Command

Objetivo

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

Também conhecido como

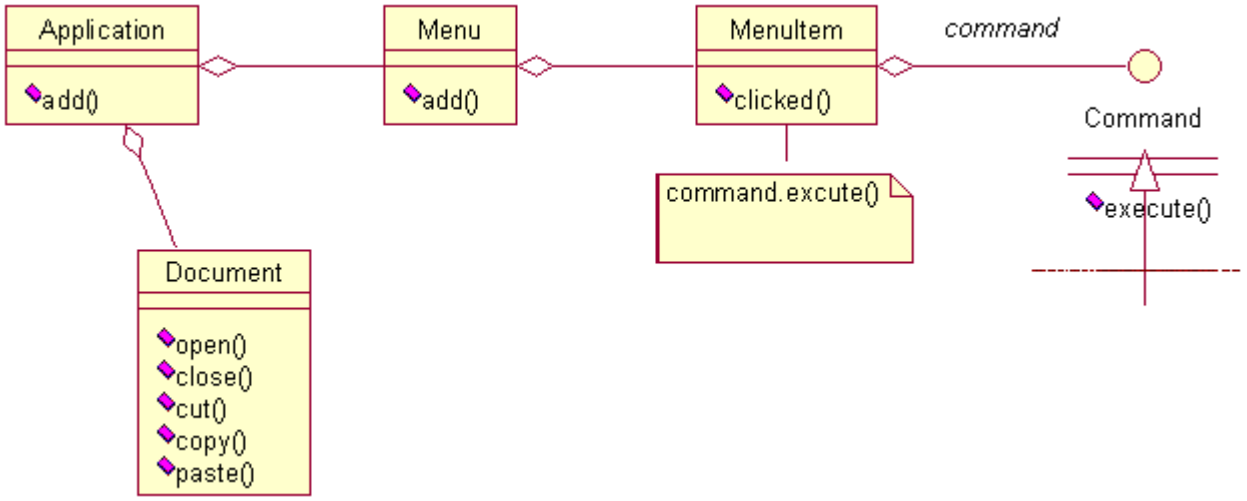
- Action, Transaction

Motivação

- Algumas vezes é necessário emitir solicitações para objetos que nada sabem sobre a operação que está sendo solicitada ou sobre o receptor da mesma.
  - 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 no 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 é 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 em um objeto. Este objeto pode ser armazenado e passado como outros objetos.
- A interface Command:
  - Define uma interface para execução de operações. Na sua forma mais simples, inclui uma única operação execute().
- As classes concretas:

- Implementam a interface Command.
- Especificam um par receptor-ação através:
  - do armazenamento do receptor como uma variável de instância, e;
  - da implementação de execute() para invocar a solicitação
- O receptor tem o conhecimento necessário para poder executar a solicitação.

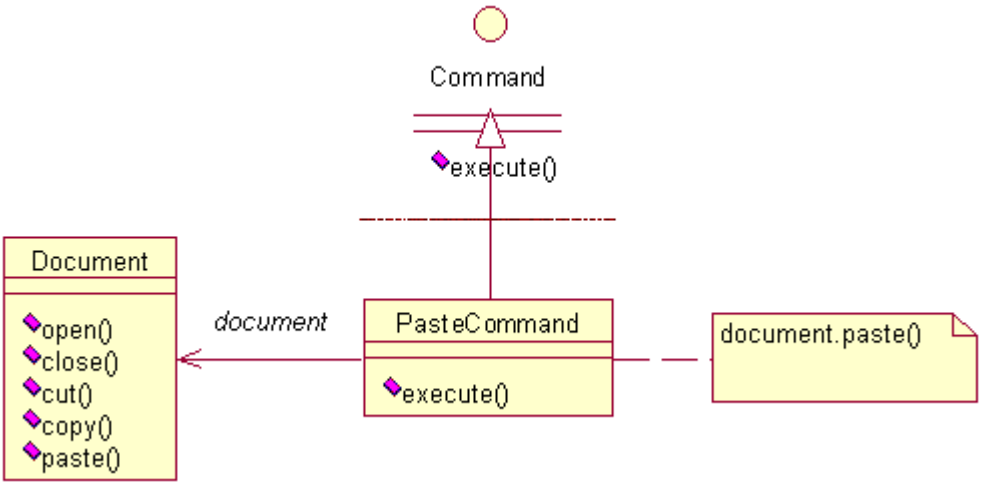
Exemplo do *Toolkit*



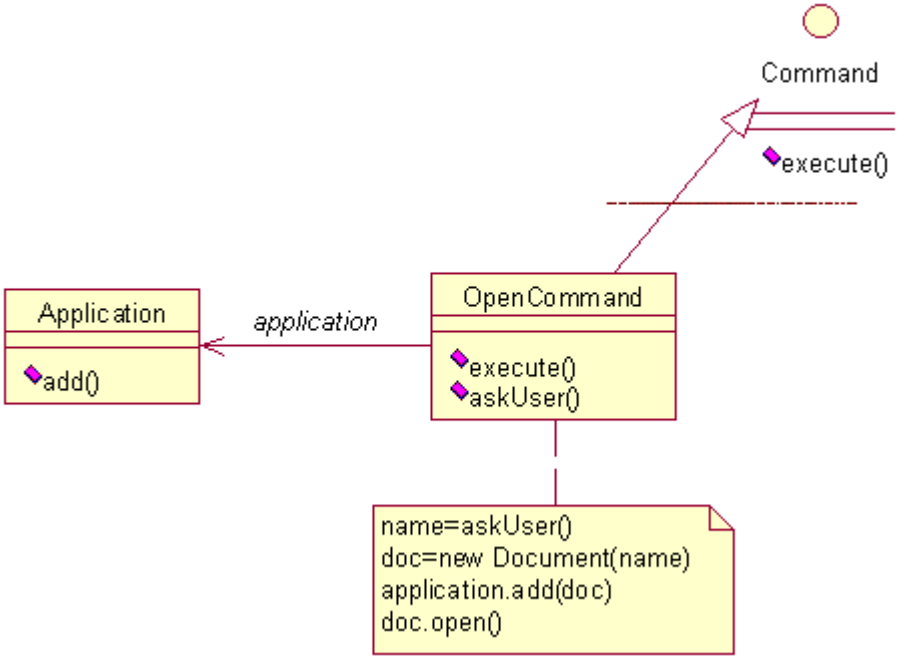
- Menus podem ser implementados facilmente com objetos Command:
  - Cada escolha num Menu é uma instância de uma classe MenuItem.
  - Uma classe Application cria estes menus e seus itens de menus juntamente com o resto da interface do usuário.
  - A classe Application também mantém um registro de acompanhamento dos objetos Document que um usuário abriu.
- A aplicação configura cada MenuItem com uma instância de uma classe concreta de Command.
- Quando o usuário seleciona um MenuItem, o MenuItem chama execute() no seu Command, e execute() executa a operação.
- Note que:

- MenuItens não sabem qual a classe concreta de Command que usam.
- As classes concretas de Command armazenam o receptor da solicitação que invoca uma ou mais operações no receptor.

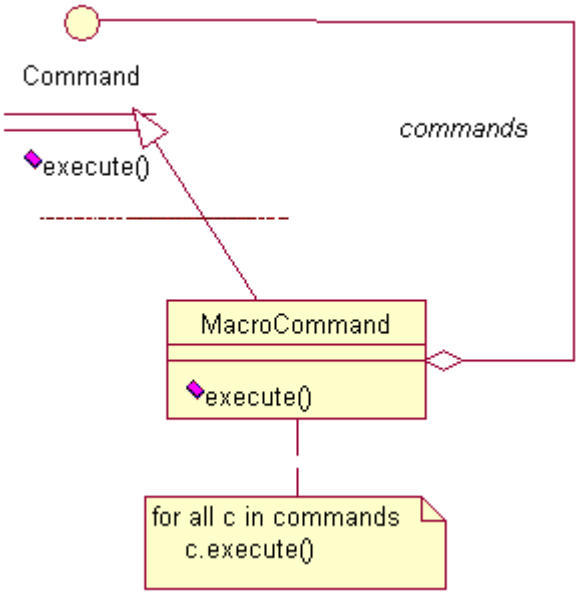
- Por exemplo, um PasteCommand suporta colar (*paste*) textos da área de transferência (*clipboard*) em um Document.
  - O receptor de PasteCommand é o objeto Document que é fornecido por instanciação.
  - A operação execute() invoca paste() no Document que está recebendo.



- 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.
- Para encadear comandos desta forma, podemos definir uma classe MacroCommand para permitir que um MenuItem execute um número aberto de comandos.
- A classe MacroCommand:
  - é uma classe concreta de Command;
  - executa uma seqüência de Commands;
  - não tem um receptor explícito porque os comandos que ele seqüencia definem seu próprio receptor.



- Observe com estes exemplos como o padrão Command desacopla o objeto que invoca a operação daquele que tem o conhecimento para executá-la.
- Isto 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 classe concreta de Command.
- 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 emití-la; ele não necessita saber como a solicitação será executada.

Aplicabilidade

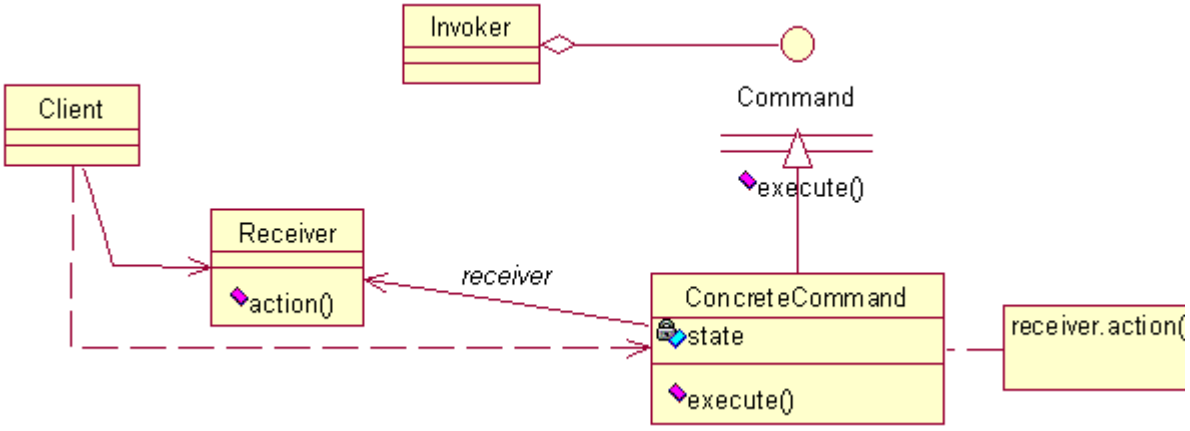
- Use o padrão Command para:
  - Parametrizar objetos por uma ação a ser executada, da forma como os objetos MenuItem fizeram acima.
    - Tal parametrização pode ser expressada numa

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 a objetos para *callbacks*.
- Especificar, enfileirar e executar solicitações em tempos diferentes.
    - Um objeto Command pode ter um tempo de vida independente da solicitação original.
    - Se o receptor de uma solicitação pode ser representado de uma maneira independente do espaço de endereçamento, então é possível 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 pode 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(), pode-se 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

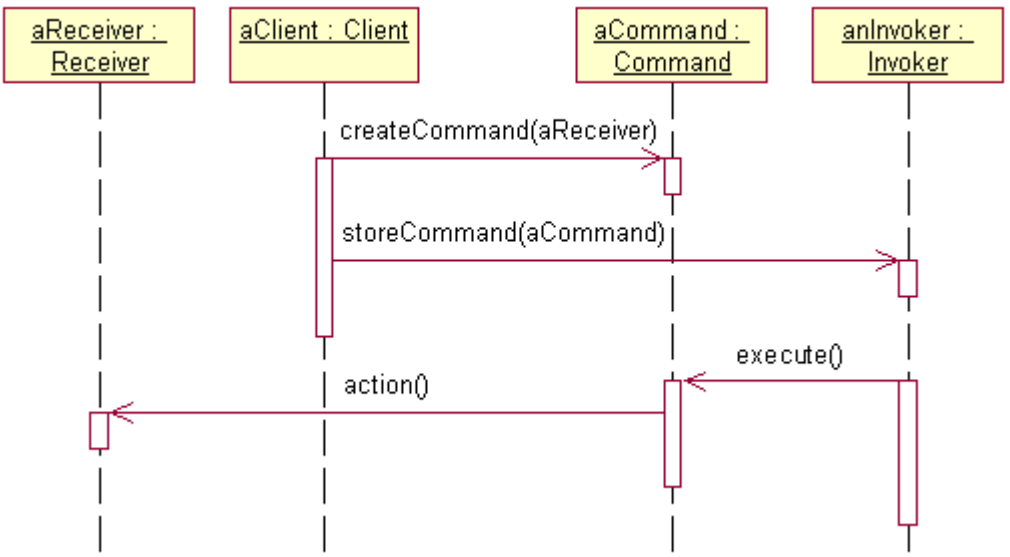
- **Command**
  - Declara uma interface para a execução de uma operação.
- **ConcreteCommand**(PasteCommand, OpenCommand)
  - Define uma vinculação entre um objeto Receiver e uma ação.
  - Implementa execute() através da invocação da (s) correspondente(s) operação(ções) no

Receiver.

- **Client**(Application)
  - Cria um objeto ConcreteCommand e estabelece o seu receptor (Receiver).
- **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 os comandos podem 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.



Conseqüências

- Command desacopla o objeto que invoca a operação daquele que sabe como executá-la.
- Commands são objetos de primeira classe, ou seja, podem ser manipulados e estendidos como qualquer outro objeto.
- Um comando pode ser composto por outros comandos. Um exemplo disso é a classe MacroCommand descrita anteriormente. Em geral, comandos compostos são uma instância do padrão Composite.
- É fácil acrescentar novos Commands porque não é preciso mudar classes existentes.

Detalhes de implementação

- Quão inteligente deveria ser um comando?
  - Em um contexto mais simples, ele define uma vinculação entre um receptor e as ações que executam a solicitação.
  - Em outro contexto mais complexo, ele implementa tudo sozinho, sem delegar para nenhum receptor. É útil quando:
    - se deseja definir comandos que são independentes de classes existentes;
    - não existe um receptor adequado;
    - um comando conhece seu receptor implicitamente.
      - 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 estes dois extremos estão os comandos que têm conhecimento suficiente para encontrar o seu receptor dinamicamente.
- Suportando desfazer e refazer
  - Command deve fornecer uma maneira de reverter sua execução através de uma operação

unexecute().

- ConcreteCommand pode necessitar armazenar estados adicionais. Estes estados podem incluir:
  - o objeto Receiver;
  - os argumentos da operação executada no receptor;
  - quaisquer valores originais no Receiver que podem mudar como resultado do tratamento da solicitação. O Receiver deve fornecer operações que permitem ao comando retornar o Receiver 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 de uma lista histórica de comandos que foram executados.

Exemplo de código

- O código mostrado aqui é referente à seção de Motivação.
- No arquivo Command.java:

```
//interface Command
public interface Command {
	public void execute();
}
```

- No arquivo OpenCommand.java:

```
//classe OpenCommand
public class OpenCommand implements Command {
	private Application application;

	public OpenCommand(Application application){
		this.application = application;
	}

	protected String askUser(){
		return "File2.doc";
```

```
	}

	public void execute(){
		String name = askUser();

		if (name.length() != 0){
			Document document = new Document(name
			this.application.add(document);
			document.open();
		}
	}
}
```

- No arquivo PasteCommand.java:

```
//classe PasteCommand
public class PasteCommand implements Command{
	private Document document;

	public PasteCommand(Document document){
		this.document = document;
	}

	public void execute(){
		this.document.paste();
	}
}
```

- No arquivo MacroCommand.java:

```
//classe MacroCommand
public class MacroCommand implements Command{
	private Collection commands;

	public MacroCommand(){
	    this.commands = new ArrayList();
	}

	public void add(Command command){
		this.commands.add(command);
	}
```

```
	public void remove(Command command){
		this.commands.remove(command);
	}

	public void execute(){
		Iterator i = commands.iterator();
		Command cada;
		while (i.hasNext()){
			cada = (Command)i.next();
			cada.execute();
		}
	}
}
```

- No arquivo Document.java:

```
//class Document
public class Document {

  private String name;

  public Document(String name) {
    this.name = name;
  }

  public void open(){
    System.out.println("Documento "+ this.name + " foi ab
  }

  public void paste(){
    System.out.println("Algo colado no Documento " + this
  }

  public void close(){}

  public void cut(){}

  public void copy(){}
}
```

- No arquivo Menu.java:

```
//classe Menu
public class Menu{

  private Collection menuItens;

  public Menu() {
    this.menuItens = new ArrayList();
  }

  public void add (MenuItem menuItem){
    this.menuItens.add(menuItem);
  }
}
```

- No arquivo MenuItem.java:

```
//classe MenuItem
public class MenuItem {

  private String label;
  private Command command;

  public MenuItem(String label, Command command) {
    this.label = label;
    this.command = command;
  }

  public void clicked(){
    command.execute();
  }
}
```

- Finalmente, precisamos de uma aplicação (arquivo Application.java):

```
//classe Application
public class Application{

  private Collection documents;

  public Application() {
```

```
    this.documents = new ArrayList();
  }

  public void add (Document document){
    this.documents.add(document);
  }

  public MenuItem createMenuItem(String label, Command co
    return new MenuItem(label, command);
  }

  public Menu createMenu(){
    return new Menu();
  }

 public static void main(String[] args){

    Application application = new Application();
    Menu menu = application.createMenu();

    Command openCommand = new OpenCommand(application);
    Command pasteCommand = new PasteCommand(new Document(
    MacroCommand macroCommand = new MacroCommand();

    macroCommand.add(openCommand);
    macroCommand.add(pasteCommand);

    MenuItem openItem = application.createMenuItem("Open"
    MenuItem pasteItem = application.createMenuItem("Past
    MenuItem macroItem = application.createMenuItem("Macr

    menu.add(openItem);
    menu.add(pasteItem);
    menu.add(macroItem);

    openItem.clicked();
    pasteItem.clicked();
    macroItem.clicked();

  }
}
```

- A execução de Application imprime:

```
Documento File2.doc foi aberto.
Algo colado no Documento File1.doc.
Documento File2.doc foi aberto.
Algo colado no Documento File1.doc.
```

- Código fonte aqui.

Padrões Relacionados

- Um Composite pode ser usado para implementar MacroCommands.
- Um Memento 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.

programa