refactoring.guru

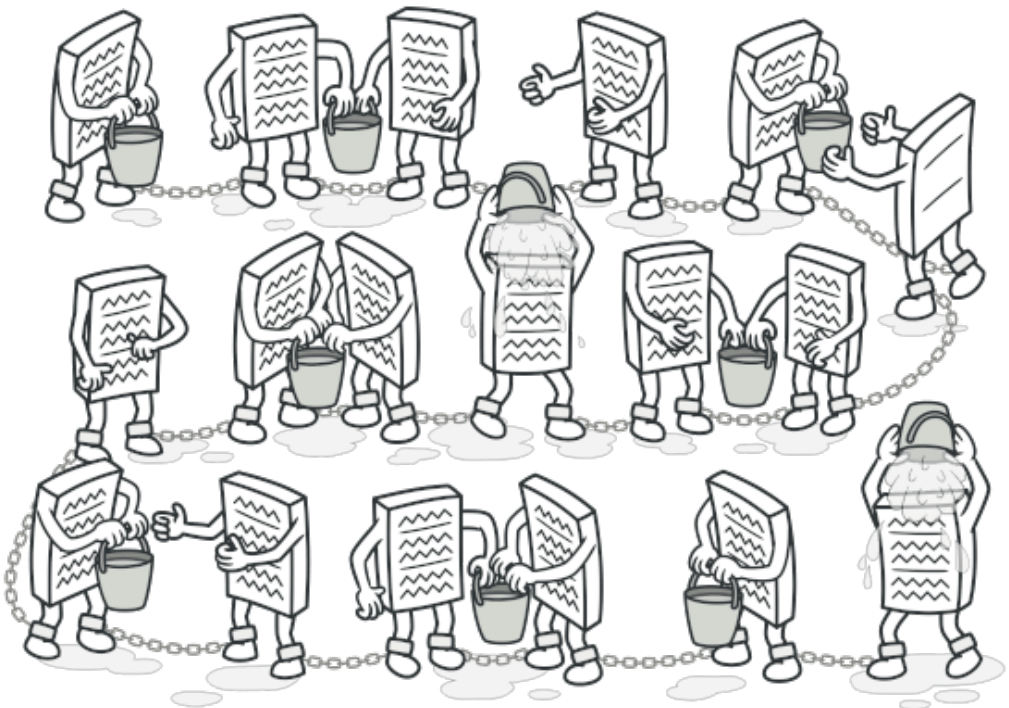
Chain of Responsibility

16–22 minutes

Também conhecido como: CoR, Corrente de responsabilidade, Corrente de comando, Chain of command

Propósito

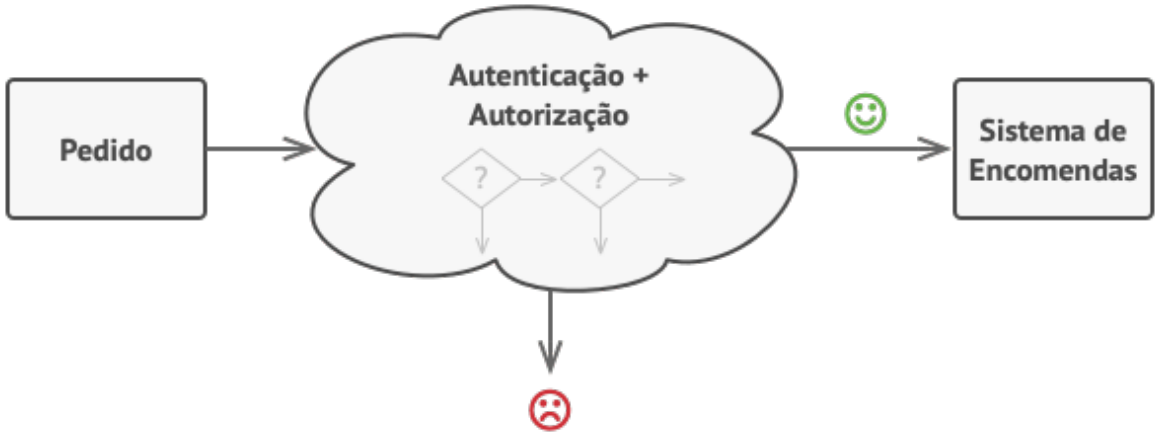
O **Chain of Responsibility** é um padrão de projeto comportamental que permite que você passe pedidos por uma corrente de handlers. Ao receber um pedido, cada handler decide se processa o pedido ou o passa adiante para o próximo handler na corrente.



Problema

Imagine que você está trabalhando em um sistema de encomendas online. Você quer restringir o acesso ao sistema para que apenas usuários autenticados possam criar pedidos. E também somente usuários que tem permissões administrativas devem ter acesso total a todos os pedidos.

Após um pouco de planejamento, você se dá conta que essas checagens devem ser feitas sequencialmente. A aplicação pode tentar autenticar um usuário ao sistema sempre que receber um pedido que contém as credenciais do usuário. Contudo, se essas credenciais não estão corretas e a autenticação falha, não há razão para continuar com outras checagens.

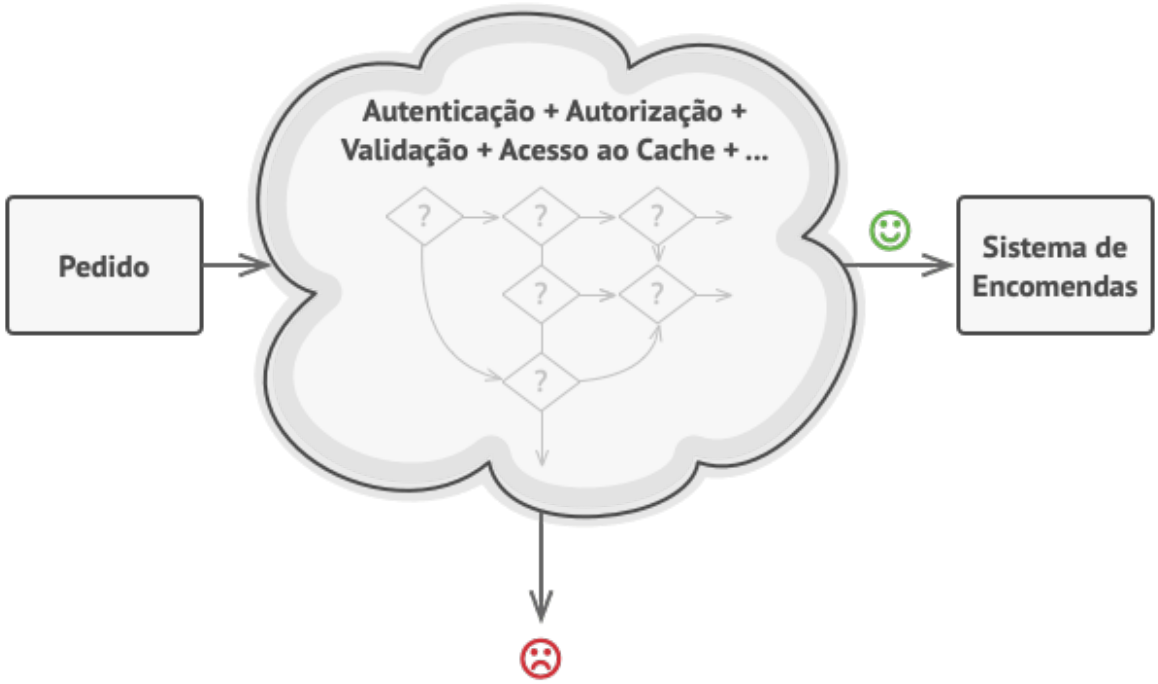


O pedido deve passar por uma série de checagens antes do sistema de encomendas possa lidar ele mesmo com o pedido.

Durante os próximos meses você implementou diversas mais daquelas checagens sequenciais.

- Um de seus colegas sugeriu que não é seguro passar dados brutos diretamente para o sistema de encomendas. Então você adicionou uma etapa adicional de validação para limpar os dados no pedido.

- Mais tarde, alguém notou que o sistema é vulnerável à ataques de força bruta. Para evitar isso, você prontamente adicionou uma checagem que filtra repetidas falhas vindas do mesmo endereço de IP.
- Outra pessoa sugeriu que você poderia agilizar o sistema se retornasse resultados de cache em pedidos repetidos contendo os mesmos dados. Portanto, você adicionou outra checagem que permite que o pedido passe através do sistema apenas se não há uma resposta adequada armazenada em cache.



Quanto mais o código cresce, mais bagunçado ele fica.

O código das checagens, que já parecia uma bagunça, ficou mais e mais inchado a medida que você foi adicionando novas funcionalidades. Mudar uma checagem às vezes afetava outras. E o pior de tudo, quando você tentou reutilizar as checagens para proteger os componentes do sistema, você teve que duplicar parte do código uma vez que esses componentes precisavam de algumas dessas checagens, mas nem todos eles.

O sistema ficou muito difícil de compreender e caro de se manter. Você lutou com o código por um tempo, até que um dia você decidiu refatorar a coisa toda.

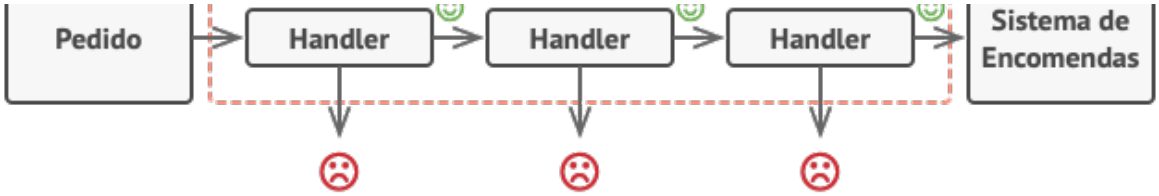
Solução

Como muitos outros padrões de projeto comportamental, o **Chain of Responsibility** se baseia em transformar certos comportamentos em objetos solitários chamados *handlers*. No nosso caso, cada checagem devem ser extraída para sua própria classe com um único método que faz a checagem. O pedido, junto com seus dados, é passado para esse método como um argumento.

O padrão sugere que você ligue esses handlers em uma corrente. Cada handler ligado tem um campo para armazenar uma referência ao próximo handler da corrente. Além de processar o pedido, handlers o passam adiante na corrente. O pedido viaja através da corrente até que todos os handlers tiveram uma chance de processá-lo.

E aqui está a melhor parte: um handler pode decidir não passar o pedido adiante na corrente e efetivamente parar qualquer futuro processamento.

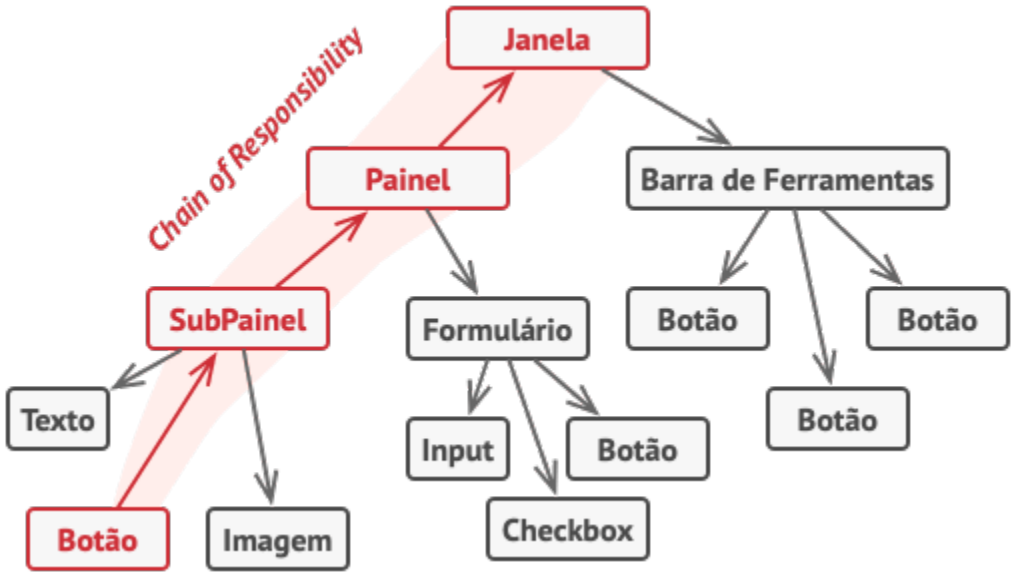
Em nosso exemplo com sistema de encomendas, um handler realiza o processamento e então decide se passa o pedido adiante na corrente ou não. Assumindo que o pedido contenha os dados adequados, todos os handlers podem executar seu comportamento principal, seja ele uma checagem de autenticação ou armazenamento em cache.



Handlers estão alinhados um a um, formando uma corrente.

Contudo, há uma abordagem ligeiramente diferente (e um tanto quanto canônica) na qual, ao receber o pedido, um handler decide se ele pode processá-lo ou não. Se ele pode, ele não passa o pedido adiante. Então é um handler que processa o pedido ou mais ninguém. Essa abordagem é muito comum quando lidando com eventos em pilha de elementos dentro de uma interface gráfica de usuário.

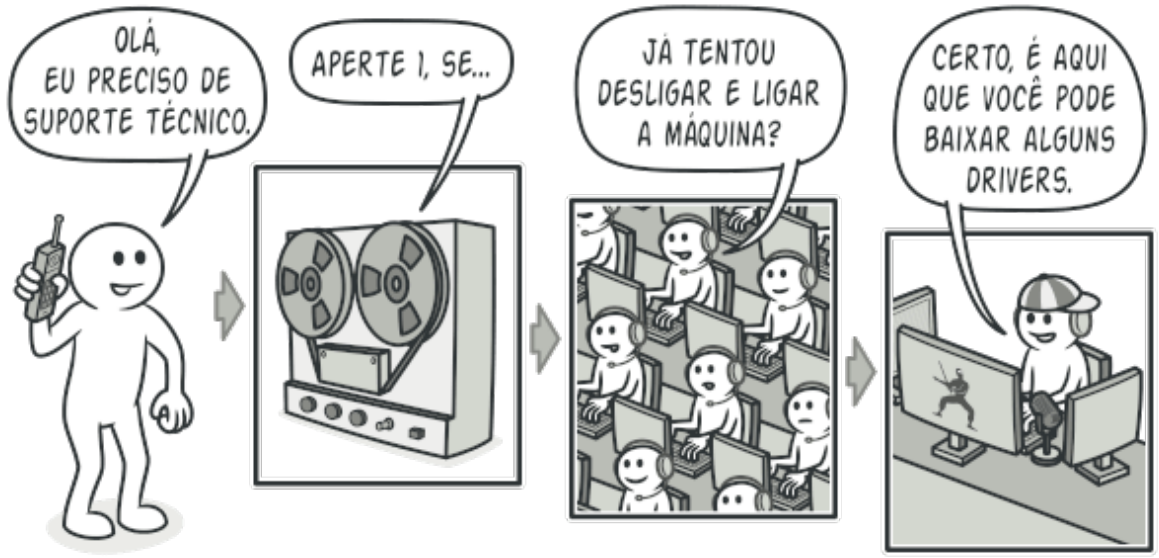
Por exemplo, quando um usuário clica um botão, o evento se propaga através da corrente de elementos GUI que começam com aquele botão, prossegue para seus contêineres (como planilhas ou painéis), e termina com a janela principal da aplicação. O evento é processado pelo primeiro elemento na corrente que é capaz de lidar com ele. Esse exemplo também é notável porque ele mostra que uma corrente pode sempre ser extraída de um objeto árvore.



Uma corrente pode ser formada por uma secção de um objeto.

É crucial que todas as classes handler implementem a mesma interface. Cada handler concreto deve se importar apenas se o seguinte tem o método `executar`. Dessa maneira você pode compor correntes durante a execução, usando vários handlers sem acoplar seu código com suas classes concretas.

Analogia com o mundo real



Uma chamada para o suporte técnico pode atravessar diversos operadores.

Você acabou de comprar e instalar um novo hardware em seu computador. Como você é um geek, o computador tem diversos sistemas operacionais instalados. Você tenta ligar todos eles para ver se o hardware é suportado. O Windows detecta e ativa o hardware automaticamente. Contudo, seu amado Linux se recusa a trabalhar com o novo hardware. Com uma pequena ponta de esperança, você decide ligar para o número do suporte técnico escrito na caixa.

A primeira coisa que você ouve é uma voz robótica do outro lado.

Ela sugere nove soluções populares para vários problemas, nenhum dos quais é relevante para seu caso. Após um tempo, a voz robótica conecta você com um operador de carne e osso.

Infelizmente, o operador não foi capaz de sugerir algo específico também. Ele continuava recitando longos protocolos do manual, se recusando a escutar seus comentários. Após escutar a frase "você tentou desligar e ligar o computador" pela décima vez, você exige ser conectado a um engenheiro.

Eventualmente o operador passa sua chamada para um dos engenheiros, que estava ansioso por contato humano já que estava sentado por horas em sua escura sala do servidor no subsolo de algum prédio. O engenheiro lhe diz onde baixar os drivers apropriados para seu novo hardware e como instalá-los no Linux. Finalmente, a solução! Você termina sua chamada, transbordando de alegria.

Estrutura

1. O **Handler** declara a interface, comum a todos os handlers concretos. Ele geralmente contém apenas um único método para lidar com pedidos, mas algumas vezes ele pode conter outro método para configurar o próximo handler da corrente.
2. O **Handler Base** é uma classe opcional onde você pode colocar o código padrão que é comum a todas as classes handler.

   Geralmente, essa classe define um campo para armazenar uma referência para o próximo handler. Os clientes podem construir uma corrente passando um handler para o construtor ou setter do handler anterior. A classe pode também implementar o comportamento padrão do handler: pode passar a execução para

o próximo handler após checar por sua existência.

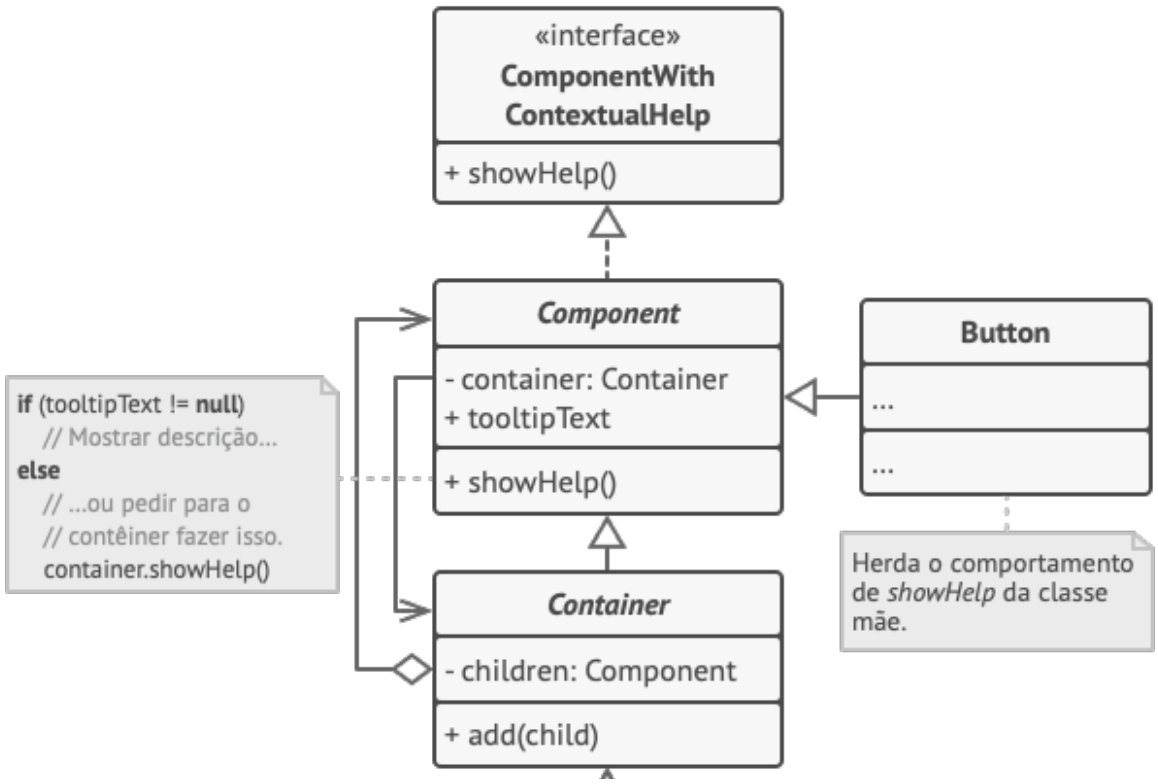
3. **Handlers Concretos** contém o código real para processar pedidos. Ao receber um pedido, cada handler deve decidir se processa ele e, adicionalmente, se passa ele adiante na corrente.

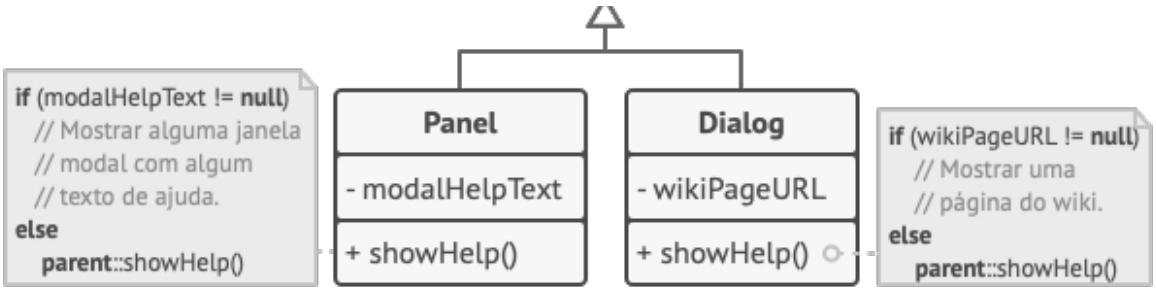
   Os handlers são geralmente auto contidos e imutáveis, aceitando todos os dados necessários apenas uma vez através do construtor.

4. O **Cliente** pode compor correntes apenas uma vez ou compô-las dinamicamente, dependendo da lógica da aplicação. Note que um pedido pode ser enviado para qualquer handler na corrente—não precisa ser ao primeiro.

Pseudocódigo

Neste exemplo, o padrão **Chain of Responsibility** é responsável por mostrar informação de ajuda contextual para elementos de GUI ativos.

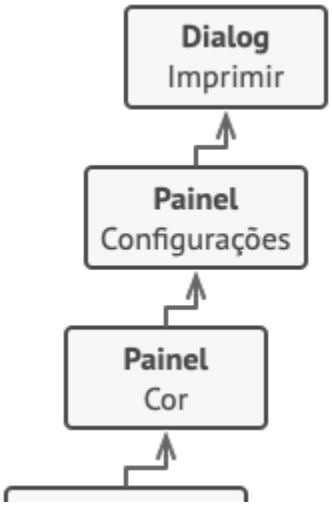




As classes de interface do usuário são construídas com o padrão Composite. Cada elemento é ligado com seu elemento contêiner. A qualquer momento, você pode construir uma corrente de elementos que começa com o elemento em si e vai através de todos os elementos do seu contêiner.

O GUI da aplicação é geralmente estruturado como uma árvore de objetos. Por exemplo, a classe `Dialog`, que renderiza a janela principal da aplicação, seria a raíz do objeto árvore. O dialog contém `Painéis`, que podem conter outros painéis ou simplesmente elementos de baixo nível como `Botões` e `CamposDeTexto`.

Um componente simples pode mostrar descrições contextuais breves, desde que o componente tenha um texto de ajuda assinalado. Mas componentes mais complexos definem seu próprio modo de mostrar ajuda contextual, tais como mostrar um pedaço do manual ou abrir uma página de navegador.



Assim é como um pedido de ajuda atravessa os objetos GUI.

Quando um usuário aponta o cursor do mouse para um elemento e aperta a tecla F1, a aplicação detecta o componente abaixo do cursor e manda um pedido de ajuda. O pedido atravessa todos os elementos do contêiner até chegar no elemento capaz de mostrar a informação de ajuda.

```
// A interface do handler declara um método para executar um
// pedido.
interface ComponentWithContextualHelp is
    method showHelp()


// A classe base para componentes simples.
abstract class Component implements
ComponentWithContextualHelp is
    field tooltipText: string

    // O contêiner do componente age como o próximo elo na
    // corrente de handlers.
    protected field container: Container

    // O componente mostra um tooltip (dica de contexto) se há
    // algum texto de ajuda assinalado a ele. Do contrário ele
    // passa a chamada adiante ao contêiner, se ele existir.
    method showHelp() is
        if (tooltipText != null)
            // Mostrar dica de contexto.
```

```
        else
            container.showHelp()


// Contêineres podem conter tanto componentes simples como
// outros contêineres como filhos. As relações da corrente são
// definidas aqui. A classe herda o comportamento showHelp de
// sua mãe.
abstract class Container extends Component is
    protected field children: array of Component

    method add(child) is
        children.add(child)
        child.container = this


// Componentes primitivos estão de bom tamanho com a
// implementação de ajuda padrão.
class Button extends Component is


// Mas componentes complexos podem sobrescrever a
implementação
// padrão. Se o texto de ajuda não pode ser fornecido de uma
// nova maneira, o componente pode sempre chamar a
implementação
// base (veja a classe Component).
class Panel extends Container is
    field modalHelpText: string
```

```
    method showHelp() is
        if (modalHelpText != null)
            // Mostra uma janela modal com texto de ajuda.
        else
            super.showHelp()

// ...o mesmo que acima...
class Dialog extends Container is
    field wikiPageURL: string

    method showHelp() is
        if (wikiPageURL != null)
            // Abre a página de ajuda do wiki.
        else
            super.showHelp()


// Código cliente.
class Application is
    // Cada aplicação configura a corrente de forma diferente.
    method createUI() is
        dialog = new Dialog("Budget Reports")
        dialog.wikiPageURL = "http://..."
        panel = new Panel(0, 0, 400, 800)
        panel.modalHelpText = "This panel does..."
        ok = new Button(250, 760, 50, 20, "OK")
        ok.tooltipText = "This is an OK button that..."
        cancel = new Button(320, 760, 50, 20, "Cancel")

        panel.add(ok)
```

```
        panel.add(cancel)
        dialog.add(panel)

    // Imagine o que acontece aqui.
    method onF1KeyPress() is
        component = this.getComponentAtMouseCoords()
        component.showHelp()
```

Aplicabilidade

Utilize o padrão Chain of Responsibility quando é esperado que seu programa processe diferentes tipos de pedidos em várias maneiras, mas os exatos tipos de pedidos e suas sequências são desconhecidos de antemão.

O padrão permite que você ligue vários handlers em uma corrente e, ao receber um pedido, perguntar para cada handler se ele pode ou não processá-lo. Dessa forma todos os handlers tem a chance de processar o pedido.

Utilize o padrão quando é essencial executar diversos handlers em uma ordem específica.

Já que você pode ligar os handlers em uma corrente em qualquer ordem, todos os pedidos irão atravessar a corrente exatamente como você planejou.

Utilize o padrão CoR quando o conjunto de handlers e suas encomendas devem mudar no momento de execução.

Se você providenciar setters para um campo de referência dentro das classes handler, você será capaz de inserir, remover, ou reordenar os handlers de forma dinâmica.

Como implementar

1. Declare a interface do handler e descreva a assinatura de um método para lidar com pedidos.

   Decida como o cliente irá passar os dados do pedido para o método. A maneira mais flexível é converter o pedido em um objeto e passá-lo para o método handler como um argumento.

2. Para eliminar código padrão duplicado nos handlers concretos, pode valer a pena criar uma classe handler base abstrata, derivada da interface do handler.

   Essa classe deve ter um campo para armazenar uma referência ao próximo handler na corrente. Considere tornar a classe imutável. Contudo, se você planeja modificar correntes no tempo de execução, você precisa definir um setter para alterar o valor do campo de referência.

   Você também pode implementar o comportamento padrão conveniente para o método handler, que vai passar adiante o pedido para o próximo objeto a não ser que não haja mais objetos. Handlers concretos irão ser capazes de usar esse comportamento ao chamar o método pai.

3. Um por um crie subclasses handler concretas e implemente seus métodos handler. Cada handler deve fazer duas decisões ao receber um pedido:

- Se ele vai processar o pedido.
- Se ele vai passar o pedido adiante na corrente.

4. O cliente pode tanto montar correntes sozinho ou receber correntes pré construídas de outros objetos. Neste último caso,

você deve implementar algumas classes fábrica para construir correntes de acordo com a configuração ou definições de ambiente.

5. O cliente pode ativar qualquer handler da corrente, não apenas o primeiro. O pedido será passado ao longo da corrente até que algum handler se recuse a passá-lo adiante ou até ele chegar ao fim da corrente.
6. Devido a natureza dinâmica da corrente, o cliente deve estar pronto para lidar com os seguintes cenários:

- A corrente pode consistir de um único elo.
- Alguns pedidos podem não chegar ao fim da corrente.
- Outros podem chegar ao fim da corrente sem terem sido tratados.

Prós e contras

- Você pode controlar a ordem de tratamento dos pedidos.
- *Princípio de responsabilidade única*. Você pode desacoplar classes que invocam operações de classes que realizam operações.
- *Princípio aberto/fechado*. Você pode introduzir novos handlers na aplicação sem quebrar o código cliente existente.
- Alguns pedidos podem acabar sem tratamento.

Relações com outros padrões

- O Chain of Responsibility, Command, Mediator e Observer abrangem várias maneiras de se conectar remetentes e destinatários de pedidos:

- O *Chain of Responsibility* passa um pedido sequencialmente ao longo de um corrente dinâmica de potenciais destinatários até que um deles atua no pedido.
- O *Command* estabelece conexões unidirecionais entre remetentes e destinatários.
- O *Mediator* elimina as conexões diretas entre remetentes e destinatários, forçando-os a se comunicar indiretamente através de um objeto mediador.
- O *Observer* permite que destinatários inscrevam-se ou cancelem sua inscrição dinamicamente para receber pedidos.
- O Chain of Responsibility é frequentemente usado em conjunto com o Composite. Neste caso, quando um componente folha recebe um pedido, ele pode passá-lo através de uma corrente de todos os componentes pai até a raiz do objeto árvore.
- Handlers em uma Chain of Responsibility podem ser implementados como comandos. Neste caso, você pode executar várias operações diferentes sobre o mesmo objeto contexto, representado por um pedido.

  Contudo, há outra abordagem, onde o próprio pedido é um objeto *comando*. Neste caso, você pode executar a mesma operação em uma série de diferentes contextos ligados em uma corrente.
- O Chain of Responsibility e o Decorator têm estruturas de classe muito parecidas. Ambos padrões dependem de composição recursiva para passar a execução através de uma série de objetos. Contudo, há algumas diferenças cruciais.

  Os handlers do *CoR* podem executar operações arbitrárias independentemente uma das outras. Eles também podem parar o

pedido de ser passado adiante em qualquer ponto. Por outro lado, vários *decoradores* podem estender o comportamento do objeto enquanto mantém ele consistente com a interface base. Além disso, os decoradores não tem permissão para quebrar o fluxo do pedido.