Decorator

O padrão decorator permite adicionar novos comportamentos a um objeto sem alterar sua classe original a sua descrição original do Padrão Decorator é: "O Padrão Decorator anexa responsabilidades adicionais a um objeto dinamicamente.

Ele possue as seguintes características:

* Eles tem o mesmo tipo supertipo dos objetos que eles decoram;
* Você pode usar um ou mais decoradores para englobar um objeto;
* Uma vez que o decorador tem o mesmo supertipo que o objeto decorado, podemos passar um objeto decorado no lugar do objeto original (englobado);
* O decorador adiciona seu próprio comportamento antes e/ou depois de delegar o objeto que ele decora o resto do trabalho;
* Os objetos podem ser decorados a qualquer momento, então podemos decorar os objetos de maneira dinâmica no tempo de execução com quantos decoradores desejarmos.

Vantagens

* Adiciona funcionalidades sem alterar o código original
* Você pode combinar vários comportamentos envolvendo um objeto em vários decoradores.
* Fácil de combinar vários comportamentos

### Desvantagens

* Pode aumentar a complexidade (muitas classes pequenas)
* Pode dificultar a depuração (debug)
* O código de configuração das camadas pode ser muito desorganizado (feio)

### Conclusão

### O padrão Decorator é ideal para ser utilizados quando precisarmos adicionar funções como por exemplo logging, compressão, criptografia e outras sem alterar o código original do programa.

Referencias

1. [Padrão de Projeto Decorator em Java - DevMedia](https://www.devmedia.com.br/padrao-de-projeto-decorator-em-java/26238)

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

Chain of Responsibility

O chain of responsibility ou também chamado de cadeia de responsabilidade e um padrão que permite passar uma solicitação por uma cadeia de objetos até que um deles resolva trata-la. Você deve utiliza-lo quando é esperado que seu programa processe diferentes tipos de pedidos de varias maneiras diferentes, mas os exatos tipos de pedidos e suas sequencias são desconhecidas de

**Vantagens**

* Reduz o **acoplamento** entre remetente e receptor.
* Facilita a **extensão**
* Segue o princípio **Open/Closed** (aberto para extensão, fechado para modificação).
* Você pode controlar a ordem de tratamento dos pedidos.

**Desvantagens**

* E agora que vimos o quão interessante é Chain of Responsibility, vamos ter em mente algumas desvantagens:
* Principalmente, pode ser quebrado facilmente:
* Se um processador não chamar o próximo processador, o comando será descartado
* Se um processador chamar o processador errado, isso pode levar a um ciclo
* Ele pode criar rastreamentos de pilha profundos, o que pode afetar o desempenho
* Isso pode levar a código duplicado entre processadores, aumentando a manutenção

Conclusão

O Chain of Responsibility permite que uma solicitação tenha a chance de ser tratado por diversos objetos isso permite criar processos desacoplados ele e ideal para situações em que varias classes podem tratar uma mesma solicitação.

Referencias

[Chain of Responsibility](https://refactoring.guru/pt-br/design-patterns/chain-of-responsibility)  
[Padrão de Design de Cadeia de Responsabilidade em Java | Baeldung](https://www.baeldung.com/chain-of-responsibility-pattern)

State

O State ou estado permite que um objeto altere o seu comportamento quando o seu estado interno muda, ele faz isso encapsulando os comportamentos relacionados a cada estado em classes separadas

Vantagens

* Simplifica o código de contexto ao eliminar condicionais de máquinas de estado pesadas.
* Facilita a manutenção e extensão de novos estados.
* Cada estado é independente e coeso.
* Segue o princípio Open/Closed (aberto para extensão, fechado para modificação).

Desvantagens

* Pode aumentar o número de classes no projeto.
* A transição de estados precisa ser bem controlada.
* Aplicar o padrão pode ser um exagero se a máquina de estado só tem alguns estados ou raramente muda eles.

Conclusão

Ele é ótimo quando queremos evitar instruções de if e else, pois em vez disso da para extrairmos

O padrão de design de estado é ótimo quando queremos **evitar instruções if/else primitivas**. Em vez disso, **extraímos a lógica para separar as classes** e deixamos nosso **objeto de contexto delegar o comportamento** aos métodos implementados na classe de estado. Além disso, podemos alavancar as transições entre os estados, onde um estado pode alterar o estado do contexto.

Em geral, esse padrão de design é ótimo para aplicações relativamente simples, mas para uma abordagem mais avançada, podemos dar uma olhada no [tutorial State Machine do Spring](https://www.baeldung.com/spring-state-machine).

Referencias

[State](https://refactoring.guru/pt-br/design-patterns/state)

[Padrão de design de estado em Java | Baeldung](https://www.baeldung.com/java-state-design-pattern)