SOLID Principles:
Open/Closed Principle

by Noura Bensaber

In this series, I will highlight some areas where you can improve the quality of your code during the development process. In the first article, I explained why code quality is paramount. Then I introduced briefly some ideas that would help you improve the quality of your code. I will provide more details on these points in future articles. Some concepts will be illustrated with examples in Java and Python. I will begin this journey with you by introducing the SOLID principles in this article. They are some of the concepts that will help you improve the quality of your code and boost your abstract thinking in the object-oriented paradigm. After introducing the Single Responsibility Principle in the previous article, I will elucidate the Open/Closed Principle in this article, with examples in Java and Python.

1. Open/Closed Principle (OCP)

This principle means that software entities such as modules, classes, or functions should be open to extension but closed to modification. In other words, it is allowed to extend the behavior of these entities but it is prohibited to modify their source code. The reasons behind this principle are:

- Existing entities are already well-tested. Therefore, any modification could lead to bugs and unexpected behavior. It is better to avoid changing what already works well as much as possible. You can extend it whenever you need instead of modifying it.

- Testers do not need to test the entire flow, just the added parts.

- Maintenance remains easy.

- Single Responsibility Principle will be respected.

2. Example of OCP in Java

To see this principle in action, consider a well-known example that calculates the total price of a shopping cart. As a shopping cart contains products, we create an interface named 'Product' with a single 'getPrice()' method. Next, we suggest the 'Item' class to implement this interface. This class has three fields 'name', 'description', and 'price'. We have overridden the 'getPrice()' method to return the price value.

```
public interface Product {
    double getPrice();
}

public class Item implements Product {
    private final String name;
    private final String description;
    private final double price;
```

```
    public Item(String name, String description, double price) {
        this.name = name;
        this.description = description;
        this.price = price;
    }

    @Override
    public double getPrice() {
        return price;
    }
}
```

We can now create the 'ShoppingCart' class with some fields like the currency used and the list of products to purchase. These two fields must be initialized at the time of instantiation of the class. We need two methods to fulfill the functionality of a shopping cart, 'addProduct' and 'getTotalPrice'. The 'addProduct' method allows you to add a product to the shopping cart.

The 'getTotalPrice()' method calculates the total price of a collection of objects of the 'Item' class in a shopping cart. If the list containing the items is empty, the method returns 0.

If the products list is not empty, it maps each item within the list to its price, then reduces the stream to a single value (the sum of all prices) using Double::sum. Finally, it uses the 'round' method to round the calculated total price to two decimal places.

```
public class ShoppingCart {
    protected final String currency;
    protected final List<Product> products;
    public ShoppingCart(String currency, List<Product> products) {
        this.currency = currency;
        this.products = products;
    }
    public String getCurrency() {
        return currency;
    }

    public void addProduct(Product product) {
        products.add(product);
    }

    public double getTotalPrice() {
        if (products.isEmpty()) return 0;
        //Calculate the total price
```

```
        double totalPrice = products.stream()
                                .map(product -> product.getPrice()).reduce(Double::sum).get();
        //Round the price to 2 decimal places
        return round(totalPrice, 2);
    }
}
```

It is common to apply discounts on products. But, from an architectural point of view, it is not a good idea to be applied directly to the 'ShoppingCart' but to its sub-class. Here the adherence to the Open/Closed Principle comes into play. We do not open the 'ShoppingCart' class to add this behavior, but we extend it with a new sub-class to implement the new feature.

Now let's create a new class called 'DiscountedShoppingCart' to implement the new behavior of applying discounts to specific items based on certain criteria. This class has two fields 'priceLimit' and 'discount' which need to be initialized during the class instantiation. The 'priceLimit' property determines the maximum price of a product, which must not be exceeded to apply a discount. The 'discount' property is the percentage of the discount.

The 'DiscountedShoppingCart' class will calculate the total price differently than its super-class. The sub-class will take into consideration the discount rule. Therefore, 'DiscountedShoppingCart' will replace the 'getTotalPrice' method with a new behavior.

The ‘getTotalPrice’ method calculates the total price of a shopping cart with a discount applied when certain conditions are met. It first calls the ‘getTotalPrice’ method of the super-class to get the base total price. It returns the base total price if the list of products in the shopping cart is empty. Otherwise, it calculates a discount based on the price of each product and a price limit (‘priceLimit’). It applies a discount rate (discount) to the price of each product if it is less than or equal to the ‘priceLimit’ value.

The discount obtained is subtracted from the base total price. The final total price is rounded to 2 decimal places using the ‘round’ and returned as a result.

```
public class DiscountedShoppingCart extends ShoppingCart {
    private final double priceLimit;
    private final double discount;
    public DiscountedShoppingCart(String currency, List<Product> products,
                                  double priceLimit, double discount) {
        super(currency, products);
        this.priceLimit = priceLimit;
        this.discount = discount;
    }

    @Override
    public double getTotalPrice() {
        double totalPrice = super.getTotalPrice();
        if (products.isEmpty()) return totalPrice;
```

```
        //Calculate the discount
        double discount = products.stream()
                                .filter(p->p.getPrice()<=this.priceLimit&&this.priceLimit>0)
                                .map(p->p.getPrice()*this.discount).reduce(Double::sum).get();
        totalPrice -= discount;
        //Round the price to 2 decimal places
        return round(totalPrice, 2);
    }
}
```

We suggest the 'ShoppingCartDemo' class, a simple demonstration of a shopping cart system. We create three items, put them in a shopping cart, and then print out the total price of the shopping cart in euros. We then create a discounted shopping cart, add the same items, and print the total discounted price. Finally, we print out the discounted total price.

```
public class ShoppingCartDemo {
    public static void main(String[] args) {
        //Create some items
        Product item1 = new Item("Shampoo", "Herbal essences, 350ml for Dry hair", 10.0);
        Product item2 = new Item("Cream", "Nivea Soft, 100ml", 31.99);
        Product item3 = new Item("Tablet", "iPad 6 (2020) | 9.7", 149.99);
```

```
        //Create a shopping cart
        ShoppingCart shoppingCart = new ShoppingCart("EUR", new ArrayList<>());
        shoppingCart.addProduct(item1);
        shoppingCart.addProduct(item2);
        shoppingCart.addProduct(item3);
        System.out.println("Total price: " + shoppingCart.getTotalPrice()
                                + " " + shoppingCart.getCurrency());
        //Create a discounted shopping cart
        DiscountedShoppingCart discShoppingCart =
                new DiscountedShoppingCart("EUR", new ArrayList<>(), 50.0, 0.25);
        discShoppingCart.addProduct(item1);
        discShoppingCart.addProduct(item2);
        discShoppingCart.addProduct(item3);

        System.out.println("Discounted Total price: " + discShoppingCart.getTotalPrice()
                                + " " + discShoppingCart.getCurrency());
    }
}
```

The execution of this code gives us the following results:

Total price: 191.98 EUR

Discounted Total price: 181.48 EUR

In summary, the 'ShoppingCart' class is closed for modification and the 'DiscountedShoppingCart' class is open for extension to add new functionality to apply discounts to certain products, without modifying the existing 'ShoppingCart' class.

3. Example of OCP in Python

We can demonstrate the open/closed principle in Python using a famous example of the 'Account' and 'SaveAccount' classes. A simple implementation of the 'Account' class can have two properties, 'acc_num' for the account name and the 'balance' property for the amount of money available.

Two methods are necessary for each account, 'deposit' and 'withdraw'. The 'deposit' method consists of saving money in the account. The 'withdraw' method involves withdrawing money from the account. To show the account number and balance, we define the 'show_state' method.

```
class Account:
    def __init__(self, acc_num, balance):
        self.acc_num = acc_num
        self.balance = balance
```

```
    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
        else:
            print("Your balance is not enough.")

    def show_state(self):
        return f"Account number: {self.acc_num}, Balance: {self.balance}"
```

To keep the 'Account' class representing the base account that encapsulates the common behavior of all accounts, we need to keep it closed and avoid any changes that might disrupt this correct functionality. But bank customers need other services, like saving money for a long time and earning interest. To do this, we extend our 'Account' class with a new class to complete this new feature.

Let's create the 'SaveAccount' class that inherits from the 'Account' class and adds an interest rate property. We define a new method with the name 'add_interest' to calculate interest based on the balance and interest rate and add it to the balance.

We can say that this example adheres to the Open/Closed Principle because it extends the system with the 'SaveAccount' class without modifying the existing 'Account' class.

```
class SaveAccount(Account):
    def __init__(self, acc_num, balance, interest_rate):
        super().__init__(acc_num, balance)
        self.interest_rate = interest_rate

    def add_interest(self):
        interest = self.balance * self.interest_rate / 100
        self.balance += interest

    def show_state(self):
        return f"Saved " + super().show_state() + f", Interest Rate: {self.interest_rate}%"

# Some example operations on account
acc = Account("A952 7693 4621 7834", 7500.0)
print(acc.show_state())
acc.withdraw(350)
print(acc.show_state())
acc.deposit(1500.5)
print(acc.show_state())
```

```
print('')


# Some example of operations on save account
save_acc = SaveAccount("A5793 7693 4621 7834", 5700.0, 2.5)
print(save_acc.show_state())
save_acc.add_interest()
print(save_acc.show_state())
```

Running the above code gives us the following result:

```
Account number: A952 7693 4621 7834, Balance: 7500.0
Account number: A952 7693 4621 7834, Balance: 7150.0
Account number: A952 7693 4621 7834, Balance: 8650.5

Saved Account number: A5793 7693 4621 7834, Balance: 5700.0, Interest Rate: 2.5%
Saved Account number: A5793 7693 4621 7834, Balance: 5842.5, Interest Rate: 2.5%
```