Decorator design pattern

- Decorator design pattern is called as wrapper design pattern.
- 'Gang of four' states decorator design pattern is about 'extension of objects dynamically.' (at run time).
- The decorator design pattern will add extra responsibilities (features) to an object dynamically, without changing underlying class.

Decorator design pattern can be applied when
- Inheritance is not feasible (possible).
- Legacy code is to be modified.

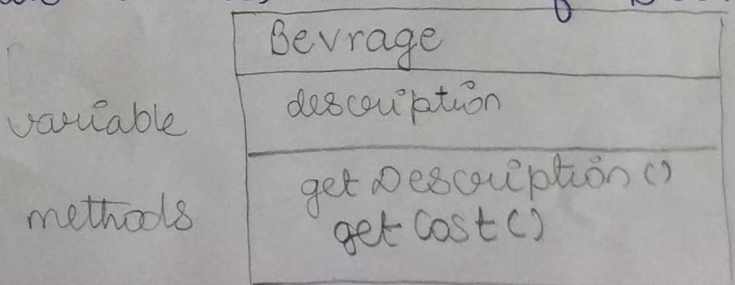
#Open closed principle

'classes should be open for extension, but not for modification'. It means whenever you want to add any feature to program. Do not modify existing class. As it may result in bugs, rather extend (stretch) the class dynamically by writing new code in new class.

- Explanatory Scenario for decorator design pattern.

suppose we have a class called 'Bevrage'. And it has members like variable description, which tells us the name of Bevrage.

And methods like getDescription(), getCost(). Which return description (name) & cost of Bevrage.

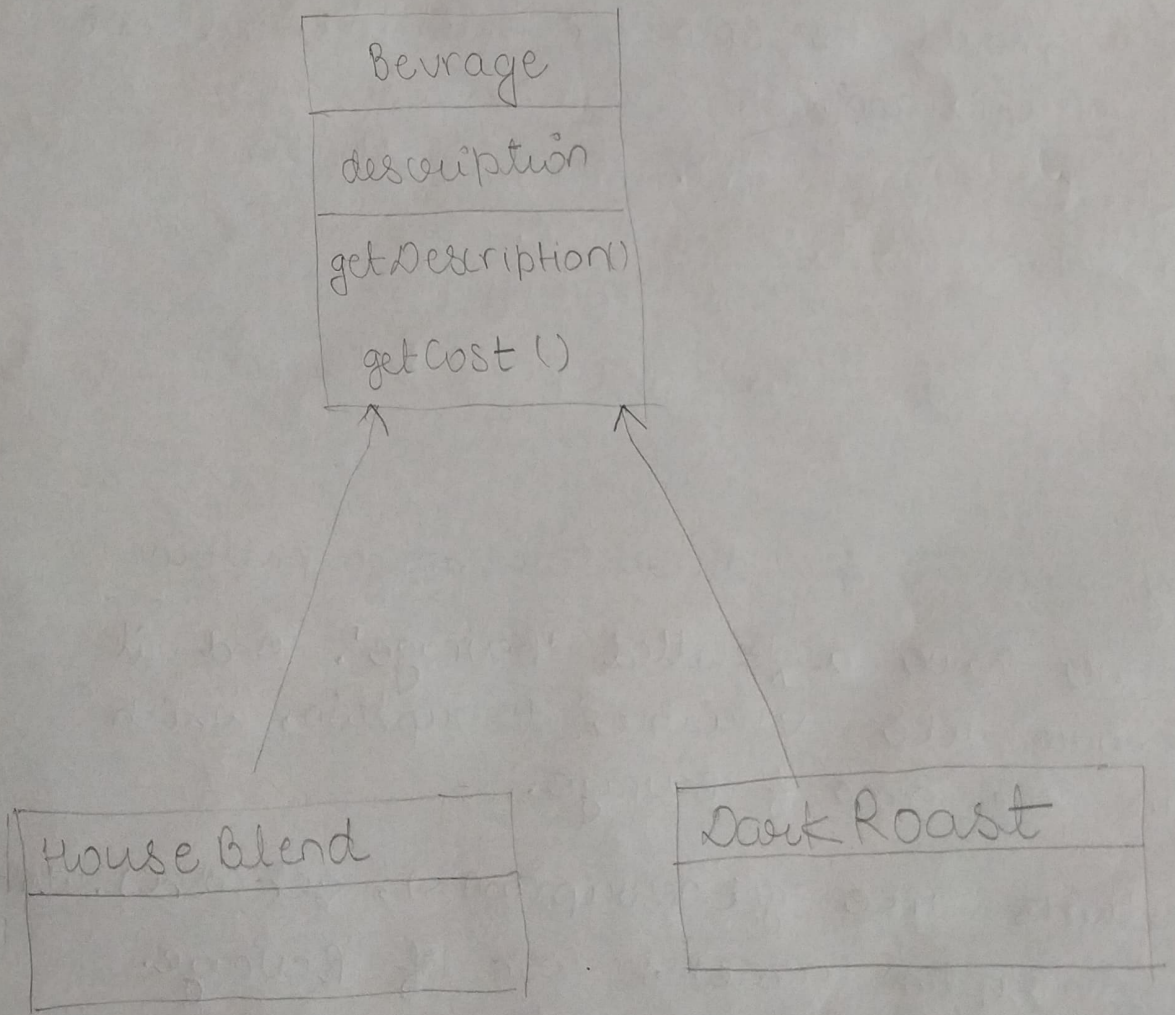


Now this 'Bevrage' class is scaffold (blue print / basic structure) of all Bevarage items like 'Houseblend' coffee and 'DarkRoast' coffee.

* (we can consider lot of coffee varities, but let us consider 'Houseblend' coffee & 'DarkRoast' coffee, as of now, for simplicity).

Blueprint of 'Bevrage' items defined as class 'Bevrage', is already with us.

Thus, for sake of reusability and not to define members like description, getDescription, getCost() in 'HouseBlend' and 'DarkRoast' coffee again. Let class 'HouseBlend' and 'DarkRoast' inherit from 'Bevrage'.



- getCost() method of 'Beverage', varies from subclass to subclass. So, let's keep getCost() as abstract and 'Beverage' as abstract class

- The inheritance approach is considered, keeping possibility in mind, that what if there are large number of coffee varities into picture.

- Inheritance will not let us write redundant code.

- Suppose there are coffee varities like 'BrewCoffee', 'MochaCoffee', 'Espresso'. Then we can directly inherit properties of coffee from class 'Beverage'.

- Without defining those properties like description, getCost(), getDescription() again and again in individual classes like 'HouseBlend', 'DarkRoast', 'BrewCoffee', 'MochaCoffee', etc.

- Inheritance seems to be perfect (well, so far)!

Decorators into picture.

Now what if, I want to use condiments (supplements) like milk & whip (cream) for my coffee 'HouseBlend' and 'DarkRoast'. And calculate their cost? & describe them?

How will I create objects of given classes.

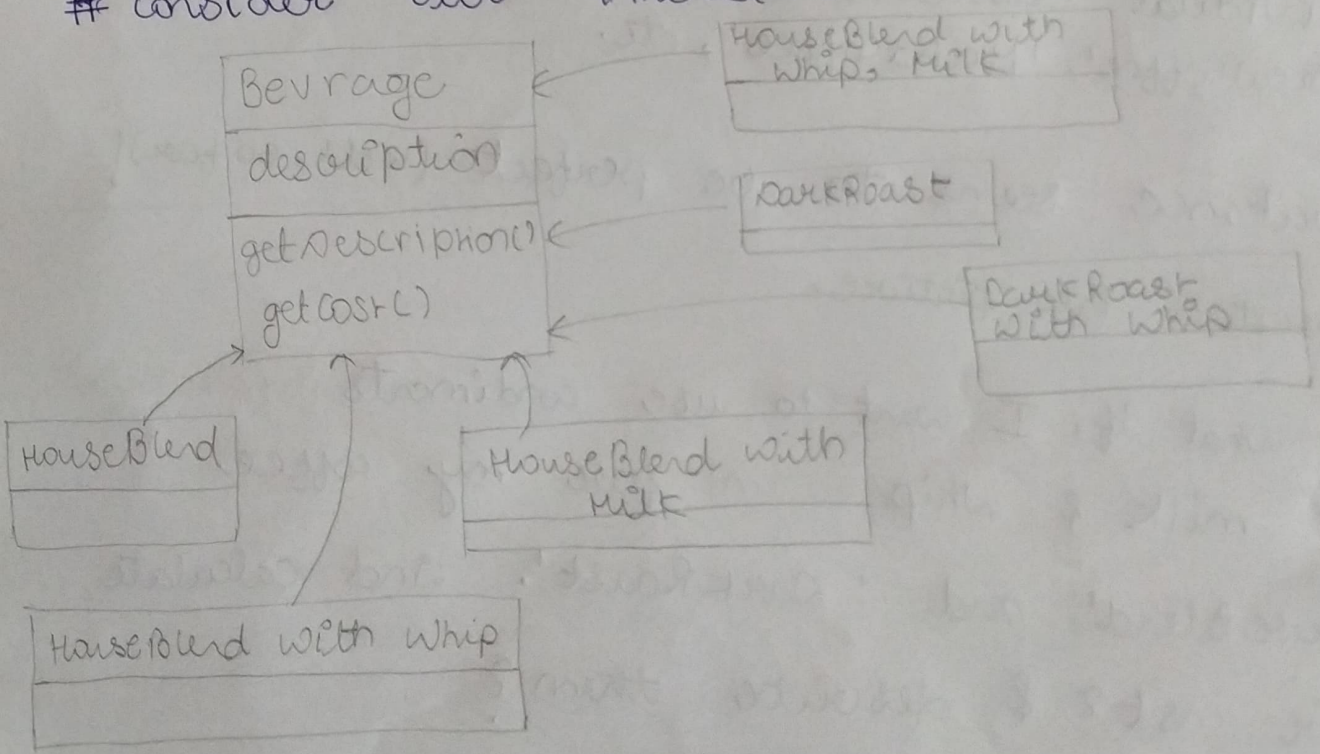
- → HouseBlend
- → HouseBlend with whip
- → HouseBlend with milk
- → HouseBlend with milk, whip
- → HouseBlend with 2 serve of milk

- → DarkRoast
- → DarkRoast with whip
- → DarkRoast with whip, milk
- → DarkRoast with milk
- → DarkRoast with 2 serve of whip, 1 serve of milk.

Do I need to write 10 concrete classes to create objects of 10 possible combination of coffee variety and condiments (supplements) ?

Well then, we can have 'N' (n = 1 .to ∞) different combination of coffee variety and condiments. So are we gonna write 'N' different classes, statically, for creation of their respective objects.

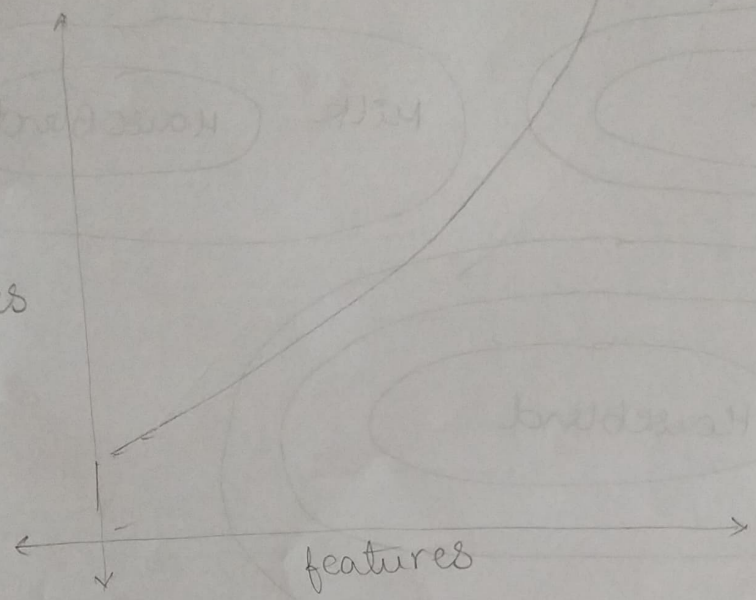
consider our inheritance model



Pitfall of Inheritance model

Well we can tell you that inheritance model is gonna fail in this case, because number of classes is set high at compile time itself. whenever number of (features) combinations of coffee variety & condiments increase.

When classes go on increasing numberwise, with addition of features. This leads to 'class explosion' in object oriented programming.



$y = e^x$

exponential rise of number of classes with increase in features.

CLASS EXPLOSION

Well, then there has to be some way against **statii, rigid** approach of Inheritance. which carries excess baggage (in this case). Number of classes (inherit) set their property at compile time itself.

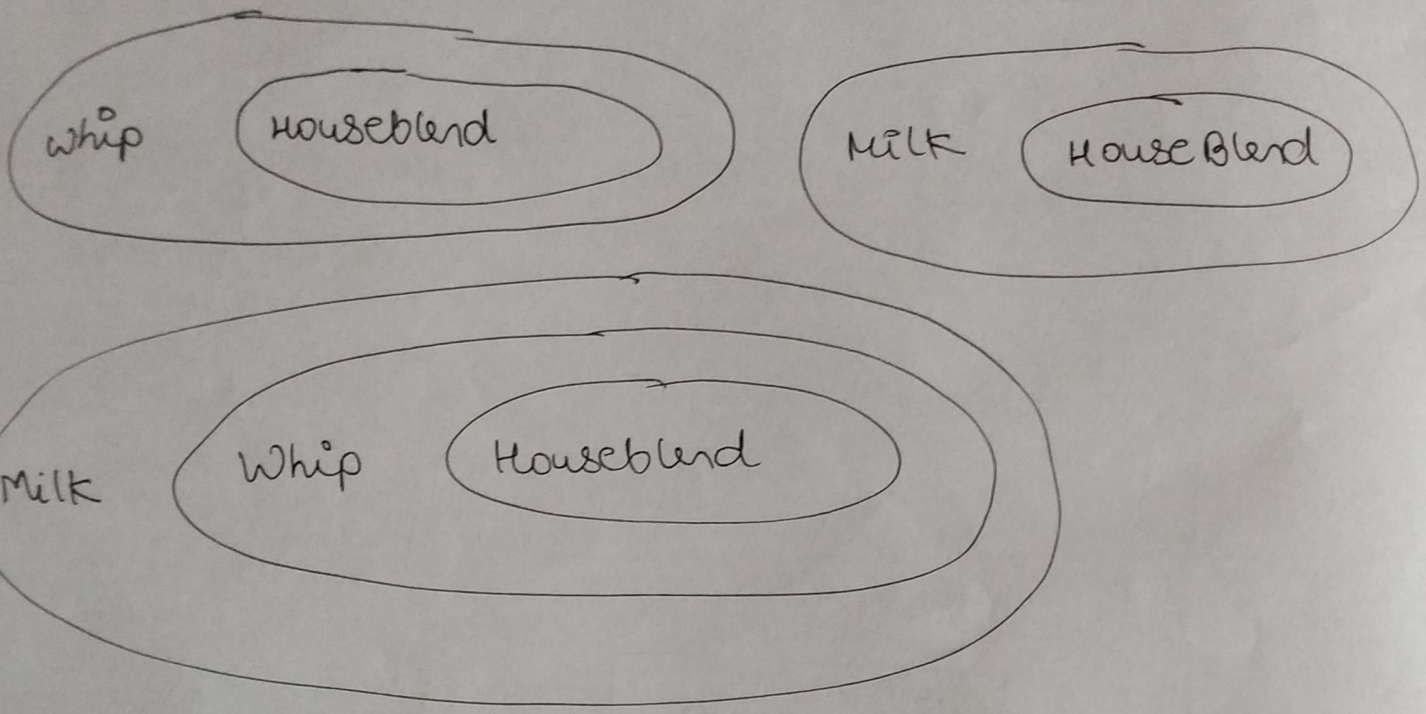
- This is the time we moved to 'composition'. (And decorator design pattern).
- composition means storing one object into another.

Composition advantage.

Composition is dynamic (takes place at runtime) & flexible, opposite to Inheritance.

Composition helps classes to hold another classes dynamically, when object is created at run time.

With help of composition, we can create given objects.



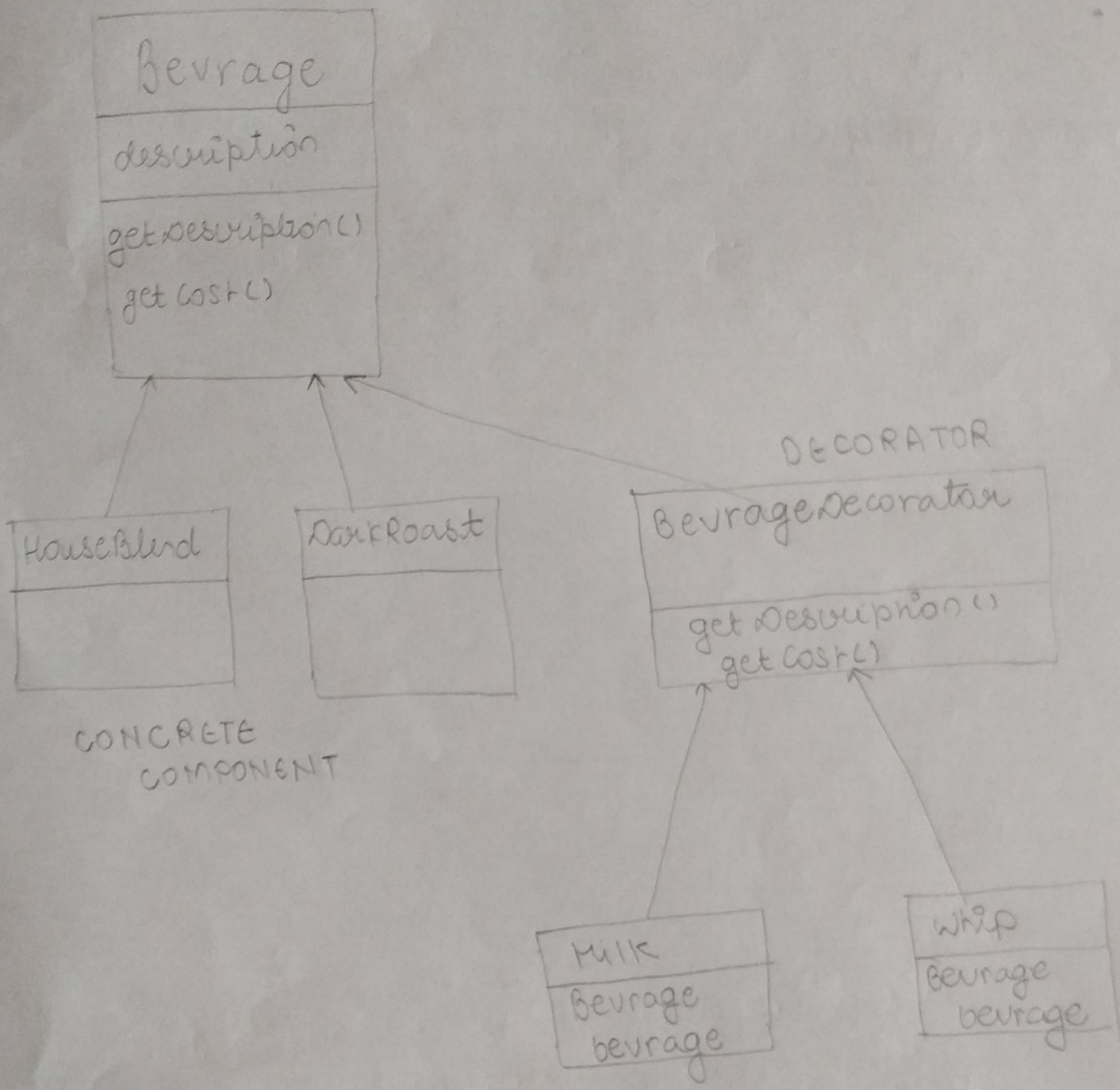
This approach of creating objects can be achieved without increasing number of classes & class explosion.

We need limited (handful) number of classes, for this to work.

Let us use architecture of decorator design pattern to make, creation of such objects possible

And later move to the code for clearer understanding.

COMPONENT


Beverage
description
getDescription()
getCost()
DECORATOR
BeverageDecorator
getDescription()
getCost()
HouseBlend
DarkRoast
CONCRETE COMPONENT
Milk
Beverage beverage
Whip
Beverage beverage