The [Open/Closed Principle](https://www.oodesign.com/open-close-principle.html) is the “O” of SOLID’s five software design principles. It was [Bertrand Meyer](https://en.wikipedia.org/wiki/Bertrand_Meyer) who coined the term in his book “Object-Oriented Software Construction”. The Open/Closed Principle states that classes, modules, microservices, and other code units should be open for extension but closed for modification.

So, you should be able to extend your existing code using OOP features like inheritance via subclasses and interfaces. However, you should never modify classes, interfaces, and other code units that already exist (especially if you use them in production), as it can lead to unexpected behavior. If you add a new feature by extending your code rather than modifying it, you minimize the risk of failure as much as possible. Besides, you also don’t have to unit test existing functionalities.

**Example of the Open/Closed Principle**

Let’s stay with our book store example. Now, the store wants to hand out cookbooks at a discount price before Christmas. We already follow the Single Responsibility Principle, so we create two separate classes: CookbookDiscount to hold the details of the discount and DiscountManager to apply the discount to the price.

class CookbookDiscount {

String getCookbookDiscount() {

String discount = "30% between Dec 1 and 24.";

return discount;

}

}

class DiscountManager {

void processCookbookDiscount(CookbookDiscount discount) {...}

}

This code works fine until the store management informs us that their cookbook discount sales were so successful that they want to extend it. Now, they want to hand out every biography with a 50% discount on the subject’s birthday. To add the new feature, we create a new BiographyDiscount class:

class BiographyDiscount {

String getBiographyDiscount() {

String discount = "50% on the subject's birthday.";

return discount;

}

}

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. Although the above code works properly, it might add new vulnerabilities to the application. We don’t know how the new addition would interact with other parts of the code that depends on the DiscountManager class. In a real-world application, this would mean that we need to test and deploy our entire app again.

But, we can also choose to refactor our code by adding an extra layer of abstraction that represents all types of discounts. So, let’s create a new interface called BookDiscount that the CookbookDiscount and BiographyDiscount classes will implement.

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;

}

}

Now, DiscountManager can refer to the BookDiscount interface instead of the concrete classes. When the processBookDiscount() method is called, we can pass both CookbookDiscount and BiographyDiscount as an argument, as both are the implementation of the BookDiscount interface.

class DiscountManager {

void processBookDiscount(BookDiscount discount) {...}

}

The refactored code follows the Open/Closed principle, as we could add the new CookbookDiscount class without modifying the existing code base. This also means that in the future, we can extend our app with other discount types (for instance, with CrimebookDiscount).

The UML graph below shows how our example code looks like before and after the refactoring. On the left, you can see that DiscountManager depends on the CookbookDiscount and BiographyDiscount classes. On the right, all three classes depend on the BookDiscount abstract layer (DiscountManager references it, while CookbookDiscount and BiographyDiscount implement it).

