Square(IShape):
    def draw(self):
        pass

class Rectangle(IShape):
    def draw(self):
        pass
```

✕ BAD

```
class IShape:
    def draw_square(self):
        raise NotImplementedError

    def draw_rectangle(self):
        raise NotImplementedError

    def draw_circle(self):
        raise NotImplementedError
```

← Ici, on pourra utiliser une combinaison de plusieurs interfaces pour créer des formes particulières: demi cercle, triancle rectangle, ...

Inversion des Dépendances

Explication

La dépendance doit porter sur les abstractions et non sur les concrétions.

✓ GOOD

```python
class Connection:
    def request(self, url: str, options: dict):
        raise NotImplementedError

class Http:
    def __init__(self, http_connection: Connection):
        self.http_connection = http_connection

    def get(self, url: str, options: dict):
        self.http_connection.request(url, 'GET')

    def post(self, url, options: dict):
        self.http_connection.request(url, 'POST')

class XMLHttpService(Connection):
    xhr = XMLHttpRequest()
    def request(self, url: str, options:dict):
        self.xhr.open()
        self.xhr.send()
```

✕ BAD

```python
class XMLHttpService(XMLHttpRequestService):
    pass

class Http:
    def __init__(self, xml_http_service: XMLHttpService):
        self.xml_http_service = xml_http_service

    def get(self, url: str, options: dict):
        self.xml_http_service.request(url, 'GET')

    def post(self, url, options: dict):
        self.xml_http_service.request(url, 'POST')
```

↑ Ici, la classe de haut-niveau (http) dépend d'une implémentation spécifique pour le bas niveau et non d'une abstraction.

02

Design patterns

Introduction

Les **patrons de conception** (design patterns) sont des solutions classiques à des problèmes récurrents de la conception de logiciels.

Chaque patron est une sorte de plan ou de schéma que vous pouvez personnaliser afin de résoudre un problème récurrent dans votre code.

Dans ce cours, nous présenterons 4 patrons de conception classiques.

Refactoring Guru contient les explications de 22 patrons de conceptions classiques et leur utilisation.

Factory Method

Fournit une interface pour la création d'objets dans une superclasse, mais permet aux sous-classes de modifier le type d'objets qui seront créés.

Strategy

Builder

Adapter

State

Observer

Patron de méthode

https://refactoring.guru/fr/design-patterns/template-method