The Dependency Inversion Principle

앞서서 OCP와 LSP에 대해서 배웠습니다. OCP는 변경은 허용하되 수정을 막자는 원칙이였고, LSP는 베이스 클래스가 서브 클래스로 치환될 수 있다는 원칙이였습니다. 이 두 원칙을 엄격히 사용하는데서 비롯되는 구조에 대해 말씀드리겠습니다.

이 구조 자체가 원칙이 되며 이 이름은 The Dependency Inversion Principle이라고 합니다

소프트웨어, 뭐가 문제야?

우리는 소프트웨어를 만드는 개발자로서, 우리 스스로 나쁜 디자인으로 내몰고 있습니다. 왜 이런 일이 일어날까요? 이 문제의 핵심은 바로 **bad design**을 정의를 하지 않았다는 것에 있습니다.

따라서 나쁜 디자인에 대해 설명드리겠습니다.

Bad Design

Bad Design의 정의

사람들마다 무엇이 나쁜 디자인인지는 다 다를 수 있습니다. 하지만, 로버트 마틴이 말하는 바에 따르면 소프트웨어가 다음 기준에 하나라도 해당한다면 나쁜 디자인임을 대다수가 공감할 수 있을 것이라 하였습니다.

이 특성들이 없다고해서 나쁜 디자인인 것은 아니나, 이 특성들이 있다면 나쁜 디자인이라 할 수 있습니다.(나쁜 디자인이기 위한 충분조건.)

1. 소프트웨어를 변경한 경우, 시스템의 수많은 다른 부분에 영향을 주기 때문에 변경하기 어려움 **[Rigidity]**

2. 소프트웨어를 변경했을 때, 시스템의 예상치못한 부분이 고장 **[Fragility]**

3. 현재 어플리케이션에서 필요한 부분을 분리하기 어려워서 다른 어플리케이션에 재사용하기 어려움 **[Immobility]**

앞으로 이 세가지 지표를 자세히 설명드려서 나쁜 디자인에 대해 말씀드리겠습니다.

Bad Design의 원인

원인은 상호의존성

상호 의존성이 경직하고, 깨지기쉽고, 이동불가능한 소프트웨어를 만듭니다.

상호의존성이 어떻게 Bad Design의 3가지 특성을 만드는가?

1. Rigidity

상호의존성있는 모듈에서 단 하나의 변화가 폭포수(cascade) 변화를 일으킵니다.

이 변화의 정도는 디자이너나 유지보수하는 사람들이 예측할 수 없습니다. 따라서 변경했을 때 비용도 예측할 수 없게 되죠. 따라서 관리자는 변경승인하기 꺼려질 것이며 결국 이는 공식적으로 rigid 디자인이 됩니다.

2. Fragility

변화가 일어났을 때 프로그램이 많은 부분에서 고장나는 경향을 Fragility라 부릅니다. **상호의존성이 강하여 개념적으로 전혀 상관없는 부분**이 변했을 때 새로운 문제가 발생합니다. 이 요소는 디자인의 신뢰도가 하락하고, 매니저가 품질을 예측하기 어렵게 합니다.

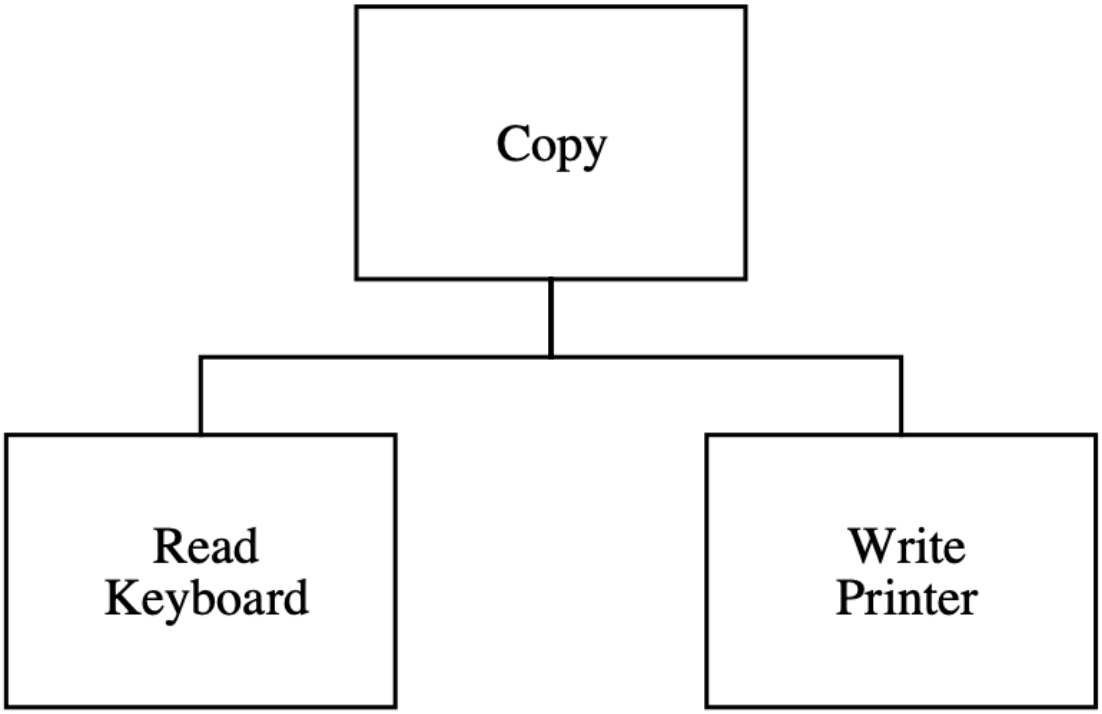
3. Imobility

디자인에서 원하는 부분이 **원하지 않는 다른 부분과 상호의존성이 강한 것**을 의미합니다. 대부분의 경우에선 떼어내는 비용이 재개발하는 비용보다 높기 때문에 결국 사용하지 못하게 됩니다.

예제: Copy Program

강한 의존성을 지닌 예제

Figure 1. Copy Program.



상황

키보드에 타이핑되는 것을 **복사**하여 프린터기로 출력하는 예제입니다. [Figure 1]은 이 상황을 구조도를 나타내준 그림입니다.

문제: 핵심 정책(policy)을 재사용할 수 없다

```
void Copy() {
  int c;
  while ((c = ReadKeyboard()) != EOF)
   WritePrinter(c);
}
```

Copy는 위 코드처럼, ReadKeyboard함수로 키보드에서 입렵된 문자를 복사한 후, 그 문자를 WriterPrinter 함수를 통해 프린터로 출력하고 있습니다.

낮은 수준 모듈인 keyboard나 printer는 서브루틴 라이브러리를 통해 재사용할 수 있는 반면, **Copy**에서 두 함수가 각각의 **특정 키보드**와 **특정 프린터**를 호출하기 때문에 Copy는 **다른 입력장치와 출력장치에선 재사용할 수 없습니다.**

하지만 우습게도 이 Copy가 우리가 재사용하고 싶은 가장 관심있는 정책을 지니고(encapsulate) 있습니다. 로버트 마틴은 이러한 경우가 매우 부끄러운 일이랍니다. Copy라는 지능을 가진 시스템을 재사용할 수 없으니까요.

다른 장치에 출력하고 싶다면?

만약에 키보드에 입력된 값을 disk file로 넘겨주는 프로그램을 생각해보겠습니다.

분명 Copy모듈을 재사용하고 싶을 것입니다. [왜? 이것이 **우리가 필요한 상위 정책**이기 때문에요.]

하지만 Copy모듈이 WritePrinter로 인해 Printer에 의존하고 있어 재사용할 수 없게 됩니다.

```
enum OutputDevice {printer, disk};

void Copy(outputDevice dev) {
  int c;
  while ((c = ReadKeyboard()) != EOF)
    if (dev == printer)
      WritePrinter(c);
    else
      WriteDisk(c);
}
```

만약에 if로 Disk에 쓰는것을 분기처리해봅시다.

결국 이는 또 다른 상호 의존성을 생성할 뿐입니다. 시간이 흐르면 Copy모듈은 더 많은 하위 모듈에 상호의존하게 될 것이고 결국 견고하고, 고장나기 쉬운 시스템이 됩니다.

그래서 답은 Dependency Inversion

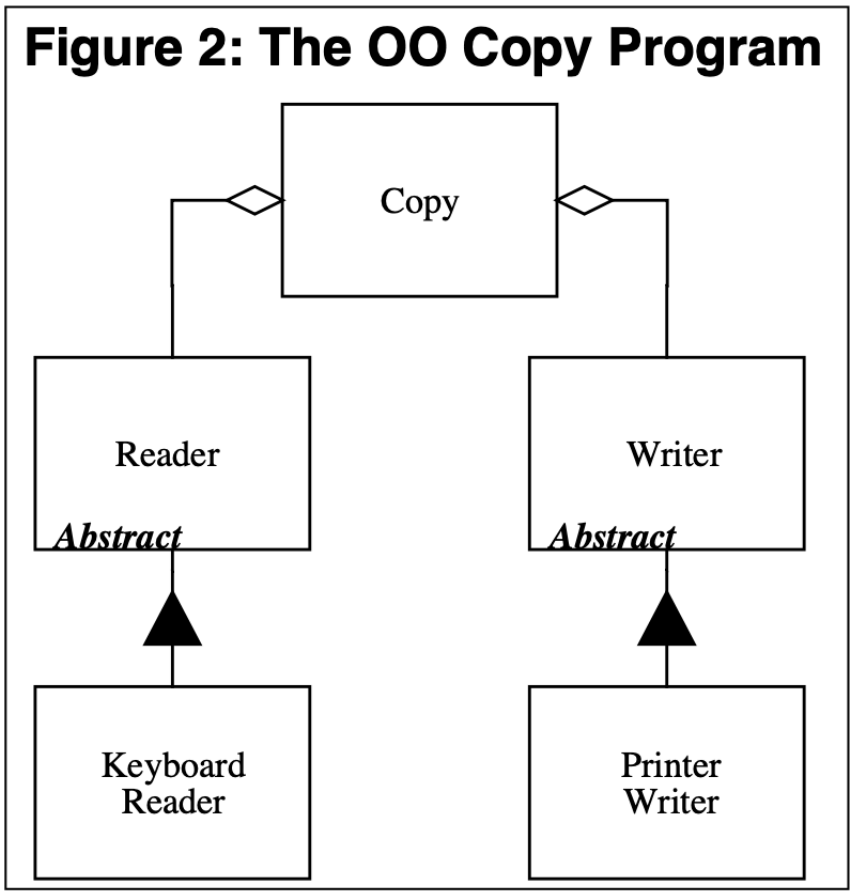
위 문제를 다음과 같이 말할 수 있습니다.

상위 레벨 정책을 가진 모듈이 하위 레벨의 구체적인 모듈에 의존하고 있기 때문에 문제가 발생했다

따라서 Copy모듈을 세부 모듈로부터 독립하게 해준다면 Copy는 재사용할 수 있게 될 것입니다.

따라서 이 문제는 다음처럼 해결 할 수 있습니다.

1. Copy모듈이 추상화된 Reader와 Writer에 의존한다.
2. Keyboard는 Reader를 상속하여 의존한다., Printer 또한 Writer를 상속하여 의존한다.




```
class Reader {
  public:
   virtual int Read() = 0;
};

class Writer {
  public:
   virtual void Write(char) = 0;
};

void Copy(Reader& r, Writer& w) {
  int c;
  while((c=r.Read()) != EOF)
  w.Write(c);
}
```

이점

이렇게 설계한다면, Copy는 추상클래스에 의존하게 되고, 특정한(detail) reader와 writer또한 동일한 추상 클래스에 의존하게 됩니다.

1. 결국 Copy클래스는 특정 클래스(detail class)의 의존성으로부터 벗어나 추상클래스에만 의존하기 때문에 재사용이 가능해지며,
2. 새로운 reader와 writer를 만들어준다 하더라도 Copy는 애당초 추상클래스에 의존했기 때문에 이 특정 클래스에게 독립하게 됩니다.
    - ocp에 배운 것처럼 fix된 Abstract에 의존하므로! 변경되지 않음.

이렇게 상호의존성이 사라짐으로써 fragile하고 rigid하지 않게 되며, 다양한 문맥에서 Copy를 사용할 수 있기 때문에 mobile하게 됩니다.

The Dependency Inversion Principle

정리

A. HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS

고수준 모듈은 저수준 모듈에 의존해서는 안되고, 둘은 추상에 의존해야 한다.

B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS

추상은 구체적인 것에 의존하면 안된다. 구체적인 것은 반드시 추상에 의존해야 한다.

Inversion의 의미

절차지향적 또는 구조적 프로그래밍같은 종래의 소프트웨어 개발 방법에선 고수준 모듈이 저 수준 모듈에 의존하는 경우가 많습니다.(Figure 1처럼)

따라서 종래의 디자인의 반전을 띄기 때문에 Inversion이란 의미를 붙였습니다.

이런 의미로 또 Framework design의 핵심 원리가 됩니다.

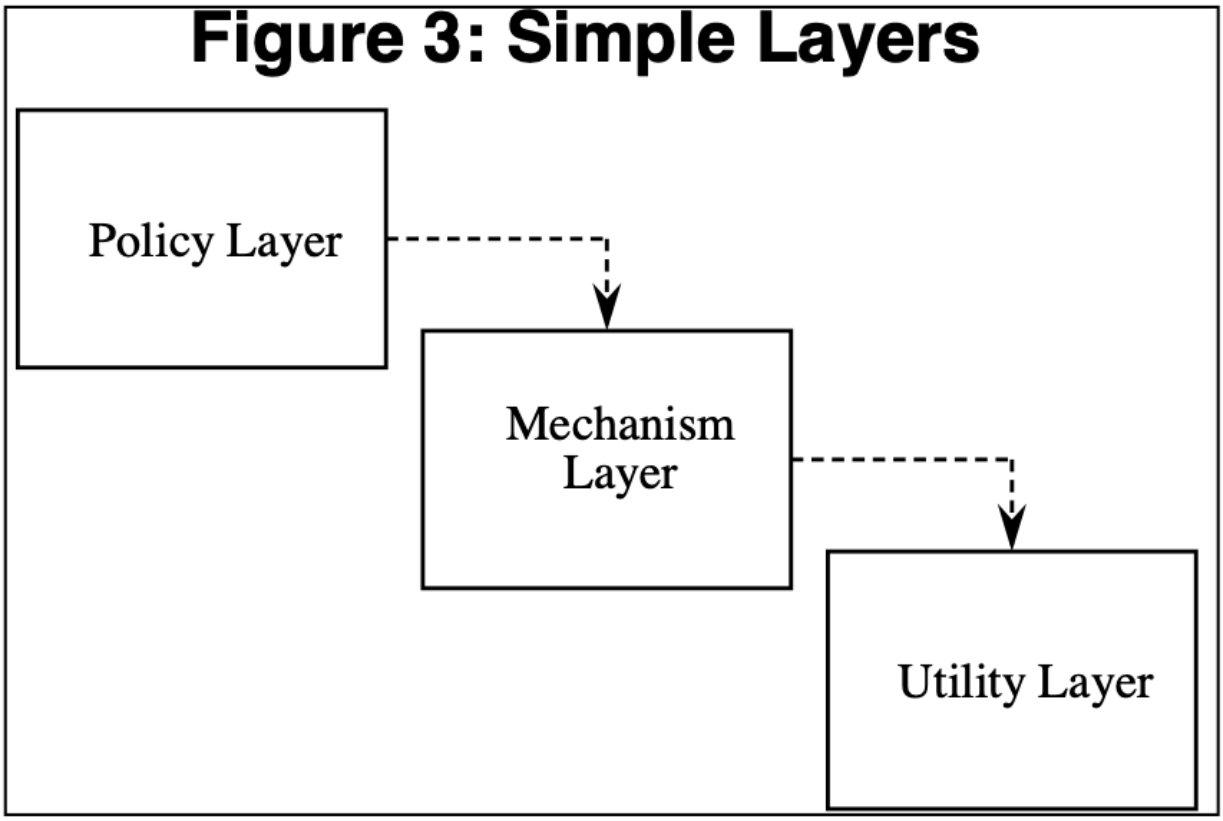
Layering이 좋다?

통념

booch라는 사람이 다음과 같이 말을 했습니다.

객체지향적으로 잘 짜진 구조는 layer가 명확히 정의되어 있으며, 각 레이어는 잘 정의된 인터페이스를 통해 일관된 서비스를 각 레이어게 제공할 수 있습니다.

하지만 사람들이 이 말을 곧이 곧대로 받아들여서 [Figure 3]과 같은 구조를 만들죠



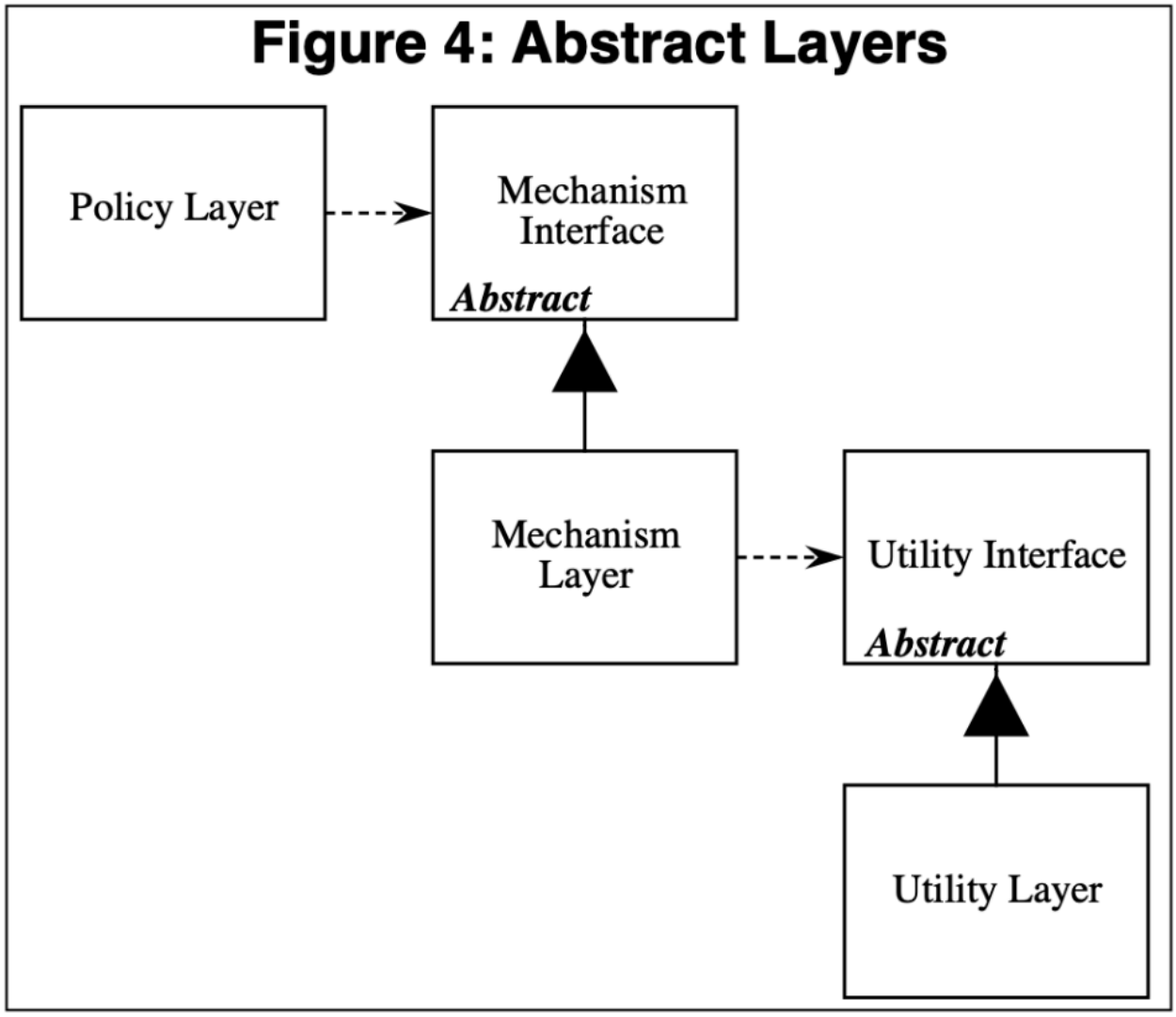
단점

그런데 이렇게 구조를 만들면 **의존성이 Utility Layer까지 전이**되버립니다!

Policy는 Mechanism Layer에 의존하고, Mechanism Layer는 Utility Layer에 의존하기 때문에 Utility Layer의 영향이 Policy Layer까지 미치게 되는 것이죠.

따라서 답은!

Abstract Layers



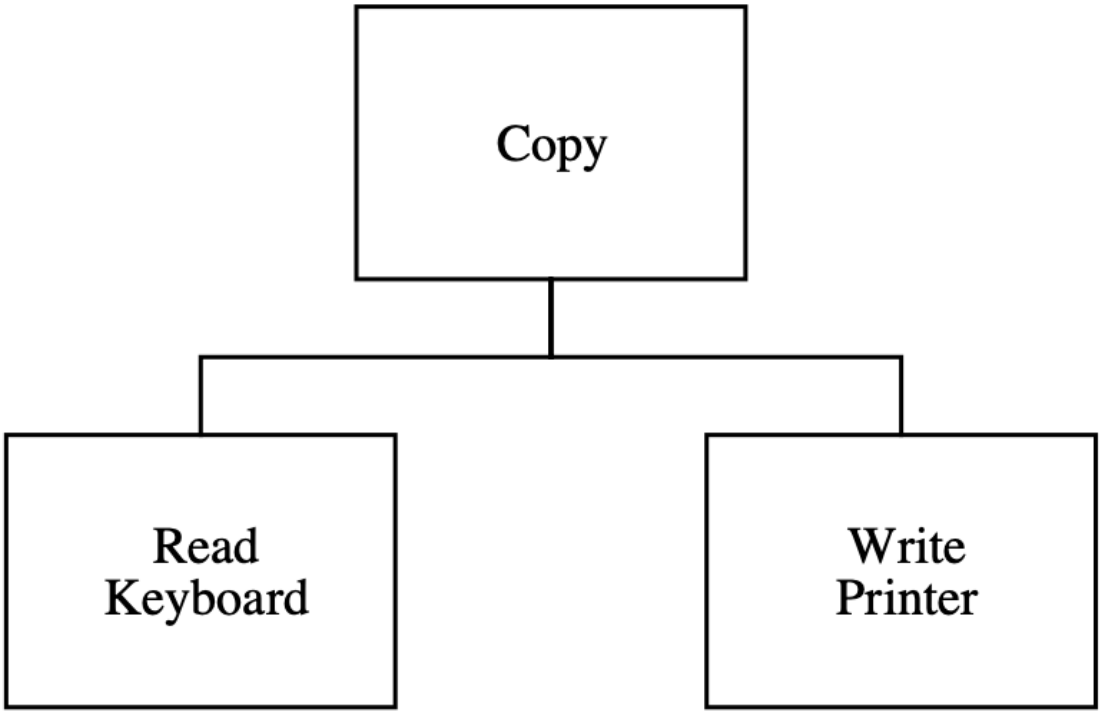
이처럼 레이어 별로 Interface를 두면 추상클래스에 의존하기 때문에 각 레이어는 독립적으로 유지하게 됩니다.

그렇다면, DIP에 따라 저수준 모듈이 추상에, 고수준 모듈이 추상에 고수준 모듈에 의존하도록 만들어야 하는데, 어떻게 무엇이 고수준 모듈인지 판단할 수 있을까요?

Finding the Underlying Abstraction

핵심은 구체적인 모듈이 변경될 때 변하지 않는 추상(Abstraction)

Figure 1. Copy Program.



이 예제에서 보시면 아무리 Reader와 Writer가 변하더라도 Copy의 기능은 변하지 않습니다. 다시 말해 입력장치로부터 입력된 데이터를 복사하여 출력장치로 데이터를 보내는 내재화된 추상은 변하지 않습니다.(underlying abstraction)

따라서 high level policy가 Copy라 말할 수 있게 되는 것입니다!

요약

OCP를 적용하기 위해, DIP에 따른 구조를 선택하고, DIP에서 추상화를 할 때 LSP를 따른다.