고급객체지향 프로그래밍
강의노트 #02

SOLID 원칙

조용주
ycho@smu.ac.kr

SOLID (S.O.L.I.D.)

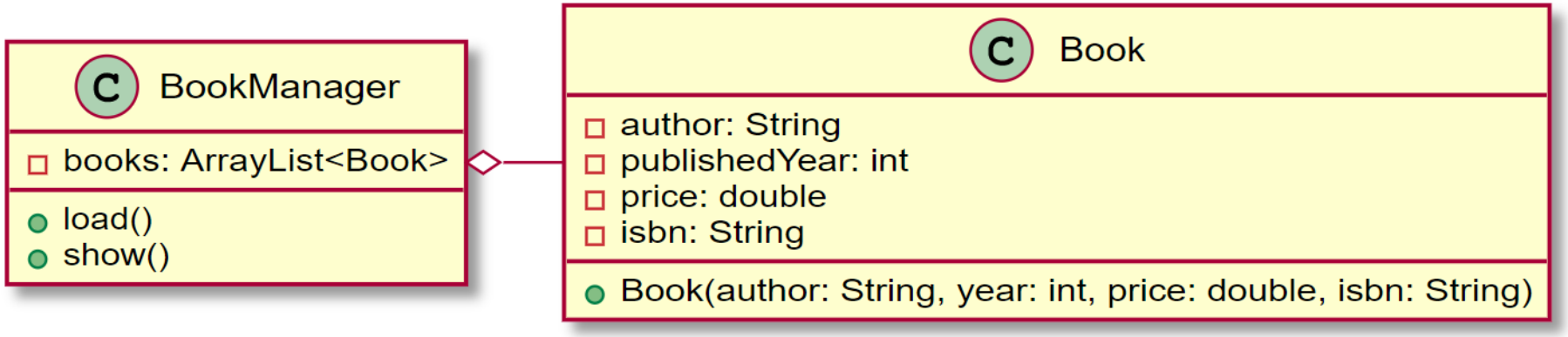
- Robert C. Martin 이 2000 년에 논문에서 발표한 내용
  - 이해하기 쉽고, 유연하며, 유지 보수가 쉬운 소프트웨어

약어	원칙	한글 명칭
SRP	Single Responsibility Principle	단일 책임 원칙
OCP	Open-Closed Principle	개방 - 폐쇄 원칙
LSP	Liskov Substitution Principle	리스코프 치환 원칙
ISP	Interface Segregation Principle	인터페이스 분리 원칙
DIP	Dependency Inversion Principle	의존 역전 원칙

Single Responsibility Principle

- *Gather together those things that change for the same reason, and separate those things that change for different reasons.*
  - *클래스가 변경되어야 하는 이유는 한 가지로만 구성*
  - *클래스의 역할을 한 가지로 구성*
  - *사용자와의 관계에 대해서 고민해야 함*
- 문제점
  - 클래스에 기능이 너무 많으면 유지 보수가 어려워짐

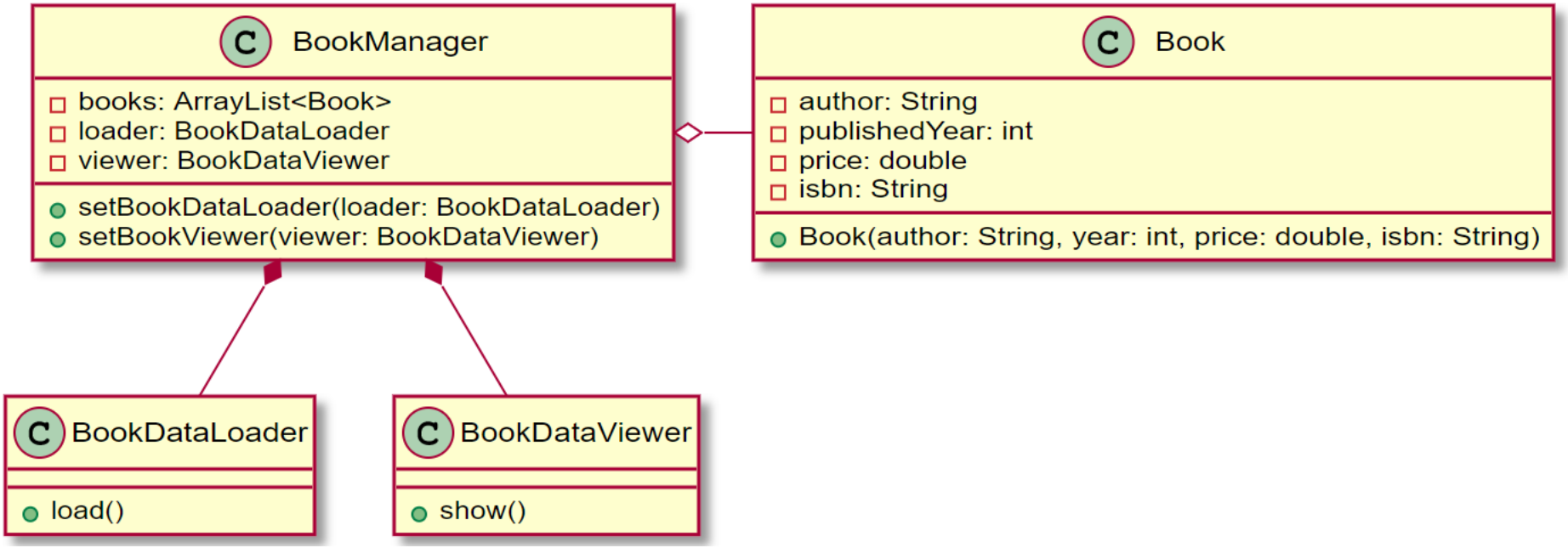
Single Responsibility Principle



- 예 :
  - Book 클래스 메소드 기능 설명
    - load()
      - 파일에서 Book 정보를 읽어서 멤버 변수들에 저장
    - show()
      - 콘솔 화면에 해당 객체의 정보를 보임
  - 프로그램을 더 이상 고치지 않는다면 이대로 SRP 를 지키는 설계가 될 수 있음

Single Responsibility Principle

- 변경되는 부분들이 생긴다면 SRP 를 다시 고려해봐야 함
  - 만약 파일이 아니라 데이터베이스에서 책 데이터를 읽어서 저장하는 load() 함수를 만든다면 ?
  - 콘솔 화면이 아니라 GUI(Graphical User Interface) 화면에 책 내용을 출력하는 show() 함수를 만든다면 ?



Open-Closed Principle

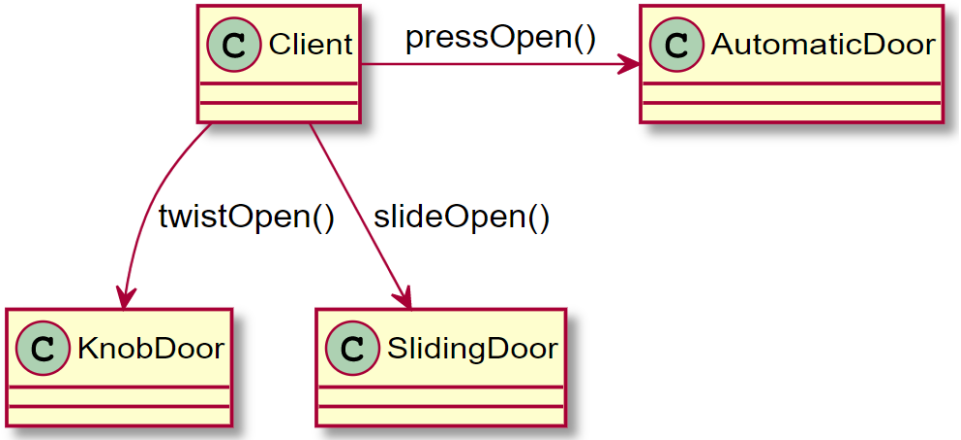
- *A module should be open for extension but closed for modification*
  - *기존 코드를 변경하지 않고 확장할 수 있도록 만들어야 함*
- 예
  - 문 (door) 을 여는 프로그램을 가정
  - 세 가지 종류의 문이 있음
    - Sliding door – 미닫이문 (밀어서 열고 닫는 문)
    - Knob door – 손잡이가 있는 문
    - AutomaticDoor – 버튼식 자동문

Open-Closed Principle

- 버전 1
  - if 문으로 문의 종류에 따라 다르게 열도록 함

```
if (door instanceof AutomaticDoor)
    client.pressOpen(door);
else if (door instanceof KnobDoor)
    client.twistOpen(door);
else if (door instanceof SlidingDoor)
    client.slideOpen(door);
```

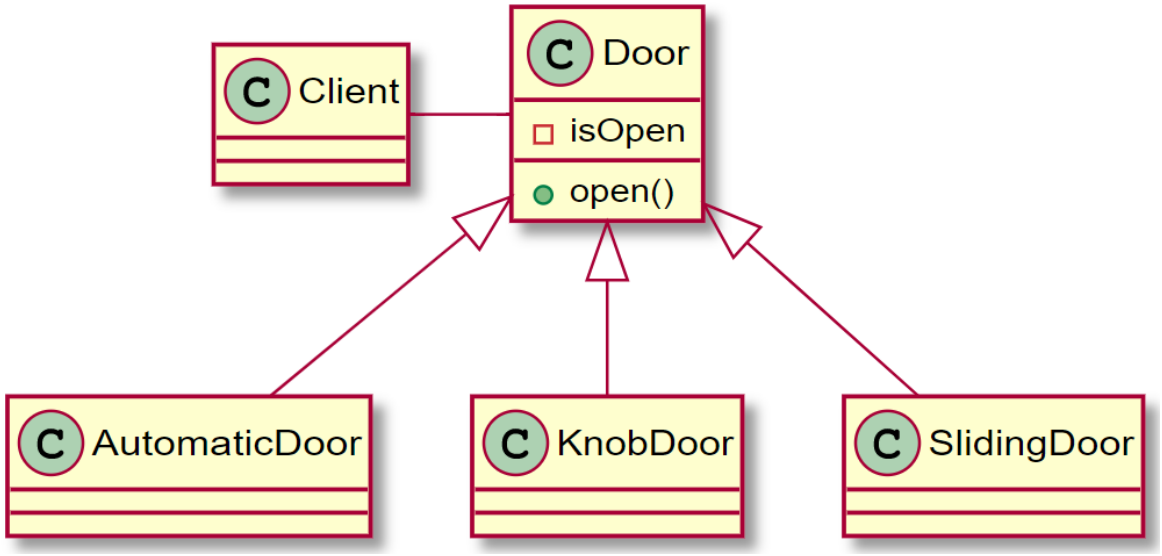
  - 새로운 문이 추가되면 코드 수정 불가피



Open-Closed Principle

- 버전 2
  - 다형성 사용

```
door.open();
```



  - 새로운 문이 추가되면 새로운 클래스 추가하고 open() 함수 오버라이딩

Open-Closed Principle

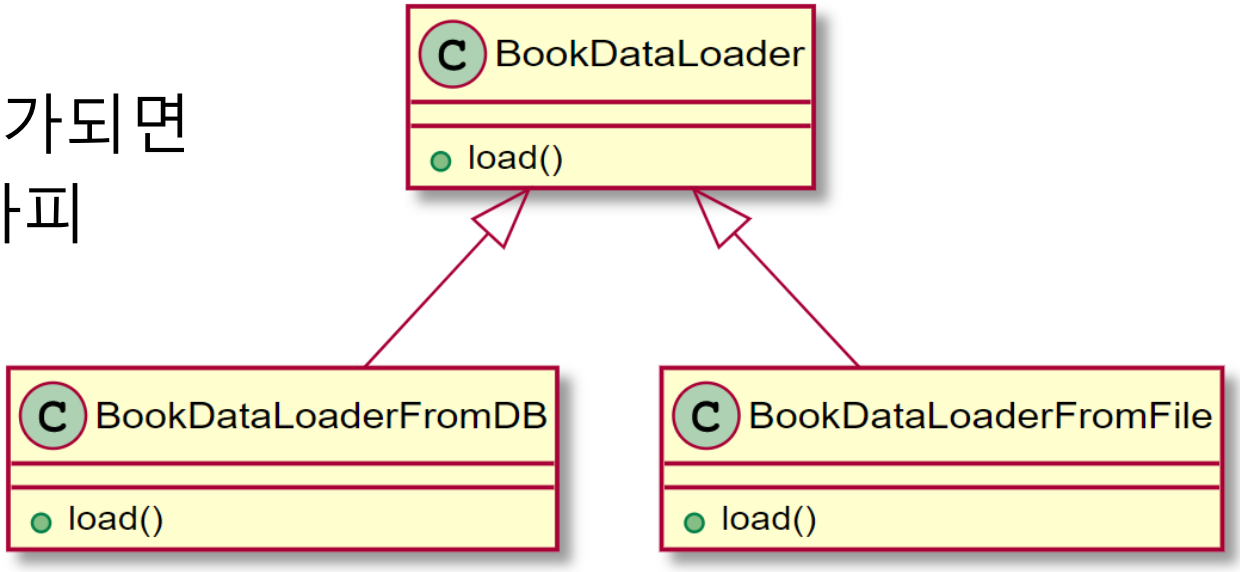
- 예
  - BookManager.load() 함수
  - BookDataLoader 클래스는 파일에서 데이터를 입력 받음
  - BookDataLoaderFromDB
  - 세 가지 종류의 문이 있음
    - Sliding door – 미닫이문 (밀어서 열고 닫는 문)
    - Knob door – 손잡이가 있는 문
    - AutomaticDoor – 버튼식 자동문

Open-Closed Principle

- 버전 1
  - if 문으로 문의 종류에 따라 다르게 열도록 함

```
if (loader instanceof BookDataLoaderFromFile)
    manager.loadFromFile(loader);
else if (loader instanceof BookDataLoaderFromDB)
    manager.loadFromDB(loader);
```

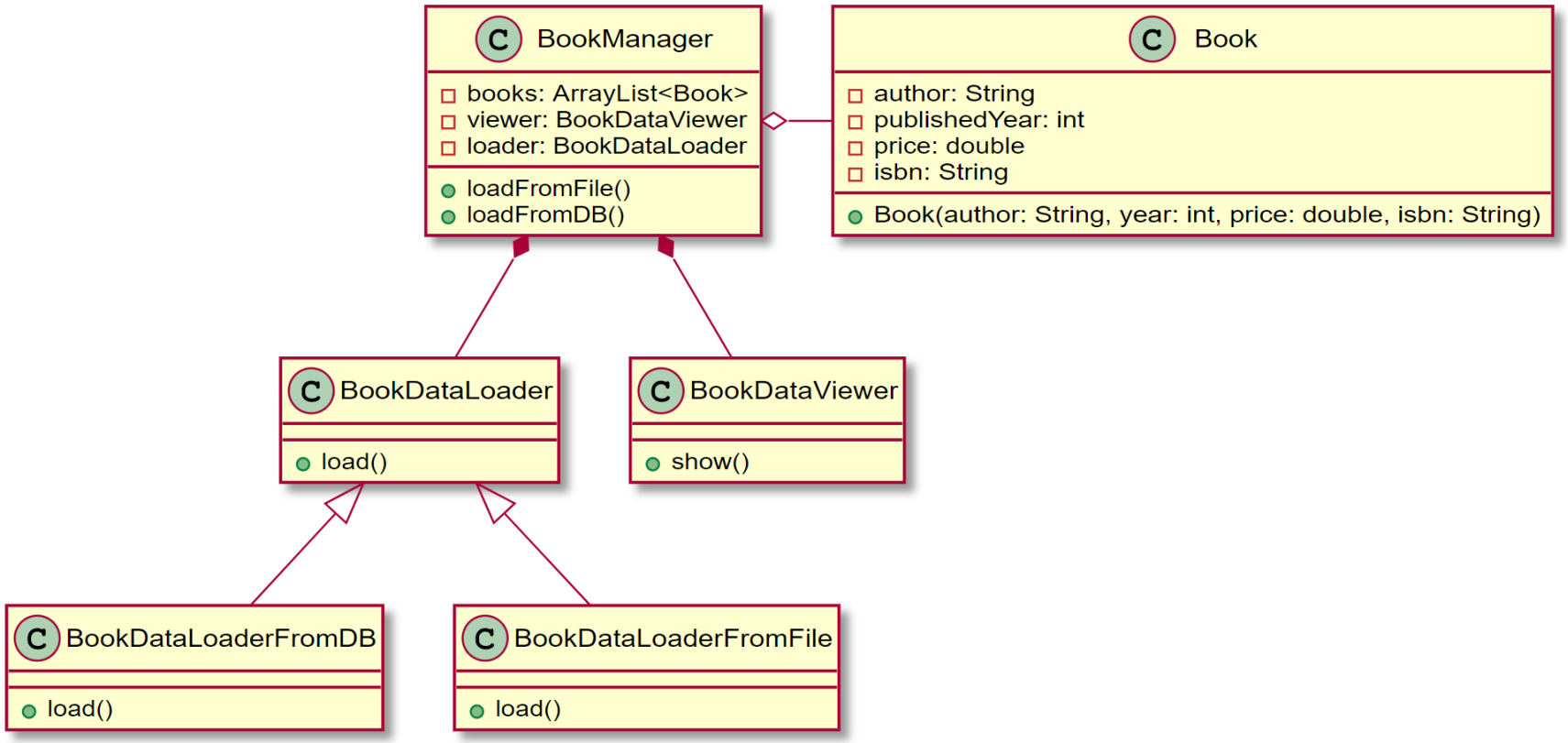
  - 새로운 문이 추가되면 코드 수정 불가피



OCP

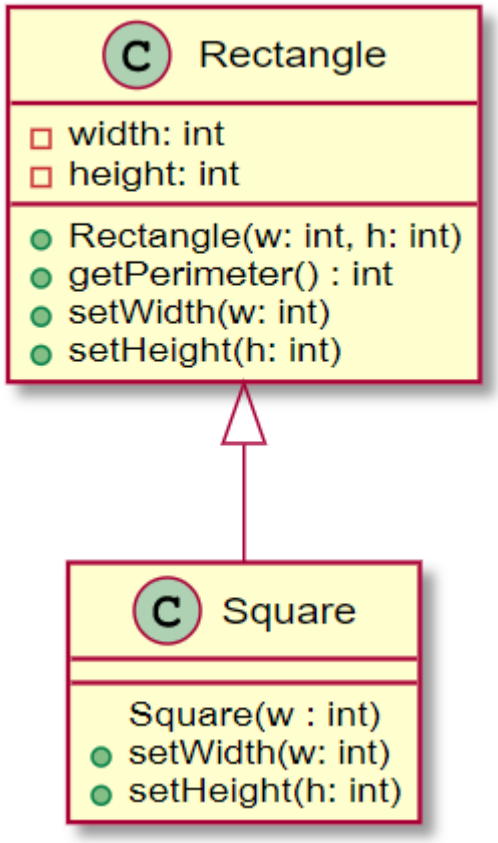
- 버전 2
  - 다형성 사용

```
loader.load();
```



Liskov Substitution Principle

- *Subclasses should be substituta ble for their base classes*
  - *자식 클래스는 부모 클래스를 대체할 수 있어야 함*
  - *부모 클래스 객체 대신 자식 클래스 혹은 후손 클래스 객체를 사용했을 때 문제없이 프로그램이 동작해야 함*
- 예 : 직사각형과 정사각형
  - 정사각형은 직사각형의 특별한 종류
  - 상속으로 처리



Liskov Substitution Principle

```
class Rectangle {
    private int width;
    private int height;

    public Rectangle(int w, int h) {
        width = w;
        height = h;
    }
    public int getPerimeter() {
        return 2 * (width + height);
    }
    public void setWidth(int w) { width = w; }
    public void setHeight(int h) { height = h; }
}
```

Liskov Substitution Principle

```
class Square extends Rectangle {
    public Square(int w) {
        super(w, w);
    }
    public void setWidth(int w) {
        super.setWidth(w);
        super.setHeight(w);
    }
    public void setHeight(int h) {
        super.setWidth(h);

        super.setHeight(h);
    }
}
```

Liskov Substitution Principle

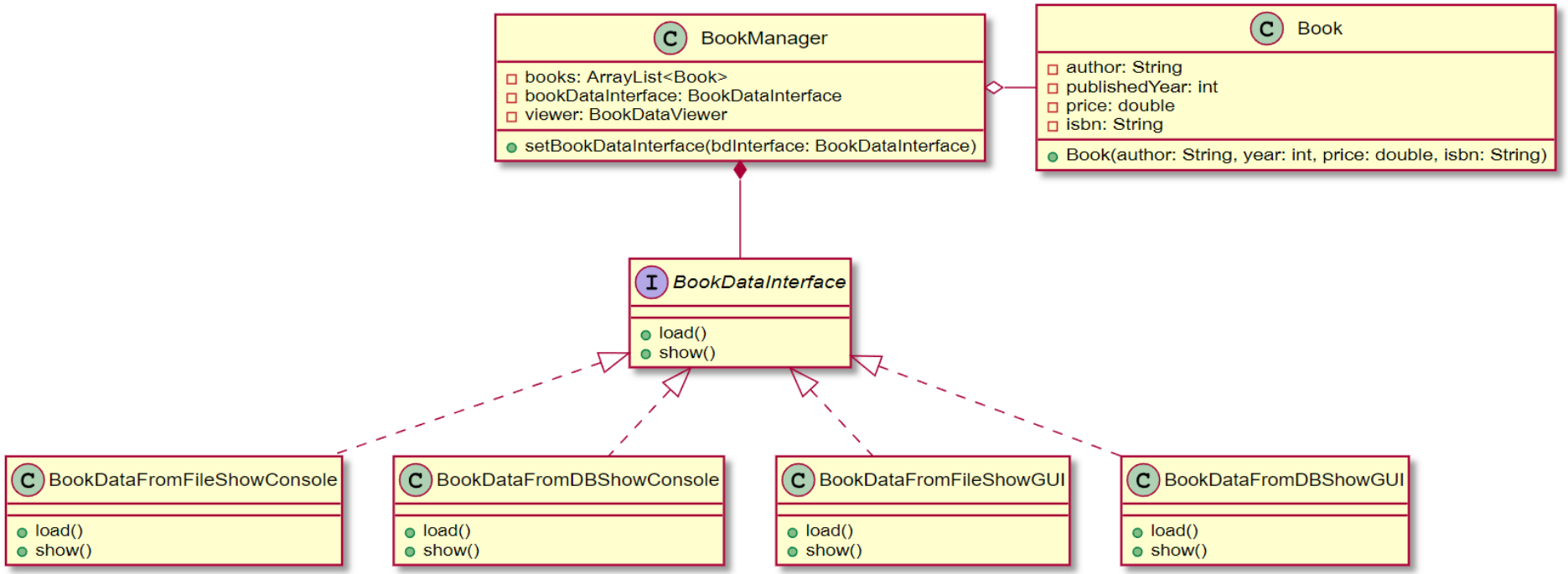
```
class Main {
    public static void main(String[] args) {
        Rectangle r = new Rectangle(3, 5);
        System.out.println(r.getPerimeter());
        Square s = new Square(3);
        System.out.println(s.getPerimeter());
        r = s;
        r.setWidth(3);
        r.setHeight(5);
        System.out.println(r.getPerimeter());

    }
}
```

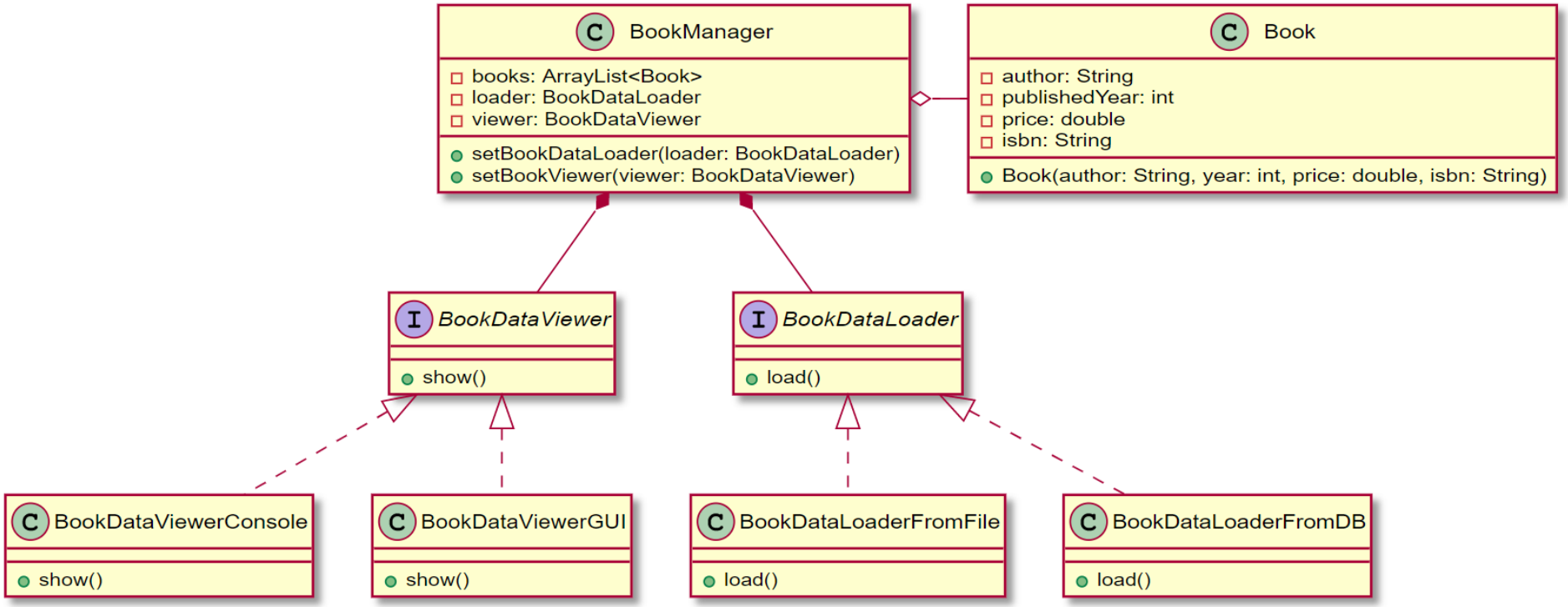
- Square 는 Rectangle 을 안전하게 대체 못함

Interface Segregation Principle

- *Many client specific interfaces are better than one general purpose interface*



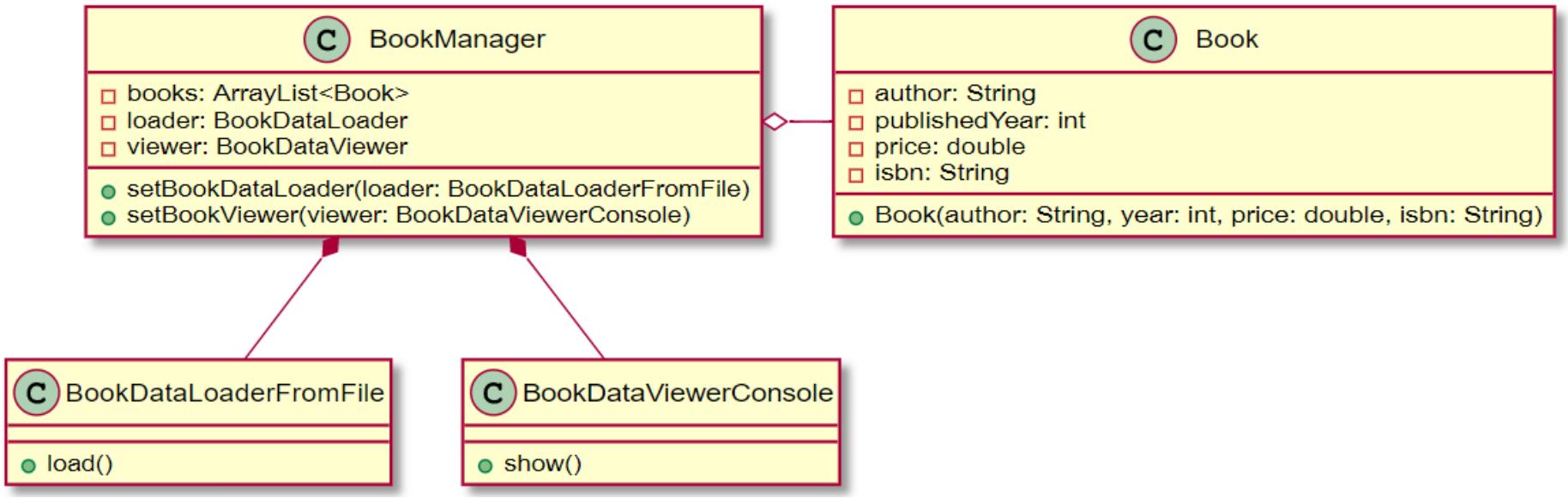
Interface Segregation Principle



Dependency Inversion Principle

- *Depend upon Abstractions. Do not depend upon Concretions*
  - *기능을 직접 구현한 구체 (구상) 클래스 (Concrete class) 또는 함수보다는 추상 클래스나 인터페이스를 사용하는 코드를 작성하라*
  - *기능을 직접 구현한 클래스나 함수는 변경될 가능성이 높음*

Dependency Inversion Principle



Dependency Inversion Principle