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

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.

---

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

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.

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

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.

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

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.

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

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.

https://stackify.com/solid-design-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

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

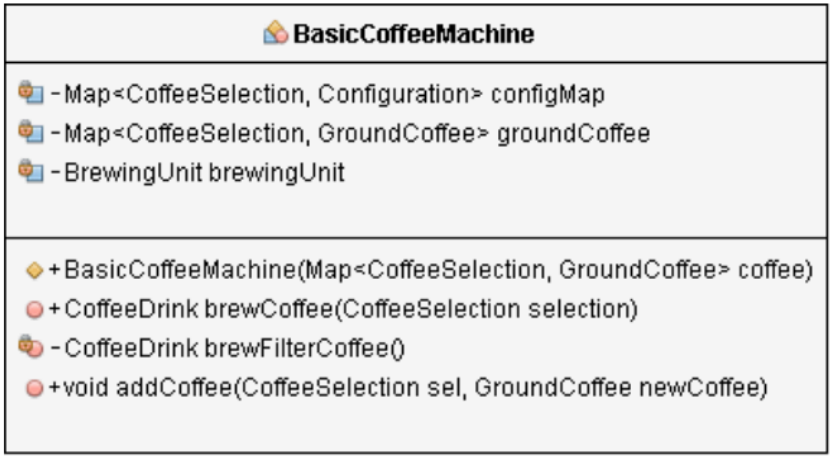
Example: Coffee Brewing

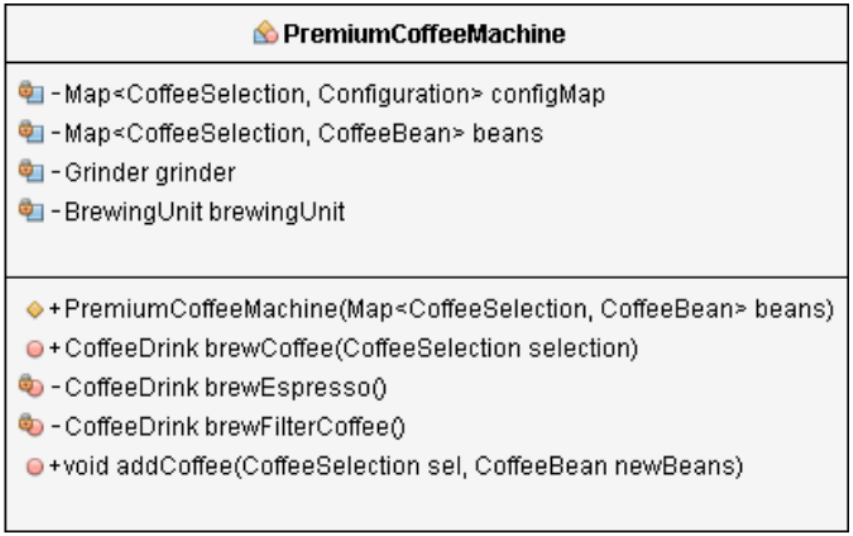
There are relatively basic ones that you can use to transform one or two scoops of ground coffee and a cup of water into a simple cup of filter coffee.

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.

https://coffee-channel.com/different-types-of-coffee-makers/

Example: Coffee Brewing





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

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

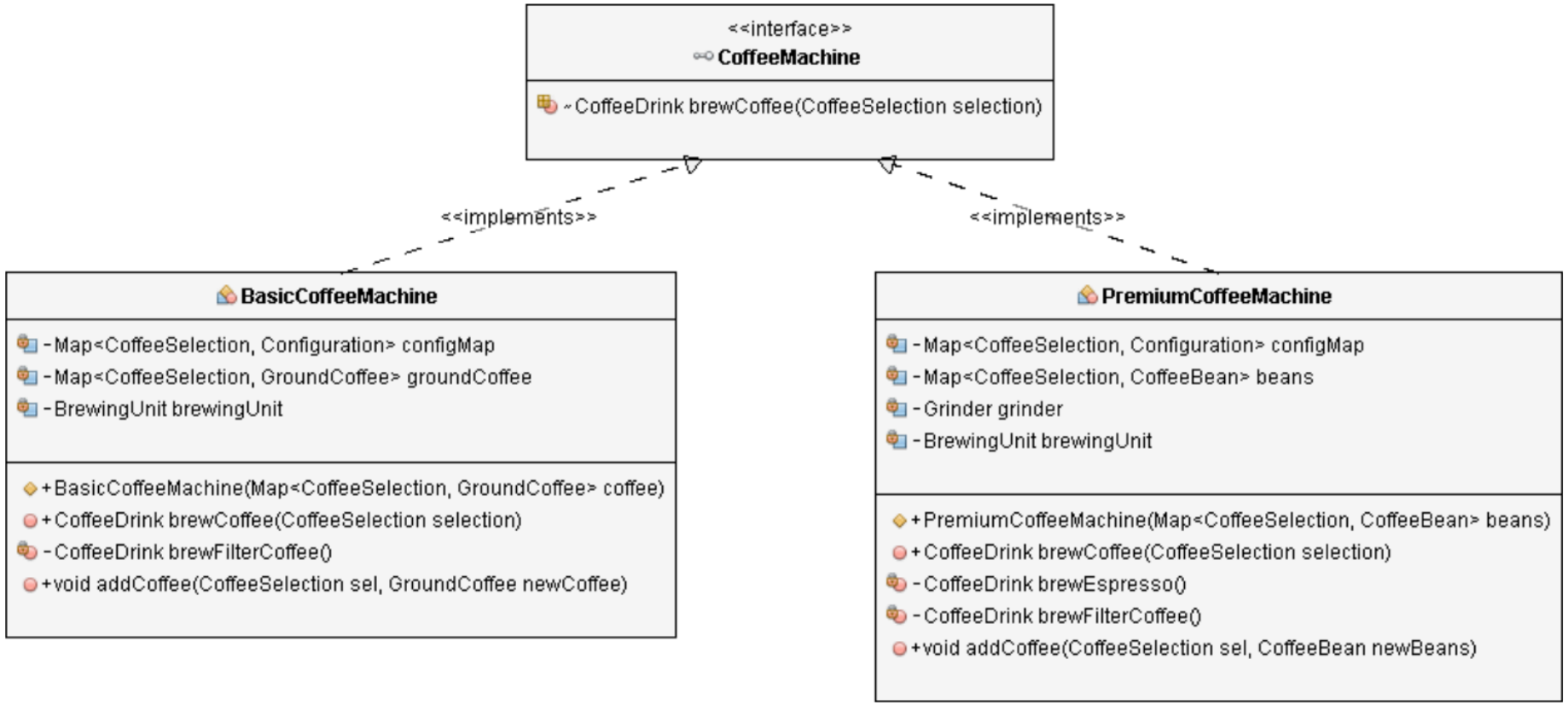
Example: Coffee Brewing
- Introducing a shared interface-

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/