DZone

The Open/Closed Principle and Strategy Pattern

by Ioan Tinca · Apr. 02, 18 · Java Zone · Tutorial

SOLID are a set of principles formulated by Robert C.Martin that are meant to guide developers to design and implement software that is easily maintainable, clear, and scalable. In other words, following these principles helps us to write a more solid software. These five principles are:

- The Single Responsibility Principle: A class should have one, and only one, reason to change.
- The Open Closed Principle: Software entities should be open to extension but closed to modification.
- The Liskov Substitution Principle: Derived classes must be substitutable for their base classes.
- The Interface Segregation Principle: Make fine-grained interfaces that are client specific.
- The Dependency Inversion Principle: Depend on abstraction, not on concretion.

In this article, I want to focus on the second one: The Open/Closed Principle.

Open to Extension

I highly doubt that there are too many software projects that don't suffer any changes from the time they were designed. Software design is not a straightforward process. This is just a utopian thought in this industry. Any project will suffer some changes, especially in an agile environment. And even if the project is not developed in an agile environment, to design it perfectly from the beginning is almost impossible. At any time, we might need to add new things or have modifications to do, and if the existing components are not open for extension, then any change would

imply a big risk.

One of our responsibilities as software developers is to anticipate what could change in what we write. We must focus to find the right abstraction level and the right point of behavior extension. We should not close our code to future extension and tie it to the current behavior because the behavior could always change and evolve. We should anticipate what could change and what could be extended. This does not sound too pragmatic, as this principle doesn't tell us how to do that, but this doesn't mean that there aren't some good practices for respecting the principle.

So keep in mind when you write a software component to make it open to extension.

But Closed for Modification?

If the scope of this principle is to grow the application's maintainability, why should the components be closed for modification? Every software entity should know how to do what it is designed to do and to do it well. For example, the Collections.sort method knows how to sort everything that implements the Comparable interface. This method is not limited to sorting just integers or just strings — it is not limited to any specific type. If you have a collection of objects that implement the Comparable interface, then you can sort it using the Collections.sort method. The sorting algorithm will work as it was designed, so we can say that it is closed to modification, but the sorting criteria will vary depending on the compareTo method implementation

The implementation of any software entity should be closed for modification. If the behavior changes, we should not change how a specific entity works, we just need to extend it. Just think of the enormous number of software applications that depend on the sort method. It is tested in the real application, it works good, and it is optimal. So if we need to sort a list of another type, should we change the sort method? Of course not!

So its implementation is closed to modification. This is how any software entity should be. But the key point is to let it be open to extension.

General Repeatable Solution to a Commonly Occurring Problem

The OCP is just a principle and not a generic solution. It describes what your entities should respect, but it does not provide a specific solution. The good guys that designed the sort method let it be open to extension by using a Comparable interface. The method sorts a list of Comparable objects and uses the compareTo method as a sorting criterion. But this is just an example. There isn't a single way of respecting this principle. However, there are some good general patterns that help us to achieve this.

Program by Interface, not by Implementation

For example, if the parameter types of a method are concrete classes, then the method is tightly coupled to those classes. It can't receive anything else, except instances of that specific type. In this case, the method is not easily open to extension.

Any method should be simple. It should use a single level of abstraction and should do just one thing. If it respects this, then I'm pretty sure it doesn't need to call all the parameters' objects methods. In this case, should the method declare a concrete class as a parameter type? No.

You could create an interface that the class implements and set it as the parameter type — exactly as the Collections.sort method. This way, you can use that method with any class that implements that specific interface. It will work the same. It will call the parameters' methods in the same way, but the behavior could be changed just by sending, as parameters, different implementations without changing the method.

Sure, you can extend that class and send, as parameters, instances of the child class, but since you cannot extend more than one class, it is more flexible (and clear) to just use interfaces.

The strategy pattern is a perfect example for programming by interface, not by implementation.

Design Patterns to the Rescue

A design pattern is a general repeatable solution to a commonly occurring problem. Like someone says here, the design pattern is a cure against diseases, and the diseases, in our case, are violations of SOLID principles. Design patterns are not the only cure, but they are an efficient one. Even if it is not the only design pattern that accomplishes the OCP, one specific pattern seems to be, by definition, extremely fit for this purpose, and that is the strategy pattern. Via the strategy pattern, you encapsulate some specific strategies and select which one to use at runtime according to some criteria. So, by definition, using this pattern makes your code open to extension.

Strategy Pattern

Like I said before, programming by interface and not by implementation is a best practice that we can use to design and implement code open to extension. Also, programming by interface is the key factor of the strategy pattern. It is a behavioral pattern, and, proven by the industry, one of the most useful design patterns. The principle is very simple: Encapsulate the strategies and decide what to use depending on some specific conditions. Following the strategy pattern, the behavior is decoupled by the classes that use it. You can switch between strategies without any class change.

How Does it Work?

The principle is very simple. All the strategy classes must implement a specific strategy interface. The class that uses the strategies, called the context class, is not bound to those specific strategy classes, but it is tied to the strategy interface. The context class encapsulates a strategy that could be injected in multiple ways (using Dependency Injection or the factory pattern or using a simple if condition — see this article for an introduction to Dependency Injection). So this mechanism is open to extension by giving you a way to use different strategies. Meanwhile, it is

closed to modification, as the class that uses a strategy does not have to be changed no matter what the strategy is encapsulating.

Let's see a simple example to better understand it. We will need a strategy interface, a class that uses the strategies, the context class, and some implementation of the strategy interface.

The context class should know just one thing about the strategies — what methods to call. This is what all the strategies have in common, so we will have a strategy interface with just the common methods (in our case, just one method).

```
public interface Strategy {
    public void doSomething();
}
```

And a context class that encapsulates a strategy implementation and executes it.

```
public class Context() {
    private Strategy strategy;
    // we set the strategy in the constructor
    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public void executeTheStrategy() {
        this.strategy.doSomething();
    }
}
```

And let's create two implementations for the Strategy interface.

```
public class Strategy1 implements Strategy {
    public void doSomething() {
        System.out.println("Execute strategy 1");
    }
}
```

```
6
7   public class Strategy2 implements Strategy {
8       public void doSomething() {
9           System.out.println(“Execute strategy 2”);
10      }
11  }
```

Now we can bind this together. The idea is to send to the context class the strategy we want to run. Like I said before, you can use Dependency Injection or the factory pattern for this, but it’s out of this article’s scope, so let’s just make a simple Demo class to see how it works.

```
1   public class Demo() {
2       public static void main(String[] args) {
3           Context context = new Context(new Strategy1()); // we inject the Strategy1
4           context.executeTheStrategy();  // it will print “Execute strategy 1”;
5
6           context = new Context(new Strategy2());  // we inject the Strategy2
7           context.executeTheStrategy();  // it will print “Execute strategy 2”
8       }
9   }
```

So the context is decoupled from a specific strategy class. You could implement however many strategies you want and no matter how they work and what you want them to do, you don’t need to modify the context class. The context class knows just that it must call doSomething method and it is enough.

This is a trivial example, but the possibilities are unlimited. Just think of the advantages you get here, and it is not hard at all to follow this simple pattern. Just use interfaces and let a concrete class to know just what it needs to know about something by tying it to an interface instead to a concrete class. This way, you can extend the behavior just by implementing different strategies and without changing the context class's functionality.

Conclusion

When you want to write code that follows the OCP, you should not limit yourself just on the strategy pattern or to “program by interface, not by implementation.” By using these best practices, I just wanted to show the power of having something open to extension and closed to modification.

Like I said before, not following this principle is like a disease, but design patterns aren’t the only cure. Strive to find the right abstraction levels and don’t use more than one single level of abstraction in a method. Find the pointcut between those levels, separate the concerns, and see how you can extend the functionality without changing the context classes. And don’t forget to look at the other SOLID principles.

Like This Article? Read More From DZone

Toss Out the Inheritance Forest

Factory Method vs. Simple Factory

Business Delegate Design Pattern[Video]

Free DZone Refcard
Java Containerization

Topics: JAVA , DESIGN PATTENS , SOLID , OPEN/CLOSED , STRATEGY PATTERN , TUTORIAL

IN PROGRESS