Object Oriented Analysis and Design with Java

UE19CS353

Prof.J.Ruby Dinakar
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

PES UNIVERSITY ONLINE

UE19CS353: Object Oriented Analysis and Design with Java

OO Design Principles and Sample Implementation of Patterns in Java

Prof. J.Ruby Dinakar

Department of Computer Science and Engineering

PES UNIVERSITY ONLINE

UE19CS353: Object Oriented Analysis and Design with Java

OO Design Principles and Sample Implementation of Patterns in Java

Liskov Substitution Principle (LSP) & Interface Segregation Principle (ISP)

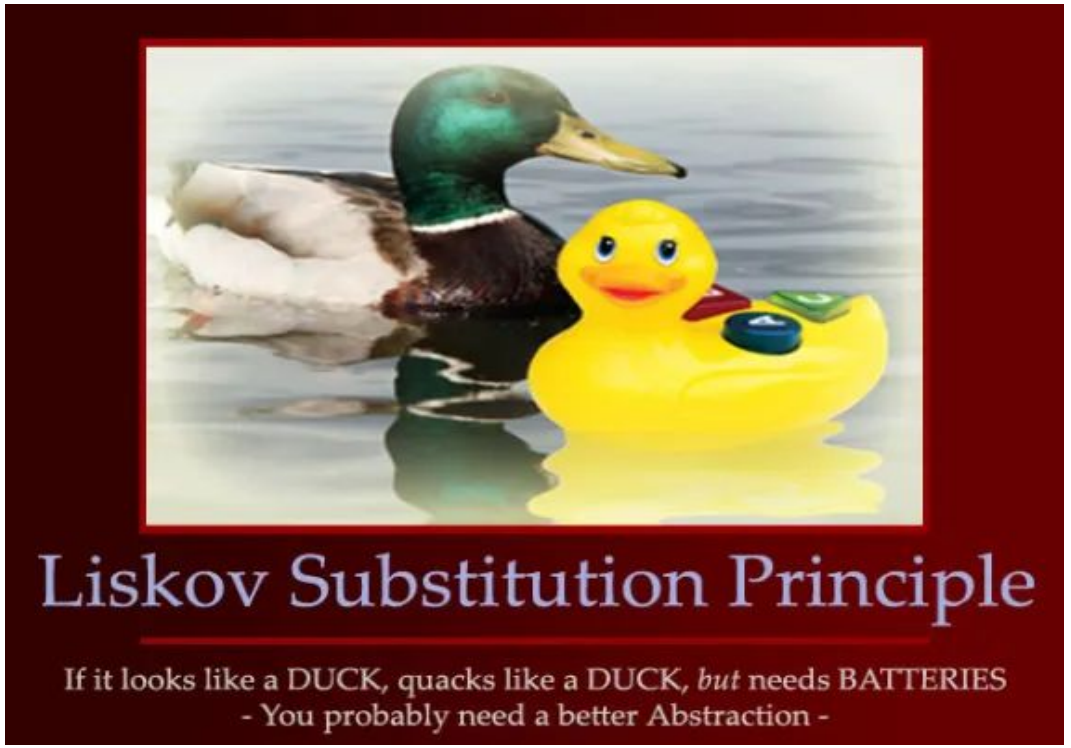
Prof. J.Ruby Dinakar

Department of Computer Science and Engineering

Liskov Substitution Principle (LSP)

- It was introduced by Barbara Liskov in 1987
- “Let S be a subtype of T, then for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2”.
- **Derived types must be completely substitutable for their base types**
- An object of a superclass should be replaceable by objects of its subclasses without causing issues in the application.

Liskov Substitution Principle (LSP)



- A child class should never change the characteristics of its parent class.
- Derived classes should never do less than their base class.
- LSP applies to inheritance hierarchies.
- Avoids overuse/misuse of inheritance.
- It helps us conform to the “is-a” relationship.

Example code for violation of LSP

Let us consider the same book store example.

The book store asks us to add a new delivery functionality to the application.

Create 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() {...}
}
```

Example code for violation of LSP

The store also sells fancy hardcovers. It wants to deliver that to their high street shops.

Create a new HardcoverDelivery subclass that extends BookDelivery and overrides the getDeliveryLocations() method with its own functionality

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

Example code for violation of LSP

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

Now, we extend the existing BookDelivery class with an AudiobookDelivery subclass.

```
class AudiobookDelivery extends BookDelivery {

    @Override
    void getDeliveryLocations() {/* can't be implemented */}
}
```

Violation of LSP

When we want to override the getDeliveryLocations() method, we realize that audiobooks can't be delivered to physical locations.

We have to change some characteristics of the getDeliveryLocations() method.

That would violate the Liskov Substitution Principle.

After the modification, we couldn't replace the BookDelivery superclass with the AudiobookDelivery subclass without breaking the application.

Solution

To solve the problem, we need to fix the inheritance hierarchy.

Let’s introduce an extra layer of abstraction that better differentiates book delivery types.

The new OfflineDelivery and OnlineDelivery classes split up the BookDelivery superclass.

Move the getDeliveryLocations() method to OfflineDelivery and create a new getSoftwareOptions() method for the OnlineDelivery class (as this is more suitable for online deliveries).

Good Design

```
class BookDelivery {

        String title;
        int userID;
}

class OfflineDelivery extends BookDelivery {

        void getDeliveryLocations() {...}
}

class OnlineDelivery extends BookDelivery {

        void getSoftwareOptions() {...}
}
```

Good Design

In the refactored code, HardcoverDelivery will be the child class of OfflineDelivery and it will override the getDeliveryLocations() method with its own functionality.

AudiobookDelivery will be the child class of OnlineDelivery, now it doesn't have to deal with the getDeliveryLocations() method.

Instead, it can override the getSoftwareOptions() method of its parent with its own implementation (for instance, by listing and embedding available audio players).

Good Design

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

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

After the refactoring, we could use any subclass in place of its superclass without breaking the application.

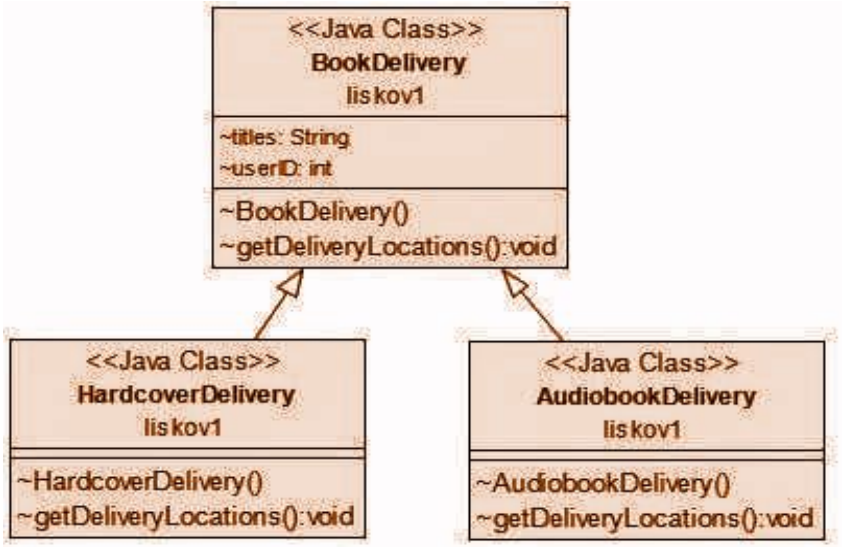
Object Oriented Analysis and Design with Java

UML Class Diagram

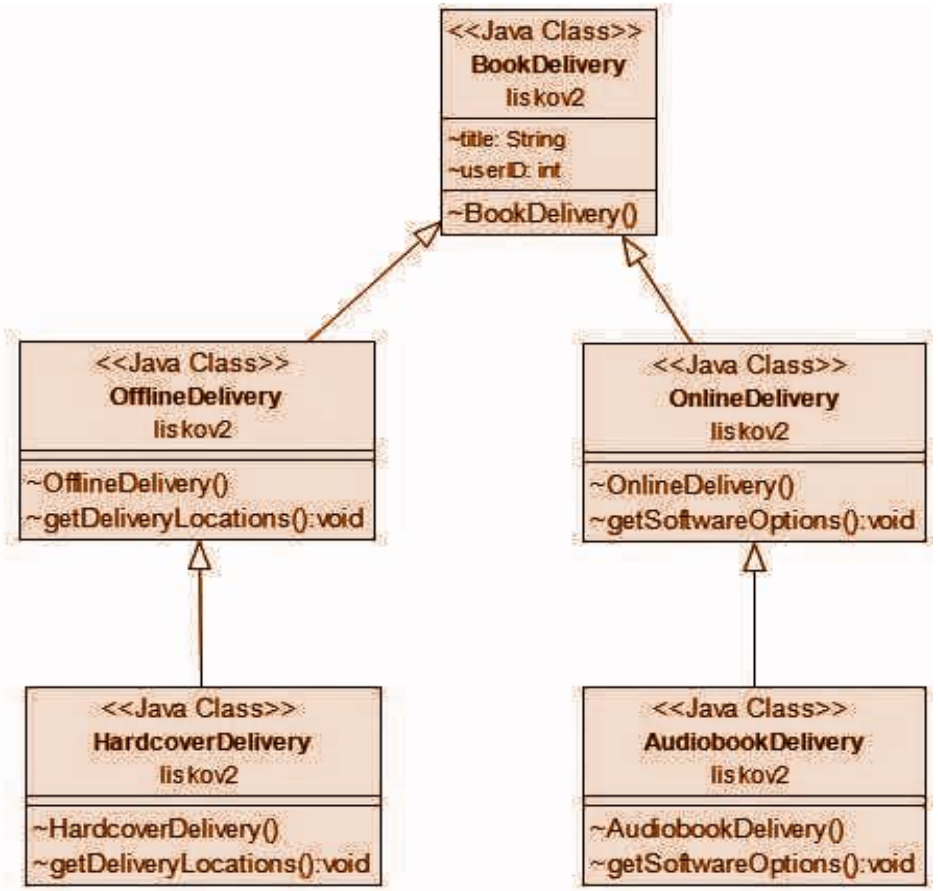
By applying the Liskov Substitution Principle, an extra layer is added to the inheritance hierarchy.

While the new architecture is more complex, it provides us with a more flexible design.

Bad Design



Good Design



Interface Segregation Principle (ISP)

Interface Segregation Principle (ISP)

"Make fine grained interfaces that are client-specific. Clients should not be forced to implement interfaces they do not use".

- Break the fat interfaces which has too many methods.
- The principle states that many client-specific interfaces are better than one general-purpose interface.
- Clients should not be forced to implement a function they do not need.
- Instead, start by building a new interface and then let your class implement multiple interfaces as needed.
- create smaller interfaces that you can implement more flexibly
- Failure to comply with this principle means that in our implementations we will have dependencies on

Example code for violation of ISP

Let's add some user actions to the online bookstore so that customers can interact with the content before making a purchase.

Create an interface called BookAction with three methods:

seeReviews(), searchSecondHand(), and listenSample().

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

}
```

Example code for violation of ISP

Create two classes: HardcoverUI and an AudiobookUI that implement the BookAction interface with their own functionalities

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

    @Override
    public void searchSecondhand() {...}

    @Override
    public void listenSample() {...}
}
```

Violates ISP

```
class AudiobookUI implements BookAction {

    @Override
    public void seeReviews() {...}

    @Override
    public void searchSecondhand() {...}

    @Override
    public void listenSample() {...}

}
```

Violation of ISP

Both classes depend on methods they don’t use, so it violates the Interface Segregation Principle.

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.

BookAction is a polluted interface that we need to segregate.

Solution

Let’s extend it 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();
}
```

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

Both the 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.

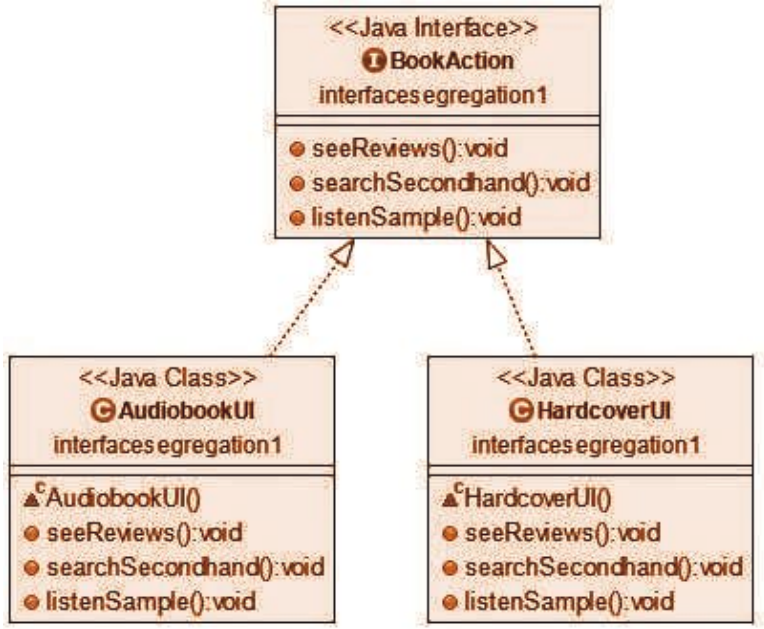

```
class HardcoverUI implements HardcoverAction {
    @Override
    public void seeReviews() {...}
    @Override
    public void searchSecondhand() {...}
}

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

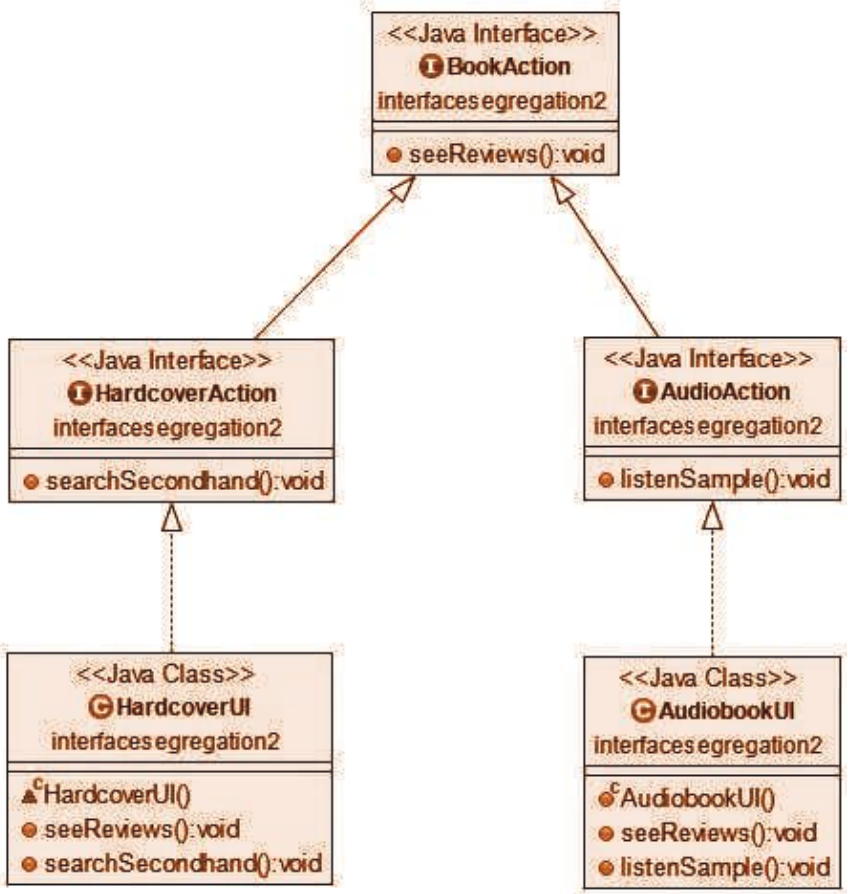
UML Class Diagram

Neither classes depend on methods they don't use. The UML diagram below excellently shows that the segregated interfaces lead to simpler classes that only implement the methods they really need:

Bad Design



Good Design



Object Oriented Analysis and Design with Java

ISP and SRP

Single Responsibility Principle is concerned with classes.

Interface Segregation Principle is concerned with interfaces.

Interface Segregation Principle is easy to understand and simple to follow. But, identifying the distinct interfaces can sometimes be a challenge.

Hence, we require careful considerations to avoid the expansion of interfaces.

Therefore, while writing an interface, consider the possibility of implementation classes having different sets of behaviors.

If so, segregate the interface into multiple interfaces, each having a specific role.

THANK YOU

J.Ruby Dinakar

Department of Computer Science and Engineering

rubydinakar@pes.edu