Seminário - Design Pattern - Técnicas de Programação 2

Chain of Reponsibility

Design Pattern

Collaborators

Renan Marques

- Criação da apresentação
- Estudo sobre o conceito do tema

Wilson

- Criação do mini-cenário prático
- Estudo sobre o conceito do tema

Willian Paulino

- Criação da apresentação
- Estudo sobre o conceito do tema

Introduction Pattern

Chain of Reponsibility ↴

É um Design Pattern comportamental que organiza uma sequência de objetos capazes de tratar uma solicitação. Cada objeto da cadeia recebe o pedido e decide se o processa ou se o encaminha para o próximo. Isso elimina estruturas rígidas de condicionais encadeadas, reduz o acoplamento entre classes e permite adicionar ou modificar responsabilidades sem alterar o restante do sistema.

Contextualização Histórica

Esse padrão surgiu para resolver situações em que vários objetos podem tratar uma mesma requisição. Em vez de criar condicionais rígidas (if/else repetidos), a responsabilidade é passada pela cadeia, seguindo uma ordem definida.

Intenção do Padrão

A intenção do Chain of Responsibility é permitir que uma solicitação seja passada por uma cadeia de objetos até que um deles consiga tratá-la. Assim, quem faz o pedido não precisa saber quem vai resolver, deixando o código mais flexível e sem depender de verificações ou condições repetidas.

Problemas que resolve

- Evita estruturas complexas de if/else.
- Reduz acoplamento entre classes.
- Facilita substituições, extensões e reuso.
- Permite adicionar novos “manipuladores” sem alterar o restante do código.

Analogia do Mundo Real

Imagine um sistema de suporte ao cliente onde as solicitações dos clientes precisam ser atendidas com base em sua prioridade. Existem três níveis de suporte: Nível 1, Nível 2 e Nível 3. O suporte de Nível 1 lida com solicitações básicas, o suporte de Nível 2 lida com solicitações mais complexas, e o suporte de Nível 3 lida com questões críticas que não podem ser resolvidas pelo Nível 1 ou Nível 2.

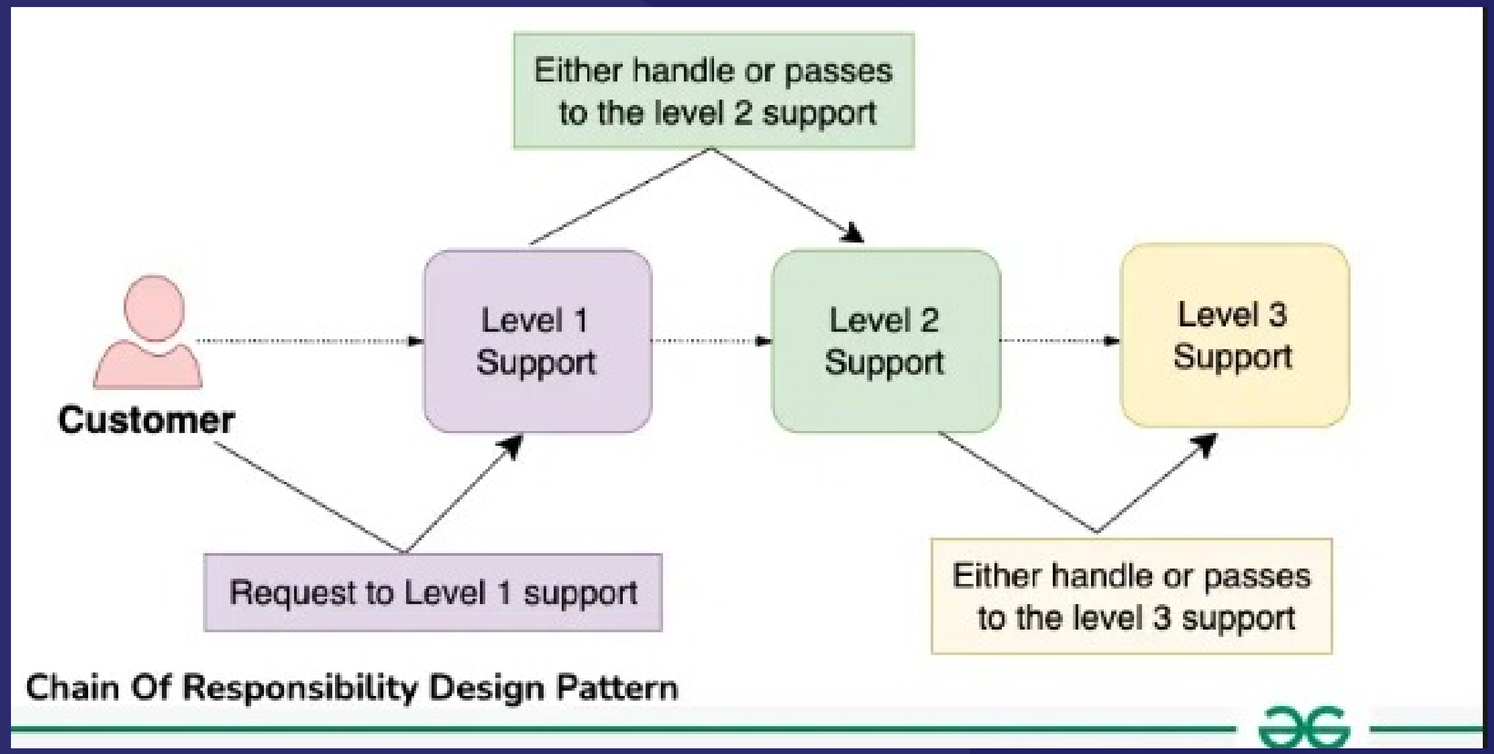
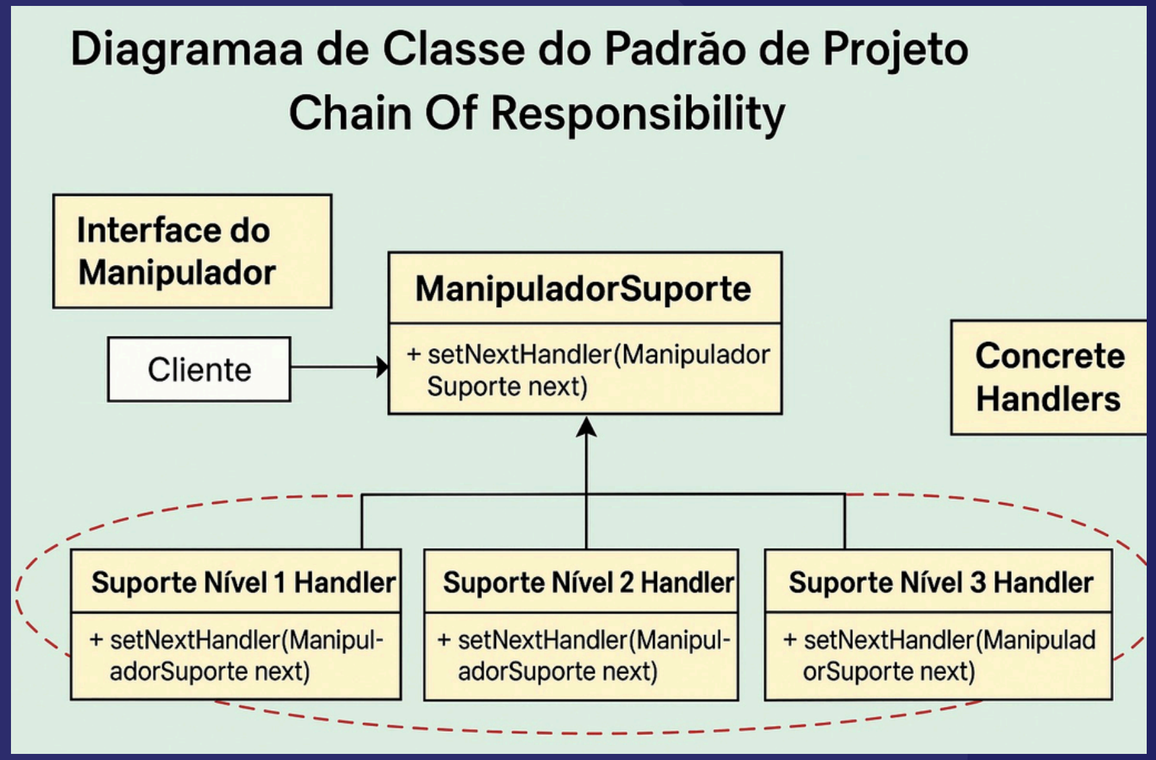
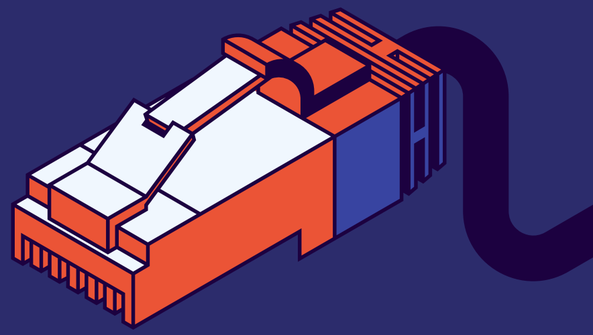


Diagrama de Classe do Padrão Chain of Responsibility

O diagrama representa o Chain of Responsibility, mostrando como uma requisição percorre uma cadeia de manipuladores. O cliente envia a requisição para o primeiro manipulador, que decide se consegue tratá-lo; caso não consiga, ele encaminha para o próximo nível. Cada classe concreta implementa essa lógica, tratando apenas o que é de sua responsabilidade. Esse padrão reduz o acoplamento, organiza o fluxo de atendimento e permite adicionar ou alterar níveis de suporte sem modificar o cliente.





Main components

Client

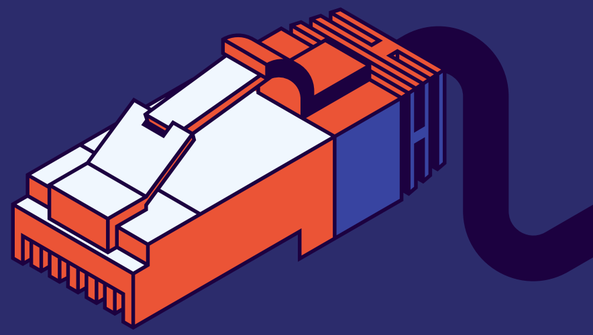
Configura a cadeia de handlers e envia a requisição inicial. Não sabe qual handler irá realmente tratar o pedido.

Handler Interface / Abstract Handler

Define o método de processamento e guarda o próximo handler na cadeia. Serve como estrutura base para todos os handlers.

Concrete Handlers

Implementam o processamento real. Cada um tenta resolver a requisição; se não puder, repassa para o próximo handler.



Relationship Between Classes

Herança:

Os Concrete Handlers herdam da classe base Handler,
que define a interface comum de tratamento.

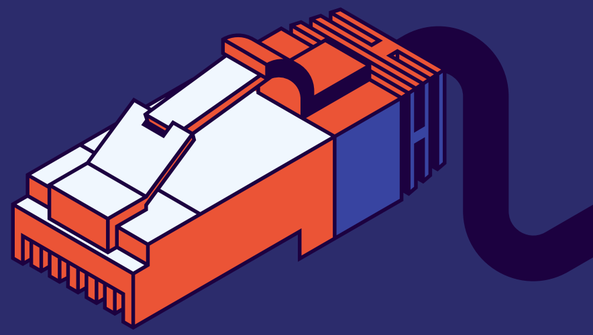
Associação (encadeamento):

A classe Handler mantém uma referência para o próximo handler da cadeia (nextHandler). Isso permite que cada objeto encaminhe a requisição adiante caso não possa tratá-la.

Dependência do Cliente:

O Cliente depende apenas do primeiro handler da cadeia, sem precisar conhecer os demais.
Com isso, o cliente fica desacoplado dos possíveis manipuladores.

Essa estrutura cria um fluxo flexível, onde cada handler pode tratar, parcialmente tratar ou delegar a requisição para o próximo objeto da cadeia.



Pattern Variations

1. **Cadeia Estática**

   Ordem fixa de handlers; simples, porém pouco flexível.

2. **Cadeia Implícita**

   Montada automaticamente por framework (reflexão/anotações).

3. **Cadeia Dinâmica**

   Ordem montada em tempo de execução; permite adicionar/remover handlers.

4. **Interrupção Condicional**

   Cada handler decide se continua ou para a cadeia.

5. **Múltiplos Finais (Menos comum)**

   A cadeia pode ramificar e terminar em pontos diferentes.

6. **Cadeia Bidirecional (Menos comum)**

   Fluxo pode ir e voltar entre handlers (raro, mais complexo).

Advantages

Desacoplamento de Objetos:

Reduz a dependência entre remetente e manipuladores. O objeto que envia a requisição não precisa saber quem vai tratá-la.

Flexibilidade e Extensibilidade:

Permite adicionar ou modificar handlers facilmente, sem alterar o código do cliente.

Ordem Dinâmica de Tratamento:

A ordem dos handlers pode mudar em tempo de execução, ajustando o fluxo conforme a necessidade.

Interações Simplificadas com Objetos:

O remetente não precisa conhecer a lógica interna do processamento, tornando a comunicação mais simples.

Mantibilidade Aprimorada:

Cada handler tem uma responsabilidade específica, facilitando alterações e manutenção sem impactar todo o sistema.

Disadvantages

Complexidade na Depuração:

Com vários manipuladores, fica mais difícil acompanhar onde a solicitação está e identificar problemas.

Possíveis Solicitações Não Tratadas:

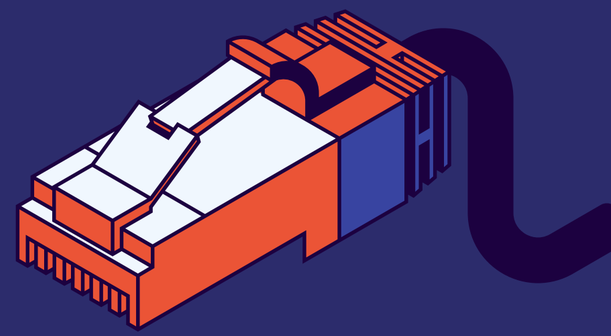
Se a cadeia não for configurada corretamente, algumas solicitações podem não ser atendidas.

Overhead de Configuração em Tempo de Execução:

Gerenciar ou modificar a cadeia dinamicamente pode ser complexo e exigir mais esforço.

Overhead de desempenho:

A requisição pode passar por vários handlers, aumentando o tempo de processamento.



Quando utilizar

É indicado quando diversas etapas podem ou não atuar sobre a mesma requisição e quando o sistema precisa permitir mudanças rápidas na ordem ou na quantidade de handlers. Funciona muito bem em sistemas que utilizam middleware ou processos sequenciais de validação.

Quando não utilizar

Não é uma boa escolha quando apenas um componente deve tomar a decisão final, ou quando o processo precisa seguir uma ordem totalmente fixa e previsível, sem flexibilidade. Também não vale a pena em fluxos simples com poucas etapas.

Comparação com outros patterns

- Chain vs Command: Command encapsula uma ação, enquanto Chain distribui a requisição entre vários possíveis processadores.

- Chain vs Pipeline: Ambos funcionam em etapas, mas o Chain é orientado a objetos e segue handlers encadeados; o pipeline costuma ser mais funcional e direto.

Let's Go on to the Case Demonstration.

Verifique o Git Hub

Principais aprendizados...

- O padrão organiza objetos em uma cadeia capaz de processar requisições de forma sequencial.
- Reduz o acoplamento entre o cliente e os manipuladores, permitindo maior flexibilidade.
- Cada handler pode tratar a requisição ou delegar para o próximo, tornando o fluxo de responsabilidade mais claro.
- A cadeia pode ser fixa, dinâmica ou até montada automaticamente, conforme a necessidade do sistema.
- Facilita manutenção, extensão e testes, já que novos handlers podem ser adicionados sem alterar os existentes.

Relevância do pattern

O Chain of Responsibility é essencial em sistemas que lidam com múltiplas regras, validações, filtros ou etapas de processamento. Ele melhora a organização do código, promove baixo acoplamento, facilita evolução do software e favorece arquiteturas modulares. Sua flexibilidade torna o padrão amplamente utilizado em APIs, middlewares, validações, processamento de eventos e pipelines de responsabilidade.

Perguntas ?

Muito obrigado pelo seu tempo, atenção e interesse em nossa apresentação!