SOLID Design Principles Explained: The Liskov Substitution Principle with Code Examples

stackify.com/solid-design-liskov-substitution-principle

April 11, 2018

The Open/Closed Principle, which I explained in a previous article, is one of the key concepts in OOP that enables you to write robust, maintainable and reusable software components. But following the rules of that principle alone is not enough to ensure that you can change one part of your system without breaking other parts. Your classes and interfaces also need to follow the Liskov Substitution Principle to avoid any side-effects.

The Liskov Substitution Principle is the 3rd of Robert C. Martin's famous SOLID design principles:

It extends the Open/Closed Principle by focusing on the behavior of a superclass and its subtypes. As I will show you in this article, this is at least as important but harder to validate that the structural requirements of the Open/Closed Principle.

Definition of the Liskov Substitution Principle

The Liskov Substitution principle was introduced by Barbara Liskov in her conference keynote "Data abstraction" in 1987. A few years later, she published a paper with Jeanette Wing in which they defined the principle as:

> Let *Φ(x)* be a property provable about objects *x* of type *T*. Then *Φ(y)* should be true for objects *y* of type *S* where *S* is a subtype of *T*.

OK, let's be honest. Such a scientific definition might be necessary, but it doesn't help a lot in our daily work as software developers. So, what does it mean for our code?

The Liskov Substitution Principle in practical software development

The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. That requires the objects of your subclasses to behave in the same way as the objects of your superclass. You can achieve that by following a few rules, which are pretty similar to the design by contract concept defined by Bertrand Meyer.

An overridden method of a subclass needs to accept the same input parameter values as the method of the superclass. That means you can implement less restrictive validation rules, but you are not allowed to enforce stricter ones in your subclass. Otherwise, any code that calls this method on an object of the superclass might cause an exception, if it gets called with an object of the subclass.

Similar rules apply to the return value of the method. The return value of a method of the subclass needs to comply with the same rules as the return value of the method of the superclass. You can only decide to apply even stricter rules by returning a specific subclass of the defined return value, or by returning a subset of the valid return values of the superclass.

Enforcing the Liskov Substitution Principle

If you decide to apply this principle to your code, the behavior of your classes becomes more important than its structure. Unfortunately, there is no easy way to enforce this principle. The compiler only checks the structural rules defined by the Java language, but it can't enforce a specific behavior.

You need to implement your own checks to ensure that your code follows the Liskov Substitution Principle. In the best case, you do this via code reviews and test cases. In your test cases, you can execute a specific part of your application with objects of all subclasses to make sure that none of them causes an error or significantly changes its performance. You can try to do similar checks during a code review. But what's even more important is that you check that you created and executed all the required test cases.

Okay, enough theory. Let's take a look at an example

Making coffee with the Liskov Substitution Principle

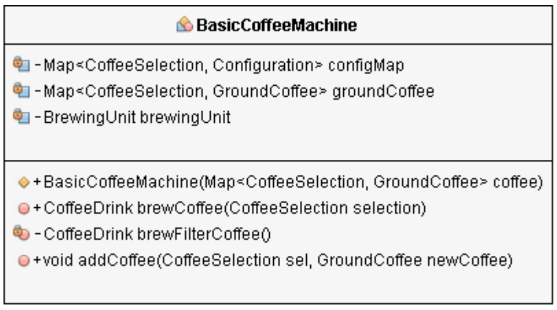
Most articles about the Liskov Substitution Principle use an example in which they implement a *Rectangle* and a *Square* class to show that you break the design principle if your *Square* class extends the *Rectangle* class.

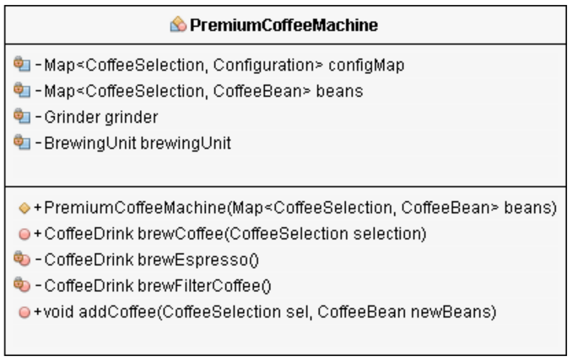
But that example is a little bit boring. There are already lots of articles about it, and I have never implemented an application that just requires a set of simple geometric shapes. So, let's create an example that's a little bit more fun.

I enjoy drinking a good cup of coffee in the morning, and I want to show you a simple application that uses different kinds of coffee machines to brew a cup of coffee. You might already know very similar examples from my previous articles about the Single Responsibility Principle or the Open/Closed Principle. You can get all source files of this example at https://github.com/thjanssen/Stackify-SOLID-Liskov.

If you enjoy coffee as much as I do, you most likely used several different coffee machines in the past. There are relatively basic ones that you can use to transform one or two scoops of ground coffee and a cup of water into a nice cup of filter coffee. And there are others that include a grinder to grind your coffee beans and you can use to brew different kinds of coffee, like filter coffee and espresso.

If you decide to implement an application that automatically brews a cup of coffee every morning so that you don't have to get out of bed before it's ready, you might decide to model these coffee machines as two classes with the methods *addCoffee* and *brewCoffee*.


BasicCoffeeMachine
-Map<CoffeeSelection, Configuration> configMap
-Map<CoffeeSelection, GroundCoffee> groundCoffee
-BrewingUnit brewingUnit
+BasicCoffeeMachine(Map<CoffeeSelection, GroundCoffee> coffee)
+CoffeeDrink brewCoffee(CoffeeSelection selection)
-CoffeeDrink brewFilterCoffee()
+void addCoffee(CoffeeSelection sel, GroundCoffee newCoffee)


PremiumCoffeeMachine
-Map<CoffeeSelection, Configuration> configMap
-Map<CoffeeSelection, CoffeeBean> beans
-Grinder grinder
-BrewingUnit brewingUnit
+PremiumCoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans)
+CoffeeDrink brewCoffee(CoffeeSelection selection)
-CoffeeDrink brewEspresso()
-CoffeeDrink brewFilterCoffee()
+void addCoffee(CoffeeSelection sel, CoffeeBean newBeans)

A basic coffee machine

The *BasicCoffeeMachine* can only brew filter coffee. So, the *brewCoffee* method checks if the provided *CoffeeSelection* value is equal to *FILTER_COFFEE* before it calls the private *brewFilterCoffee* method to create and return a *CoffeeDrink* object.

```
public class BasicCoffeeMachine {

    private Map configMap;
    private Map groundCoffee;
    private BrewingUnit brewingUnit;

    public BasicCoffeeMachine(Map coffee) {
        this.groundCoffee = coffee;
        this.brewingUnit = new BrewingUnit();

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

    public CoffeeDrink brewCoffee(CoffeeSelection selection)
        throws CoffeeException {

        switch (selection) {
            case FILTER_COFFEE:
                return brewFilterCoffee();
            default:
                throw new CoffeeException(
                    "CoffeeSelection [" + selection + "] not supported!");
        }
    }

    private CoffeeDrink brewFilterCoffee() {
        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 addCoffee(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);
        }
    }
}
```

The *addCoffee* method expects a *CoffeeSelection* enum value and a *GroundCoffee* object. It uses the *CoffeeSelection* as the key of the internal *groundCoffee Map*.

These are the most important parts of the *BasicCoffeeMachine* class. Let's take a look at the *PremiumCoffeeMachine*.

A premium coffee machine

The premium coffee machine has an integrated grinder, and the internal implementation of the *brewCoffee* method is a little more complex. But you don't see that from the outside. The method signature is identical to the one of the *BasicCoffeeMachine* class.

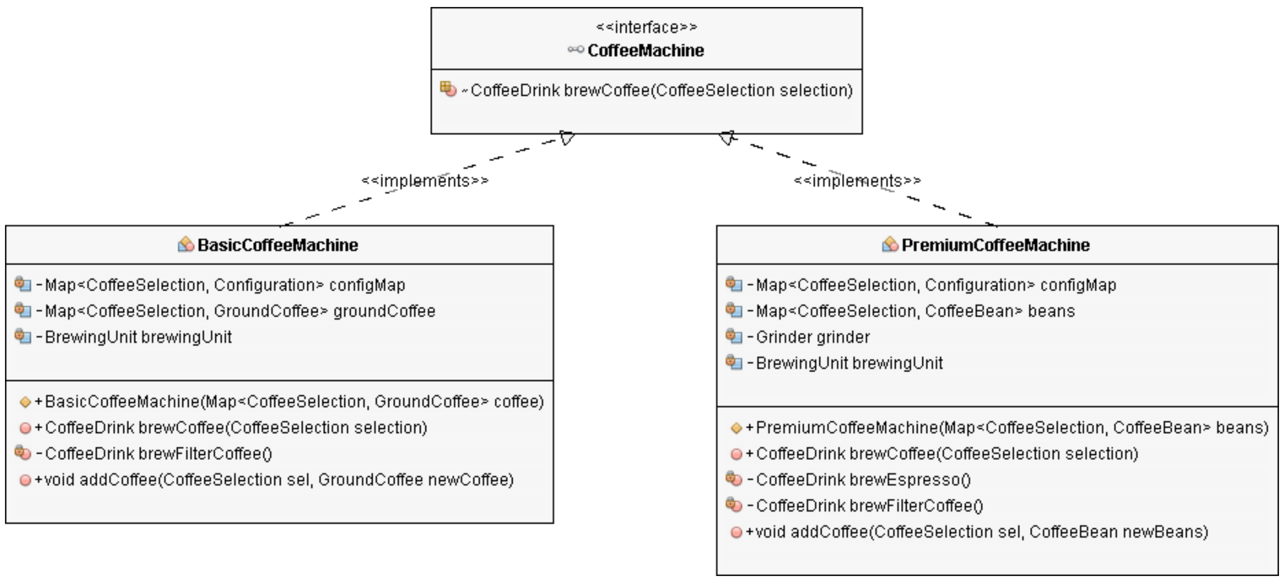
```
public class PremiumCoffeeMachine {

    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 CoffeeDrink 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 CoffeeDrink 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 CoffeeDrink 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 addCoffee(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);
        }
    }
}
```

But that's not the case for the *addCoffee* method. It expects an object of type *CoffeeBean* instead of an object of type *GroundCoffee*. If you add a shared superclass or an interface that gets implemented by the *BasicCoffeeMachine* and the *PremiumCoffeeMachine* class, you will need to decide how to handle this difference.

Introducing a shared interface

You can either create another abstraction, e.g., *Coffee*, as the superclass of *CoffeeBean* and *GroundCoffee* and use it as the type of the method parameter. That would unify the structure of both *addCoffee* methods, but require additional validation in both methods. The *addCoffee* method of the *BasicCoffeeMachine* class would need to check that the caller provided an instance of *GroundCoffee*, and the *addCoffee* implementation of the *PremiumCoffeeMachine* would require an instance of *CoffeeBean*. This would obviously break the Liskov Substitution Principle because the validation would fail if you provide a *BasicCoffeeMachine* object instead of a *PremiumCoffeeMachine* and vice versa.

The better approach is to exclude the *addCoffee* method from the interface or superclass because you can't interchangeably implement it. The *brewCoffee* method, on the other hand, could be part of a shared interface or a superclass, as long as the superclass or interface only guarantees that you can use it to brew filter coffee. The input parameter validation of both implementations accept the *CoffeeSelection* value *FILTER_COFFEE*. The *addCoffee* method of the *PremiumCoffeeMachine* class also accepts the enum value *ESPRESSO*. But as I explained at the beginning of this article, the different subclasses may implement less restrictive validation rules.



<<interface>>
CoffeeMachine
~CoffeeDrink brewCoffee(CoffeeSelection selection)
<<implements>>
<<implements>>
BasicCoffeeMachine
-Map<CoffeeSelection, Configuration> configMap
-Map<CoffeeSelection, GroundCoffee> groundCoffee
-BrewingUnit brewingUnit
+BasicCoffeeMachine(Map<CoffeeSelection, GroundCoffee> coffee)
+CoffeeDrink brewCoffee(CoffeeSelection selection)
-CoffeeDrink brewFilterCoffee()
+void addCoffee(CoffeeSelection sel, GroundCoffee newCoffee)
PremiumCoffeeMachine
-Map<CoffeeSelection, Configuration> configMap
-Map<CoffeeSelection, CoffeeBean> beans
-Grinder grinder
-BrewingUnit brewingUnit
+PremiumCoffeeMachine(Map<CoffeeSelection, CoffeeBean> beans)
+CoffeeDrink brewCoffee(CoffeeSelection selection)
-CoffeeDrink brewEspresso()
-CoffeeDrink brewFilterCoffee()
+void addCoffee(CoffeeSelection sel, CoffeeBean newBeans)

Summary

The Liskov Substitution Principle is the third of Robert C. Martin’s SOLID design principles. It extends the Open/Closed principle and enables you to replace objects of a parent class with objects of a subclass without breaking the application. This requires all subclasses to behave in the same way as the parent class. To achieve that, your subclasses need to follow these rules:

- Don’t implement any stricter validation rules on input parameters than implemented by the parent class.
- Apply at the least the same rules to all output parameters as applied by the parent class.