Skip to navigation

# Marcos Brizeno

## Desenvolvimento de Software #showmethecode

# Decorator

agosto 31, 2011

# Mão na massa: Decorator

Continuando com a série de post sobre padrões de projeto na prática, vamos falar agora sobre o padrão Decorator.

# ***Problema***

Imagine que você está desenvolvendo um sistema para um bar especializado em coquetéis, onde existem vários tipos de coquetéis que devem ser cadastrados para controlar a venda. Os coquetéis são feitos da combinação de uma bebida base e vários outros adicionais que compõe a bebida. Por exemplo:

Conjunto de bebidas:

- Cachaça
- Rum
- Vodka
- Tequila

Conjunto de adicionais:

- Limão
- Refrigerante
- Suco
- Leite condensado
- Gelo
- Açúcar

Então, como possíveis coquetéis temos:

- Vodka + Suco + Gelo + Açúcar
- Tequila + Limão + Sal
- Cachaça + Leite Condensado + Açúcar + Gelo

E então, como representar isto em um sistema computacional?

# Uma solução?

Bom, poderíamos utilizar como uma solução simples uma classe abstrata Coquetel extremamente genérica e, para cada tipo de coquetel construir uma classe concreta.

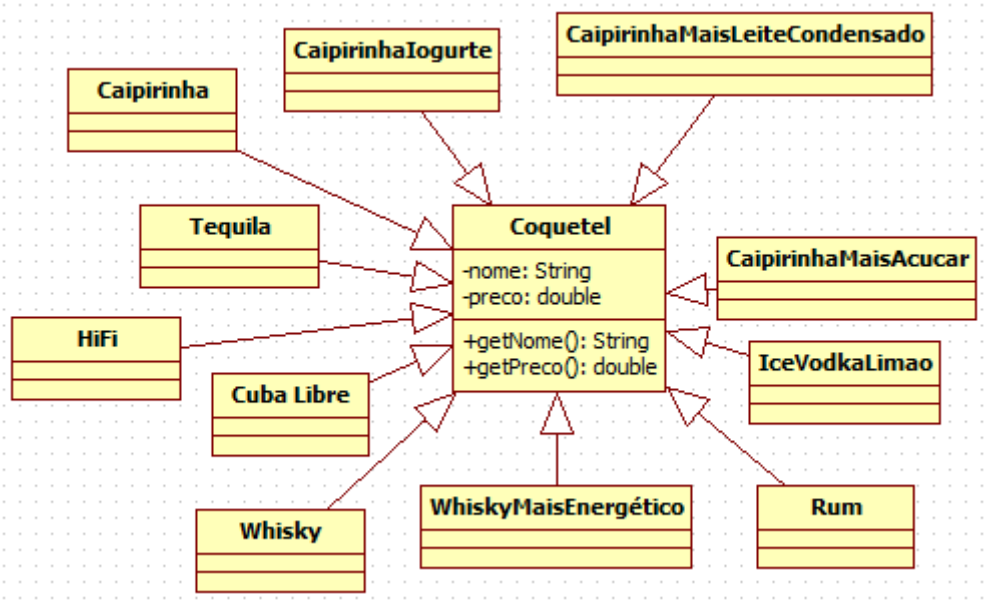
Então teríamos a classe base Coquetel:

```
1  public abstract class Coquetel {
2      String nome;
3      double preco;
4
5      public String getNome() {
6          return nome;
7      }
8
9      public double getPreco() {
10         return preco;
11     }
12 }
```

A nossa classe define apenas o nome e o preço da bebida para facilitar a exemplificação. Uma classe coquetel concreta seria, por exemplo, a Caipirinha:

```
1  public class Caipirinha extends Coquetel {
2      public Caipirinha() {
3          nome = "Caipirinha";
4          preco = 3.5;
5      }
6  }
```

No entanto, como a especialidade do bar são coquetéis, o cliente pode escolher montar seu próprio coquetel com os adicionais que ele quiser. De acordo com nosso modelo teríamos então que criar várias classes para prever o que um possível cliente solicitaria! Imagine agora a quantidade de combinações possíveis? Veja o diagrama UML abaixo para visualizar o tamanho do problema:



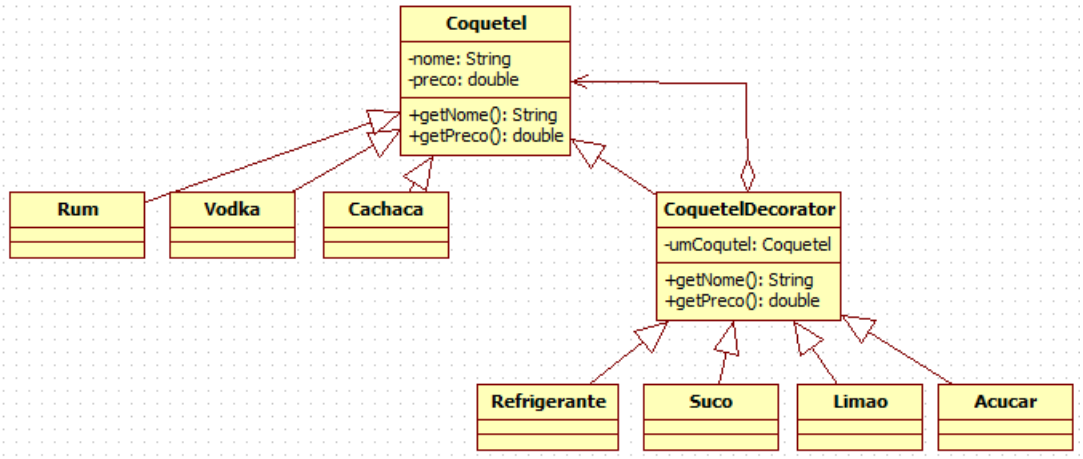

Além disso, pode ser que o cliente deseje adicionar doses extras de determinados adicionais, desse modo não seria possível modelar o sistema para prever todas as possibilidades! Então, como resolver o problema?

# ***Decorator***

Vamos ver qual a Intenção do padrão Decorator:

"Dinamicamente, agregar responsabilidades adicionais a objetos. Os Decorators fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades." [1]

Perfeito, exatamente a solução do nosso problema! Queremos que, dado um objeto Coquetel, seja possível adicionar funcionalidades a ele, e somente a ele, em tempo de execução. Vamos ver a arquitetura sugerida pelo padrão:




Certo, então todos os objetos possuem o mesmo tipo Coquetel, esta classe define o que todos os objeto possuem e é igual a classe já feita antes. As classes de bebidas concretas definem apenas os dados relativos a ela. Como exemplo vejamos o código da bebida Cachaça:

```
public class Cachaca extends Coquetel {
    public Cachaca() {
        nome = "Cachaça";
        preco = 1.5;
    }
}
```

Todas as classes de bebidas possuirão a mesma estrutura, apenas definem os seus atributos. A classe Decorator abstrata define que todos os decoradores possuem um objeto Coquetel, ao qual decoram, e um método que é aplicado a este objeto. Vejamos o código para exemplificar:

```
public abstract class CoquetelDecorator extends Coquetel {
    Coquetel coquetel;

    public CoquetelDecorator(Coquetel umCoquetel) {
        coquetel = umCoquetel;
    }

    @Override
    public String getNome() {
        return coquetel.getNome()  + " + " + nome;
    }

    public double getPreco() {
        return coquetel.getPreco() + preco;
    }
}
```

Lembre-se de que como o decorador também é um Coquetel ele herda os atributos nome e preço. Nas classes concretas apenas definimos os modificadores que serão aplicados, de maneira semelhante as classes de bebidas concretas, vejamos o exemplo do adicional Refrigerante:

```
public class Refrigerante extends CoquetelDecorator {

    public Refrigerante(Coquetel umCoquetel) {
        super(umCoquetel);
        nome = "Refrigerante";
        preco = 1.0;
    }

}
```

Perceba que no construtor do decorador é necessário passar um objeto Coquetel qualquer, este objeto pode ser tanto uma bebida quanto outro decorador. Ai está o conceito chave para o padrão Decorator. Vamos acrescentando vários decoradores em qualquer ordem em uma bebida. Vamos ver agora como o padrão seria utilizado, veja o seguinte código do método main:

```
public static void main(String[] args) {
        Coquetel meuCoquetel = new Cachaca();
        System.out.println(meuCoquetel.getNome() + " = "
                + meuCoquetel.getPreco());

        meuCoquetel = new Refrigerante(meuCoquetel);
        System.out.println(meuCoquetel.getNome() + " = "
                + meuCoquetel.getPreco());
    }
```

Perceba que o tipo do coquetel varia de acordo com o decorador aplicado. Então, quando chamamos o método getNome ou getPreco o primeiro método chamado é o método do último decorador aplicado.

O método do decorador por sua vez chama o método da classe mãe, este método então chama o método do Coquetel ao qual ele decora. Se esse coquetel for outro decorador o pedido é repassado até chegar a um coquetel que é uma bebida de fato e finalmente responde a requisição sem repassar a nenhum outro objeto.

De maneira semelhante a recursão, os valores calculados vão sendo retornados até chegar no último decorador aplicado e então são repassados ao objeto. É como se os decoradores englobassem tanto outros decoradores quanto o componente em si.

# Um pouco de teoria

Como já dito o padrão Decorator adiciona funcionalidades ao objeto em tempo de execução. Note bem que, ao contrário da herança que aplica funcionalidades a todos os objetos dela, o padrão decorator permite aplicar funcionalidades apenas a um objeto específico.

Justamente devido a essa propriedade é que o padrão Decorator possui uma flexibilidade maior que a herança estática. Além disso, como o Decorator aplica apenas as funcionalidades necessárias ao objeto nós evitamos o problema de classes sobrecarregadas, que possuem funcionalidade que nunca são utilizadas.

# Problemas com o Decorator

No entanto este comportamento altamente dinâmico do Decorator traz alguns problemas, como por exemplo, dado um objeto Coquetel não é possível verificar se ele possui um decorador Limão, Refrigerante ou qualquer outro. Assim, caso cada um dos decoradores implementasse outros métodos específicos, um desenvolvedor que utilizasse um coquetel qualquer não possui nenhuma garantia sobre o tipo do coquetel. Por exemplo, se o decorador Limão implementa um método arder(), não é possível afirmar que um coquetel qualquer possui este método.

O problema é pior ainda pois não é possível sequer verificar o tipo do coquetel, por exemplo, adicione este código no final do método main:

```
1  System.out.println(meuCoquetel instanceof Cachaca);
```

Será exibido no console “false” pois, como aplicamos o decorador Refrigerante modificamos o tipo do coquetel.

Além disso é necessário criar vários pequenos objetos, que possuem o mesmo comportamento, para criar os coquetéis necessários. No primeiro modelo apresentado teríamos várias classes, mas apenas um objeto para representar um Coquetel. Utilizando a estrutura do Decorator precisamos criar um objeto para cada novo decorador, além do objeto bebida.

## Código fonte completo

O código completo pode ser baixado no seguinte repositório Git: https://github.com/MarcosX/Padr-es-de-Projeto.

Os arquivos estão como um projeto do eclipse, então basta colocar no seu Workspace e fazer o import.

Se gostou do post compartilhe com seus amigos e colegas, senão, comente o que pode ser melhorado. Encontrou algum erro no código? Comente também. Possui alguma outra opinião ou alguma informação adicional? Comenta ai! 😀

## *Referências:*

[1] GAMMA, Erich et al. Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos.

 Decorator, Padrões de Projeto  Decorator, Java, Padrões, Projeto  10 Comentários

__PRESENT