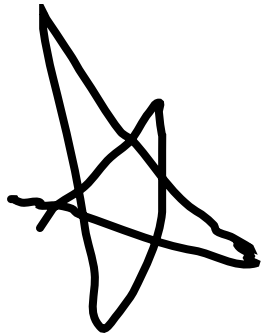
KB금융그룹 | 국민의 평생 금융파트너



2025년 상반기 K-디지털 트레이닝

객체지향 설계원칙-SOLID

[KB] IT's Your Life

KB 국민은행

SOLID 설계 원칙

- 단일 책임 원칙(Single Responsibility Principle: SRP)
- 개방-폐쇄 원칙(Open-Closed Principle: OCP) 확장에는 열려있고 변화에는 닫
- 리스코프 치환 원칙(Liskov Substitution Principle: LSP) 부모타입으로 참조되어있을 때 자식으로 치환해서 사용할 수 있어야 한다
- 인터페이스 분리 원칙(Interface Segregation Principle: ISP)
- 의존 역전 원칙(Dependency Inversion Principle: DIP)

1 단일 책임 원칙

- **객체 지향의 기본**
  - 책임을 객체에게 할당

- **클래스는 단 한 개의 책임을 가져야 한다.**
  - 클래스가 여러 책임을 갖게 되면 그 클래스는 각 책임마다 변경되는 이유가 발생
  - 클래스가 한 개의 이유로만 변경되려면 클래스는 한 개의 책임만 가져야 함

→ 클래스를 변경하는 이유는 단 한 개여야 한다.

단일 책임 원칙

단일 책임 원칙 위반이 불러오는 문제점

```
public class DataViewer {

    public void display() {
        String data = loadHtml();
        updateGui(data);
    }

    public String loadHtml() {
        HttpClient client = new HttpClient();
        client.connect(url);
        return client.getResponse();
    }

    private void updateGui(String data) {
        GuiData guiModel = parseDataToGuiData(data);
        tableUI.changeData(guiModel);
    }
    private GuiData parseDataToGuiData(String data) {
        … // 파싱 처리 코드
    }
    … // 기타 필드 등 다른 코드
}
```

DataView는 화면 출력이 목표

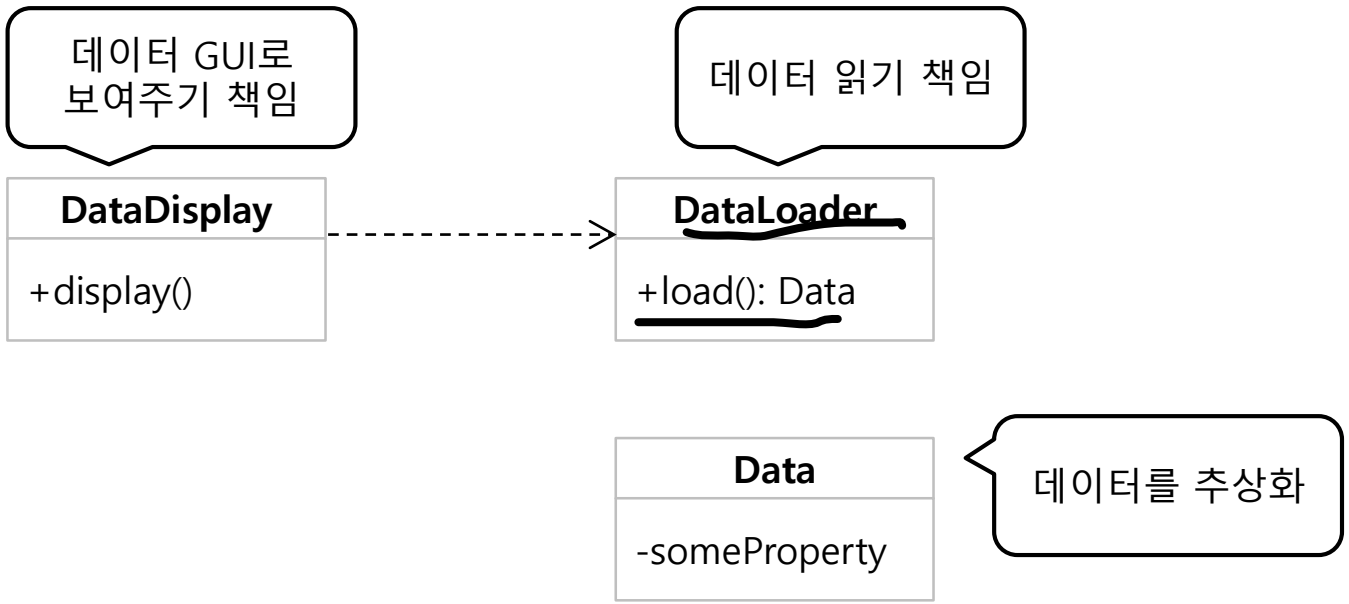
하지만 웹에서 데이터를 읽어오는 거까지 함.

요구사항이 생길 때마다 수정사항이 발생함.

단일 책임 원칙

해결 책!

책임의 분리

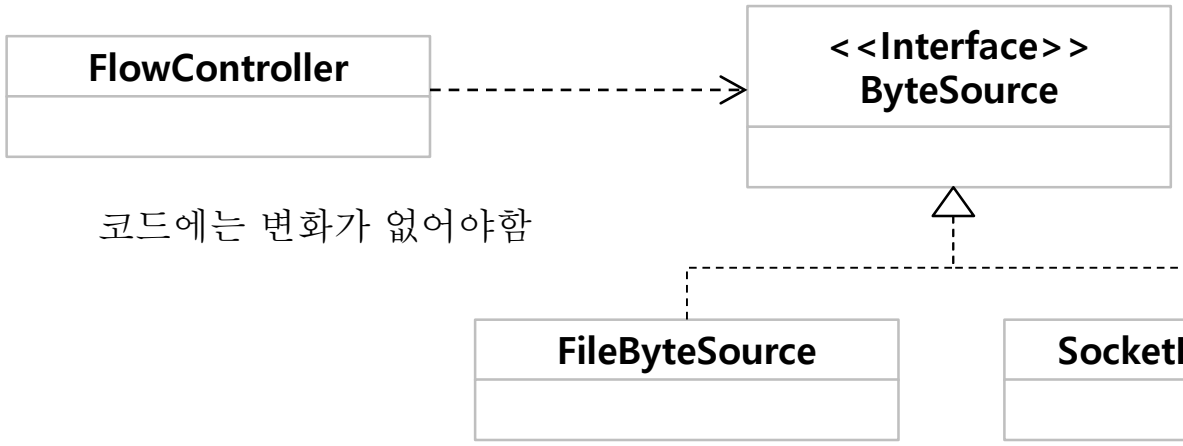


개방 폐쇄 원칙

개방 폐쇄 원칙(Open-Closed Principle)

- 기능확장에는 열려 있고, 변화에는 닫혀 있어야 함
- 추상화와 다형성(상속)을 이용해서 구현

변화는 코드 변화를 의미

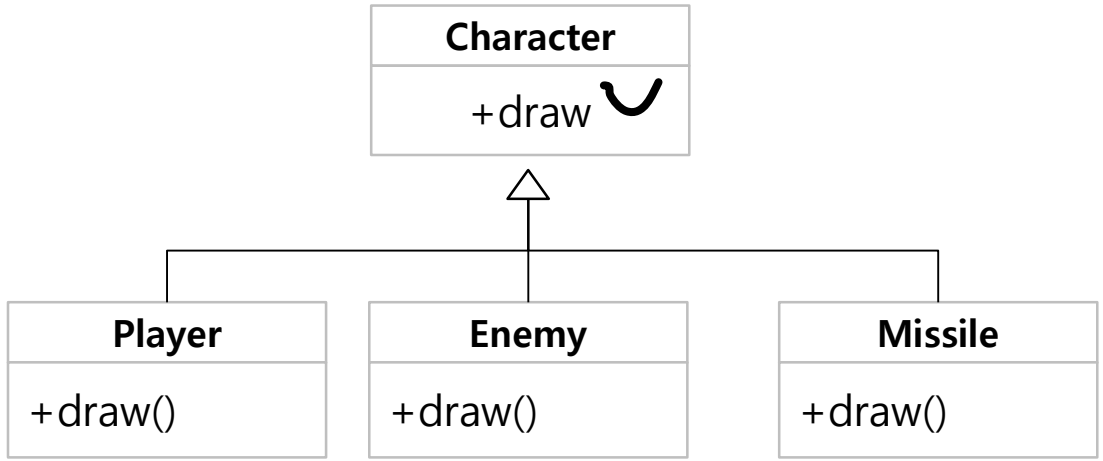


코드에는 변화가 없어야함

자식이 확장되어도

개방 폐쇄 원칙

개방 폐쇄 원칙(Open-Closed Principle)



상속과
다형성 의 열매

```
public void drawCharacter(Character character) {
   character.draw();
}
```

메소드 정의시도
set(int age, string name)
보다는
set(User user)
가 더 OCP를 잘 갖춰죠?

나중에 필요 데이터가 늘어나도
set(User user)형식이 덜 바뀌죠?

검사 코드 활용시에도 하드코딩은 하지마. 그리고 검사코드는 객체안에

if(user.getAge()>19){.//}보다는
if(user.isAdult())가 좋다.

=> 요구 사항이 바뀌었을 때 코드 수정이 최소게끔.

리스코프 치환 원칙

리스코프 치환 원칙(Liskov Substitution Principle)

- 상위 타입의 객체를 하위 타입의 객체로 치환해도 상위 타입을 사용하는 프로그램은 정상적으로 동작해야 한다.
  - 개방 폐쇄 원칙을 받쳐 주는 다형성에 관한 원칙을 제공

```
public void someMethod(SuperClass sc) {
   sc.someMethod();
}
```

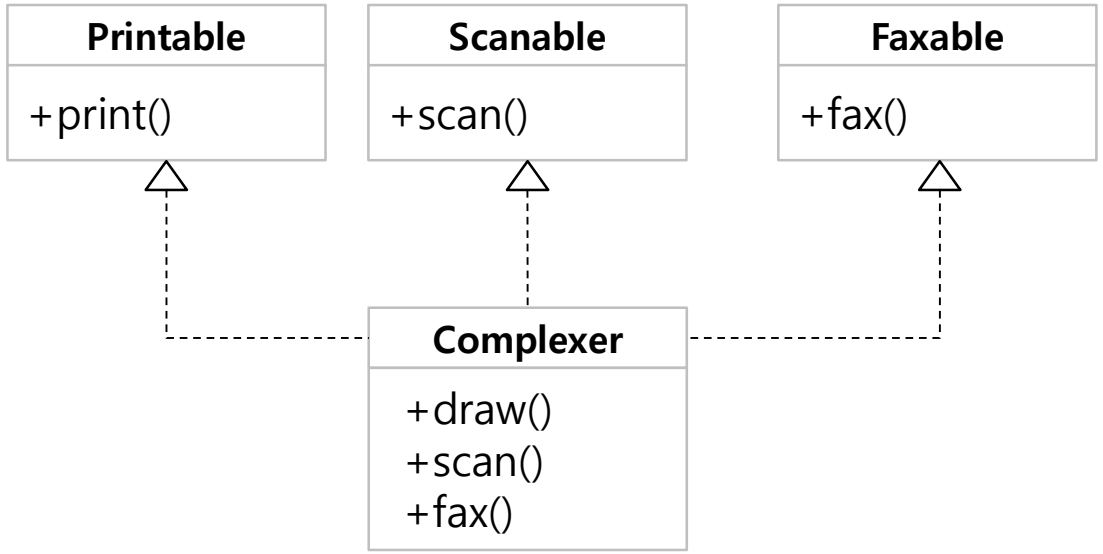
```
someMethod(new SubClass());
```

인터페이스 분리 원칙

인터페이스 분리 원칙(Interface Segregation Principle: ISP)

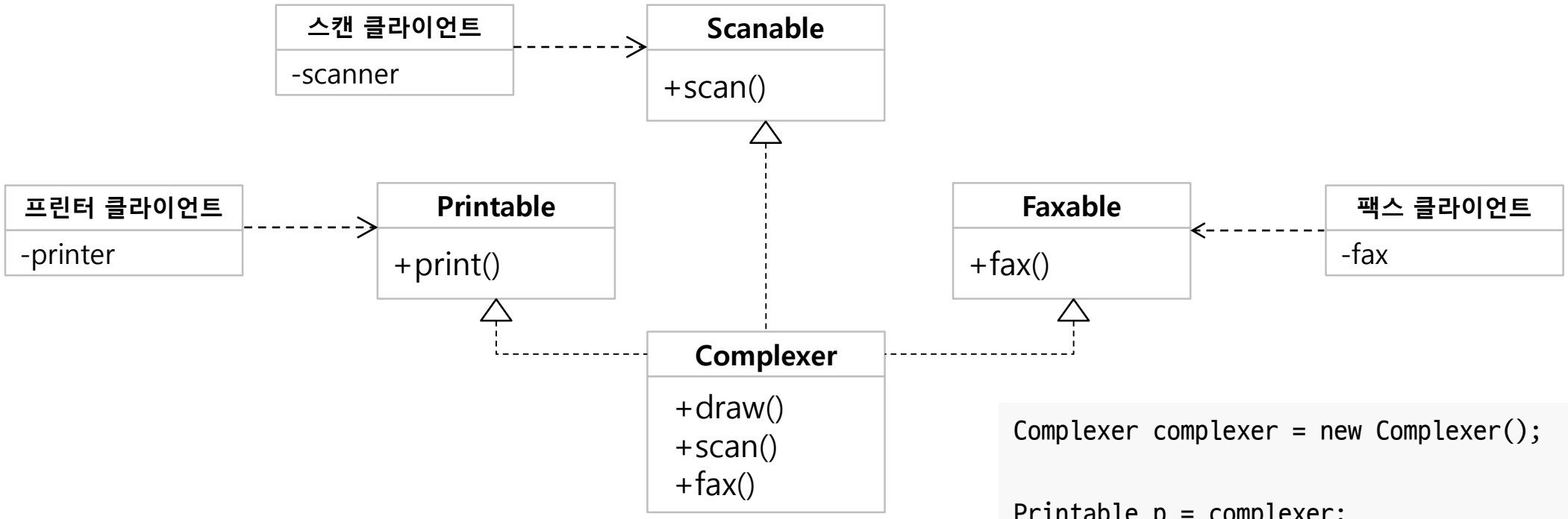
- 클라이언트는 자신이 사용하는 메서드에만 의존해야 한다.

→ 인터페이스는 그 인터페이스를 사용하는 클라이언트를 기준으로 분리해야 한다.



인터페이스 분리 원칙

인터페이스 분리 원칙(Interface Segregation Principle: ISP)




```
public class Complexer implements Printable, Scanable, Faxable {
    …
}
```

```
Complexer complexer = new Complexer();

Printable p = complexer;
p.print();

Scanable s = complexer;
s.scan();

Faxable f = complexer;
f.fax();
```

의존 역전 원칙

스프링에서 늘 하는 것

의존 역전 원칙(Dependency Inversion Principle)

- **고수준 모듈은 저수준 모듈의 구현에 의존해서는 안된다. 저수준 모듈이 고수준 모듈에서 정의한 추상 타입에 의존해야 한다.**
  - 고수준 모듈: 어떤 의미 있는 단일 기능을 제공하는 모듈
  - 저수준 모듈: 고수준 모듈의 기능을 구현하기 위해 필요한 하위 기능의 실현 구현으로 정의

필요한 객체가 있으먄 외부에서 제공 주입 전달 해주겠다.
=> loosly coupled하게 코드 짜라

고수준 모듈

바이트 데이터를 읽어와 암호화하고 결과 바이트 데이터를 쓴다.

저수준 모듈

파일에서 바이트 데이터를 읽어온다.

AES 알고리즘으로 암호화한다.

파일에 바이트 데이터를 쓴다.

클래스 만들때 필드 지정시
하드 코드로 의존 객체를 만들지 않고

생성자 메서드(예를 들어 setter)로
외부에서 사용할 필드 멤버를 주입 전달 제공 하게끔
만들어라!!!!

의존 역전 원칙

클래스로만 설계한 경우

의존 역전 원칙(Dependency Inversion Principle)

- 고수준 모듈로 프래그램을 만들어 두고,
- 런타임시에 구체적인 저수준의 모듈을 결정해서 전달할 수 있음.

SOLID 많은 경험 필요.
항상 염두하며 지키려 하세요

"programing to interface"
"인터페이스로 프로그래밍 해라"

=>

다형성을 가지기 위해서

=>

인터페이스로 프로그래밍을 하게 되면 자연스럽게 solid 지키게 됨!

상속보다는
위임이 코드 재사용성에 가깝다
위임 예

```
public class MyController {

    //has a 관계
    MyRepository myRepository;

    public void save() {
        //myRepository에게 save 권한을 위임에서 처리
        myRepository.save();
    }
}
```