PROGRAMMING SYSTEMS

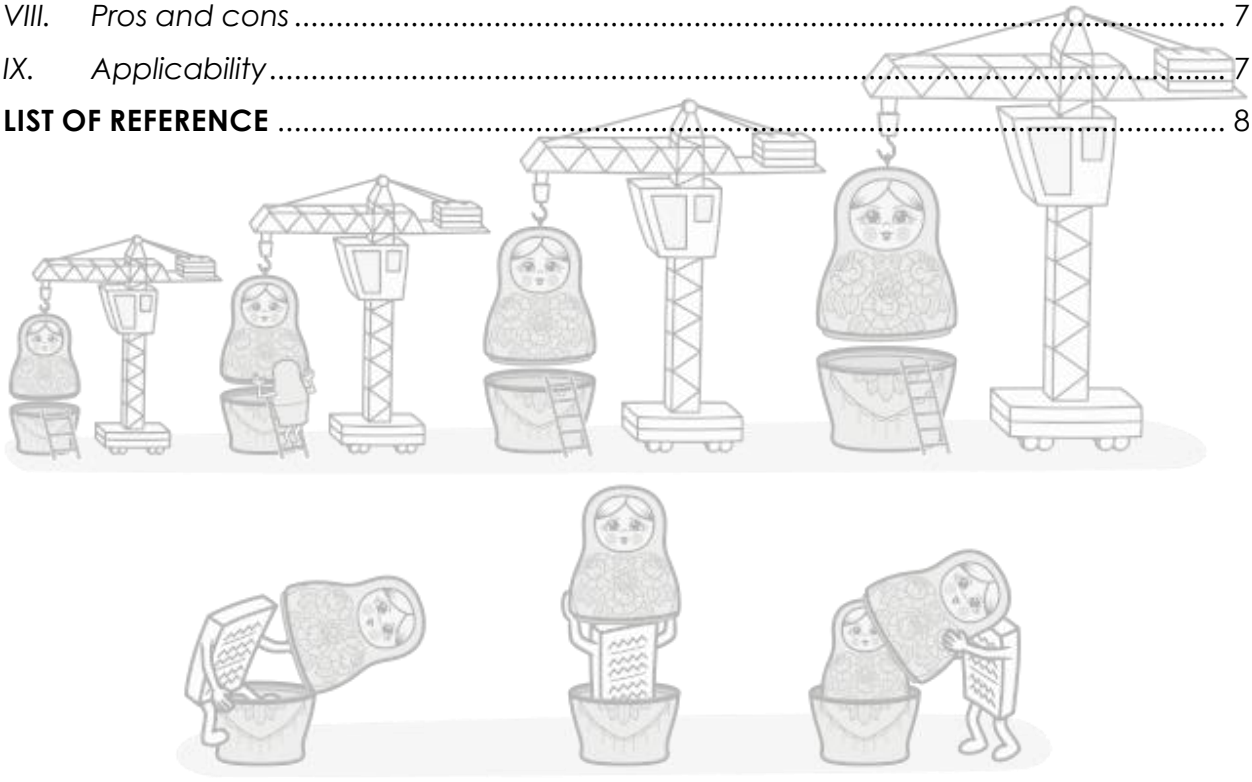
2019

DECORATOR PATTERN

THE SHADOW

18125035 Nguyễn Thành Đạt
18125044 Phạm Hoài Phú Thịnh
18125049 Huỳnh Minh Hiếu

Contents



I. Problem

Imagine that you are going to open a beverage shop. Customers choose different beverages from its composite. For example: black tea and 3 different types of topping: black bubble, white bubble, pudding.

Now, you want to develop an application to calculate the cost of beverage in your shop.

Figure 1 - Beverage shop

II. Traditional solution

Extending a class is the first thing that comes to mind when you need to alter an object's behavior.

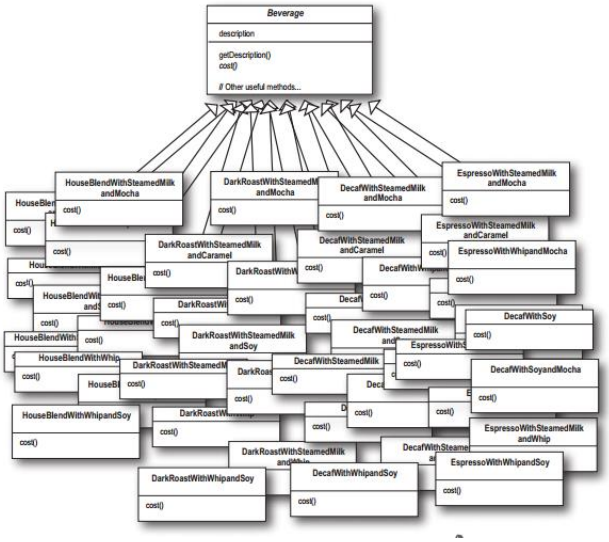


Figure 2 - Inheritance

However, inheritance has several serious caveats that you need to be aware of.

- Inheritance is static. You can't alter the behavior of an existing object at runtime. You can only replace the whole object with another one that's created from a different subclass.
- Subclasses can have just one parent class. In most languages, inheritance doesn't let a class inherit behaviors of multiple classes at the same time.

III. 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.

Wrapper is the alternative nickname for the Decorator pattern that clearly expresses the main idea of the pattern. A "wrapper" is an object that can be linked with some "target" object. The wrapper contains the same set of methods as the target and delegates to it all requests it receives. However, the wrapper may alter the result by doing something either before or after it passes the request to the target.

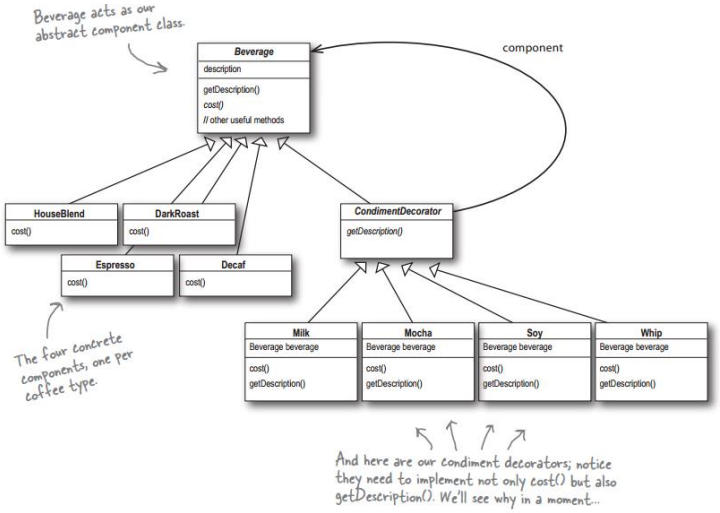


Figure 3 - Decorator pattern

The wrapper implements the same interface as the wrapped object. That's why from the client's perspective these objects are identical. Make the wrapper's reference field accept any object that follows that interface. This will let you cover an object in multiple wrappers, adding the combined behavior of all the wrappers to it.

IV. Real-world analogy

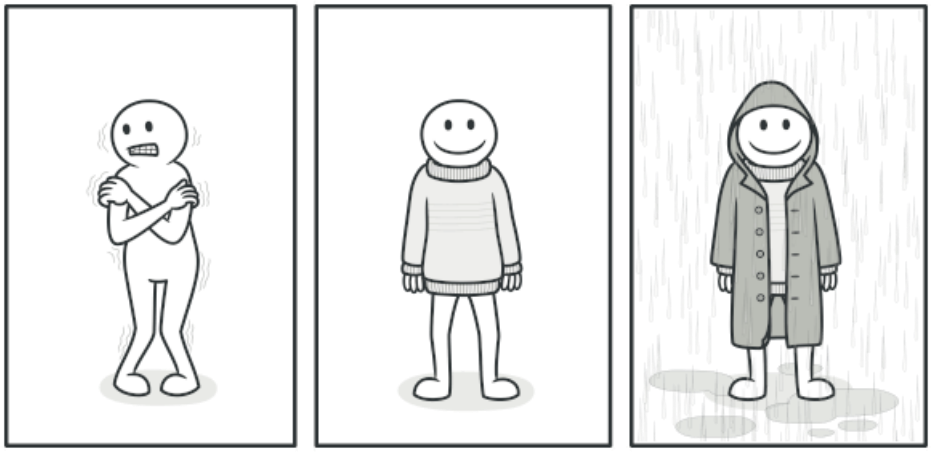


Figure 4 - You get a combined effect from wearing multiple pieces of clothing

Wearing clothes is an example of using decorators. When you're cold, you wrap yourself in a sweater. If you're still cold with a sweater, you can wear a jacket on top. If it's raining, you can put on a raincoat. All of these garments "extend" your basic behavior but aren't part of you, and you can easily take off any piece of clothing whenever you don't need it.

V. Structure

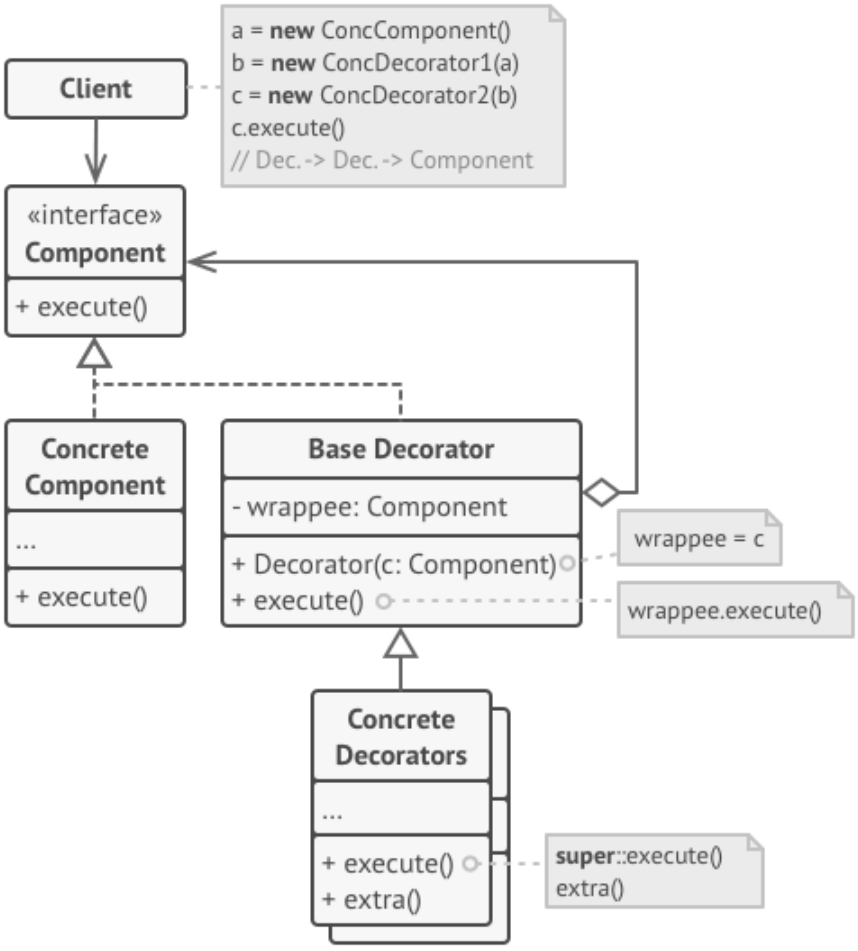


Figure 5 - Structure

Component declares the common interface for both wrappers and wrapped objects.
Concrete Component is a class of objects being wrapped. It defines the basic behavior, which can be altered by decorators.
Base Decorator class has a field for referencing a wrapped object. The field's type should be declared as the component interface so it can contain both concrete components and decorators. The base decorator delegates all operations to the wrapped object.
Concrete Decorators define extra behaviors that can be added to components dynamically. Concrete decorators override methods of the base decorator and execute their behavior either before or after calling the parent method.
Client can wrap components in multiple layers of decorators, as long as it works with all objects via the component interface.

VI. How to implement

- Make sure your business domain can be represented as a primary component with multiple optional layers over it.
- Figure out what methods are common to both the primary component and the optional layers. Create a component interface and declare those methods there.
- Create a concrete component class and define the base behavior in it.
- Create a base decorator class. It should have a field for storing a reference to a wrapped object. The field should be declared with the component interface type to allow linking to concrete components as well as decorators. The base decorator must delegate all work to the wrapped object.
- Make sure all classes implement the component interface.
- Create concrete decorators by extending them from the base decorator. A concrete decorator must execute its behavior before or after the call to the parent method (which always delegates to the wrapped object).
- The client code must be responsible for creating decorators and composing them in the way the client needs.

VII. Example implementation

Back to previous problem, we design a simple application as below:

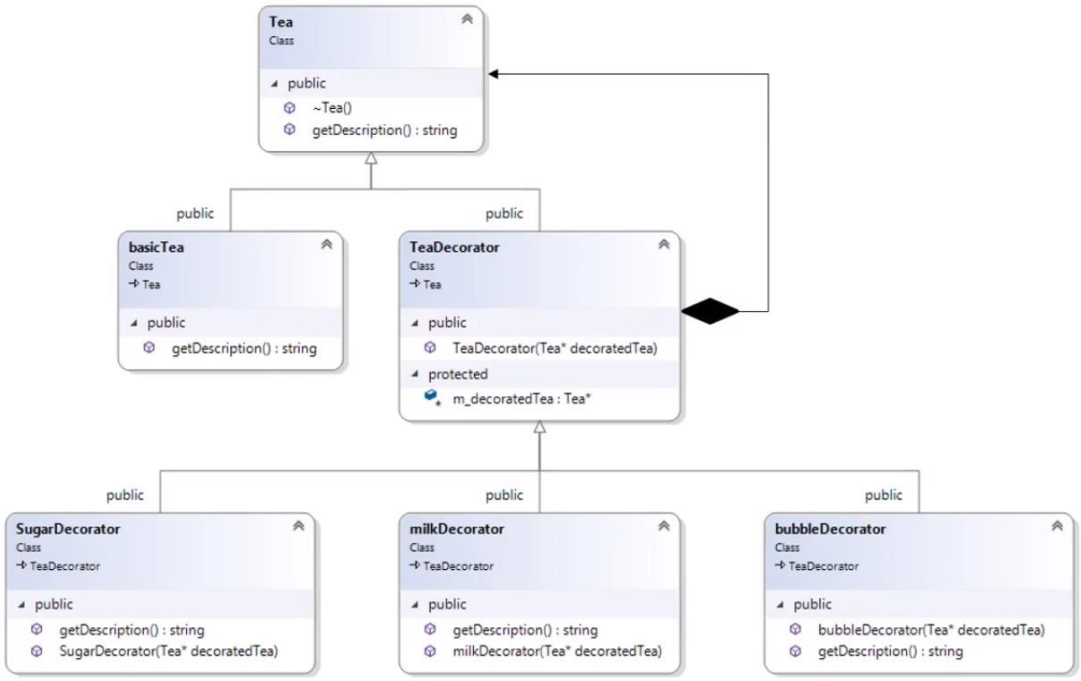


Figure 6 - Class diagram

Source code

```
class Tea{
    public:
        virtual string getDescription() = 0;
        virtual ~Tea(){};
};

class basicTea : public Tea {
    public:
        string getDescription(){
            return "basic Tea\n";
        }
};

class TeaDecorator : public Tea{
    protected:
        Tea* m_decoratedTea;
    public:
        TeaDecorator(Tea* decoratedTea) : m_decoratedTea(decoratedTea){}
};

class milkDecorator : public TeaDecorator{
    public:
        milkDecorator(Tea* decoratedTea) : TeaDecorator(decoratedTea){}
        string getDescription(){
            return m_decoratedTea->getDescription() + "add milk\n";
        }
};

class bubbleDecorator : public TeaDecorator{
    public:
        bubbleDecorator(Tea* decoratedTea) : TeaDecorator(decoratedTea){}
        string getDescription(){
            return m_decoratedTea->getDescription() + "add bubble\n";
        }
};

class sugarDecorator : public TeaDecorator{
    public:
        sugarDecorator(Tea* decoratedTea) : TeaDecorator(decoratedTea){}
        string getDescription(){
            return m_decoratedTea->getDescription() + "add sugar\n";
        }
};
```

```
int main(){                                                    Result:
    Tea* basic = new basicTea();                               basic Tea
    cout << basic->getDescription() << endl;
                                                               basic Tea
    Tea* bubble = new bubbleDecorator(new basicTea())          add bubble
;
    cout << bubble->getDescription() << endl;                  basic Tea
                                                               add sugar
    Tea* sugar = new sugarDecorator(new basicTea());
    cout << sugar->getDescription() << endl;                   basic Tea
                                                               add milk
    Tea* bubbleMilkTea = new bubbleDecorator(new milk          add bubble
Decorator(new basicTea()));
    cout << bubbleMilkTea->getDescription() << endl;
    return 0;
}
```

VIII. Pros and cons

- ✓ You can extend an object's behavior without making a new subclass.
- ✓ You can add or remove responsibilities from an object at runtime.
- ✓ You can combine several behaviors by wrapping an object into multiple decorators.
- ✓ Single Responsibility Principle. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes.
- × It's hard to remove a specific wrapper from the wrappers stack.
- × It's hard to implement a decorator in such a way that its behavior doesn't depend on the order in the decorators stack.
- × The initial configuration code of layers might look pretty ugly.

IX. Applicability

- Use the Decorator pattern when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.
- The Decorator lets you structure your business logic into layers, create a decorator for each layer and compose objects with various combinations of this logic at runtime. The client code can treat all these objects in the same way, since they all follow a common interface.
- Use the pattern when it's awkward or not possible to extend an object's behavior using inheritance.
- Many programming languages have the final keyword that can be used to prevent further extension of a class. For a FINAL class, the only way to reuse the existing behavior would be to wrap the class with your own wrapper, using the Decorator pattern.

LIST OF REFERENCE

[1] Alexander Shvets - Dive Into Design Patterns (2019)

[2] https://refactoring.guru/design-patterns/decorator

[3] https://techblog.vn/design-pattern-decorator-pattern