Prof. Me. Alexandre Henrick

Análise e Desenvolvimento de Sistemas - 6º período Sistemas de Informação

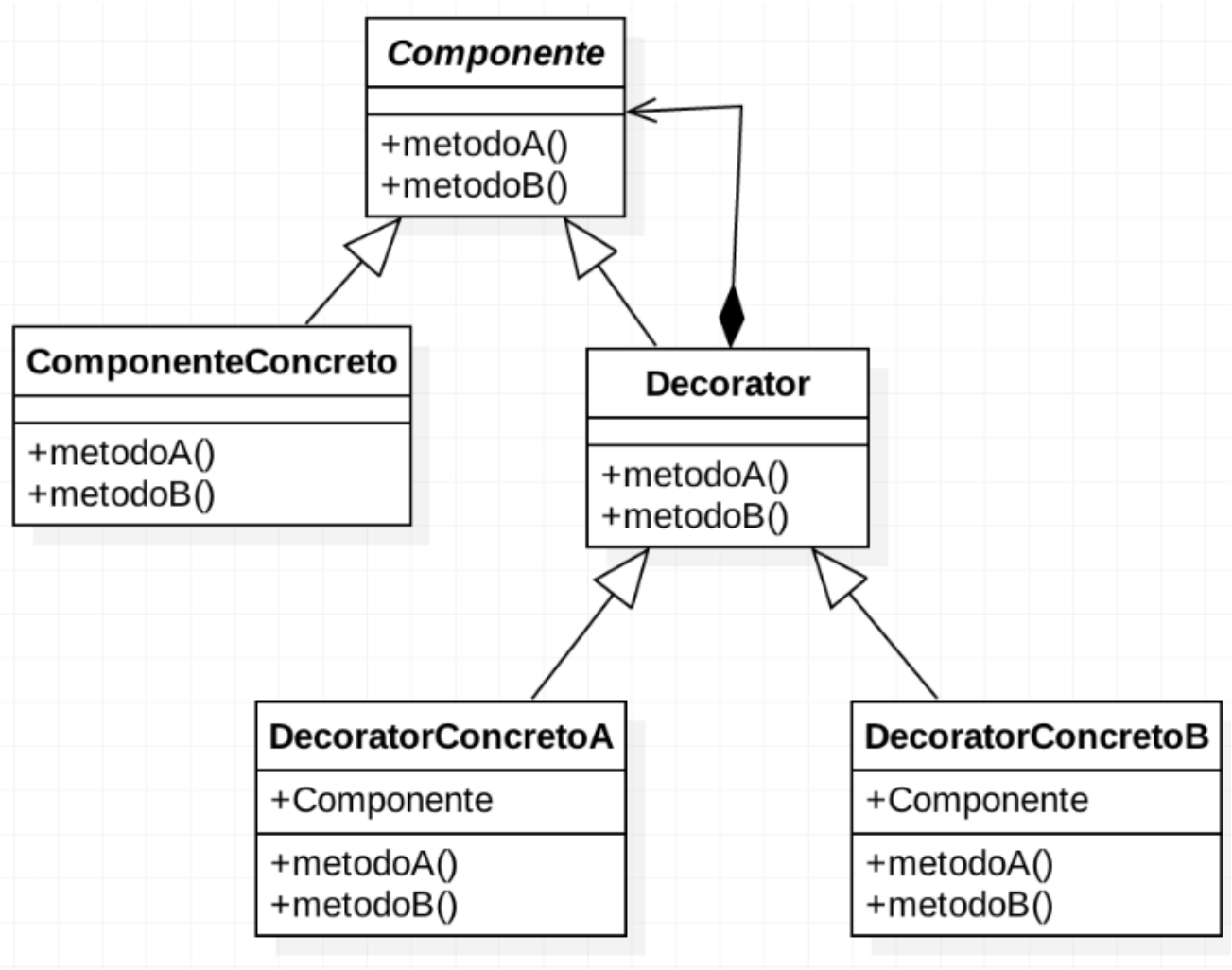
Decorator

Descrição

O **Decorator** é um padrão de projeto estrutural que permite adicionar de maneira dinâmica novos comportamentos em nossos objetos, sem alterar a estrutura de classes. Ele é usado quando você precisa estender as funcionalidades de objetos de maneira flexível e reutilizável, sem criar subclasses excessivas. **Ou seja, uma alternativa para estender as funcionalidades sem criar subclasses extras**.

Mas como isso é possível? O Decorator, assim como o Bridge, abre mão do uso de heranças para utilizar relacionamentos do tipo **agregação** ou **composição**. Como vimos no Bridge, o uso excessivo de heranças pode criar um cenário em que o número de classes filhas cresce exponencialmente. Portanto, se precisamos adicionar comportamentos adicionais aos nossos obejetos, fazer isso por meio de herança pode acarretar no mesmo problema. E essa é a principal motivação para a criação do Decoratos: **Estender as funcionalidades dos nossos objetos usando agregação/composição**.

Imagine que temos um objeto que executa uma funcionalidade. Queremos adicionar mais outra funcionalidade a ele. Isso significa que ele "**tem uma (composição)**" funcionalidade a mais agora. Veja que esse é relacionamento diferente de Herança ("**é um**").



Bridge X Decorator: O Bridge se preocupa em separar as "abstrações", permitindo o crescimento delas individualmente e fazendo uma "ponte" através de uma composição entre as abstrações. Essa ponto permite a comunicação entre os objetos dessas abstrações. O Decorator se preocupa em adicionar novas funcionalidades aos objetos por meio de composição ou agregação. Ambos seguem o mesmo princípio de não crescer utilizando herança.

Composite X Decorator: Ambos possuem o mesmo diagrama de classes, mas o propósito é diferente. No Composite queremos criar hierarquias, permitindo que objetos individuais e compostos sejam tratados de maneira uniforme. No Decorator queremos adicionar funcionalidades aos objetos de maneira flexível, sem herança, usando um "wrapper" para estender as capacidades.

Pontos positivos:

- Adicionar funcionalidades extras em tempo de maneira dinâmica/em tempo de execução;
- Incluir novos decoradores sem modificar as classes existentes;
- Reutilização: podemos criar decoradores que podem ser reutilzados em diferentes objetos.

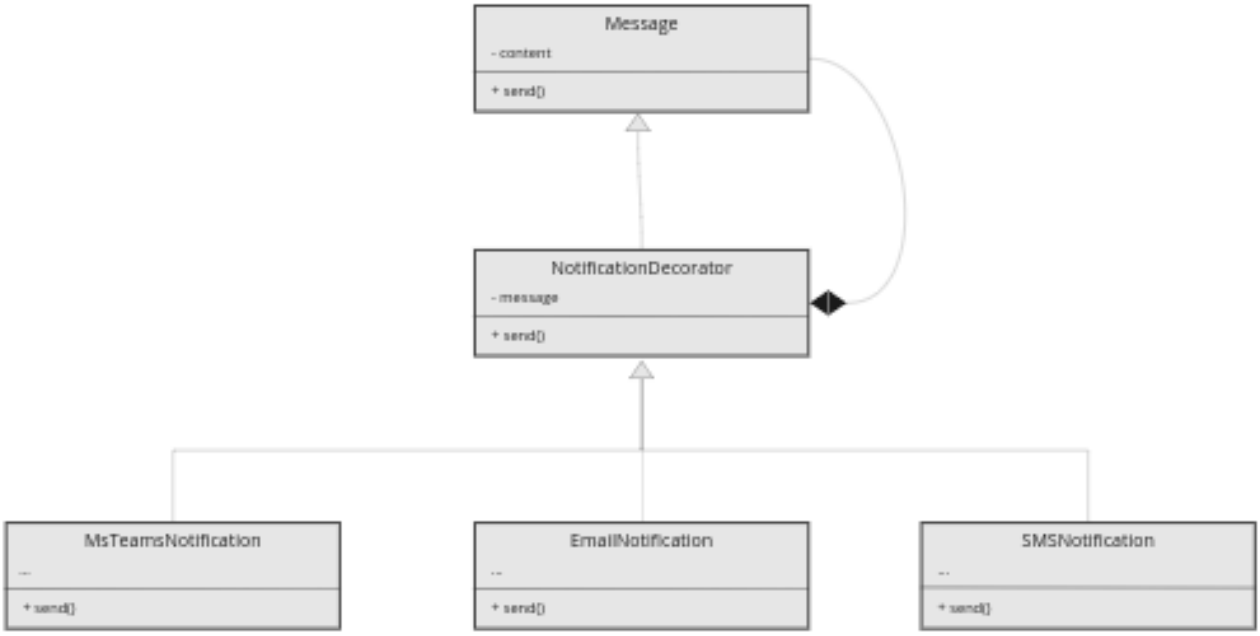
Pontos negativos:

- Muitos decoradores pode levar a uma complexidade adicional;
- Dificuldade de compreensão do comportamento do código quando temos a aplicação de múltiplos decoradores.

Cenário

Imagine que estamos construindo uma funcionalidade em um app que dispara notificações. Já temos uma funcionalidade que faz o disparo de uma notificação simples pelo app. Mas queremos que a mesma também seja capaz de fazer o disparo via email, SMS e MS Teams. Usando o decorator, podemos criar uma classe (**NotificationDecorator**) que irá conter um relacionamento do tipo composição com a classe (**Message**) que contém a funcionalidade de disoparo via app. Depois, basta criar as classes dos outros decoradores (Email, SMS, MsTeams) que vão herdar de NotificationDecorator.

Com isso, podemos fazer com que os objetos dessas classes deleguem as execuções para suas funcionalidades apropriadas. Se um objeto é passado como argumento para outro objeto decorador, o decorador irá adicionar sua funcionalidade ao objeto decorado. **O exemplo de código abaixo implementa esse cenário**.



Exemplo de código

```
# Componente base
class Message:
    def __init__(self, content):
        self._content = content

    def send(self):
        return self._content

# Decorador base
class NotificationDecorator(Message):
    def __init__(self, message):
        self._message = message

    def send(self):
        return self._message.send()

# Decoradores concretos
class EmailNotification(NotificationDecorator):
    def send(self):
```

```
        email_content = f"{self._message.send()}" + "\nenviado por email"
        return email_content

class SMSNotification(NotificationDecorator):
    def send(self):
        sms_content = f"{self._message.send()}" + "\nenviado por SMS"
        return sms_content

class MsTeamsNotification(NotificationDecorator):
    def send(self):
        push_content = f"{self._message.send()}" + "\nenviado por Ms Teams"
        return push_content

# Uso
basic_message = Message("Hello, world!")

email_notification = EmailNotification(basic_message) # Decorando
componente mais simples
sms_notification = SMSNotification(email_notification)
teams_notification = MsTeamsNotification(sms_notification) # Decorando
objeto que também é um decorador

#print(basic_message.send())
#print(email_notification.send())
#print(sms_notification.send())
print(teams_notification.send())
```