UNIVERSIDADE DO MINHO
ESCOLA DE ENGENHARIA

# Arquiteturas Aplicacionais

## Design Patterns - Decorator

Ana Marta Santos Ribeiro A82474
Jéssica Andreia Fernandes Lemos A82061
Miguel José Dias Pereira A78912

**MIEI - 4º Ano - Arquiteturas Aplicacionais**
Braga, 9 de Março de 2020

# Conteúdo

**Conteúdo** **1**

**1 Introdução** **2**

**2 Decorator** **2**

**3 Estrutura** **2**

**4 Exemplo Prático** **3**

**5 Conclusão** **4**

**Bibliografia** **5**

# 1 Introdução

Este relatório é o resultado de uma pequena pesquisa sobre *design pattern* proposto no âmbito da unidade curricular Arquiteturas Aplicacionais do perfil de Engenharia de Aplicações. Neste optamos por selecionar o *pattern* estrutural *Decorator* dos padrões do livro Design Patterns: Elements of Reusable Object-Oriented Software.

Assim sendo, inicialmente será feita uma abordagem geral deste *pattern* passando de seguida para a apresentação da estrutura do *Decorator* e por fim apresentamos um pequeno protótipo.

# 2 Decorator

O *design pattern Decorator* pertence à classe de padrões *Structural*, e permite adicionar novos comportamentos ou funcionalidades a um objeto dinamicamente. Para tal, poderiamos utilizar herança para adicionar novos comportamentos em *runtime*, através de várias sub-classes contendo os comportamentos que queremos acrescentar, mas com o crescente número de sub-classes adicionadas tornar-se-ia uma solução complicada de gerir e manter, para além de que a herança não permite alterar o comportamento de um objeto dinamicamente.

Assim sendo, o *Decorator* permite acrescentar comportamento em *runtime*, recorrendo à composição ao invés de utilizar sub-classes, o que permite o uso do *Decorator* de modo recursivo, isto é, podemos ter um número ilimitado de comportamentos adicionais sobre o objeto original. O *Decorator* funciona então como um embrulho à volta de uma classe, que é transparente a quem utiliza a classe, pois este apenas encaminha os pedidos para a classe base com o comportamento extra acrescentada.

# 3 Estrutura

Para este *design pattern* existe uma estrutura base que se encontra representada na Figura 1, constituída por 4 classes, que devemos ter em consideração aquando da implementação de um *Decorator*.

A classe *Component* é uma interface dos objetos sobre os quais queremos ser capazes de adicionar comportamento, como tal, esta define as diferentes operações que devem ser implementadas em cada *ConcreteComponent* ou *Decorator*. Assim, o *Concre-*

*teComponent* representa o objeto que pode ter associado novas propriedades e a classe *Decorator* contêm uma referência para o objeto. Esta última utiliza composição em vez de herança e portanto agrupa o *Component* para definir a interface comum de cada *Decorator* para efetuar reencaminhamento de pedidos e de garantir as propriedades do *Component*. Por fim, o *ConcreteDecorator* é implementado para adicionar, modificar ou remover o comportamento de objetos durante o tempo de execução.

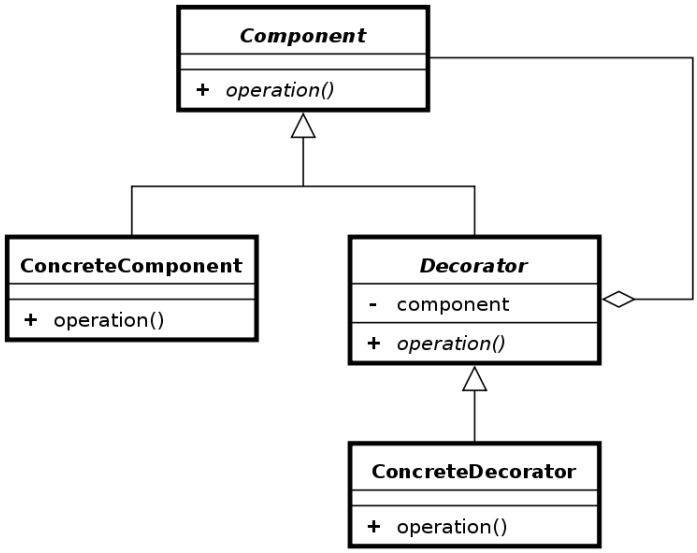


**Figura 1.** Diagrama UML do Decorator Pattern

Assim, temos acesso a um padrão flexível que permite adicionar funcionalidades de uma forma dinâmica em *runtime*. Através do uso da política *pay-as-you-go* evita-se os custos de propriedades que não iremos tirar partido. Contudo, existem desvantagens decorrentes do frequente uso do mesmo padrão no mesmo sistema uma vez que existem custos de gestão do *ConcreteDecorators*. Para além disso, devido à semelhança estrutural o *debug* da aplicação poderá tornar-se mais complicado.

# 4 Exemplo Prático

De forma a consolidar os conceitos explicados anteriormente implementamos um *Decorator* num pequeno exemplo. Assim optamos por recorrer a um cenário em que é necessário fazer um crepe. Basicamente consideramos que existe o crepe base de chocolcate com duas bolas de gelado e que tem um custo de 5 euros. Adicionalmente, pode

ser adicionado ao crepe tendo em conta a preferência do cliente novos ingredientes, nomeadamente chantilly e morangos no exemplo apresentado. É importante salientar que caso sejam acrecentados ingredientes o preço aumenta, tal como é possível verificar na imagem ilustrada de seguida.

```
O crepe contém chocolate, duas bolas de gelado e tem um custo de 5 euros.
O crepe contém chocolate, duas bolas de gelado, morangos e tem um custo de 7 euros.
O crepe contém chocolate, duas bolas de gelado, morangos, chantilly e tem um custo de 8 euros.
```

**Figura 2.** Exemplo prático

De modo a implementar este exemplo foi necessário desenvolver 6 classes, nomeadamente:

- **Crepe** – Interface que define as operações relativas à composição do crepe e ao seu preço e às quais podem ser adicionados novos comportamentos.
- **CrepeBase** – Contém as implementações das operações defaults para o crepe, isto é, a composição e o preço base do crepe sem adicionar extras.
- **CrepeDecorator** – É a classe do decorator que contém um atributo do tipo da interface, neste caso *Crepe*.
- **MorangosDecorator** - É uma classe concreta do decorator, que altera a composição e o preço do crepe, tendo em conta que foram adicionados morangos.
- **ChantillyDecorator** - Possui um comportamento semelhante à classe anterior.
- **FazerCrepe** - Esta classe implementa a main, onde primeiro é criado um crepe base, sendo posteriormente adicionados morangos e depois chantilly, como se pode observar na imagem apresentada anteriormente.

# 5 Conclusão

Com a pesquisa realizada ficamos a conhecer o *design pattern Decorator*, um padrão que tira partido da composição ao invés da herança, e permite adicionar comportamento a um objeto de forma flexivel em runtime, fazendo uso da política *pay-as-you-go* que evita custos desnecessários. Apesar das vantagens que este padrão apresenta, também temos de considerar os pontos negativo da sua utilização, tais como os custos de gestão e debug mais complicado.

# Bibliografia

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

[2] https://www.carloscaballero.io/design-patterns-decorator/

[3] https://sourcemaking.com/design_patterns/decorator