SOLID Design Principles Explained: The Open/Closed Principle with Code Examples

stackify.com/solid-design-open-closed-principle

March 28, 2018

The Open/Closed Principle is one of five design principles for object-oriented software development described by Robert C. Martin. They are best known as the SOLID principles:

All 5 of these design principles are broadly used, and all experienced software developers should be familiar with them. But don't worry, if you haven't heard about them yet. I had been working as a software developer for a few years before I learned about the SOLID principles and quickly recognized that they described the rules and principles my coworkers had taught me about writing good code. So, even if you don't know them by name, you might be already using them.

But that doesn't mean that we shouldn't talk and learn about the SOLID principles. In this article, I will focus on the Open/Closed Principle, and I will explain the other principles in future articles.

Definition of the Open/Closed Principle

Robert C. Martin considered this principle as the "the most important principle of object-oriented design". But he wasn't the first one who defined it. Bertrand Meyer wrote about it in 1988 in his book Object-Oriented Software Construction. He explained the Open/Closed Principle as:

> "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

The general idea of this principle is great. It tells you to write your code so that you will be able to add new functionality without changing the existing code. That prevents situations in which a change to one of your classes also requires you to adapt all depending classes. Unfortunately, Bertrand Mayer proposes to use inheritance to achieve this goal:

> "A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as parent, adding new features. When a descendant class is defined, there is no need to change the original or to disturb its clients."

But as we've learned over the years and as other authors explained in great details, e.g., Robert C. Martin in his articles about the SOLID principles or Joshua Bloch in his book Effective Java, inheritance introduces tight coupling if the subclasses depend on implementation details of their parent class.

That's why Robert C. Martin and others redefined the Open/Closed Principle to the Polymorphic Open/Closed Principle. It uses interfaces instead of superclasses to allow different implementations which you can easily substitute without changing the code that uses them. The interfaces are closed for modifications, and you can provide new implementations to extend the functionality of your software.

The main benefit of this approach is that an interface introduces an additional level of abstraction which enables loose coupling. The implementations of an interface are independent of each other and don't need to share any code. If you consider it beneficial that two implementations of an interface share some code, you can either use inheritance or composition.

Let's take a look at an example that uses the Open/Closed Principle.

Brewing coffee with the Open/Closed Principle

You can buy lots of different coffee machines. There are relatively basic ones that just brew filter coffee, and others that include grinders to brew different kinds of coffee, e.g., espresso and filter coffee. All of them serve the same purpose: They brew delicious coffee which wakes us up in the morning.

The only problem is that you need to get out of bed to switch on the coffee machine. So, why not ignore all the challenges of the physical world, e.g., how to put water and ground coffee into the machine or how to put a mug under it without getting out of bed, and implement a simple program that serves you a freshly brewed coffee?

To show you the benefits of the Open/Closed Principle, I wrote a simple application that controls a basic coffee machine to brew you a delicious filter coffee in the morning.

The *BasicCoffeeMachine* class

The implementation of the *BasicCoffeeMachine* class is relatively simple. It just has a constructor, a public method to add ground coffee, and a method that brews a filter coffee.

```
import java.util.HashMap;
import java.util.Map;

public class BasicCoffeeMachine {

    private Map<CoffeeSelection, Configuration> configMap;
    private Map<CoffeeSelection, GroundCoffee>; groundCoffee;
    private BrewingUnit brewingUnit;

    public BasicCoffeeMachine(Map<CoffeeSelection, GroundCoffee> coffee) {
    this.groundCoffee = coffee;
    this.brewingUnit = new BrewingUnit();

    this.configMap = new HashMap<>();
        this.configMap.put(CoffeeSelection.FILTER_COFFEE, new Configuration(30, 480));
    }

    public Coffee brewCoffee(CoffeeSelection selection) {
    Configuration config = configMap.get(CoffeeSelection.FILTER_COFFEE);

    // get the coffee
    GroundCoffee groundCoffee = this.groundCoffee.get(CoffeeSelection.FILTER_COFFEE);

    // brew a filter coffee
    return this.brewingUnit.brew(CoffeeSelection.FILTER_COFFEE, groundCoffee,
config.getQuantityWater());
    }

    public void addGroundCoffee(CoffeeSelection sel, GroundCoffee newCoffee) throws
CoffeeException {
    GroundCoffee existingCoffee = this.groundCoffee.get(sel);
    if (existingCoffee != null) {
        if (existingCoffee.getName().equals(newCoffee.getName())) {
        existingCoffee.setQuantity(existingCoffee.getQuantity() + newCoffee.getQuantity());
        } else {
        throw new CoffeeException("Only one kind of coffee supported for each
CoffeeSelection.");
        }
    } else {
        this.groundCoffee.put(sel, newCoffee);
    }
    }
}
```

You can easily control such a simple coffee machine via an app, right? So, let's do that.

The *BasicCoffeeApp* class

The *main* method of the *BasicCoffeeApp* prepares a *Map* with ground coffee, instantiates a *BasicCoffeeMachine* object, and calls the *prepareCoffee* method to brew the coffee.

```
public class BasicCoffeeApp {

    private BasicCoffeeMachine coffeeMachine;

    public BasicCoffeeApp(BasicCoffeeMachine coffeeMachine) {
    this.coffeeMachine = coffeeMachine;
    }

    public Coffee prepareCoffee(CoffeeSelection selection) throws CoffeeException {
    Coffee coffee = this.coffeeMachine.brewCoffee(selection);
    System.out.println("Coffee is ready!");
    return coffee;
    }

    public static void main(String[] args) {
    // create a Map of available coffee beans
    Map<CoffeeSelection, GroundCoffee> beans = new HashMap<CoffeeSelection,
GroundCoffee>();
    beans.put(CoffeeSelection.FILTER_COFFEE, new GroundCoffee(
        "My favorite filter coffee bean", 1000));

    // get a new CoffeeMachine object
    BasicCoffeeMachine machine = new BasicCoffeeMachine(beans);

    // Instantiate CoffeeApp
    BasicCoffeeApp app = new BasicCoffeeApp(machine);

    // brew a fresh coffee
    try {
        app.prepareCoffee(CoffeeSelection.FILTER_COFFEE);
    } catch (CoffeeException e) {
        e.printStackTrace();
    }
    } // end main
} // end CoffeeApp
```

That's it. From now on, you can stay in bed until you smell the fresh coffee prepared by your *BasicCoffeeApp*.

Applying the Open/Closed principle

But what happens when you replace your *BasicCoffeeMachine*? You might get a better one with an integrated grinder, which can brew more than just filter coffee. Unfortunately, the *CoffeeApp* doesn't support this kind of coffee machine.

It would be great if your app could control both types of coffee machines. But that will require a few code changes. And as you're already on it, why not change it so that you will not need to adapt it to future coffee machines.

Extracting the *CoffeeMachine* interface

Following the Open/Closed Principle, you need to extract an interface that enables you to control the coffee machine. That's often the critical part of the refactoring. You need

to include the methods that are mandatory for controlling the coffee machine, but none of the optional methods which would limit the flexibility of the implementations.

In this example, that's only the *brewCoffee* method. So, the *CoffeeMachine* interface specifies only one method, which needs to be implemented by all classes that implement it.

```
public interface CoffeeMachine {

    Coffee brewCoffee(CoffeeSelection selection) throws CoffeeException;
}
```

Adapting the *BasicCoffeeMachine* class

In the next step, you need to adapt the *BasicCoffeeMachine* class. It already implements the *brewCoffee* method and provides all the functionality it needs. So, you just need to declare that the *BasicCoffeeMachine* class implements the *CoffeeMachine* interface.

```
public class BasicCoffeeMachine implements CoffeeMachine { ... }
```

Add more implementations

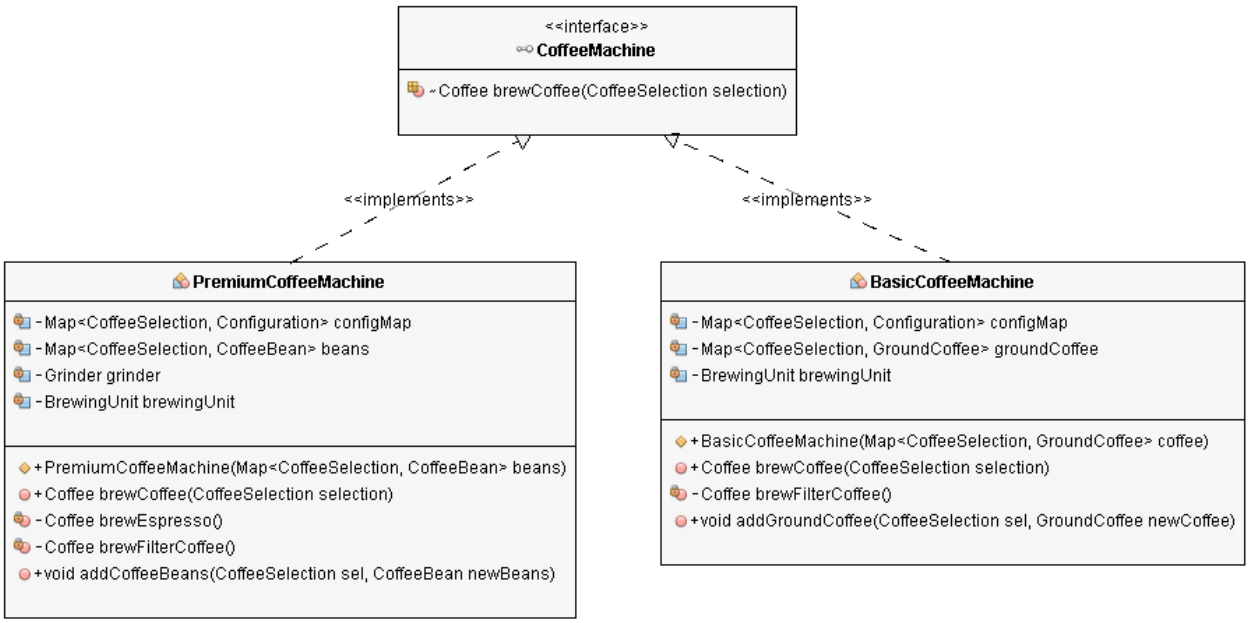
You can now add new implementations of the *CoffeeMachine* interface.

The implementation of the *PremiumCoffeeMachine* class is more complex than the *BasicCoffeeMachine* class. Its *brewCoffee* method, which is defined by the *CoffeeMachine* interface, supports two different *CoffeeSelection*s. Based on the provided *CoffeeSelection*, the method calls a separate, private method that brews the selected coffee. As you can see in the implementation of these methods, the class also uses composition to reference a *Grinder,* which grinds the coffee beans before brewing the coffee.

```
import java.util.HashMap;
import java.util.Map;

public class PremiumCoffeeMachine implements CoffeeMachine {

    private Map<CoffeeSelection, Configuration> configMap;
    private Map<CoffeeSelection, CoffeeBean> beans;
    private Grinder grinder;
    private BrewingUnit brewingUnit;

    public PremiumCoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans) {
    this.beans = beans;
    this.grinder = new Grinder();
    this.brewingUnit = new BrewingUnit();

    this.configMap = new HashMap<>();
    this.configMap.put(CoffeeSelection.FILTER_COFFEE, new Configuration(30, 480));
    this.configMap.put(CoffeeSelection.ESPRESSO, new Configuration(8, 28));
    }
```

```
    @Override
    public Coffee brewCoffee(CoffeeSelection selection) throws CoffeeException {
    switch(selection) {
    case ESPRESSO:
        return brewEspresso();
    case FILTER_COFFEE:
        return brewFilterCoffee();
    default:
        throw new CoffeeException("CoffeeSelection [" + selection + "] not supported!");
    }
    }

    private Coffee brewEspresso() {
    Configuration config = configMap.get(CoffeeSelection.ESPRESSO);

    // grind the coffee beans
    GroundCoffee groundCoffee = this.grinder.grind(
        this.beans.get(CoffeeSelection.ESPRESSO),
            config.getQuantityCoffee());

    // brew an espresso
    return this.brewingUnit.brew(CoffeeSelection.ESPRESSO, groundCoffee,
        config.getQuantityWater());
    }

    private Coffee brewFilterCoffee() {
    Configuration config = configMap.get(CoffeeSelection.FILTER_COFFEE);

    // grind the coffee beans
    GroundCoffee groundCoffee = this.grinder.grind(
        this.beans.get(CoffeeSelection.FILTER_COFFEE),
            config.getQuantityCoffee());

    // brew a filter coffee
    return this.brewingUnit.brew(CoffeeSelection.FILTER_COFFEE, groundCoffee,
        config.getQuantityWater());
    }

    public void addCoffeeBeans(CoffeeSelection sel, CoffeeBean newBeans) throws
CoffeeException {
    CoffeeBean existingBeans = this.beans.get(sel);
    if (existingBeans != null) {
        if (existingBeans.getName().equals(newBeans.getName())) {
            existingBeans.setQuantity(existingBeans.getQuantity() + newBeans.getQuantity());
        } else {
        throw new CoffeeException("Only one kind of coffee supported for each
CoffeeSelection.");
        }
    } else {
        this.beans.put(sel, newBeans);
    }
    }
}
```

You're done with most of the refactoring work. You applied the Open/Closed Principle by introducing the *CoffeeMachine* interface and providing two independent implementations of it.



The only thing that's left is the app to use different implementations of that interface.

Adapting the CoffeeApp

The *CoffeeApp* class consists of 2 parts:

1. the *CoffeeApp* class and
2. the *main* method

You need to instantiate a specific *CoffeeMachine* implementation in the *main* method. So, you will always need to adopt this method, if you replace your current coffee machine. But as long as the *CoffeeApp* class uses the *CoffeeMachine* interface, you will not need to adapt it.

```
import java.util.HashMap;
import java.util.Map;

public class CoffeeApp {

    private CoffeeMachine coffeeMachine;

    public CoffeeApp(CoffeeMachine coffeeMachine) {
    this.coffeeMachine = coffeeMachine;
    }

    public Coffee prepareCoffee(CoffeeSelection selection) throws CoffeeException {
    Coffee coffee = this.coffeeMachine.brewCoffee(selection);
    System.out.println("Coffee is ready!");
    return coffee;
    }

    public static void main(String[] args) {
    // create a Map of available coffee beans
    Map<CoffeeSelection, CoffeeBean>; beans = new HashMap<CoffeeSelection, CoffeeBean>
();
    beans.put(CoffeeSelection.ESPRESSO, new CoffeeBean(
        "My favorite espresso bean", 1000));
    beans.put(CoffeeSelection.FILTER_COFFEE, new CoffeeBean(
        "My favorite filter coffee bean", 1000));

    // get a new CoffeeMachine object
    PremiumCoffeeMachine machine = new PremiumCoffeeMachine(beans);

    // Instantiate CoffeeApp
    CoffeeApp app = new CoffeeApp(machine);

    // brew a fresh coffee
    try {
        app.prepareCoffee(CoffeeSelection.ESPRESSO);
    } catch (CoffeeException e) {
        e.printStackTrace();
    }
    } // end main
} // end CoffeeApp
```

Summary

After taking a closer look at the Single Responsibility Principle in the previous post of this series, we now discussed the Open/Closed Principle. It is one of the five SOLID design principle described by Robert C. Martin. It promotes the use of interfaces to enable you to adapt the functionality of your application without changing the existing code.

We used this principle in the example application to control different kinds of coffee machines via our *CoffeeApp*. As long as a coffee machine implements the *CoffeeMachine* interface, you can control it via the app. The only thing you need to do

when you replace your existing coffee machine is to provide a new implementation of the interface and change the main method which instantiates the specific implementation. If you want to take it one step further, you can use dependency injection, reflection or the service loader API to replace the instantiation of a specific class.