리팩토링을 위한 디자인패턴

CONTENTS

목차

Chapter1 GoF 디자인패턴 소개

Chapter2 Factory Method Pattern

Chapter3 Builder Pattern

Chpater4 Singleton Pattern

Chapter5 Pattern 을 사용한 개발 PJT 1

Chapter6 Command Pattern

잘 설계된 S/W란?

- 고객의 요청들

  제작 요청 : 이거 만들어주세요.
  기능 추가 요청 : 이거 기능 추가해주세요.
  변경 요청 : 이 부분 수정해주세요.

- 잘 설계된 S/W 조건

  고객 요구사항 대로 개발이 되었는가? 기본
  구조가 쉽게 파악이 되는가?
  기능 추가를 쉽게 할 수 있는가?
  변경이 편리한가?

고객의 요구사항

잠만 잘수있게 해주세요.

- 저는 잠에서 잘 안 깨요.
- 나중에 수정사항은 전달 드릴게요.

이렇게 만들어진 결과물

오른쪽 집은,
변경 요청에 대응이 어렵다.

- 구조 파악이 힘들다.
- 창문 교체 요청
  (집 전체가 무너질 수 있음)

수정이 편리한 S/W 결과물

조립식 건물

- 지붕 교체 가능
- 문 교체 가능
- 창문 교체 가능

하지만, 아무리 잘 만든 S/W 라도 …

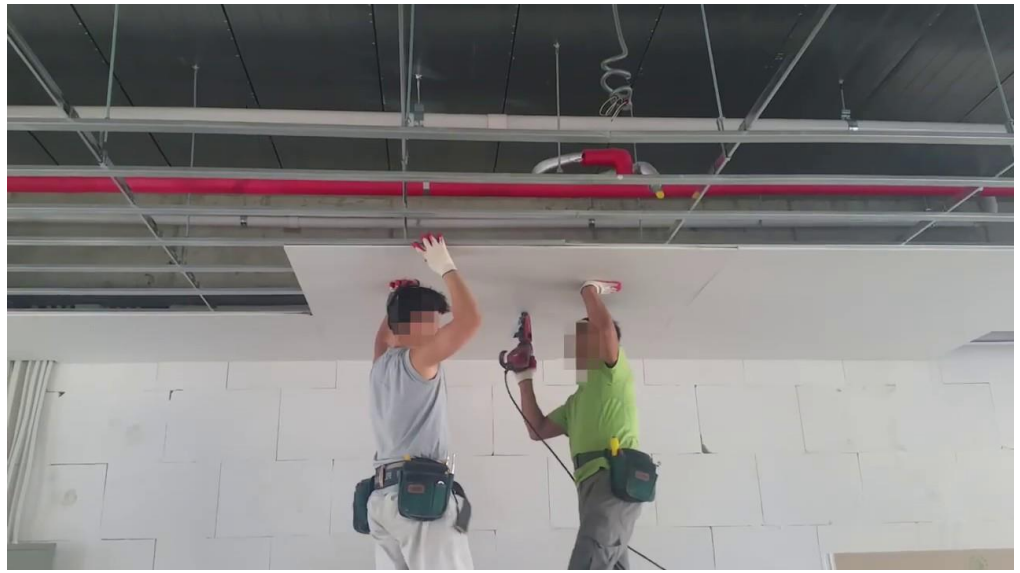
당연하게도
무리한 고객 변경 요구사항은 수용 불가

변경요청
수용불가

아름다운 구조? 복잡하다.

구조가 단순해야, 유지보수가 된다.

유지보수가 힘든 복잡한 구조의 천장



오랜시간이 지나도, 유지보수가 용이한 구조

깔끔한 구조로 OOP 개발 하기 위한 훈련

1. 객체지향 원칙을 학습하는 방법
   - SOLID 원칙
2. 깔끔한 코딩 뼈대를 학습하는 방법
   - 디자인 패턴
3. 리팩토링
   - 깔끔한 구조로, 지속적인 개선하는 훈련

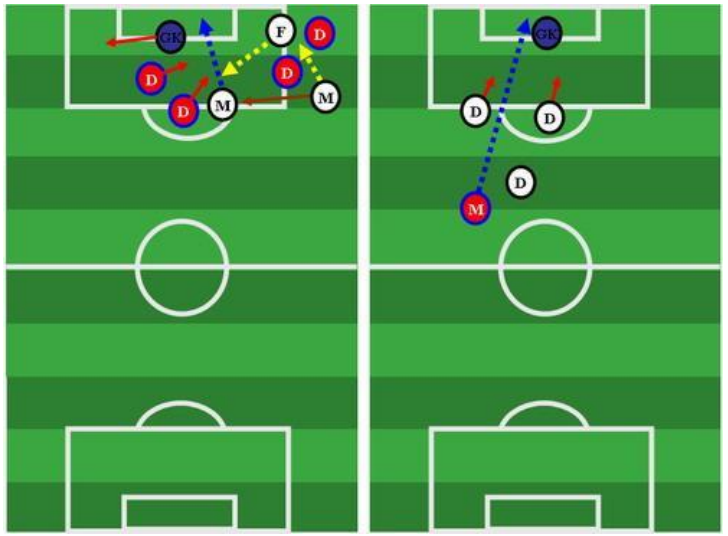
Chapter1

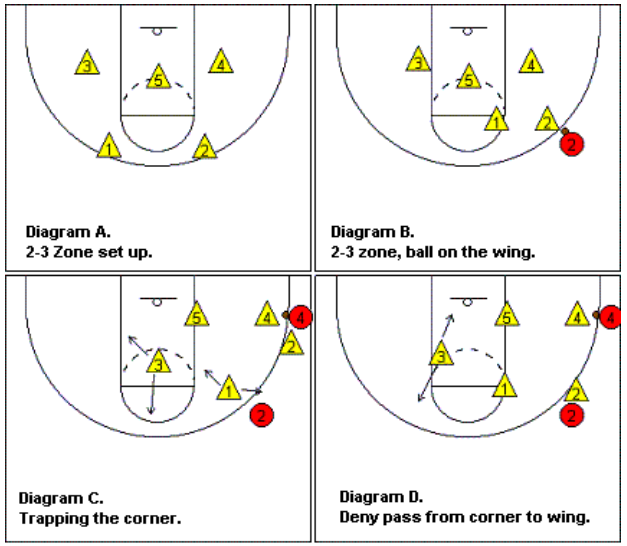
GoF 디자인패턴 소개

Gang of Four 가 만든, 소스코드 패턴

디자인 패턴이란?

자주 사용되는 설계를 패턴화 하여, **이름을 붙인 것**





“디자인패턴” 의 정확한 정의

- **디자인패턴**
  건축에서 먼저 사용된 용어

- 정의
  정의 1 : **SW 개발에서 문제를 해결하기 위해 반복적으로 사용되는 솔루션**
  정의 2 : SW의 Context / Problem / Solution 의 관계를 나타냄
  Context : 상황, 배경
  Problem : Context 안에서 발생된 이슈
  Solution : 해결책

  > 어떤 Context에서 Problem이 생겼을 때 적용 가능한 Solution

디자인 패턴의 장단점

장점

- 개발자간 의사소통
- S/W 구조 파악이 쉽다.
- 설계 변경 요청에 대해 유연한 대처 가능

디자인 패턴의 장단점

단점

- 개발에 참여하는 팀원 전체가 학습되지 않으면,
  유지보수가 더 어려워진다.

- 구현 난이도가 올라간다.

GoF 디자인 패턴

소프트웨어 설계에 공통된 문제에 대한 표준 해법

- 네 명의 학자가 쓴 책이기에 Gang of Four, GoF 디자인패턴으로 불리운다.

Factory Pattern

객체를 생성해주는 공장

Factory Method Pattern 학습 순서

1. Simple Factory

> 객체를 생성해주는 클래스

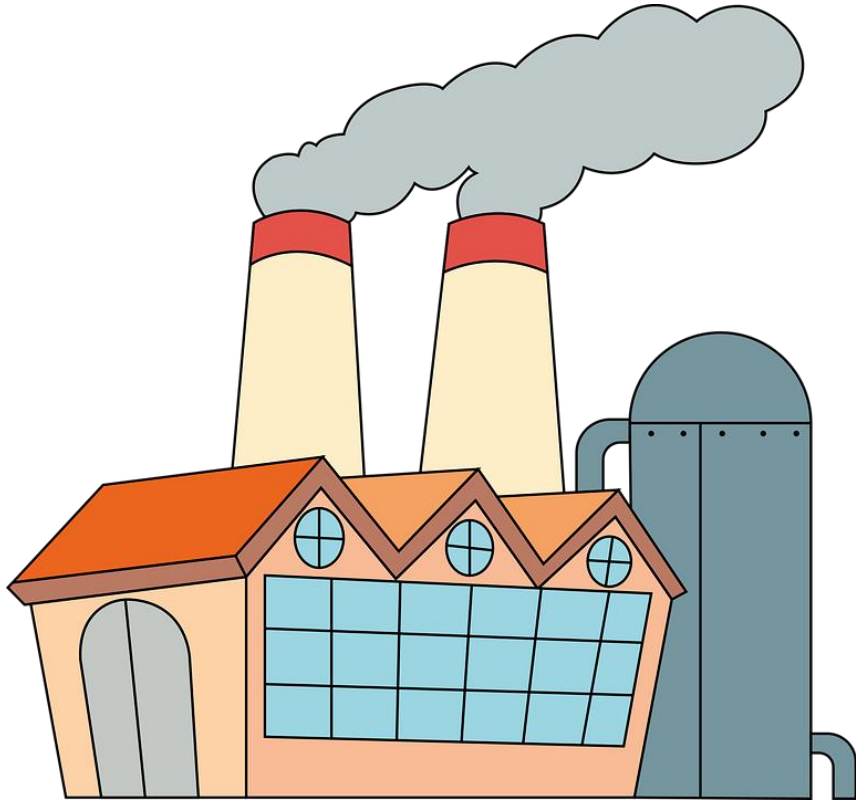
2. Factory Method Pattern

> 객체 생성을 하위 클래스에서 담당하도록 함

Simple Factory 형태

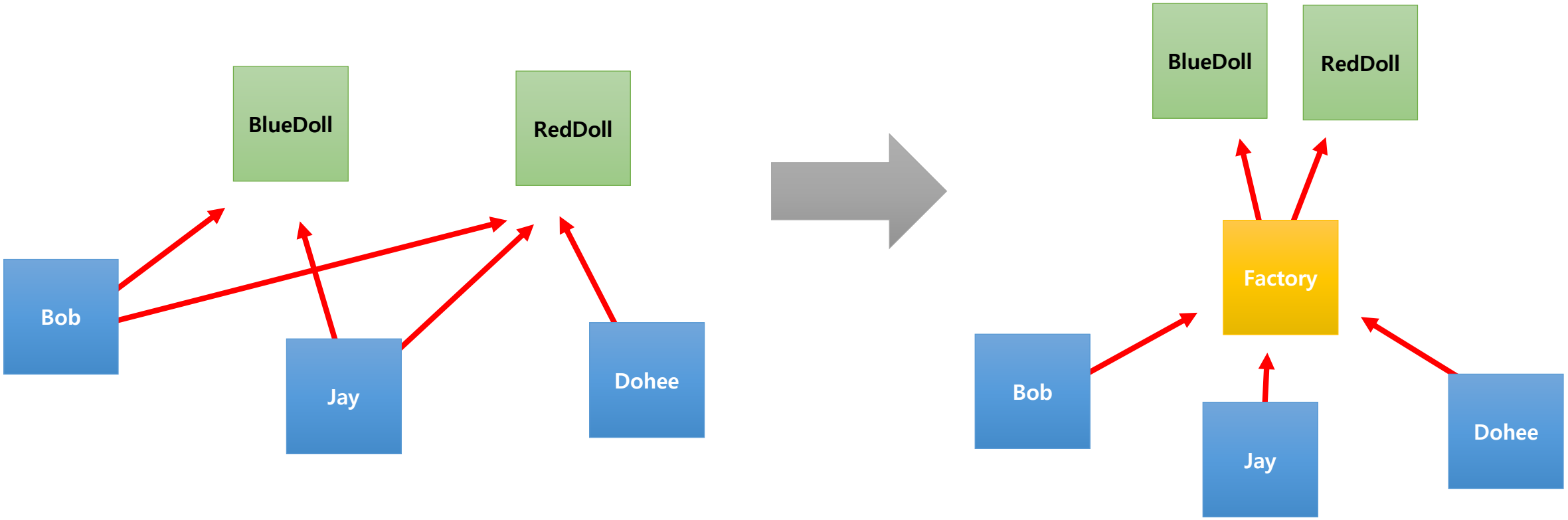
- 객체를 생성해주는 곳이 한곳이다.
  한 곳에서 생성, 관리 가능

- 생성코드를 Client에 공개하지 않는다.
  클라이언트는 factory 에게 product 생성을 요청한다.



Simple Factory를 쓰는 이유

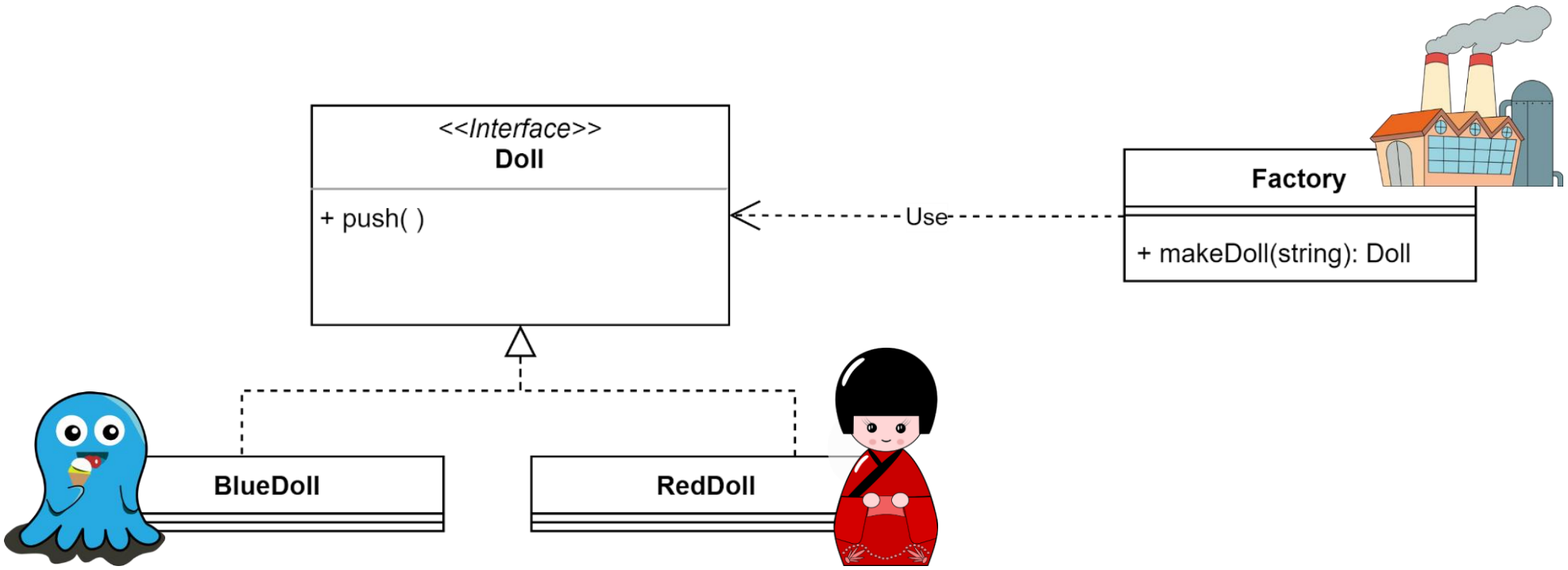
객체 생성시 의존복잡도가 높을 때 의존 복잡도를 낮출 수 있다.



UML 분석 1 (Factory 형태)

makeDoll(string)

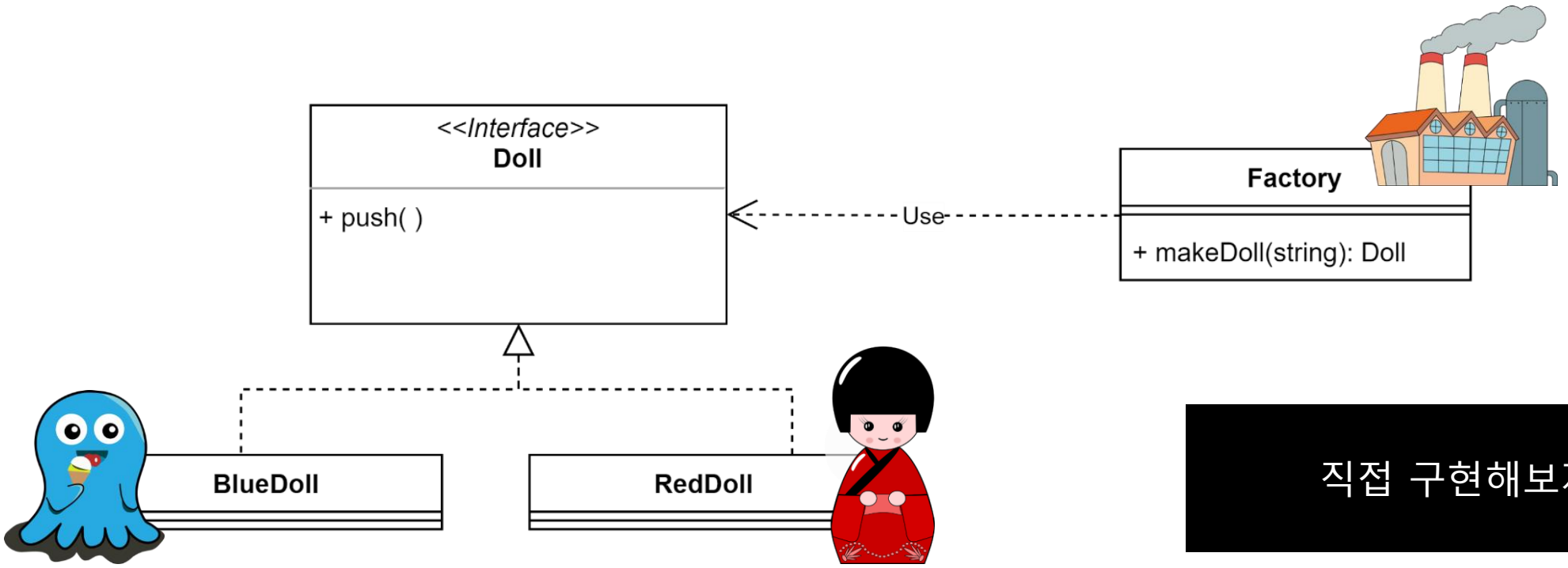
- makeDoll("RED") : Red 인형 생성
- makeDoll("BLUE") : Blue 인형 생성



[도전] doll factory

makeDoll(string)

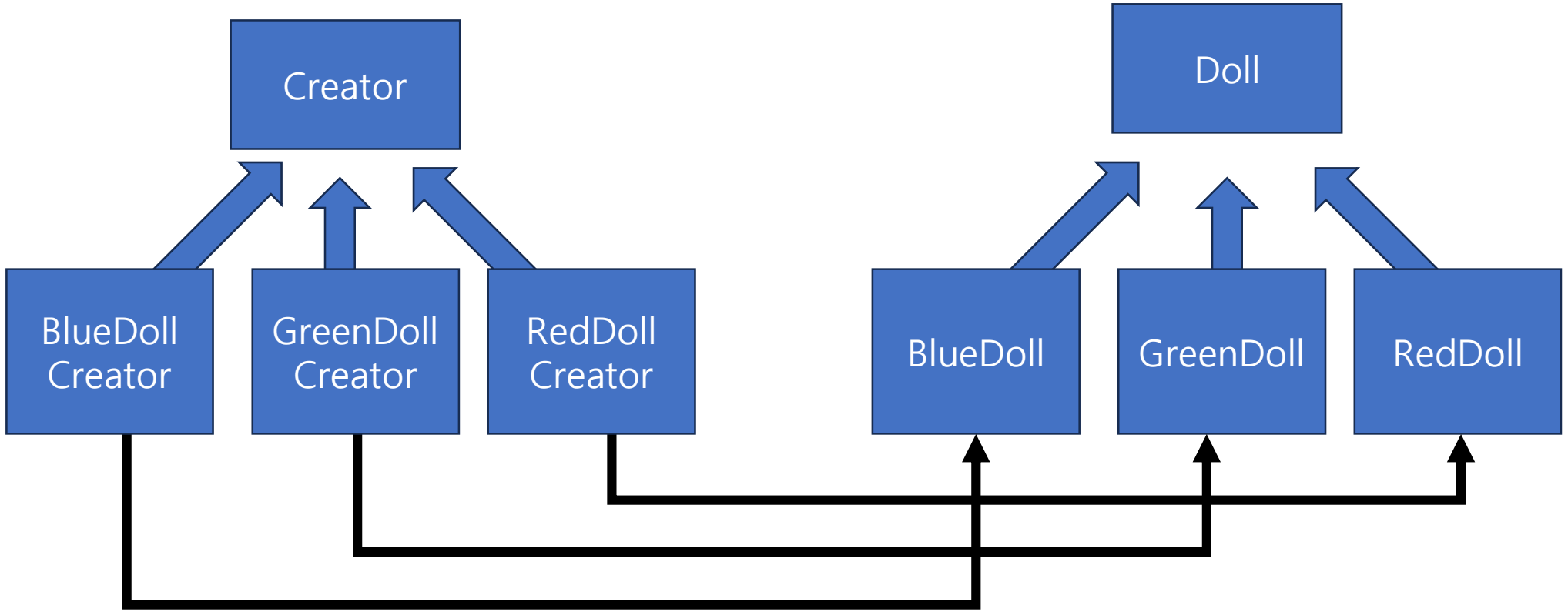
- makeDoll("RED") : Red 인형 생성
- makeDoll("BLUE") : Blue 인형 생성



직접 구현해보자

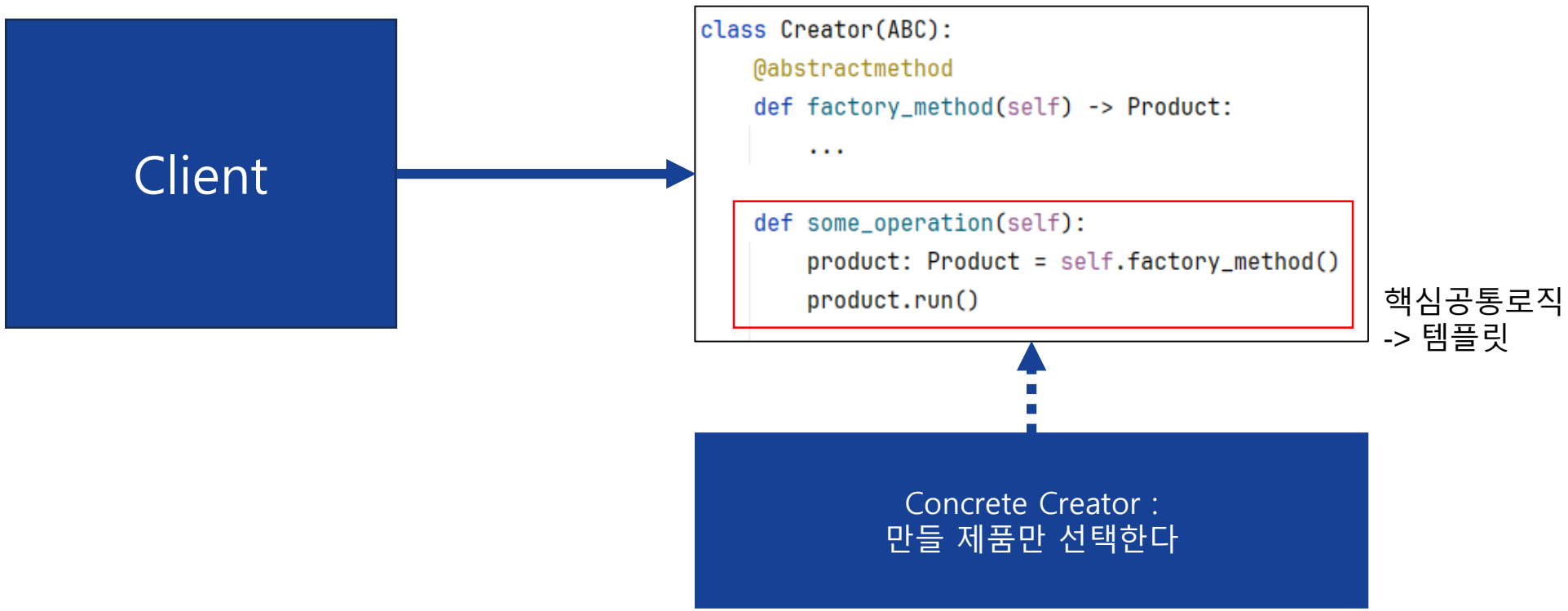
Factory Method Pattern

새로운 Doll이 추가되더라도,
기존 코드에 변경이 없다.



gof 의 factory method 패턴

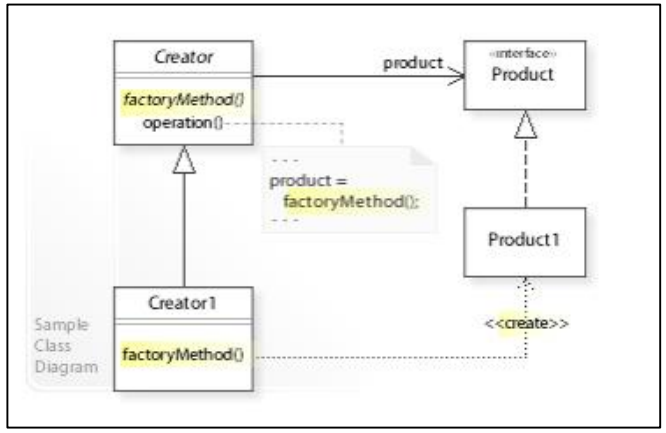
팩토리 메서드 패턴은 하위 클래스에서 어떤 객체를 생성할지 정한다.
상위 클래스에는 추상 메서드(factory_method)를 만들어 두고,
하위 클래스에는 추상 메서드의 구현을 결정한다.

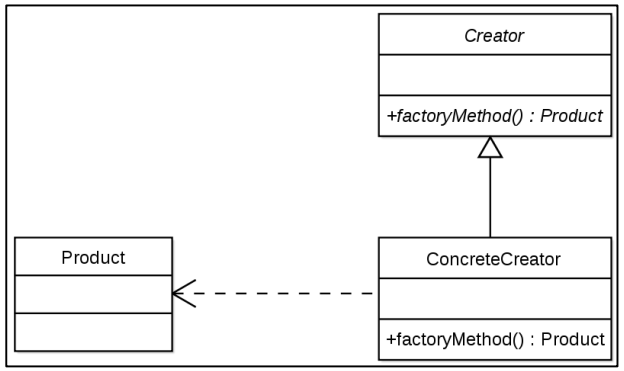


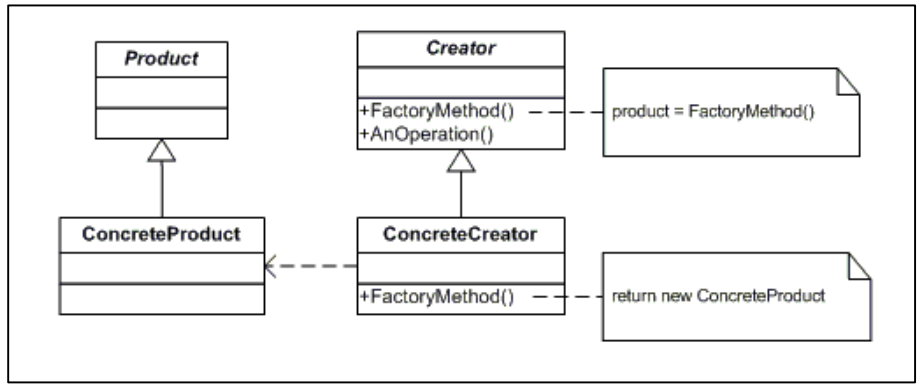
[참고] factory method UML 표현 예시

Creator : factory method 를 추상함수로 만든다.

ConcreteCreator : factory method 의 구현을 제작한다. 즉, 어떤 제품을 생성해 줄 지 코드를 작성한다.







[도전] factory method

Button 을 클릭했을 때 수행되는 Action 들을 구성할 것이다.
factory method 패턴을 적용해본다.

이후에 새로운 Button 과 Action 들을 조합해서 확장하기 좋은 설계로 만든다

```python
class SaveAction:
    def run(self):
        print("파일 저장...")


class ExitAction:
    def run(self):
        print("프로그램 종료...")


class SaveButton:
    def click(self):
        action = SaveAction()
        action.run()


class ExitButton:
    def click(self):
        action = ExitAction()
        action.run()
```

https://gist.github.com/jeonghwan-seo/a1b53b26e4d6f2cbbeecc035a93d791d

Singleton Pattern

단 하나의 Instance만을 사용한다.

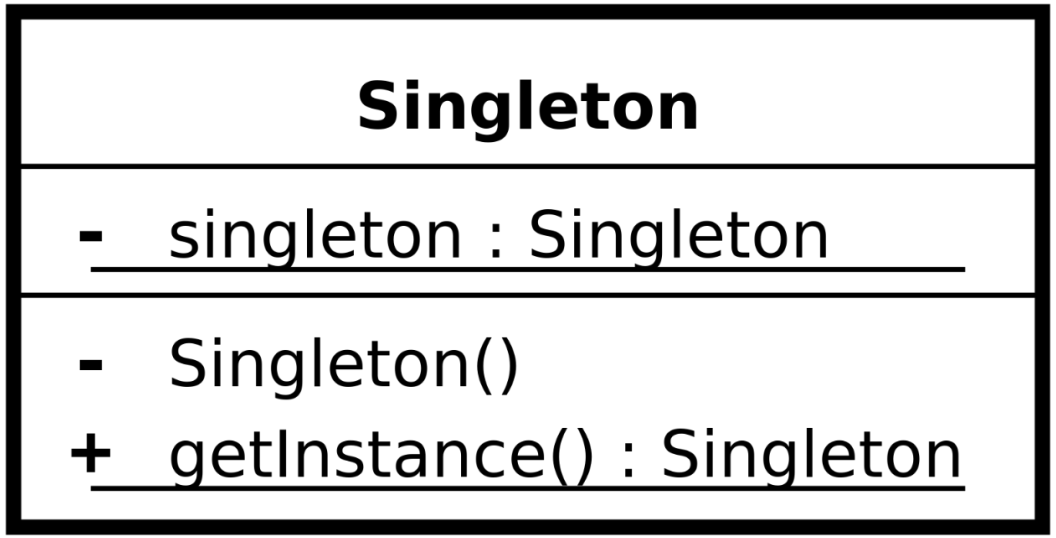
싱글톤이란

하나의 인스턴스만을 사용하도록 하는 경우, 사용하는 디자인 패턴

전역적으로 하나의 인스턴스를 사용하는 패턴이다

```
wife1 = Wife()
wife2 = Wife()

wife1.date()
wife2.date()

assert wife1 is wife2
```

Singleton UML



1. 생성자를 직접 이용하지 못하게 하여, Client에서 생성하지 못하도록 막는다.
2. getInstance( ) 메서드를 통해서 해당 객체를 얻는다.
3. 전역 인스턴스를 제공해준다.

모듈자체를 싱글턴으로

모듈은 기본적으로 한 번만 로드된다. 모듈 내의 변수를 싱글턴처럼 사용한다.

singleton.py

```
class Singleton:
    def __init__(self):
        self.value = None


singleton_instance = Singleton()
```

singleton.py 모듈

다른 파일

```
from singleton import singleton_instance


singleton_instance.value = "HI"
print(singleton_instance.value)
```

클래스로 싱글턴 구현

```python
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super().__new__(cls)

        return cls._instance

    def __init__(self,value):
        if not hasattr(self, 'initialized'):
            self.value = value
            self.initialized = True
```

인스턴스 생성시, _instance 여부 확인

생성된 인스턴스 초기화

[도전] 싱글톤 직접 구현해보기

싱글톤을 이용해서 프로그램 전체에 동일한 인스턴스를 공유해야 한다.

Cursor, AudioManager 등 이런 클래스들은 프로그램 전체에서 하나의 인스턴스만 필요하다.

디바이스의 상태를 추적해주는 DeviceState 클래스를 제작한다.
battery level , temperature 를 관리한다.

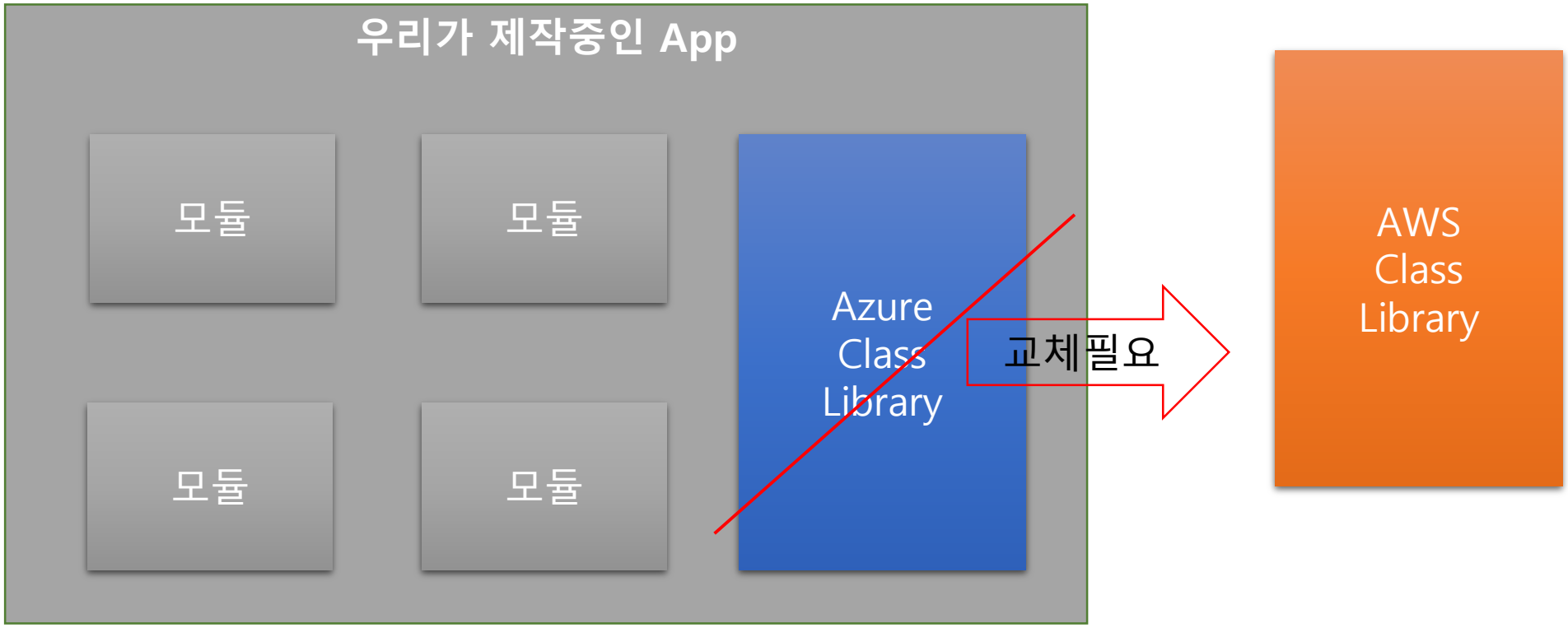
1. battery level, temperature 를 업데이트하는 함수
2. 디바이스 연결, 해제를 하는 함수 (connect, disconnect)

Adapter Pattern

다른 인터페이스를 갖는 객체의 인터페이스 통일. 변환

어댑터 패턴을 쓰는 이유

Library 교체가 필요한 상황이다.



그냥 교체하면 문제점

Azure API 쓰던 기존 코드 전부 수정 필요!

- 모듈에 대한 테스트도 처음부터 다시 해야 한다.

Azure
Class
Library

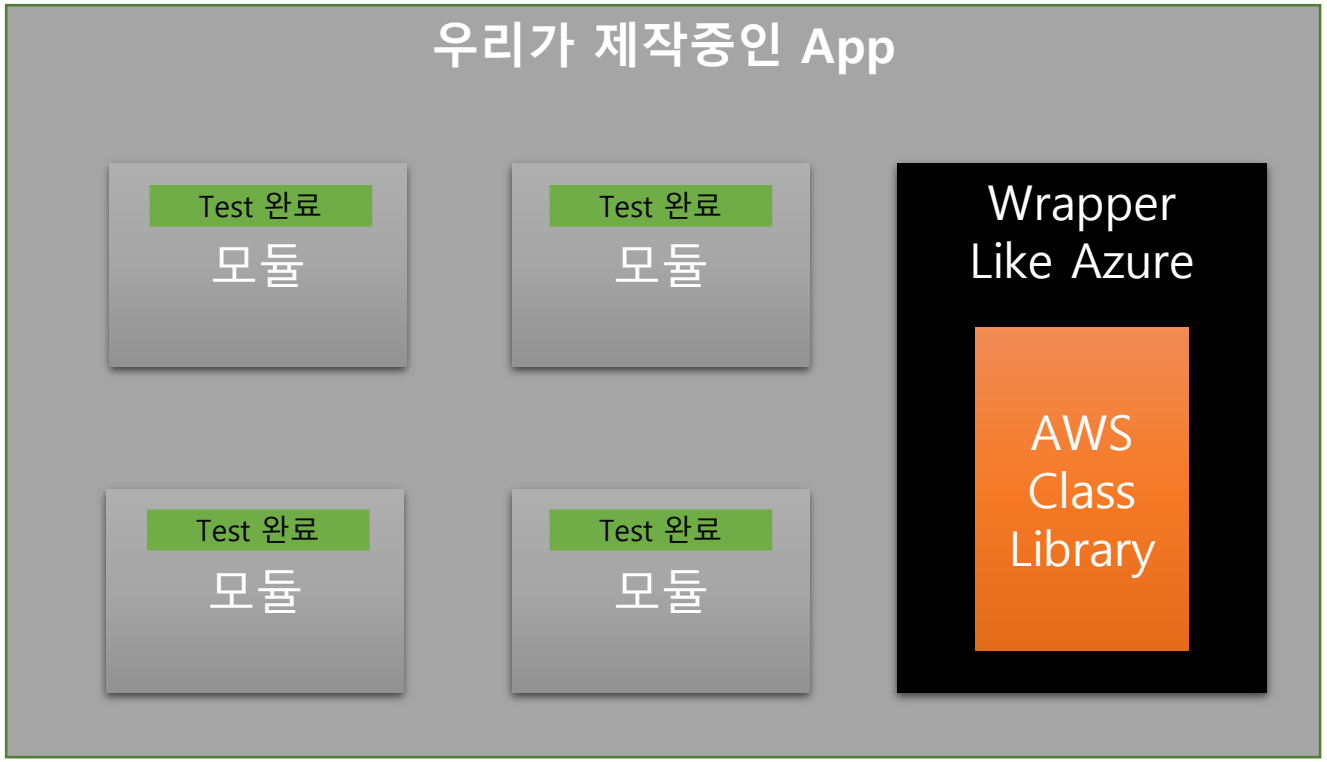
기존 소스코드를 수정을 하기 싫다면???

새로운 Library API에 맞게,
Test가 완료된 모듈을 수정할 생각이 없다면??

Azure 사용방법과 동일한 Wrapper

Azure API와 동일하게 사용하면서

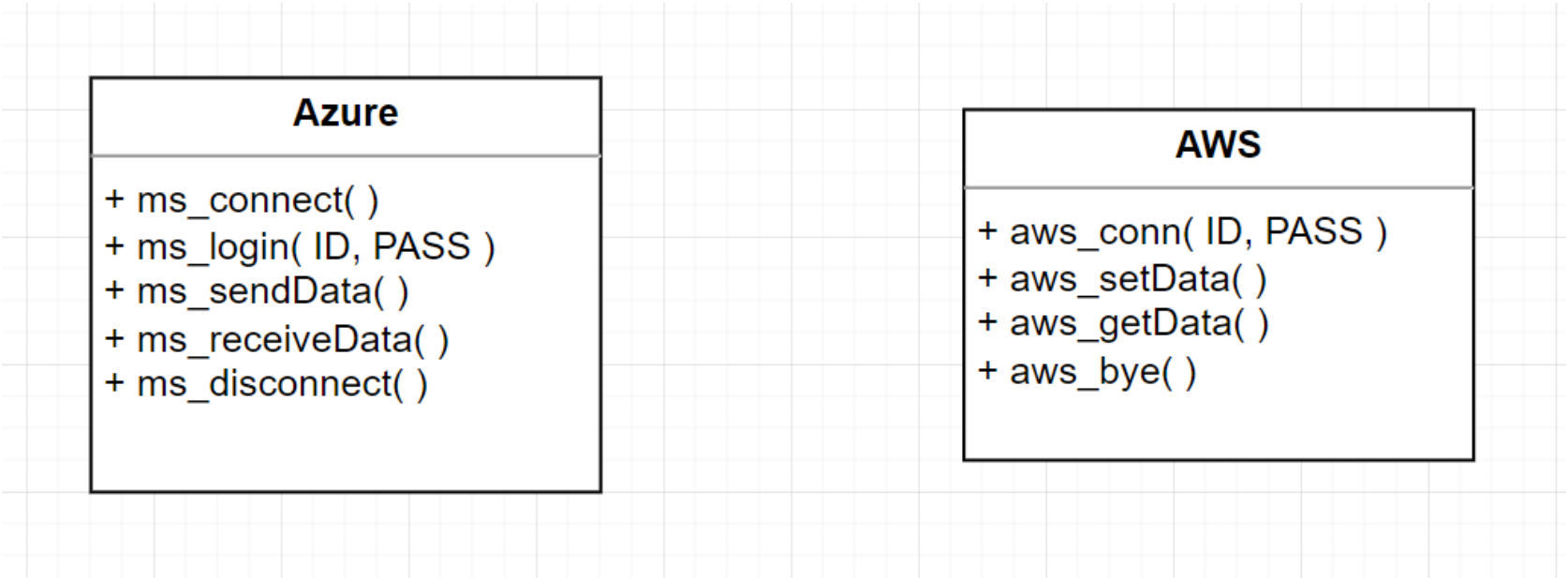
AWS Class Library 를 쓸 수 있도록 Wrapper를 만들자!



2개의 클래스 만들기

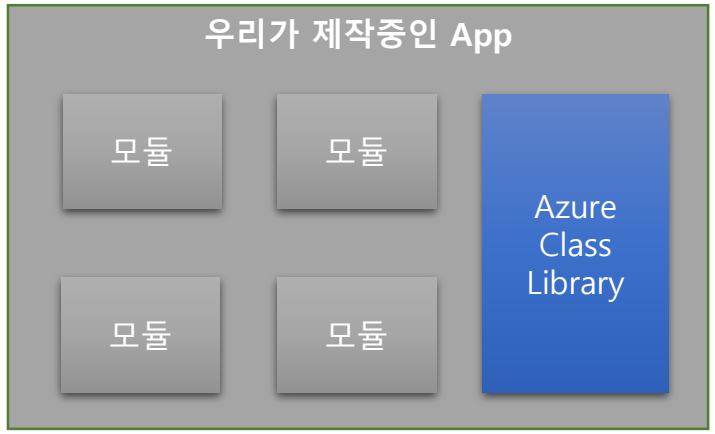
Azure와 AWS 클래스 제작

- 단순 출력



Client 코드 작성하기

Main 코드에서
Azure Class를 사용하는
코드 작성하기




```
def run(az: Azure):
    az.ms_connect()
    az.ms_login("KFC", "1234")
    az.ms_send_data()
    az.ms_disconnect()


def client():
    run(Azure())
```

[도전] 상황 : 긴급교체

run 메서드는 테스트가 많이 된 상태
안정적인 Client 코드를 최소한으로 변경하기.

상속을 이용한
Wrapper Class를 제작할 것.

```
def run(az):
    az.ms_connect()
    az.ms_login("KFC", "1234")
    az.ms_sendData()
    az.ms_receiveData()
    az.ms_disconnect()


if __name__ == "__main__":
    run(Azure())
```

[도전] 다음 상속코드를 이해하고, 안보고 직접 구현해보기

- Adapter Class 하나 제작

  Azure와 AWS Library 수정은 불가한 상황에서는 다음과 같이 구현할 수 있다.

```python
# Main
if __name__ == "__main__":
    az = Adapter(AWS())

    az.ms_connect()
    az.ms_login("ADMIN", "BBQ")
    az.ms_sendData()
    az.ms_sendData()
    az.ms_disconnect()
```

```python
# Adapter class
class Adapter(Azure):
    def __init__(self, aws):
        self.aws = aws

    def ms_connect(self):
        # 아무것도 안함
        pass

    def ms_login(self, id, password):
        self.aws.aws_conn(id, password)

    def ms_sendData(self):
        self.aws.aws_setData()

    def ms_receiveData(self):
        self.aws.aws_getData()

    def ms_disconnect(self):
        self.aws.aws_bye()
```

Diagram 이해하기

방금 제작했던
상속을 이용한 Wrapper, Diagram

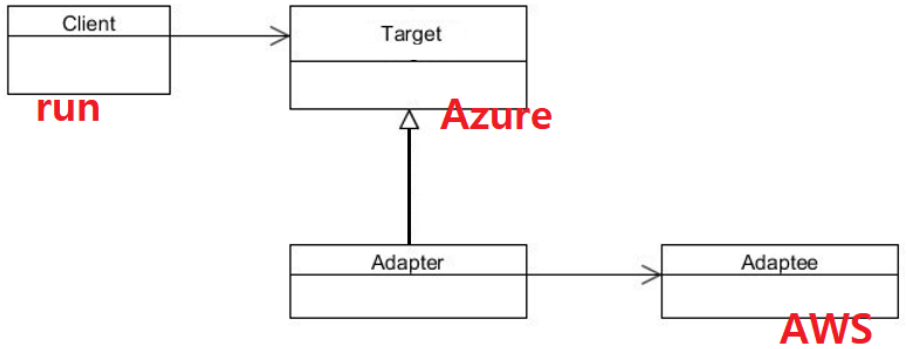
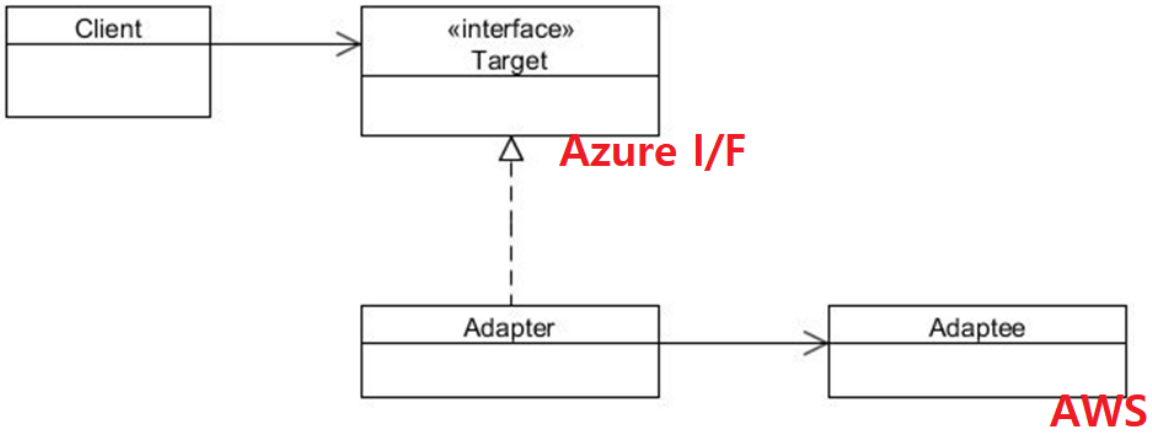
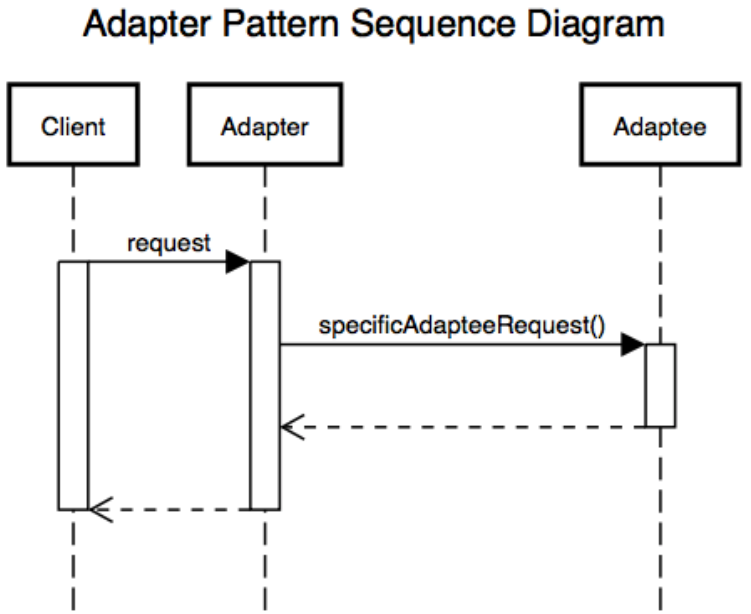


Diagram 이해하기

GoF 어댑터 패턴은 Target Interface를 상속

- Azure API와 똑같이 생긴 Interface를 추가하고,
  실제 Azure API를 걷어낸다.





[도전] 어댑터패턴 구현하기

Logger

Logger 클래스는 파일 인터페이스에 의존한다.

MessageList

MessageList 는 write, flush 가 없고, 기존 코드는 변경하지 못한다.

Logger 가 MessageList를 이용할 수 있도록 Adapter 패턴을 구현

```python
class Logger:
    def __init__(self, file):
        self.file = file

    def log(self, message):
        self.file.write(message + '\n')
        self.file.flush()
```


```python
class MessageList:
    def __init__(self):
        self.data = []

    def append_message(self, msg):
        self.data.append(msg)

    def get_all(self):
        return self.data
```

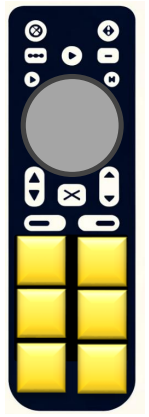
Chapter6

Command Pattern

Command 를 객체로 캡슐화. 실행 / 취소 / 저장을 처리하는 패턴

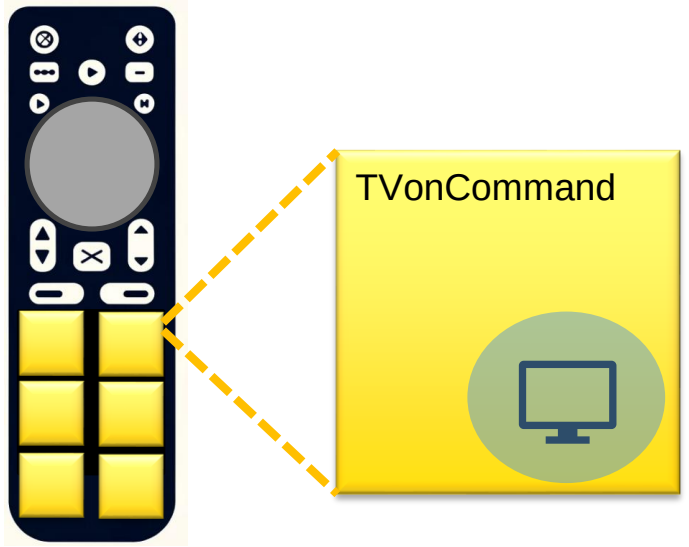
커맨드 패턴

명령을 하는 측과 수행을 하는 측을 **분리**하는 패턴



명령하는 측
(Invoker)

수행하는 측
(Receiver)



일반적인 방식

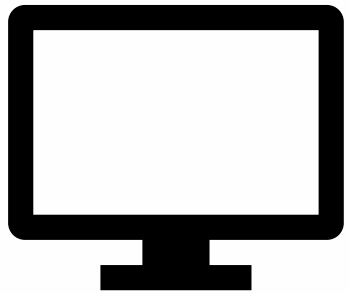
일반적인 방식은 Receiver 를 직접 호출한다.

이런 경우에는

1. 어떤 요청을 할지 (요청)
2. 언제 실행할지 (실행 시점)
3. 어떻게 실행할지 (수행 방법)

가 **한 줄에 구현**이 된다.

```
class TV():
    def on(self):
        print("TV 실행됨")


tv: TV = TV()
tv.on()
```



1. TV on 을 요청
2. 지금 당장 실행이 된다
3. TV 인스턴스가 실행

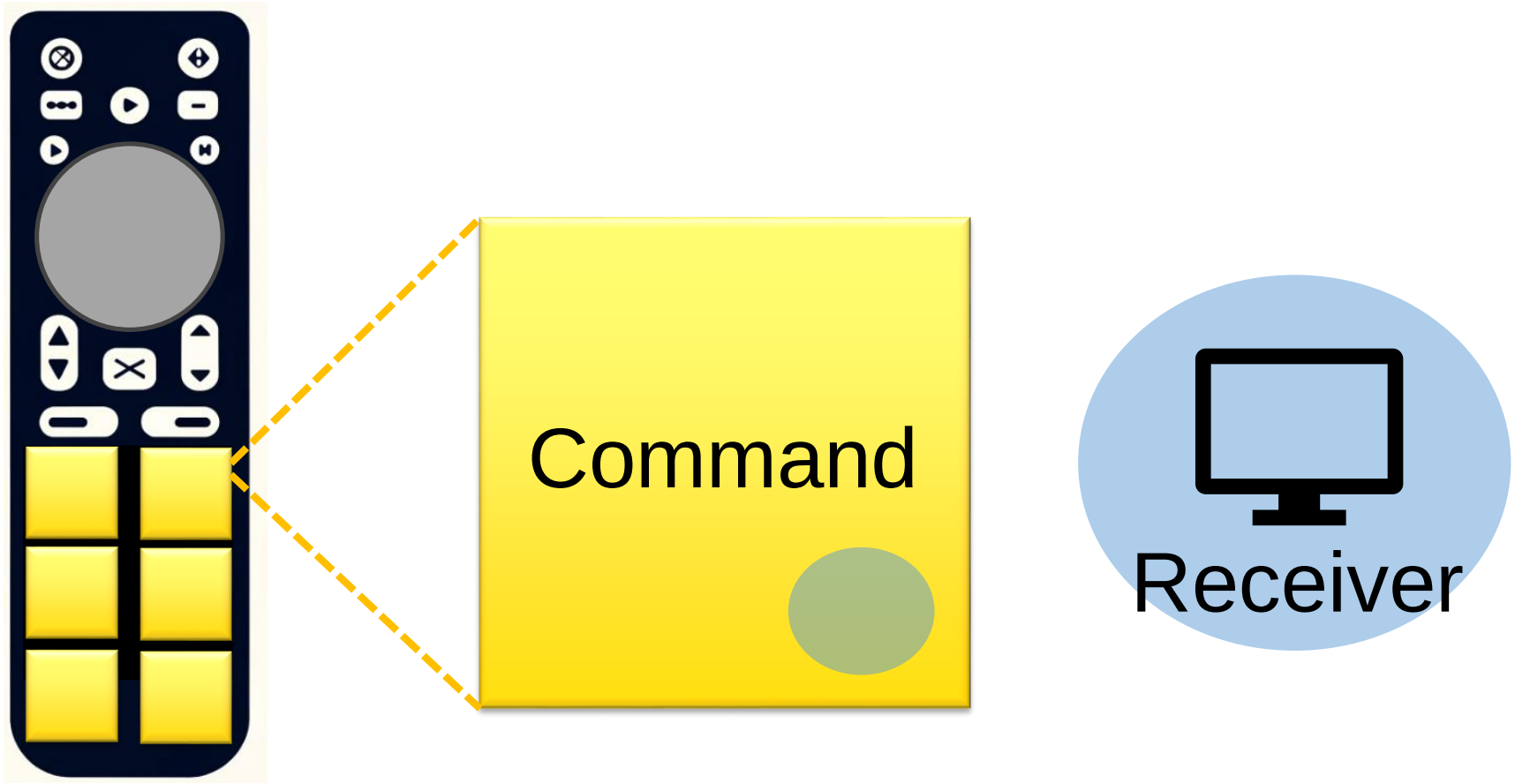
커맨드 패턴 적용

커맨드 패턴을 이용하면 다음과 같이 분리된다.

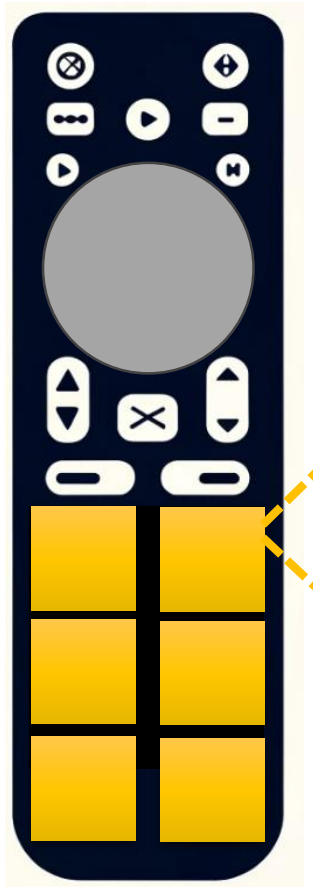
```
tv = TV()
tv_cmd = TVonCommand(tv)

remote_controller = Invoker()
remote_controller.set_command(tv_cmd)

remote_controller.run()
```

무엇을 실행하는가?	Tv를 켜는 동작을 실행한다
언제 실행하는가?	remote_controller이 결정
어떻게 실행하는가?	tv.on() 이 run 안에 캡슐화 됨

Command 패턴의 각 역할



Command 패턴의 각 역할



Invoker

Receiver

Receiver는 실제 동작을 수행을 하는 객체이다

```
class TV:
    def turn_on(self):
        print("tv 켜짐")
        # ...
```

Command

Receiver 의 동작은 Command 객체에 의해 캡슐화 되고,
Command 객체 뒤에 Receiver의 동작이 감춰진다.

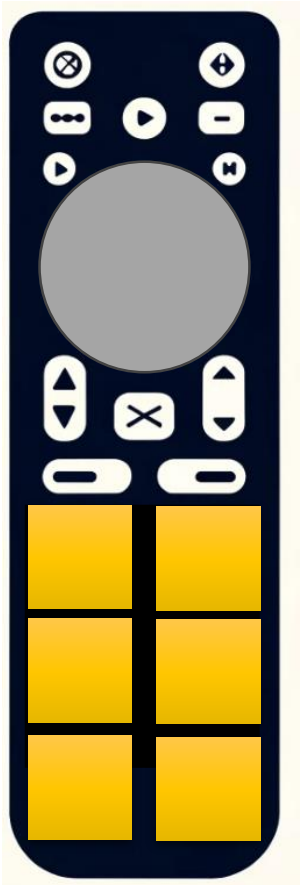
```
class Command(ABC):
    @abstractmethod
    def execute(self):
        raise NotImplementedError("구현부를 만드시오")
```

```
class TVonCommand(Command):
    def __init__(self, tv: TV):
        self.my_tv = tv  # reciever

    def execute(self):
        self.my_tv.turn_on()
```

Invoker

Invoker는 캡슐화 된 Receiver 의 동작을 전혀 모른 채 Command 를 다룰 수 있다

```
class Invoker:
    def __init__(self):
        self.command = None

    def set_command(self, command):
        self.command = command

    def run(self):
        self.command.execute()
```

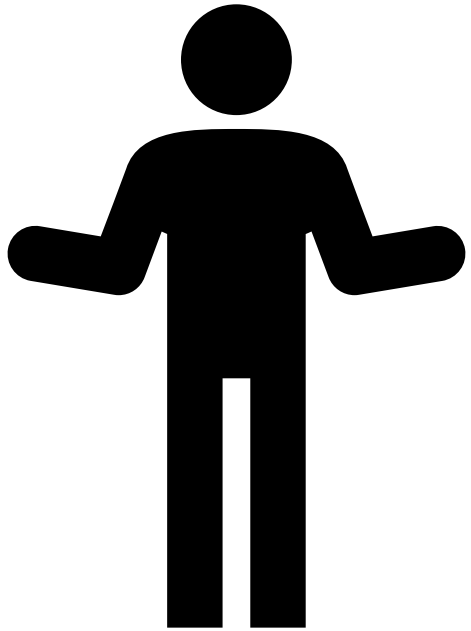


Invoker

Command 패턴 흐름

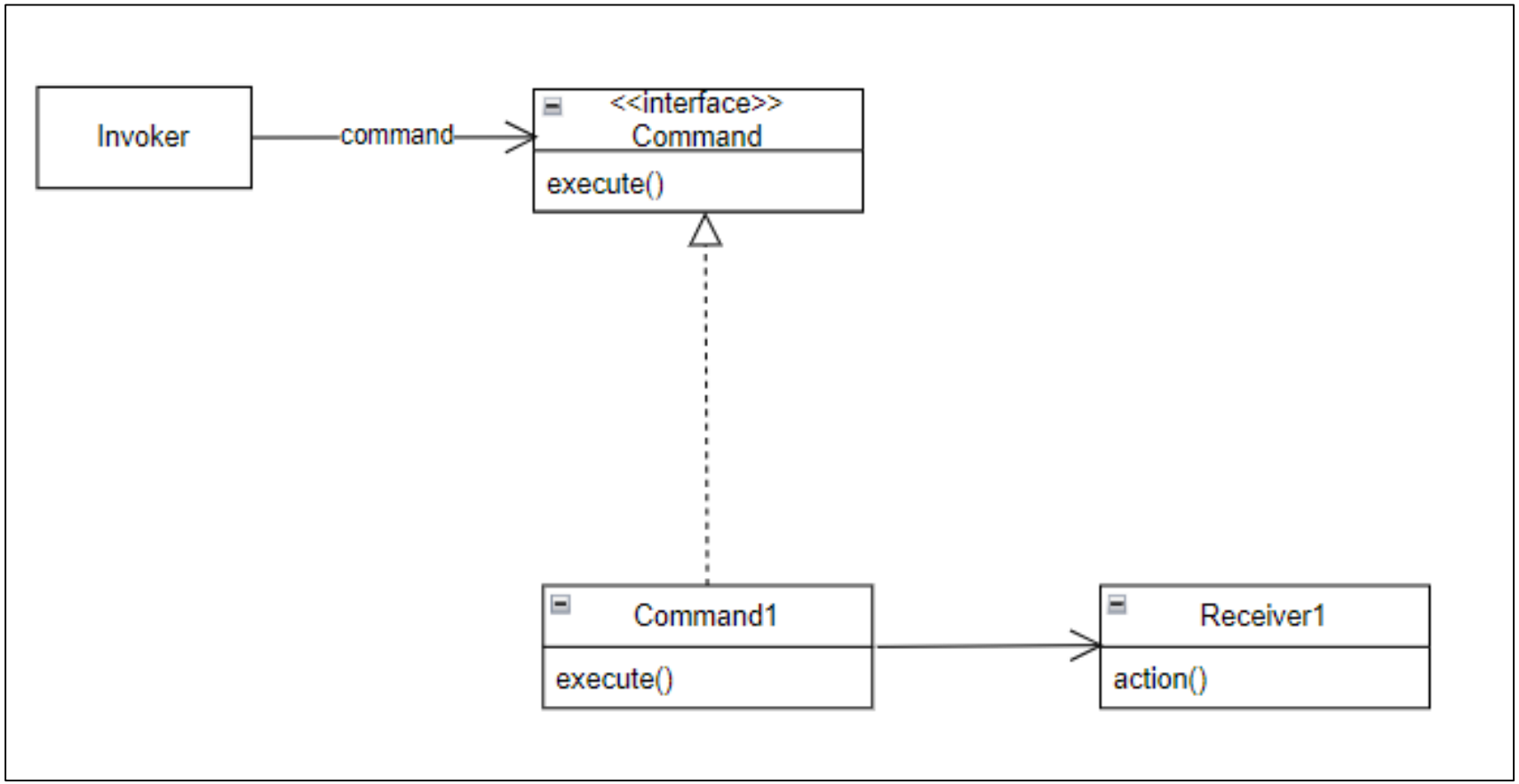
1. Client 가 Receiver 와 Command 를 생성하고 Invoker 에 전달
2. Invoker 는 Command.excute( ) 호출
3. Command 가 내부적으로 Receiver 의 동작을 호출

```
tv = TV()
tv_cmd = TVonCommand(tv)

remote_controller = Invoker()
remote_controller.set_command(tv_cmd)

remote_controller.run()
```



Command 패턴 구조

Invoker -> Command Interface -> ConcreteCommand -> Receiver



[도전] Command 객체 만들기

- TextEditor 클래스를 만든다.
  write 기능 : 입력되는 text 를 덧붙인다.
  delete 기능 : 특정 개수만큼 맨 뒤에서부터 지운다.

- Invoker 에는 명령어를 여러 개 기록할 수 있다.

- WriteCommand은 undo 기능까지도 사용 가능하다

(hint. delete 이용하기)

```
class TextEditor:
    def __init__(self):
        self.text = ""

    def write(self, text):
        self.text += text

    def delete(self, count):
        self.text = self.text[:-count]

    def __str__(self):
        return self.text
```

WriteCommand 와 DeleteCommand 를 이용한 Command 패턴을 구현한다

감사합니다.