Professional Skills

Agile Fundamentals

Jira

Git

Databases Introduction

Java Beginner

Maven

Testing (Foundation)

Java Intermediate

- ○ Optionals
- ○ JDBC CRUD
- ○ Exceptions
- ○ SOLID Principles
- ○ Single Responsibility
- ○ Open/Closed
- ○ Liskov Substituiton
- ○ Interface Segregation
- ○ Dependency Inversion
- ○ Best Practice
- ○ Design Patterns
- ○ Creational Design Patterns
- ○ Structural Design Patterns
- ○ Behavioural Design Patterns
- ○ Collection & Map
- ○ HashSets
- ○ HashMaps
- ○ Enums
- ○ Logging
- ○ Generics
- ○ Lambda Expressions
- ○ Streams
- ○ Complexity
- ○ Input and Output
- ○ Local Type Inference

HTML

CSS

Open/Closed

Contents

Overview

In object-oriented programming, the second of the **SOLID Principles** is **O** - which stands for **Open/Closed**.

When talking about the way in which our classes are laid out, we generally say that they should be *open for extension* but *closed for modification*.

You want developers to extend and add to your functionality, *without having to directly modify your classes* - this is usually because our previous code has gone through testing, and we can be sure that it works!

Essentially, we should try to write code that doesn't have to be changed every single time the requirements change.

The most common way to *extend functionality* while adhering to the Open/Closed Principle is to use **inheritance** (sub-classes) and **polymorphism**.

Open/Closed In Action

The `AreaCalculator.java` class

Let's say that we need to calculate the area of several different shapes. We'll start with a `Rectangle` and a `Circle`:

▶ Rectangle

▶ Circle

We'll now make an `AreaCalculator` which works out the area of a shape depending on what it is:

▶ AreaCalculator

While this solution works fine, if we start adding more shapes into the mix, we're going to find ourselves constantly changing `AreaCalculator` to keep up.

If we keep having to edit the code, then the class isn't *closed for modification*.

Because we're having to add more functionality into the `AreaCalculator.java` class, rather than by using sub-classes and inheritance, it isn't *open for extension* either!

Fixing the `AreaCalculator.java` class

The best way for us to fix this issue is to make an interface called `Shape`. This is just a base type which we can *implement* into any new shape class we define:

▶ Shape

This interface means that any object which implements `Shape` will automatically have a `calculateArea()` function, which can then be tailored depending on the class it's being used in:

- Javascript
- Spring Boot
- Selenium
- Sonarqube
- Advanced Testing (Theory)
- Cucumber
- MongoDB
- Express
- NodeJS
- React
- Express-Testing
- Networking
- Security
- Cloud Fundamentals
- AWS Foundations
- AWS Intermediate
- Linux
- DevOps
- Jenkins Introduction
- Jenkins Pipeline
- Markdown
- IDE Cheatsheet

▶ Rectangle

▶ Circle

Since we're no longer editing a single class to account for every shape, our interface is *closed for modification* - but because every new shape we make is implementing its own way of dealing with the `Shape` interface, it's also *open for extension*.

We're essentially pointing the program to `Shape` instead of the individual classes for each shape whenever we want it to calculate an area - now we can edit `AreaCalculator` to point to `Shape` as well:

▶ AreaCalculator

Tutorial

There is no tutorial for this module.

Exercises

Greeter

Consider the god-class `Greeter.java`, which is meant to return a greeting message depending on the type selected:

▶ Greeter

`Greeter.java` violates the Open/Closed Principle, because any time we would want to change the style of greeting, we would have to change the functionality of `greet()` every single time.

Implement the following classes and/or interfaces to complete this exercise:

- `Greeting.java`
- `FormalGreeting.java`
- `CasualGreeting.java`
- `Greeter.java`

This is an implementation of one of the Design Patterns which you may have already looked at - which one is it?

▶ Show solution