Decorator Pattern

Lecture-6

Decorator Pattern

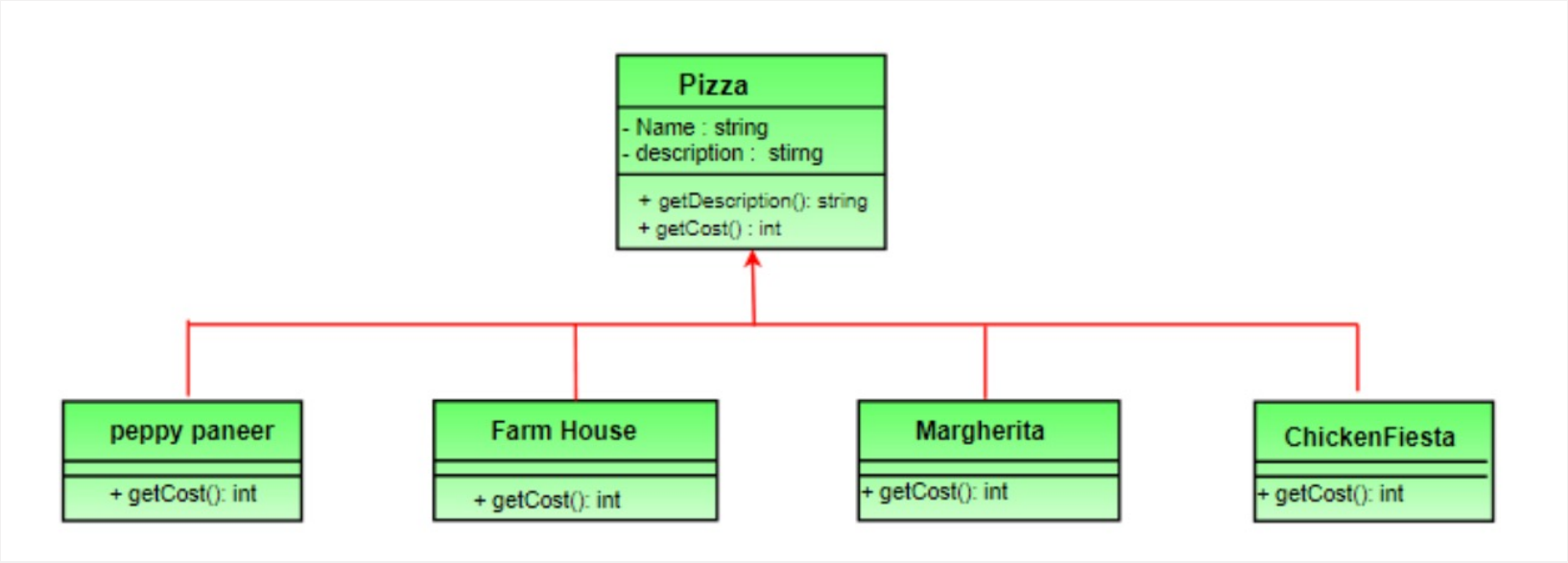
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Client-specified embellishment of a core object by recursively wrapping it.
- Wrapping a gift, putting it in a box, and wrapping the box.

Problem

You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

Scenario

Suppose we are building an application for a pizza store and we need to model their pizza classes. Assume they offer four types of pizzas namely Peppy Paneer, Farmhouse, Margherita and Chicken Fiesta. Initially we just use inheritance and abstract out the common functionality in a **Pizza** class.

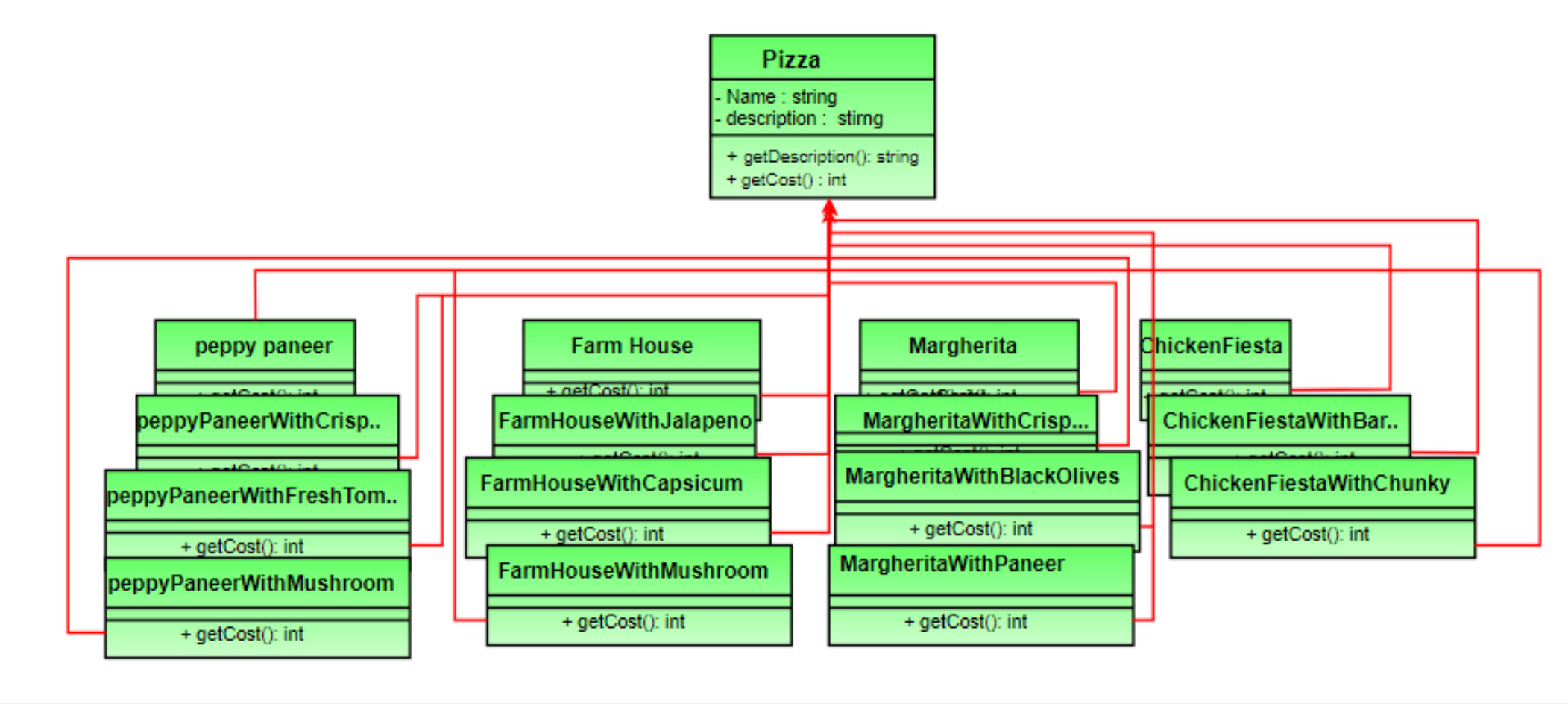


New Requirement

Each pizza has different cost. We have overridden the getCost() in the subclasses to find the appropriate cost. Now suppose a new requirement, in addition to a pizza, customer can also ask for several toppings such as Fresh Tomato, Paneer, Jalapeno, Capsicum, Barbeque, etc. Let us think about for sometime that how do we accommodate changes in the above classes so that customer can choose pizza with toppings and we get the total cost of pizza and toppings the customer chooses.

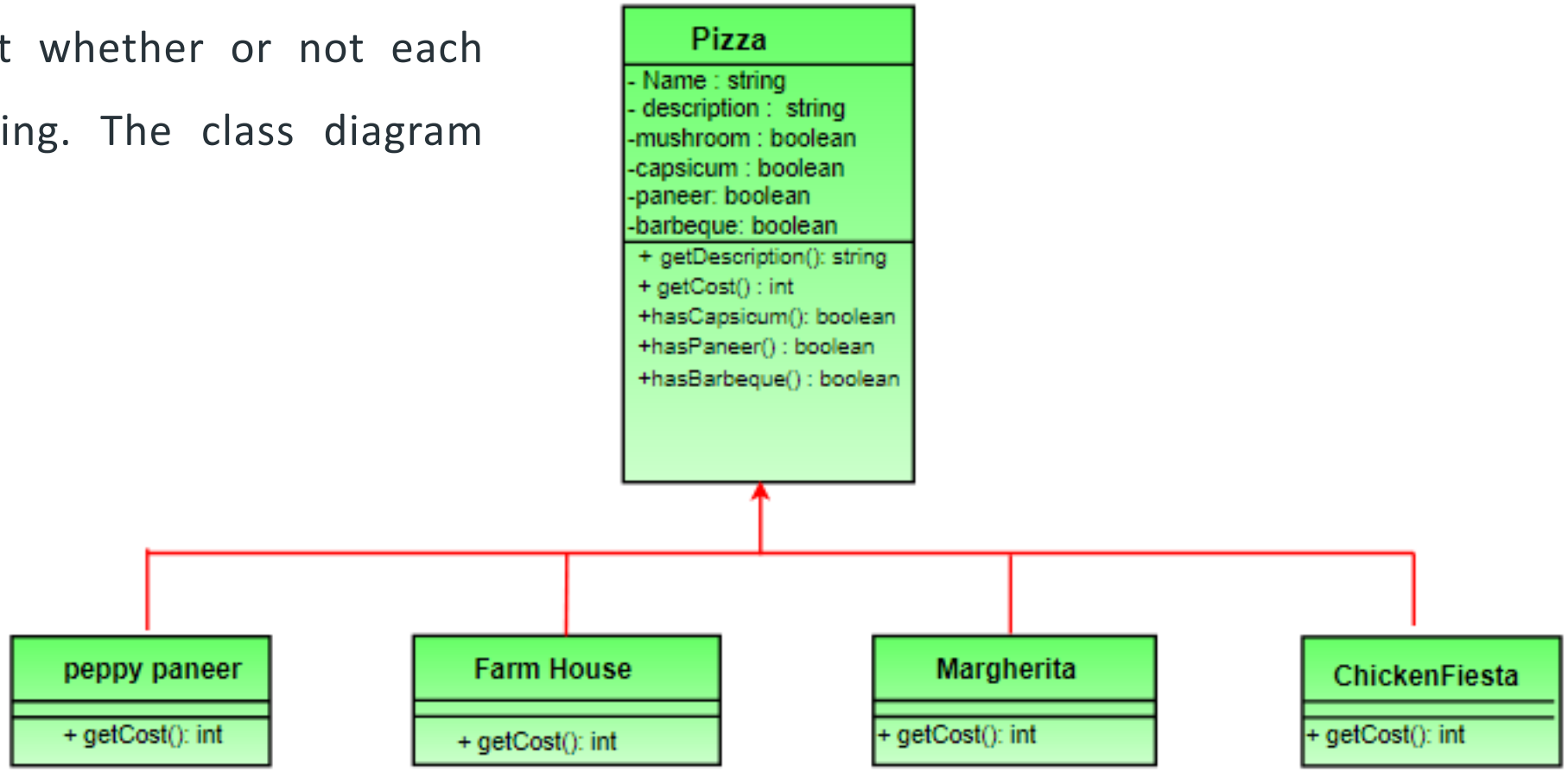
Option 1

Create a new subclass for every topping with a pizza. The class diagram would look like



Option 2

Let's add instance variables to pizza base class to represent whether or not each pizza has a topping. The class diagram would look like



Option 2 - Implementation

```
// Sample getCost() in super class
public int getCost()
{
    int totalToppingsCost = 0;
    if (hasJalapeno() )
        totalToppingsCost += jalapenoCost;
    if (hasCapsicum() )
        totalToppingsCost += capsicumCost;

    // similarly for other toppings
    return totalToppingsCost;
}
```

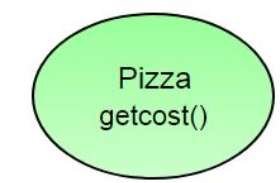
```
// Sample getCost() in subclass
public int getCost()
{
    // 100 for Margherita and super.getCost()
    // for toppings.
    return super.getCost() + 100;
}
```

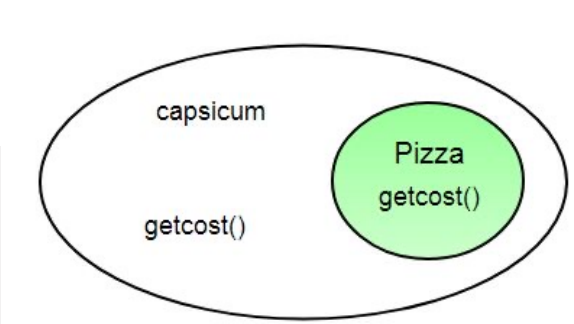
Option 2 - Drawbacks

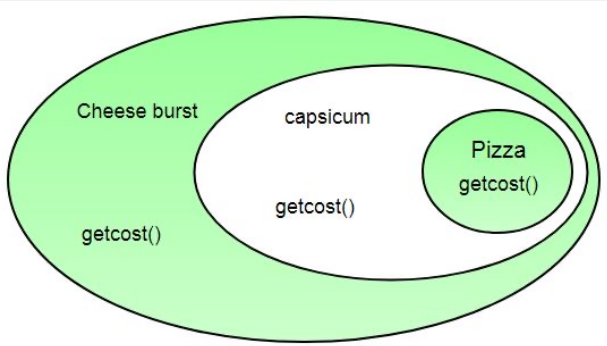
- Price changes in toppings will lead to alteration in the existing code.
- New toppings will force us to add new methods and alter getCost() method in superclass.
- For some pizzas, some toppings may not be appropriate yet the subclass inherits them.
- What if customer wants double capsicum or double cheeseburst?

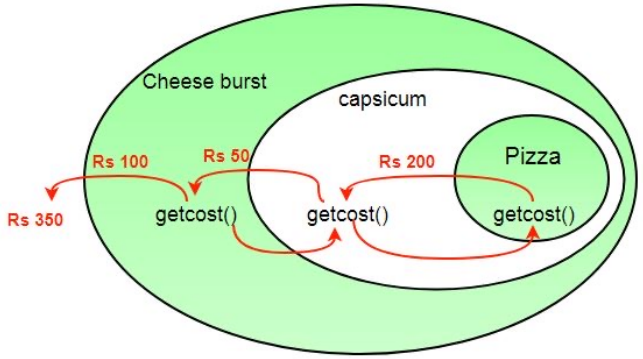
In short our design violates one of the most popular design principle – **The Open-Closed Principle** which states that classes should be open for extension and closed for modification

Solution - Decorator Pattern







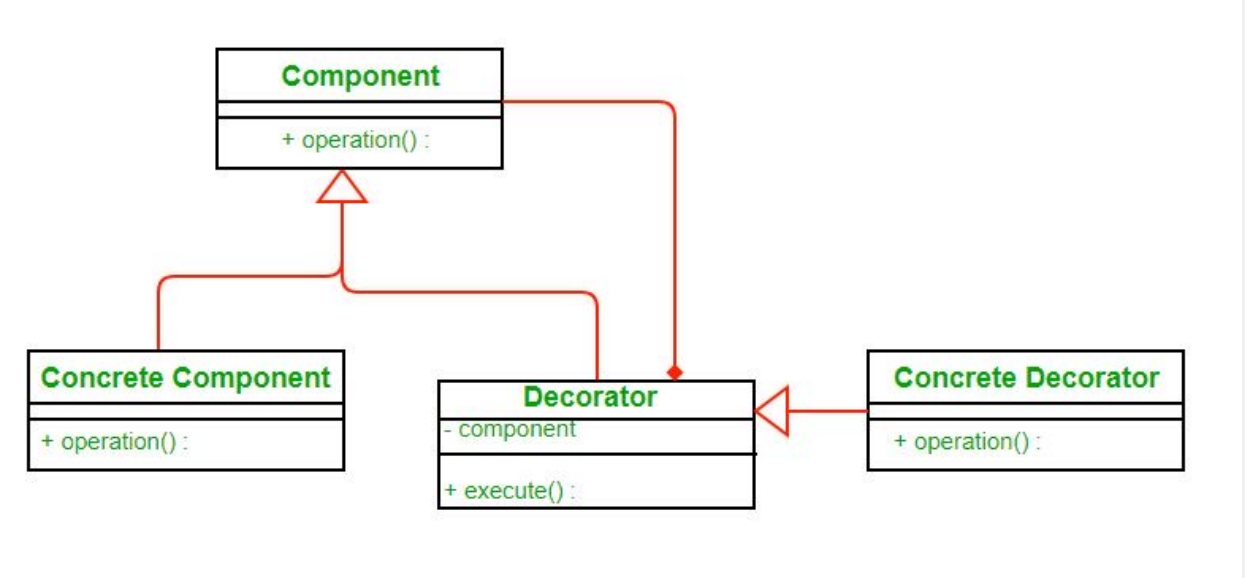


Properties of Decorator

- Decorators have the same super type as the object they decorate.
- You can use multiple decorators to wrap an object.
- Since decorators have same type as object, we can pass around decorated object instead of original.
- We can decorate objects at runtime.

Class Diagram

- Each component can be used on its own or may be wrapped by a decorator.
- Each decorator has an instance variable that holds the reference to component it decorates(HAS-A relationship).
- The ConcreteComponent is the object we are going to dynamically decorate.



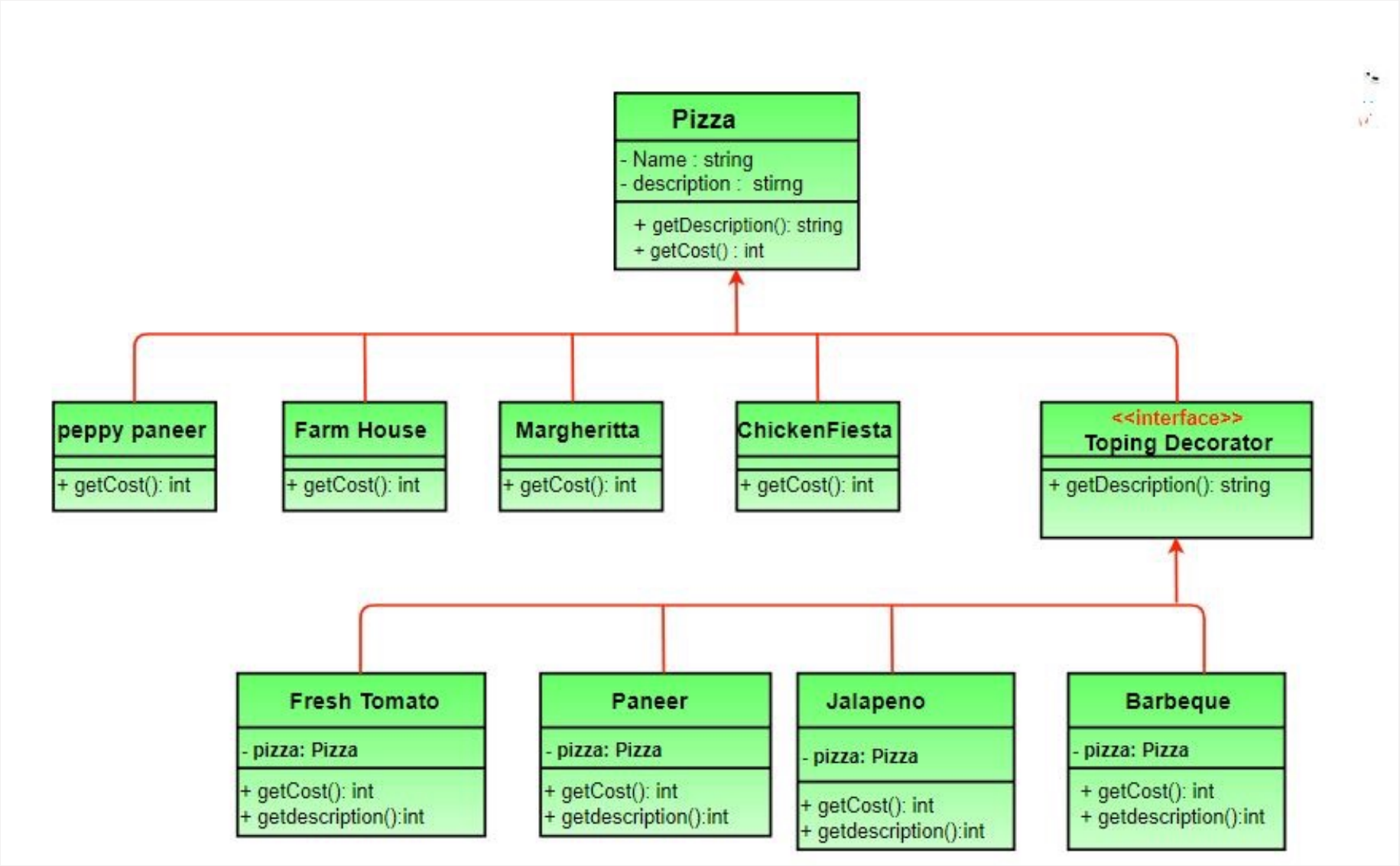
Advantages

- The decorator pattern can be used to make it possible to extend (decorate) the functionality of a certain object at runtime.
- The decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at runtime for individual objects.
- Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects.

Disadvantages

- Decorators can complicate the process of instantiating the component because you not only have to instantiate the component, but wrap it in a number of decorators.
- It can be complicated to have decorators keep track of other decorators, because to look back into multiple layers of the decorator chain starts to push the decorator pattern beyond its true intent.

Decorator Class Diagram



Decorator Implementation

```
// Abstract Pizza class (All classes extend
// from this)
abstract class Pizza
{
    // it is an abstract pizza
    String description = "Unkknown Pizza";

    public String getDescription()
    {
        return description;
    }

    public abstract int getCost();
}
```

```
// The decorator class :  It extends Pizza to be
// interchangeable with it toppings decorator can
// also be implemented as an interface
abstract class ToppingsDecorator extends Pizza
{
    public abstract String getDescription();
}

// Concrete pizza classes
class PeppyPaneer extends Pizza
{
    public PeppyPaneer() { description = "PeppyPaneer"; }
    public int getCost() {  return 100; }
}
class FarmHouse extends Pizza
{
    public FarmHouse() {  description = "FarmHouse"; }
    public int getCost() { return 200; }
}
class Margherita extends Pizza
{
    public Margherita()  { description = "Margherita"; }
    public int getCost() { return 100;  }
}
class ChickenFiesta extends Pizza
{
    public ChickenFiesta() { description = "ChickenFiesta";}
    public int getCost() { return 200; }
}
class SimplePizza extends Pizza
{
public SimplePizza() { description = "SimplePizza"; }
public int getCost() {  return 50;  }
}
```

Decorator Implementation

```
// Concrete toppings classes
class FreshTomato extends ToppingsDecorator
{
    // we need a reference to obj we are decorating
    Pizza pizza;

    public FreshTomato(Pizza pizza) { this.pizza = pizza; }
    public String getDescription() {
        return pizza.getDescription() + ", Fresh Tomato ";
    }
    public int getCost() { return 40 + pizza.getCost(); }
}
class Barbeque extends ToppingsDecorator
{
    Pizza pizza;
    public Barbeque(Pizza pizza) {  this.pizza = pizza;  }
    public String getDescription() {
        return pizza.getDescription() + ", Barbeque ";
    }
    public int getCost() { return 90 + pizza.getCost(); }
}
class Paneer extends ToppingsDecorator
{
    Pizza pizza;
    public Paneer(Pizza pizza)  {  this.pizza = pizza; }
    public String getDescription() {
        return pizza.getDescription() + ", Paneer ";
    }
    public int getCost()  {  return 70 + pizza.getCost(); }
}
```

```
// Driver class and method
class PizzaStore
{
    public static void main(String args[])
    {
        // create new margherita pizza
        Pizza pizza = new Margherita();
        System.out.println( pizza.getDescription() +
                            " Cost :" + pizza.getCost());

        // create new FarmHouse pizza
        Pizza pizza2 = new FarmHouse();

        // decorate it with freshtomato topping
        pizza2 = new FreshTomato(pizza2);

        //decorate it with paneer topping
        pizza2 = new Paneer(pizza2);

        System.out.println( pizza2.getDescription() +
                            " Cost :" + pizza2.getCost());
    }
}
```