개발자라면, SOLID 원칙

좋은 설계를 위한 5가지 프로그래밍 원칙

© sjquant

SOLID 원칙이란?

SOLID 원칙이란?

객체지향 프로그래밍(OOP) 설계의 5가지 '이상향'

SOLID 원칙이란?

객체지향 프로그래밍(OOP) 설계의 5가지 '이상향'

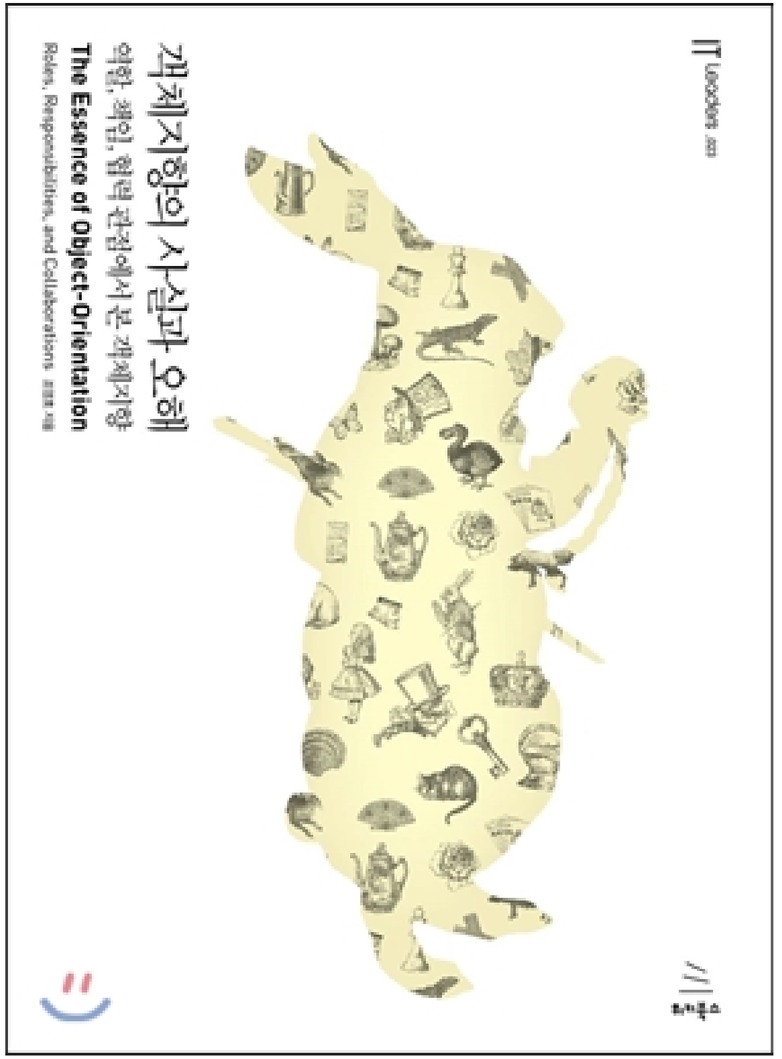
객체지향 프로그래밍이 아니더라도 전반적으로 적용될 수 있는 원칙들!

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 바꿔야 할 이유가 단 하나만 존재한다.

BAD

```
class FinancialReport:

  # ...

  def report(self):
    return {
      "date": self.date,
      "title": self.title,
      "content": self.content
    }

  def send_report(self, email):
    report = self.report
    # 이메일 전송
```

- FinancialReport의 책임이 두 개: report와 send_report
- 만약, MarketingReport가 생긴다면? → 같은 함수를 두번 구현 (중복 발생)
- 만약, Email이 아닌 다른 수단으로 send_report를 한다면? → 수정해야할 이유가 한가지 이상

GOOD

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

  def send(self, email):
    data = self.report.report()
    # 이메일 전송

class Report(abc.ABC):

  @abc.abstractmethod
  def report(self):
    pass

class FinancialReport(Report):
  # ...

# ...
report_sender = ReportSender(marketing_report)
report_sender.send("somebody@help.me")
```

Open Closed

개방 폐쇄 원칙

확장에는 열려있고, 변경에는 닫혀있어야 한다.

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

BAD

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

  def send(self, send_type, receiver):
    if send_type == "email":
      # 이메일 전송
    if send_type == "printer":
      # 프린터 출력
    if send_type == "fax":
      # 팩스 출력
```

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

  → send 함수의 수정이 필요함

  → 기존에 잘 동작하던 함수를 깨뜨릴 위험성

GOOD

```
class ReportSender(abc.ABC):

  @abc.abstractmethod
  def send(self):
    pass

class EmailReportSender(ReportSender):

  def send(self):
    # 이메일 전송

class MSTeamsReportSender(ReportSener):

  def send(self):
    # MSTeams 알림 전송
```

새로운 Sender가 추가된다면, ReportSender를 구현한 새로운 클래스만 추가해주면 된다!

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!!")
```

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

  → is-a 관계라고 해서 모두 상속이 바람직한 것은 아님

GOOD

```
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
```

- 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("I attack {other}")

  @abc.abstractmehtod
  def talk(self, other):
    print("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("Monster attack {other}")

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

  # monster.talk 도 호출 가능 (가능해선 안됨)
```

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

  # npc.attack 이나 npc.move 도 호출 가능 (가능해선 안됨)
```

- 사용하지 않는 인터페이스(추상클래스)의 메소드에도 의존

GOOD

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

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

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

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

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

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

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

```
class TeamsBot:
  def send_message_to_teams(self, message):
    # send mesage logic is here

class SlackBot:
  def send_alert_to_slack(self, channel, message):
    # send mesage logic is here

class AlertService:

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

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

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

Good

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

class TeamsBot(MessageSender):
  def send(self, message):
    # send message logic

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

  def send(self, message):
    # send message logic

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 모듈로서 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.

감사합니다.