Padrão Command

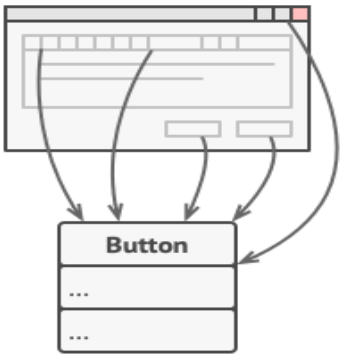
Alex Soares
Gleiceara Azevedo
Sander Piva

1. Definição, características

O Command é um padrão de projeto comportamental cujo objetivo é transformar um pedido em objeto independente que contém toda a informação sobre esse pedido. Esse procedimento permite que parametrize métodos com diferentes pedidos, atrase ou coloque a execução do pedido em fila, e suporte operações que não sejam feitas.
Um objeto comando encapsula uma solicitação vinculando um conjunto de ações em um receptor específico. Para fazer isso, ele empacota as ações e o receptor em um objeto que expõe um único método execute (). Quando execute() é chamado, as ações são vinculadas ao receptor. Externamente nenhum outro objeto sabe realmente quais ações são executadas em qual receptor; eles apenas sabem que suas solicitações são atendidas quando o método execute() é acionado.

2. Problema

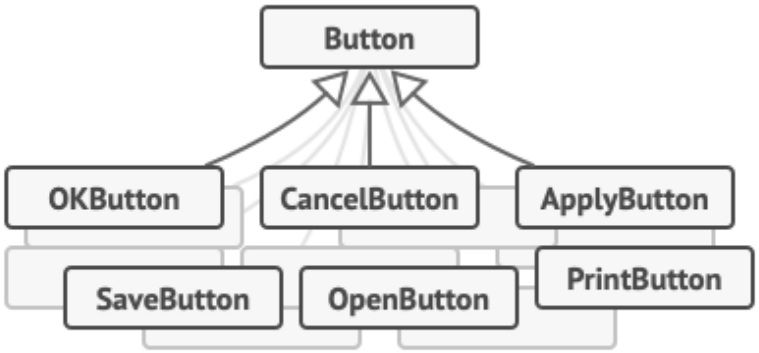
Imagine que você está trabalhando em uma nova aplicação de editor de texto. Sua tarefa atual é criar uma barra de tarefas com vários botões para várias operações do editor. Você criou uma classe Botão muito bacana que pode ser usada para botões na barra de tarefas, bem como para botões genéricos de diversas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



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

Fig.1

Embora todos esses botões pareçam iguais, todos devem fazer coisas diferentes. Onde deveria colocar o código para os vários handlers (manipuladores) desses botões? A solução mais simples é criar várias 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?

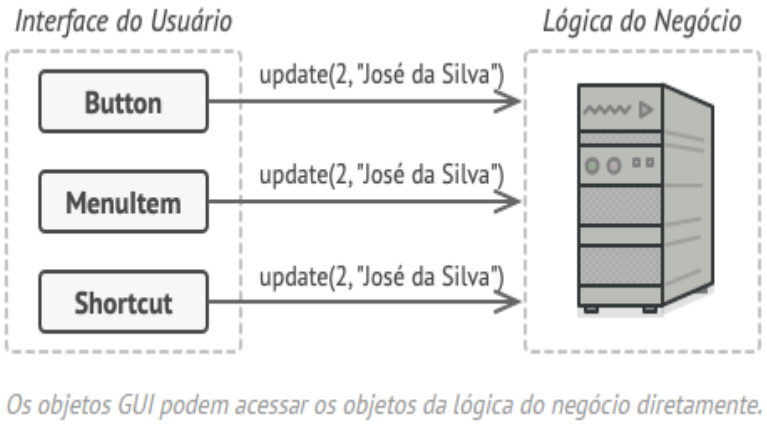
Fig.2

Como logo pode perceber, existem alguns problemas nessa abordagem. Por exemplo, existem muitas subclasses herdando da classe base Botão, ou seja, o acoplamento é alto. Qualquer alteração na classe base Botão provocaria

um efeito cascata de refatoração nas subclasses herdeiras. Operações como copiar/colar texto, precisariam ser invocadas de diversos lugares, ou seja, um usuário poderia criar um pequeno botão “Copiar” na barra de ferramentas, ou copiar algo por meio do menu de contexto, ou ainda apenas apertando o Ctrl + C do teclado.
À medida que a implementação aumenta com menus de contextos, atalhos, e outras coisas, há necessidade de duplicar o código da operação em muitas classes ou fazer menus dependentes de botões, o que é ainda pior.

3. Solução

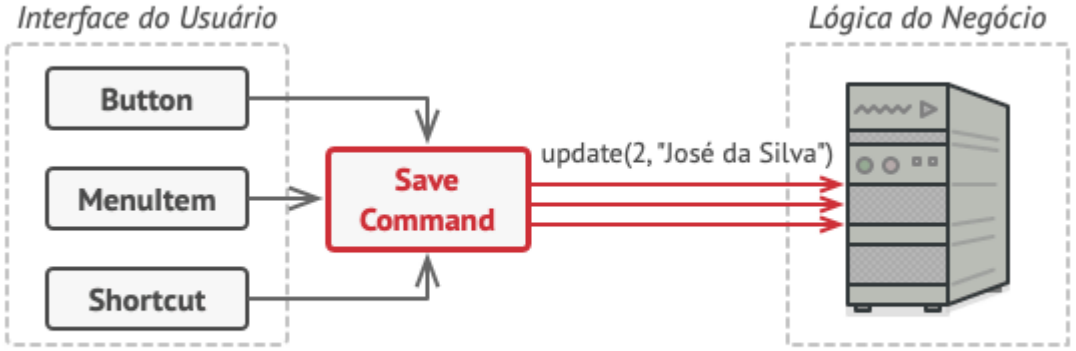
Um bom projeto de software quase sempre se norteia no princípio da separação de interesses, o que costuma provocar a divisão da aplicação em camadas. Uma situação: uma camada para interface gráfica do usuário (GUI) e outra para a lógica de negócio. A GUI fica responsável por renderizar uma bonita imagem na tela capaz de capturar quaisquer dados e mostrar resultados do que o usuário e a aplicação estão fazendo. Todavia, 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 aparecer assim: um objeto GUI chama um método da lógica de 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.

Fig.3

O padrão Command recomenda que os objetos GUI não enviem esses pedidos diretamente. Ao invés disso, deve ser extraído 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.

Fig.4

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 usar inúmeros comandos com o mesmo remetente do pedido, sem acoplá-lo com as classes concretas dos comandos. Como um bônus, agora é possível trocar os objetos comando ligados ao remetente, efetivamente mudando o comportamento do remetente no momento da execução.

Porém, há 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.

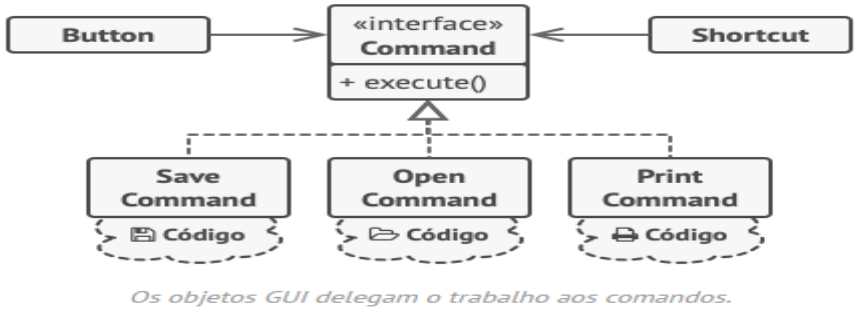


Fig.4

Voltando ao exemplo do editor de texto. Após aplicar o padrão Command, não será necessário todas aquelas subclasses de botões para implementar vários comportamentos de cliques. É suficiente colocar apenas um campo na classe base Botão que armazena a referência para um objeto comando e faz o botão executar aquele comando com um clique.
Será implementada várias classes comando para cada possível operação, ligando-os 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. Esses serão ligados a um comando que será executado quando um usuário interage com um elemento GUI. 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.
As principais aplicações do padrão Command são para:

1- Parametrizar objetos com operações;
2- Colocar operações em fila, agendar sua execução, ou executá-las remotamente;
3- Implementar operações reversíveis;

4. 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.

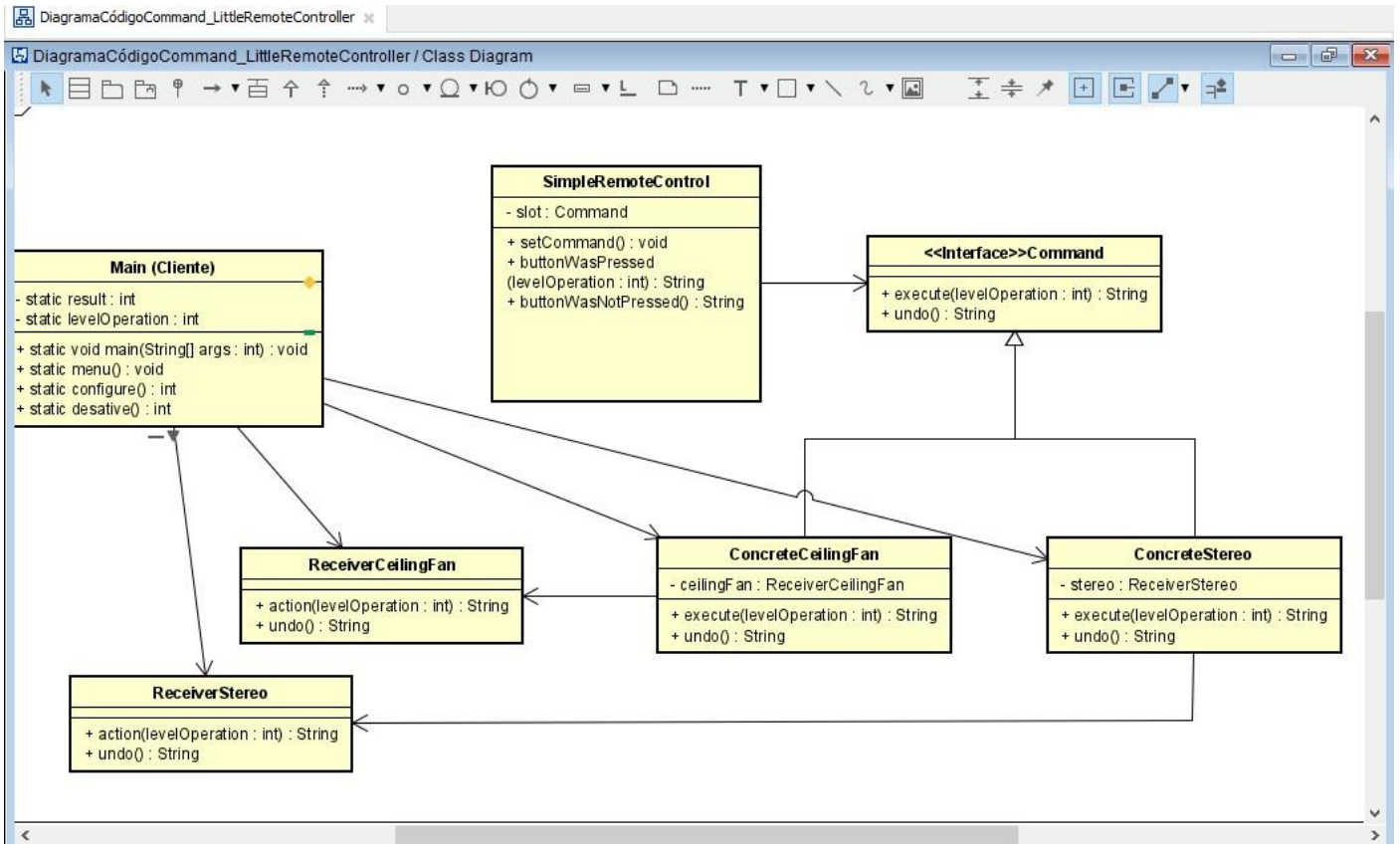
5. Prós e contras

- *Princípio de responsabilidade única*. Permite desacoplar classes que invocam operações de classes que fazem essas operações.
- *Princípio aberto/fechado*. Introduz novos comandos na aplicação sem quebrar o código cliente existente.
- Implementa desfazer/refazer.
- Implementa a execução adiada de operações.
- Desenvolve 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.

6. 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, 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, pode executar a mesma operação em uma série de diferentes contextos ligados em uma corrente.
- É possível 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.
    - Permite 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 precisa salvar cópias de comandos no histórico.
- Um Visitor pode ser uma poderosa versão do padrão Command. Seus objetos podem executar operações sobre vários objetos de diferentes classes.

7. Diagrama de classe do código implementado pelo grupo: LittleRemoteController



8. Conclusão

O Command é um padrão de projeto comportamental empregado para transformar um pedido em objeto independente que contém toda a informação sobre esse pedido. Esse procedimento permite que parametrize métodos com diferentes pedidos, atrase ou coloque a execução do pedido em fila, e suporte operações que não sejam feitas.

Um objeto comando encapsula uma solicitação vinculando um conjunto de ações em um receptor específico. Externamente nenhum outro objeto sabe realmente quais ações são executadas em qual receptor; eles apenas sabem que suas solicitações são atendidas quando o método execute(), presente na interface Command, é acionado.

Apresenta inúmeras vantagens como baixo acoplamento, alto encapsulamento, enfileiramento de operações, operações de desfazer e refazer, além de permitir fácil extensão e manipulação do código. No entanto, principalmente entre os programadores iniciantes, a aplicação do padrão Command pode gerar um pouco de dúvida quanto a maneira como as relações dentro do código são estabelecidas.

9. Referências

FREEMAN, Eric, Freeman, Elizabeth. Use a Cabeça Padrões de Projeto. 2ª Edição Revisada. Alta Books. Rio de Janeiro. 2009. Páginas: 151,152, 154, 155, 156, 157, 170, 171.

??

https://refactoring.guru/pt-br/design-patterns/command

livro

?? video