OOP Design Principles

SOLID

COMP3607
Object Oriented Programming II

1-Oct-2018

Outline

- SOLID Design Principles
  - Single Responsibility Principle
  - Open/Closed Principle
  - Liskov Substitution Principle
  - Interface Segregation Principle
  - Dependency Inversion

SOLID Design Principles

SOLID is one of the most popular sets of design principles in object-oriented software development. It's a mnemonic acronym for the following five design principles:

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion

Liskov Substitution Principle

https://deviq.com/liskov-substitution-principle/

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.*

Liskov Substitution Principle

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.

Liskov Substitution Principle - Methods-

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.

Liskov Substitution Principle -Values-

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

The behaviour 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 behaviour.

You therefore need to implement your own checks to ensure that your code follows the Liskov Substitution Principle.

Enforcing the Liskov Substitution Principle

Execute a specific part of your application with objects of all subclasses to make sure that none of them cause an error or significantly change the performance of your application.

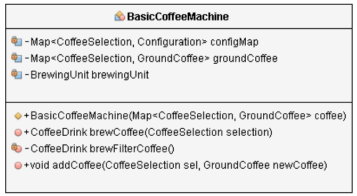
- Code reviews
- Test cases

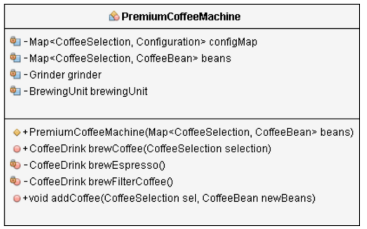
Example: Coffee Brewing

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.

Example: Coffee Brewing





Example: Coffee Brewing
- 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 CoffeeDrink brewCoffee(CoffeeSelection selection)
    throws CoffeeException {

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


Example: Coffee Brewing
- A Basic Coffee Machine-

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

```
    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 CoffeeSelec
tion.");
            }
        } else {
            this.groundCoffee.put(sel, newCoffee);
        }
    }
}
```


Example: Coffee Brewing
- A Premium Coffee Machine-

The premium coffee machine has an integrated grinder.

```
public class PremiumCoffeeMachine {

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


Example: Coffee Brewing
- A Premium Coffee Machine-

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.

```
@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!");
    }
}
```

Example: Coffee Brewing
- A Premium Coffee Machine-

But that's not the case for the *addCoffee* method. It expects an object of type *CoffeeBean* instead of an object of type *GroundCoffee*.

```
    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 CoffeeSelec
tion.");
            }
        } else {
            this.beans.put(sel, newBeans);
        }
    }
}
```


Example: Coffee Brewing
- A Premium Coffee Machine-

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.

```
    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 CoffeeSelec
tion.");
            }
        } else {
            this.beans.put(sel, newBeans);
        }
    }
}
```


Example: Coffee Brewing
- 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

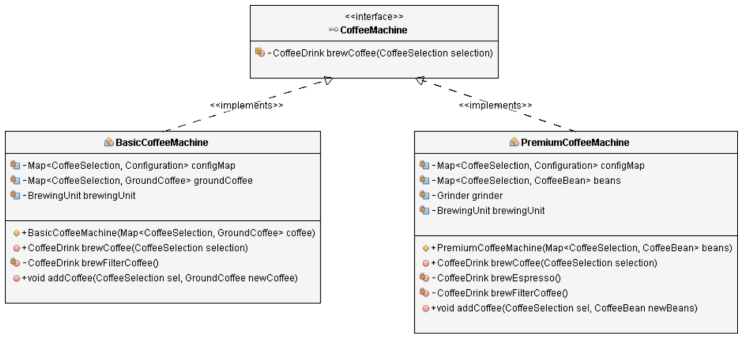
Example: Coffee Brewing
- Introducing a shared interface-

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*. The different subclasses may then implement less restrictive validation rules.

Example: Coffee Brewing
- Introducing a shared interface-



Enforcing the
Liskov Substitution Principle

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.

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

Interface Segregation Principle

https://deviq.com/interface-segregation-principle/

Interface Segregation Principle

"Clients should not be forced to depend upon interfaces that they do not use." - Robert C. Martin

Interface Segregation Principle

The goal of the Interface Segregation Principle is to reduce the side effects and frequency of required changes by splitting the software into multiple, independent parts.

This is only achievable if you define your interfaces so that they fit a specific client or task.

Violating the Interface Segregation Principle

This happens quite often when an application gets used for multiple years, and when its users regularly request new features.

The implementation of each change bears a risk.

It's tempting to add a new method to an existing interface even though it implements a different responsibility and would be better separated in a new interface.

That's often the beginning of interface pollution, which sooner or later leads to bloated interfaces that contain methods implementing several responsibilities.

Simple Example: Coffee Brewing

The project used the BasicCoffeeMachine class to model a basic coffee machine which uses ground coffee to brew a delicious filter coffee.

At that time, it was perfectly fine to *extract the CoffeeMachine interface* with the methods *addGroundCoffee* and *brewFilterCoffee*.

Simple Example: Coffee Brewing

These are the two essential methods of a coffee machine and should be implemented by all future coffee machines.

```
public interface CoffeeMachine {
    CoffeeDrink brewFilterCoffee() throws CoffeeException;
    void addGroundCoffee(GroundCoffee newCoffee) throws CoffeeException;
}
```

Simple Example: Coffee Brewing -Polluting the Interface-

But then somebody decided that the application also needs to support espresso machines. The development team modeled it as the *EspressoMachine* class that was pretty similar to the *BasicCoffeeMachine* class.

The developer decided that an espresso machine is just a different kind of coffee machine. So, it has to implement the *CoffeeMachine* interface.

The only difference is the *brewEspresso* method, which the *EspressoMachine* class implements instead of the *brewFilterCoffee* method.

Simple Example: Coffee Brewing -Polluting the Interface-

```
@Override
public CoffeeDrink brewEspresso() {
    Configuration config = configMap.get(CoffeeSelection.ESPRESSO);

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

```
    @Override
    public CoffeeDrink brewFilterCoffee() throws CoffeeException {
       throw new CoffeeException("This machine only brew espresso.");
    }

}
```


Simple Example: Coffee Brewing -Polluting the Interface-

Let's ignore the Interface Segregation Principle for now and perform the following three changes:

1. The *EspressoMachine* class implements the *CoffeeMachine* interface and its *brewFilterCoffee* method.

```
public CoffeeDrink brewFilterCoffee() throws CoffeeException {
    throw new CoffeeException("This machine only brews espresso.");
}
```


Simple Example: Coffee Brewing -Polluting the Interface-

2. We add the *brewEspresso* method to the *CoffeeMachine* interface so that the interface allows you to brew an espresso.

```
public interface CoffeeMachine {

    CoffeeDrink brewFilterCoffee() throws CoffeeException;
    void addGroundCoffee(GroundCoffee newCoffee) throws CoffeeException;
    CoffeeDrink brewEspresso() throws CoffeeException;
}
```

Simple Example: Coffee Brewing -Polluting the Interface-

3. You need to implement the *brewEspresso* method on the *BasicCoffeeMachine* class because it's defined by the *CoffeeMachine* interface. You can also provide the same implementation as a default method on the *CoffeeMachine* interface.

```
    @Override
    public CoffeeDrink brewEspresso() throws CoffeeException {
        throw new CoffeeException("This machine only brews filter coffee.")
;
    }
```


Simple Example: Coffee Brewing -Polluting the Interface-

The 2nd and 3rd change should show you that the *CoffeeMachine* interface is not a good fit for these two coffee machines.

The *brewEspresso* method of the *BasicCoffeeMachine* class and the *brewFilterCoffee* method of the *EspressoMachine* class throw a *CoffeeException* because these operations are not supported by these kinds of machines.

You only had to implement them because they are required by the *CoffeeMachine* interface.

Simple Example: Coffee Brewing -Polluting the Interface-

The problem is that the *CoffeeMachine* interface will change if the signature of the *brewFilterCoffee* method of the *BasicCoffeeMachine* method changes.

That will also require a change in the *EspressoMachine* class and all other classes that use the *EspressoMachine*, even so, the *brewFilterCoffee* method doesn't provide any functionality and they don't call it.

Follow the Interface Segregation Principle

So, how can you fix the *CoffeeMachine* interface and its implementations *BasicCoffeeMachine* and *EspressoMachine*?

You need to split the *CoffeeMachine* interface into multiple interfaces for the different kinds of coffee machines.

All known implementations of the interface implement the *addGroundCoffee* method. So, there is no reason to remove it.

```
public interface CoffeeMachine {

    void addGroundCoffee(GroundCoffee newCoffee) throws CoffeeException;
}
```

Follow the Interface Segregation Principle

That's not the case for the *brewFilterCoffee* and *brewEspresso* methods. You should create two new interfaces to segregate them from each other.

And in this example, these two interfaces should also extend the *CoffeeMachine* interface. But that doesn't have to be the case if you refactor your own application.

Please check carefully if an interface hierarchy is the right approach, or if you should define a set of interfaces.

Follow the Interface Segregation Principle

The *FilterCoffeeMachine* interface extends the *CoffeeMachine* interface, and defines the *brewFilterCoffee* method.

```
public interface FilterCoffeeMachine extends CoffeeMachine {

    CoffeeDrink brewFilterCoffee() throws CoffeeException;
}
```

And the *EspressoCoffeeMachine* interface also extends the *CoffeeMachine* interface, and defines the *brewEspresso* method.

```
public interface EspressoCoffeeMachine extends CoffeeMachine {

    CoffeeDrink brewEspresso() throws CoffeeException;
}
```


Follow the Interface Segregation Principle

As a result, the *BasicCoffeeMachine* and the *EspressoMachine* class no longer need to provide empty method implementations and are independent of each other.

Follow the Interface Segregation Principle

The *BasicCoffeeMachine* class now implements the *FilterCoffeeMachine* interface, which only defines the *addGroundCoffee* and the *brewFilterCoffee* methods.

```
public class BasicCoffeeMachine implements FilterCoffeeMachine {
```

```
    @Override
    public CoffeeDrink brewFilterCoffee() {
        Configuration config = configMap.get(CoffeeSelection.FILTER_COFFEE)
;

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

Follow the Interface Segregation Principle

And the *EspressoMachine* class implements the *EspressoCoffeeMachine* interface with its methods *addGroundCoffee* and *brewEspresso*.

```
public class EspressoMachine implements EspressoCoffeeMachine {

    @Override
    public CoffeeDrink brewEspresso() throws CoffeeException {
        Configuration config = configMap.get(CoffeeSelection.ESPRESSO);

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

https://stackify.com/interface-segregation-principle/

Additional Exercise

After you segregated the interfaces so that you can evolve the two coffee machine implementations independently of each other, explore how you can add different kinds of coffee machines to your applications.

https://stackify.com/interface-segregation-principle/

References

- https://stackify.com/solid-design-principles/
- https://stackify.com/dependency-inversion-principle/
- https://deviq.com/solid/