Padrão Decorator

Padrão Estrutural

I. Problema: Decorador de codificação e compactação

O presente cenário de problema ilustra como podemos ajustar o comportamento de um objeto sem alterar seu código, independentemente de quem verdadeiramente usa esses dados. Inicialmente, consideremos a classe pertencente à lógica de negócio **FileDataSource** cuja implementação está pronta e tem a responsabilidade de ler - `readData()` - e gravar dados textuais - `writeData()` - em um arquivo externo, sem formatação. O **FileDataSource** é o componente concreto que será envolvido pelo decorador.

Porém, novos requisitos foram incorporados à aplicação e agora há a necessidade de criar duas classes *wrapper* para adicionar novos comportamentos após a execução das operações padrão do objeto empacotado (**FileDataSource**).

O primeiro *wrapper* criptografa e descriptografa dados (**EncryptionDecorator**), enquanto o segundo compacta e descompacta dados (**CompressionDecorator**). É possível combinar esses *wrappers* envolvendo um decorador com outro. A classe **DataSource** atua como Decorador Base, logo, terá seu comportamento herdado pelos decoradores concretos **EncryptionDecorator** e **CompressionDecorator.** A interface **DataSource** atual como o Componente do padrão, declarando os métodos em comum tanto para os *wrappers* quanto para os objetos envolvidos. A Figura 1 ilustra o diagrama de classes do modelo.

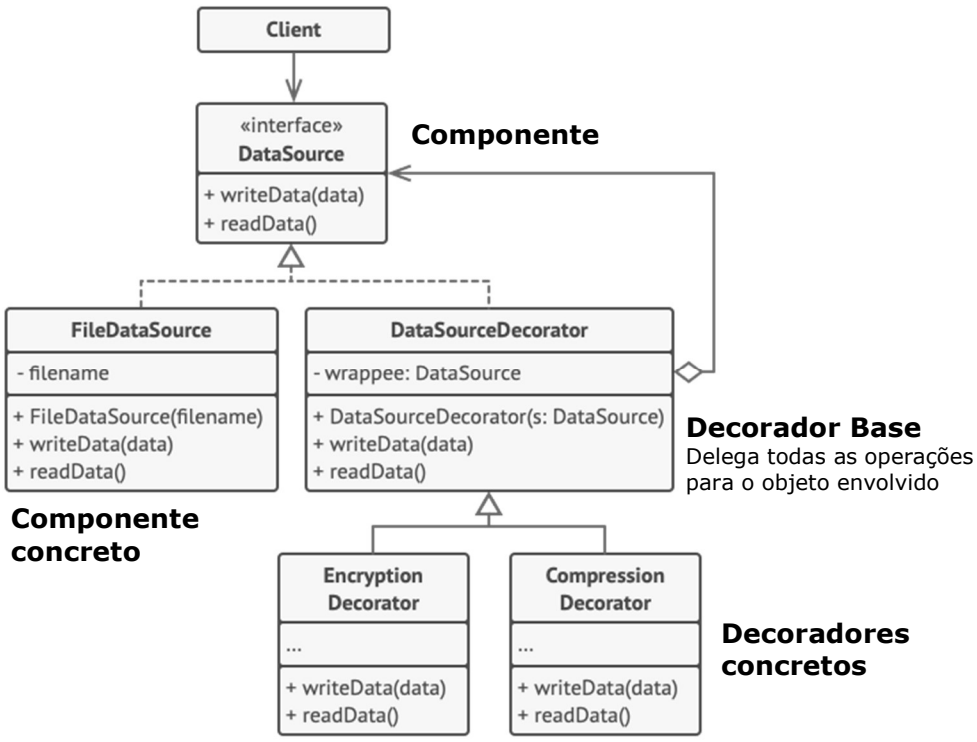

Figura 1. Decoradores para compressão e encriptação de dados

A aplicação envolve o objeto da fonte de dados (**FileDataSource**) com um par de decoradores. Ambos os invólucros mudam a maneira com que os dados são escritos e lidos no disco, conforme discriminado a seguir:

- Antes dos dados serem **escritos no disco**, os decoradores realizam a tarefa de *encriptação* e *compressão* dos dados. A classe original (**FileDataSource**) escreve os dados protegidos e encriptados para o arquivo sem saber da mudança, pois passaram previamente pelos decoradores. A classe **EncryptionDecorator** define seus métodos particulares **`encode`**`()` e **`decode`**`()` para realizarem as tarefas de criptografar e descriptografar dados, usando o esquema de codificação binário para texto **Base64**[1]. Não faz parte do escopo deste exercício entender o nível de detalhamento de tal esquema de codificação. Portanto, ao usar o **EncryptionDecorator**, o objeto **FildeDataSource** lê ou grava o texto criptografado sem preceber.

- Da mesma forma, o decorador **CompressionDecorator** realiza a tarefa de compressão e descompressão de dados por meio dos seus métodos **`compress`**`()` e **`decompress`**`()`. Esses métodos são invocados, respectivamente, na implementação sobrescrita dos métodos **`readData`**`()` e **`writeData`**`()`.

- Logo que os dados são **lidos do disco** pelo objeto **FileDataSource**, ele passa pelos mesmos decoradores que descomprimem e decodificam eles.

Os **decoradores** e a classe da **fonte de dados** implementam a mesma interface, o que os torna intercomunicáveis dentro do código cliente.

A classe cliente cria os decoradores e os compõem em camadas da forma que necessita. Neste exercício, a criação dos decoradores é aninhada de tal forma que o **FileDataSource** é envolvido pelo **EncryptionDecorator,** e este é envolvido pelo **CompressionDecorator.**

```
DataSourceDecorator encoded = new CompressionDecorator(
                                  new EncryptionDecorator(
                                  new
                                  FileDataSource("OutputDemo.txt")));
```

II. A tarefa

A partir do diagrama disposto na Figura 1, analise e discuta a aplicação do padrão **decorator** para o problema em questão.

[1] Mais informações em https://en.wikipedia.org/wiki/Base64