개발자라면, SOLID 원칙

좋은 설계를 위한 5가지 프로그래밍 원칙

SOLID 원칙이란?

SOLID 원칙이란?

객체지향 프로그래밍(OOP) 설계의 5가지 '이상향'

SOLID 원칙이란?

객체지향 프로그래밍(OOP) 설계의 5가지 '이상향'

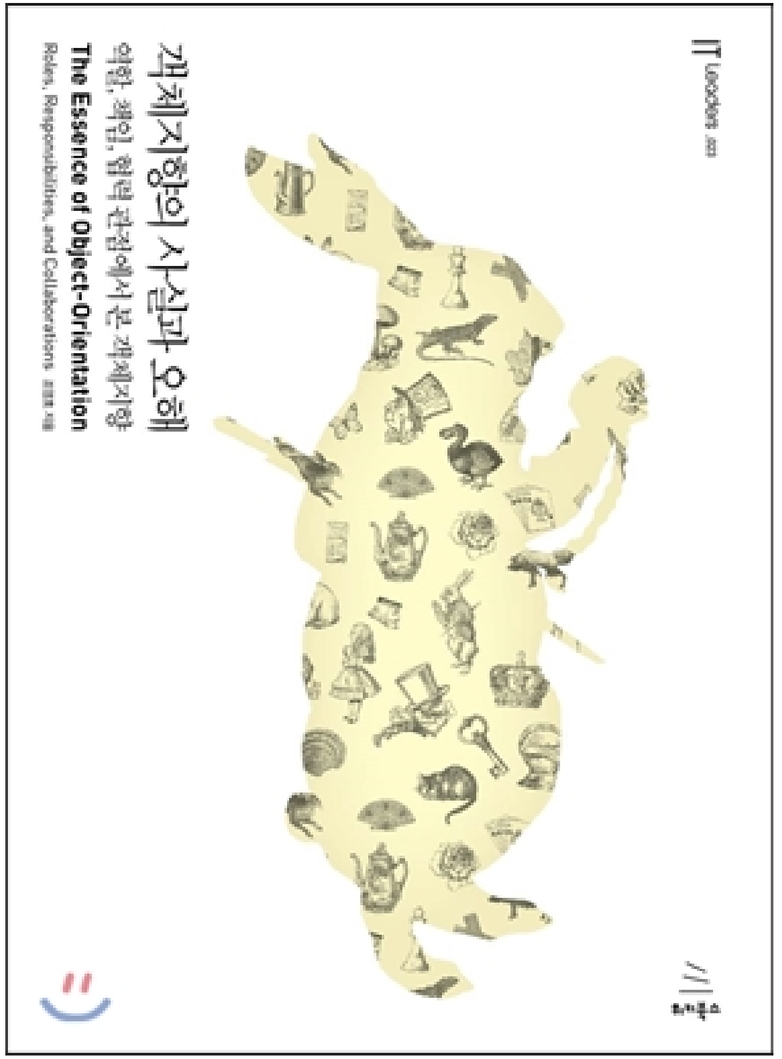
객체지향 프로그래밍이 아니더라도 전반적으로 적용될 수 있는 원칙들!

객체지향 프로그래밍 맛보기

- 객체중심적 생각
- 책임을 가진다.
- 서로 다른 책임을 가진 객체들이 서로 협력한다.
- 상상력 (이상한 나라의 앨리스)

객체지향 프로그래밍 맛보기

- 객체중심적 생각
- 책임을 가진다.
- 서로 다른 책임을 가진 객체들이 서로 협력한다.
- 상상력 (이상한 나라의 앨리스)



나쁜 설계?

나쁜 설계?

모듈 간의 결합도가 높아서 변경하기 어려움

나쁜 설계?

응집도가 낮아서 하나의 객체가 하나의 책임을 온전히 이행하지 못함

좋은 설계?

좋은 설계?

결합도가 낮고 응집도가 높은 코드

S O L I D !

Single Responsibility

단일 책임 원칙

모든 객체는 단 하나의 책임을 가져야 한다.

= 해당 객체를 변경해야 할 이유가 단 하나만 존재한다.

BAD

```python
class DataManager:
    #...
    def read(self):
        print("read data...")

    def parse(self):
        print("parsing data...")

    def save(self):
        print("save data...")

manager = DataManager()
manager.read()
manager.parse()
manager.save()
```

- DataSaver 책임: 읽기, 파싱, 저장 (해당 객체에 대한 책임 파악이 어려움 → 가독성⬇)
- 클래스 내 다른 역할을 수행하는 코드간의 의존성이 높아짐
- 코드 변경의 어려움/부작용

  💬 기존에는 csv에서 read를 하고 있었는데, json read로 수정한다면?

  💬 혹은 json read도 지원해야 한다면?

GOOD

```python
class Reader:
    def read(self):
        # reading data...
        print("read data...")

class Parser:
    def parse(self, data):
        # parsing data...
        print("parsing data...")

class Writer:
    def save(self, data):
        # saving data...
        print("save data...")

reader = Reader()
parser = Praser()
writer = Writer()

data = reader.read()
parsed_data = parser.parse(data)
writer.save(parsed_data)
```

- 각 책임별로 클래스 분리
- 결합도(의존도) ⬇

  : 코드 변경이 영향을 줄 가능성 ⬇
- 확장성 ⬆

  : 인터페이스만 통일하면, `CsvReader`, `JsonReader` 등을 만들어서 사용할 수 있음

Open Closed

개방 폐쇄 원칙

확장에는 열려있고, 변경에는 닫혀있어야 한다.

= 기존 객체의 구현을 수정하는 것이 아니라, 새로운 객체를 추가함으로써 기능을 추가한다.

BAD

```python
class ReportSender:
    def __init__(self, report):
        self.report = report

    def send(self, send_type, receiver):
        if send_type == "email":
            print("email 전송")
        elif send_type == "printer":
            print("printer")
        elif send_type == "fax":
            print("fax 전송")

sender = ReportSender("report data")
sender.send("email", "john@test.com")
sender.send("printer", "http://localhost:9100")
sender.send("fax", "012-345-6789")
```

- 새로운 기능을 추가하게 될 때, 기존 클래스의 함수 (send)를 건드리게 됨 → 다른 기능을 깨뜨릴 위험성 (결합도⬆)

  💬 만약 다른 유형의 sender가 추가 된다면? (e.g, MS팀즈)

  💬 만약 추가적인 정보가 더 필요하다면? (e.g, API키)

GOOD

```python
import abc
class BaseReportSender(abc.ABC):
    def __init__(self, report):
        self._report = report

    @abc.abstractmethod
    def send(self, to):
        pass

class EmailSender(BaseReportSender):
    def __init__(self, report):
        super().__init__(report, from_email)
        self._from = from_email

    def send(self, to):
        print(f"send email from: {self._from} to: {to}")

class MSTeamsSender(BaseReportSender):
    def __init__(self, report, api_key):
        super().__init__(report)
        self._api_key = api_key

    def send(self, to):
        print(f"send msteam to: {to} using apikey")
```

```python
def get_sender(sender_type):
    if sender_type == "email":
        return EmailSender("report data", "admin@test.com")
    elif sender_type == "msteams":
        return MSTeamsSender("report data", "key-xxxxx")
    else:
        raise ValueError("Invalid Sender type")

sender = get_sender("email")
sender.send("john@test.com")
sender = get_sender("msteams")
sender.send("jane")
```

- 추상화 클래스를 사용하여 `send` 메소드 강제 (다른 언어에서는 interface 또는 trait으로 구현)
- 해당 클래스를 상속하고, send메소드만 구현하면 기존 구현을 수정하지 않고, 다른 Sender를 쉽게 추가할 수 있음

Liskov Substitution

리스코프 치환 원칙

자식 클래스가 부모 클래스를 대체하여도 프로그램이 의도한대로 동작하여야 한다.

BAD

```
class Rectangle:
    def __init__(self):
        self._width = 0
        self._height = 0

    def set_width(self, w):
        self._width = w

    def set_height(self, h):
        self._height = h

    @property
    def area(self):
         return self._width * self._height

class Square(Rectangle):
    pass

rect = Square()
rect.set_width(4)
rect.set_height(5)
print(rect.area)  # It prints 25, instead of 20
```

- 논리적으로 정사각형은 직사각형이지만,

  상속관계를 두는 것은 적절치 않음

- 리스코프 원칙을 위반할 경우,

  Open-Closed 원칙을 위반할 확률이 높다.

  (if 구문등으로 다르게 처리해야하므로)

GOOD

```
class Shape(abc.ABC):
    @property
    @abc.abstractmethod
    def area(self):
      pass

class Rectangle(Shape):
    def __init__(self, w, h):
        self._width = w
        self._heiht = h

    @property
    def area(self):
      return self._width * self._height

class Square(Shape):
    def __init__(self, l):
        self._length = l

    def area(self):
        return self._length * self._length
```

```
rect1 = Square(4)
rect2 = Rectangle(4, 5)
print(rect1.area) # 16
print(rect2.area) # 20
```

- 정사각형-직사각형 관계보다 더 포괄적인 도형을 상속하도록 변경하고 코드를 리팩토링

Another Example

BAD

```
class Bird:
    def eat(self, food):
        print(f"I can eat {food}")

    def fly(self):
        print("Fly, Fly!!")

class Duck(Bird):
    def fly(self):
        print("I quack, quack while flying!!")

class Chicken(Bird):
    def fly(self):
        raise Exception("I cannot fly!!")


birds = [Duck(), Chicken()]
for bird in birds:
    bird.fly()
```

- Chicken Object의 경우 fly 메소드가 있어선 안된다.
- Chicken도 분명히 Bird이지만 리스코프 원칙 위배

  → is-a 관계라고 해서 모두 상속이 바람직한 것은 아님

GOOD

```python
class Eatable:
    def eat(self, food):
        print(f"I can eat {food}")

class Flyable:
    def fly(self):
        print("Fly, Fly!!")

class Duck(Eatable, Flyable):
    def fly(self):
        print("I quack, quack while flying!!")

class Chicken(Eatable):
  pass

class Plane(Flyable):
    pass

flyables = [Duck(), Plane()]
for each in flyables:
    each.fly()
```

- Mixin을 활용 (다른 언어에서는 interface 활용 가능)

  *Mixin: 클래스를 최소한의 행동(책임)으로 정의하여 상속받는 형태로 구현하는 설계방식

- 요즘에는 상속보다 Interface (Trait)등을 활용하는 쪽으로 언어가 발전하고 있다. (e.g. `golang`, `rust`)

Interface Segregaton

인터페이스 분리 원칙

클래스는 자신이 이용하지 않는 메서드에 의존해서는 안된다.

= 인터페이스가 한가지 책임을 하게 해야한다. (의존성 ⬇)

BAD

```python
class Character(abc.ABC):
    @abc.abstractmehtod
    def attack(self, other):
        print(f"I attack {other}")

    @abc.abstractmehtod
    def talk(self, other):
        print(f"I talk to {other}")

    @abc.abstractmehtod
    def move(self, x, y):
        print(f"I move to ({x}, {y})")

class Monster(Character):
    def attack(self, other):
        print(f"Monster attack {other}")

    def move(self, x, y):
        print(f"Monster move to ({x}, {y})")

class NPC(Character):
    def talk(self, other):
        print(f"NPC talk to {other}")
```

```python
monster = Monster()
monster.talk("someone") # Shouldn't be possible
npc = NPC()
npc.move(5, 10) # Shouldn't be possible
```

- 사용하지 않는 인터페이스(추상클래스)의 메소드에도 의존하고 있음 → 사이드이펙트
- 인터페이스 관점에서 SRP가 제대로 이루어지지 않고 있음

GOOD

```python
class Attackable(abc.ABC):
    def attack(self, other):
        print(f"I attack {other}")

class Talkable(abc.ABC):
    def talk(self, other):
        print(f"I talk to {other}")

class Movable(abc.ABC):
    def move(self, x, y):
        print(f"I move to ({x}, {y})")

class NPC(Talkable):
    def talk(self, other):
        print(f"NPC talk to {other}")

class Monster(Attackable, Movable):
    def attack(self, other):
        print(f"Monster attack {other}")

    def move(self, x, y):
        print(f"Monster move to ({x}, {y})")
```

```python
monster = Monster()
monster.move(10, 15)
monster.attack("john")
npc = NPC()
npc.talk("jane")
```

- 필요한 속성(역할)만 사용
- 가독성 ⬆ - 각 클래스가 어떤 역할을 하는지 쉽게 알 수 있음
- 우리가 Vue에서 사용하는 Mixin도 유사한 관점에서 바라보자

Dependcy Inversion

의존성 역전 원칙

- 상위 모듈은 하위 모듈에 의존해서는 안된다. **둘 다 추상 모듈에 의존**해야 한다.
- 추상 모듈은 구체화된 모듈에 의존해서는 안된다. **구체화된 모듈은 추상 모듈에 의존해야 한다.**

BAD

```python
class TeamsBot:
    def send_message_to_teams(self, message):
        print("send message to teams")

class SlackBot:
    def send_alert_to_slack(self, channel, message):
        print("send message to slack")

class AlertService:
    def __init__(self):
        self.teams_bot = TeamsBot()

    def alert(self, message):
        self.teams_bot.send_message_to_teams(message)

alert_service = AlertService()
alert_service.alert()
```

- 상위모듈 (`AlertService`)이 하위모듈 (`TeamsBot`) 에 의존하고 있음
- Alert를 Teams가 아닌 Slack에 보내야한다면 상위 모듈 (AlertService) alert로직이 수정되어야 함 (OCP 위반)

Another BAD

```
class MessageSender(abc.ABC):
  def send(self, message):
    if isinstance(self, TeamsBot):
        self.send_message_to_teams(message)
    elif isinstance(self, Slack):
        self.send_message_to_slack(channel, message)
    else:
        raise ValueError("Something went wrong")

class TeamsBot:
    def send_message_to_teams(self, message):
        print("send message to teams")

class SlackBot:
    def __init__(self, channel):
        self.channel = channel

    def send_alert_to_slack(self, message):
        print("send message to slack")

class AlertService:
  def __init__(self, sender):
    self.sender = sender

  def alert(message):
```

```
sender = TeamsBot()
alert_service = AlertService(sender)
alert_service.alert()
```

- 추상 모듈(`MessageSender`)이 구체화된 모듈(`TeamsBot`, `SlackBot`)에 의존하고 있음 (역시 OCP 위반)

Good

```python
class MessageSender(abc.ABC):
    @abc.abstractmethod
    def send(self, message):
        pass

class TeamsBot(MessageSender):
    def send(self, message):
        print("send message to teams")

class SlackBot(MessageSender):
  def __init__(self, channel):
      self.channel = channel

  def send(self, message):
      print("send message to slack")

class AlertService:
  def __init__(self, sender):
      self.sender = sender

  def alert(message):
      sender.send(message)
```

- 하위 모듈과 상위 모듈이 모두 추상화된 모듈

  `MessageSender`에 의존하게 함으로써 다른 모듈로 변경이 자유로움

SOLID는 이상향이다.

SOLID는 이상향이다.

항상 지킬 수 있는 것은 아니다. (시간의 제약, 코드의 복잡도⬆)

SOLID는 이상향이다.

항상 지킬 수 있는 것은 아니다. (시간의 제약, 코드의 복잡도⬆)

그래도 우리가 항상 바라봐야하는 지향점이다.

코드리뷰에 SOLID를 근거로 잘 활용합시다.

무조건적으로 지켜야하는 것은 아니지만 지키지 않는다면 충분한 근거가 있어야 한다.

감사합니다.