**Open-closed Principle**

The open-closed principle is a software engineering principle that states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means that the behavior of the entity can be extended or modified without changing the source code of the entity itself.

This principle helps to achieve a high level of maintainability and flexibility in software design. It enables developers to add new features and functionality to the software without breaking existing code, reducing the risk of introducing bugs and ensuring that the software can evolve and adapt to changing requirements over time.

The open-closed principle is often used in conjunction with other design principles, such as the single responsibility principle and the dependency inversion principle, to create modular, reusable, and extensible software.

Example:

For example, consider a class that represents a shape, such as a rectangle or a circle. If we follow the open-closed principle, we would design the class in such a way that it can be extended to support new shapes (e.g. a triangle or a square) without modifying the existing code.

For instance, we could define the class with an abstract method for calculating the area of the shape, and implement this method in each of the concrete subclasses that represent specific shapes. Then, when we need to add support for a new shape, we can simply create a new subclass that implements the abstract method. This way, we can add new functionality without modifying the existing code, which helps to ensure that the software remains maintainable and extensible over time.

Here is an example implementation of the shape class in Java:

In this implementation, the **Shape** class is designed to be open for extension (we can add new shapes by extending this class) but closed for modification (we don't need to modify the existing code in order to add new shapes). This way, we can easily add support for new shapes without breaking the existing code.

```java
public abstract class Shape {
  // Abstract method for calculating the area of the shape
  public abstract double calculateArea();
}

public class Rectangle extends Shape {
  private double width;
  private double height;
  public Rectangle(double width, double height) {
    this.width = width;
    this.height = height;
  }
  @Override
  public double calculateArea() {
    return width * height;
  }
}

public class Circle extends Shape {
  private double radius;
  public Circle(double radius) {
    this.radius = radius;
  }
  @Override
  public double calculateArea() {
    return Math.PI * radius * radius;
  }
}
```