refactoring.guru

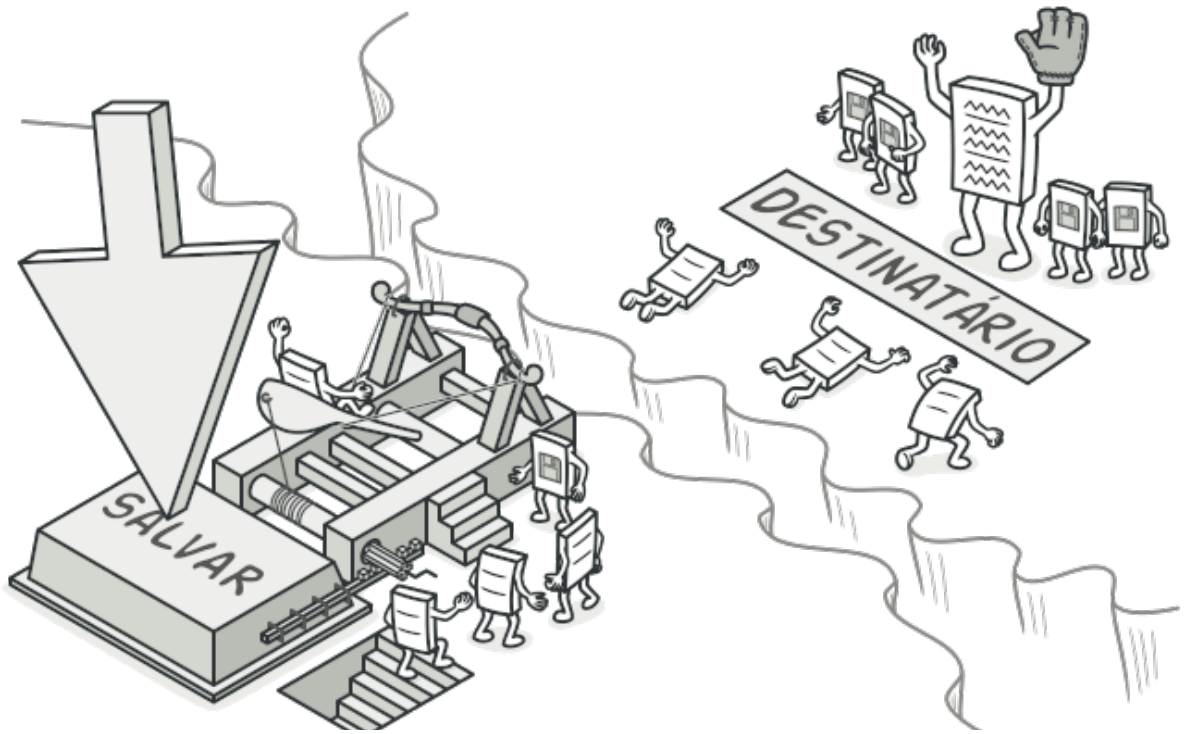
Command

19–26 minutes

Também conhecido como:
Comando, Ação, Action, Transação, Transaction

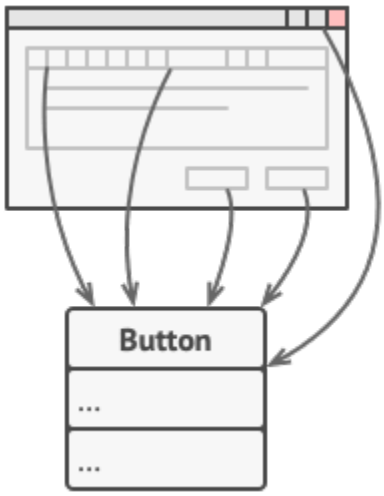
Propósito

O **Command** é um padrão de projeto comportamental que transforma um pedido em um objeto independente que contém toda a informação sobre o pedido. Essa transformação permite que você parametrize métodos com diferentes pedidos, atrase ou coloque a execução do pedido em uma fila, e suporte operações que não podem ser feitas.



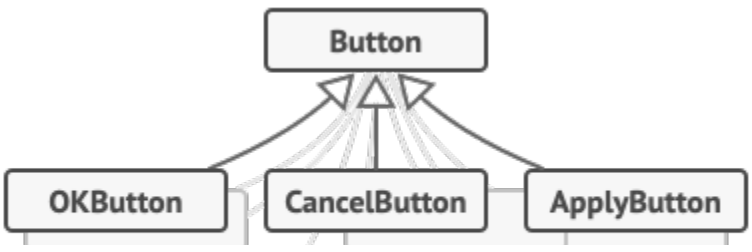
Problema

Imagine que você está trabalhando em uma nova aplicação de editor de texto. Sua tarefa atual é criar uma barra de tarefas com vários botões para várias operações do editor. Você criou uma classe `Botão` muito bacana que pode ser usada para botões na barra de tarefas, bem como para botões genéricos de diversas caixas de diálogo.



Todos os botões de uma aplicação são derivadas de uma mesma classe.

Embora todos esses botões pareçam similares, eles todos devem fazer coisas diferentes. Aonde você deveria colocar o código para os vários handlers de cliques desses botões? A solução mais simples é criar um monte de subclasses para cada local que o botão for usado. Essas subclasses conteriam o código que teria que ser executado em um clique de botão.



Várias subclasses de botões. O que pode dar errado?

Não demora muito e você percebe que essa abordagem é falha. Primeiro você tem um enorme número de subclasses, e isso seria okay se você não arriscasse quebrar o código dentro dessas subclasses cada vez que você modificar a classe base `Botão`. Colocando em miúdos: seu código GUI se torna absurdamente dependente de um código volátil da lógica do negócio.

Várias classes implementam a mesma funcionalidade.

E aqui está a parte mais feia. Algumas operações, tais como copiar/colar texto, precisariam ser invocadas de diversos lugares. Por exemplo, um usuário poderia criar um pequeno botão “Copiar” na barra de ferramentas, ou copiar alguma coisa através do menu de contexto, ou apenas apertando `Crtl+C` no teclado.

Inicialmente, quando sua aplicação só tinha a barra de ferramentas, tudo bem colocar a implementação de várias operações dentro das subclasses do botão. Em outras palavras, ter o código de cópia de texto dentro da subclasse `BotãoCópia` parecia certo. Mas então, quando você implementou menus de contexto, atalhos, e outras coisas, você teve que ou duplicar o código da operação em muitas classes ou fazer menus dependentes de botões, o que é uma opção ainda pior.

Solução

Um bom projeto de software quase sempre se baseia no princípio da separação de interesses, o que geralmente resulta em dividir a aplicação em camadas. O exemplo mais comum: uma camada para a interface gráfica do usuário e outra camada para a lógica do negócio. A camada GUI é responsável por renderizar uma bonita imagem na tela, capturando quaisquer dados e mostrando resultados do que o usuário e a aplicação estão fazendo. Contudo, quando se trata de fazer algo importante, como calcular a trajetória da lua ou compor um relatório anual, a camada GUI delega o trabalho para a camada inferior da lógica do negócio.

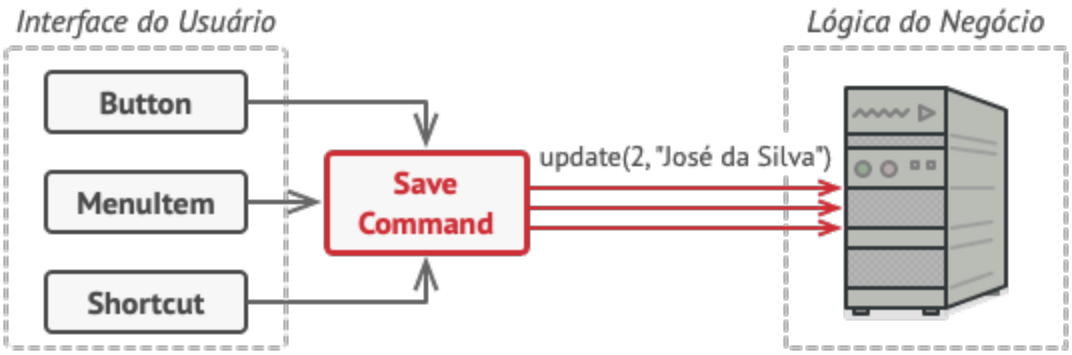
Dentro do código pode parecer assim: um objeto GUI chama um método da lógica do negócio, passando alguns argumentos. Este processo é geralmente descrito como um objeto mandando um *pedido* para outro.

Os objetos GUI podem acessar os objetos da lógica do negócio diretamente.

O padrão Command sugere que os objetos GUI não enviem esses pedidos diretamente. Ao invés disso, você deve extrair todos os detalhes do pedido, tais como o objeto a ser chamado, o nome do método, e a lista de argumentos em uma classe *comando*

separada que tem apenas um método que aciona esse pedido.

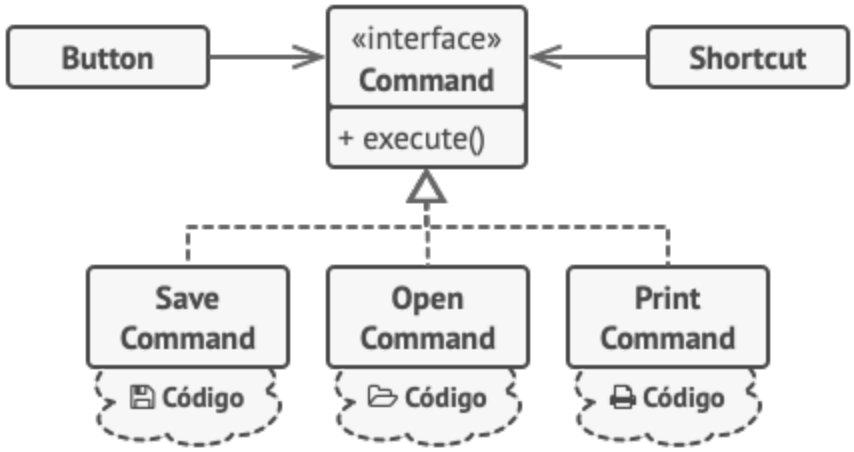
Objetos comando servem como links entre vários objetos GUI e de lógica de negócio. De agora em diante, o objeto GUI não precisa saber qual objeto de lógica do negócio irá receber o pedido e como ele vai ser processado. O objeto GUI deve acionar o comando, que irá lidar com todos os detalhes.



Acessando a lógica do negócio através do comando.

O próximo passo é fazer seus comandos implementarem a mesma interface. Geralmente é apenas um método de execução que não pega parâmetros. Essa interface permite que você use vários comandos com o mesmo remetente do pedido, sem acoplá-lo com as classes concretas dos comandos. Como um bônus, agora você pode trocar os objetos comando ligados ao remetente, efetivamente mudando o comportamento do remetente no momento da execução.

Você pode ter notado uma peça faltante nesse quebra cabeças, que são os parâmetros do pedido. Um objeto GUI pode ter fornecido ao objeto da camada de negócio com alguns parâmetros, como deveremos passar os detalhes do pedido para o destinatário? Parece que o comando deve ser ou pré configurado com esses dados, ou ser capaz de obtê-los por conta própria.



Os objetos GUI delegam o trabalho aos comandos.

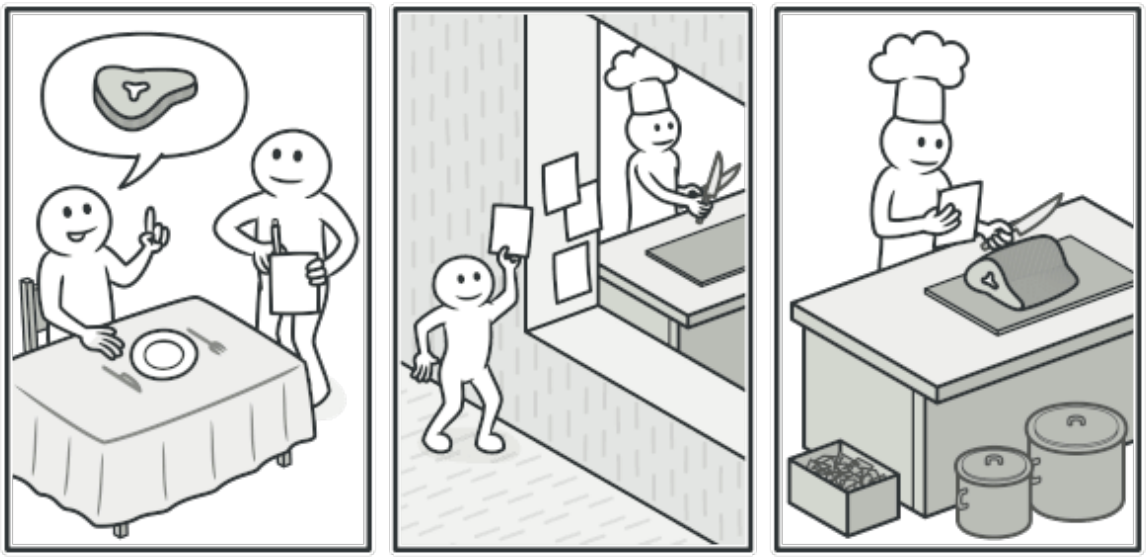
Vamos voltar ao nosso editor de texto. Após aplicarmos o padrão Command, nós não mais precisamos de todas aquelas subclasses de botões para implementar vários comportamentos de cliques. É suficiente colocar apenas um campo na classe `Botão` base que armazena a referência para um objeto comando e faz o botão executar aquele comando com um clique.

Você vai implementar um monte de classes comando para cada possível operação e ligá-los aos seus botões em particular, dependendo do comportamento desejado para os botões.

Outros elementos GUI, tais como menus, atalhos, ou caixas de diálogo inteiras, podem ser implementados da mesma maneira. Eles serão ligados a um comando que será executado quando um usuário interage com um elemento GUI. Como você provavelmente adivinhou, os elementos relacionados a mesma operação serão ligados aos mesmos comandos, prevenindo a duplicação de código.

Como resultado, os comandos se tornam uma camada intermédia conveniente que reduz o acoplamento entre as camadas GUI e de lógica do negócio. E isso é apenas uma fração dos benefícios que o padrão Command pode oferecer.

Analogia com o mundo real



Fazendo um pedido em um restaurante.

Após uma longa caminhada pela cidade, você chega a um restaurante bacana e senta numa mesa perto da janela. Um garçom amigável se aproxima e rapidamente recebe seu pedido, escrevendo-o em um pedaço de papel. O garçom vai até a cozinha e prende o pedido em uma parede. Após algum tempo, o pedido chega até o chef, que o lê e cozinha a refeição de acordo. O cozinheiro coloca a refeição em uma bandeja junto com o pedido. O garçom acha a bandeja, verifica o pedido para garantir que é aquilo que você queria, e o traz para sua mesa.

O pedido no papel serve como um comando. Ele permanece em uma fila até que o chef esteja pronto para serví-lo. O pedido contém todas as informações relevantes necessárias para cozinhar a refeição. Ele permite ao chef começar a cozinhar imediatamente ao invés de ter que ir até você para clarificar os detalhes do pedido pessoalmente.

Estrutura

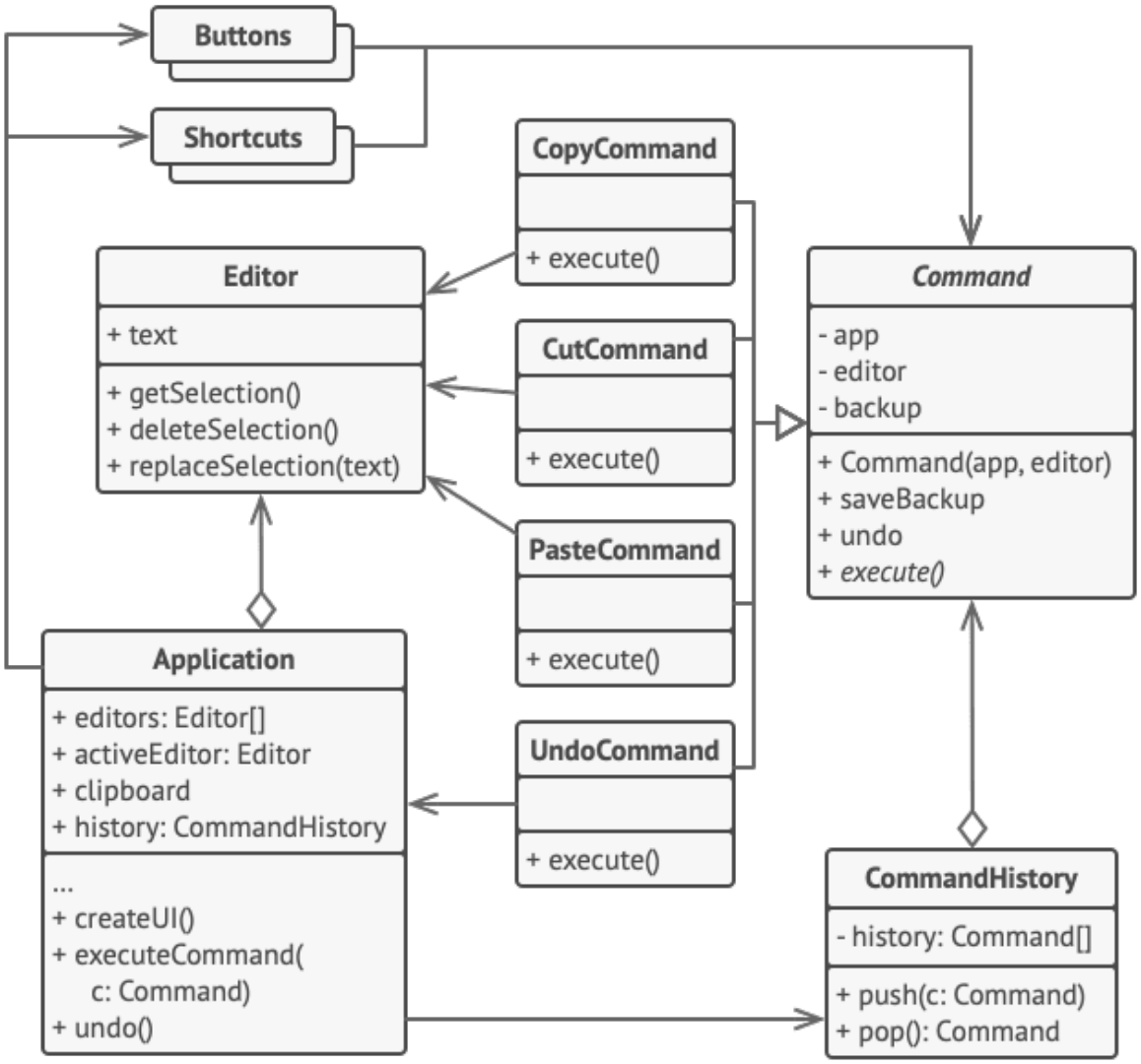
1. A classe **Remetente** (também conhecida como *invocadora*) é responsável por iniciar os pedidos. Essa classe deve ter um campo para armazenar a referência para um objeto comando. O remetente aciona aquele comando ao invés de enviar o pedido diretamente para o destinatário. Observe que o remetente não é responsável por criar o objeto comando. Geralmente ele é pré criado através de um construtor do cliente.
2. A interface **Comando** geralmente declara apenas um único método para executar o comando.
3. **Comandos Concretos** implementam vários tipos de pedidos. Um comando concreto não deve realizar o trabalho por conta própria, mas passar a chamada para um dos objetos da lógica do negócio. Contudo, para simplificar o código, essas classes podem ser fundidas.

   Os parâmetros necessários para executar um método em um objeto destinatário podem ser declarados como campos no comando concreto. Você pode tornar os objetos comando imutáveis ao permitir que apenas inicializem esses campos através do construtor.
4. A classe **Destinatária** contém a lógica do negócio. Quase qualquer objeto pode servir como um destinatário. A maioria dos comandos apenas lida com os detalhes de como um pedido é passado para o destinatário, enquanto que o destinatário em si executa o verdadeiro trabalho.
5. O **Cliente** cria e configura objetos comando concretos. O cliente deve passar todos os parâmetros do pedido, incluindo uma instância do destinatário, para o construtor do comando. Após isso, o comando resultante pode ser associado com um ou

múltiplos destinatários.

Pseudocódigo

Neste exemplo, o padrão **Command** ajuda a manter um registro da história de operações executadas e torna possível reverter uma operação se necessário.



Operações não executáveis em um editor de texto.

Os comandos que resultam de mudanças de estado do editor (por exemplo, cortando e colando) fazem uma cópia de backup do estado do editor antes de executarem uma operação associada com o comando. Após o comando ser executado, ele é colocado

em um histórico de comando (uma pilha de objetos comando) junto com a cópia de backup do estado do editor naquele momento. Mais tarde, se o usuário precisa reverter uma operação, a aplicação pode pegar o comando mais recente do histórico, ler o backup associado ao estado do editor, e restaurá-lo.

O código cliente (elementos GUI, histórico de comandos, etc.) não é acoplado às classes comando concretas porque ele trabalha com comandos através da interface comando. Essa abordagem permite que você introduza novos comandos na aplicação sem quebrar o código existente.

```
// A classe comando base define a interface comum para todos
// comandos concretos.
abstract class Command is
    protected field app: Application
    protected field editor: Editor
    protected field backup: text

    constructor Command(app: Application, editor: Editor) is
        this.app = app
        this.editor = editor

    // Faz um backup do estado do editor.
    method saveBackup() is
        backup = editor.text

    // Restaura o estado do editor.
    method undo() is
        editor.text = backup
```

```
    // O método de execução é declarado abstrato para forçar
    // todos os comandos concretos a fornecer suas próprias
    // implementações. O método deve retornar verdadeiro ou
    // falso dependendo se o comando muda o estado do editor.
    abstract method execute()


// Comandos concretos vêm aqui.
class CopyCommand extends Command is
    // O comando copy (copiar) não é salvo no histórico já que
    // não muda o estado do editor.
    method execute() is
        app.clipboard = editor.getSelection()
        return false

class CutCommand extends Command is
    // O comando cut (cortar) muda o estado do editor, portanto
    // deve ser salvo no histórico. E ele será salvo desde que o
    // método retorne verdadeiro.
    method execute() is
        saveBackup()
        app.clipboard = editor.getSelection()
        editor.deleteSelection()
        return true

class PasteCommand extends Command is
    method execute() is
        saveBackup()
        editor.replaceSelection(app.clipboard)
        return true
```

```
// A operação undo (desfazer) também é um comando.
class UndoCommand extends Command is
    method execute() is
        app.undo()
        return false


// O comando global history (histórico) é apenas uma pilha.
class CommandHistory is
    private field history: array of Command

    // Último a entrar...
    method push(c: Command) is
        // Empurra o comando para o fim do vetor do histórico.

    // ...primeiro a sair.
    method pop():Command is
        // Obter o comando mais recente do histórico.


// A classe do editor tem verdadeiras operações de edição de
// texto. Ela faz o papel de destinatária: todos os comandos
// acabam delegando a execução para os métodos do editor.
class Editor is
    field text: string

    method getSelection() is
        // Retorna o texto selecionado.
```

```
    method deleteSelection() is
        // Deleta o texto selecionado.

    method replaceSelection(text) is
        // Insere os conteúdos da área de transferência na
        // posição atual.


// A classe da aplicação define as relações de objeto. Ela age
// como uma remetente: quando alguma coisa precisa ser feita,
// ela cria um objeto comando e executa ele.
class Application is
    field clipboard: string
    field editors: array of Editors
    field activeEditor: Editor
    field history: CommandHistory

    // O código que assinala comandos para objetos UI pode se
    // parecer como este.
    method createUI() is

        copy = function() { executeCommand(
            new CopyCommand(this, activeEditor)) }
        copyButton.setCommand(copy)
        shortcuts.onKeyPress("Ctrl+C", copy)

        cut = function() { executeCommand(
            new CutCommand(this, activeEditor)) }
        cutButton.setCommand(cut)
        shortcuts.onKeyPress("Ctrl+X", cut)
```

```
        paste = function() { executeCommand(
            new PasteCommand(this, activeEditor)) }
        pasteButton.setCommand(paste)
        shortcuts.onKeyPress("Ctrl+V", paste)

        undo = function() { executeCommand(
            new UndoCommand(this, activeEditor)) }
        undoButton.setCommand(undo)
        shortcuts.onKeyPress("Ctrl+Z", undo)

    // Executa um comando e verifica se ele foi adicionado ao
    // histórico.
    method executeCommand(command) is
        if (command.execute())
            history.push(command)

    // Pega o comando mais recente do histórico e executa seu
    // método undo(desfazer). Observe que nós não sabemos a
    // classe desse comando. Mas nós não precisamos saber, já
    // que o comando sabe como desfazer sua própria ação.
    method undo() is
        command = history.pop()
        if (command != null)
            command.undo()
```

Aplicabilidade

Utilize o padrão Command quando você quer parametrizar objetos com operações.

O padrão Command podem tornar uma chamada específica para um método em um objeto separado. Essa mudança abre várias possibilidades de usos interessantes: você pode passar comandos como argumentos do método, armazená-los dentro de outros objetos, trocar comandos ligados no momento de execução, etc.

Aqui está um exemplo: você está desenvolvendo um componente GUI como um menu de contexto, e você quer que os usuários sejam capazes de configurar os items do menu que aciona as operações quando um usuário clica em um item.

Utilize o padrão Command quando você quer colocar operações em fila, agendar sua execução, ou executá-las remotamente.

Como qualquer outro objeto, um comando pode ser serializado, o que significa convertê-lo em uma string que pode ser facilmente escrita em um arquivo ou base de dados. Mais tarde a string pode ser restaurada no objeto comando inicial. Dessa forma você pode adiar e agendar execuções do comando. Mas isso não é tudo! Da mesma forma, você pode colocar em fila, fazer registro de log ou enviar comandos por uma rede.

Utilize o padrão Command quando você quer implementar operações reversíveis.

Embora haja muitas formas de implementar o desfazer/refazer, o padrão Command é talvez a mais popular de todas.

Para ser capaz de reverter operações, você precisa implementar o histórico de operações realizadas. O histórico do comando é uma pilha que contém todos os objetos comando executados junto com seus backups do estado da aplicação relacionados.

Esse método tem duas desvantagens. Primeiro, se não for fácil salvar o estado da aplicação por parte dela ser privada. Esse problema pode ser facilmente mitigado com o padrão Memento.

Segundo, os backups de estado podem consumir uma considerável quantidade de RAM. Portanto, algumas vezes você pode recorrer a uma implementação alternativa: ao invés de restaurar a um estado passado, o comando faz a operação inversa. A operação reversa também cobra um preço: ela pode ter sua implementação difícil ou até impossível.

Como implementar

1. Declare a interface comando com um único método de execução.
2. Comece extraindo pedidos para dentro de classes concretas comando que implementam a interface comando. Cada classe deve ter um conjunto de campos para armazenar os argumentos dos pedidos junto com uma referência ao objeto destinatário real. Todos esses valores devem ser inicializados através do construtor do comando.
3. Identifique classes que vão agir como *remetentes*. Adicione os campos para armazenar comandos nessas classes. Remetentes devem sempre comunicar-se com seus comandos através da interface comando. Remetentes geralmente não criam objetos comando por conta própria, mas devem obtê-los do código cliente.
4. Mude os remetentes para que executem o comando ao invés de enviar o pedido para o destinatário diretamente.
5. O cliente deve inicializar objetos na seguinte ordem:

- Crie os destinatários.

- Crie os comandos, e os associe com os destinatários se necessário.
- Crie os remetentes, e os associe com os comandos específicos.

Prós e contras

- *Princípio de responsabilidade única*. Você pode desacoplar classes que invocam operações de classes que fazem essas operações.
- *Princípio aberto/fechado*. Você pode introduzir novos comandos na aplicação sem quebrar o código cliente existente.
- Você pode implementar desfazer/refazer.
- Você pode implementar a execução adiada de operações.
- Você pode montar um conjunto de comandos simples em um complexo.
- O código pode ficar mais complicado uma vez que você está introduzindo uma nova camada entre remetentes e destinatários.

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.
- 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.
- Você pode usar o Command e o Memento juntos quando implementando um “desfazer”. Neste caso, os comandos são responsáveis pela realização de várias operações sobre um objeto alvo, enquanto que os mementos salvam o estado daquele objeto momentos antes de um comando ser executado.
- O Command e o Strategy podem ser parecidos porque você pode usar ambos para parametrizar um objeto com alguma ação. Contudo, eles têm propósitos bem diferentes.
- Você pode usar o *Command* para converter qualquer operação em um objeto. Os parâmetros da operação se transformam em campos daquele objeto. A conversão permite que você atrase a execução de uma operação, transforme-a em uma fila, armazene o histórico de comandos, envie comandos para serviços remotos, etc.
- Por outro lado, o *Strategy* geralmente descreve diferentes

maneiras de fazer a mesma coisa, permitindo que você troque esses algoritmos dentro de uma única classe contexto.

- O Prototype pode ajudar quando você precisa salvar cópias de comandos no histórico.
- Você pode tratar um Visitor como uma poderosa versão do padrão Command. Seus objetos podem executar operações sobre vários objetos de diferentes classes.