

SOLID Design Principles

514770-1

Fall 2025

9/10/2025

Kyoung Shin Park
Computer Engineering
Dankook University

S.O.L.I.D.: First 5 Principles of OOD

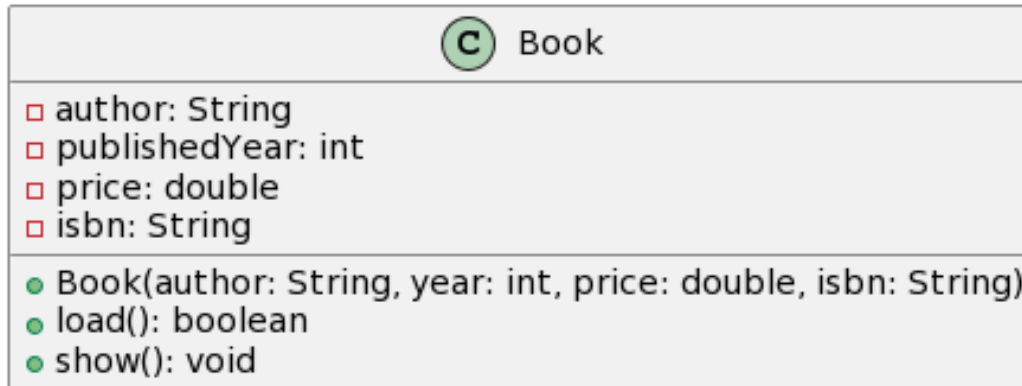
- Robert C. Martin collected 10 principles of Object Oriented Design (2000).
 - The first 5 principles - so called **SOLID** – deal with **the design of classes**. This principles is for easy-to-understand, flexible, and easy-to-maintain software development.

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

Single Responsibility Principle

- ❑ A class should only have a **single responsibility**. In other words, it should have only **one reason to change**.
- ❑ *Responsibility as a 'reason to change'*
- ❑ ***Gather*** together those things that change for the same reason, and ***separate*** those things that change for different reasons.
- ❑ If there are too many features in a class, it makes difficult to maintain.

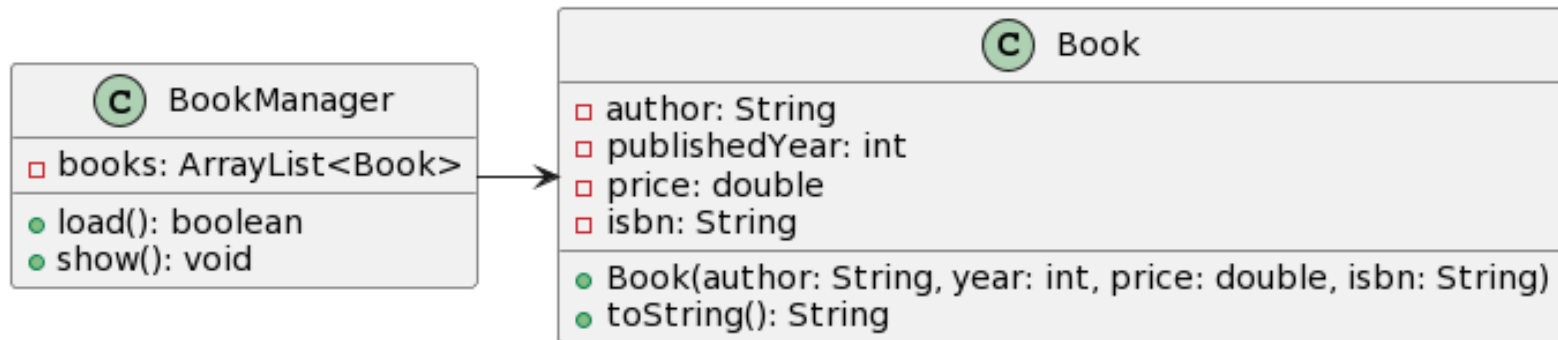
Single Responsibility Principle



■ Book class example

- `load()` reads the Book information and store it in member variables
- `show()` displays the Book information on the console screen

Single Responsibility Principle

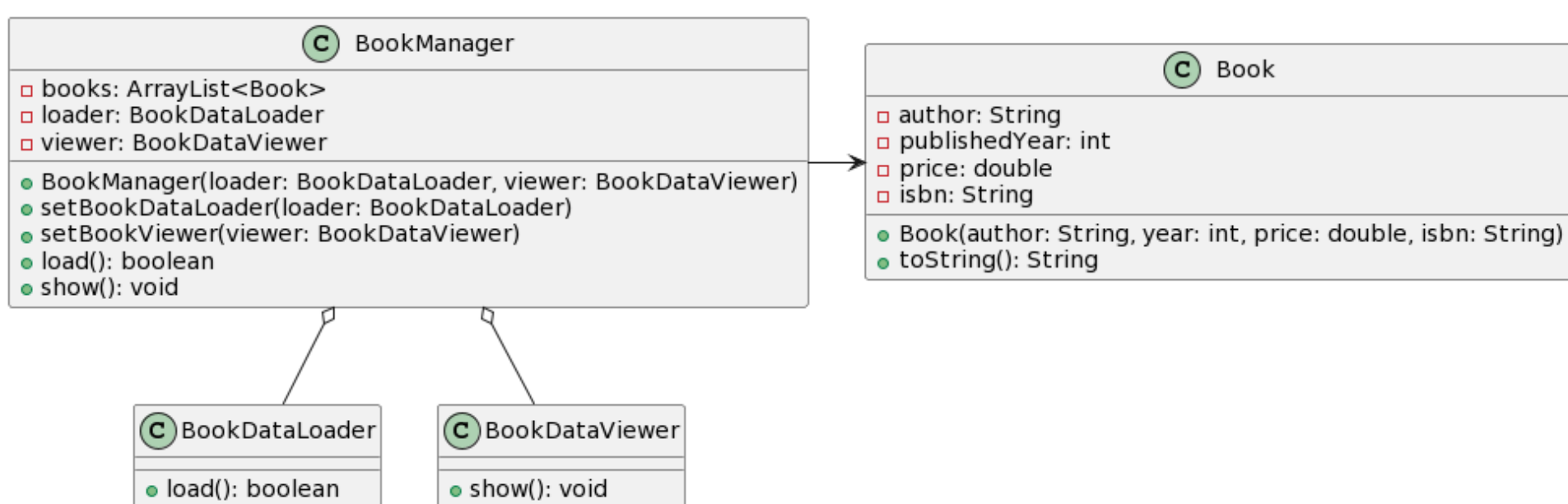


■ Book & BookManager class example

- Book remove load() and show()
- BookManager add load() & show()
 - load() reads the Book information from a file and store it in member variables
 - show() displays the books on the console screen
- If the program is no longer modified, this design keeps SRP.

Single Responsibility Principle

- However, if you add features or new behavior, you must reconsider SRP.
 - What if you create load() that reads and stores book data from a database rather than a file?
 - What if you create show() that displays the contents of a book on the GUI(Graphical User Interface) screen instead of the console screen?



Open-Closed Principle

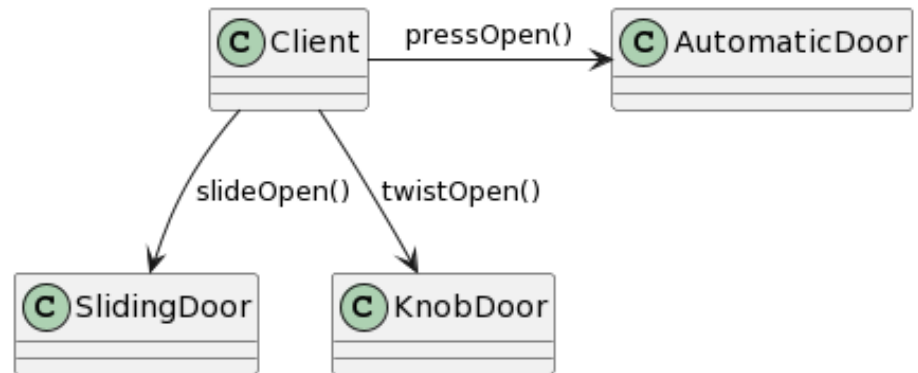
- ❑ “Software entities (class, module, etc) should be **open for extension, but closed for modification.**”
- ❑ You should be able to **extend a class behavior, without modifying it.**
- ❑ Example: Assume a program that opens a door
 - There are three types of doors
 - ❑ Sliding door – door that slide
 - ❑ Knob door – door with a handle
 - ❑ Automatic door – button type automatic door

Open-Closed Principle

❑ Version 1

- Using the if-statement depending on the type of door
- However, if a new door is added, the code modification is inevitable.

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

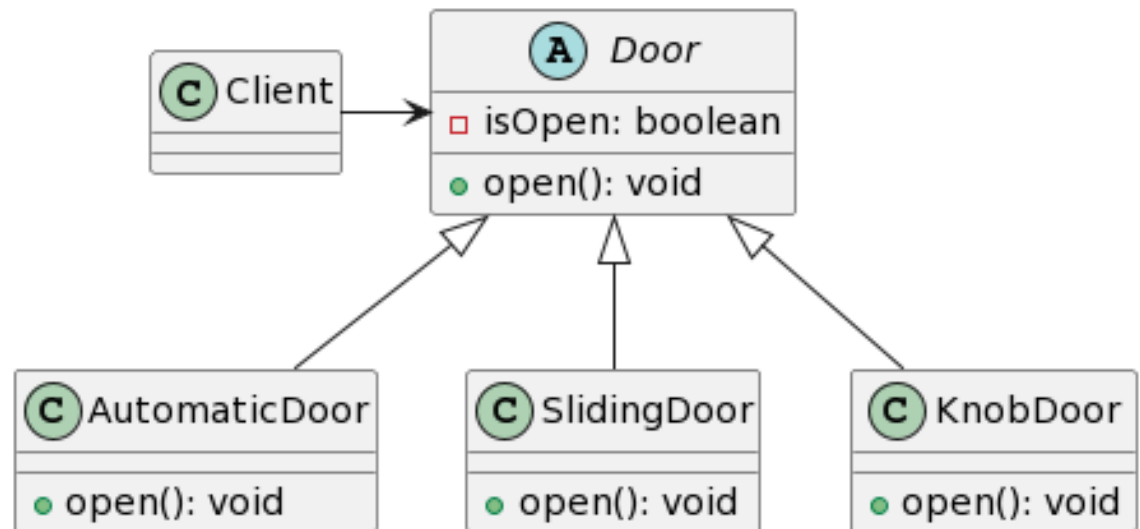


Open-Closed Principle

□ Version 2

- Using polymorphism
- If a new door is added, you just add a new door class and override the open() method.

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



Open-Closed Principle

□ Another example

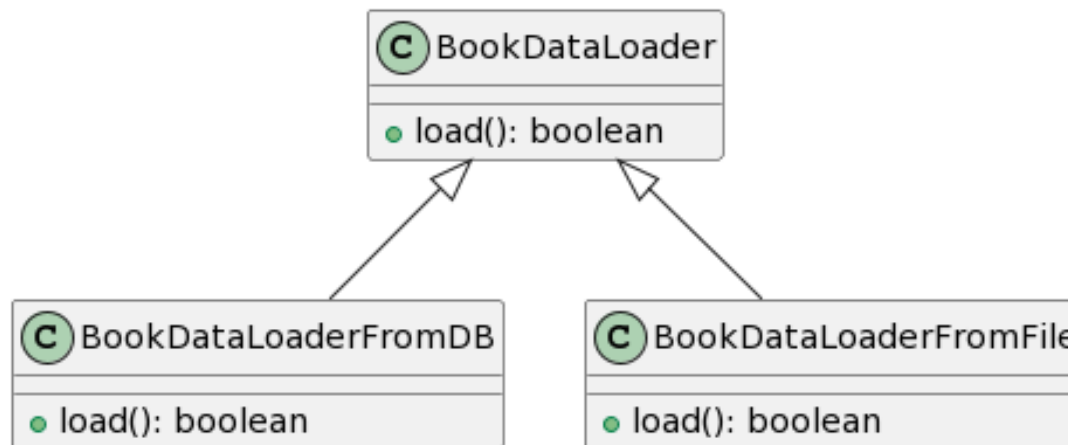
- BookManager.load() method
- BookDataLoader class reads the data from the file.
- BookDataLoaderFromDB class reads the data from the database.

Open-Closed Principle

❑ Version 1

- Using the if-statement depending on the type of loader
- if a new loader is added, the code modification is inevitable.

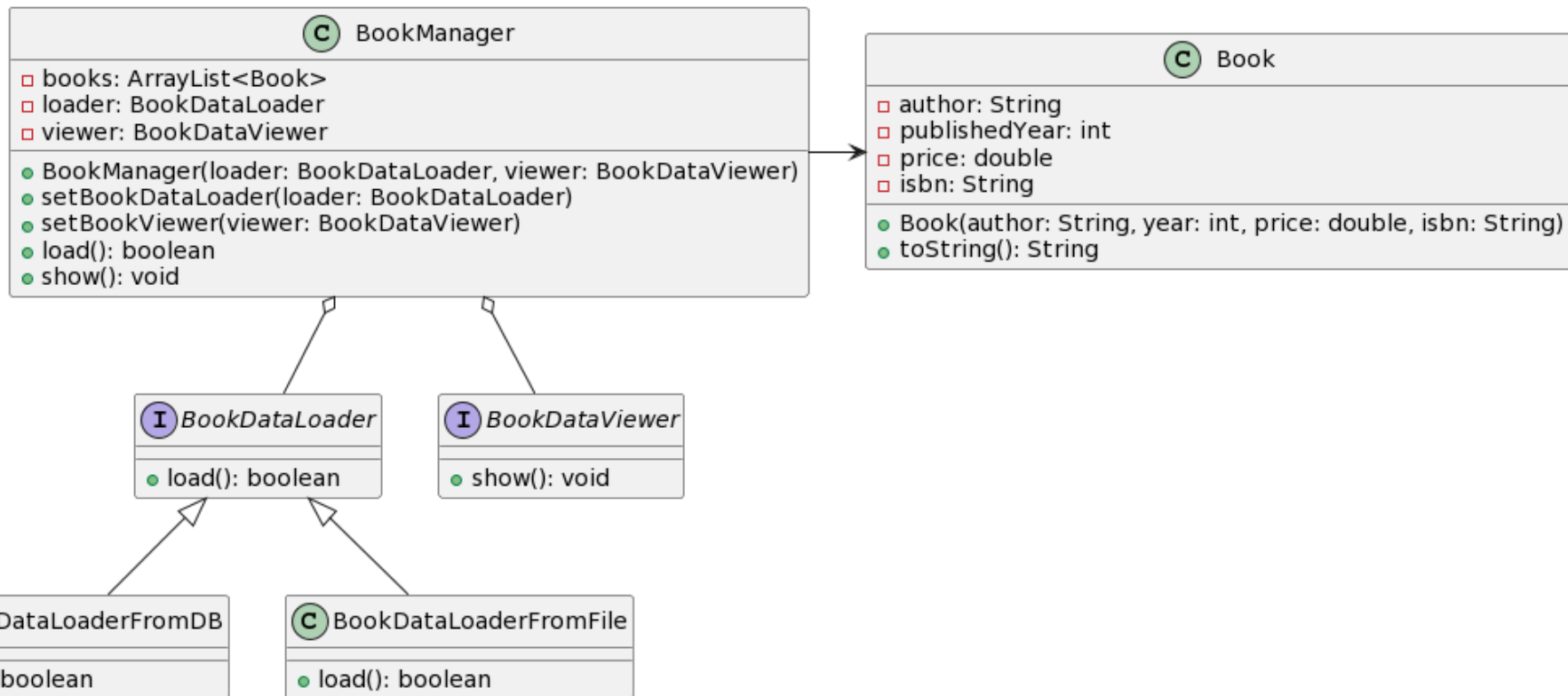
```
if (loader instanceof BookDataLoaderFromFile)  
    manager.loadFromFile(loader);  
else if (loader instanceof BookDataLoaderFromDB)  
    manager.loadFromDB(loader);
```



Open-Closed Principle

- ❑ Version 2
 - Using polymorphism

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

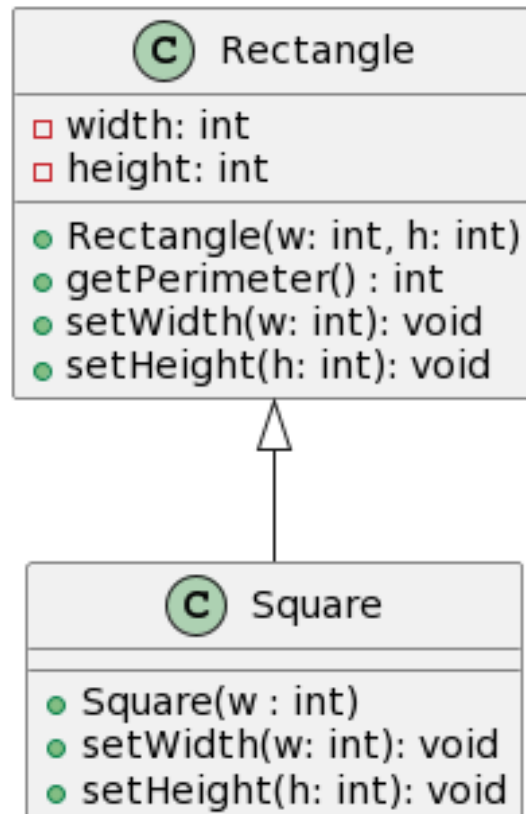


Liskov Substitution Principle

- ❑ "Objects in a program should be **replaceable with instances of their subtypes** without altering the correctness of that program."
- ❑ *Subtypes should be substitutable for their base types.*
- ❑ *Child classes should never break the parent class' type definitions.*
- ❑ *In other words, even if you do upcasting, there should be no problem.*
- ❑ "a violation of LSP is a latent violation of OCP"

Liskov Substitution Principle

- ❑ Example: Rectangle and Square class
 - Square is a special kinds of rectangle.
 - Is the Square class really the subclass of the Rectangle class in programming?



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);  
    }  
    @Override  
    public void setWidth(int w) {  
        super.setWidth(w);  
        super.setHeight(w);  
    }  
    @Override  
    public void setHeight(int h) {  
        super.setWidth(h);  
        super.setHeight(h);  
    }  
}
```

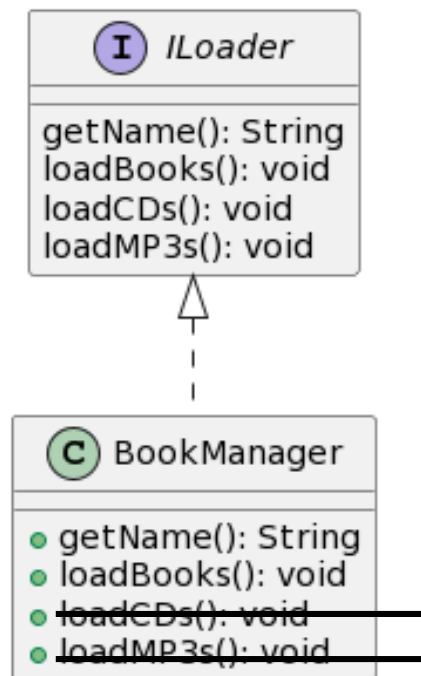
Liskov Substitution Principle

```
class Main {  
    public static void main(String[] args) {  
        Rectangle r = new Rectangle(3, 5);  
        System.out.println(r.getPerimeter()); // 16 (2*8)  
        Square s = new Square(3);  
        System.out.println(s.getPerimeter()); // 12 (2*6)  
        r = s;  
        r.setWidth(3); // set w=3, h=3  
        r.setHeight(5); // set w=5, h=5  
        System.out.println(r.getPerimeter()); // 20 (2*10)  
    }  
}
```

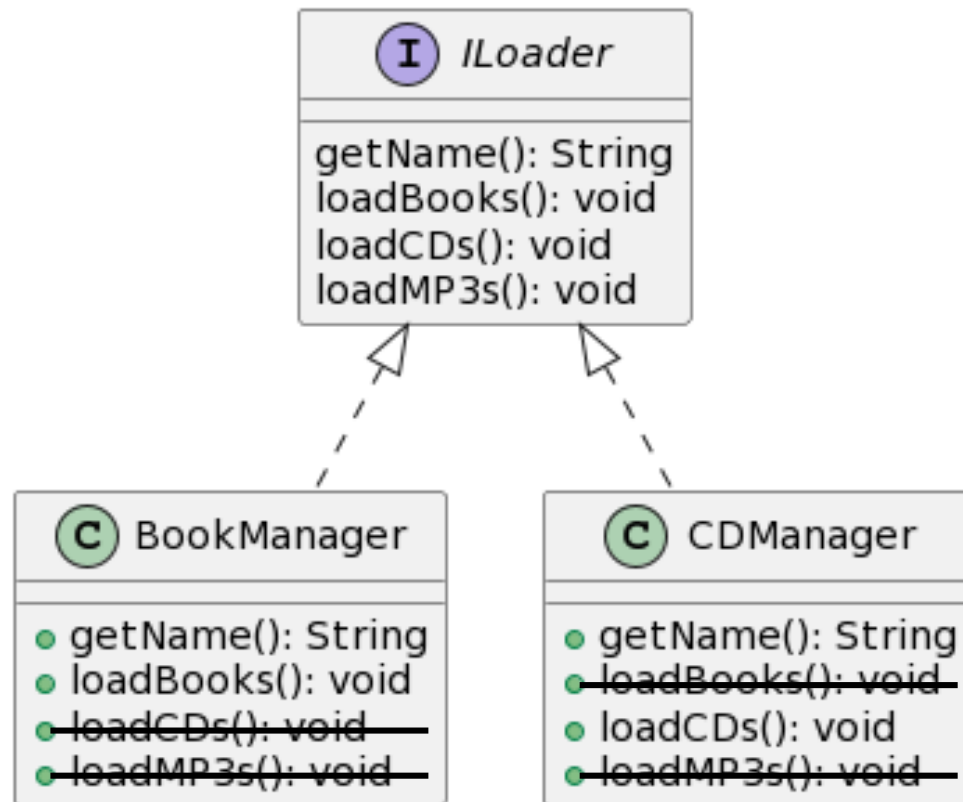
- ❑ Square cannot completely substitute Rectangle. The correct design should be both Rectangle and Square derive from a common Shape class.

Interface Segregation Principle

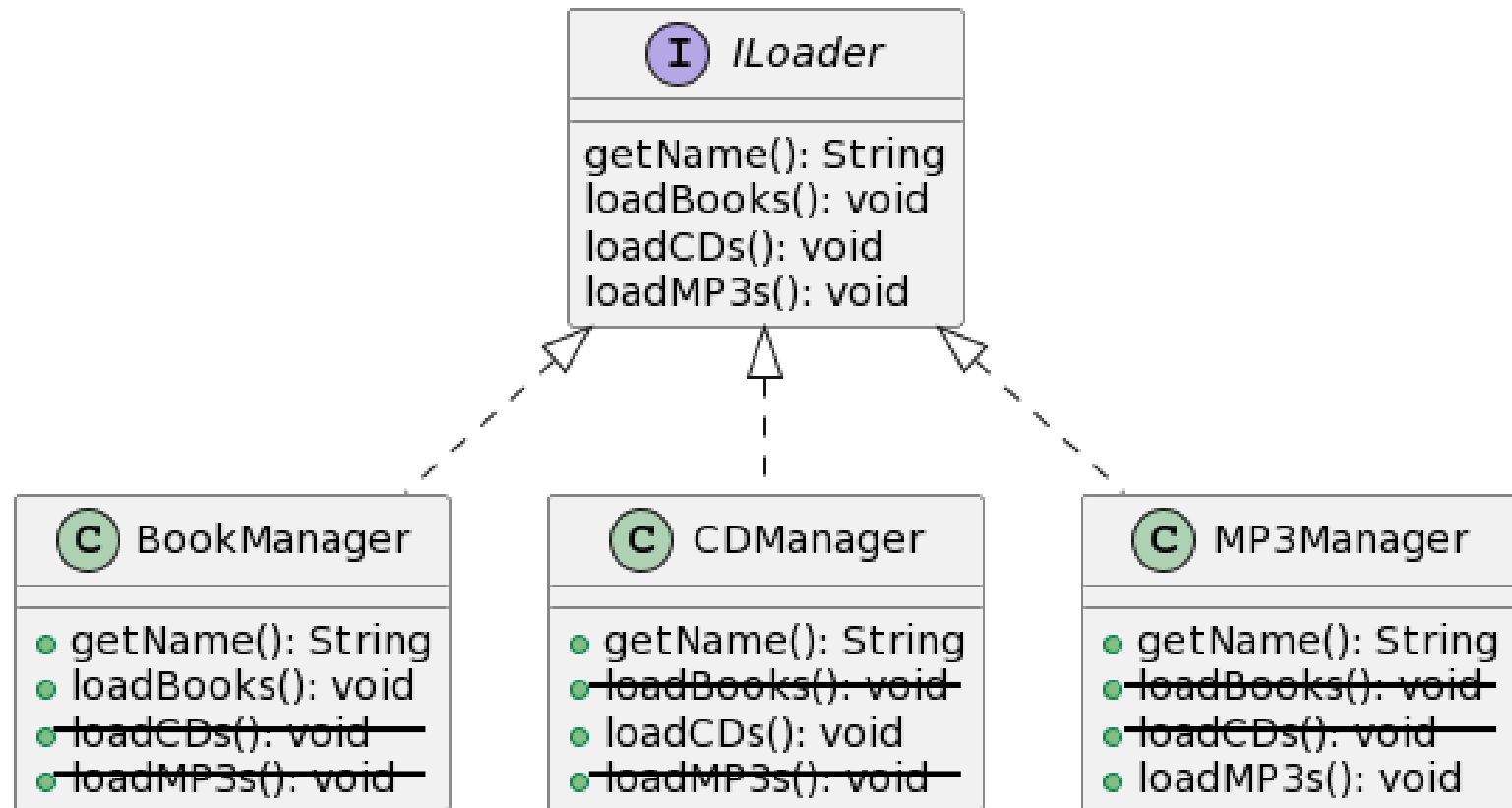
- ❑ "Many client-specific interfaces are better than one general-purpose interface."
- ❑ *"do not force any client to implement an interface which is irrelevant to them"*
- ❑ Each interface should have a specific responsibility.



Interface Segregation Principle

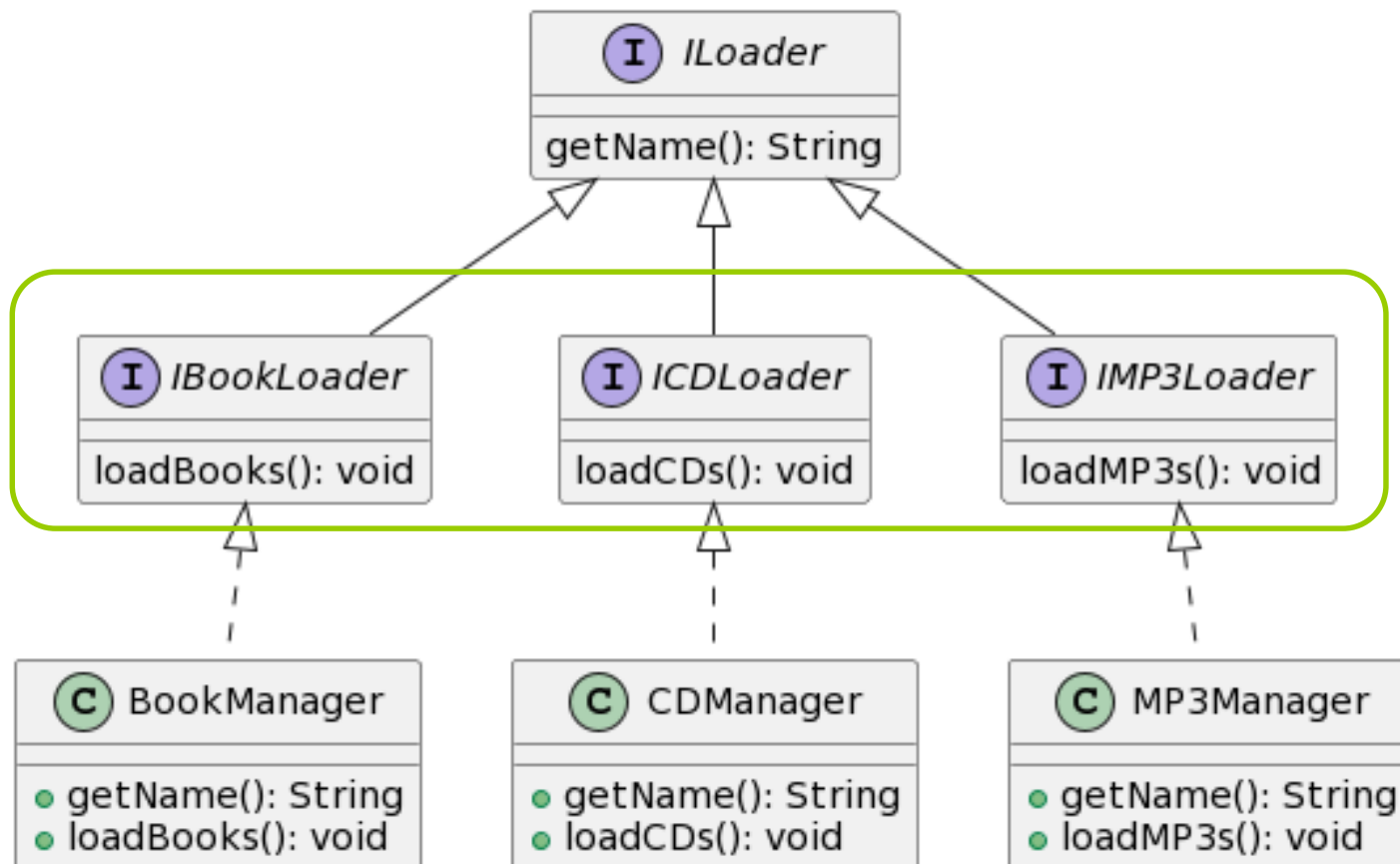


Interface Segregation Principle



Interface Segregation Principle

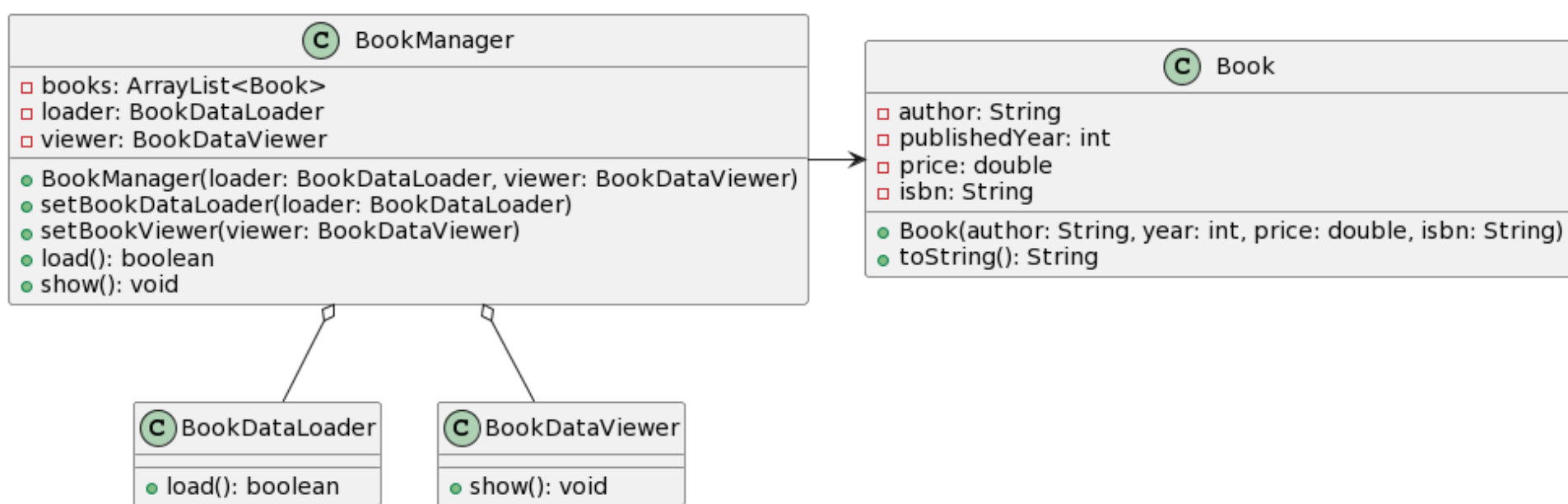
▣ Interface Segregation



Dependency Inversion Principle

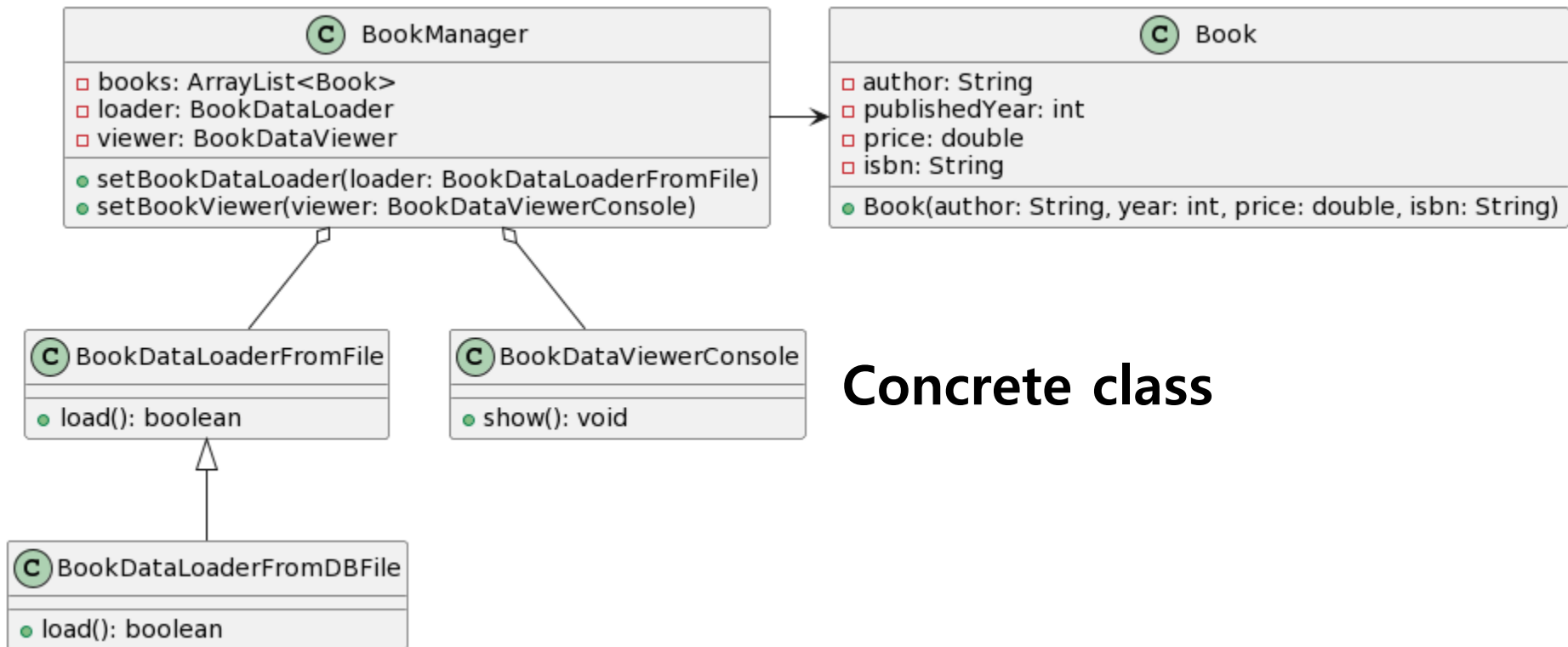
- ❑ “One should **depend upon abstractions, not concretions.**”
- ❑ You should write a code that uses abstract classes or interfaces rather than concrete classes or methods that implement the functionality.
- ❑ What is a dependency between classes?
 - When one class performs a function, and needs a service of another class.
 - To become OCP, DIP must be satisfied basically.
- ❑ How do you distinguish between easy-to-change and hard-to-change?
 - Hard-to-change: “policy”, “strategy”
 - Easy-to-change: “concrete way”, “things”

Dependency Inversion Principle



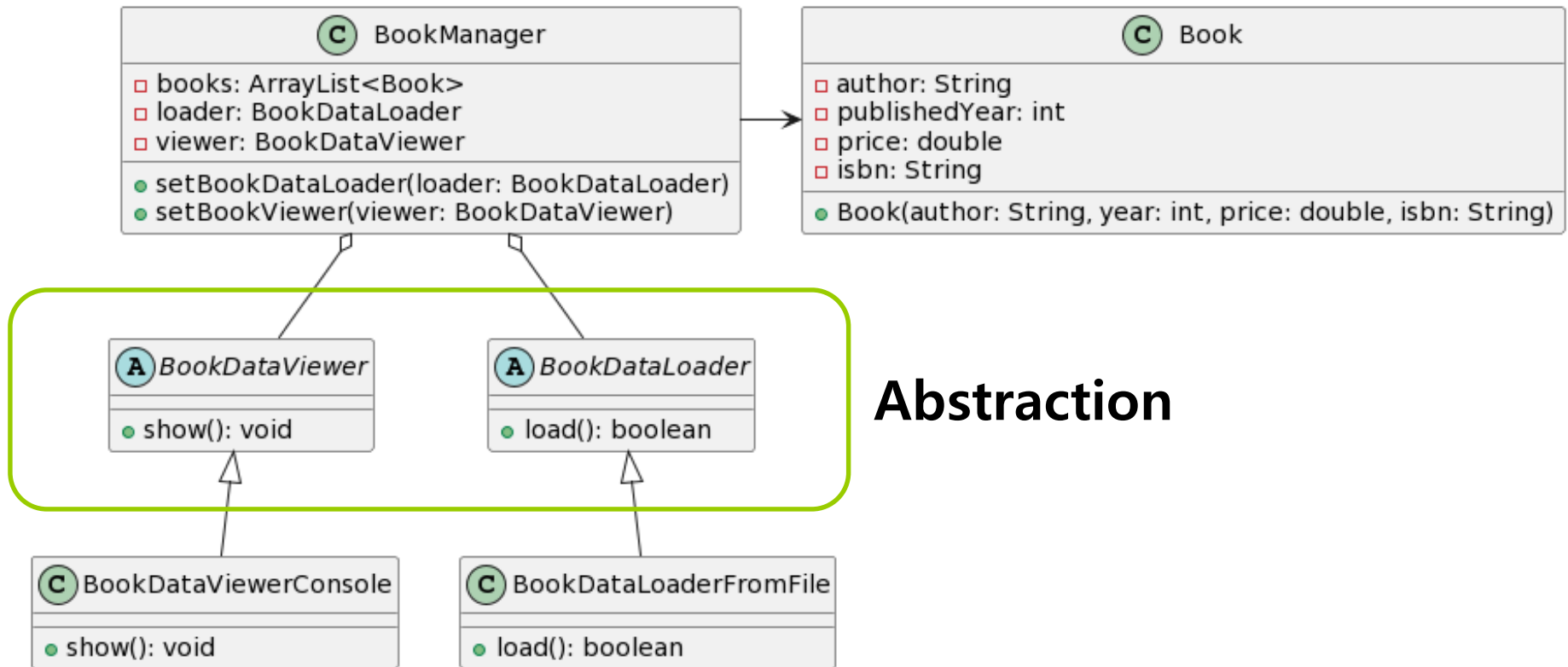
Dependency Inversion Principle

- ❑ Violation of DIP - High-level modules, which provide complex logic, should not import anything from low-level modules, which provide utility features.



Dependency Inversion Principle

- Apply DIP – Need to introduce an abstraction that decouples the high-level and low-level modules from each other



Dependency Inversion Principle

