Design pattern - Decorator

Decorator pattern allows a user to add new functionality dynamically to an existing object without altering its structure or rewriting the code. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

The decorator design pattern follows the principle of "open-closed" design, which means that classes should be open for extension but closed for modification.

The thing in the middle is an main object that we want to alter the behavior of. The thing wrapping it is the decorator. The decorator has a component and is a component.

The wrapper must be the same type as the component, so it must inherit or implement from the same abstract class or interface. Leading the decorator to be easily interchangeable with the another decorator or the actual object.

The wrapper provides the extra functionality by adding new behaviors or modifying existing ones. This is a way to use **composition** instead of inheritance to extend the functionality of an object.

Composition

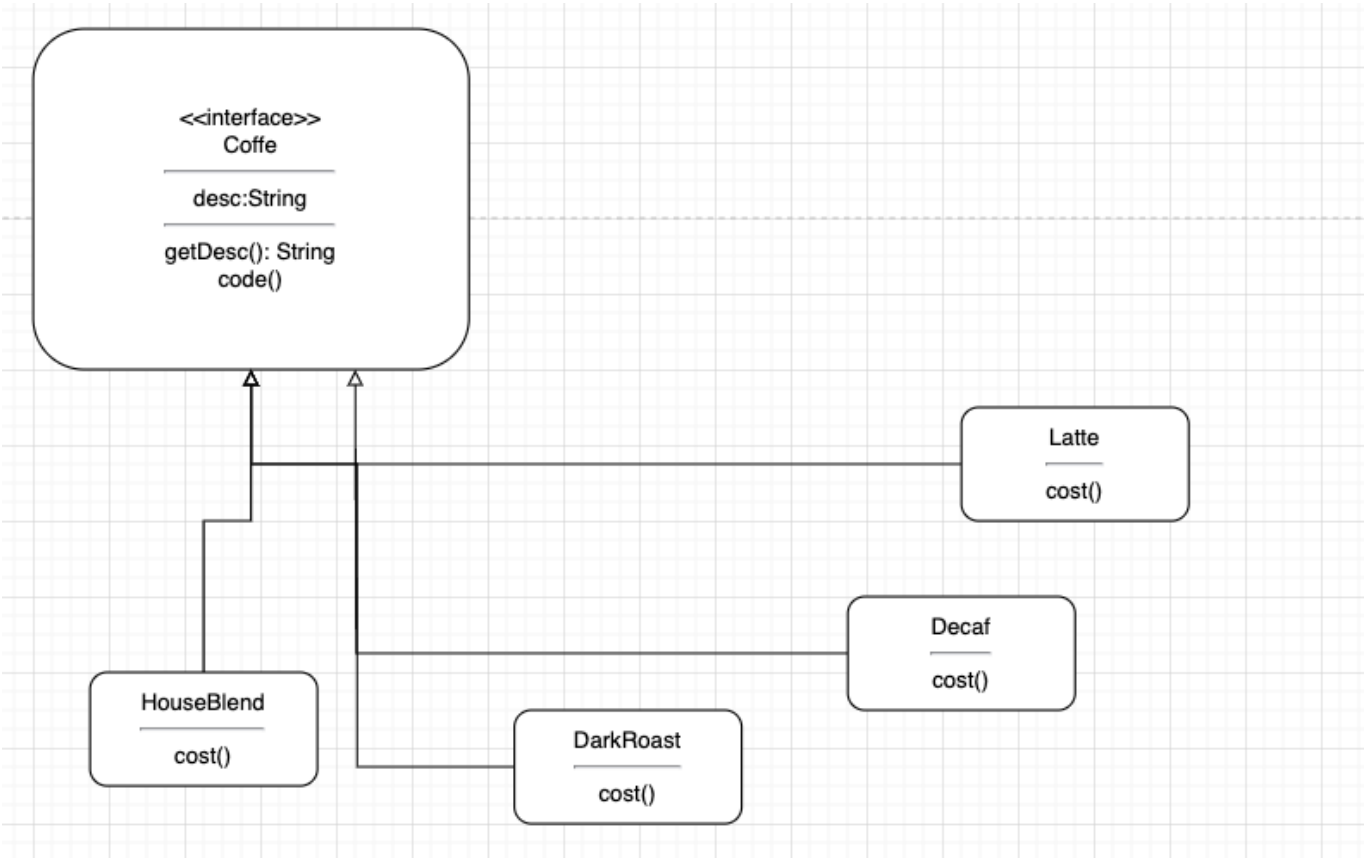
Composition means that means building complex objects by combining simpler or smaller objects. It is a way to create more complex and specialized objects by organizing them into a hierarchical structure.

This is done by creating classes that have references to other classes as member variables. These member variables are called "components" that make up the whole object. The relationship between the composite object and its parts is described as a "has-a" relationship, where the composite object has references to its component objects. This makes the code reuse, modularity, and flexibility in the design.

Composition differs from inheritance, which establishes an "is-a" relationship between classes. In composition, objects are combined by reference, whereas in inheritance, objects are derived from a base class.

It allows you to break down the problem into smaller, more manageable pieces, and build the overall functionality by combining these pieces.

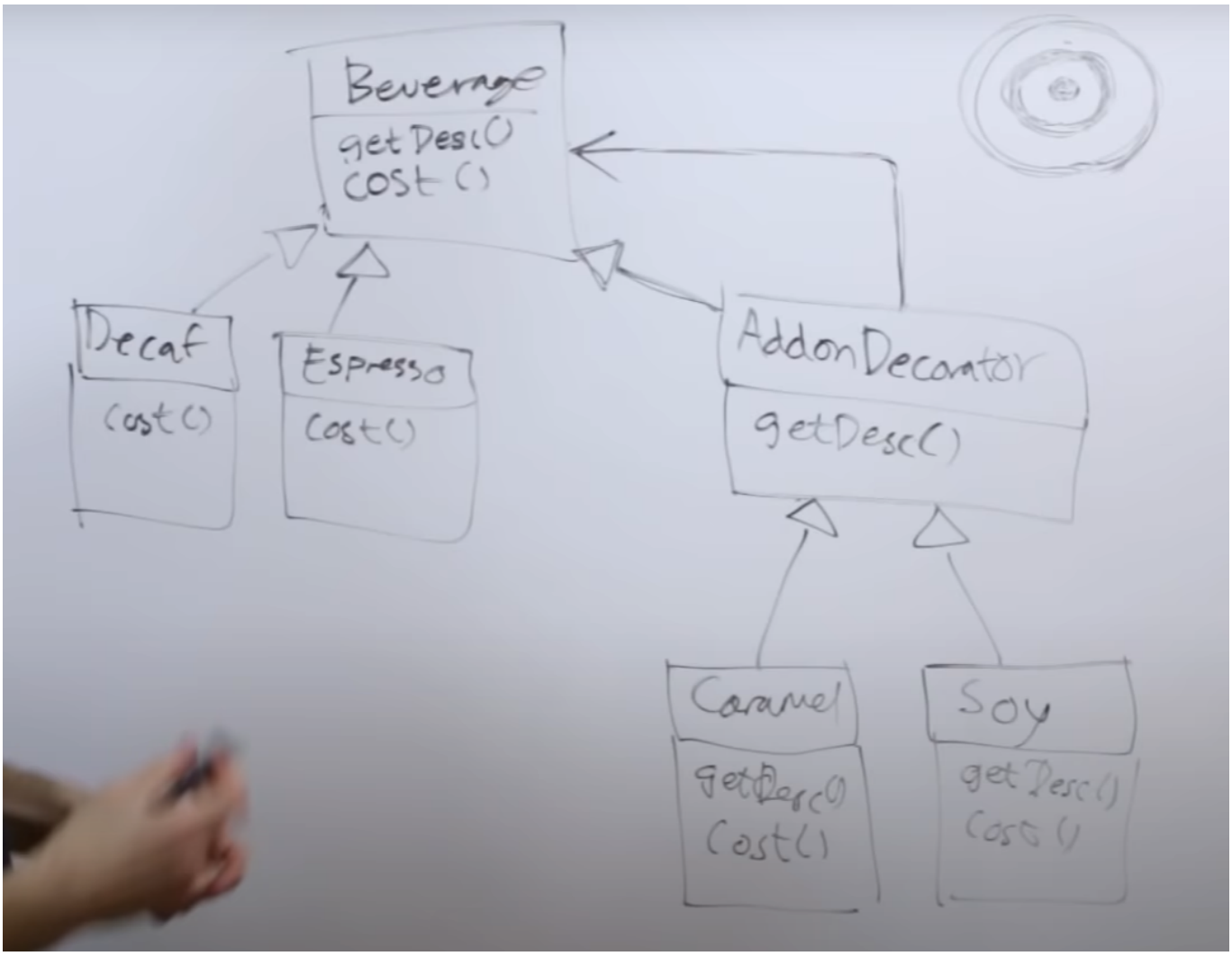
Inheritance (what it would look like if we used inheritance)



The problem with this is that we have to create a new class for every combination of the components. This is not scalable and will lead to a lot of classes.

When there a lot of classes, it is hard to keep track of them and it is hard to maintain them, this is called a class explosion.

With composition



Example

```
public interface Pizza {
    public String getDescription();
    public double getCost();
}
```

```
public class PlainPizza implements Pizza {
    public String getDescription() {
        return "Thin dough";
    }

    public double getCost() {
        return 4.00;
    }
}
```

```
public abstract class ToppingDecorator implements Pizza {
    protected Pizza tempPizza;
```

```
    public ToppingDecorator(Pizza newPizza) {
        tempPizza = newPizza;
    }

    public String getDescription() {
        return tempPizza.getDescription();
    }

    public double getCost() {
        return tempPizza.getCost();
    }
}
```

```
public class Mozzarella extends ToppingDecorator {
    public Mozzarella(Pizza newPizza) {
        super(newPizza);
        System.out.println("Adding Dough");
        System.out.println("Adding Moz");
    }

    public String getDescription() {
        return tempPizza.getDescription() + ", mozzarella";
    }

    public double getCost() {
        return tempPizza.getCost() + 0.50;
    }
}
```

```
public class TomatoSauce extends ToppingDecorator {
    public TomatoSauce(Pizza newPizza) {
        super(newPizza);
        System.out.println("Adding Sauce");
    }

    public String getDescription() {
        return tempPizza.getDescription() + ", tomato sauce";
    }

    public double getCost() {
        return tempPizza.getCost() + 0.35;
    }
}
```

```
public class TestPizza {
    public static void main(String[] args) {
```

```
        Pizza basicPizza = new TomatoSauce(new Mozzarella(new
PlainPizza()));

        System.out.println("Ingredients: " + basicPizza.getDescription());
        System.out.println("Price: " + basicPizza.getCost());
    }
}
```

Output:

```
Adding Dough
Adding Moz
Adding Sauce
Ingredients: Thin dough, mozzarella, tomato sauce
Price: 4.85
```