SOLID
Open Closed Principle

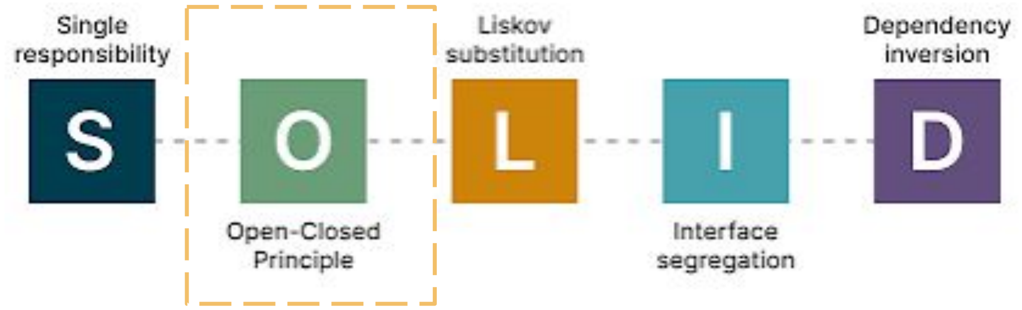
Upcode Software
Engineer Team

-2023-

CONTENT

1. What is **OCP** ?
2. WHY needs to **OCP** ?
3. HOW to use **OCP** ?
4. WHERE to use **OCP** ?
5. SOURCE code

1. What is OCP ? (1/n)



- In object-oriented programming, the **open–closed principle** (OCP) states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"; that is, such an entity can allow its behaviour to be extended without modifying its source code.
- The name **open–closed principle** has been used in two ways. Both ways use generalizations (for instance, inheritance or delegate functions) to resolve the apparent dilemma, but the goals, techniques, and results are different.
- Open–closed principle is one of the five SOLID principles of object-oriented design.

1. What is OCP ? (2/n)

- Everyone who starts learning any programming language is introduced to many theoretical concepts such as object-oriented paradigm, SOLID principles, clean code, design patterns, and much more.
- All these subjects, individually, represent a huge challenge in terms of transforming them into actual implementation in real-scenarios once it requires, sometimes, experience and flexibility.
- It is not different with SOLID principles in general.
- So, this post has the purpose of explaining the Open-Closed principle using a real-scenario example, demonstrating the importance of it in order to avoid issues and bugs in software maintenance over time.

2. WHY needs to OCP ?

- **The purpose** of those principles is to allow developers to write better software.
- The software is **easier and cheaper to maintain,** easier to understand, faster to develop in a team, and easier to test.
- Following those **principles doesn’t guarantee success,** but avoiding them will lead in most cases to at least **sub-optimal results in terms of functionality**, cost, or both.
- It's now time for the O in SOLID, known as the **open-closed principle.** Simply put, **classes should be open for extension but closed for modification.**
- **In doing so, we stop ourselves from modifying existing code and causing potential new bugs** in an otherwise happy application.

3. HOW to use OCP ? (1/n)

- **Open for extension** – This means that the module's behavior can be extended. As the requirements of the application change, we can extend the module with new behaviors that satisfy those changes. In other words, *we are able to change what the module does*.
- **Closed for modification** – Extending the behavior of a module does not result in changes to the source or binary code of the module. The *binary executable version* of the module, whether in a linkable library, a DLL, or a Java .jar, remains untouched. Abstractions shouldn't depend on details. Details should depend on abstractions.

Open-Closed Principal

Classes should be open for extension
but closed for modification.

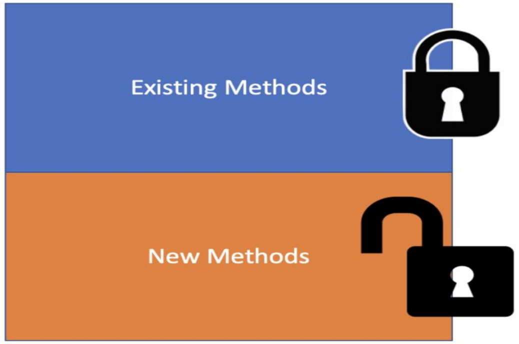
3. HOW to use OCP ? (2/n)

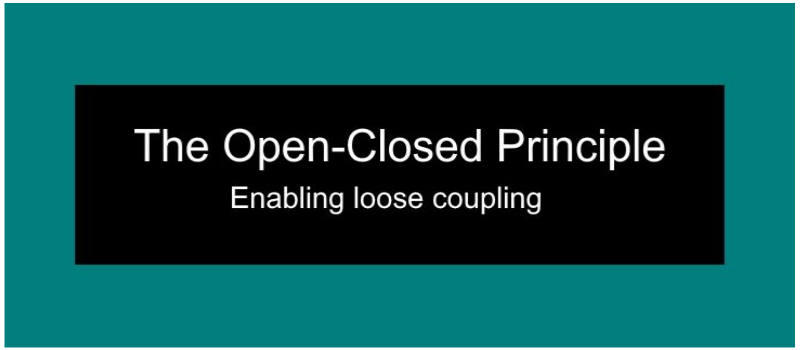
Why Should You Apply the Open/Closed Principle?

- Software systems that are not carefully designed, tend to develop an increasing amount of unwanted interdependencies.
- Often these interdependencies are accidental, and they accumulate to such an extent that it becomes increasingly hard to keep track of them.
- Changes in one part of such a codebase often necessitate changes somewhere else. These changes, in turn, require modifications in another part of the system leading to a domino effect.
- If you have to change the code in various parts of the codebase to fulfill a new requirement, your system will become unstable and increasingly behave in a non-deterministic manner.

3. HOW to use OCP ? (3/n)

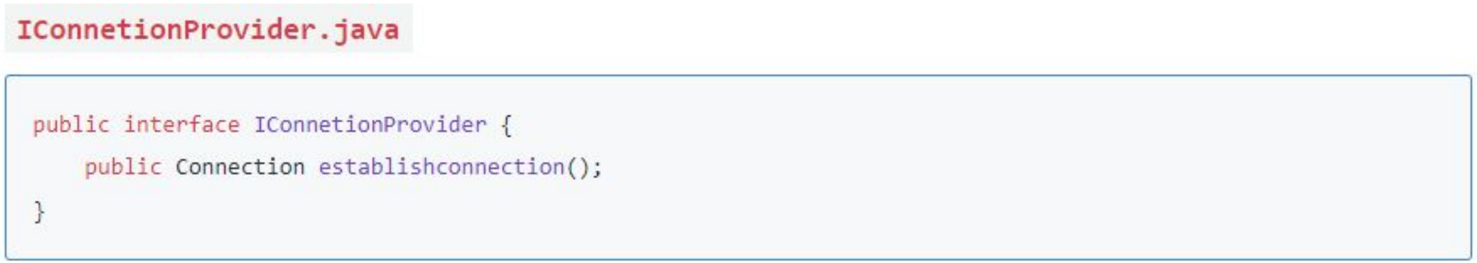
- The open/closed principle helps you avoid such a scenario by requiring you to design your system in a manner that makes modifications largely unnecessary.
- All extensions to the functionality of the system should be achieved by adding new code rather than changing existing code. This way, you are much less likely to run into unwanted ripple effects that necessitate changes in several parts of the system and lead to instability.
- This all sounds very abstract.





4. WHERE to use OCP ? (1/n)

- Let's take the same example of Database Connection Provider, ideal we can have a connection to any RDBMS data source as per our requirements.
- Let's design or develop source code using this design principle.
- Code for the interface so first create IConnetionProvider interface and create a separate class to implement this interface for each database connection like H2ConnectionProvider, MySQLConnectionProvider.



IConnetionProvider.java

```java
public interface IConnetionProvider {
    public Connection establishconnection();
}
```

4. WHERE to use OCP ? (2/n)

`H2ConnectionProvider.java`

```java
public class H2ConnectionProvider implements IConnetionProvider{

    public Connection establishconnection() {
        // TODO  : provide connection for H2 database and return the connection object
         return null;
    }
}
```

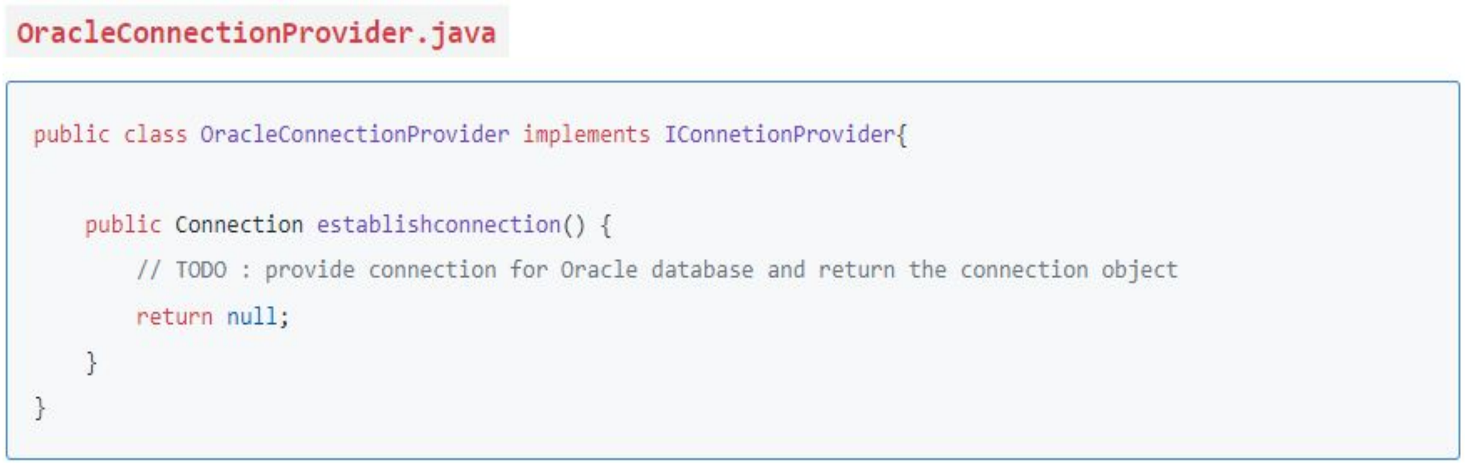
`MySQLConnectionProvider.java`

```java
public class MySQLConnectionProvider implements IConnetionProvider{

    public Connection establishconnection() {
        // TODO : provide connection for MySQL database
        return null;
    }
}
```

4. WHERE to use OCP ? (3/n)

- In future, if Client wants to connect MS-Server or Oracle then we need to create the class like MsServerConnectionProvider or OracleConnectionProvider and just implement IConnectionProviderinterface that's it.



OracleConnectionProvider.java

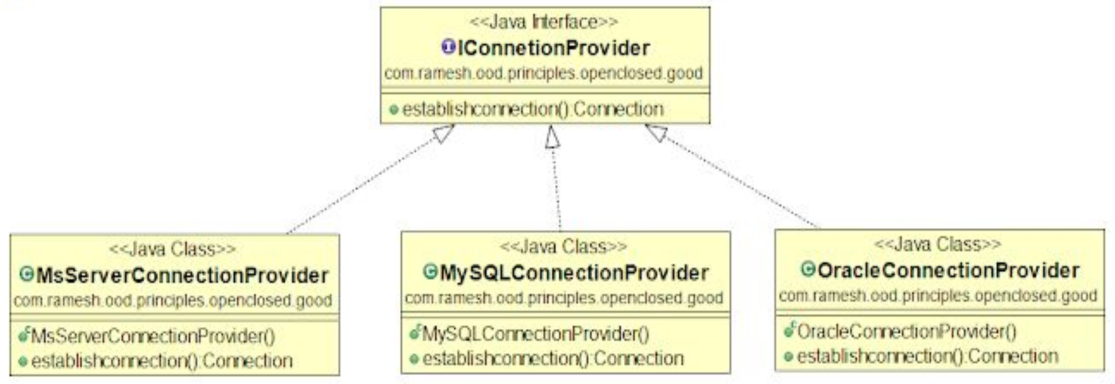
```
public class OracleConnectionProvider implements IConnetionProvider{

    public Connection establishconnection() {
        // TODO : provide connection for Oracle database and return the connection object
        return null;
    }
}
```

4. WHERE to use OCP ? (4/n)

- We are not modifying existing code here, we are just extending as per our requirement.

Class diagram



<<Java Interface>>
IConnetionProvider
com.ramesh.ood.principles.openclosed.good
establishconnection():Connection
<<Java Class>>
MsServerConnectionProvider
com.ramesh.ood.principles.openclosed.good
MsServerConnectionProvider()
establishconnection():Connection
<<Java Class>>
MySQLConnectionProvider
com.ramesh.ood.principles.openclosed.good
MySQLConnectionProvider()
establishconnection():Connection
<<Java Class>>
OracleConnectionProvider
com.ramesh.ood.principles.openclosed.good
OracleConnectionProvider()
establishconnection():Connection

5. SOURCE code (1/n)

- **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();
}
```

5. SOURCE code (2/n)

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

HealthInsuranceSurveyor.java

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


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

5. SOURCE code (3/n)

VehicleInsuranceSurveyor.java

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


public class VehicleInsuranceSurveyor extends InsuranceSurveyor{
    @Override
    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.

5. SOURCE code (4/n)

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.

5. SOURCE code (5/n)

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);
   }
}
```

5. SOURCE code (6/n)

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

- 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.

Reference Resources

1. What is OCP?
2. WHY needs to OCP ?
3. Open Closed principle.
4. Dive into design pattern compression (book).

Thank you!

Presented by **Sherjonov Jakhongir**

jakhongirsherjonov@gmail.com