# اصل Open/Close Principle(OCP) چیست؟

اصل OCP می‌گوید که کلاس‌ها باید برای توسعه (Extension) باز (Open to extension) و در برابر تغییرات (Modification) مقاوم و بسته (Closed for modification) باشند.
هدف این اصل این است که به کلاس‌ها اجازه دهیم به راحتی توسعه پیدا کنند و رفتارهای جدید را بپذیرند، بدون اینکه کد موجود تغییر کند. به عبارت دیگر، طراحی سیستم باید به‌گونه‌ای باشد که نیازهای جدید را با توسعه دادن سیستم پوشش دهیم، نه با تغییر یا اصلاح کدهای موجود.

**چرا رعایت OCP مهم است؟**

زمانی که طراحی سیستم بر اساس OCP انجام شود، تغییرات بیزینسی باعث **شکستن** کد یا **تغییر زیاد** در بخش‌های دیگر سیستم نمی‌شود. این رویکرد باعث انعطاف‌پذیری و پایداری سیستم می‌شود و امکان اضافه کردن مسئولیت‌های جدید بدون تأثیر بر کدهای موجود را فراهم می‌کند.

نکته: در طراحی سیستم‌های مدرن امروزی، نوشتن سناریوهای مختلف تست از اهمیت ویژه‌ای برخوردار است. رعایت اصول طراحی، مانند اصلOCP ، به پایداری سیستم کمک شایانی می‌کند. این امر باعث می‌شود که تست‌های قبلی بدون نیاز به تغییر همچنان معتبر باقی بمانند و بتوان برای کدهای جدیدی که توسعه داده شده‌اند، سناریوهای تست مجزایی طراحی کرد. چنین رویکردی ضمن افزایش قابلیت اطمینان سیستم، فرآیند نگهداری و توسعه آن را نیز ساده‌تر و کم‌هزینه‌تر می‌کند.

**چگونه می‌توان OCP را رعایت کرد؟**

رعایت OCP معمولاً نیازمند اضافه کردن سطوح جدیدی از انتزاع (Abstraction) است. این امر می‌تواند کد را پیچیده‌تر کند، اما در مقابل، تغییرات آینده را ساده‌تر می‌سازد. با این حال، نباید این اصل را در تمام بخش‌های سیستم پیاده‌سازی کنیم. تمرکز باید بر بخش‌هایی باشد که احتمال تغییر در آن‌ها بیشتر است.

برای شناسایی این بخش‌ها، به دو عامل نیاز داریم:

۱. **تجربه در طراحی سیستم‌های شی‌گرا :(Object-Oriented Design)** تجربه به ما کمک می‌کند تا نقاط حساس به تغییر را شناسایی کنیم.

۲. **شناخت دامنه :(Domain Knowledge)** آشنایی با حوزه‌ی کاری کمک می‌کند تا بخش‌هایی از سیستم که احتمال تغییر بیشتری دارند شناسایی شوند.

**نمونه‌ای از کاربردOCP**

جاهایی از بیزینس که به‌طور مکرر دستخوش تغییرات می‌شوند، بهترین مکان برای اعمال OCP هستند. به عنوان مثال، در سیستم‌هایی که روی یک داده، روش‌های پردازشی متفاوتی را ارایه میدهند مثل فرمت های مختلف گزارش دهی یا فرمت های مختلف ارسال مرسوله به مشتری یا روش های مختلف محاسباتی، می‌توان از این اصل استفاده کرد.

**الگوهای طراحی مرتبط باOCP**

بعضی از الگوهای طراحی به خوبی از این اصل پشتیبانی می‌کنند، از جمله:

- **Strategy Pattern**: برای جداسازی الگوریتم‌ها و انتخاب دینامیک آن‌ها.
- **Decorator Pattern**: برای اضافه کردن رفتارهای جدید به اشیا، بدون تغییر کلاس اصلی.

**جمع‌بندی**

اصل OCP یکی از مهم‌ترین اصول طراحی شی‌گرا است که باعث افزایش انعطاف‌پذیری و کاهش وابستگی‌ها در سیستم می‌شود. اگرچه رعایت کامل این اصل در تمام بخش‌های سیستم ممکن نیست، اما با تمرکز بر نقاط حساس به تغییر و استفاده از الگوهای طراحی مناسب می‌توان به سیستم‌هایی قابل توسعه و پایدار دست یافت.

**نکته پایانی** :مطالعه مثال‌های دیگر و بررسی پیاده‌سازی‌های واقعی به شما کمک می‌کند تا مهارت بیشتری در شناسایی و اعمال OCP در طراحی‌های خود پیدا کنید.

بریم یک مثالی در این زمینه ببینیم.

یک سیستمی را می خواهیم طراحی کنیم که نیازمندیهای اولیه مشتری به این صورت می باشد که عملیات های جمع و تفریق را بتونیم انجام بدیم.

در این طراحی، کلاس CalculatorProcessor مسئول پردازش عملیات‌های ریاضی (Addition,Subtraction) است. یعنی مدیریت بیزینس را به عهده دارد و تمرکز اصلی ما روی این کلاس می باشد که می خواهیم به بهترین حالت ممکن پیاده سازی شود.

طراحی را ببینیم که اصل OCP در ان رعایت نشده است.

```
package mehdi.sample.core.solid.ocp.without;

public interface CalculatorOperation {
}
```

```
package mehdi.sample.core.solid.ocp.without;

public class Addition implements CalculatorOperation{
   private double left;
   private double right;

   public Addition(double left, double right) {
      this.left = left;
      this.right = right;
   }

   public double getLeft() {
      return left;
   }

   public double getRight() {
      return right;
   }
}
```

```
package mehdi.sample.core.solid.ocp.without;

public class Subtraction implements CalculatorOperation{
   private double left;
   private double right;

   public Subtraction(double left, double right) {
      this.left = left;
      this.right = right;
   }

   public double getLeft() {
```

```
        return left;
    }

    public double getRight() {
        return right;
    }
}


package mehdi.sample.core.solid.ocp.without;

import java.security.InvalidParameterException;

public class CalculatorProcessor {
    private final CalculatorOperation operation;
    private double result;

    public CalculatorProcessor(CalculatorOperation calculatorOperation) {
        this.operation = calculatorOperation;
    }

    public void processCalculate() {
        if (operation == null) {
            throw new InvalidParameterException("Can not perform operation");
        }

        if (operation instanceof Addition) {
            Addition addition = (Addition) operation;
            this.result = addition.getLeft() + addition.getRight();
        } else if (operation instanceof Subtraction) {
            Subtraction subtraction = (Subtraction) operation;
            this.result = subtraction.getLeft() - subtraction.getRight();
        }
    }

    public double getResult() {
        return result;
    }
}
```

این قسمت را در نظر بگیرید.

داخل بیزینس از instanceof استفاده شده است و این یعنی وابستگی کلاس به المان های دیگر و اینجا دقیقا جایی است که این کد با تغییر نیازمندیهای بیزینس مثل اضافه شدن عملیات ضرب باید تغییر کند و اصل OCP را نقض می کند.

**مشکلات طراحی:**

۱. **وابستگی کلاس CalculatorProcessor به انواع عملیات:**

- کلاس CalculatorProcessor که داره بیزینس اقبه انواع خاصی از عملیات وابسته است (Addition, Subtraction).
- برای اضافه کردن یک عملیات جدید (مثل ضرب)، باید کد CalculatorProcessor تغییر کند که اصل OCP را نقض می‌کند.

۲. **استفاده از :instanceof**

- این روش باعث افزایش وابستگی به پیاده‌سازی‌های خاص می‌شود.

۳. **کاهش انعطاف‌پذیری سیستم:**

- با توجه به این طراحی، هر بار که نیاز به افزودن عملیات جدید باشد، تغییراتی در کد پردازشگر (CalculatorProcessor) ایجاد می‌شود که می‌تواند منجر به بروز باگ‌های جدید شود.

بریم سراغ یک طراحی بهتر که اصل OCP را رعایت کرده است.

```
public interface CalculatorOperation {
   public Double perform();
}

public class Addition implements CalculatorOperation {
   private double left;
   private double right;

   public Addition(double left, double right) {
      this.left = left;
      this.right = right;
   }

   public double getLeft() {
      return left;
   }

   public double getRight() {
      return right;
   }

   @Override
   public Double perform() {
      return this.left + this.right;
   }
}


package mehdi.sample.core.solid.ocp.with;

public class Subtraction implements CalculatorOperation {
   private double left;
   private double right;

   public Subtraction(double left, double right) {
      this.left = left;
      this.right = right;
   }
```

```
    public double getLeft() {
        return left;
    }

    public double getRight() {
        return right;
    }

    @Override
    public Double perform() {
        return this.left - this.right;
    }
}


package mehdi.sample.core.solid.ocp.with;

import java.security.InvalidParameterException;

public class CalculatorProcessor {
    private final CalculatorOperation operation;
    private double result;

    public CalculatorProcessor(CalculatorOperation calculatorOperation) {
        this.operation = calculatorOperation;
    }

    public void processCalculate() {
        if (operation == null) {
            throw new InvalidParameterException("Can not perform operation");
        }

        result = operation.perform();
    }

    public double getResult() {
        return result;
    }
}
```

```
package mehdi.sample.core.solid.ocp.with;

public class CalculatorDemo {
   public static void main(String[] args) {
      CalculatorOperation additionOperation = new Addition(12,13);
      CalculatorProcessor calculatorProcessor = new CalculatorProcessor(additionOperation);
      calculatorProcessor.processCalculate();
      System.out.println(calculatorProcessor.getResult());
   }
}
```

**نقاط قوت طراحی:**

۱. **رعایت اصل:OCP (Open/Closed Principle)**

- کلاس CalculatorProcessor به جای وابستگی به انواع خاص عملیات (مثل Addition ) به رابط CalculatorOperation وابسته است.
- اگر بخواهید یک عملیات جدید (مثل ضرب یا تقسیم) اضافه کنید، تنها کافی است یک کلاس جدید ایجاد کرده و رابط CalculatorOperationرا پیاده‌سازی کنید. نیازی به تغییر در کدهای موجود نیست. این دقیقاً هدف OCP است.

۲. **کاهش وابستگی به جزئیات:**

- در این طراحی، پردازشگر عملیات (CalculatorProcessor) فقط رابط CalculatorOperationرا می‌شناسد و از جزئیات پیاده‌سازی بی‌اطلاع است. این باعث کاهش وابستگی و افزایش انعطاف‌پذیری می‌شود.

۳. **حذف نیاز به :instanceof**

- با انتقال منطق اجرای عملیات به متد ()perform در هر کلاس، نیاز به استفاده از instanceof در کد حذف شده است. این کار طراحی را تمیزتر و قابل گسترش‌تر می‌کند.

شاد باشید

مهدی شیوایی فر