

Padrões de projeto de software

Aula 6: Padrões GoF – Parte 5

Apresentação

Na engenharia de software, os padrões de design comportamentais identificam e realizam padrões comuns de comunicação entre objetos. Ao fazerem isso, eles aumentam a flexibilidade na execução dessa comunicação.

Nesta aula, mediremos de que maneira os padrões de projeto comportamentais apresentam estratégias validadas para a modelagem da colaboração dos objetos entre si em um sistema, oferecendo comportamentos especiais apropriados para uma ampla variedade de aplicativos. Para fazer isso, listaremos os seguintes padrões: Interpreter, Template Method e Chain of Responsibility.

Objetivos

- Descrever as características do Interpreter;
- Discutir as idiossincrasias do Template Method;
- Ilustrar a estrutura do Chain of Responsibility.

Primeiras palavras

Os padrões comportamentais estabelecem padrões de comunicação para facilitar a troca de informações entre as classes e os objetos. Seu objetivo é mediar a atribuição de responsabilidades entre os objetos do sistema. Gamma e outros autores (1994) destacam a contribuição desses padrões para facilitar a comunicação entre objetos. Eles podem ser classificados em dois tipos:

a) Padrões de escopo de classe

Utilizam o mecanismo da herança para realizar a distribuição do comportamento dos objetos.

Exemplo: o Template Method produz um algoritmo (comportamento) padrão e deixa livre a definição de pontos de execução dele que serão modificados pelas subclasses.

b) Padrões de escopo de objeto

Atuam para compor os objetos a fim de que se comuniquem entre si.

Exemplo: o Mediator (em português, mediador) permite que a comunicação entre uma grande quantidade de objetos seja mediada por um único objeto controlador.

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

Interpreter

O conceito clássico deste padrão, criado pela GoF (abreviação do inglês *Gang of Four*; em português, gangue dos quatro), é o seguinte:

"Dada uma linguagem, o padrão Interpreter define uma representação para sua gramática juntamente com um interpretador, que usa essa representação para interpretar sentenças da linguagem".

- (GAMMA et al., 1994)

Com este padrão, é possível mapear:

Um domínio de um sistema para
um idioma

Deste idioma para uma gramática

Desta gramática para um design
orientado a objetos hierárquico

Identificar padrões pode ser algo complicado, porém, se conseguirmos criar uma gramática para o domínio do problema que o sistema deve resolver, a solução torna-se mais fácil de ser encontrada. Por exemplo, o problema de conversão de uma String representando um número romano em um inteiro que represente seu valor decimal. Neste problema podemos usar o padrão Interpreter.

Problema

Quando uma classe de problemas ocorre repetidamente em um domínio bem compreendido e conhecido, ela pode, se o domínio for caracterizado como uma *linguagem*, ser tratada por meio da interpretação de regras de linguagem.

Se precisarmos criar uma interface de comunicação para converter dados de um sistema para outro, poderemos gerar uma gramática de símbolos para cada sistema e as regras de conversão deles: gramática sistema A: {data: dd/mm/aaaa; código cliente: 2345-5; CEP: 60-867-987}; gramática sistema B: {data: dd.mm.aa; código do cliente: 23455; CEP: 60867987}.

Há duas regras aplicáveis para o exemplo acima:

Regra 1

Substituir o símbolo / da data do sistema A pelo símbolo . do B;

Regra 2

Retirar o símbolo - do código do cliente e do CEP do A para armazenamento apenas dos números do código do cliente e do CEP no sistema B.

Estrutura

O Interpreter possui uma estrutura parecida com a do padrão Composite, pois ela define inicialmente uma classe abstrata, que é a classe base para todas as interpretadoras. A classe base disponibiliza uma interface padrão e um método para interpretar as mensagens recebidas e enviadas pelo objeto.

Ao propor a definição de uma linguagem de domínio (um modo de representar os problemas) como uma *gramática* de linguagem simples, o Interpreter representa as regras de domínio como sentenças de linguagem e as interpreta para resolver o problema.

Este padrão usa uma classe para representar cada regra gramatical.

As gramáticas geralmente possuem estruturas hierárquicas que podem ser usadas para mapear uma hierarquia de herança de classes de regras. Uma classe base abstrata especifica o método *interpret()*. Cada subclasse concreta implementa esse método aceitando (como um argumento) o estado atual do fluxo de idioma e adicionando sua contribuição ao processo de solução de problemas.

A seguir, apresentaremos um passo-a-passo para implementar o padrão Interpreter:

1 Decida se é possível definir uma linguagem e se o esforço para tal tarefa tem um retorno justificável sobre o investimento.

2 Defina uma gramática para a linguagem.

3 Mapeie cada produção na gramática para uma classe.

4 Organize o conjunto de classes na estrutura do padrão Composite.

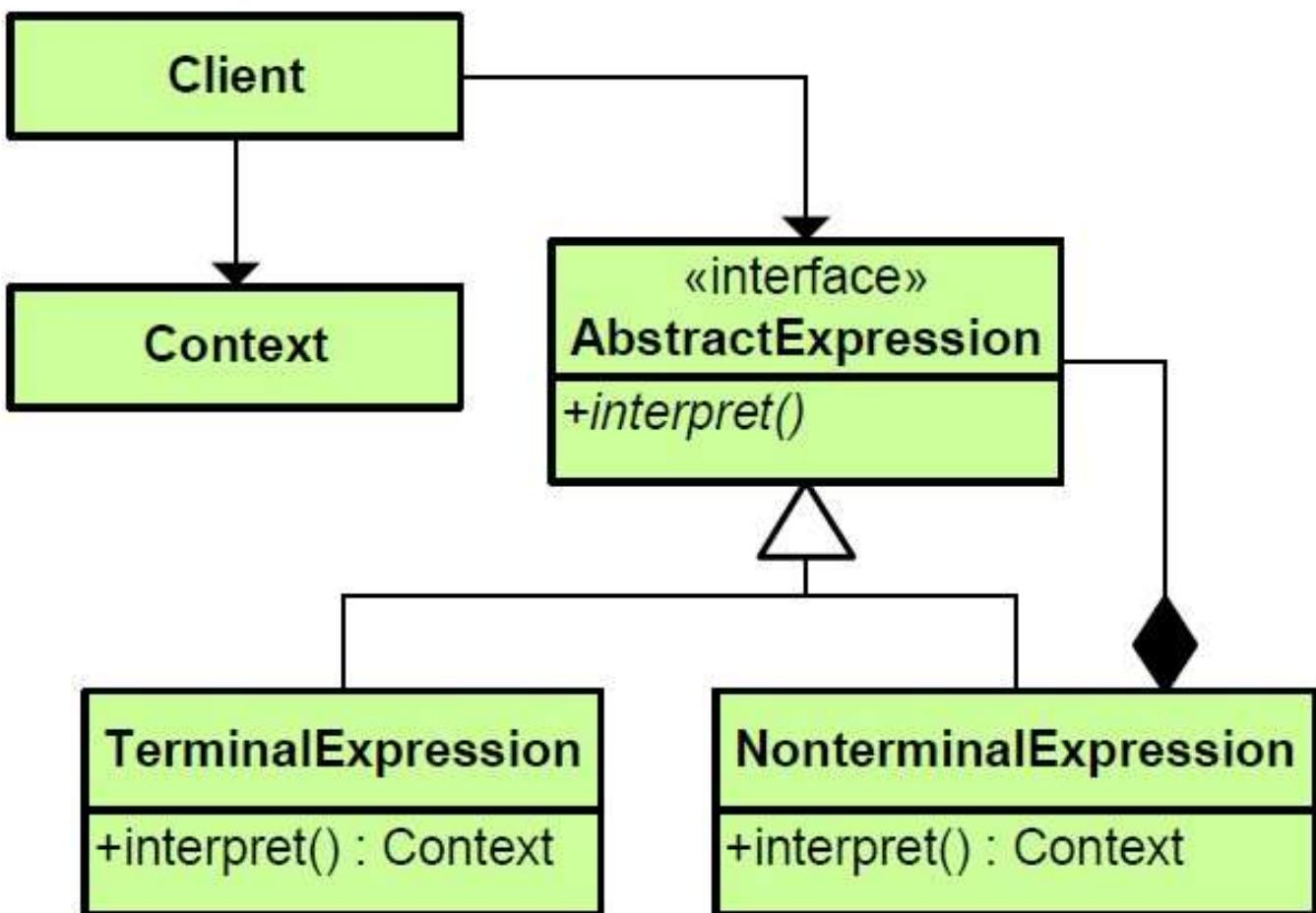
5 Defina um método *interpret* (Context) na hierarquia Composite.

6 O objeto Context encapsula o estado atual da **entrada (para análise)** e **saída (para acumulação)**. O objeto é usado por cada classe de gramática, pois o processo de interpretação transforma a entrada na saída.

Diagrama UML do padrão

A seguir, apresentaremos os componentes deste padrão:

- **Client** (InterpreterApp): Constrói (ou recebe) uma árvore de sintaxe abstrata representando uma frase específica na linguagem que a gramática define. Essa árvore é montada a partir de instâncias das classes NonTerminalExpression e TerminalExpression. O cliente invoca a operação Interpret da classe AbstractExpression;
- **Context** (contexto): Contém informações que são globais para o intérprete;
- **AbstractExpression** (expressão): Declara uma interface para executar uma operação;
- **TerminalExpression** (ExpressãoMilhar, ExpressãoCentena, ExpressãoDezena e ExpressionUnitária): Implementa uma operação Interpret associada ao último símbolo de uma gramática. É preciso criar uma instância para cada símbolo final na sentença;
- **NonterminalExpression** (expressão não terminal): Uma classe deste tipo é criada para cada regra $R = R_1, R_2, R_3 \dots R_n$ na gramática. Essa classe deve manter variáveis de instância do tipo AbstractExpression para cada um dos símbolos (R_1 a R_n). Ela também precisa implementar uma operação Interpret para símbolos não terminais na gramática. O Interpret geralmente é executado recursivamente para as variáveis que representarem R_1 a R_n .



Fonte: (GAMMA et al., 1994)

O Interpreter pode usar o padrão State para definir contextos de análise. Sua árvore de sintaxe abstrata é um Composite; portanto, os padrões Iterator e Visitor também são aplicáveis.

Lembre-se de que o padrão Interpreter não contém recomendações sobre a forma de se analisar a linguagem. Quando a gramática for muito complexa, outras técnicas (como um analisador) serão mais apropriadas.

Padrões relacionados:

- Composite;
- Decorator;
- Chain of Responsibility.

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

Method

O conceito clássico do Template Method criado pela GoF é o seguinte: "Definir o esqueleto de um algoritmo dentro de uma operação, deixando alguns passos serem preenchidos pelas subclasses. Template Method permite que suas subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura". (GAMMA et al., 1994)

Problema

a) Conceito

Este padrão define um algoritmo em termos de operações abstratas cujas subclasses as sobreponem para oferecer comportamento concreto.

b) Aplicação

Quando a estrutura de um algoritmo puder ser definida pela classe mãe (classe abstrata), ela deixará certas partes dele serem preenchidas por classes filhas (classes concretas) que se responsabilizarão pelas implementações do algoritmo.

O problema tratado por este padrão se refere a dois componentes diferentes que, embora não tenham semelhanças significativas, não demonstram a reutilização de interface ou uma implementação comum.

Responsável pela definição dos componentes, o arquiteto de software deve decidir quais partes de um algoritmo são:

- **Invariante**s (devem ser padronizadas): São implementadas em uma classe base abstrata;
- **Variante**s (são personalizáveis): Recebem uma implementação padrão ou nenhuma do tipo. As partes variantes representam *ganchos* ou *espacos reservados*, que podem ou devem ser fornecidos pelo cliente do componente em uma classe derivada concreta.

Estrutura

Em todos os sistemas, existe um programa principal (main). Trata-se de uma estrutura que é chamada no início da execução do sistema.

Considere esta estrutura de laço principal de um programa:

```
Initialize()
While (!Condicao()) // laço principal
{
    Funcao() // faça algo útil
}
Limpeza();
```

Primeiramente, iniciaremos o programa. Em seguida, entraremos no laço principal para fazer o que ele precisa cumprir, como processar eventos GUI ou registros de banco de dados. Por fim, quando terminarmos, sairemos do laço principal, limpando a memória antes de sairmos.

Essa estrutura é tão comum que podemos a capturar em uma classe chamada Application. Podemos reutilizá-la para cada novo programa que quisermos escrever.

Nunca mais teremos de escrever esse laço principal novamente.

Considere este programa FTOC que leia do teclado a temperatura em Fahrenheit e mostre-a na tela em graus Celsius:

```
using System;
using System.IO;
public class FtoCRaw
{
    public static void Main(string[] args)
    {
        bool done = false;
        while (!done)
        {
            string fahrString = Console.In.ReadLine();
            if (fahrString == null || fahrString.Length == 0)
                done = true;
            else
            {
                double fahr = Double.Parse(fahrString);
                double celcius = 5.0/9.0*(fahr - 32);
                Console.Out.WriteLine("Fahrenheit ={0}, convertido para Celsius =
{1}",fahr,celcius);
            }
        }
        Console.Out.WriteLine("Fim do Programa");
    }
}
```

Este programa do exemplo acima tem todos os elementos da estrutura do laço principal anterior. Ele faz uma pequena inicialização; em seguida, realiza o seu trabalho no laço principal; e, por fim, limpa e encerra.

O padrão Template Method pode ser utilizado para separar a estrutura do programa FTOC por meio do armazenamento do código geral em uma classe abstrata. O restante do código que não for genérico deverá ser implementado posteriormente. A seguir, é apresentado o código das classes do padrão:

```

public abstract class Application
{
    private bool isDone = false;
    protected abstract void Init();
    protected abstract void Idle();
    protected abstract void Cleanup();
    protected void SetDone()
    {
        isDone = true;
    }
    protected bool Done()
    {
        return isDone;
    }
    public void Run()
    {
        Init();
        while (!Done())
            Idle();
        Cleanup();
    }
}

```

Application descreve um aplicativo genérico de laço principal. Podemos ver o laço principal na implementação da função *Run()*. Também podemos verificar que todo o trabalho está sendo adiado para os métodos abstratos *Init()*, *Idle()*, e *Cleanup()*. O método *Init()* cuida de qualquer inicialização que precisarmos fazer. Já *Idle()*, que faz o trabalho principal do programa, será chamado repetidamente até a convocação de *SetDone()*.

Cleanup() faz todo o trabalho necessário antes de encerrar o programa. Agora já será possível reescrever a classe FTOC herdada de Application. Simplesmente preencheremos o resumo dos métodos:

```

using System;
using System.IO;
public class FtoCTemplateMethod : Application
{
    private TextReader input;
    private TextWriter output;
    public static void Main(string[] args)
    {
        new FtoCTemplateMethod().Run();
    }
    protected override void Init()
    {
        input = Console.In;
        output = Console.Out;
    }
    protected override void Idle()
    {
        string fahrString = input.ReadLine();
        if (fahrString == null || fahrString.Length == 0)
            SetDone();
        else
        {
            double fahr = Double.Parse(fahrString);
            double celcius = 5.0/9.0*(fahr - 32);
            output.WriteLine("Fahrenheit ={0}, convertido para Celsius =
{1}", fahr, celcius);
        }
    }
    protected override void Cleanup()
    {
        output.WriteLine("Fim do Programa ");
    }
}

```

Este exemplo mostra, de forma simples, a mecânica de uso do Template Method, mas não é recomendado o uso deste padrão para tal programa.

Trata-se, portanto, de um abuso de padrão.

Usar o Template Method para esta aplicação particular não é algo adequado, pois ele complica o programa e o torna maior. Encapsular o laço principal de cada aplicação soava como algo maravilhoso quando começamos, mas sua aplicação prática, neste caso, é infrutífera.

Apresentaremos a seguir um passo-a-passo para implementar o Template Method:

1 Examine o algoritmo e decida quais etapas podem ser padronizadas e quais são peculiares a cada uma das classes atuais.

2 Defina uma nova classe base abstrata para hospedar a estrutura *não ligue para nós, vamos chamá-lo*.

3 Mova o *shell* do algoritmo (agora chamado de método de modelo) e a definição de todas as etapas padronizadas para a nova classe base.

4 Defina um espaço reservado (*placeholder*) ou um método de gancho (*hook*) na classe base para cada etapa que requerer muitas implementações diferentes. Esse método pode hospedar uma implementação padrão ou pode ser definido como abstrato (Java) ou virtual puro (C++).

5 invoque o(s) método(s) de gancho (*hook*) do método de modelo.

6 Cada uma das classes existentes declara um tipo de relacionamento *é-um* para a nova classe base abstrata.

7 Remova das classes existentes todos os detalhes da implementação movidos para a classe base.

8 Os únicos detalhes que permanecem nas classes existentes são aqueles de implementação peculiares a cada classe derivada.

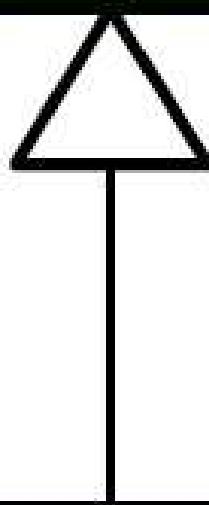
Diagrama UML do padrão

A seguir, divulgaremos os componentes do padrão Template Method:

- **AbstractClass** (objeto de dado): Define as operações abstratas que as subclasses concretas implementarão para executar o algoritmo. Também implementa um método modelo (*template method*) que serve como esqueleto do algoritmo, sendo responsável por chamar as operações primitivas, bem como aquelas definidas nas classes abstratas de outros objetos;
- **ConcreteClass** (objeto de dado do cliente): Implementa as operações primitivas e executa etapas específicas da subclasse do algoritmo.

AbstractClass

+templateMethod()
#subMethod()



ConcreteClass

+subMethod()

Fonte: (GAMMA et al., 1994)

Dica

O Template Method usa a herança para variar parte de um algoritmo e modifica a lógica da classe inteira. Já o padrão Factory Method é uma especialização do Template Method.

Exemplos

Clique no botão acima.

Exemplos

Como podemos melhorar o desempenho do algoritmo de ordenação *bubble sort* (em português, *por bolha*)?

Este algoritmo de ordenação por flutuação é bastante simples. A ideia é percorrer uma lista de itens não ordenados diversas vezes e, a cada passagem, fazer flutuar para o topo o maior elemento dela. Essa movimentação lembra a forma como as bolhas em um tanque de água procuram o próprio nível: vem daí o nome do algoritmo.

Como este algoritmo não é muito eficiente, ele não é utilizado em programas que precisem de velocidade ou operem com quantidade elevada de dados. Podemos melhorar sua funcionalidade com o padrão Template Method.

Vejamos a listagem do algoritmo *Bubblesorter* a seguir:

```
public class BubbleSorter
{
    static int operations = 0;
    public static int Sort(int [] array)
    {
        operations = 0;
        if (array.Length <= 1)
            return operations;
        for (int nextToLast = array.Length-2;
             nextToLast >= 0; nextToLast--)
            for (int index = 0; index <= nextToLast; index++)
                CompareETroquePosicao(array, index);
        return operations;
    }
    private static void TroquePosicao(int[] array, int index)
    {
        int temp = array[index];
        array[index] = array[index+1];
        array[index+1] = temp;
    }
    private static void CompareETroquePosicao(int[] array, int index)
    {
        if (array[index] > array[index+1])
            TroquePosicao (array, index);
        operations++;
    }
}
```

Fonte: (MARTIN; MARTIN, 2006)

A classe BubbleSorter sabe como classificar uma matriz de inteiros usando o algoritmo de classificação de bolhas. Já o método Sort de BubbleSorter contém o algoritmo que sabe como fazer um tipo de bolha, enquanto os dois métodos auxiliares CompareETroquePosicao e TroquePosicao lidam com os detalhes de inteiros e matrizes e a mecânica que o algoritmo de ordenação requer.

O Template Method será utilizado neste exemplo para processar o algoritmo de classificação de bolhas. A classe BubbleSorter implementa o método Sort, que contém as instruções para chamar os métodos abstratos OutofOrder e TroquePosição.

A função desses métodos é a seguinte:

- **OutOfOrder**: Compara dois elementos adjacentes na matriz e retorna verdade (true) se os elementos estiverem fora de ordem;
- **TroquePosicao**: Troca duas células adjacentes na matriz. O método Sort não sabe nada sobre os elementos da lista nem se importa com que tipos de objetos ficam armazenados nela. Ele simplesmente chama OutOfOrder para vários índices na lista e determina se eles devem ser trocados.

```
public abstract class BubbleSorter
{
    private int operations = 0;
    protected int length = 0;
    protected int DoSort()
    {
        operations = 0;
        if (length <= 1)
            return operations;
        for (int nextToLast = length-2;
        nextToLast >= 0; nextToLast--)
            for (int index = 0; index <= nextToLast; index++)
            {
                if (OutOfOrder(index))
                    TroquePosicao(index);
                operations++;
            }
        return operations;
    }
    protected abstract void TroquePosicao(int index);
    protected abstract bool OutOfOrder(int index);
}
```

Usando o BubbleSorter, nós poderemos agora criar classes derivadas capazes de ordenar diferentes tipos de objetos. Podemos criar uma classe para ordenar números inteiros (*integer*) e outra para fazer o mesmo com listas de números do tipo *double*.

Padrões relacionados:

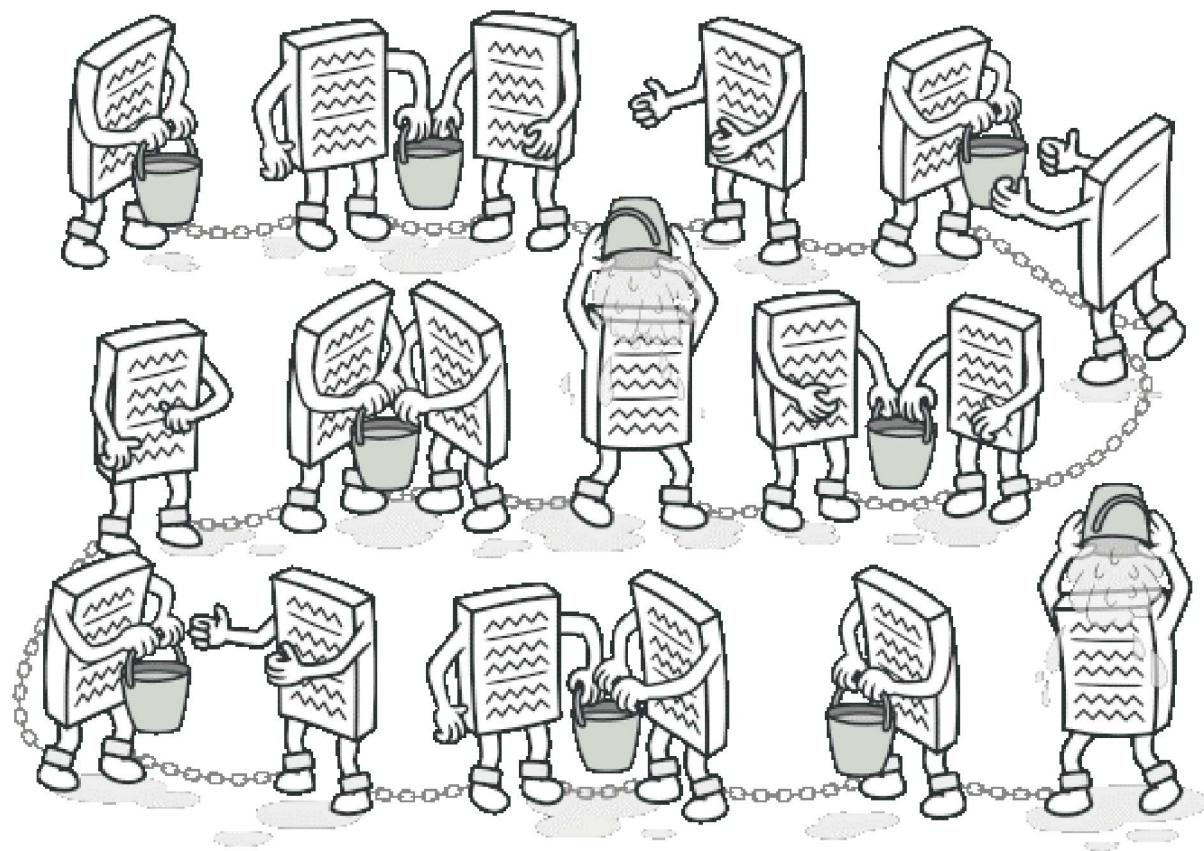
Abstract Factory

Singleton

Flyweight

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

Chain of Responsibility



Fonte: refactoring.guru

O conceito clássico do padrão Chain of Responsibility criado pela GoF é o seguinte:

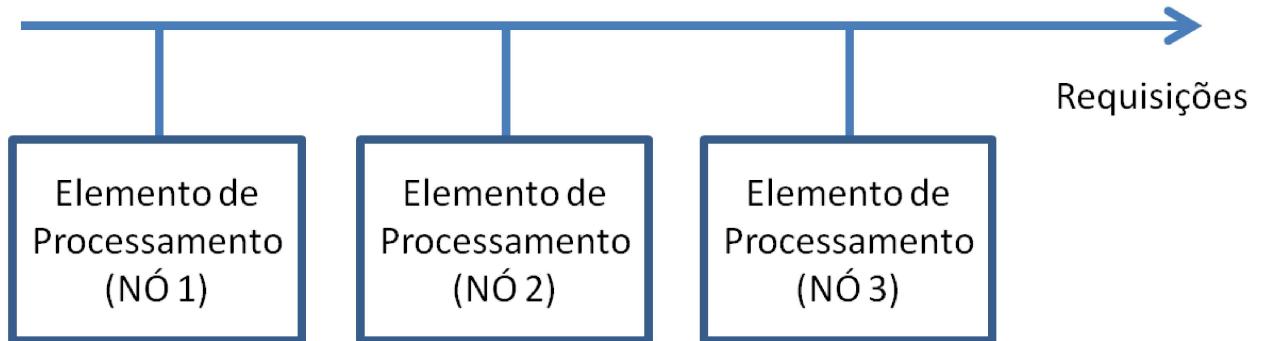
"Evite acoplar o remetente de uma solicitação ao seu destinatário, dando a mais de um objeto a chance de atender à solicitação. Encadeie os objetos de recebimento e passe a solicitação ao longo da cadeia até que um objeto a manipule".

- (GAMMA et al., 1994)

Problema

- Como podemos permitir que vários objetos sirvam a uma requisição ou repassá-la?
- Como conseguimos consentir que a divisão de responsabilidades ocorra de forma transparente?
- Como tratar requisições em que existe um número potencialmente variável de objetos do tipo *manipulador*, elemento de processamento ou nós e um fluxo de requisições que devem ser manipulados?

Cliente



Os objetos de um sistema orientado a objetos trocam informações utilizando mensagens enviadas e recebidas. Durante a execução do sistema, eles fazem chamadas aos métodos de outros objetos que devem ser atendidas de acordo com as regras de funcionamento do sistema.

Exemplo

Considere o projeto de um sistema de telefonia com três linhas para um escritório. Quando alguém ligar para lá, a primeira linha tratará a chamada; caso ela esteja ocupada, a segunda fará isso; se ela também estiver sendo usada, a terceira cuidará dessa chamada. Se todas as linhas no sistema estiverem ocupadas, uma secretária eletrônica irá instruir o originador da chamada a esperar a próxima linha disponível. Quando uma linha estiver disponível, ela, por fim, poderá atender a essa chamada.

Chain of Responsibility permite que um sistema determine, durante sua execução, qual objeto irá cuidar de uma mensagem.

Este padrão de projeto permite que um objeto envie uma mensagem para vários objetos encadeados em sequência.

Cada um deles pode a tratar ou repassá-la para o próximo objeto.

Exemplo

A primeira linha no sistema de telefonia é o primeiro objeto na cadeia de responsabilidades (em inglês, chain of responsibility); a segunda linha, o segundo objeto; a terceira linha, o terceiro; e a secretária eletrônica, o quarto.

O objeto final na cadeia é a próxima linha livre que recebe e trata da mensagem.

Essa cadeia é criada dinamicamente em resposta à presença ou à ausência de operadores específicos de mensagens.

Estrutura

A seguir, exibiremos os procedimentos de utilização do Chain of Responsibility em um projeto de software:

1 A classe base mantém um ponteiro *próximo*.

2 Cada classe derivada implementa sua contribuição para manipular a solicitação.

3 Se a solicitação precisar ser *passada*, a classe derivada a *chamará de volta* para a classe base, que a delega para o ponteiro *next*.

4 O cliente (ou algum terceiro) cria e vincula a cadeia (que pode incluir um link do último nó para o nó raiz).

5 O cliente *inicia e sai* de cada solicitação com a raiz da cadeia.

6 Delegação recursiva produz a ilusão de alta capacidade de atendimento da demanda.

Diagrama UML do padrão

Anunciaremos a seguir os componentes do padrão Chain of Responsibility:

a) Client (ChainApp)

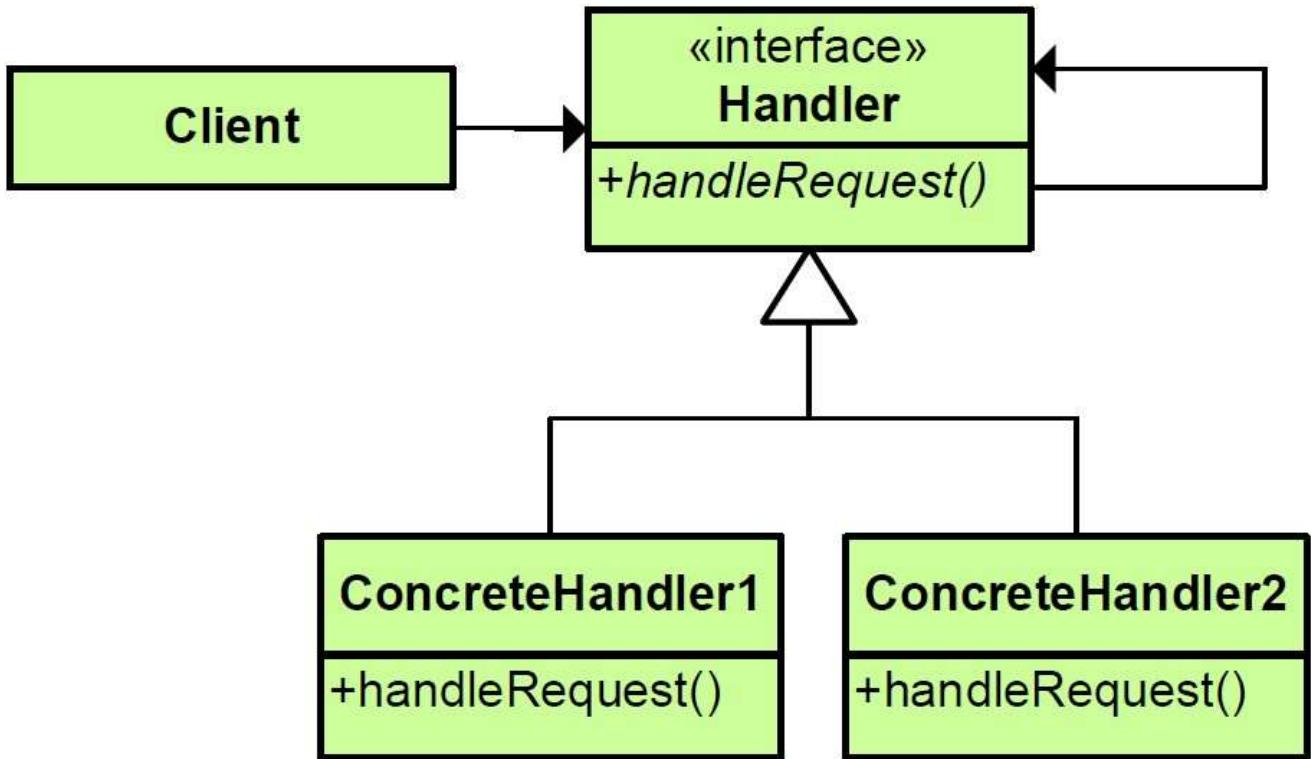
Inicia a requisição para um objeto ConcreteHandler na cadeia.

b) Handler (aprovador)

Define uma interface para manipular (handle) os pedidos. Opcionalmente, ele implementa o link sucessor.

c) ConcreteHandler (Diretor, VicePresidente, Presidente)

- Manipula as requisições sob sua responsabilidade;
- Pode acessar seu sucessor na cadeia;
- Se o ConcreteHandler puder manipular a requisição, isso acontecerá; caso contrário, ele encaminhará a solicitação ao seu sucessor.



Fonte: (GAMMA et al., 1994)

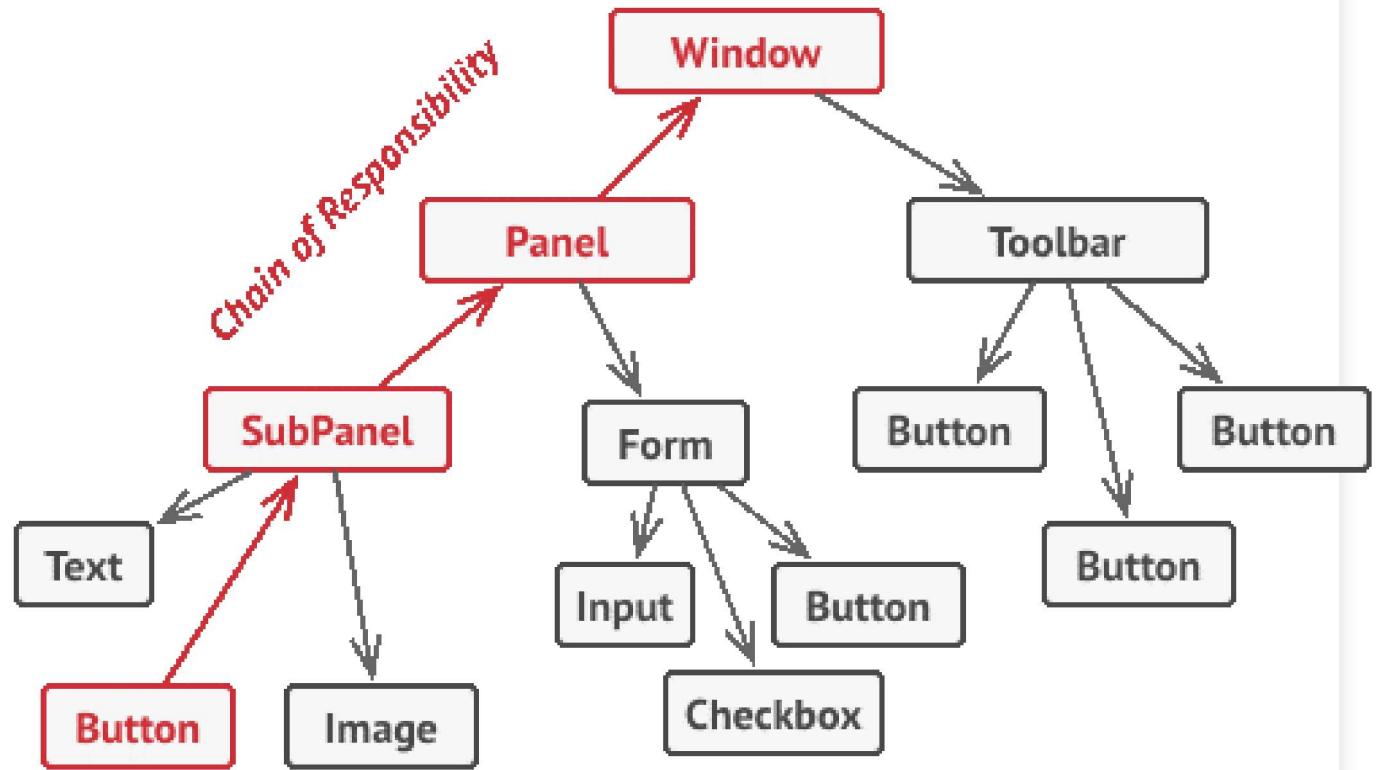
Os padrões Chain of Responsibility, Command, Mediador e Observer abordam como você pode dissociar remetentes e destinatários, mas com diferentes compensações.

O Chain of Responsibility passa uma solicitação do remetente ao longo de uma cadeia de possíveis receptores. Ele pode usar o Command para representar solicitações como objetos. Este padrão é frequentemente aplicado em conjunto com o Composite, que pode tratar os objetos da composição para atuarem como uma cadeia de objetos.

 Exemplo

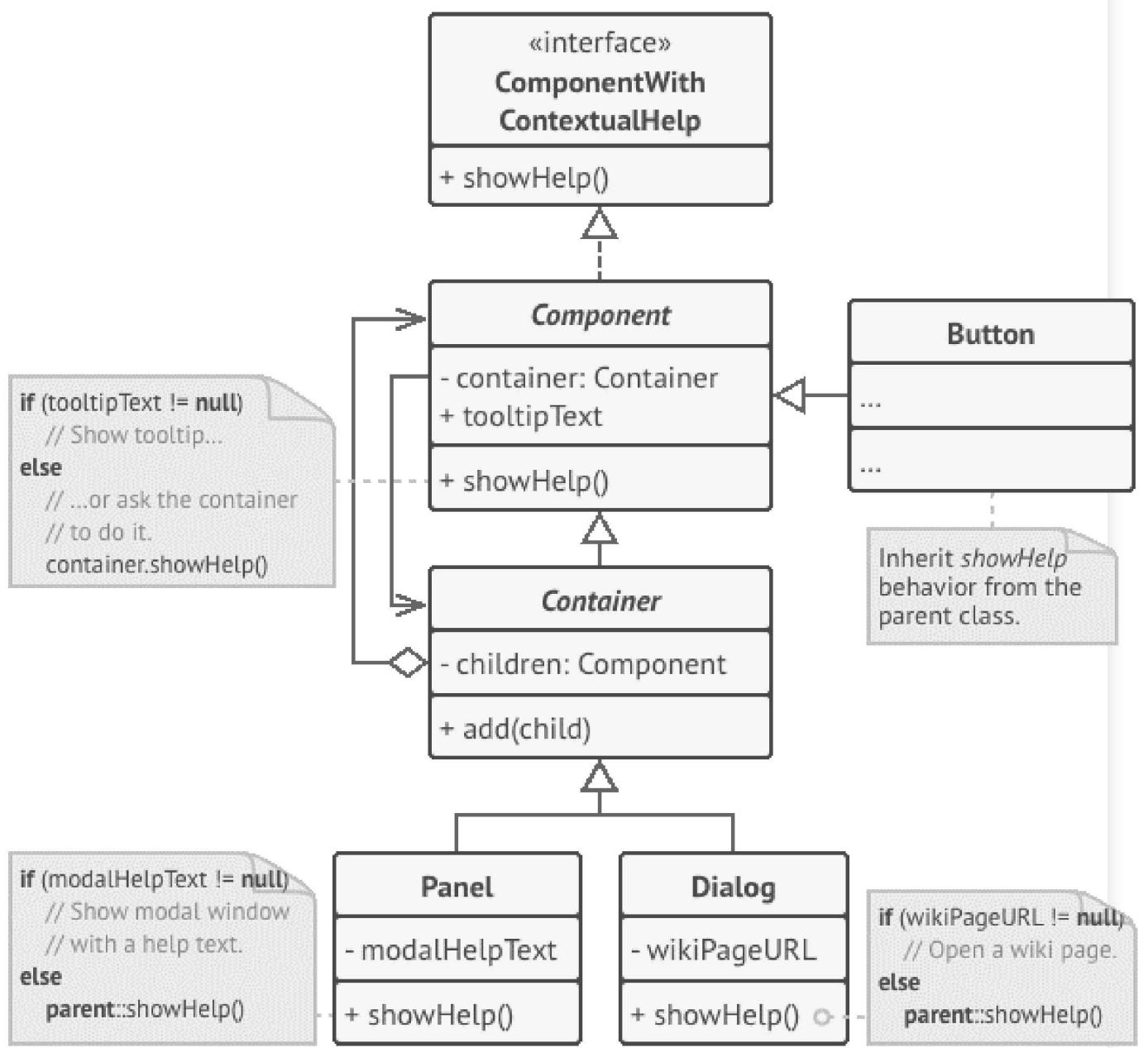
 Clique no botão acima.

Exemplo



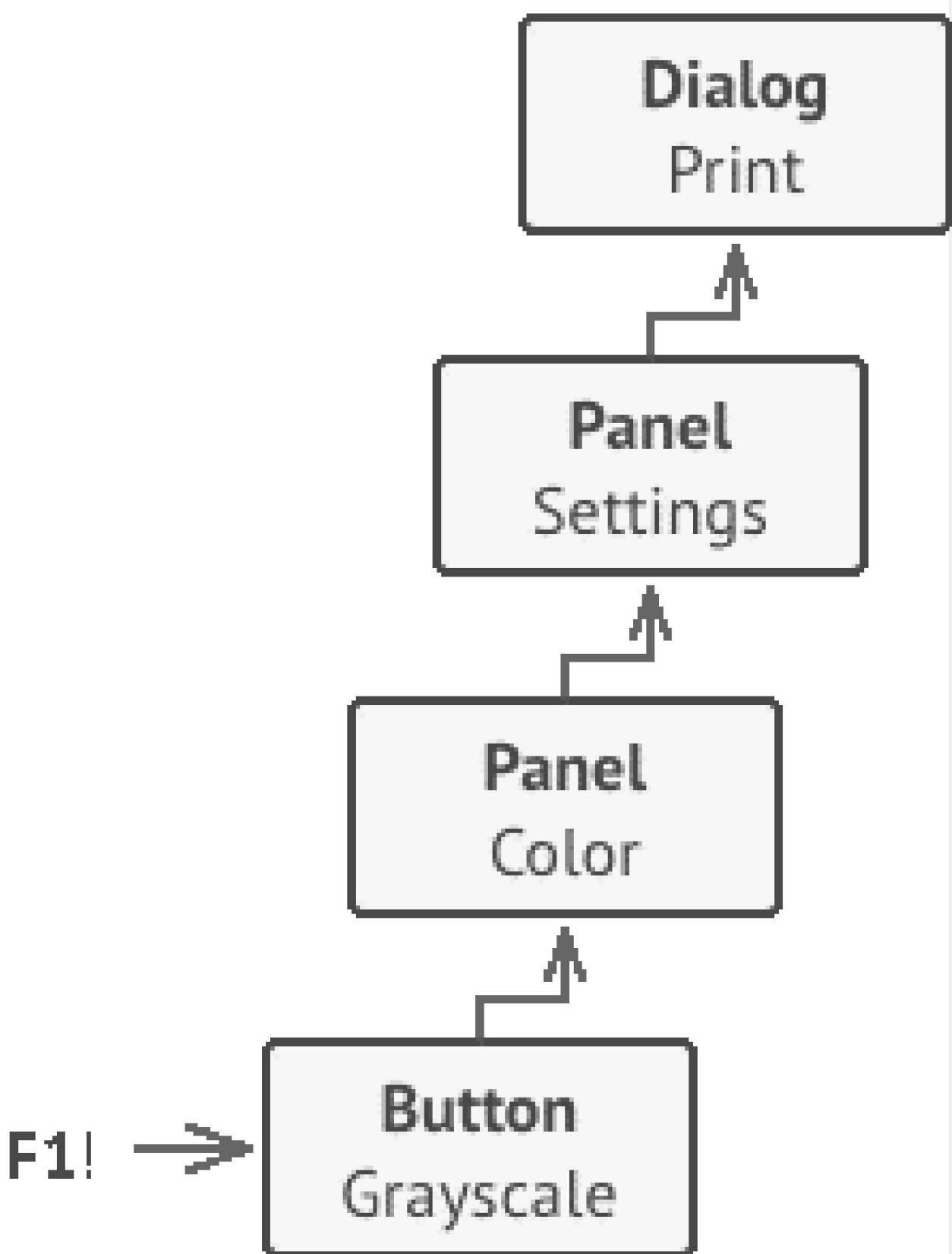
Fonte: [refactoring.guru](https://refactoring.guru/design-patterns/chain-of-responsibility)

Neste exemplo, Chain of Responsibility é responsável por exibir informações de ajuda contextual para elementos da GUI ativa.



Fonte: refactoring.guru

A interface gráfica (GUI) do aplicativo geralmente é estruturada como uma árvore de objetos. A classe Dialog, que renderiza a janela principal do aplicativo, seria a raiz da árvore de objetos. A caixa de diálogo contém Painéis (Panel), que podem possuir outros painéis ou elementos simples de baixo nível, como Botões (Buttons) e Campos de Texto (TextFields). Um componente simples pode mostrar breves dicas contextuais desde que o componente tenha algum texto de ajuda atribuído. No entanto, componentes mais complexos definem a própria maneira de mostrar a ajuda contextual, como mostrar um trecho do manual ou abrir uma página em um navegador.



 Fonte: refactoring.guru

Quando um usuário aponta o cursor do mouse em um elemento e pressiona a tecla F1, o aplicativo detecta o componente sob o ponteiro e envia uma solicitação de ajuda. Essa solicitação passa por todos os contêineres do elemento até alcançar o elemento capaz de exibir as informações de ajuda.

Composite

Decorator

Interpreter

Atividade

1. Quando temos problemas no projeto de desenvolvimento de um software que são bastante conhecidos e podem ser mapeados em um conjunto de regras de uma linguagem, qual padrão pode ser utilizado pela equipe?

- a) Integer
 - b) Facade
 - c) Bridge
 - d) Template Method
 - e) Singleton
-

2. O problema tratado por este padrão se refere a dois componentes diferentes que têm semelhanças significativas, embora ambos não demonstrem a reutilização de interface ou a implementação comum. Estamos nos referindo ao padrão:

- a) Integer
 - b) Facade
 - c) Bridge
 - d) Template Method
 - e) Singleton
-

3. São características do padrão Chain of Responsibility as seguintes características, **exceto**:

- a) Diminui o acoplamento entre os objetos.
 - b) Permite que a divisão de responsabilidades ocorra de forma transparente no sistema.
 - c) Permite que um sistema determine, em tempo de execução, qual objeto tratará uma mensagem.
 - d) Passa uma solicitação do remetente ao longo de uma cadeia de possíveis receptores.
 - e) Propõe a definição de uma linguagem de domínio (um modo de representar os problemas) como uma *gramática* de linguagem simples, representando regras de domínio, como sentenças de linguagem, e interpretando essas sentenças para resolver o problema.
-

Notícias

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns**: elements of reusable object-oriented software. New York: Addison-Wesley Professional, 1994.

FRIEMAN, E. **Use a cabeça!** Padrões de projeto. 2. ed. Rio de Janeiro: Elsevier, 2007.

LEÃO, L. **Padrões de projeto de software**. Rio de Janeiro: SESES, 2018.

MARTIN, R. C.; MARTIN, M. **Agile principles, patterns, and practices in C#**. New Jersey: Prentice-Hall, 2006.

- Apresentação dos padrões Command, Iterator, Mediator e Memento.

Explore mais

Exemplos de código C# para os padrões:

- [Interpreter](#);
- [Template Method](#);
- [Chain of Responsibility](#).