**Liskov Substitution Principle**

The [Liskov Substitution Principle](https://www.oodesign.com/liskov-s-substitution-principle.html" \t "_blank) is the third principle of SOLID, represented by the letter “L”. It was [Barbara Liskov](https://en.wikipedia.org/wiki/Barbara_Liskov) who introduced the principle in 1987 in her conference keynote talk “Data Abstraction”. The original phrasing of the Liskov Substitution Principle is a bit complicated, as it asserts that:

*“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. So, a child class should never change the characteristics of its parent class (such as the argument list and return types). You can implement the Liskov Substitution Principle by paying attention to the correct inheritance hierarchy.

**Example of the Liskov Substitution Principle**

Now, the book store asks us to add a new delivery functionality to the application. So, we 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() {...}

}

However, the store also sells fancy hardcovers they only want to deliver to their high street shops. So, we create a new HardcoverDelivery subclass that extends BookDelivery and overrides the getDeliveryLocations() method with its own functionality:

class HardcoverDelivery extends BookDelivery {

@Override

void getDeliveryLocations() {...}

}

Later, the store asks us to create delivery functionalities for audiobooks, too. Now, we extend the existing BookDelivery class with an AudiobookDelivery subclass. But, when we want to override the getDeliveryLocations() method, we realize that audiobooks can’t be delivered to physical locations.

class AudiobookDelivery extends BookDelivery {

@Override

void getDeliveryLocations() {/\* can't be implemented \*/}

}

We could change some characteristics of the getDeliveryLocations() method, however, 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.

To solve the problem, we need to fix the inheritance hierarchy. Let’s introduce an extra layer that better differentiates book delivery types. The new OfflineDelivery and OnlineDelivery classes split up the BookDelivery superclass. We also move the getDeliveryLocations() method to OfflineDelivery and create a new getSoftwareOptions() method for the OnlineDelivery class (as this is more suitable for online deliveries).

class BookDelivery {

String title;

int userID;

}

class OfflineDelivery extends BookDelivery {

void getDeliveryLocations() {...}

}

class OnlineDelivery extends BookDelivery {

void getSoftwareOptions() {...}

}

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 which is good news, as 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).

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.

On the UML graph below, you can see that by applying the Liskov Substitution Principle, we added an extra layer to the inheritance hierarchy. While the new architecture is more complex, it provides us with a more flexible design.

