# COMMAND comportamental de objetos

## Intenção

Encapsular uma solicitação como um objeto, desta forma permitindo parametrizar clientes com diferentes solicitações, enfileirar ou fazer o registro *(log)* de solicitações e suportar operações que podem ser desfeitas.

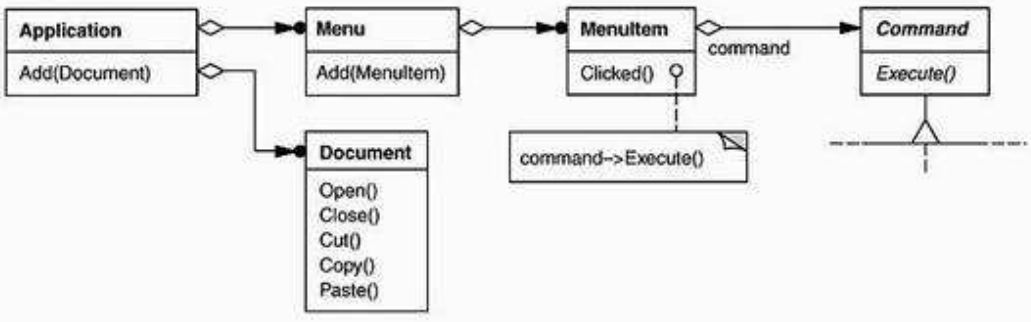
## Também conhecido como

Action, Transaction

## Motivação

Algumas vezes é necessário emitir solicitações para objetos sem nada saber sobre a operação que está sendo solicitada ou sobre o seu receptor. Por exemplo, *toolkits* para construção de interfaces de usuário incluem objetos como botões de menus que executam uma solicitação em resposta à entrada do usuário. Mas o *toolkit* não pode implementar a solicitação explicitamente no botão ou menu porque somente as aplicações que utilizam o *toolkit* sabem o que deveria ser feito e em qual objeto. Como projetistas de *toolkits*, não temos meios de saber qual o receptor da solicitação ou as operações que ele executará.

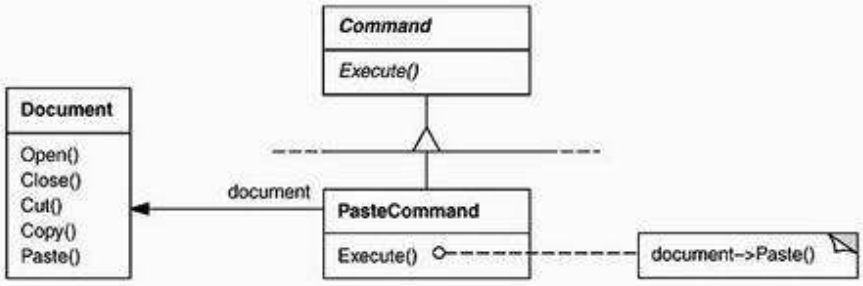
O padrão Command permite a objetos de *toolkit* fazer solicitações de objetos-aplicação não especificados, transformando a própria solicitação num objeto. Esse objeto pode ser armazenado e passado como outros objetos. A chave desse padrão é uma classe abstrata Command, a qual declara uma interface para execução de operações. Na sua forma mais simples, essa interface inclui uma operação abstrata Execute. As subclasses concretas de Command especificam um par receptor-ação através do armazenamento do receptor como uma variável de instância e pela implementação de Execute para invocar a solicitação. O receptor tem o conhecimento necessário para poder executar a solicitação.



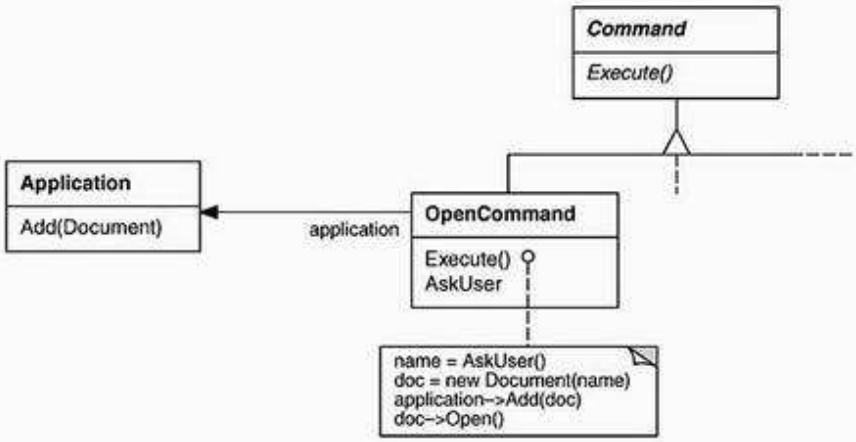

Menus podem ser implementados facilmente com objetos Command. Cada escolha num Menu é uma instância de uma classe MenuItem. Uma classe Application cria esses menus e seus itens de menus juntamente com o resto da interface do usuário. A classe Application também mantém um registro de acompanhamento dos objetos Document que um usuário abriu.

A aplicação configura cada MenuItem com uma instância de uma subclasse concreta de Command. Quando o usuário seleciona um MenuItem, o MenuItem chama Execute no seu Command, e Execute executa a operação. MenuItens não sabem qual a subclasse de Command que usam. As subclasses de Command armazenam o receptor da solicitação que invoca uma ou mais operações no receptor.

Por exemplo, um PasteCommand suporta colar textos da área de transferência (*clipboard*) num Document. O receptor de PasteCommand é o objeto Document que é fornecido por instanciação. A operação Execute invoca Paste no Document que está recebendo.

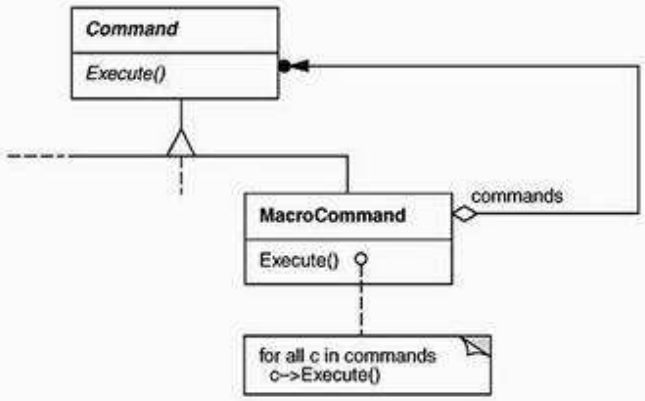



A operação Execute do OpenCommand, é diferente: ela solicita ao usuário o nome de um documento, cria o correspondente objeto Document, adiciona este documento à aplicação receptora e abre o documento.




Algumas vezes, um MenuItem necessita executar uma *seqüência* de comandos. Por exemplo, um MenuItem para centralizar uma página, no tamanho normal, poderia ser construído a partir de um objeto CenterDocumentCommand e de um objeto NormalSizeCommand. Como é comum encadear comandos desta forma, nós podemos definir uma classe MacroCommand para permitir que um MenuItem execute um número aberto de comandos.

O MacroCommand é uma subclasse concreta de Command que simplesmente executa uma seqüência de Commands. O MacroCommand não tem um receptor explícito, porque os comandos que ele seqüencia definem seu próprio receptor.




Observe em cada um destes exemplos como o padrão Command desacopla o objeto que invoca a operação daquele que tem o conhecimento para executá-la. Isso nos dá bastante flexibilidade no projeto da nossa interface de usuário. Uma aplicação pode oferecer tanto uma interface com menus como uma interface com botões para algum recurso seu, simplesmente fazendo com que o menu e o botão compartilhem uma instância da mesma subclasse concreta Command. Nós podemos substituir comandos dinamicamente, o que poderia ser útil para a implementação de menus sensíveis ao contexto. Também podemos suportar *scripts* de comandos compondo comandos em comandos maiores. Tudo isto é possível porque o objeto que emite a solicitação somente necessita saber como emiti-la; ele não necessita saber como a solicitação será executada.

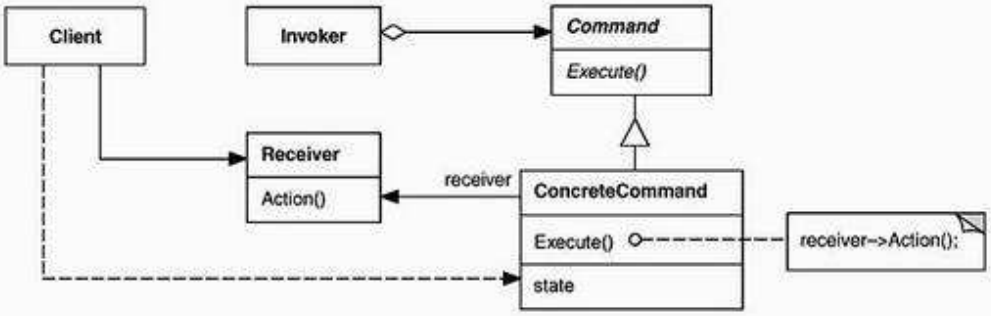
## Aplicabilidade

Use o padrão Command quando você deseja:

- parametrizar objetos por uma ação a ser executada, da forma como os objetos MenuItem fizeram acima. Você pode expressar tal parametrização numa linguagem procedural através de uma função ***callback***, ou seja, uma função que é registrada em algum lugar para ser chamada em um momento mais adiante. Os Commands são uma substituição orientada o objetos para *callbacks*;
- especificar, enfileirar e executar solicitações em tempos diferentes. Um objeto Command pode ter um tempo de vida independente da solicitação orginal. Se o receptor de uma solicitação pode ser representado de uma maneira independente do espaço de endereçamento, então você pode transferir um objeto command para a solicitação para um processo diferente e lá atender a solicitação;
- suportar desfazer operações. A operação Execute, de Command, pode armazenar estados para reverter seus efeitos no próprio comando. A interface de Command deve ter acrescentada uma operação Unexecute, que reverte os efeitos de uma chamada anterior de Execute. Os comandos executados são armazenados em uma lista histórica. O nível ilimitado de desfazer e refazer operações é obtido percorrendo esta lista para trás e para frente, chamando operações Unexecute e Execute, respectivamente;

- suportar o registro (*logging*) de mudanças de maneira que possam ser reaplicadas no caso de uma queda de sistema. Ao aumentar a interface de Command com as operações carregar e armazenar, você pode manter um registro (*log*) persistente das mudanças. A recuperação de uma queda de sistema envolve a recarga dos comandos registrados a partir do disco e sua reexecução com a operação Execute.
- estruturar um sistema em torno de operações de alto nível construídas sobre operações primitivas. Tal estrutura é comum em sistemas de informação que suportam **transações**. Uma transação encapsula um conjunto de mudanças nos dados. O padrão Command fornece uma maneira de modelar transações. Os Commands têm uma interface comum, permitindo invocar todas as transações da mesma maneira. O padrão também torna mais fácil estender o sistema com novas transações.

## Estrutura




## Participantes

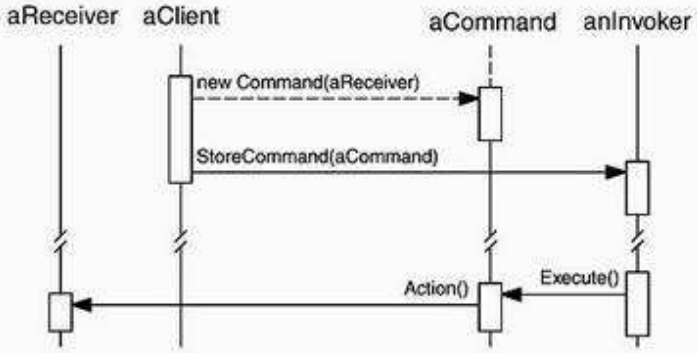
- **Command**
  - declara uma interface para a execução de uma operação.
- **ConcreteCommand** (PasteCommand, OpenCommand)
  - define uma vinculação entre um objeto Receiver e uma ação;
  - implementa Execute através da invocação da(s) correspondente(s) operação(ões) no Receiver.
- **Client** (Application)
  - cria um objeto ConcreteCommand e estabelece o seu receptor.
- **Invoker** (MenuItem)
  - solicita ao Command a execução da solicitação.
- **Receiver** (Document, Application)
  - sabe como executar as operações associadas a uma solicitação. Qualquer classe pode funcionar como um Receiver.

## Colaborações

- O cliente cria um objeto ConcreteCommand e especifica o seu receptor.
- Um objeto Invoker armazena o objeto ConcreteCommand.
- O Invoker emite uma solicitação chamando Execute no Command. Quando se deseja que os comandos possam ser desfeitos, ConcreteCommand armazena estados para desfazer o comando antes de invocar Execute.

- O objeto ConcreteCommand invoca operações no seu Receiver para executar a solicitação.

O diagrama a seguir mostra as interações entre esses objetos, ilustrando como Command desacopla o Invoker do Receiver (e da solicitação que ele executa).




## Conseqüências

O padrão Command tem as seguintes conseqüências:

1. Command desacopla o objeto que invoca a operação daquele que sabe como executá-la.
2. Commands são objetos de primeira classe, ou seja, podem ser manipulados e estendidos como qualquer outro objeto.
3. Você pode montar comandos para formar um comando composto. Um exemplo disso é a classe MacroCommand descrita anteriormente. Em geral, comandos compostos são uma instância do padrão Composite (160).
4. É fácil acrescentar novos Commands porque você não tem que mudar classes existentes.

## Implementação

Considere os seguintes aspectos quando implementar o padrão Command:

1. *Quão inteligente deveria ser um comando?* Um comando pode ter uma grande gama de habilidades. Em um extremo mais simples, ele define uma vinculação entre um receptor e as ações que executam a solicitação. No outro extremo, o mais complexo, ele implementa tudo sozinho, sem delegar para nenhum receptor. Este último caso extremo é útil quando você deseja definir comandos que são independentes de classes existentes, quando não existe um receptor adequado ou quando um comando conhece o seu receptor implicitamente. Por exemplo, um comando que cria uma outra janela de aplicação pode ser tão capaz de criar uma janela como qualquer outro objeto. Em algum ponto entre esses dois extremos estão os comandos que têm conhecimento suficiente para encontrar o seu receptor dinamicamente.
2. *Suportando desfazer e refazer.* Commands podem suportar capacidades de desfazer e refazer se eles fornecerem uma maneira de reverter sua execução (por exemplo, uma operação Unexecute ou Undo). Uma classe

ConcreteCommand pode necessitar armazenar estados adicionais para fazer isso. Esses estados podem incluir:

- o objeto Receptor (*Receiver*), o qual efetivamente executa as operações em resposta à solicitação;
- os argumentos da operação executada no receptor;
- quaisquer valores originais no receptor que podem mudar como resultado do tratamento da solicitação. O receptor deve fornecer operações que permitem ao comando retornar o receptor ao seu estado anterior.

Para suportar um nível apenas de desfazer, uma aplicação necessita armazenar somente o último comando executado. Para suportar múltiplos níveis de desfazer e refazer, a aplicação necessita uma **lista histórica** de comandos que foram executados onde o máximo comprimento da lista determina o número de níveis de desfazer/refazer. A lista histórica armazena seqüências de comandos que foram executados. Percorrendo a lista para trás e executando de maneira reversa os comandos, cancelam-se os seus efeitos; percorrendo a lista para frente e executando os comandos, reexecuta-se os comandos a serem refeitos.

Um comando que pode ser desfeito poderá ter que ser copiado, antes de ser colocado na lista histórica. Isso se deve ao fato de que o objeto comando que executou a solicitação original, digamos, a partir de um MenuItem, executará outras solicitações em instantes posteriores. A cópia é necessária para distinguir diferentes invocações do mesmo comando se o seu estado pode variar entre invocações. Por exemplo, um DeleteCommand que deleta objetos selecionados deve armazenar diferentes conjuntos de objetos, cada vez que é executado. Portanto, o objeto DeleteCommand deve ser copiado logo após a execução e ter a cópia colocada na lista histórica. Se na execução o estado do comando nunca muda, então a cópia não é necessária – somente uma referência para o comando necessita ser colocada na lista histórica. Commands que devem ser copiados antes de serem colocados na lista histórica se comportam como protótipos (ver Prototype, 121).

3. *Evitando a acumulação de erros no processo de desfazer.* Histerese pode ser um problema ao tentarmos garantir um mecanismo confiável de desfazer/refazer que preserve a semântica da aplicação. Erros podem se acumular à medida que os comandos são executados, desexecutados e reexecutados repetidamente, de modo que o estado de uma aplicação eventualmente poderia divergir dos valores originais. Portanto, pode ser necessário armazenar mais informações no comando para assegurar que os objetos sejam restaurados ao seu estado original. O padrão Memento pode ser aplicado para dar ao comando acesso a essas informações, sem expor aspectos internos de outros objetos.
4. *Usando templates C++.* Para comandos que (1) não possam ser desfeitos e (2) não exijam argumentos, podemos usar *templates* em C++ a fim de evitar a criação de uma subclasse de Command para cada tipo de ação e receptor. Mostraremos como fazer isto na seção Exemplo de Código.

## Exemplo de código

O código C++ mostrado aqui esboça a implementação das classes Command da seção de Motivação. Nós definiremos `OpenCommand`, `PasteCommand` e `MacroCommand`. Primeiramente, a classe abstrata `Command`:

```
class Command {
public:
    virtual ~Command();

    virtual void Execute() = 0;
protected:
    Command();
};
```

Um `OpenCommand` abre um documento cujo nome é fornecido pelo usuário. Para um `OpenCommand` deve ser passado um objeto `Application` no seu constructor. `AskUser` é uma rotina de implementação que solicita ao usuário o nome do documento a ser aberto.

```
class OpenCommand : public Command {
public:
    OpenCommand(Application*);

    virtual void Execute();
protected:
    virtual const char* AskUser();
private:
    Application* _application;
    char* _response;
};

OpenCommand::OpenCommand (Application* a) {
    _application = a;
}

void OpenCommand::Execute () {
    const char* name = AskUser();

    if (name != 0) {
        Document* document = new Document(name);
        _application->Add(document);
        document->Open();
    }
}
```

Para um `PasteCommand` deve ser passado um objeto `Document` como seu receptor. O receptor é fornecido como um parâmetro para o constructor de `PasteCommand`.

```
class PasteCommand : public Command {
public:
    PasteCommand(Document*);

    virtual void Execute();
private:
    Document* _document;
};

PasteCommand::PasteCommand (Document* doc) {
    _document = doc;
}

void PasteCommand::Execute () {
    _document->Paste();
}
```

Para comandos simples, que não podem ser desfeitos e não necessitam de argumentos, nós podemos usar um *template* de uma classe para parametrizar o receptor do comando. Definiremos uma subclasse *template* `SimpleCommand` para tais comandos. O `SimpleCommand` é parametrizado pelo tipo do `Receiver` e mantém uma vinculação entre um objeto receptor e uma ação armazenada como um apontador para uma função-membro.

```
template <class Receiver>
class SimpleCommand : public Command {
public:
    typedef void (Receiver::* Action)();

    SimpleCommand(Receiver* r, Action a) :
        _receiver(r), _action(a) { }

    virtual void Execute();
private:
    Action _action;
    Receiver* _receiver;
};
```

O construtor armazena o receptor e a ação nas variáveis de instância correspondentes. `Execute` simplesmente aplica a ação ao receptor.

```
template <class Receiver>
void SimpleCommand<Receiver>::Execute () {
    (_receiver->*_action)();
}
```

Para criar um comando que chama `Action` numa instância da classe `MyClass`, um cliente simplesmente escreve

```
MyClass* receiver = new MyClass;
// ...
Command* aCommand =
    new SimpleCommand<MyClass>(receiver, &MyClass::Action);
// ...
aCommand->Execute();
```

Tenha em mente que essa solução funciona somente para comandos simples. Comandos mais complexos, que mantêm controle não somente de seus receptores mas também de argumentos ou estados para desfazer, exigem uma subclasse `Command`.

Um `MacroCommand` administra uma seqüência de subcomandos e fornece operações para acrescentar e remover subcomandos. Não é necessário um receptor explícito porque estes subcomandos já definem seus receptores.

```
class MacroCommand : public Command {
public:
    MacroCommand();
    virtual ~MacroCommand();

    virtual void Add(Command*);
    virtual void Remove(Command*);

    virtual void Execute();
private:
    List<Command*>* _cmds;
};
```

A chave para o `MacroCommand` é a sua função-membro `Execute`. Ela percorre todos os subcomandos e executa `Execute` em cada um deles.

```
void MacroCommand::Execute () {
    ListIterator<Command*> i(_cmds);

    for (i.First(); !i.IsDone(); i.Next()) {
        Command* c = i.CurrentItem();
        c->Execute();
    }
}
```

Note que se `MacroCommand` implementasse uma operação `Unexecute`, então seus subcomandos deveriam ser revertidos na ordem *inversa* à da implementação de `Execute`.

Por fim, `MacroCommand` deve fornecer operações para administrar seus subcomandos. A `MacroCommand` também é responsável por deletar seus subcomandos.

```
void MacroCommand::Add (Command* c) {
    _cmds->Append(c);
}

void MacroCommand::Remove (Command* c) {
    _cmds->Remove(c);
}
```

## Usos conhecidos

Talvez o primeiro exemplo do *padrão* Command tenha aparecido em um artigo de Lieberman [Lie85]. O MacApp [App89] popularizou a noção de comandos para implementação de operações que podem ser desfeitas. O ET++ [WGM88], o Inter Views [LCI+92] e o Unidraw [VL90] também definem classes que seguem o *padrão* Command. InterViews define uma classe abstrata Action que fornece a funcionalidade de comando. Ele também define um *template* ActionCallback, parametrizado pelo método ação, que instancia automaticamente subclasses Command.

A biblioteca de classes THINK [Sym93b] também utiliza comandos para suportar ações que podem ser desfeitas. Comandos em THINK são chamados "Tasks" (tarefas). Os objetos Tasks são passados ao longo de uma Chain of Responsibility (212) para serem consumidos.

Os objetos de comando de Unidraw são únicos no sentido de que eles se comportam como mensagens. Um comando Unidraw pode ser enviado a outro objeto para interpretação, e o resultado da interpretação varia de acordo com o objeto receptor. Além do mais, o receptor pode delegar a interpretação para um outro objeto, normalmente o pai do receptor, numa estrutura maior como a de uma Chain of Responsibility. Assim, o receptor de um comando Unidraw é computado, em vez de armazenado. O mecanismo de interpretação do Unidraw depende de informações de tipo em tempo de execução.

Coplien descreve como implementar **functors**, objetos que são funções em C++ [Cop92]. Ele obtém um grau de transparência no seu uso através da sobrecarga do operador de chamada (`operator ()`). O *padrão* Command é diferente; o seu foco é a manutenção de um *vínculo entre* um receptor e uma função (isto é, uma ação), e não somente a manutenção de uma função.

## Padrões relacionados

Um Composite (160) pode ser usado para implementar MacroCommands.
Um Memento (266) pode manter estados que o comando necessita para desfazer o seu efeito.
Um comando que deve ser copiado antes de ser colocado na lista histórica funciona como um Prototype (121).

# INTERPRETER — comportamental de classes

## Intenção

Dada uma linguagem, definir uma representação para a sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças dessa linguagem.

## Motivação

Se um tipo específico de problema ocorre com freqüência suficiente, pode valer a pena expressar instâncias do problema como sentenças de uma linguagem simples. Então, você pode construir um interpretador que soluciona o problema interpretando estas sentenças. Por exemplo, pesquisar cadeias de caracteres que correspondem a um determinado padrão (*pattern matching*) é um tipo de problema comum. Expressões regulares são uma linguagem-padrão para especificação de padrões de cadeias de caracteres. Em vez de construir algoritmos customizados para comparar cada padrão com as cadeias, algoritmos de busca poderiam interpretar uma expressão regular que especifica um conjunto de cadeias a serem encontradas.