OOP Design Principles

SOLID

COMP3607
Object Oriented Programming II

Week 3

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

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.

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

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.

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

Simple Example: Coffee Brewing

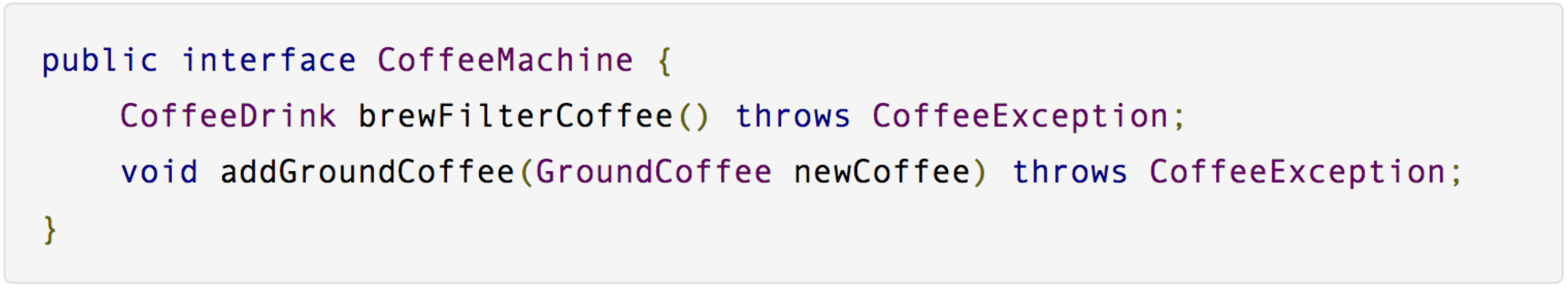
The project used the BasicCoffeeMachine class to model a basic coffee machine which uses ground coffee to brew a delicious filter coffee.

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

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

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;
}
```

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

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 *PremiumCoffeeMachine* class.

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

Simple Example: Coffee Brewing -Polluting the Interface-

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.

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

Simple Example: Coffee Brewing -Polluting the Interface-

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

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

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

}
```

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

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.");
}
```

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

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.

```java
public interface CoffeeMachine {

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

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

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.")
;
    }
```

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

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.

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

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.

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

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;
}
```

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

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.

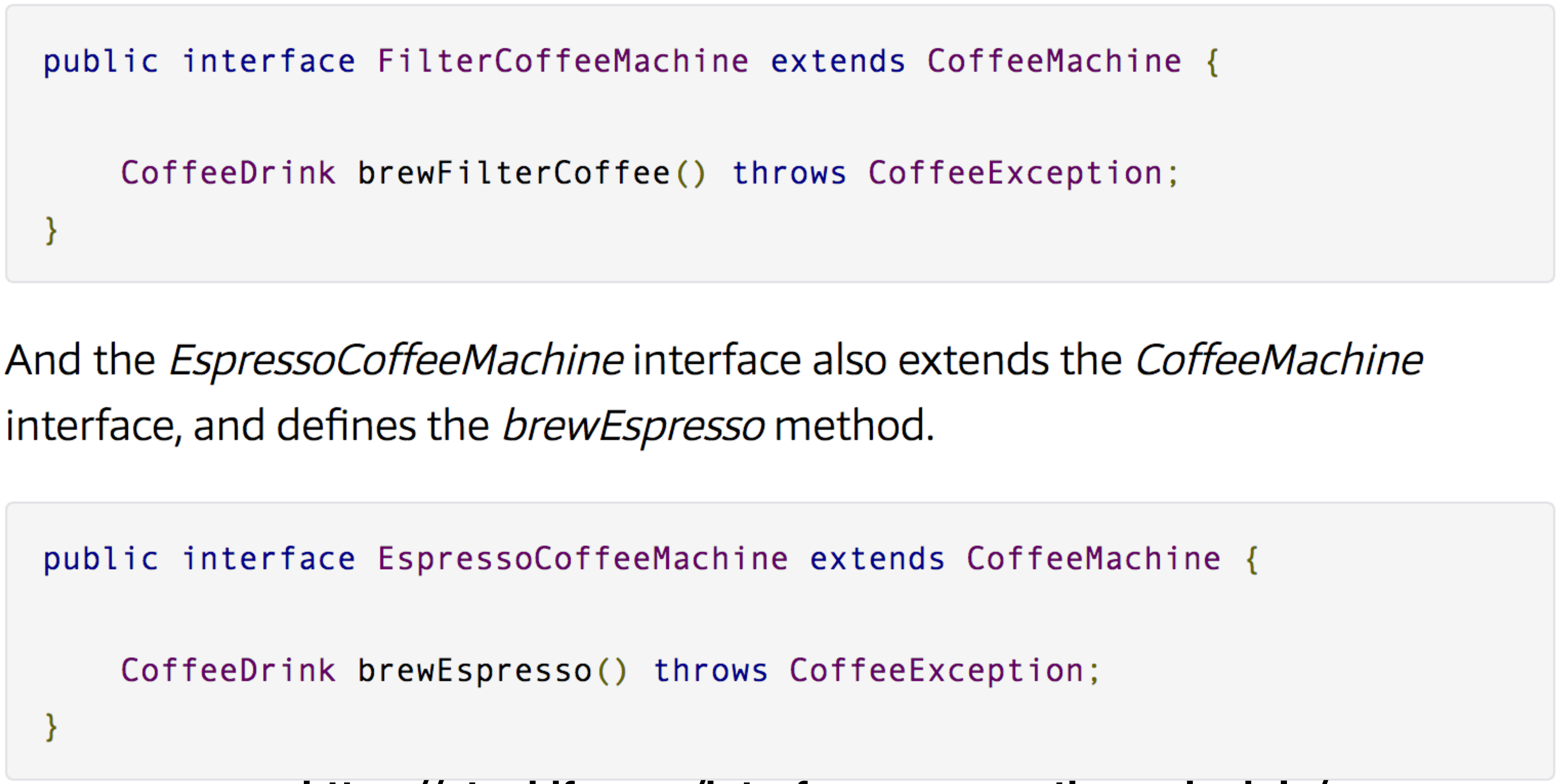
And in this example, these two interfaces should also extend the *CoffeeMachine* interface. But that doesn't have to be the case if you refactor your own application.

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

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

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;
}
```

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

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.

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

Follow the Interface Segregation Principle

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

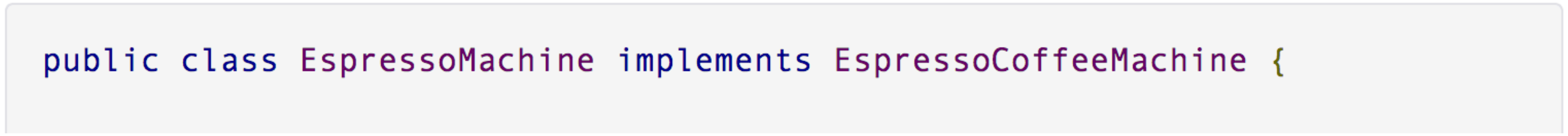
```
public class BasicCoffeeMachine implements FilterCoffeeMachine {
```

```

    @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());
    }
```

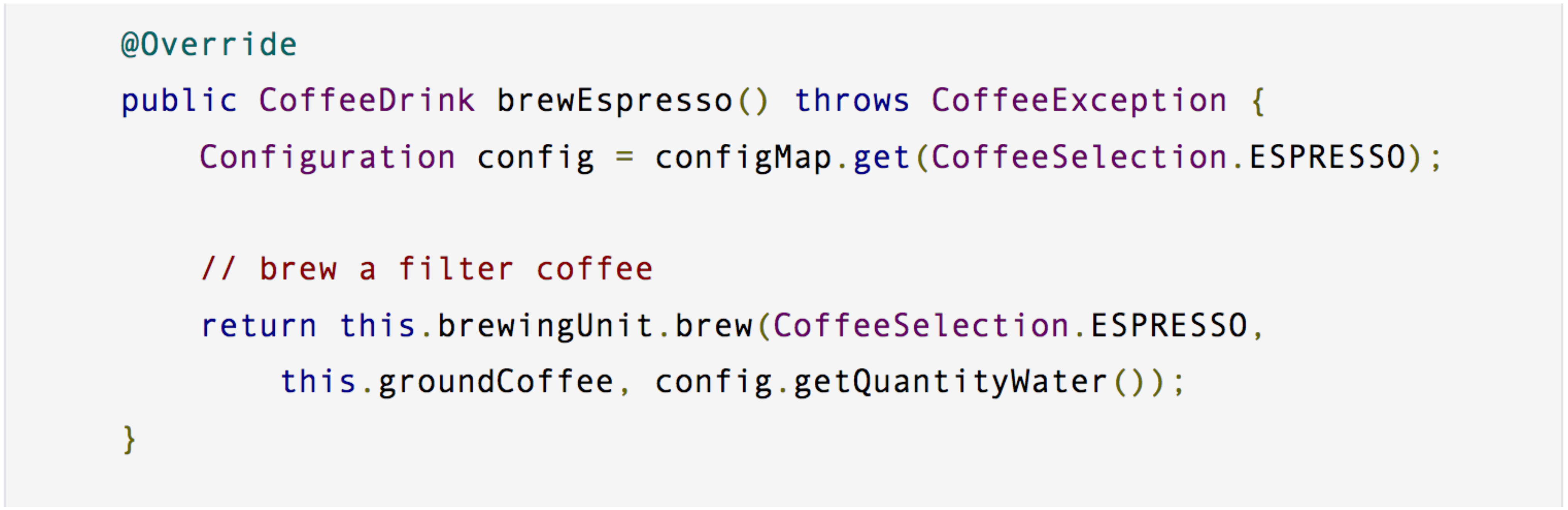
https://stackify.com/interface-segregation-principle/

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/

1 min

Exercise

List the names of the design principles in the SOLID acronym:

- S
- O
- L
- I
- Dependency Inversion

References

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