SOLID Principals

소프트웨어융합학부

노기섭 교수

(kafa46@cju.ac.kr)

Overview

객체지향 설계?

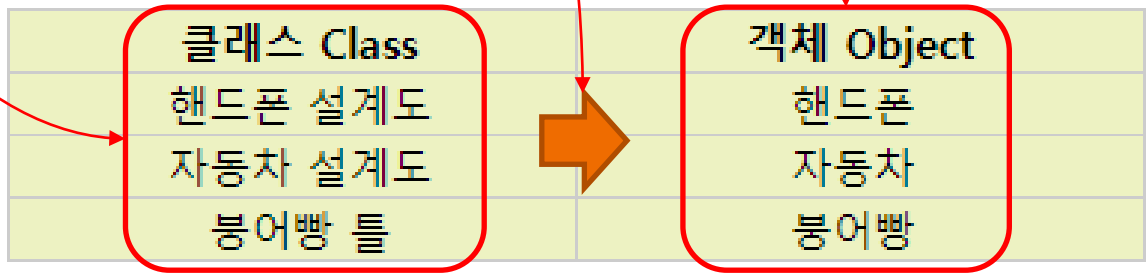
■ 객체지향 설계에 등장하는 개념들

클래스 (class): 공통되는 것들을 묶어서 대표적인 이름을 붙인 것 (추상화 결과)

인스턴스 (instance): 클래스가 메모리 공간에 할당된 실체

객체 (object)

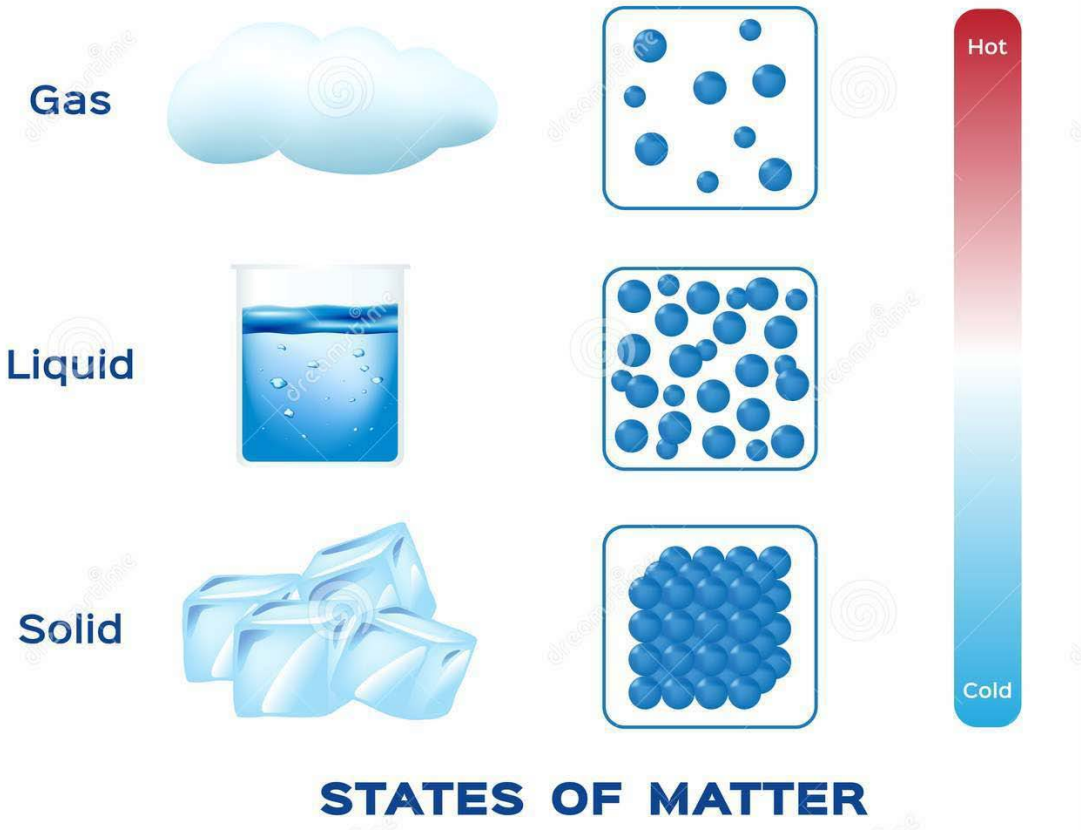
- 명확한 의미를 담고 있는 대상 (설계자 관점)
- 클래스에서 생성된 변수 (개발자 관점)
- 유일한 식별자, 상태(state) 존재, 연산 가능한 메서드(method)



SOLID??

■ Solid?

- In physics

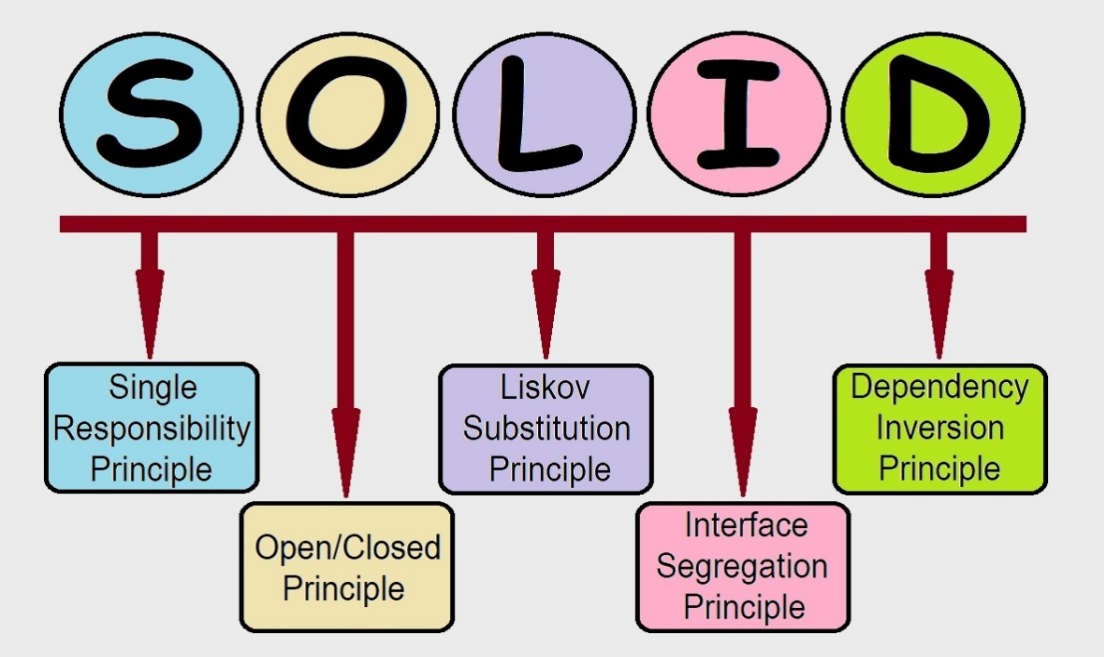


이미지 출처: https://www.dreamstime.com/states-matter-solid-liquid-gas-vector-white-background-states-matter-solid-liquid-gas-vector-image110701771

SOLID??

■ Solid?

- In SW Engineering
  - 객체지향프로그래밍(OOP: Object Oriented Programming) 대표적 원칙



이미지 출처: https://abhinavranaweb.wordpress.com/2019/01/31/solid-design-principles/

Concepts in SOLID

① 단일 책임 원칙(SRP: Single-Responsibility Principle)

클래스는 하나의 책임만 가진다.

② 개방 폐쇄의 원칙(OCP: Open-Closed Principle)

확장(상속)에는 열려 있어야 하고 변경에는 닫혀 있어야 한다.

③ 리스코프 교체의 원칙(LSP: Liskov Substitution Principle)

기반 클래스는 파생 클래스로 대체할 수 있어야 한다.

④ 인터페이스 분리의 원칙(ISP: Interface Segregation Principle)

하나의 일반적인 인터페이스보다는 구체적인 여러 개의 인터페이스가 낫다.

⑤ 의존 관계 역전의 원칙(DIP: Dependency Inversion Principle)

클라이언트는 구체 클래스가 아닌 추상 클래스(인터페이스)에 의존해야 한다.

Tackling SOLID Principals (Step by Step Learning)

SOLID #1. SRP - 단일 책임 원칙
(Single Responsibility Principal)

참고자료:
Title: Uncle Bob's SOLID principles made easy ❁ - in Python!
Author: Arjan
URL: https://youtu.be/pTB30aXS77U

Principal #1. SRP 이론

■ 단일 책임 원칙 (SRP: Single Responsibility Principal)

Robert Martin (1952 ~) - “Uncle Bob” 으로 불림

- ‘Agile SW Development’ 책에서
- ‘객체지향 설계’ 부분에서 응집도(cohesion) 원칙을 설명하면서 SRP 개념 소개(2003)

이미지 출처: https://en.wikipedia.org/wiki/Robert_C._Martin

- 클래스는 하나의 책임만 가진다.
- 클래스가 제공하는 모든 서비스(methods)는 그 책임을 수행하는데 집중한다.
- 환경이 바뀌어서 클래스를 변경해야 하는 이유는 오직 하나 뿐이어야 한다.
  - 환경 변화로 하나의 클래스가 여러 책임을 갖는 경우 → 클래스 분할

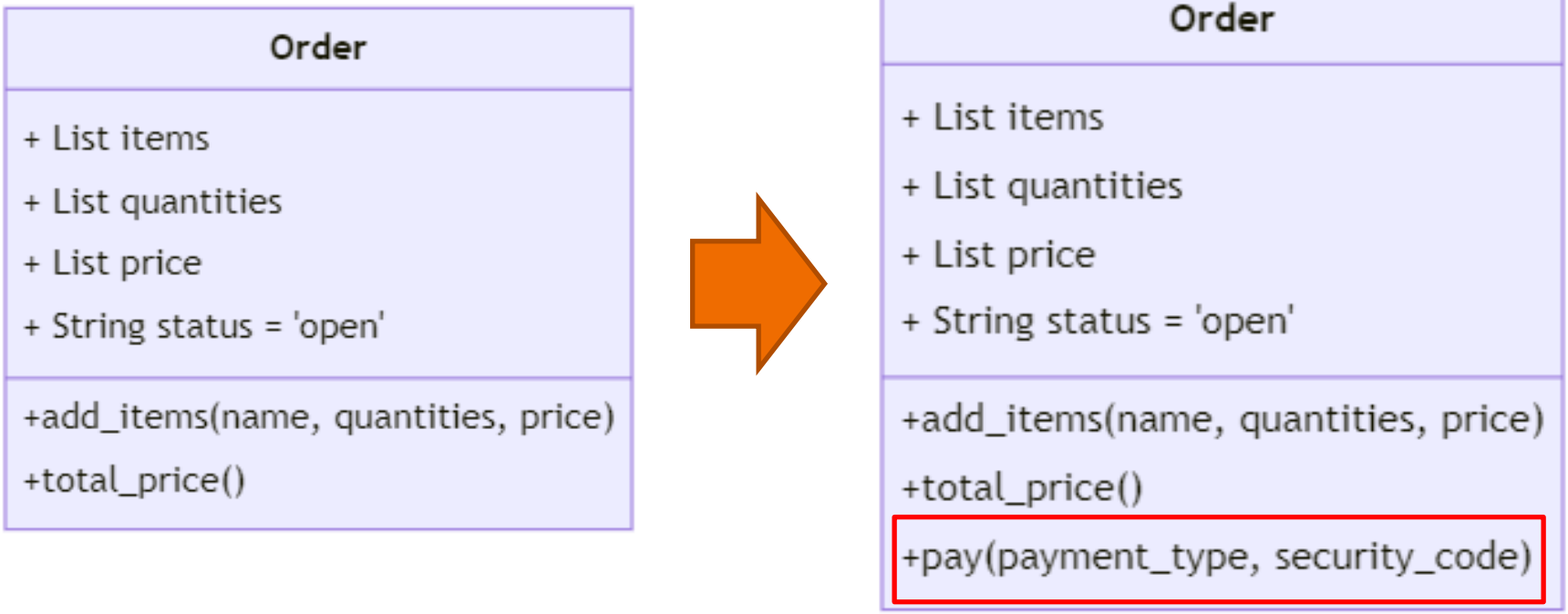
Principal #1. SRP 이론

■ 단일 책임 원칙 (SRP: Single Responsibility Principal)

- 비교적 단순한 원칙입니다.
- 하지만 복잡한 프로세스를 구현하거나 경험이 부족할 경우 지키기 어려움
- 대부분의 SW 위협 원인이 SRP 미준수에서 비롯되는 경우가 많습니다.
- SRP 장점
  - 클래스 책임 영역 확실 → 하나의 책임 변경에 따른 연쇄 변경에서 Free!
  - 응집도(cohesion) 강화, 결합도(coupling) 약화, 가독성 향상, 유지보수 용이
- SRP 준수 전략
  - 중복된 책임은 추상 클래스로 구현
  - 기존의 클래스로 해결할 수 없다면 새로운 클래스 구현

Principal #1. SRP 사례 연구 (Case Study)

■ 환경 변화에 따른 클래스 책임 증가: 단일 책임 → 2개 책임



주문 (order) 클래스
- 주문 아이템을 추가 기능
- 전체 주문 가격 확인 기능

환경 변화

주문 (order) 클래스 기능 추가
- 주문 상품에 대한 금액 지급

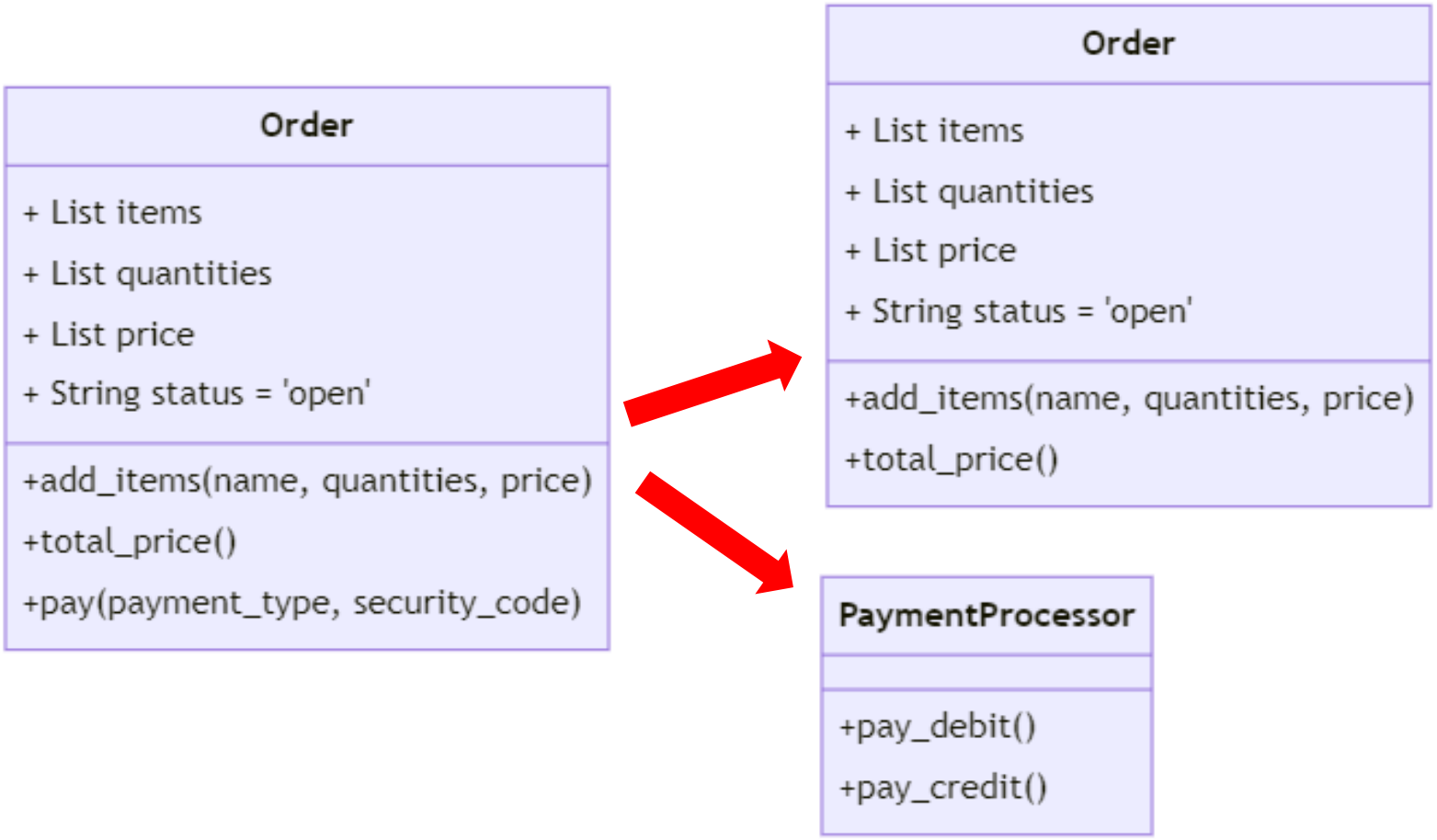
단일책임 원칙 위배

Python 으로 'Order' 클래스를 구현하면?

```python
class Order:

    def __init__(self):
        self.items = []
        self.quantities = []
        self.prices = []
        self.status = "open"

    def add_item(self, name, quantity, price):
        self.items.append(name)
        self.quantities.append(quantity)
        self.prices.append(price)

    def total_price(self):
        total = 0
        for i in range(len(self.prices)):
            total += self.quantities[i] * self.prices[i]
        return total

    def pay(self, payment_type, security_code):
        if payment_type == "debit":
            print("Processing debit payment type")
            print(f"Verifying security code: {security_code}")
            self.status = "paid"
        elif payment_type == "credit":
            print("Processing credit payment type")
            print(f"Verifying security code: {security_code}")
            self.status = "paid"
        else:
            raise Exception(f"Unknown payment type: {payment_type}")


if __name__=='__main__':
    order = Order()
    order.add_item("Keyboard", 1, 50)
    order.add_item("SSD", 1, 150)
    order.add_item("USB cable", 2, 5)

    print(order.total_price())
    order.pay("debit", "0372846")
```

Principal #1. SRP 사례 연구 (Case Study)

■ Order 클래스를 2개로 분할



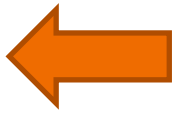
Principal #1. SRP 사례 연구 (Case Study)

■ Order 클래스

- Order와 관련 없는 책임(pay)을 추가로 갖게 되었습니다.
- pay( ) 메서드 내부에서도 다양한 기능을 구현하였습니다.
  - 직불 (debit) 카드
  - 신용 (credit) 카드
- 해결 방법
  - 별도의 클래스로 책임 분리
    - ‘PaymentProcessor’ 클래스 구현
    - ‘Order’ 클래스 수정

Principal #1. SRP 사례 연구 (Case Study)

```python
class Order:
    def __init__(self):
        self.items = []
        self.quantities = []
        self.prices = []
        self.status = "open"

    def add_item(self, name, quantity, price):
        self.items.append(name)
        self.quantities.append(quantity)
        self.prices.append(price)

    def total_price(self):
        total = 0
        for i in range(len(self.prices)):
            total += self.quantities[i] * self.prices[i]
        return total
```



Order 클래스에서
pay( ) 메서드 제거

pay( ) 기능만 책임지는
PaymentProcessor 클래스 구현

```python
class PaymentProcessor:
    def pay_debit(self, order, security_code):
        print("Processing debit payment type")
        print(f"Verifying security code: {security_code}")
        order.status = "paid"

    def pay_credit(self, order, security_code):
        print("Processing credit payment type")
        print(f"Verifying security code: {security_code}")
        order.status = "paid"
```

```python
if __name__=='__main__':
    # Order 객체 생성 및 실행
    order = Order()
    order.add_item("Keyboard", 1, 50)
    order.add_item("SSD", 1, 150)
    order.add_item("USB cable", 2, 5)
    print(order.total_price())

    # PaymentProcess 객체 생성 및 실행
    processor = PaymentProcessor()
    processor.pay_debit(order, "0372846")
```

실행 코드

SOLID #2. OCP - 개방 폐쇄 원칙 (Open Close Principal)

참고자료:

Title: Uncle Bob's SOLID principles made easy ❀ - in Python!

Author: Arjan

URL: https://youtu.be/pTB30aXS77U

Principal #2. OCP 이론

■ 개방 폐쇄 원칙 (Open Close Principal)

Bertrand Meyer (1950 ~), France

- 'Object Oriented SW Construction (1998)' 책에서 정의한 내용
- SW 구성요소는 확장에는 열려 있고, 변경에는 닫혀 있어야 한다는 원칙 소개(2003)

이미지 출처: https://en.wikipedia.org/wiki/Bertrand_Meyer

- 변경을 위한 비용은 최소화, 확장을 위한 비용은 극대화
  - 요구사항의 변경이 발생하더라도 기존 요소의 수정을 발행하지 않아야 한다.
  - 기존 요소를 쉽게 재활용해서 쉽게 확장할 수 있어야 한다.
- 추상화/다형성이 핵심

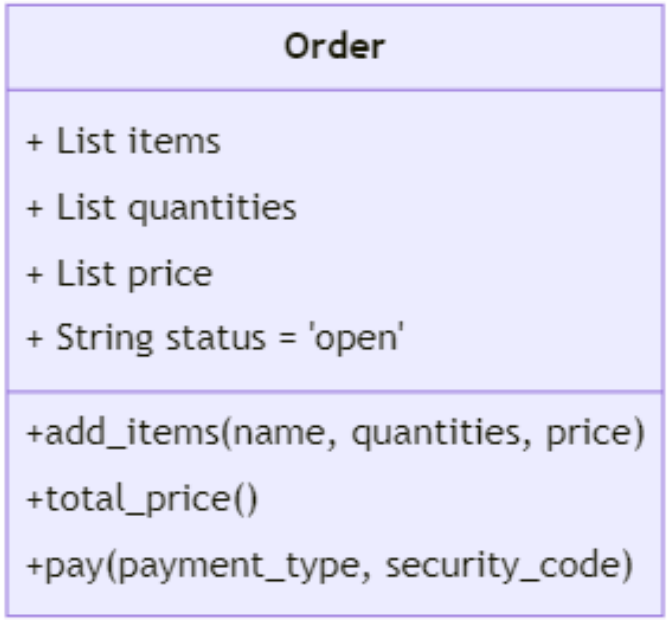
Principal #2. OCP 이론

■ 개방 폐쇄 원칙 (Open Close Principal)

- 객체지향 설계의 핵심 메커니즘
- 변화로부터 시작된 삽질
  - 변화를 막을 수 있는 사람은 없다.
  - 그러나 개발자를 괴롭힐 변화를 그대로 수용하기 보다는,
  - 적절히 대응할 전략이 필요하다.
- 접근 방법
  - 변하지 않는 것과 변하게 될 것을 모듈로 구분
  - 이 모듈이 만나는 지점에 인터페이스 정의
  - 인터페이스에서 정의한 대로 구현(코딩)

Principal #2. OCP 사례 연구 (Case Study)

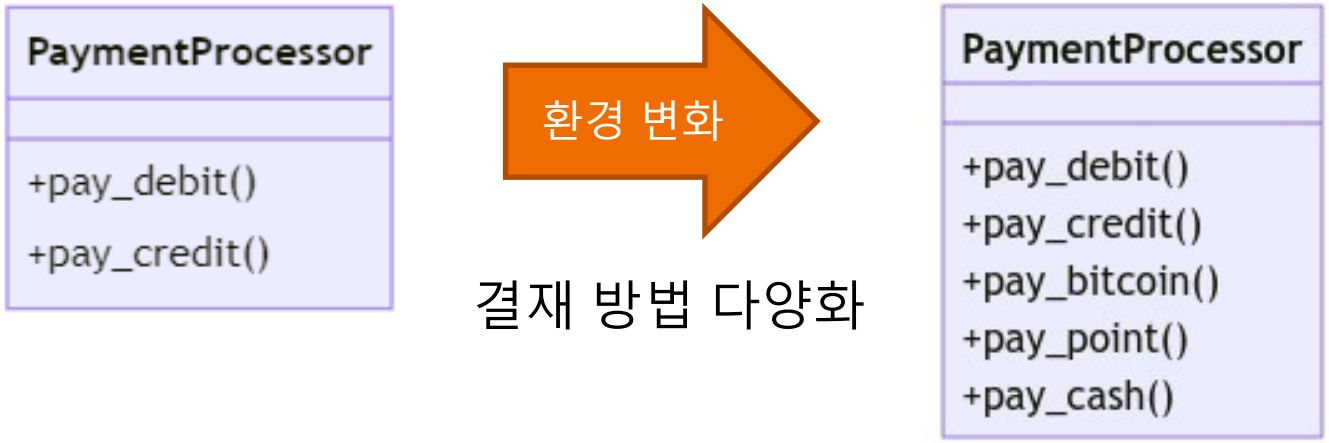
■ 단일 책임 원칙 (SRP)에서 살펴본 예제를 다시 확인해 봅니다.



환경 변화에 따른
문제점 예상(다음 슬라이드)

Principal #2. OCP 사례 연구 (Case Study)

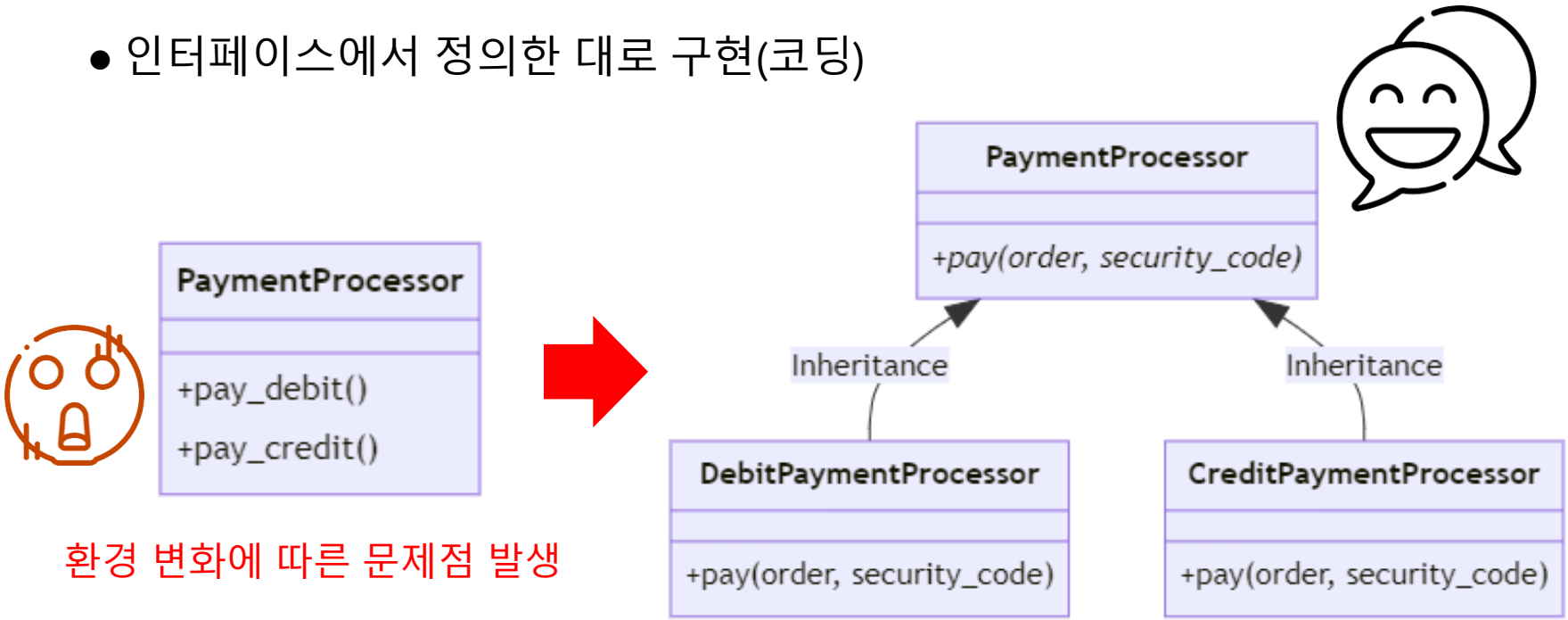
■ 단일 책임 원칙 (SRP)에서 살펴본 예제를 다시 확인해 봅니다.



Principal #2. OCP 사례 연구 (Case Study)

■ 해결 방법

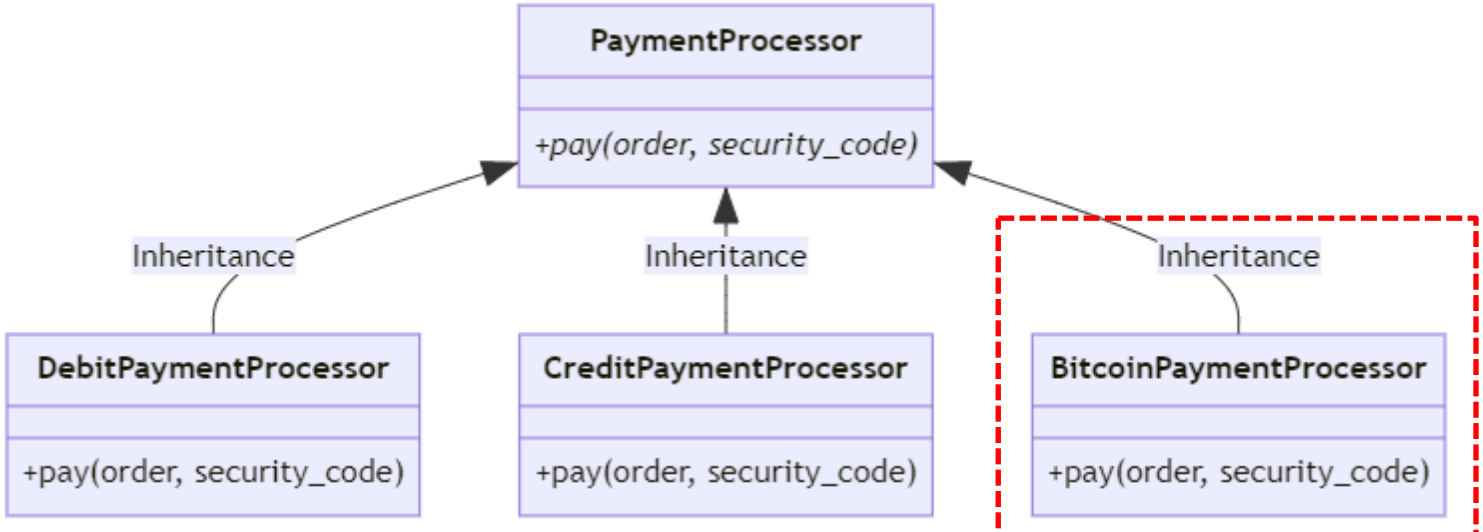
- 변하지 않는 것과 변하게 될 것을 모듈로 구분
- 이 모듈이 만나는 지점에 인터페이스 정의
- 인터페이스에서 정의한 대로 구현(코딩)



Principal #2. OCP 사례 연구 (Case Study)

■ OCP 장점

- 미래에 어떤 환경 변화가 오더라도 기존 코드는 그대로 사용
- 모듈 내부 응집도(Cohesion)↑, 모듈 간 결합도(Coupling) ↓



환경 변화에 따른 기능 추가: 어떤 코드의 변경도 없이 손쉽게 추가

Principal #2. OCP 사례 연구 (Case Study)

OCP를 준수한 코딩

기존 코드

```python
class PaymentProcessor:
    '''주문 결재 처리기:
    Odder 클래스에서 책임(pay)을
    분리하기 구현한 클래스
    '''

    def pay_debit(self, order, security_code):
        '''직불 카드 결재'''
        print("직불카드 결재를 시작합니다.")
        print(f"비밀번호 확인: {security_code}")
        print('결재가 완료되었습니다.')
        order.status = "paid"

    def pay_credit(self, order, security_code):
        '''신용 카드 결재'''
        print("신용카드 결재를 시작합니다.")
        print(f"비밀번호 확인: {security_code}")
        print('결재가 완료되었습니다.')
        order.status = "paid“
```

```python
class PaymentProcessor(ABC):
    '''OCP 원칙 준수를 위한 추상 클래스'''

    @abstractmethod
    def pay(self, order, security_code):
        pass


class DebitPaymentProcessor(PaymentProcessor):
    '''직불 카드를 이용한 주문 결재 처리기'''

    def pay(self, order, security_code):
        '''직불 카드 결재'''
        print("직불카드 결재를 시작합니다.")
        print(f"비밀번호 확인: {security_code}")
        print('결재가 완료되었습니다.')
        order.status = "paid"

class CreditPaymentProcessor(PaymentProcessor):
    '''신용 카드를 이용한 주문 결재 처리기'''

    def pay(self, order, security_code):
        '''신용 카드 결재'''
        print("신용카드 결재를 시작합니다.")
        print(f"비밀번호 확인: {security_code}")
        print('결재가 완료되었습니다.')
        order.status = "paid“
```

SOLID #3. LSP - 리스코프 교환 원칙 (Liskov Substitution Principal)

Principal #3. LSP 이론

■ 리스코프 교체 원칙 (LSP: Liskov Substitution Principal)

Barbara Liskov (1939 ~), MIT 교수, USA

- 프로그래밍 언어, 분산 컴퓨팅의 선구자
- 데이터 추상화의 본질을 구성하는 리스코프 치환 원칙 개발
- 리스코프 치환 원칙 개발 공로로 튜링상 수상(2008)

이미지 출처: https://en.wikipedia.org/wiki/Barbara_Liskov

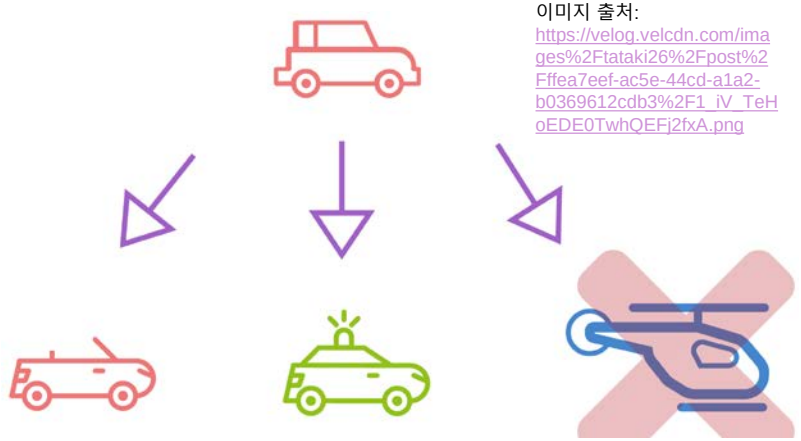
뭔 말여?

이미지 출처: https://icon-icons.com/ko/ 아이콘/엉뚱-얼굴-이모티콘

- SOLID 5개 원칙 중에서 가장 이해하기 어려운 원칙입니다. ㅠㅠ
  - 서브 타입은 언제나 기반 타입으로 교체 가능해야 한다.
  - 서브 타입은 언제나 기반 타입과 호환될 수 있어야 한다.
- 다시 말하면, “기반 클래스는 파생 클래스로 대체할 수 있어야 한다.”

Principal #3. LSP 이론

■ 조금 더 쉬운 예제



이미지 출처:
https://velog.velcdn.com/images%2Ftataki26%2Fpost%2Ffffea7eef-ac5e-44cd-a1a2-b0369612cdb3%2F1_iV_TeHoEDE0TwhQEFj2fxA.png

- 자동차 클래스를 구현

  → 상속받아 다양한 종류의 자동차 구현

  → 헬리콥터를 구현했다면? IS-A 관계 성립 안됨

- 부모 클래스 인스턴스 자리에 자식 클래스의 인스턴스가 들어가도 작동해야 한다.
  - 자식이 부모 자리에서 작동하려면 부모와 동일하게 해동해야 함
  - 어려운 말로 표현하면, “자식은 부모의 행동 규약을 준수해야 한다”
  - 부모 클래스의 속성과 메서드를 그대로 물려받으면 아무 문제 없다.

- 문제가 발생하는 경우 ➔ 대부분은 오버라이딩(overriding) 과정에서 발생
  - 오버라이딩 과정에서 변수타입 변경, 메서드의 파라미터나 리턴값 변경
  - 부모의 의도와 다르게 메서드를 변경하는 오버라이딩

Principal #3. LSP 이론

부모와 다른
자식이 생성

자동차를 상속받아서
비행기 클래스를 만드는 상황

IS-A 관계
성립 안됨

상속에는 닫혀 있고
개방에는 열려 있어야 하는
기본 원칙을 위반

객체지향 설계의
취지 위반

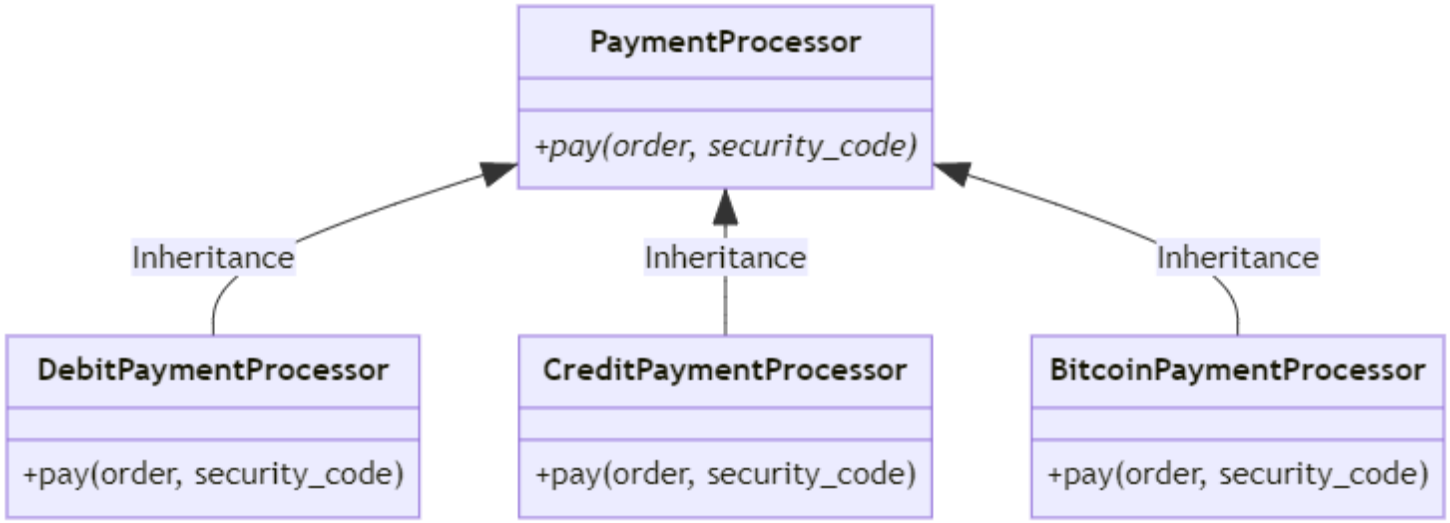
Principal #3. LSP 이론

■ 해결 방법

- 추상클래스, 인터페이스를 활용
- 두 객체가 하는 일에 무언가를 추가해야 한다면 상속을 활용
- 하는 일에 따라
  - 하는 일이 같으면 하나의 클래스로 구성하고 구분 필드를 추가
  - 하는 일이 다르면 별개의 클래스로 구성

Principal #3. LSP 사례 연구 (Case Study)

■ 사례로 살펴보는 LSP 위반 사례



KakaopayPayment Processor

환경 변화: 카카오 머니를 이용해 결재하려는 사용자가 증가하였다.

PaymentProcessor를 상속받아 KakaopayPaymentProcessor 클래스를 만들자!!

카카오 페이는 이메일 필요

→ **손쉬운 해결책: security_code를 오버라이딩 하여 이메일을 사용한다.**

→ **➔ LSP 위반 (다음 슬라이드에서 좀 더 구체적으로 살펴봅니다)**

Principal #3. LSP 사례 연구 (Case Study)

■ 환경 변화 발생

- 카카오 머니를 이용한 결재 기능 필요
- PaymentProcessor를 상속받아 구현

```python
class PaymentProcessor(ABC):
    '''OCP 원칙 준수를 위한 추상 클래스'''

    @abstractmethod
    def pay(self, order, security_code):
        pass
```

```python
class KakaopayPaymentProcessor(PaymentProcessor):
    '''카카오 머니를 이용한 주문 결재 처리기'''

    def pay(self, order, email):
        '''신용 카드 결재'''
        print("카카오 머니 결재를 시작합니다.")
        print(f"이메일 확인: {email}")
        print('결재가 완료되었습니다.')
        order.status = "paid"
```

- 카카오 머니는 이메일 인증
- security_code를 email로 오버라이딩하고, pay 메서드 내부를 수정
- 추상 클래스가 의도했던 `security_code` 대신에 다른 파라미터(정보)를 이용하여 작동 ➔ LSP 위반

Principal #3. LSP 사례 연구 (Case Study)

■ **앞서 살펴봤듯이 LSP를 위반한 경우 대처 방법은 다음과 같습니다.**

1. 추상클래스 또는 인터페이스를 활용한다.
2. 두 객체가 하는 일이 다르다면 상속을 활용한다.
3. 객체가 하는 일에 따라,
    - 하는 일이 같으면: 하나의 클래스로 구성하고 구분 필드를 추가한다.
    - 하는 일이 다르면: 별개의 클래스로 구현한다.

■ **해결 방법**

- 이미 추상클래스를 이용해 구현한 상태이니 "1. 추상클래스 또는 인터페이스를 활용한다."는
    → 신경 쓰지 않아도 됩니다.
- `KakaopayPaymentProcessor` 클래스는 기존 자식 클래스들과는 인증 과정이 약간 다름
    → 상속을 통해 구현
- `KakaopayPaymentProcessor` 객체가 수행할 일은 기존 자식 클래스들이 하는 일과 동일하게 `결재`를
    → 클래스 초기화 함수 `inintializer`(`__init__`)를 이용

Principal #3. LSP 사례 연구 (Case Study)

```python
class PaymentProcessor(ABC):
    '''OCP 원칙 준수를 위한 추상 클래스'''

    # pay 메서드 파라미터 security_code 제거
    @abstractmethod
    def pay(self, order):
        pass
```

```python
class DebitPaymentProcessor(PaymentProcessor):
    '''직불 카드를 이용한 주문 결재 처리기'''

    def __init__(self, security_code):
        self.security_code = security_code

    def pay(self, order):
        '''직불 카드 결재'''
        print("직불카드 결재를 시작합니다.")
        print(f"비밀번호 확인: {self.security_code}")
        print('결재가 완료되었습니다.')
        order.status = "paid"
```

...

```python
class KakaopayPaymentProcessor(PaymentProcessor):
    '''카카오 머니를 이용한 주문 결재 처리기'''

    def __init__(self, email):
        self.email = email

    def pay(self, order):
        '''신용 카드 결재'''
        print("카카오 머니 결재를 시작합니다.")
        print(f"이메일 확인: {self.email}")
        print('결재가 완료되었습니다.')
        order.status = "paid"
```

```python
if __name__=='__main__':
    # Order 객체 생성 및 실행
    order = Order()
    order.add_item("Keyboard", 1, 50)
    order.add_item("SSD", 1, 150)
    order.add_item("USB cable", 2, 5)
    print(f'결재 금액은 {order.total_price()} 입니다.')

    # KakaopayPaymentProcess 객체 생성 및 실행
    processor = KakaopayPaymentProcessor("abc@company.com")
    processor.pay(order)
```

SOLID #4. ISP - 인터페이스 분리 원칙 (Interface Segregation Principal)

참고자료:

Title: Uncle Bob's SOLID principles made easy ❀ - in Python!

Author: Arjan

URL: https://youtu.be/pTB30aXS77U

Principal #4. ISP 이론

■ 인터페이스 분리 원칙 (ISP: Interface Segregation Principal)

Robert Martin (1952 ~) - “Uncle Bob” 으로 불림

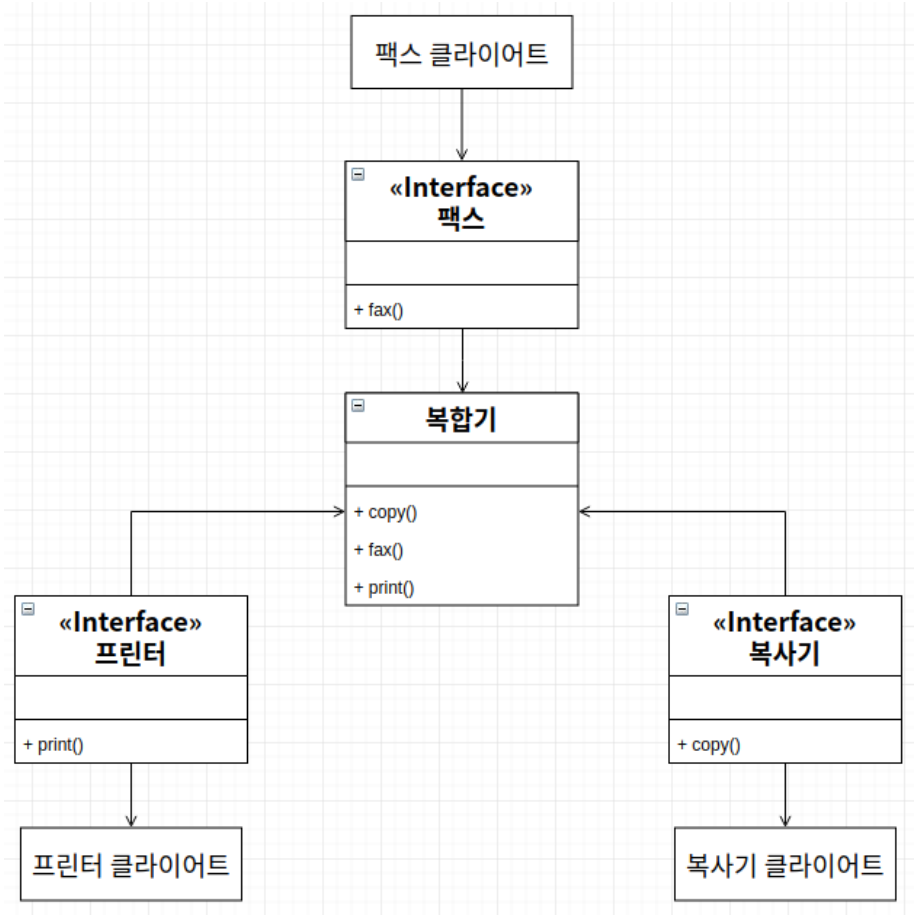
- SRP 개념 소개한 분,
- Xerox사 컨설팅 과정에서 제시한 개념
- “사용자는 자신이 필요한 인터페이스만 사용”

이미지 출처: https://en.wikipedia.org/wiki/Robert_C._Martin

- 클래스는 자신이 사용하지 않는 인터페이스는 구현하지 말아야 한다.
- 어떤 클래스가 다른 클래스에 종속될 경우에는 최소한의 인터페이스만 사용
- 인터페이스를 다수의 작은 단위로 구분하고, 사용자는 자신이 필요한 인터페이스만 사용하도록 구현

Principal #4. ISP 이론

■ **Xerox:** https://www.xerox.com/en-us



이미지 출처: https://walbatrossw.github.io/oop/2018/07/27/07-solid-isp.html

이미지 출처:
https://en.wikipedia.org/wiki/Robert_C._Martin

Principal #4. ISP 사례 연구 (Case Study)

■ **보안 강화를 위해 SMS 인증을 추가한 Two-factor Auth 기능이 추가된 상황**

- `PaymentProcessor` 클래스에 추상 메서드를 추가로 구현

```python
class PaymentProcessor(ABC):

    @abstractmethod
    def auth_sms(self, code):
        pass

    @abstractmethod
    def pay(self, order):
        pass
```

- 추상 클래스를 상속한 클래스에서 필요한 인터페이스를 구현
  - 다음 슬라이드 참고

Principal #4. ISP 사례 연구 (Case Study)

■ 추상 클래스를 상속받아 3개의 지불 방법을 구현

```python
class DebitPaymentProcessor(PaymentProcessor):

    def __init__(self, security_code):
        self.security_code = security_code
        self.verified = False

    def auth_sms(self, code):
        print(f"Verifying SMS code {code}")
        self.verified = True

    def pay(self, order):
        if not self.verified:
            raise Exception("Not authorized")
        print("Processing debit payment type")
        print(f"Verifying security code: {self.security_code}")
        order.status = "paid"
```

```python
class PaypalPaymentProcessor(PaymentProcessor):

    def __init__(self, email_address):
        self.email_address = email_address
        self.verified = False

    def auth_sms(self, code):
        print(f"Verifying SMS code {code}")
        self.verified = True

    def pay(self, order):
        if not self.verified:
            raise Exception("Not authorized")
        print("Processing paypal payment type")
        print(f"Using email address: {self.email_address}")
        order.status = "paid"
```

직불카드, 페이팔 결재 시 SMS 인증이 안되면 에러 발생하도록 구현

```python
class CreditPaymentProcessor(PaymentProcessor):

    def __init__(self, security_code):
        self.security_code = security_code

    def auth_sms(self, code):
        raise Exception("Credit card payments don't support SMS code authorization.")

    def pay(self, order):
        print("Processing credit payment type")
        print(f"Verifying security code: {self.security_code}")
        order.status = "paid"
```

신용카드 결재는 SMS 인증을 지원하지 않는 상황

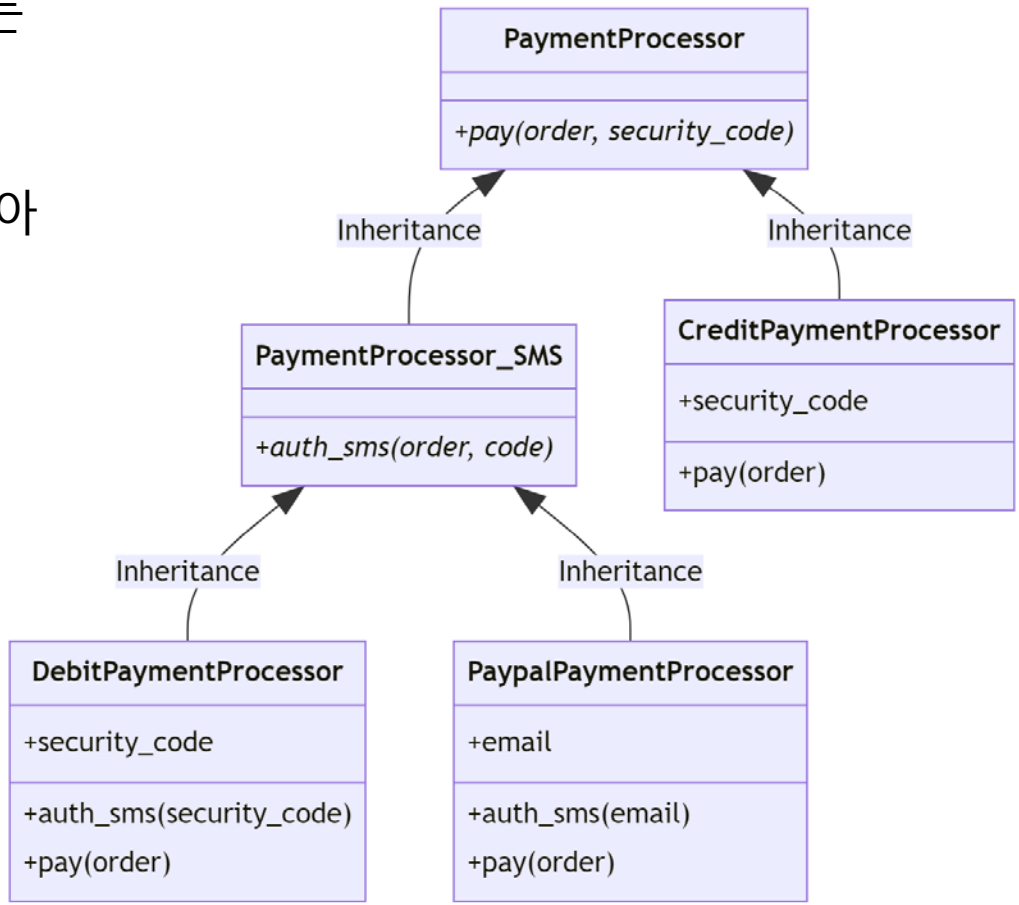
- 모든 클래스가 적절하게 인터페이스를 사용할 수 없음 (ISP 위반)

새로운 인터페이스 정의가 필요한 상황

Principal #4. ISP 사례 연구 (Case Study)

■ 해결 방법 1. Inheritance (상속)을 이용한 해결

- 기존 `PaymentProcessor`에서는 pay 기능만 구현하도록 수정
- `PaymentProcessor`를 상속받아 SMS 인증을 지원하는 새로운 `PaymentProcessor_SMS` 클래스 구현
- 결재기능을 구현할 때 상황에 맞게 필요한 클래스를 상속 받아 구현



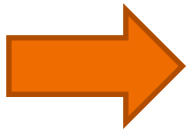
Principal #4. ISP 사례 연구 (Case Study)

해결 방법 1-1. Inheritance (상속)을 이용한 해결

- PaymentProcessor 수정

```python
class PaymentProcessor(ABC):

    @abstractmethod
    def auth_sms(self, code):
        pass

    @abstractmethod
    def pay(self, order):
        pass
```

SMS 인증을 지원하는
auth_sms( ) 메서드 삭제



```python
class PaymentProcessor(ABC):

    @abstractmethod
    def pay(self, order):
        pass
```

상속

- PaymentProcessor_SMS 클래스 생성(구현)

```python
class PaymentProcessor_SMS( PaymentProcessor ):

    @abstractmethod
    def auth_sms(self, sms_code):
        pass
```

Principal #4. ISP 사례 연구 (Case Study)

■ 해결 방법 1-2. SMS 인증이 필요한 클래스 구현(PaymentProcessor_SMS 상속)

```python
class PaymentProcessor_SMS(PaymentProcessor):

    @abstractmethod
    def auth_sms(self, sms_code):
        pass
```

상속

상속

```python
class DebitPaymentProcessor( PaymentProcessor_SMS ):

    def __init__(self, security_code):
        self.security_code = security_code
        self.verified = False

    def auth_sms(self, code):
        print(f"Verifying SMS code: {code}")
        self.verified = True

    def pay(self, order):
        if not self.verified:
            raise Exception("Not authorized")
        print("Processing debit payment type")
        print(f"Verifying security code: {self.security_code}")
        order.status = "paid"
```

```python
class PaypalPaymentProcessor( PaymentProcessor_SMS ):

    def __init__(self, email_address):
        self.email_address = email_address
        self.verified = False

    def auth_sms(self, code):
        print(f"Verifying SMS code: {code}")
        self.verified = True

    def pay(self, order):
        if not self.verified:
            raise Exception("Not authorized")
        print("Processing paypal payment type")
        print(f"Using email address: {self.email_address}")
        order.status = "paid"
```

Principal #4. ISP 사례 연구 (Case Study)

해결 방법 1-3. SMS 인증이 불필요한 클래스 구현(PaymentProcessor 상속)

```python
class PaymentProcessor(ABC):

    @abstractmethod
    def pay(self, order):
        pass
```

상속

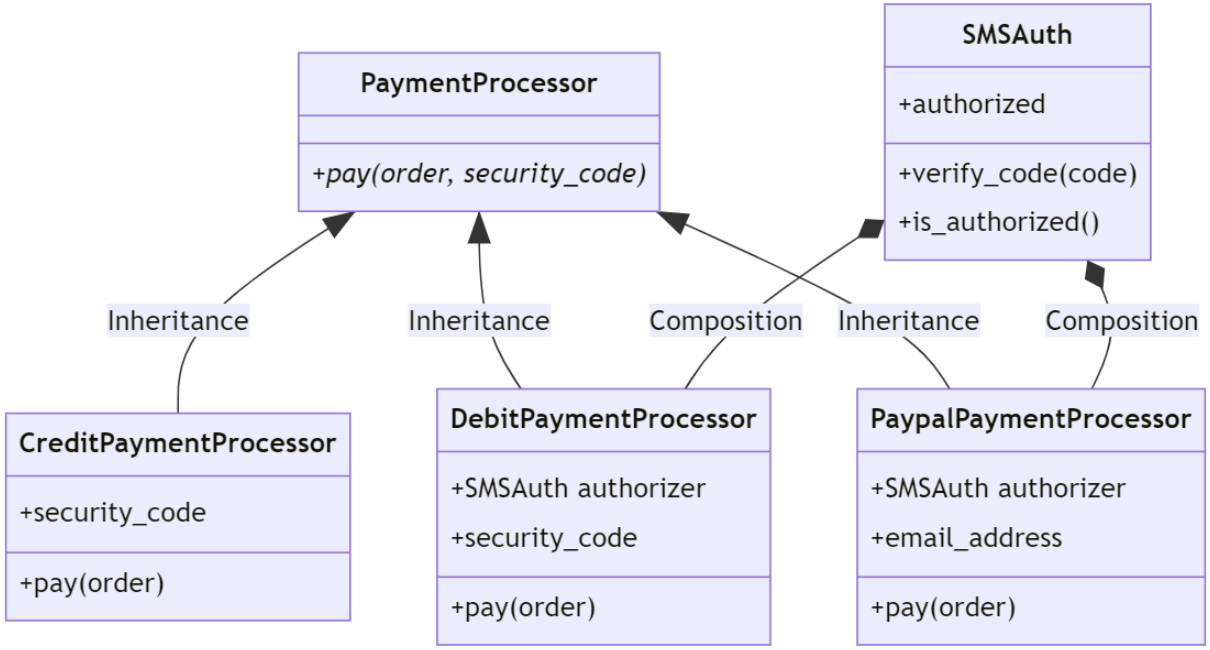
```python
class CreditPaymentProcessor( PaymentProcessor ):

    def __init__(self, security_code):
        self.security_code = security_code

    # def auth_sms(self, code):
    #     raise Exception("Credit card payments don't support SMS code authorization.")

    def pay(self, order):
        print("Processing credit payment type")
        print(f"Verifying security code: {self.security_code}")
        order.status = "paid"
```

신용카드 결재에서
더 이상 필요하지 않게 된
auth_sms( ) 메서드 삭제

Principal #4. ISP 사례 연구 (Case Study)

■ 해결 방법 2. Composition 이용한 해결

- 기존 PaymentProcessor에서는 pay 기능만 구현하도록 수정
- SMS 인증을 담당하는 별도의 클래스 SMSAuth 구현
- 결재 기능을 담당하는 클래스는 필요한 클래스를 합성하여 구현



SOLID #5 DIP. - 의존관계 역전 원칙
(Dependency Inverse Principal)

Principal #5. DIP 이론

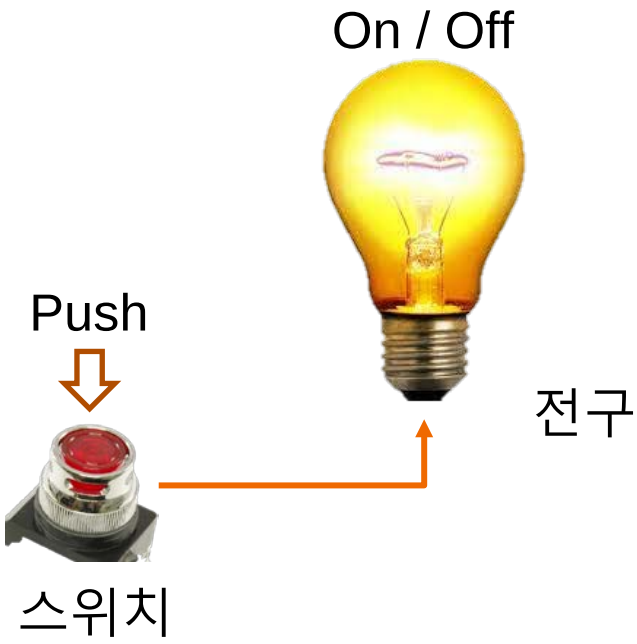
■ 의존관계 역전 원칙 (DIP: Dependency Inverse Principal)

- 의존관계 역전 원칙 (DIP: Dependency Inversion Principle) 원칙은 '클라이언트는 구체 클래스가 아닌 추상 클래스(인터페이스)에 의존해야 한다는 원칙입니다.
- 조금 더 이론적으로 표현하면 다음과 같습니다.
  - 상위 모듈은 하위 모듈에 의존해서는 안됩니다. 상위/하위 모듈 모두 추상화에 의존해야 합니다.
  - 추상화는 세부 사항에 의존해서는 안됩니다. 세부 사항은 추상화에 의존해야 합니다.
- 쉽게 설명하면 다음과 같습니다.
  - 클래스 사이에서 의존관계를 맺을 때는 쉽게 변하는 것 보다는 변화가 없는 것 (또는 변하기 어려운 것)에 의존하라는 의미입니다.
- 여전히 어렵습니다. ㅠㅠ

Principal #5. DIP 이론

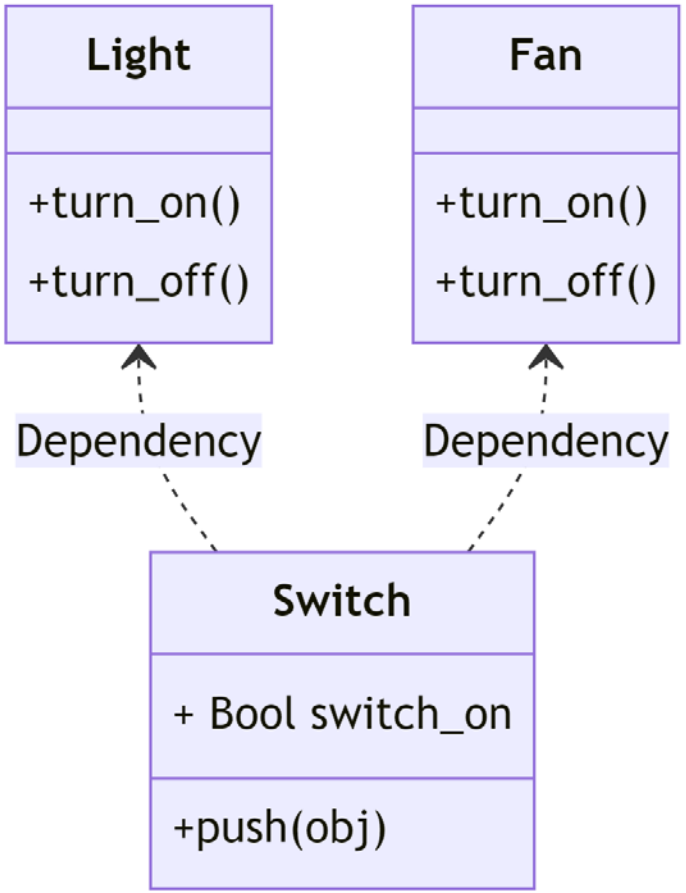
■ **'전구 스위치' 예를 통해서 살펴보도록 하겠습니다.**

- 작동 방식
  - 스위치가 'On' 상태이면, 전구에 불이 들어옵니다.
  - 스위치가 'Off' 상태이면, 전구 불이 꺼집니다.
  - 스위치는 누를 때마다 'On'과 'Off' 상태로 왔다 갔다 합니다.
- 전구
  - turn_on( ) : 전기를 켜는 기능 (메서드)
  - turn_off( ) : 전기를 끄는 기능 (메서드)
- 스위치
  - light_on: 전구 상태를 표시하는 변수 (속성, 멤버 변수)
  - push( ) : 스위치 버튼을 누르는 기능(메서드)
    - 스위치가 Off 상태에서 누르면 light_on 변수가 'On'으로 변경됩니다.
    - 스위치가 On 상태에서 누르면 light_on 변수가 'Off'으로 변경됩니다.



Principal #4. DIP 사례 연구 (Case Study)

■ DIP 준수하지 않는 구현



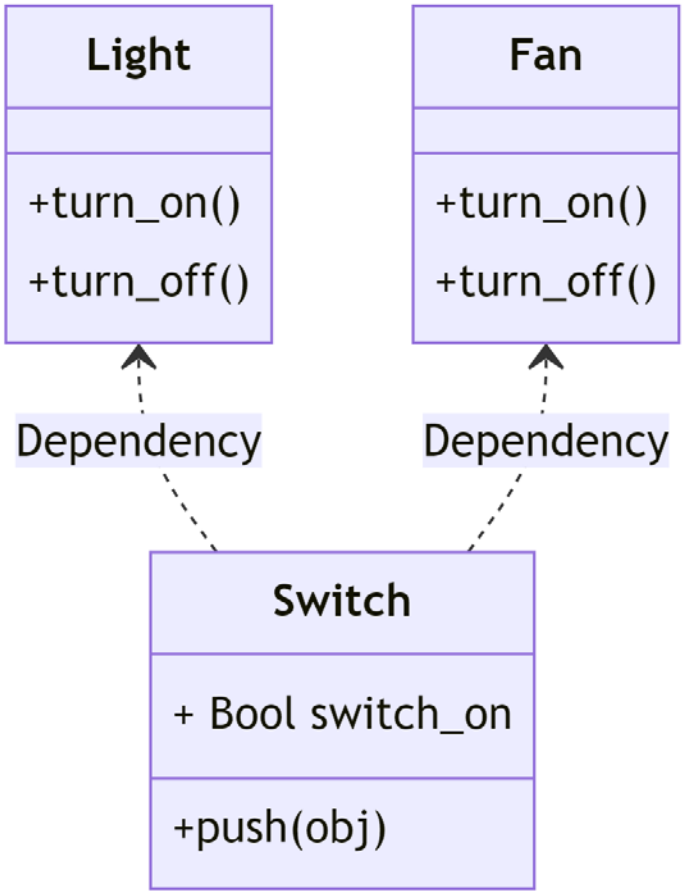

```python
class Light:
    '''전구 클래스'''

    def turn_on(self,):
        print('전구: 전원이 켜졌습니다.')

    def turn_off(self,):
        print('전구: 전원이 꺼졌습니다.')
```

```python
class Fan:
    '''선풍기 클래스'''

    def turn_on(self,):
        print('선풍기: 선풍기가 돌아갑니다.')

    def turn_off(self,):
        print('선풍기: 선풍기가 멈춥니다.')
```

```python
class Switch:
    '''스위치 클래스'''

    def __init__(self, status: bool = False) -> None:
        self.switch_on = status

    def push(self, obj: object):
        '''전구 on/off를 제어하는 메서드'''
        if self.switch_on:
            obj.turn_off()
            self.switch_on = False
        else:
            obj.turn_on()
            self.switch_on = True
```

Principal #5. DIP 이론

■ 문제점



`Light`, `Fan` 클래스는 `turn_on()`, `turn_off()` 기능이라는 저수준의 기능을 공통적으로 가지고 있습니다. 2가지 기능은 항상 필요한 것입니다.

필요에 따라서 항상 필요한 기능 이외에 각각의 클래스에 필요한 기능을 추가할 수 있습니다.

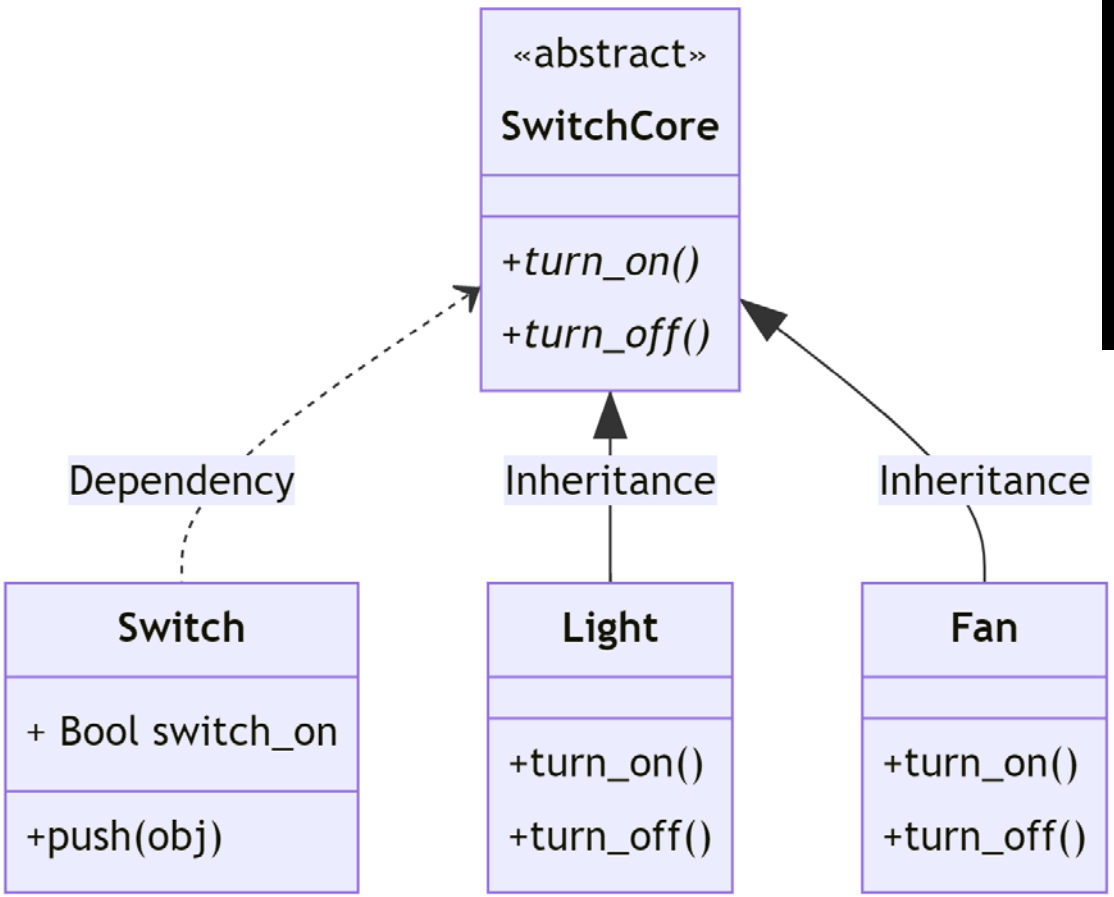
DIP 원칙의 핵심인 `추상화에 의존 해야지 구체화에 의존하면 안된다` 라는 원칙을 위반하고 있습니다.

- `Switch` 클래스는 기능이 구체화된 클래스보다는 변하지 않는 기능에 의존해야 합니다.
- `Light`, `Fan` 클래스가 어떻게 변하더라도 변화가 없는 `turn_on()`, `turn_off()` 기능에 의존하도록 구현해야 됩니다.

추상 클래스를 이용하여 해결!

Principal #4. DIP 사례 연구 (Case Study)

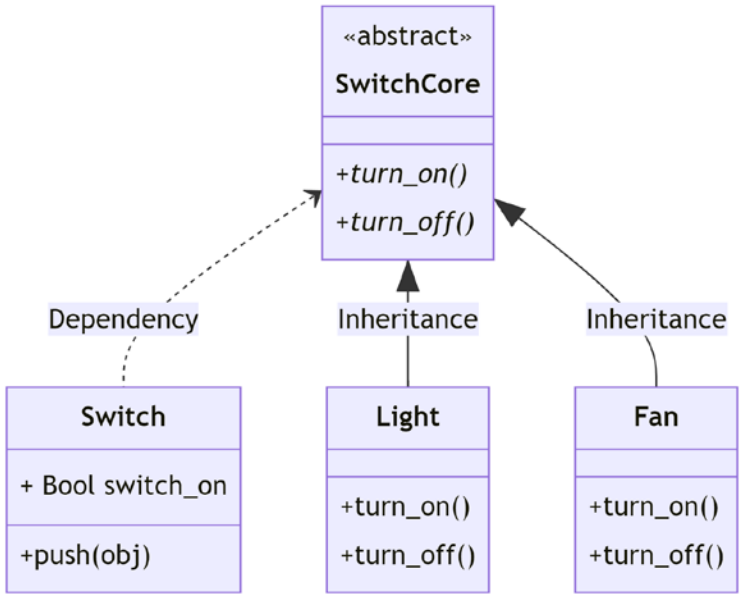
■ DIP 준수하는 구현




```python
from abc import ABC, abstractmethod

class SwitchCore(ABC):
    @abstractmethod
    def turn_on(self,):
        pass

    @abstractmethod
    def turn_off(self,):
        pass
```

Principal #4. DIP 사례 연구 (Case Study)

■ DIP 준수하는 구현




```python
from abc import ABC, abstractmethod

class SwitchCore(ABC):
    @abstractmethod
    def turn_on(self,):
        pass

    @abstractmethod
    def turn_off(self,):
        pass
```

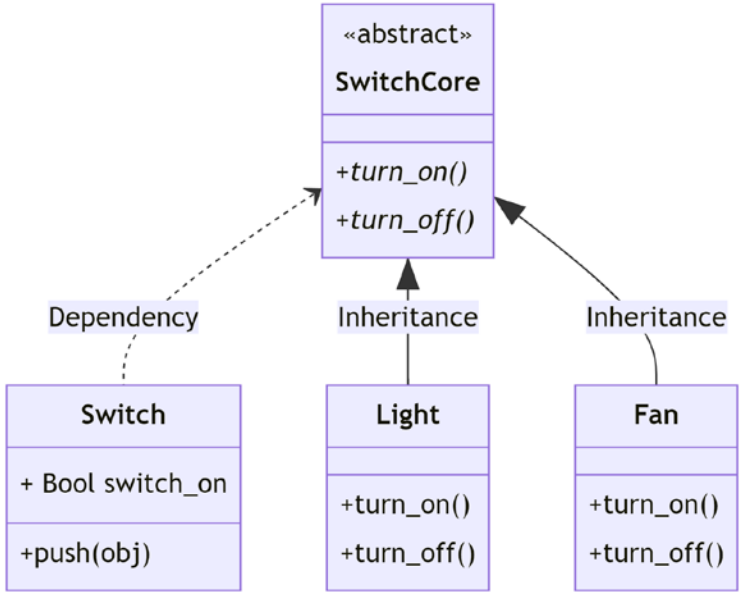
상속 상속

```python
class Light(SwitchCore):
    '''추상 클래스 SwitchCore를 상속받아
    전구 클래스 구체화'''

    def turn_on(self,):
        print('전구: 전원이 켜졌습니다.')

    def turn_off(self,):
        print('전구: 전원이 꺼졌습니다.')
```

```python
class Fan(SwitchCore):
    '''추상 클래스 SwitchCore를 상속받아
    선풍기 클래스 구체화'''

    def turn_on(self,):
        print('선풍기: 선풍기가 돌아갑니다.')

    def turn_off(self,):
        print('선풍기: 선풍기가 멈춥니다.')
```

Principal #4. DIP 사례 연구 (Case Study)

■ DIP 준수하는 구현




```python
from abc import ABC, abstractmethod

class SwitchCore(ABC):
    @abstractmethod
    def turn_on(self,):
        pass

    @abstractmethod
    def turn_off(self,):
        pass
```

의존

```python
class Switch:
    '''스위치 클래스'''

    def __init__(self, status: bool = False) -> None:
        self.switch_on = status

    def push(self, obj: SwitchCore):
        '''전구 on/off를 제어하는 메서드'''
        if self.switch_on:
            obj.turn_off()
            self.switch_on = False
        else:
            obj.turn_on()
            self.switch_on = True
```

소프트웨어
꼰대 강의!

수고하셨습니다 ..^^..