

Software Engineering

Lecture 4 (Design Principle)

Working on legacy code

- ☐ Re-reading code multiple times to get to the part you need to change.
- ☐ Hard to understand what a method does.
- ☐ Spending a lot of time to fix a minor bug.
- ☐ **You spend more time reading than writing code.**

Principles of Software Engineering

- By the definition, **software engineering** is known as a **systematic and procedural approach** to software development.
- There are some **basic principles governing good software engineering**, of which three popular ones are described below:
 - **DRY** (*“Don’t Repeat Yourself”*)
 - **KISS** (*“Keep It Short and Simple/Stupid”*)
 - **SOLID** design principles

Principles of Software Engineering

DRY (*“Don’t Repeat Yourself”*)

❑ The *“Don’t Repeat Yourself”* (or *“Duplication Is Evil”*) principle tells us that

- Every software engineer should aim to reduce repetition of information or methods within their work in order to avoid redundancy.
- Therefore, it’s recommended to *Separate* segregate the entire system of consideration into fragments.
- *Dividing the code into smaller segments* can help manage the code and use a single segment at any point, by calling, whenever required.

❑ DRY Benefits

- Less code is good: It saves time and effort, is easy to maintain, and also reduces the chances of bugs.

Principles of Software Engineering

KISS (*“Keep It Short and Simple/Stupid”*)

❑ The *“Keep It Short and Simple/Stupid”* principle reminds us that

- All software or applications design and deployment **should be done as simple as possible with least complexities** and clear to understand development procedures.
- This principle ensures that the source code is **made easy to debug** (whenever required) **and future maintenance** by any other operations and maintenance software engineer becomes easier.
- **Each method should only solve one small problem**, not many use cases.
- After all, **programming languages are for humans to understand**.

❑ KISS Benefits

- It will not only be easier to read and maintain, but it can help find bugs a lot faster.

Working on a startup product

- ❑ In charge of the development process
- ❑ **Constantly** adding new features
- ❑ No formal process
- ❑ Very dynamic environment, no time to worry about code structure
- ❑ What is like to go back to your code after 2 years?

The purpose of SOLID design principles

- ❑ To make the code **more maintainable**.
- ❑ To make it **easier to quickly extend** the system with new functionality **without breaking the existing ones**.
- ❑ To make the code **easier to read and understand**, thus spend less time figuring out what it does and more time actually developing the solution.
- ❑ Introduced by **Robert Martin (Uncle Bob)**, named by Michael Feathers.

The purpose of SOLID design principles

❑ In general, SOLID helps us manage code complexity.

It leads to more maintainable and extensible code. Even with big change requests, it's easier to update the code.

❑ **SOLID** is really a guideline and not a rule.

SOLID design principles

- ❑ The SOLID Principles are five principles of Object-Oriented class design.
- ❑ They are a set of rules and best practices to follow while designing a class structure.
- ❑ These five principles help us understand the need for certain design patterns and software architecture in general.
- ❑ It becomes easier for you to develop software that can be managed easily. The other features of using S.O.L.I.D are:
 - It avoids code smells.
 - Quickly refactor code.
 - Can do adaptive or agile software development.

gradually improve

What is the meaning of S.O.L.I.D?

❑ Following the SOLID acronym, they are:

- The **S**ingle Responsibility Principle
- The **O**pen-Closed Principle
- The **L**iskov Substitution Principle
- The **I**nterface Segregation Principle
- The **D**ependency Inversion Principle



Single Responsibility Principle (SRP)

- ❑ A class should have one, and only one, reason to change.
- ❑ A class should only be responsible for one thing.
- ❑ There's a place for everything is in its place.
- ❑ Find one reason to change and take everything else out of the class.
- ❑ Very precise names for many small classes > generic names for large classes.

Single Responsibility Principle (SRP)

```
class Book {  
    String title;  
    String author;  
  
    String getTitle() {  
        return title;  
    }  
    void setTitle(String title) {  
        this.title = title;  
    }  
    String getAuthor() {  
        return author;  
    }  
    void setAuthor(String author) {  
        this.author = author;  
    }  
    void searchBook() {...}  
}
```

- ❑ This code **violates the SRP**, as the Book class has **two responsibilities**. **First**, it sets the data related to the books (title and author). **Second**, it searches for the book in the inventory.
- ❑ The **setter** methods change the Book object, which **might cause problems when we want to search the same book** in the inventory.

To apply the Single Responsibility Principle, we need to decouple the two responsibilities

Single Responsibility Principle (SRP)

Before refactoring

```
class Book {  
  
    String title;  
    String author;  
  
    String getTitle() {  
        return title;  
    }  
    void setTitle(String title) {  
        this.title = title;  
    }  
    String getAuthor() {  
        return author;  
    }  
    void setAuthor(String author) {  
        this.author = author;  
    }  
    void searchBook() {...}  
}
```

- In the refactored code, the **Book** class will **only be responsible for getting and setting the data** of the Book object.
- **InventoryView** class is **only responsible for checking/searching** the inventory.

After refactoring

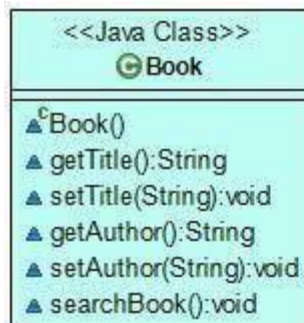
```
class Book {  
  
    String title;  
    String author;  
  
    String getTitle() {  
        return title;  
    }  
    void setTitle(String title) {  
        this.title = title;  
    }  
    String getAuthor() {  
        return author;  
    }  
    void setAuthor(String author) {  
        this.author = author;  
    }  
}
```

```
class InventoryView {  
    Book book;  
    InventoryView(Book book) {  
        this.book = book;  
    }  
    void searchBook() {...}  
}
```

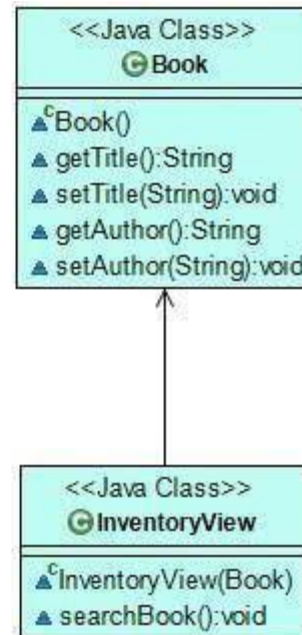
Single Responsibility Principle (SRP)

- On the UML diagram below, we can see how the architecture changed after we refactored the code following the Single Responsibility Principle. We split the initial Book class that had two responsibilities into two classes, each having its own single responsibility.

Before refactoring



After refactoring



Single Responsibility Principle (SRP)

- ❑ When SRP is followed, **testing is easier**. With a single responsibility, the class will have fewer test cases.
- ❑ Less functionality also means **less dependencies** to other modules or classes.
- ❑ It leads to better code organization since smaller and well-purposed classes are easier to search.

Open/Closed Principle

- ❑ The **Open/Closed Principle** states that **classes, modules, microservices, and other code units** should be open for extension **but closed for modification**.
- ❑ So, we should be able to **extend the existing code using OOP features like inheritance via subclasses and interfaces**. However, **we should never modify classes, interfaces, and other code units that already exist (especially if those codes are in production already)**, as it can lead to unexpected behavior.
- ❑ If we add a new feature by extending our code rather than modifying it, it will **minimize the risk of failure as much as possible**. Besides, we also **don't have to unit test existing functionalities**.

Open/Closed Principle

```
class CookbookDiscount {  
    String getCookbookDiscount() {  
        String discount = "30% between Dec 1 and 24.";  
        return discount;  
    }  
}  
  
class DiscountManager {  
    void processCookbookDiscount(CookbookDiscount discount) {...}  
}
```

- ❑ **CookbookDiscount** to hold the details of the discount.
- ❑ **DiscountManager** to apply the discount to the price.

Open/Closed Principle

```
class BiographyDiscount {  
    String getBiographyDiscount() {  
        String discount = "50% on the subject's birthday.";  
        return discount;  
    }  
}
```

To add the new feature, for example, every biography with a 50% discount on the subject's birthday, we create a new **BiographyDiscount** class.

To process the new type of discount, we **need to add the new functionality** to the **DiscountManager** class, too

```
class DiscountManager {  
    void processCookbookDiscount(CookbookDiscount discount) {...}  
    void processBiographyDiscount(BiographyDiscount discount) {...}  
}
```

However, as we changed existing functionality, we violated the Open/Closed Principle.

Open/Closed Principle

```
public interface BookDiscount {  
    String getBookDiscount();  
}  
  
class CookbookDiscount implements BookDiscount {  
    @Override  
    public String getBookDiscount() {  
        String discount = "30% between Dec 1 and 24.";   
        return discount;  
    }  
}  
  
class BiographyDiscount implements BookDiscount {  
    @Override  
    public String getBookDiscount() {  
        String discount = "50% on the subject's birthday.";   
        return discount;  
    }  
}
```

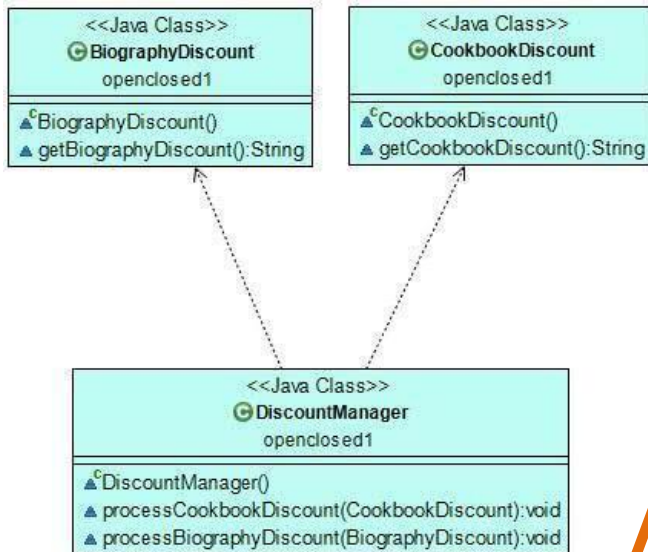
```
class DiscountManager {  
    void processBookDiscount(BookDiscount discount) {...}  
}
```

- ❑ We need to **refactor** the code **by adding an extra layer of abstraction** that represents all types of discounts.
- ❑ Let's create a new interface called **BookDiscount** that the **CookbookDiscount** and **BiographyDiscount** classes will implement.

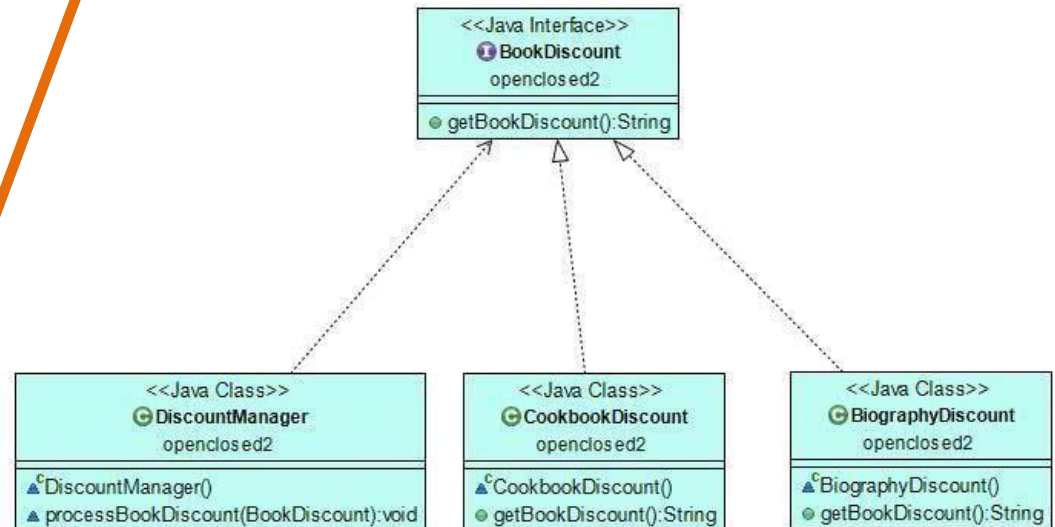
Now, **DiscountManager** can refer to the **BookDiscount** interface **instead of the concrete classes**.

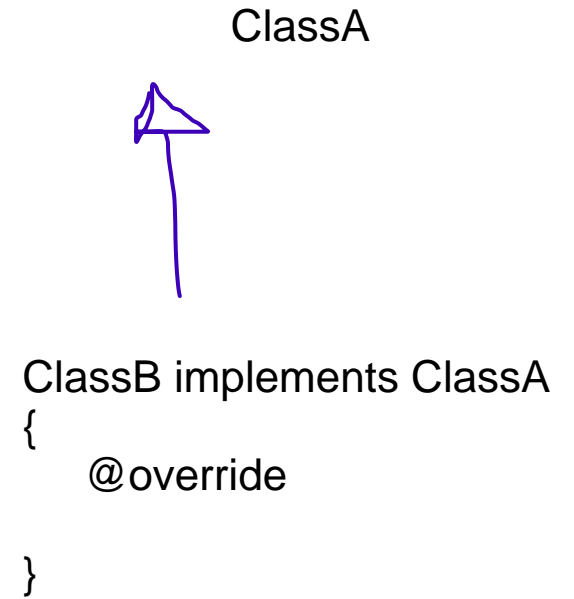
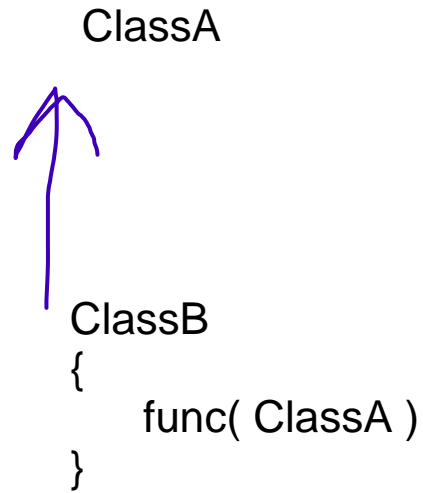
Open/Closed Principle

Before refactoring



After refactoring





Open/Closed Principle

- ❑ An entity should be open for extension but closed for modification.
- ❑ Extend functionality by adding new code instead of changing the existing code.
- ❑ Separate the behaviors, so the system can easily be extended, but never broken.
- ❑ **Goal: get a point where you can never break the core of your system.**

Open/Closed Principle

- ❑ What this means in essence is that you should design your classes and modules with possible future updates in mind, so they should have a ^{Common} generic design that you won't need to change the class itself in order to extend their behavior.
- ❑ You can add more fields or methods, but in such a way that you don't need to rewrite old methods, delete old fields and modify the old code in order to make it work again. Thinking ahead will help you write stable code, before and after an update of requirements.

Liskov Substitution Principle

- ❑ “In a computer program, if S is a subtype of T , then objects of type T **may be replaced with objects of type S** (i.e., objects of type S may substitute objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.).”
- ❑ In layman’s terms, it states that an object of a superclass should be replaceable by objects of its subclasses without causing issues in the application.
- ❑ We can implement the **Liskov Substitution Principle** by paying attention to the **correct inheritance hierarchy**.

```
Class3 book = new Class4();
```

Class3



Class4

Liskov Substitution Principle

A **BookDelivery** class that informs customers about the number of locations where they can collect their order.


```
class BookDelivery {  
    String titles;  
    int userID;  
  
    void getDeliveryLocations() {...}  
}
```

The store also sells fancy hardcovers they only want to deliver to their high street shops.

```
class HardcoverDelivery extends BookDelivery {  
  
    @Override  
    void getDeliveryLocations() {...}  
  
}
```

Later, the store asks us to create delivery functionalities for audiobooks, too.

```
class AudiobookDelivery extends BookDelivery {  
  
    @Override  
    void getDeliveryLocations() {...}  
  
}
```



Can't be implemented

Liskov Substitution Principle

To solve the problem, we need to fix the inheritance hierarchy by adding an extra layer that better differentiates book delivery types.

After refactoring

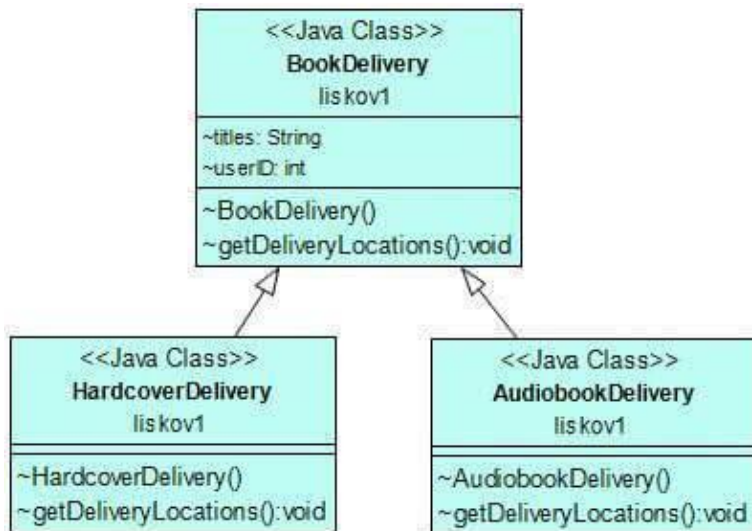
```
class HardcoverDelivery extends OfflineDelivery {  
  
    @Override  
    void getDeliveryLocations() {...}  
  
}  
  
class AudiobookDelivery extends OnlineDelivery {  
  
    @Override  
    void getSoftwareOptions() {...}  
  
}
```

```
class BookDelivery {  
  
    String title;  
    int userID;  
  
}  
  
class OfflineDelivery extends BookDelivery {  
  
    void getDeliveryLocations() {...}  
  
}  
  
class OnlineDelivery extends BookDelivery {  
  
    void getSoftwareOptions() {...}  
  
}
```

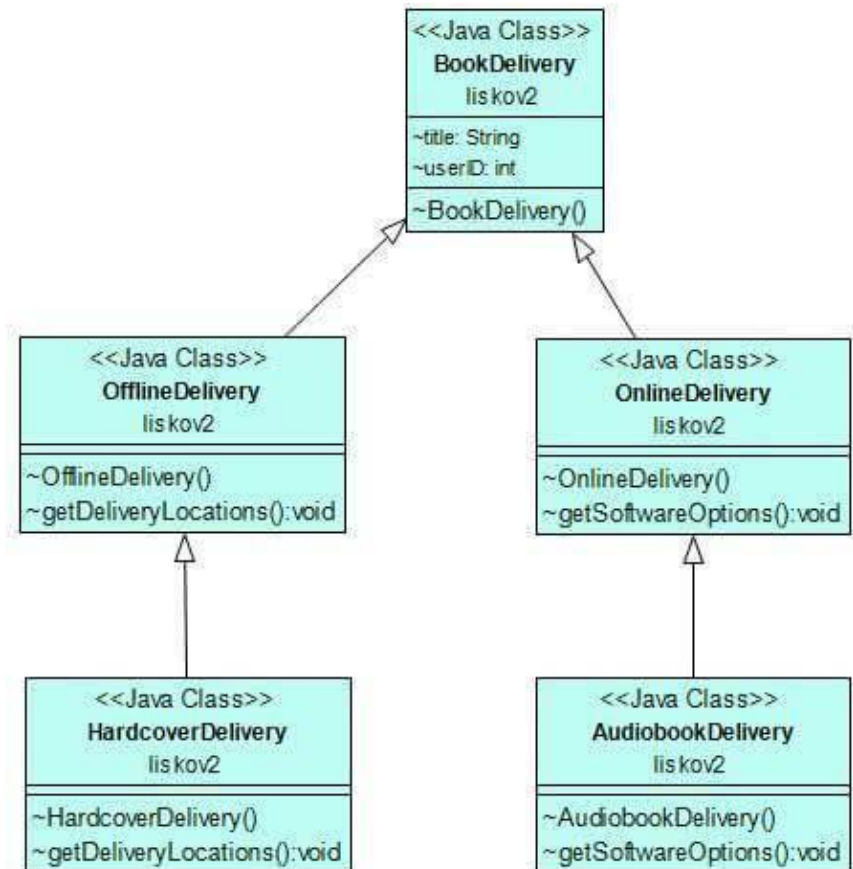
Liskov Substitution Principle

Liskov er gula Class

Before refactoring



After refactoring



Liskov Substitution Principle

- ❑ Any derived class should be able to substitute its parent class without the consumer knowing it.
- ❑ Every class that implements an interface, must be able to substitute any reference throughout the code that implements that same interface.
- ❑ Every part of the code should get the expected result no matter what instance of a class you send to it, given it implements the same interface.

A
B
C

Interface Segregation Principle

- ❑ **Classes** (that implement interfaces) **should not be forced to depend on methods they don't use.** In other words, **interfaces shouldn't include too many functionalities.** Same as Liskov Liskov er gula class eigula interface
- ❑ The **violation** of Interface Segregation Principle **harms code readability** and **forces** programmers to write **dummy methods that do nothing.**
- ❑ In a well-designed application, we **should avoid interface pollution** (also called **fat interfaces**). The **solution** is to create **smaller interfaces that can be implemented more flexibly.**

Interface Segregation Principle

Let's add **some user actions** to our online bookstore so that **customers can interact with the content before making a purchase**. To do so, we **create an interface** called ***BookAction*** with three methods: ***seeReviews()***, ***searchSecondHand()***, and ***listenSample()***.

```
public interface BookAction {  
  
    void seeReviews();  
    void searchSecondhand();  
    void listenSample();  
  
}
```

Then, we create two classes: **HardcoverUI** and an **AudiobookUI** that implement the ***BookAction*** interface with their own functionalities:

- Both **HardcoverUI** and an **AudiobookUI** **depend on methods they don't use**, so **Interface Segregation Principle is violated**. Hardcover books can't be listened to, so the **HardcoverUI** class **doesn't need the listenSample()** method. Similarly, audiobooks don't have second-hand copies, so the **AudiobookUI** class doesn't need it, either.
- Thus, ***BookAction*** is a **polluted interface** that we **need to segregate**.

```
class HardcoverUI implements BookAction {  
  
    @Override  
    public void seeReviews() {...}  
  
    @Override  
    public void searchSecondhand() {...}  
  
    @Override  
    public void listenSample() {...}  
  
}  
  
class AudiobookUI implements BookAction {  
  
    @Override  
    public void seeReviews() {...}  
  
    @Override  
    public void searchSecondhand() {...}  
  
    @Override  
    public void listenSample() {...}  
  
}
```

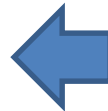
Interface Segregation Principle

Solution is to extend *BookAction* interface with two more specific sub-interfaces: *HardcoverAction* and *AudioAction*.



```
public interface BookAction {  
    void seeReviews();  
}  
  
public interface HardcoverAction extends BookAction {  
    void searchSecondhand();  
}  
  
public interface AudioAction extends BookAction {  
    void listenSample();  
}
```

```
class HardcoverUI implements HardcoverAction {  
    @Override  
    public void seeReviews() {...}  
  
    @Override  
    public void searchSecondhand() {...}  
}  
  
class AudiobookUI implements AudioAction {  
    @Override  
    public void seeReviews() {...}  
  
    @Override  
    public void listenSample() {...}  
}
```

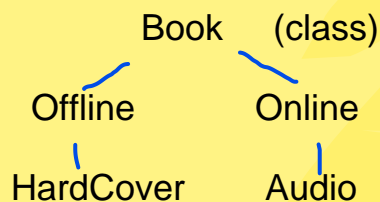
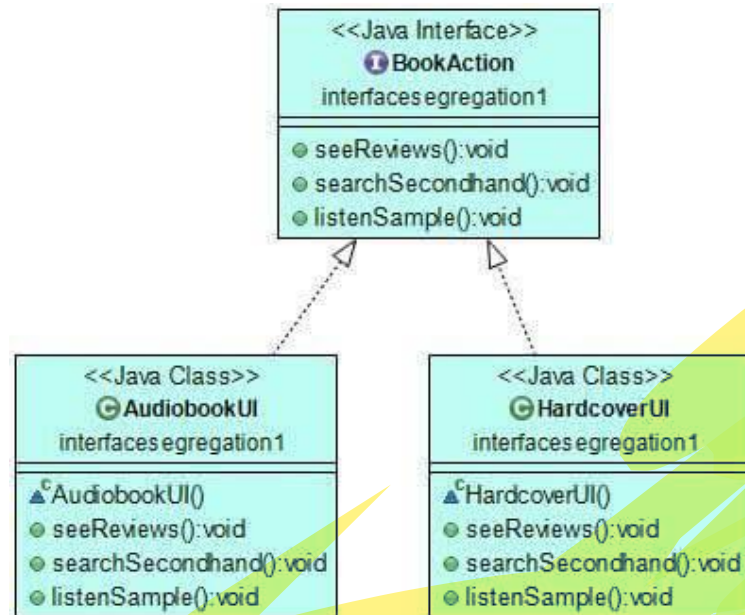


Now, the **HardcoverUI** class can implement the *HardcoverAction* interface and the **AudiobookUI** class can implement the *AudioAction* interface.

This way, both classes can implement the `seeReviews()` method of the **BookAction** super-interface. However, **HardcoverUI** doesn't have to implement the irrelevant `listenSample()` method and **AudioUI** doesn't have to implement `searchSecondhand()`, either.

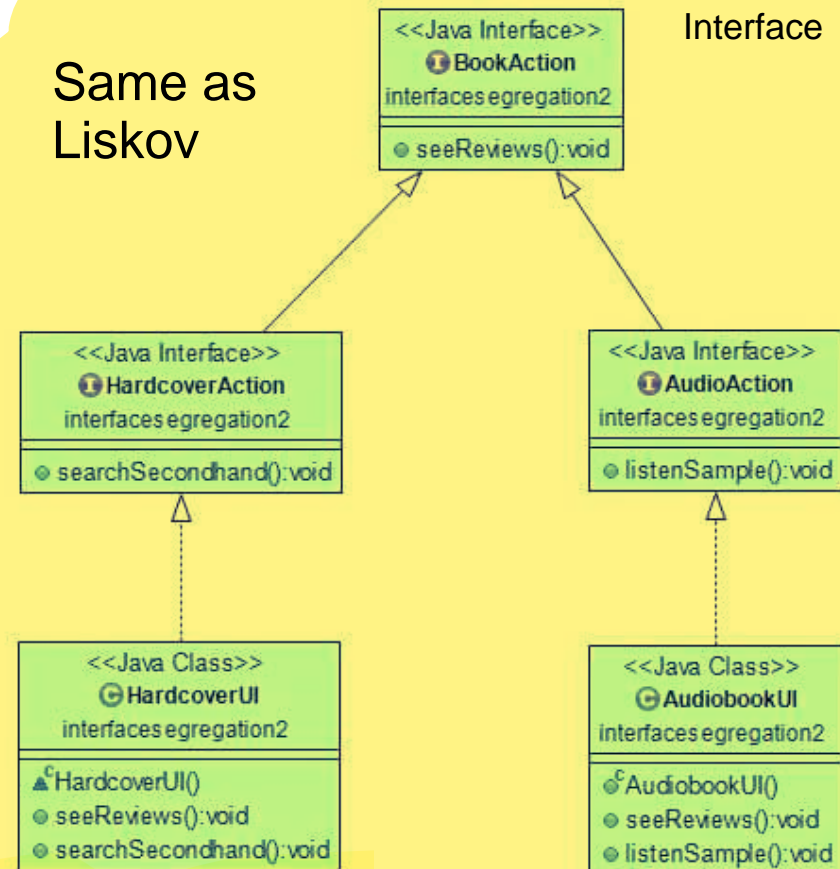
Interface Segregation Principle

Before refactoring



After refactoring

Same as
Liskov



Interface Segregation Principle

- ❑ A client should never be forced to depend on methods it doesn't use.

Interface or subgula method

- ❑ Or, a client should never depend on anything more than the method it's calling.

- ❑ Changing one method in a class shouldn't affect classes that don't depend on it.

- ❑ Replace fat interfaces with many small, specific interfaces

Interface Segregation Principle

- The *Interface Segregation Principle* (ISP) states that the client should never be forced to depend on an interface they aren't using in its entirety. This means that **an interface should have a minimum set of methods necessary for the functionality it ensures,** and should be limited to only one functionality.

Dependency Inversion Principle

- The goal of the **Dependency Inversion Principle** is to avoid tightly coupled code, as it easily breaks the application. The principle states that:
 - *“High-level modules should not depend on low-level modules. Both should depend on abstractions.”*
 - *“Abstractions should not depend on details. Details should depend on abstractions.”*
- In other words, we need to decouple high-level and low-level classes. High-level classes usually encapsulate complex logic while low-level classes include data or utilities. Typically, most people would want to make high-level classes depend on low-level classes. However, according to the Dependency Inversion Principle, we need to invert the dependency. Otherwise, when the low-level class is replaced, the high-level class will be affected, too.
- As a solution, you need to create an abstract layer for low-level classes, so that high-level classes can depend on abstraction rather than concrete implementations.
- Dependency Inversion Principle is a specific combination of the Open/Closed and Liskov Substitution Principles.

Dependency Inversion Principle

- Now, the book store requires to add a new feature that enables customers to put their favorite books on a shelf.
Sprint-1
- To implement the new functionality, we create a lower-level **Book class** and a higher-level **Shelf class**.
- Next, the store requires to enable customers to add DVDs to their shelves, too.
Sprint-2
- To fulfill the demand, we create a new **DVD class**:
- and modify the **Shelf class** so that it can accept **DVDs, too**.

```
class Book {  
  
    void seeReviews() {...}  
    void readSample() {...}  
  
}  
  
class Shelf {  
  
    Book book;  
  
    void addBook(Book book) {...}  
    void customizeShelf() {...}  
  
}
```

```
class DVD {  
  
    void seeReviews() {...}  
    void watchSample() {...}  
  
}
```

```
class Shelf {  
  
    Book book;  
  
    DVD dvd;  
  
    void addBook(Book book) {...}  
    void addDVD(DVD dvd) {...}  
    void customizeShelf() {...}  
  
}
```

This clearly breaks the Open/Closed Principle


Dependency
ultai dibe .

age book super
ekhon product super

Dependency Inversion Principle

The **solution** is to create an abstraction layer for the lower-level classes (Book and DVD) by introducing the **Product interface** that both classes will implement.

Now, Shelf can reference the Product interface instead of its implementations (Book and DVD).



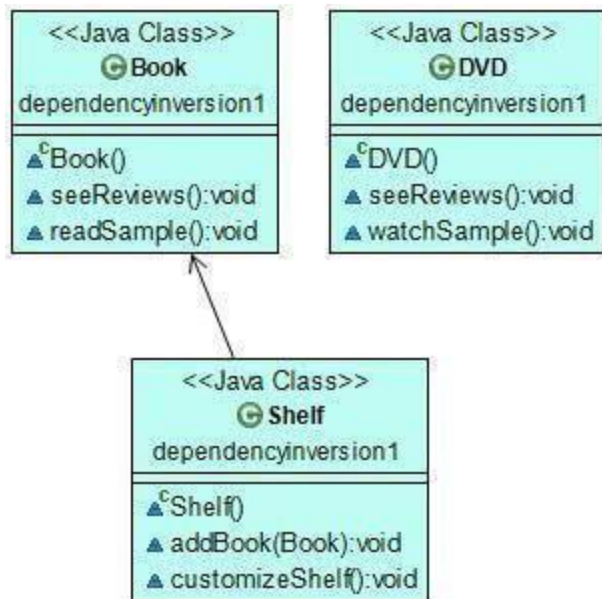
```
class Shelf {  
    Product product;  
    void addProduct(Product product) {...}  
    void customizeShelf() {...}  
}
```

```
public interface Product {  
    void seeReviews();  
    void getSample();  
}  
  
class Book implements Product {  
    @Override  
    public void seeReviews() {...}  
    @Override  
    public void getSample() {...}  
}  
  
class DVD implements Product {  
    @Override  
    public void seeReviews() {...}  
    @Override  
    public void getSample() {...}  
}
```

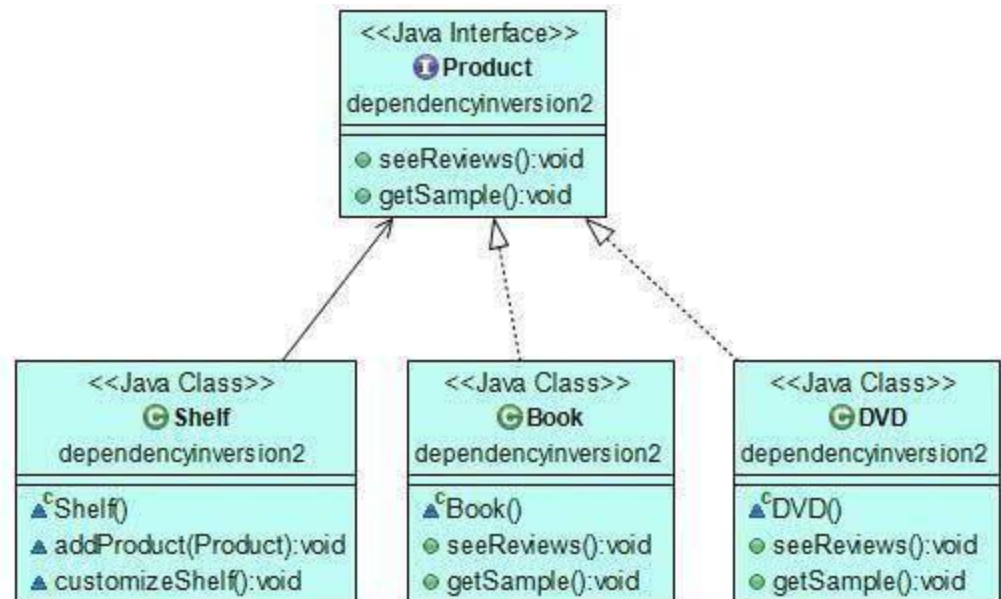
The above code also follows the Liskov Substitution Principle, as the Product type can be substituted with both of its subtypes (Book and DVD) without breaking the program. At the same time, it also implemented the Dependency Inversion Principle, as in the refactored code, high-level classes don't depend on low-level classes, either.

Dependency Inversion Principle

Before refactoring



After refactoring



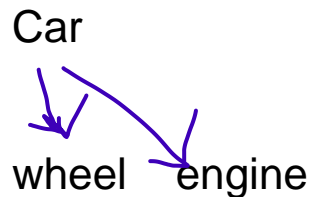
Liskov

Dependency inverted
now product superclass

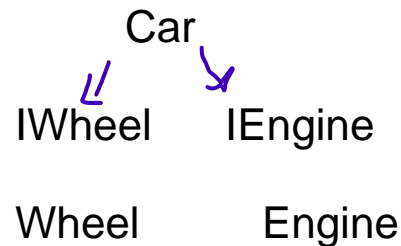
Product a = new Book();
Product b = new DVD();

Dependency Inversion Principle

- ❑ Never depend on anything concrete, only depend on abstractions.
- ❑ High level modules should not depend on low level modules. They should depend on abstractions.
- ❑ Able to change an implementation easily without altering the high level code.



depend on concrete class



depend on abstraction

Dependency Inversion Principle

- ❑ If Dependency Inversion is not implemented in the code, we run the risk of:
 - ❑ Damaging the higher level code that uses the lower level classes.
 - ❑ Requiring too much time and effort to change the higher level code when a change occurs in the lower level classes.
 - ❑ Producing less-reusable code.

Dependency Inversion Principle

- According to the Dependency Inversion Principle (DIP), high-level and low-level modules should be decoupled in such a way that changing (or even replacing) low-level modules doesn't require (much) rework of high-level modules. Given that, both low-level and high-level modules shouldn't depend on each other, but rather they should depend on abstractions, such as interfaces.
- Another important thing **DIP** states is:
 - Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.
- This principle is important because it decouples modules, making the system less complex, easier to maintain and update, easier to test, and more reusable. I can't stress enough how much of a game changer this is, especially for unit testing and reusability. If the code is written generically enough, it can easily find application in another project, while code that's too specific and interdependent with other modules of the original project will be hard to decouple from it.





Don't get trapped by SOLID

- ☐ SOLID design principles are principles, not rules.
- ☐ Always use common sense when applying SOLID.
- ☐ Avoid over-fragmenting your code for the sake of SRP of SOLID.
- ☐ Don't try to achieve SOLID, use SOLID to achieve maintainability.