Open Closed Principle

As applications evolve, changes are required. Changes are required when a new functionality is added or an existing functionality is updated in the application. Often in both situations, you need to modify the existing code, and that carries the risk of breaking the application's functionality. For good application design and the code writing part, you should avoid change in the existing code when requirements change. Instead, you should extend the existing functionality by adding new code to meet the new requirements. You can achieve this by following the Open Closed Principle.

The Open Closed Principle represents the "O" of the five SOLID software engineering principles to write well-designed code that are more readable, maintainable, and easier to upgrade and modify. Bertrand Meyer coined the term Open Closed Principle, which first appeared in his book Object-Oriented Software Construction, release in 1988. This was about eight years before the initial release of Java.

This principle states: "*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification* ". Let's zero in on the two key phrases of the statement:

1. "*Open for extension* ": This means that the behavior of a software module, say a class can be extended to make it behave in new and different ways. It is important to note here that the term "*extended* " is not limited to inheritance using the Java `extend` keyword. As mentioned earlier, Java did not exist at that time. What it means here is that a module should provide extension points to alter its behavior. One way is to make use of polymorphism to invoke extended behaviors of an object at run time.
2. "*Closed for modification* ": This means that the source code of such a module remains unchanged.

It might initially appear that the phrases are conflicting- How can we change the behavior of a module without making changes to it? The answer in Java is abstraction. You can create abstractions (Java interfaces and abstract classes) that are fixed and yet represent an unbounded group of possible behaviors through concrete subclasses.

Before we write code which follows the Open Closed Principle, let's look at the consequences of violating the Open Closed principle.

Open Closed Principle Violation (Bad Example)

Consider an insurance system that validates health insurance claims before approving one. We can follow the complementary Single Responsibility Principle to model this requirement by creating two separate classes. A `HealthInsuranceSurveyor` class responsible to validate claims and a `ClaimApprovalManager` class responsible to approve claims.

HealthInsuranceSurveyor.java

```java
package guru.springframework.blog.openclosedprinciple;

public class HealthInsuranceSurveyor{

    public boolean isValidClaim(){
        System.out.println("HealthInsuranceSurveyor: Validating health insurance claim...");
        /*Logic to validate health insurance claims*/
        return true;
    }
}
```

ClaimApprovalManager.java

```java
package guru.springframework.blog.openclosedprinciple;

public class ClaimApprovalManager {

    public void processHealthClaim (HealthInsuranceSurveyor surveyor)
    {
        if(surveyor.isValidClaim()){
            System.out.println("ClaimApprovalManager: Valid claim. Currently processing claim for approval....");
        }
    }
}
```

Both the `HealthInsuranceSurveyor` and `ClaimApprovalManager` classes work fine and the design for the insurance system appears perfect until a new requirement to process vehicle insurance claims arises. We now need to include a new `VehicleInsuranceSurveyor` class, and this should not create any problems. But, what we also need is to modify the `ClaimApprovalManager` class to process vehicle insurance claims. This is how the modified `ClaimApprovalManager` will be:

Modified ClaimApprovalManager.java

```java
package guru.springframework.blog.openclosedprinciple;

public class ClaimApprovalManager {

    public void processHealthClaim (HealthInsuranceSurveyor surveyor)
    {
        if(surveyor.isValidClaim()){
            System.out.println("ClaimApprovalManager: Valid claim. Currently processing claim for approval....");
        }
    }

    public void processVehicleClaim (VehicleInsuranceSurveyor surveyor)
    {
        if(surveyor.isValidClaim()){
            System.out.println("ClaimApprovalManager: Valid claim. Currently processing claim for approval....");
        }
    }
}
```

In the example above, we modified the `ClaimApprovalManager` class by adding a new `processVehicleClaim()` method to incorporate a new functionality (claim approval of vehicle insurance).

As apparent, this is a clear violation of the Open Closed Principle. We need to modify the class to add support for a new functionality. In fact, we violated the Open Closed Principle at the very first instance we wrote the `ClaimApprovalManager` class. This may appear innocuous in the current example, but consider the consequences in an enterprise application that needs to keep pace with fast changing business demands. For each change, you need to modify, test, and deploy the entire application. That not only makes the application fragile and expensive to extend but also makes it prone to software bugs.

Coding to the Open Closed Principle

The ideal approach for the insurance claim example would have been to design the `ClaimApprovalManager` class in a way that it remains:

- **Open** to support more types of insurance claims.
- **Closed** for any modifications whenever support for a new type of claim is added.

To achieve this, let's introduce a layer of abstraction by creating an abstract class to represent different claim validation behaviors. We will name the class `InsuranceSurveyor`.

InsuranceSurveyor.java

```java
package guru.springframework.blog.openclosedprinciple;


public abstract class InsuranceSurveyor {
    public abstract boolean isValidClaim();
}
```

Next, we will write the specific classes for each type of claim validation.

HealthInsuranceSurveyor.java

```java
package guru.springframework.blog.openclosedprinciple;


public class HealthInsuranceSurveyor extends InsuranceSurveyor{
    public boolean isValidClaim(){
        System.out.println("HealthInsuranceSurveyor: Validating health insurance claim...");
        /*Logic to validate health insurance claims*/
        return true;
    }

}
```

VehicleInsuranceSurveyor.java

```java
package guru.springframework.blog.openclosedprinciple;


public class VehicleInsuranceSurveyor extends InsuranceSurveyor{
    public boolean isValidClaim(){
       System.out.println("VehicleInsuranceSurveyor: Validating vehicle insurance claim...");
        /*Logic to validate vehicle insurance claims*/
        return true;
    }

}
```

In the examples above, we wrote the `HealthInsuranceSurveyor` and `VehicleInsuranceSurveyor` classes that extend the abstract `InsuranceSurveyor` class. Both classes provide different implementations of the `isValidClaim()` method. We will now write the `ClaimApprovalManager` class to follow the Open/Closed Principle.

ClaimApprovalManager.java

```java
package guru.springframework.blog.openclosedprinciple;


public class ClaimApprovalManager {
    public void processClaim(InsuranceSurveyor surveyor){
        if(surveyor.isValidClaim()){
            System.out.println("ClaimApprovalManager: Valid claim. Currently processing claim for approval....");
        }
    }

}
```

In the example above, we wrote a `processClaim()` method to accept a `InsuranceSurveyor` type instead of specifying a concrete type. In this way, any further addition of `InsuranceSurveyor` implementations will not affect the `ClaimApprovalManager` class. Our insurance system is now **open** to support more types of insurance claims, and **closed** for any modifications whenever a new claim type is added. To test our example, let's write this unit test.

ClaimApprovalManagerTest.java

```java
package guru.springframework.blog.openclosedprinciple;

import org.junit.Test;

import static org.junit.Assert.*;


public class ClaimApprovalManagerTest {

    @Test
    public void testProcessClaim() throws Exception {
      HealthInsuranceSurveyor healthInsuranceSurveyor=new HealthInsuranceSurveyor();
      ClaimApprovalManager claim1=new ClaimApprovalManager();
      claim1.processClaim(healthInsuranceSurveyor);

```

```
        VehicleInsuranceSurveyor vehicleInsuranceSurveyor=new VehicleInsuranceSurveyor();
        ClaimApprovalManager claim2=new ClaimApprovalManager();
        claim2.processClaim(vehicleInsuranceSurveyor);
    }
}
```

The output is:

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.2.3.RELEASE)

Running guru.springframework.blog.openclosedprinciple.ClaimApprovalManagerTest
HealthInsuranceSurveyor: Validating health insurance claim...
ClaimApprovalManager: Valid claim. Currently processing claim for approval....
VehicleInsuranceSurveyor: Validating vehicle insurance claim...
ClaimApprovalManager: Valid claim. Currently processing claim for approval....
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 sec - in guru.springframework.blog.openclosedprinciple.ClaimApprovalManagerTest
```

Summary

Most of the times real closure of a software entity is practically not possible because there is always a chance that a change will violate the closure. For example, in our insurance example a change in the business rule to process a specific type of claim will require modifying the ClaimApprovalManager class. So, during enterprise application development, even if you might not always manage to write code that satisfies the Open Closed Principle in every aspect, taking the steps towards it will be beneficial as the application evolves.

Get The Code

I've committed the source code for this post to github. It is a Maven project which you can download and build. If you wish to learn more about the Spring Framework, I have a free introduction to Spring tutorial. You can sign up for this tutorial in the section below.

Source Code

The source code for this post is available on github. You can download it here.

4 comments on “Open Closed Principle”

Myo Myint November 22, 2015 at 11:11 pm # Reply

how about to used the interface instead of abstract class? can u explain me what is the difference?

Robert February 14, 2016 at 6:23 pm # Reply

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. We should be able to extend software entities without actually modifying them. This is a crucial principle of SOLID and continuous integration. We usually apply this principle to OOP and we should apply it in every day javascript programming as well. The following video starting from the definition of this principle, identifies some common code smell and puts it into practice with a javascript example https://youtu.be/t7RgyY9OOd0

Krzysztof July 28, 2017 at 3:49 pm # Reply

Hello. Thanks for the post. Would it not be better in the test to use same instance of the ClaimApprovalManager rahter than creating new one for each case? After all, the example is to show, that we can accept anything that follows InsuranceSurveyor type.