OOP Design Principles

SOLID - Dependency Inversion

COMP3607
Object Oriented Programming II

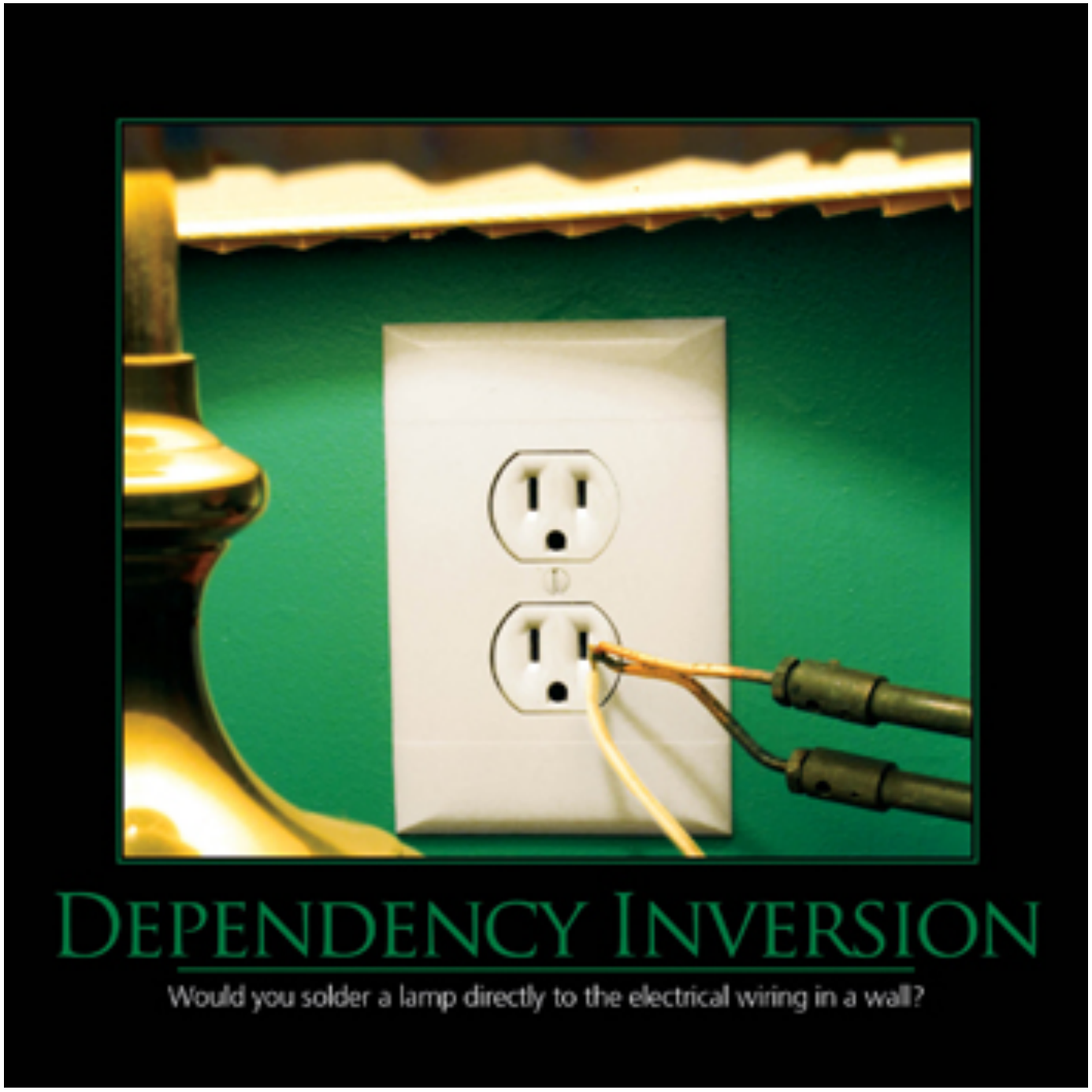
Week 4

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

Dependency Inversion



https://deviq.com/dependency-inversion-principle/

Dependency Inversion Principle

Robert C. Martin's definition of the Dependency Inversion Principle consists of two parts:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

https://stackify.com/dependency-inversion-principle/

High-level vs Low-level Abstractions

Both high-level and low-level modules depend on the abstraction.

High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features.

To achieve that, you need to introduce an **abstraction** that decouples the high-level and low-level modules from each other.

https://stackify.com/dependency-inversion-principle/

Based on other SOLID principles

If you apply the Open/Closed Principle and the Liskov Substitution Principle to your code, it will also follow the Dependency Inversion Principle.

https://stackify.com/dependency-inversion-principle/

Relationship to other principles -Open/Closed Principle-

The Open/Closed Principle required a software component to be open for extension, but closed for modification. You can achieve that by introducing interfaces for which you can provide different implementations. The interface itself is closed for modification, and you can easily extend it by providing a new interface implementation.

https://stackify.com/dependency-inversion-principle/

Relationship to other principles -Liskov Substitution Principle-

Your implementations should follow the Liskov Substitution Principle so that you can replace them with other implementations of the same interface without breaking your application.

https://stackify.com/dependency-inversion-principle/

Coffee Example

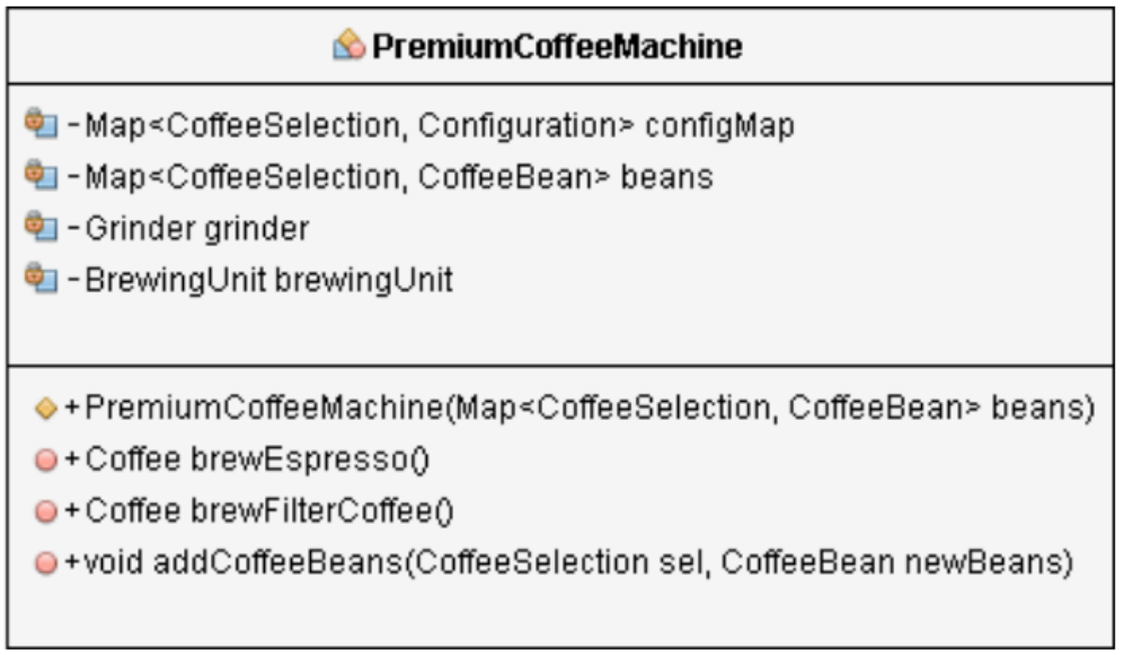
Consider that there are many different types of coffee machines that brew coffee:

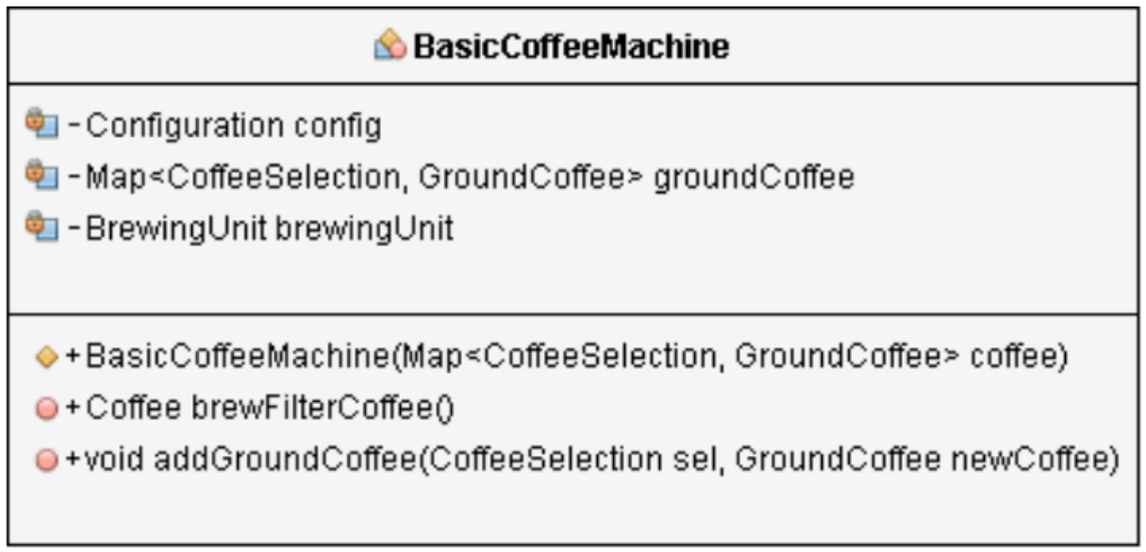
- Simple ones: use water and ground coffee to brew filter coffee
- Premium ones:
  - include a grinder to freshly grind the required amount of coffee beans
  - brew different kinds of coffee.

https://stackify.com/dependency-inversion-principle/

Coffee Example

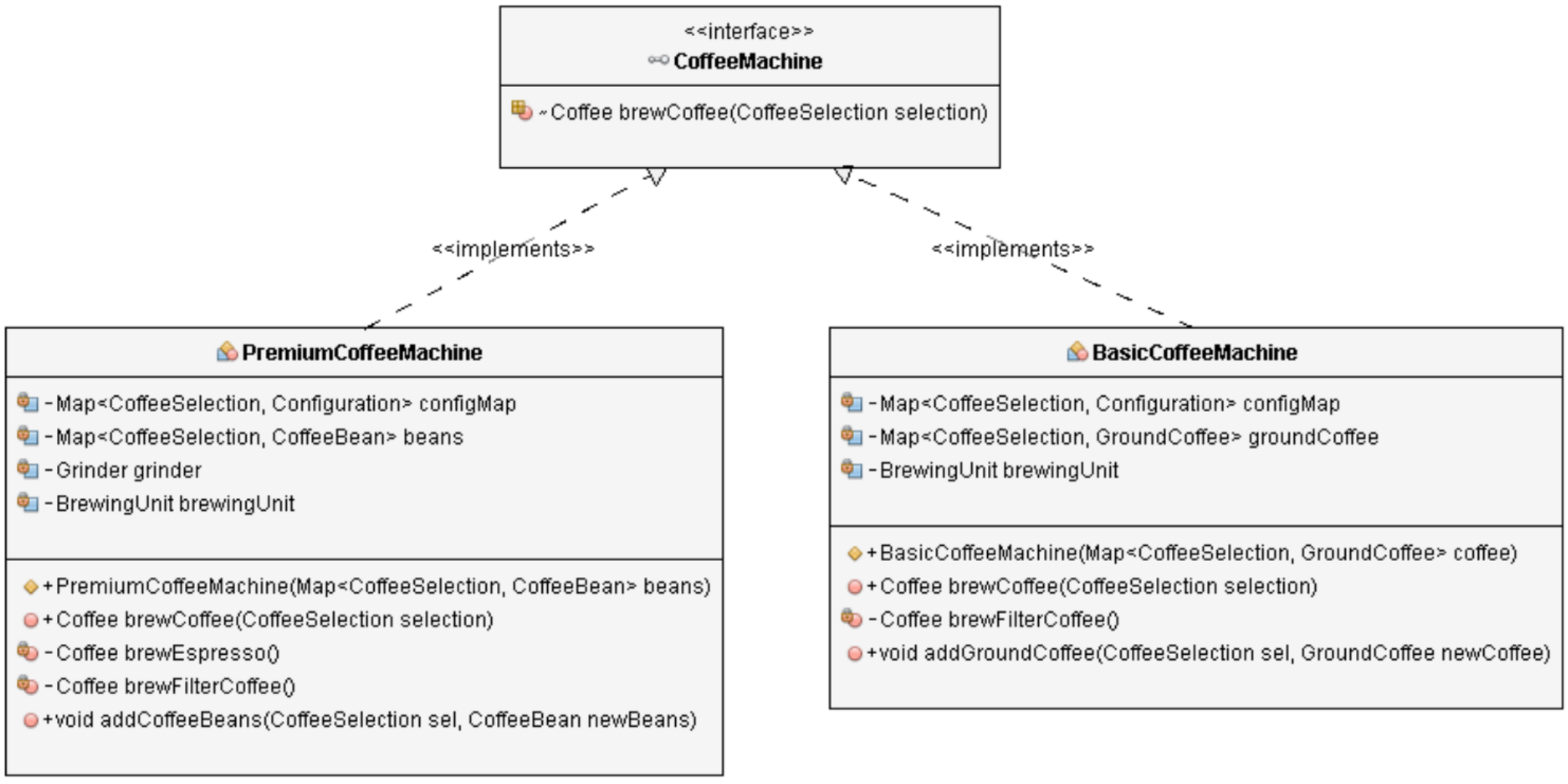
If you build a coffee machine application that automatically brews you a fresh cup of coffee in the morning, you can model these machines as a *BasicCoffeeMachine* and a *PremiumCoffeeMachine* class.





https://stackify.com/dependency-inversion-principle/

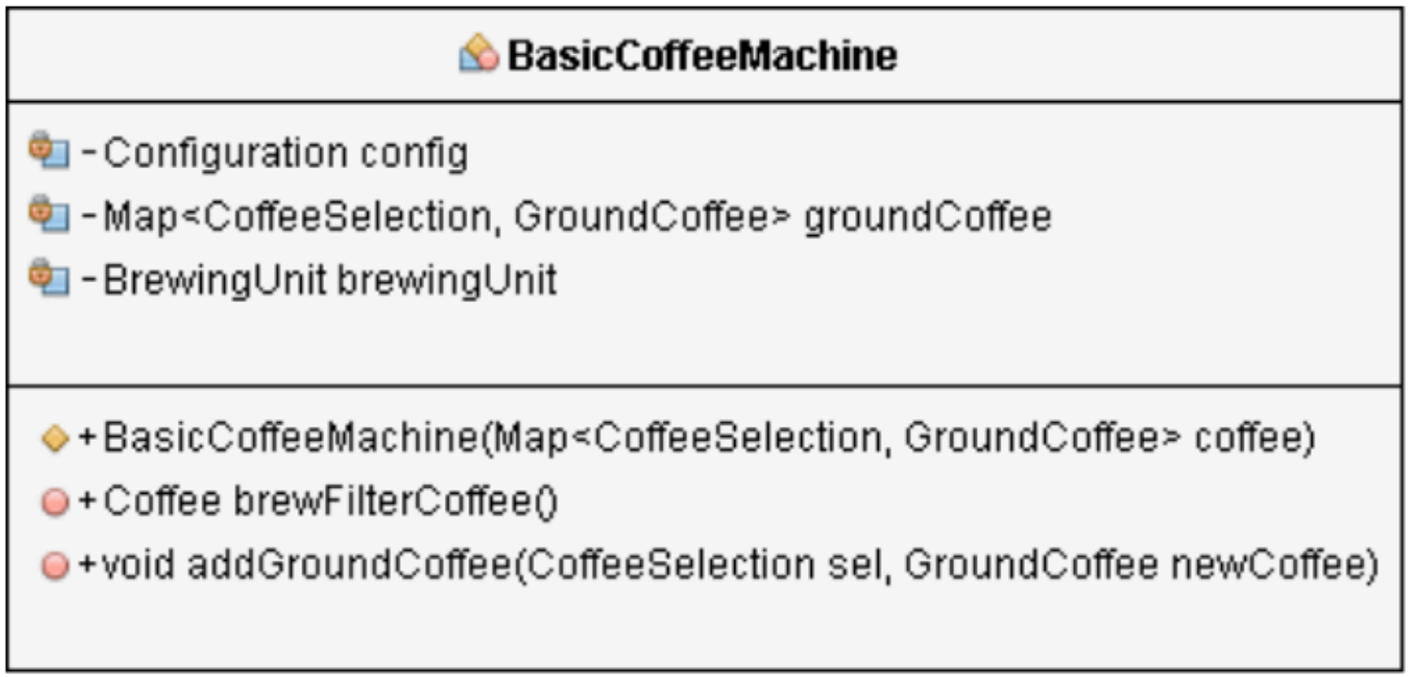
Coffee Example:
Applying the Open/Closed Principle



https://stackify.com/dependency-inversion-principle/

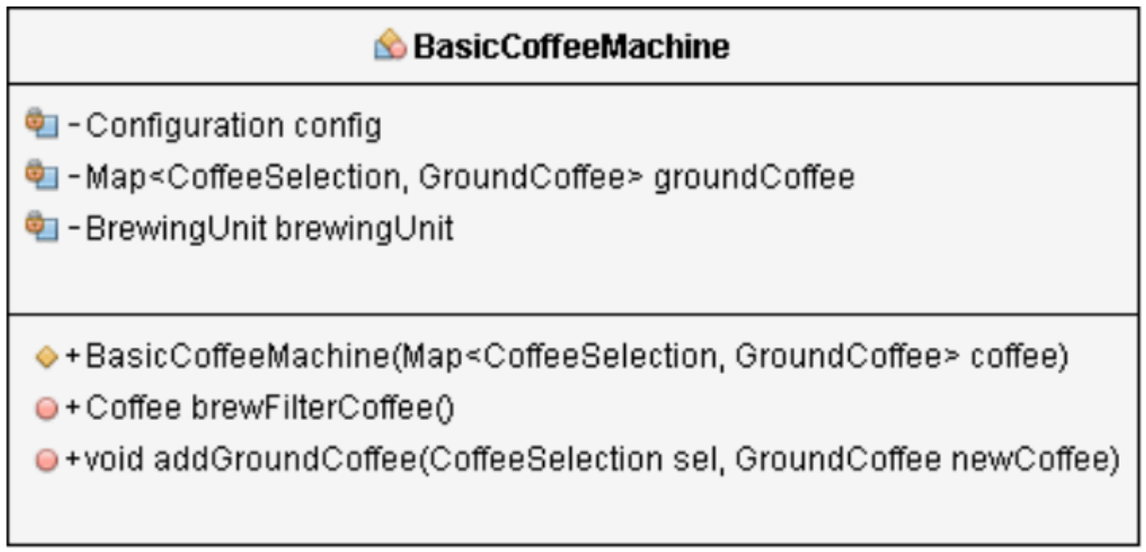
Implementing the *BasicCoffeeMachine*

The *BasicCoffeeMachine* class is quite simple. It only implements a constructor and two public methods. You can call the *addGroundCoffee* method to refill ground coffee, and the *brewFilterCoffee* method to brew a cup of filter coffee.



https://stackify.com/dependency-inversion-principle/

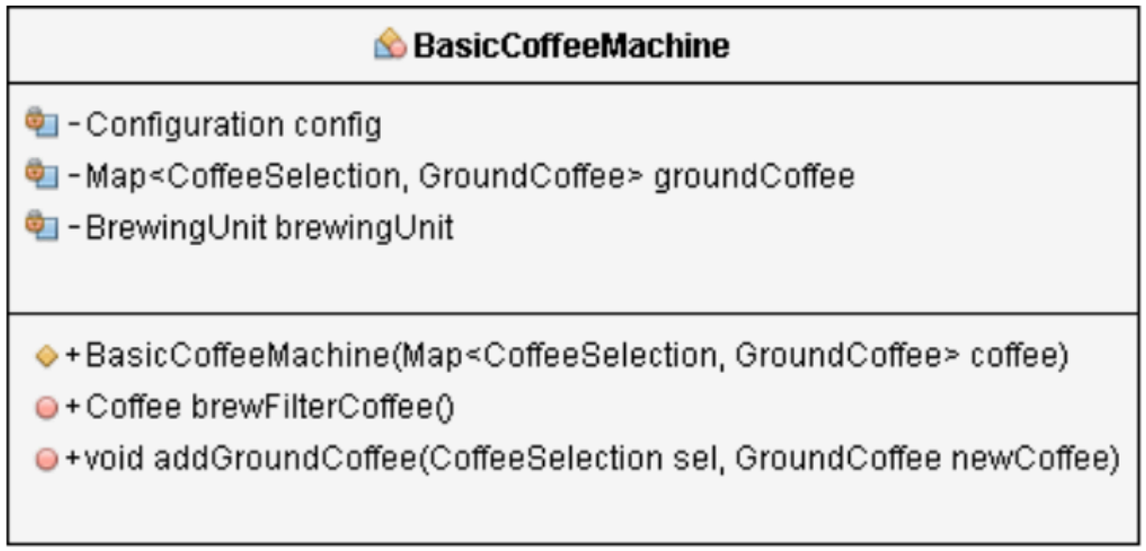
Implementing the *BasicCoffeeMachine*




```java
public class BasicCoffeeMachine implements CoffeeMachine {

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

    public BasicCoffeeMachine(Map<CoffeeSelection, GroundCoffee> coffee).
        this.groundCoffee = coffee;
        this.brewingUnit = new BrewingUnit();
        this.config = new Configuration(30, 480);
    }
```

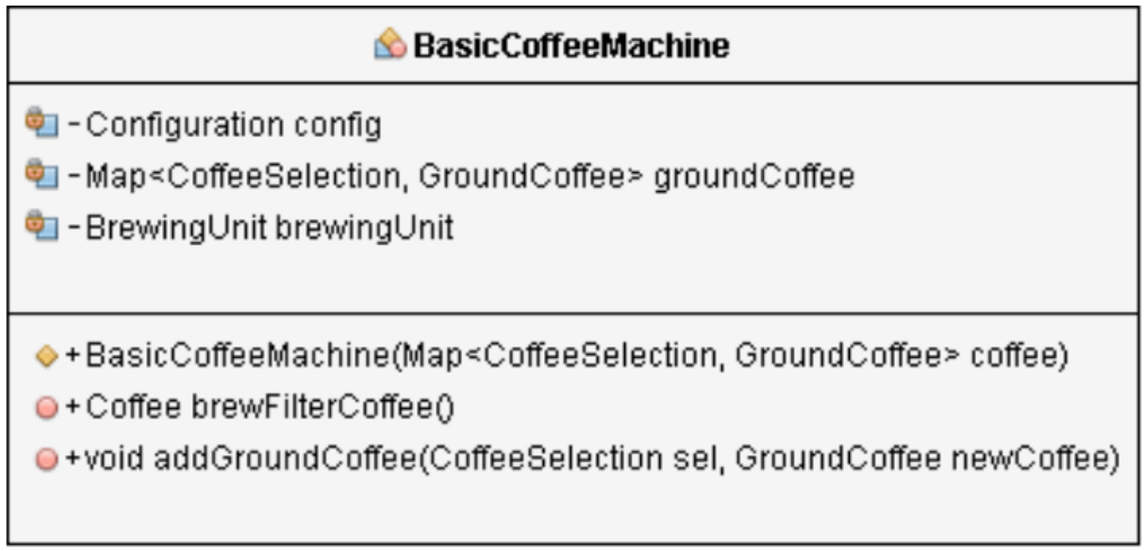
https://stackify.com/dependency-inversion-principle/

Implementing the *BasicCoffeeMachine*




```
    @Override
    public Coffee brewFilterCoffee() {
        // get the coffee
        GroundCoffee groundCoffee = this.groundCoffee.get(CoffeeSelection.F
ILTER_COFFEE);
        // brew a filter coffee
       return this.brewingUnit.brew(CoffeeSelection.FILTER_COFFEE, groundCo
ffee, this.config.getQuantityWater());
    }
```

Implementing the *BasicCoffeeMachine*

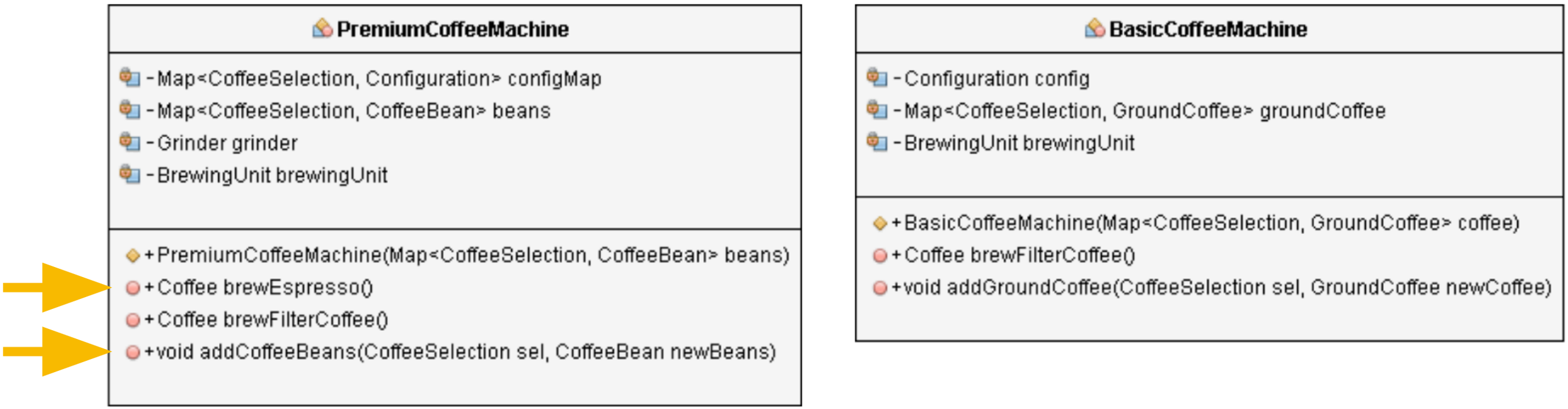



```
    public void addGroundCoffee(CoffeeSelection sel, GroundCoffee newCoffee
) throws CoffeeException {
        GroundCoffee existingCoffee = this.groundCoffee.get(sel);
        if (existingCoffee != null) {
            if (existingCoffee.getName().equals(newCoffee.getName())) {
                existingCoffee.setQuantity(existingCoffee.getQuantity()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

Implementing the *PremiumCoffeeMachine*

The implementation of the *PremiumCoffeeMachine* class looks very similar. The main differences are:

- It implements the *addCoffeeBeans* method instead of the *addGroundCoffee* method.
- It implements the additional *brewEspresso* method.



Implementing the *PremiumCoffeeMachine*

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

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, 2
8));
    }
```

Implementing the *PremiumCoffeeMachine*

The *brewFilterCoffee* method is identical to the one provided by the *BasicCoffeeMachine*.

```
    public 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, groundC
offee,
            config.getQuantityWater());
    }
```

Implementing the *PremiumCoffeeMachine*

The *addCoffeeBeans* method is a new method provided by the *PremiumCoffeeMachine*. It sets the type of *CoffeeBean* used.

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

Implementing the *PremiumCoffeeMachine*

The *brewExpresso* method is a new method provided by the *PremiumCoffeeMachine*. It uses the *Grinder* object.

```
public 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());
}
```

Coffee Example with the Dependency Inversion Principle

To implement a class that follows the Dependency Inversion Principle and can use the **BasicCoffeeMachine** or the **PremiumCoffeeMachine** class to brew a cup of coffee, you need to apply the Open/Closed and the Liskov Substitution Principle.

That requires a small refactoring during which you introduce **interface abstractions** for both classes.

Introducing abstractions

If you use a **BasicCoffeeMachine**, you can only brew filter coffee, but with a **PremiumCoffeeMachine**, you can brew filter coffee or espresso.

So, which **interface abstraction** would be a good fit for both classes?

Introducing abstractions

- The **FilterCoffeeMachine** interface defines the Coffee **brewFilterCoffee( )** method and gets implemented by all coffee machine classes that can brew a filter coffee.
- All classes that you can use to brew an espresso, implement the **EspressoMachine** interface, which defines the Coffee **brewEspresso( )** method.

Introducing abstractions

The definition of both interfaces is pretty simple.

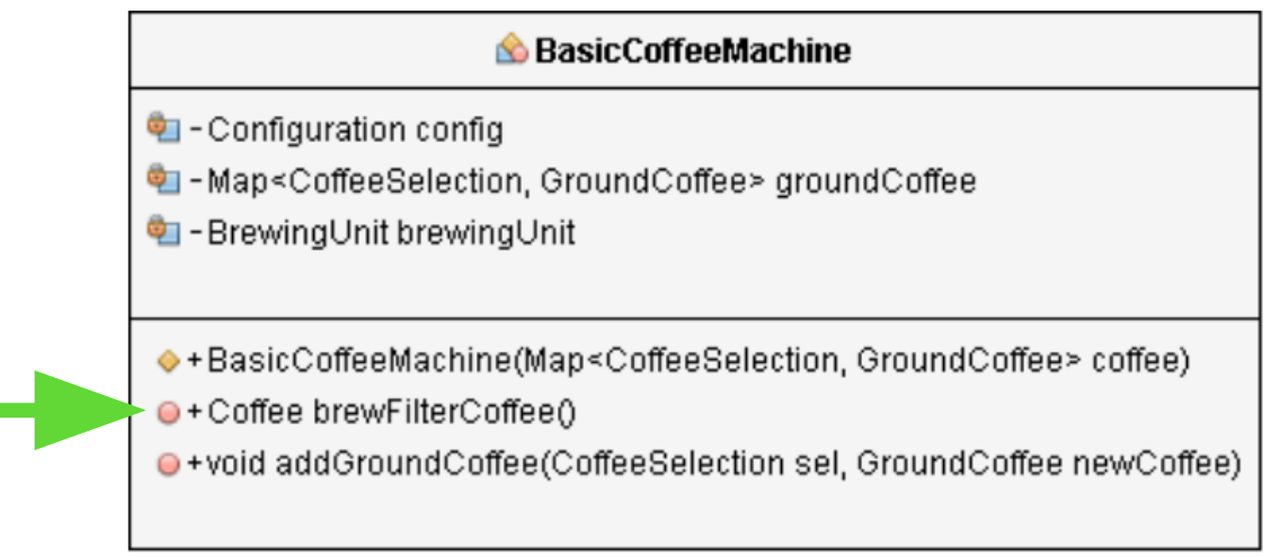
```
public interface CoffeeMachine {
    Coffee brewFilterCoffee();
}

public interface EspressoMachine {
    Coffee brewEspresso();
}
```

Both coffee machine classes now need to be refactored so that they implement one or both of these interfaces

Refactoring the BasicCoffeeMachine class

You can use the *BasicCoffeeMachine* to brew a filter coffee, so it should implement the *CoffeeMachine* interface.

The class already implements the *brewFilterCoffee()* method. You only need to add ***implements CoffeeMachine*** to the class definition.



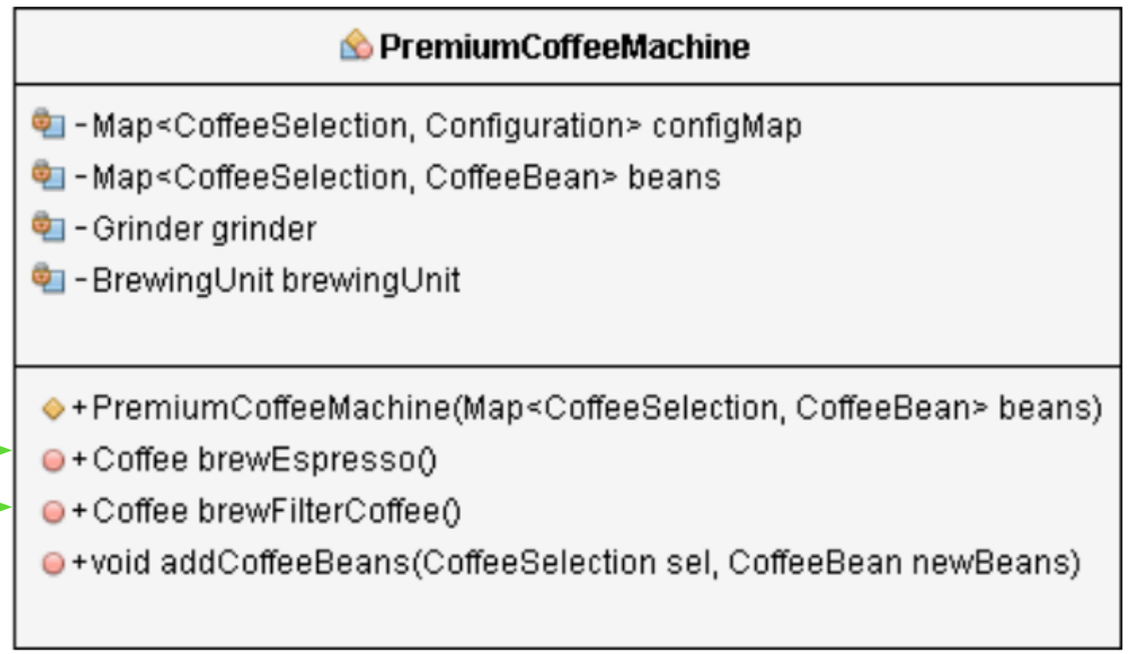

```java
public interface CoffeeMachine {
    Coffee brewFilterCoffee();
}

public interface EspressoMachine {
    Coffee brewEspresso();
}
```

```java
public class BasicCoffeeMachine implements CoffeeMachine {
```

Refactoring the PremiumCoffeeMachine class

You can use the premium coffee machine to brew filter coffee and espresso, so the *PremiumCoffeeMachine* class should implement the *CoffeeMachine* and the *EspressoMachine* interfaces.

The class already implements the methods defined by both interfaces. You just need to declare that it implements both interfaces.



✓
```
public interface CoffeeMachine {
    Coffee brewFilterCoffee();
}
```

✓
```
public interface EspressoMachine {
    Coffee brewEspresso();
}
```

```
public class PremiumCoffeeMachine implements CoffeeMachine, EspressoMachine
{
```

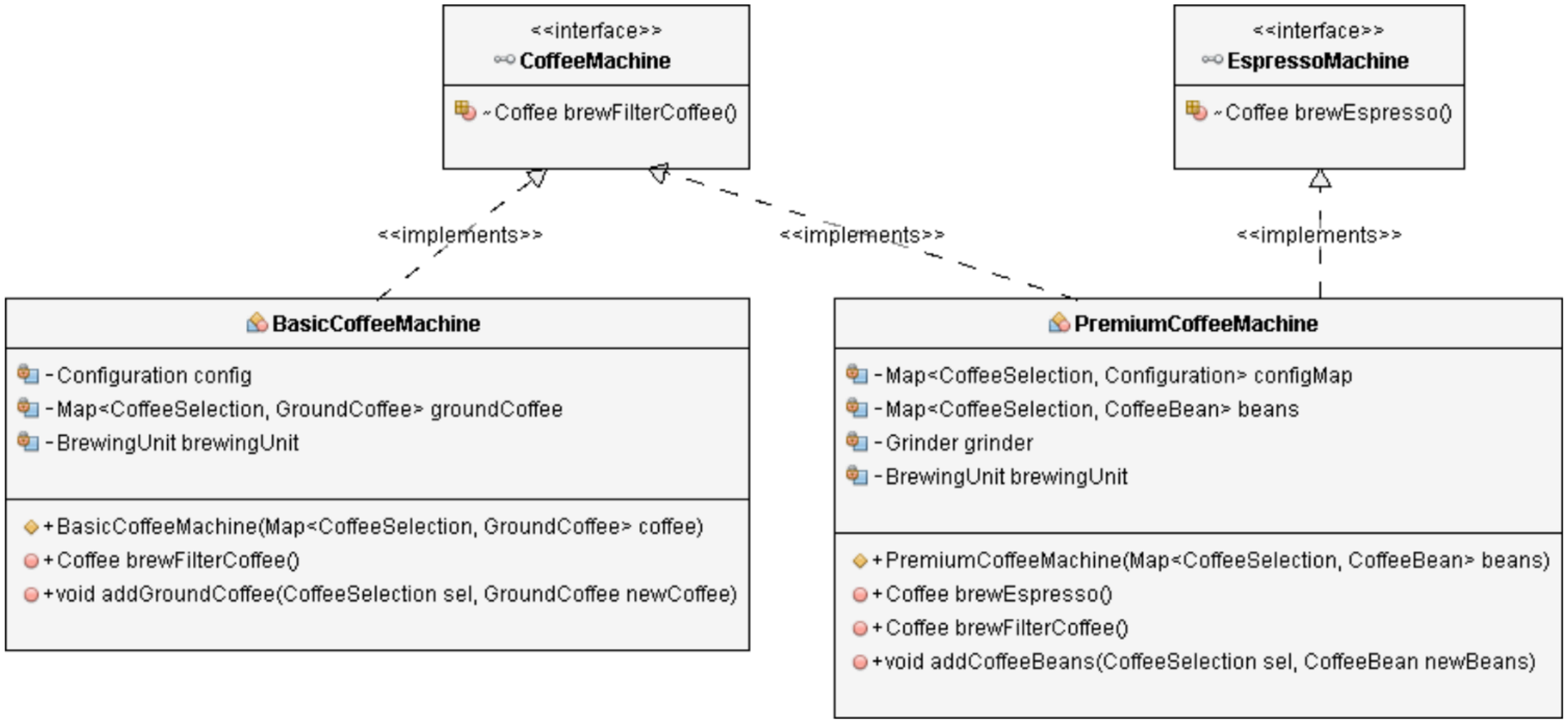
Open/Closed and Liskov Substitution Principles

The *BasicCoffeeMachine* and the *PremiumCoffeeMachine* classes now follow the Open/Closed and the Liskov Substitution principles.

The interfaces enable you to add new functionality without changing any existing code by adding new interface implementations.

By splitting the interfaces into *CoffeeMachine* and *EspressoMachine*, you separate the two kinds of coffee machines and ensure that all *CoffeeMachine* and *EspressMachine* implementations are interchangeable.

Open/Closed and Liskov Substitution Principles



Implementing the coffee machine application

```
public class CoffeeApp {
    private CoffeeMachine coffeeMachine;

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

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

Due to the abstraction of the *CoffeeMachine* interface and its provided functionality, the implementation of the *CoffeeApp* is very simple. It requires a *CoffeeMachine* object as a constructor parameter and uses it in the *prepareCoffee* method to brew a cup of filter coffee.

Implementing the coffee machine application

The only code that directly depends on one of the implementation classes is the *CoffeeAppStarter* class, which instantiates a *CoffeeApp* object and provides an implementation of the *CoffeeMachine* interface.

Implementing the coffee machine application

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

public class CoffeeAppStarter {
    public static void main(String[] args) {
        // create a Map of available coffee beans
        Map<CoffeeSelection, CoffeeBean> beans = new HashMap<CoffeeSelectio
n, 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();
```

Open/Closed, Liskov Substitution and Dependency Inversion Principles

You can now create additional, higher-level classes that use one or both of these interfaces to manage coffee machines without directly depending on any specific coffee machine implementation.

Your classes therefore also comply with the Dependency Inversion Principle ie. you can change higher-level and lower-level components without affecting any other classes, as long as you don’t change any interface abstractions.

References

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