3. Decorator - Design Pattern

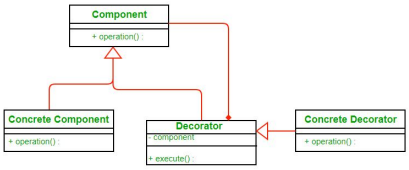
✏️ Wrapping class with decorator to provide extra features to it.

Not using it may cause problem like **Class Explosion.**

The Decorator Pattern | Set 2 (Introduction and Design) - GeeksforGeeks

A Computer Science portal for geeks. It contains well written, well thought and well explained computer science and programming articles, quizzes and practice/competitive programming/company interview Questions.

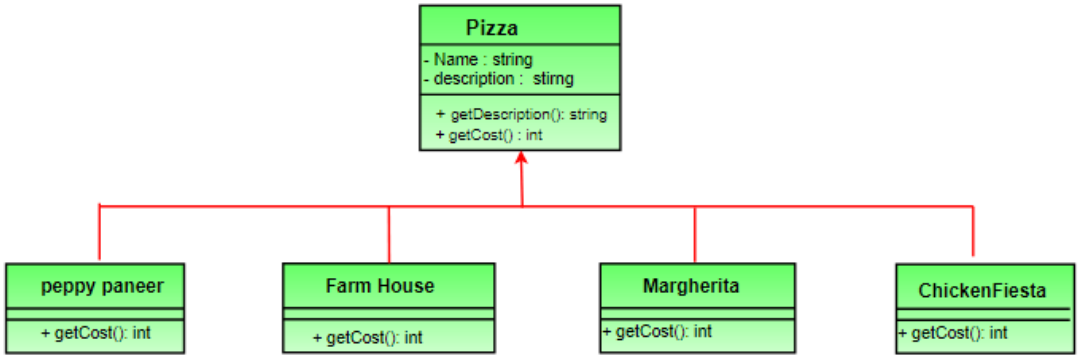
https://www.geeksforgeeks.org/the-decorator-pattern-set-2-introduction-and-design/



Lets take an example of modelling pizza order for a shop.

We can create a interface for base Pizza and then for every different kind of pizza, we will inherit this base class to new subclass of that particular pizza type.

Example:



It's good **BUT** in real life**,** customer may ask for different topping on their pizza which can be of many different types. How to model that?

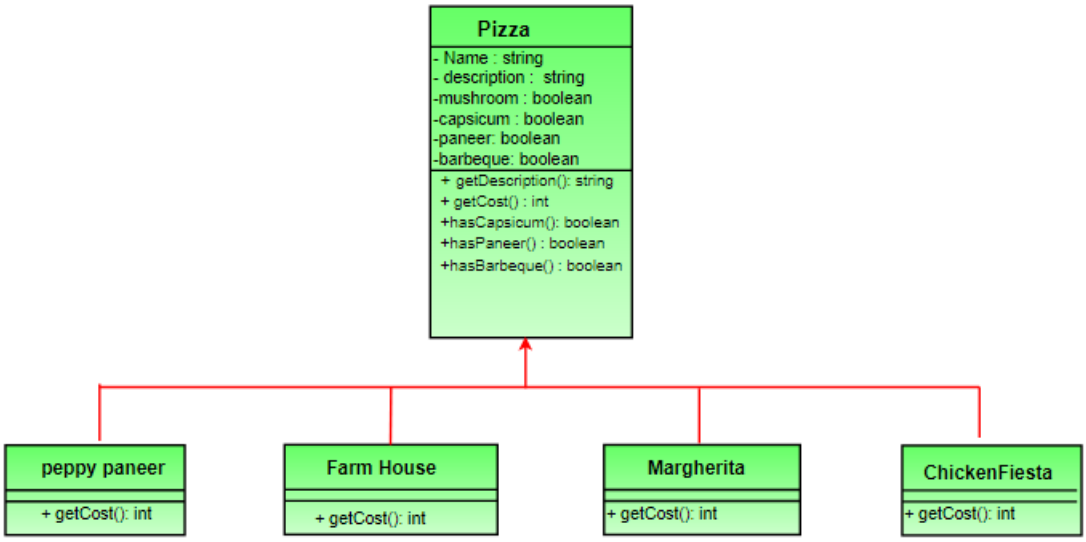
Approach 1:

One way could be, we make subclass for every combination of pizza type with toppings.

Very BAD IDEA → it will cause Class Explosion problem

Approach 2:

Instead, we can add boolean in base class to track which topping customer want and not. It's fine model but this model will break **Open-Close Principle,** as if we are required to add more toppings we will have to change the code inside base class which is not good.



Approach 3: Decorater-Pattern

Decorator class had both **is-a** and **has-a** relationship with base-Pizza class.

Because decorator concrete class has Generic base-Pizza class to take any of its subclass as input, and also decorator abstract class is inheriting from base-Pizza class.

Code:

```
#include <iostream>
using namespace std;

class BasePizza
{
public:
    virtual int cost() = 0;
};

class FarmHouse : public BasePizza
{
public:
    int cost()
    {
        return 200;
    }
};

class Margherita : public BasePizza
{
public:
    int cost()
    {
        return 100;
    }
};

// #### TOPPINGS

class ToppingDecorator : public BasePizza
{
public:
    virtual int cost() = 0;
};

class ExtraCheese : public ToppingDecorator
{
    BasePizza *basePizza;

public:
    ExtraCheese(BasePizza *pizza)
    {
        this->basePizza = pizza;
    }

    int cost()
    {
        return basePizza->cost() + 20;
```

```
    }
};

class Mushroom : public ToppingDecorator
{
    BasePizza *basePizza;

public:
    Mushroom(BasePizza *pizza)
    {
        this->basePizza = pizza;
    }

    int cost()
    {
        return basePizza->cost() + 50;
    }
};

int main()
{

    // my Order : margherita + ExtraCheese

    BasePizza *myPizza = new ExtraCheese(new Margherita());

    cout << myPizza->cost() << "\n"; // 100+20=120

    // Now add Mushroom to it

    myPizza = new Mushroom(myPizza);

    cout << myPizza->cost() << "\n"; // 100+20+50=170

    // Wow, Isn't it awesome!

    return 0;
}
```