DECORATOR DESSIGN PATTERN

WELCOME TO OUR GROUP

Hao Trinh The

10421017

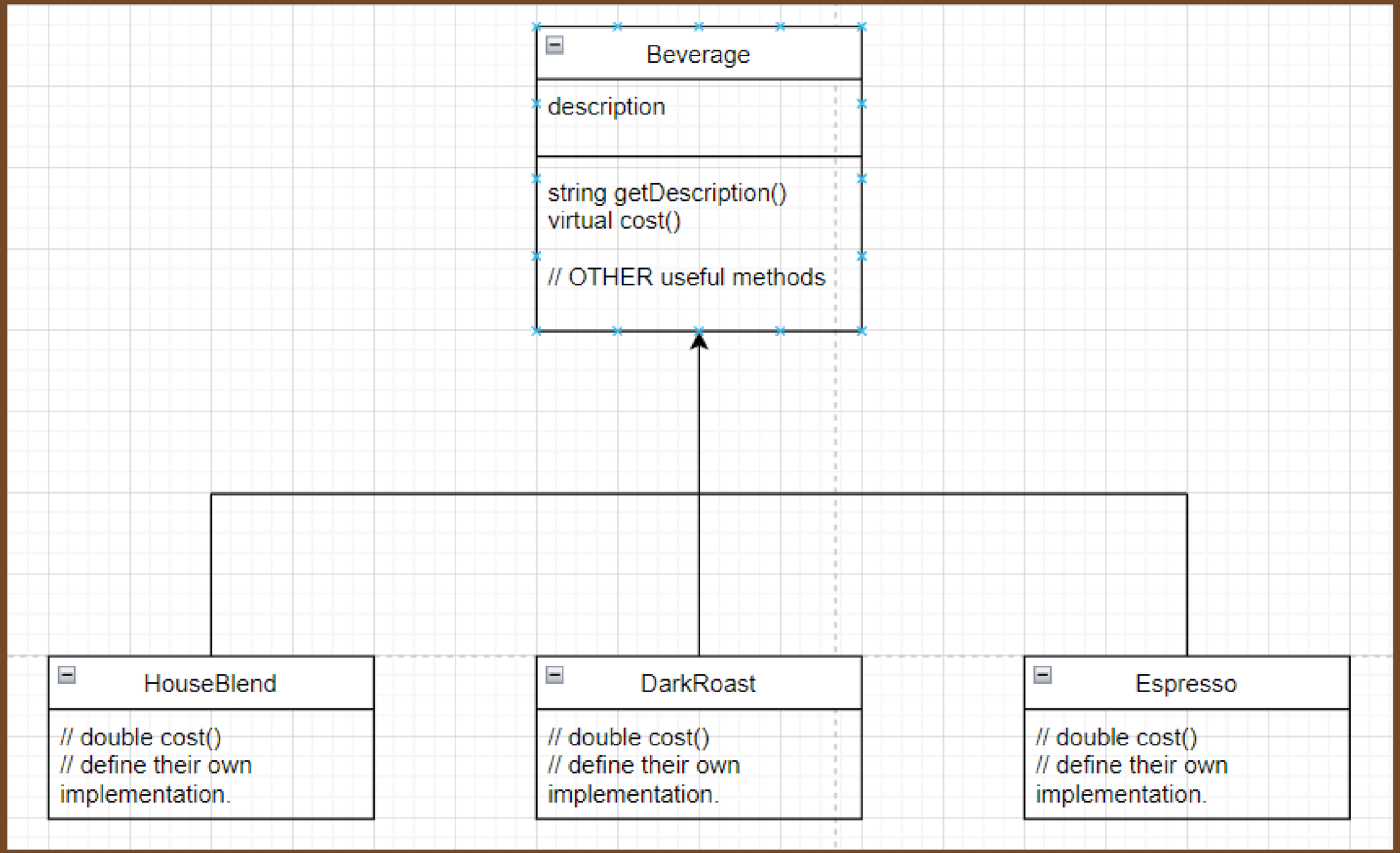
Huy Vo Vuong Bao

10421021

TABLE OF CONTENT

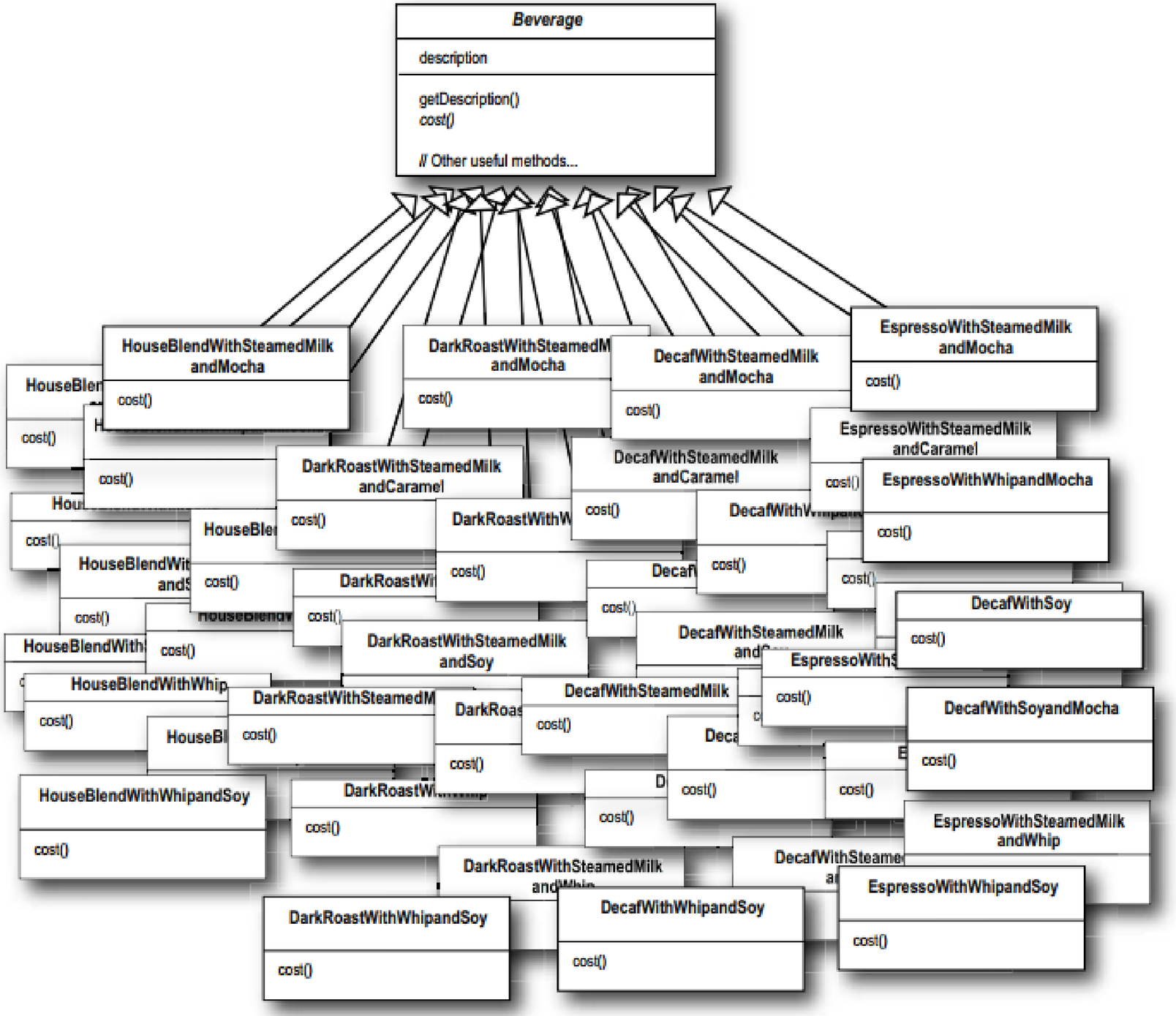
STAR BUZZ COFFEE

The fastest growing coffee shop in the neighborhood



Beverage Class Design

ARISED PROBLEMS



COFFEE + CONDIMENTS WHAT WILL HAPPEN?

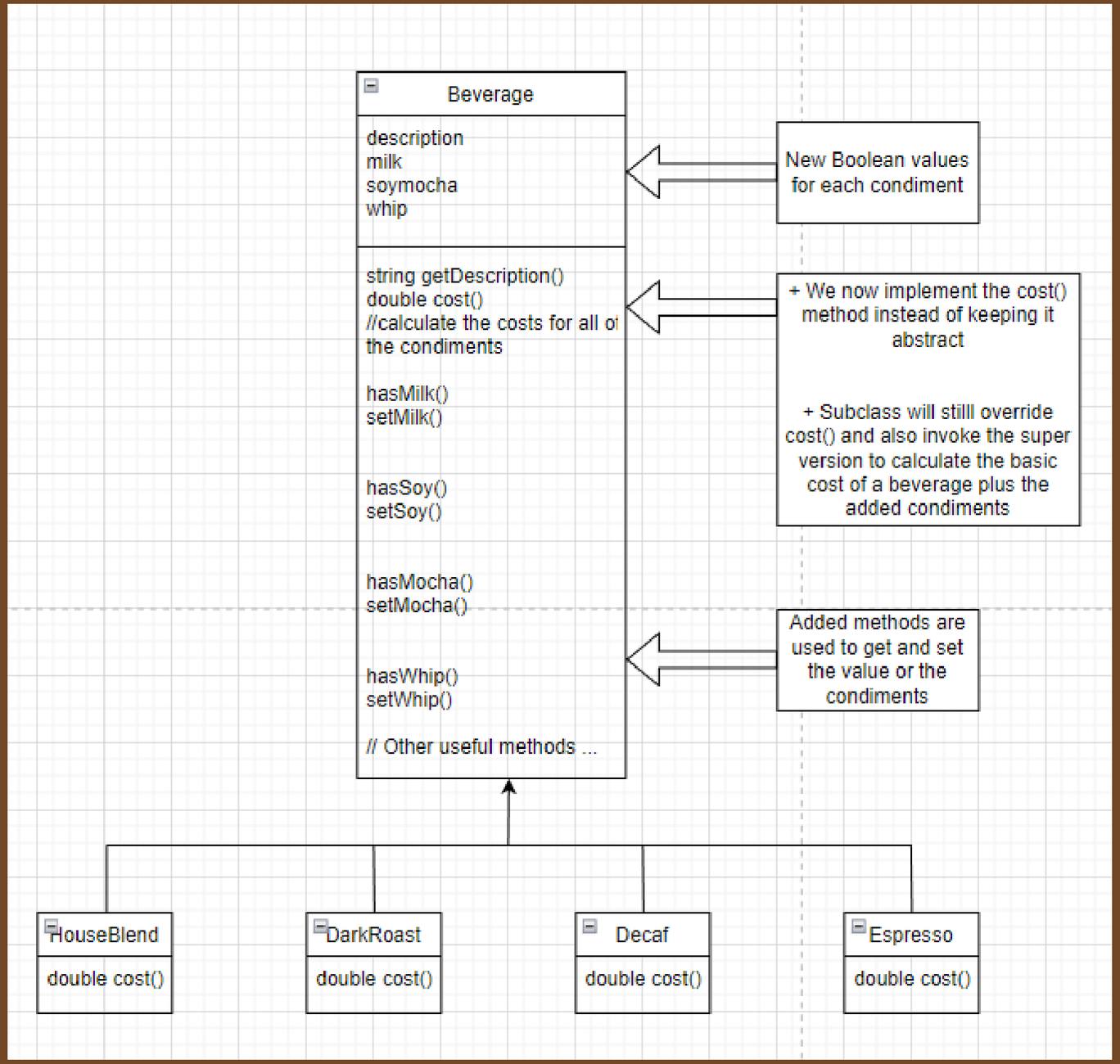
Class Explosion

- What will happen if the price of milk goes up?
- What will happen if they add a new caramel topping?
- **=> MAINTENANCE NIGHTMARE**

Hold a second!!!

- First, why do we need all that classes?
- We can use instance variables and inheritance in the superclass to keep track of the condiments instead.
- Let me show you how it look likes:)))

5 classes in total



REMAINING PROBLEMS

Price changes

will force us to change existing code

New condiments

will force us to add new methods and alter the cost method in the superclass

New beverages

old condiments are not appropriate

Special orders

what if a customer wants a double mocha espresso?

=> Maintenance nightmare again:))

DECORATOR DESIGN PATTERN

THE OPEN-CLOSED DESIGN PRINCIPLE

Classes should be opened for extension, but closed for modification.

MAIN GOALS

- Allow classess to be easily extended to incorporate new behavior without modifying existing code.

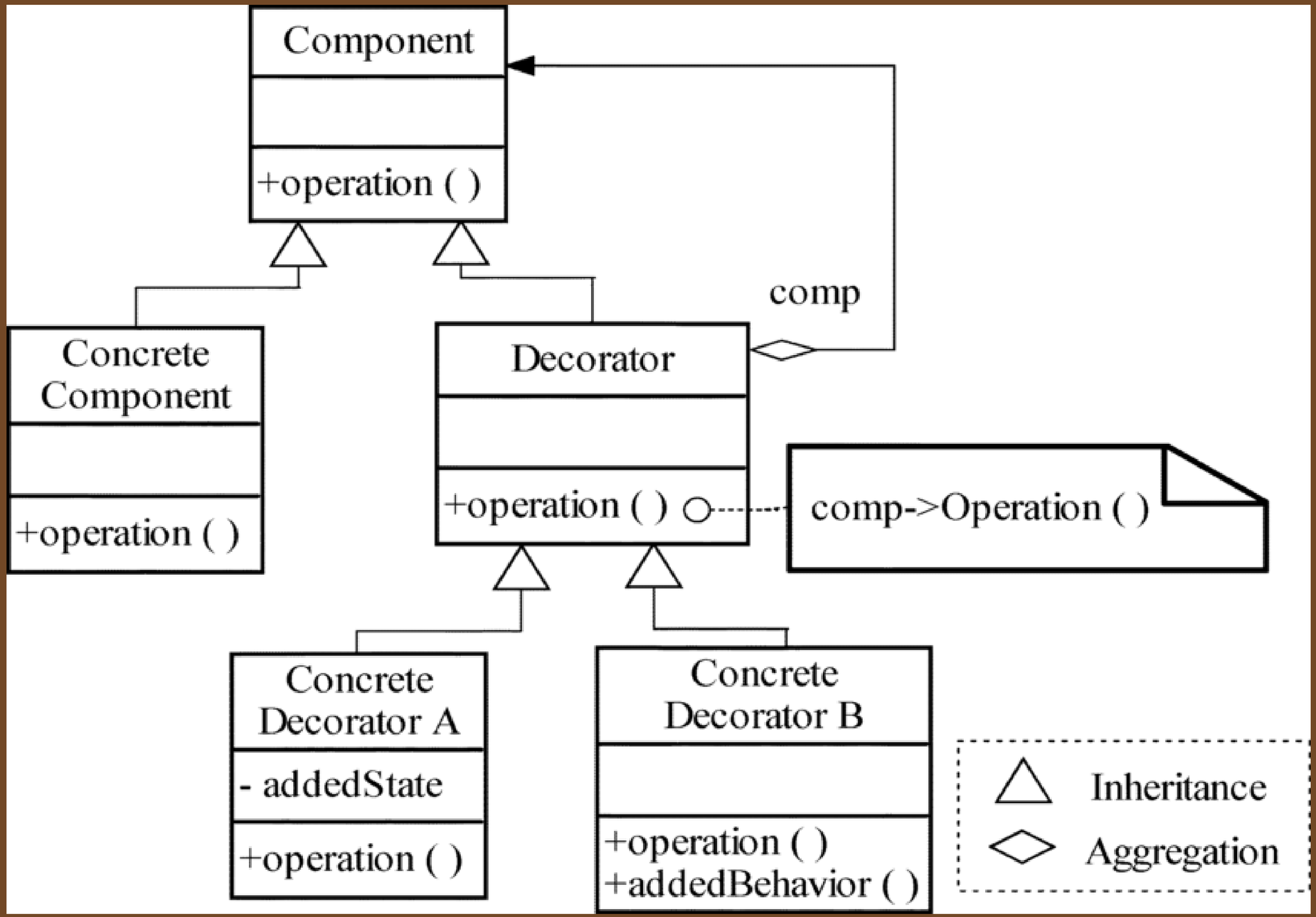
- Make designs resilient to change and flexible enough to take on new functionality to meet changing requirements.

KEEP CALM AND MEET

Decorator Pattern

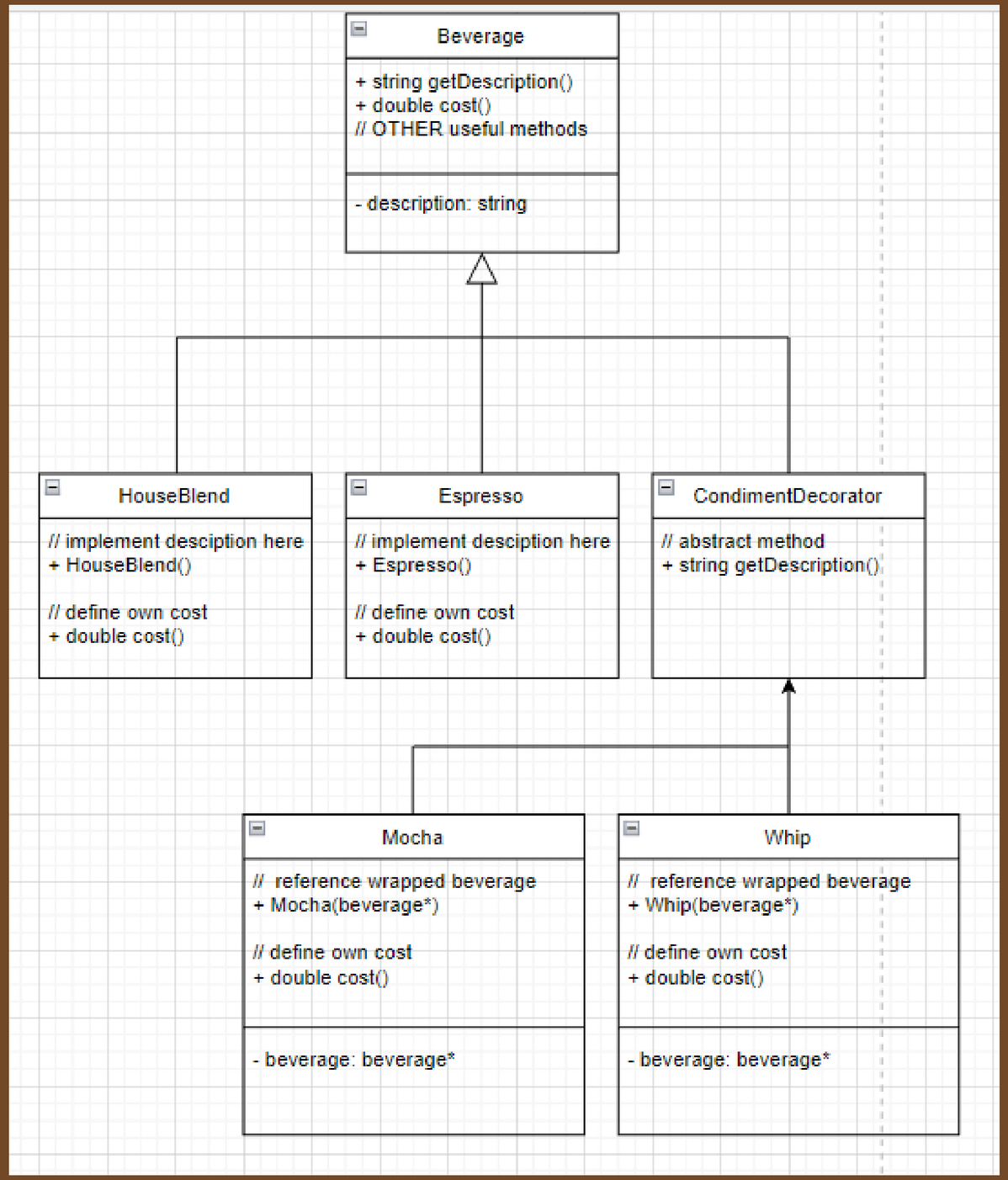
Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

General UML for Decorator Design Pattern



CODE IN ACTION

Use Decorator Design Pattern to redesign the UML then implement it.



UML redesign

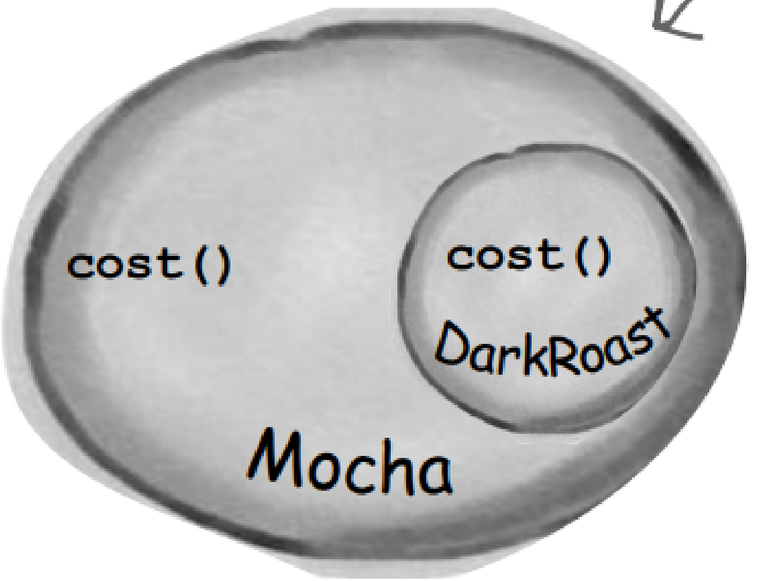
EXAMPLE:

1. Take the DarkRoast object
2. Decorate it with Mocha object
3. Decorate it with Whip object
4. Call the cost() method and rely on delegation to add on the condiment costs

1 **We start with our DarkRoast object.**

Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

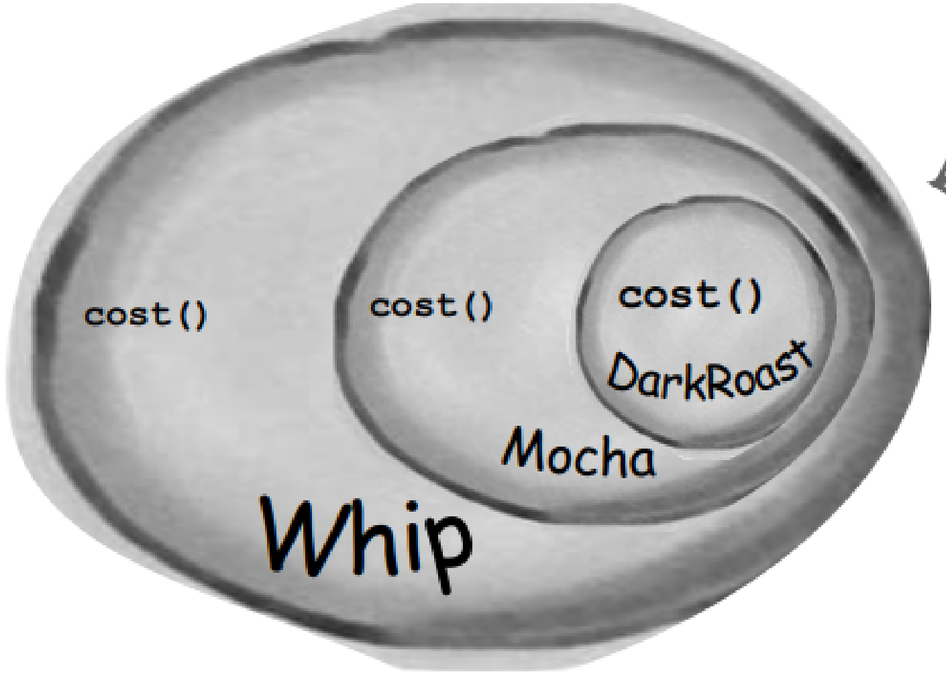
❷ **The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.**



The Mocha object is a decorator. Its type mirrors the object it is decorating, in this case, a Beverage. (By "mirror", we mean it is the same type..)

So, Mocha has a cost() method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

❸ The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.



Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.

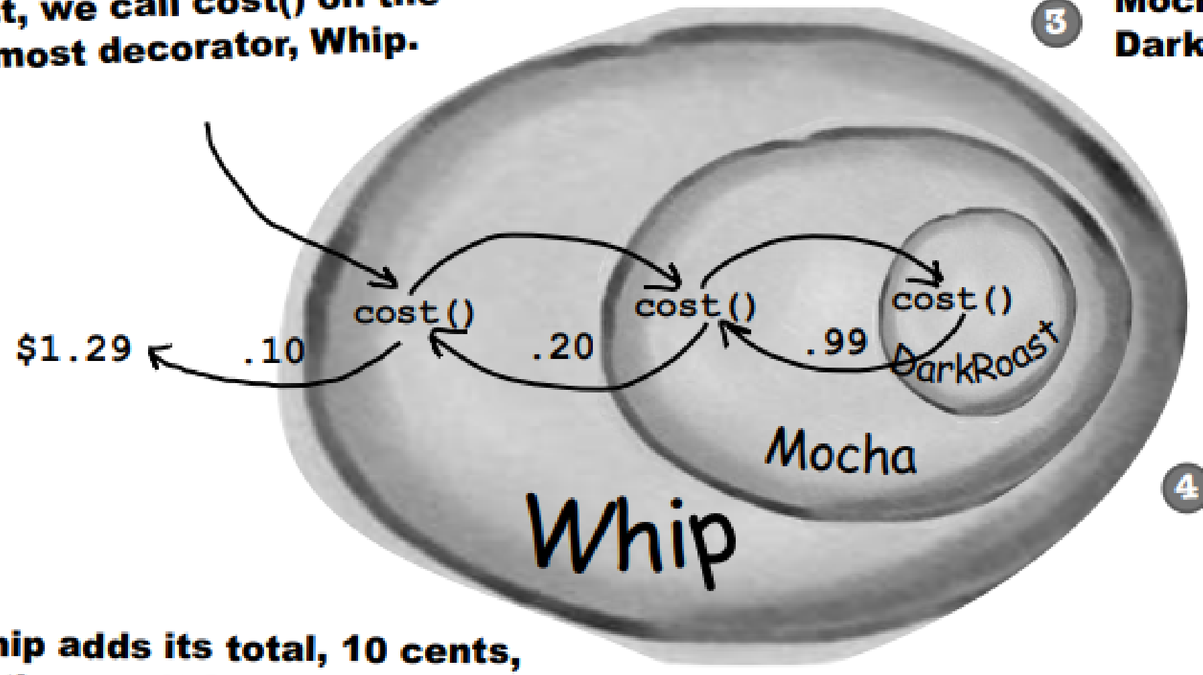
4 **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.**

(You'll see how in a few pages.)

2 **Whip calls cost() on Mocha.**

1 **First, we call cost() on the outmost decorator, Whip.**

3 **Mocha calls cost() on DarkRoast.**



4 **DarkRoast returns its cost, 99 cents.**

5 **Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.**

5 **Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.**

Implementing the Beverage class

Beverage class has 2 virtual methods: getDescription() and cost()

```
6   class Beverage{
7   protected:
8       string description{};
9   public:
10      virtual string getDescription(){
11          return description;
12      }
13
14      virtual double cost() = 0;
15  };
```

More specific Beverages

```
17  class Espresso : public Beverage{
18  public:
19      Espresso(){
20          description = "Espresso";
21      }
22
23      double cost() override{
24          return 1.99;
25      }
26  };
```

```
28  class HouseBlend : public Beverage{
29  public:
30      HouseBlend(){
31          description = "House Blend Coffee";
32      }
33
34      double cost() override{
35          return 0.89;
36      }
37  };
```

Condiment Decorator class

```
39    class CondimentDecorator : public Beverage{
40    public:
41        virtual string getDescription() = 0;
42    };
```

We can add multiple condiments to a beverage.

Implementing the concrete decorator

```
44  class Mocha : public CondimentDecorator{
45  private:
46      Beverage* beverage{};
47  public:
48      Mocha(Beverage* beverage){
49          this->beverage = beverage;
50      }
51
52      string getDescription() override{
53          return beverage->getDescription() + ", Mocha";
54      }
55
56      double cost() override{
57          return 0.20 + beverage->cost();
58      }
59  };
```

It's time to order some coffee!!!

```
78  int main(){
79      Beverage* order1 = new Espresso();
80      order1 = new Mocha(order1);
81      order1 = new Mocha(order1);
82      order1 = new Whip(order1);
83      cout<< order1->getDescription() << ": $" << order1->cost() << endl;
84
85      Beverage* order2 = new HouseBlend();
86      order2 = new Whip(order2);
87      cout<< order2->getDescription() << ": $" << order2->cost() << endl;
88
89      return 0;
90  }
```

```
Espresso, Mocha, Mocha, Whip: $2.49
House Blend Coffee, Whip: $0.99
```

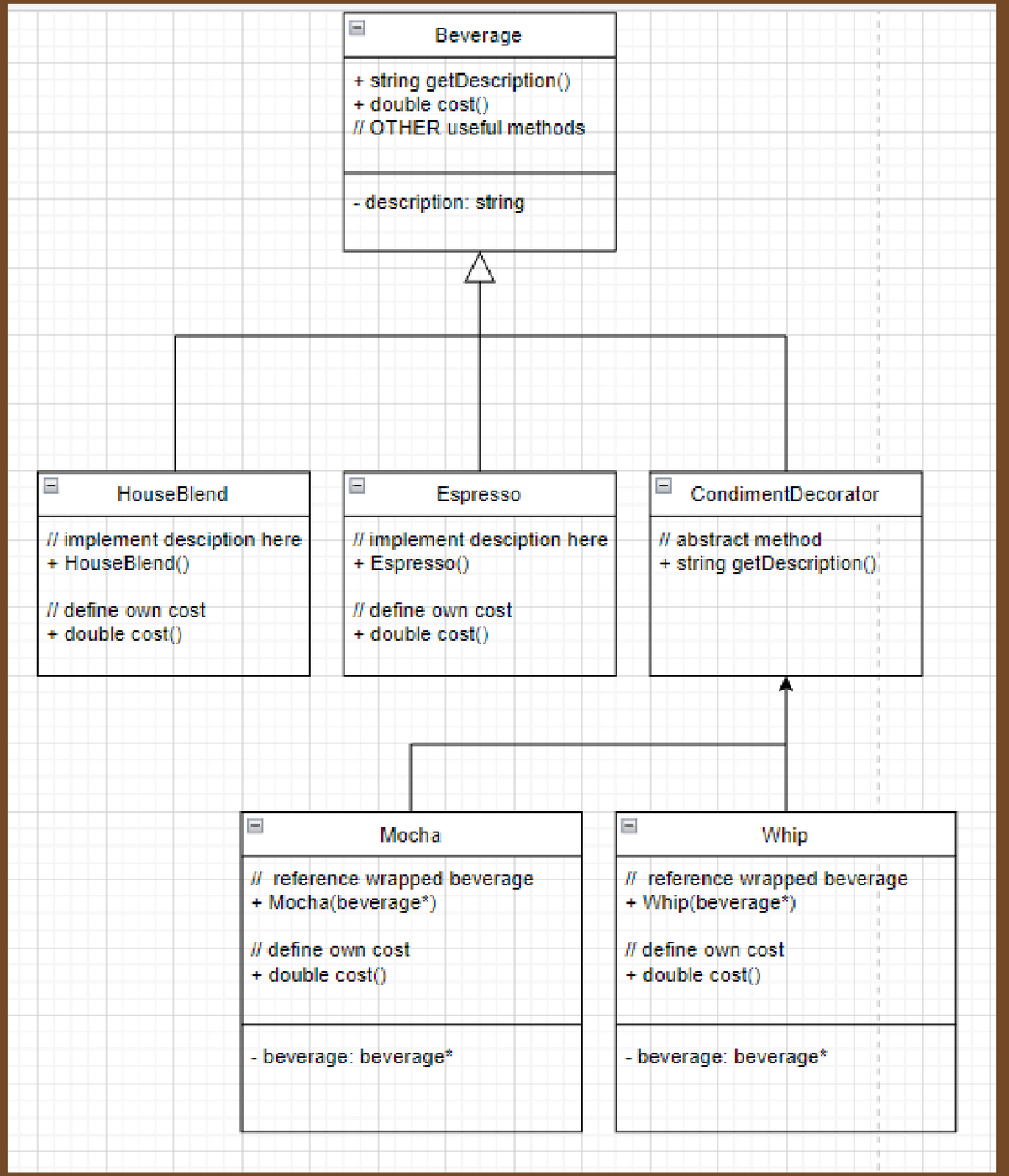
REMARK!!!

Advantages:

- Flexibility
- Extensibility
- Simplicity

Disadvantages:

- Hard to remove a specific layer of decorator.
- Layers' behaviour are dependent on the order.




THANK YOU FOR LISTENING