Design Patterns

Strategy

Padrão comportamental que permite definir uma familia de algoritmos, encapsular cada um deles e torná-los intercambiáveis.
Permite que o algoritmo varie independentemente dos clientes que o utilizam.
Uma abordagem promove um codigo mais limpo, flexível e de fácil manutenção.
Especialmente em casos onde múltiplas variações de uma mesma operaçoes são necessárias.

Intenção e Propósito

Extrair algoritmos complexos ou atributos de uma classe principal(conhecida como *contexto*) e encapsulá-los em classes separadas(as *estratégias*).
Permite que o **contexto** se comunique com essas **estratégias** através de uma interface comum, sem conhecer detalhes ou comportamento.
É possível adicionar novas estratégias ou modificar as existentes sem alterar o codigo do contexto.

Componentes do padrão

O padrão *Strategy* é composto por 3 elementos principais:

Contexto

Classe que contem uma referência a um objeto da estratégia.
O contexto não implementa codigo diretamente.
Mas delega essas responsabilidades para a estratégia que lhe foi configurada.
Comunica com o objeto estratégia através da interface comum.

Estratégia

Interface ou classe abstrata que define um método comum para todos os algoritmos suportados.
Contexto utiliza esta interface para invocar o algoritmo da estratégia concreta.

Estratégia concretas

Classes que implementam a interface da estratégia, cada um fornecendo um implementação específica de um algoritmo (ou regra).

Aplicação

Múltiplas variações de um algoritmo

Uma classe precisa executar uma tarefa de diferentes maneiras.
Por exemplo:
Calcular frete de um produto com base em diferentes modalidades de entrega (normal, expresso, Sedex).

Eliminar condicionais complexas

Em vez de usar grandes estruturas `if-else` ou `switch-case` para selecionar um comportamento.
Permite selecionar a implementação desejada em tempo de execução.

Isolamento do código

Quando os detalhes de implementação de um algoritmo não saõ importantes para a classe que o utiliza.
Facilita a manutenção e a evolução do código, pois as estratégias podem ser modificadas ou adicionadas sem impactar o contexto.

Reutilização de Algoritmos

Algoritmos encapsulados nas estratégias podem ser reutilizados de diferentes maneiras.

Vantagens e desvantagens

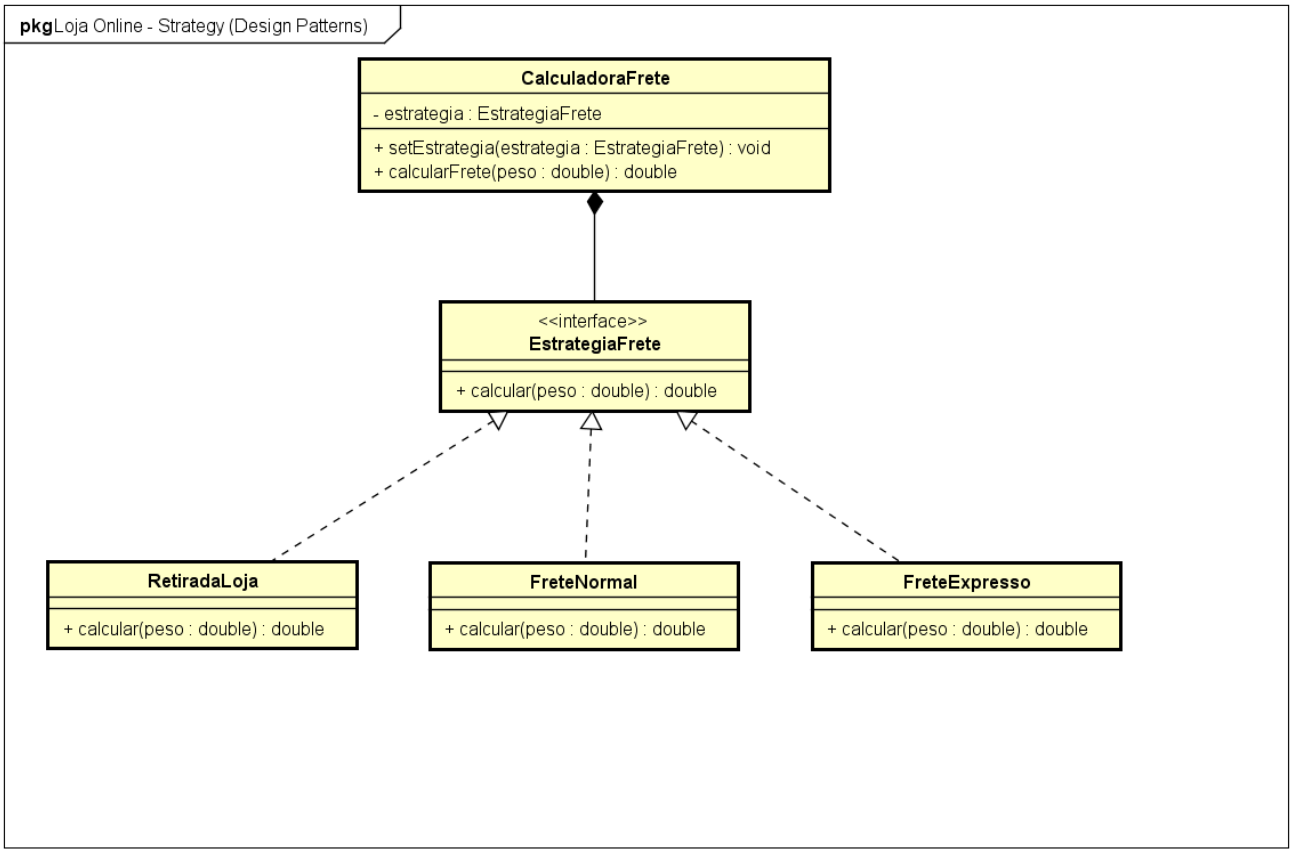
Vantagens

Flexibilidade e extensibilidade
Novas estratégias podem ser adicionadas facilmente sem modificar o contexto.
Codigo mais limpo e coeso
Reduz o acoplamento entre o contexto e as implementações dos algoritmos.
Manutenção simplificada
Alterações em um algoritmo específico ficam contidas em sua respectiva classe de estratégia.
Segregação de responsabilidade
Contexto foca em sua responsabilidade principal, enquanto as estratégias se concentram na implementação dos algoritmos.

Desvantagens

Aumento do número de classes
Implementação do padrão leva a aumento no número de classes e objetos no sistema.
Comunicação entre Estratégia e Contexto
A estratégia pode precisar de dados do contexto, que pode aumentar a complexidade da comunicação entre eles.

UML



Command

Analogia

Imagine que você está em um restaurante. Você (o **Cliente**) faz um pedido para o garçom (o **Invocador**). O garçom anota seu pedido em uma comanda (o **Comando**). A comanda é um objeto que contém todos os detalhes da sua solicitação (exemplo: "um bife mal passado").
O garçom então leva essa comanda para a cozinha e a coloca em uma fila. Ele não precisa saber como cozinhar o bife; sua única responsabilidade é entregar o pedido. O cozinheiro (o **Receptor**) pega a comanda e, sabendo como executar aquela tarefa específica, prepara o prato.

Visão geral

Padrão comportamental mais útil e interessante.
O padrão funciona da seguinte maneira: ele transforma uma solicitação em um objeto autônomo (a comanda) que contém todas as informaçoes sobre a solicitação
Isso permite desacoplar quem cria a solicitação de quem executa.
Como fosse um "independência" entre objetos.

Pontos importantes

Intenção e propósito

O objetivo principal é **encapsular um solicitação como um objeto**, permitindo parametrizar clientes com diferentes solicitações, enfileirar ou registrar (log) solicitações e suportar operações que podem ser desfeitas (undo).
Ele desacopla o objeto que invoca uma operação do objeto que sabe como realizá-la.

Componentes do padrão

O Command é formado por cinco principais componentes:
Command (Comando)
Interface que geralmente declara um único método, como `execute()` .
Concrete Command (Comando Concreto)
Implementação da interface Command. Um comando concreto armazena um referencia ao objeto Receptor e implementa o método `execute()` chamando a ação apropriada no Receptor.
Contém todos os parâmetros necessários para que o Receptor execute a ação.
Receiver (Receptor)
É o objeto que de fato executa o trabalho.
Ele contém a lógica do negócio real.
Quais quer classe pode atuar como um Receptor.
Invoker (Invocador)
Objeto que solicita a execução do comando.
Invocador não sabe como o trabalho será feito, ele apenas sabe como passas a solicitação para o objeto de comando.
Exemplos:
Botões em uma interface gráfica, itens de menu ou controle remoto.
Client (Cliente)
Parte da aplicação que cria e configura os objetos.
Cliente cria o *Comando Concreto*, define seu *Receptor* e o associa a um *Invocador*.

Cenários de aplicação - quando usar

Para parametrizar objetos com ações

Em vez de uma classe ter um comportamento fixo, você pode dar a ela um objeto de comando para executar, tomando seu comportamento dinâmico.

Para enfileirar, agendar ou executar operações remotamente

Como o comando é um objeto, ele pode ser facilmente serializado, armazenando em uma fila enviado, enviado pela rede ou agendado para execução posterior.

Para implementar operações de desfazer/refazer (undo/redo)

A aplicação mais famosa do padrão. para implementar o "undo", a interface do Comando pode ter um método `undo()`. O Comando Concreto, ao ser executado, pode salvar o estado anterior do Receptor para restaurá-lo caso o `undo()` seja chamado.

Vantagens e desvantagens

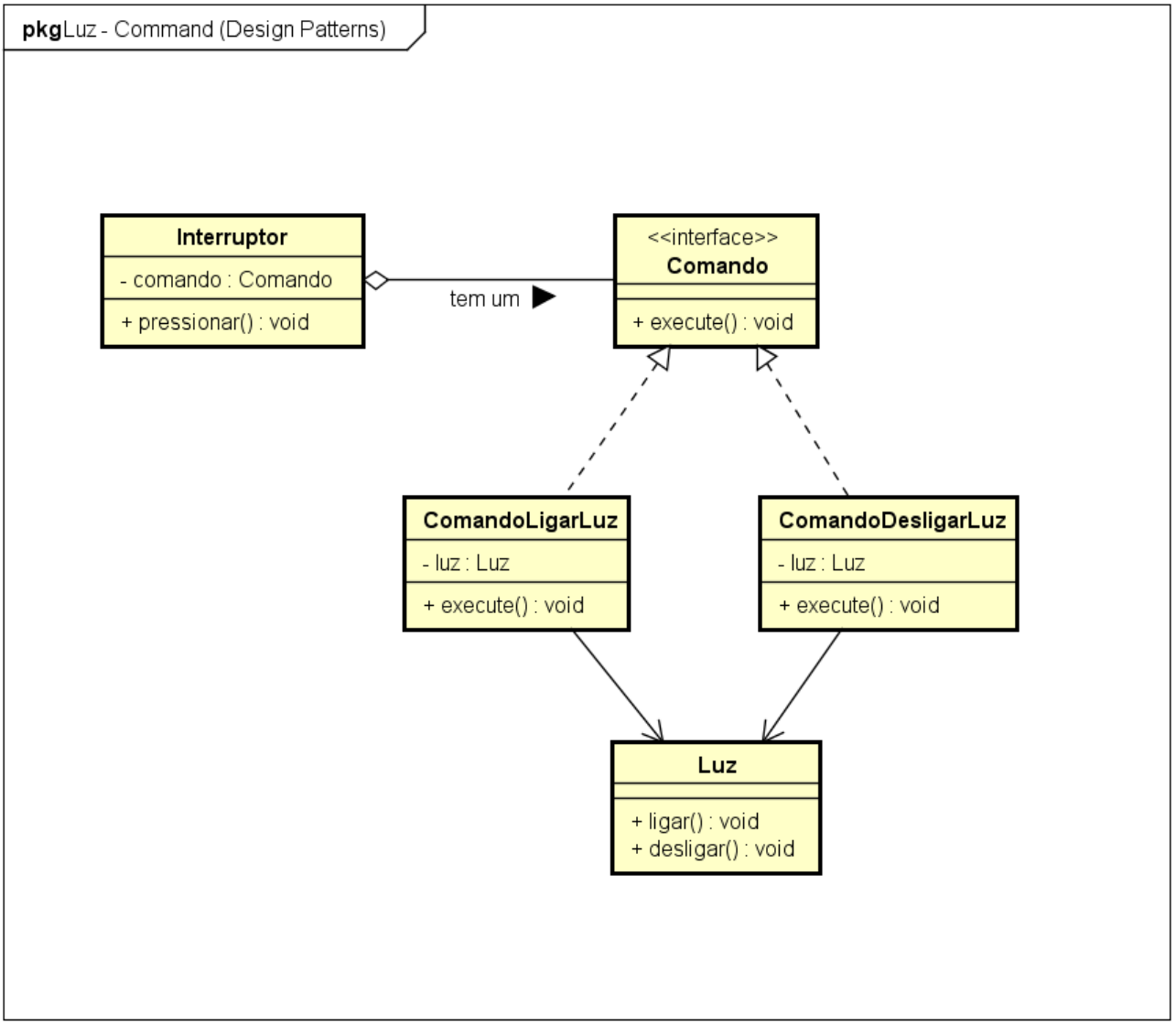
Vantagens

Desacoplamento
O Invocador é completamente desacoplado do Receptor.
Invocador só reconhece a interface do Comando.
Principio da responsabilidade única
As classes invocam operações são separadas das classes que executam.
Princípio do aberto/fechado
É fácil adicionar novos comandos à aplicação sem precisar refatorar o código do Invocador ou dos Receptores.
Composição de Comandos
É possível montar uma sequência de comandos (um "macro comando") que executa várias operações de uma vez.

Desvantagens

Aumento da complexidade
Padrão introduz muitas classes e objetos novos, o que pode complicar o código se a operação for muito simples.
Para uma ligação direta entre um botão e uma ação simples, pode ser um exagero.

UML



Referência

[Padrões de Projeto]