
Padrões de Projeto

Padrões Comportamentais

— Dra. Alana Morais —

Padrões de Projeto

		Propósito		
		1. Criação	2. Estrutura	3. Comportamento
Escopo	Classe	Factory Method	Class Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Objetivo Aula

Apresentar e discutir sobre os padrões comportamentais:

- Iterator

- Command

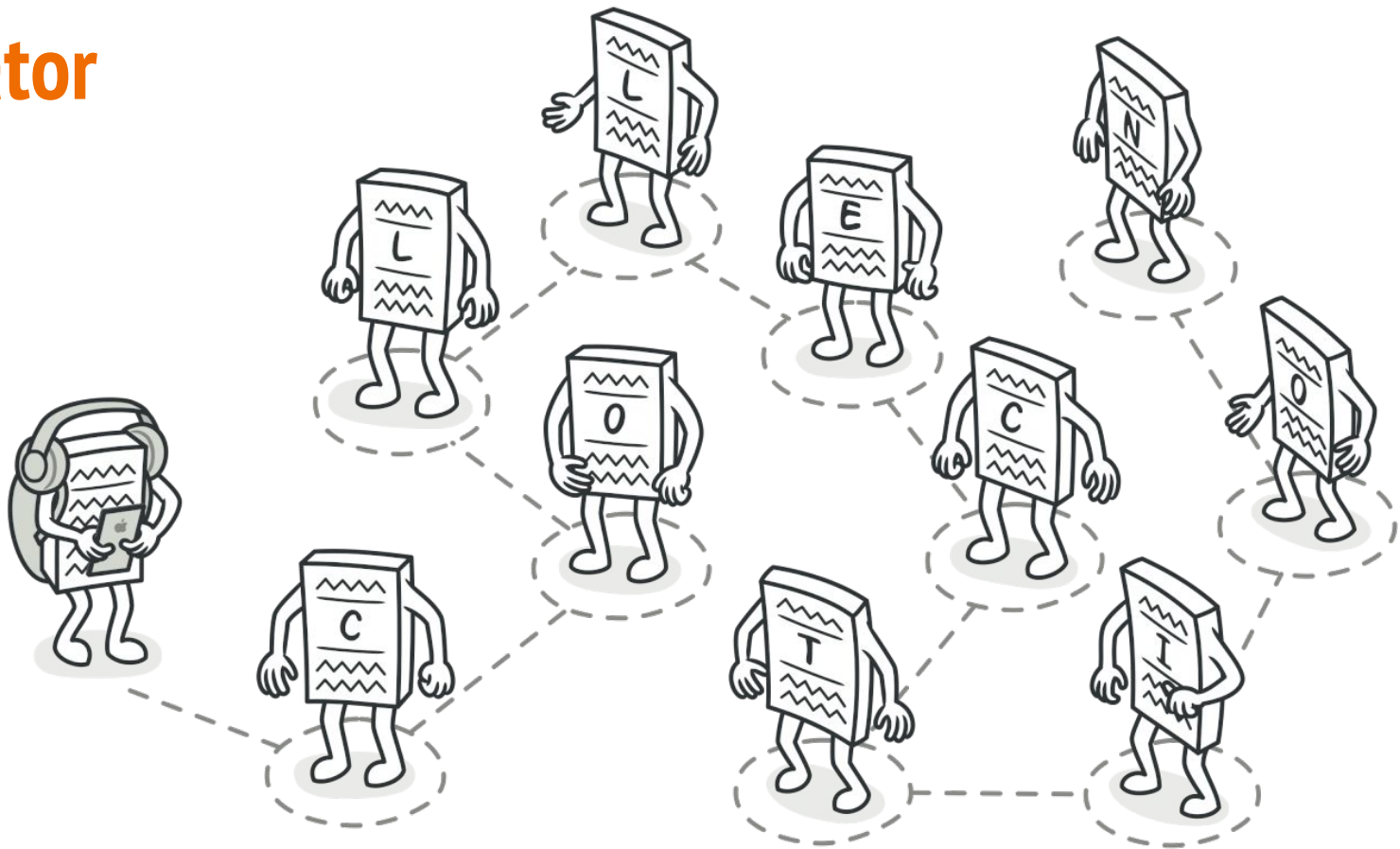
Padrões de Projeto

		Propósito		
		1. Criação	2. Estrutura	3. Comportamento
Escopo	Classe	Factory Method	Class Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Iterator

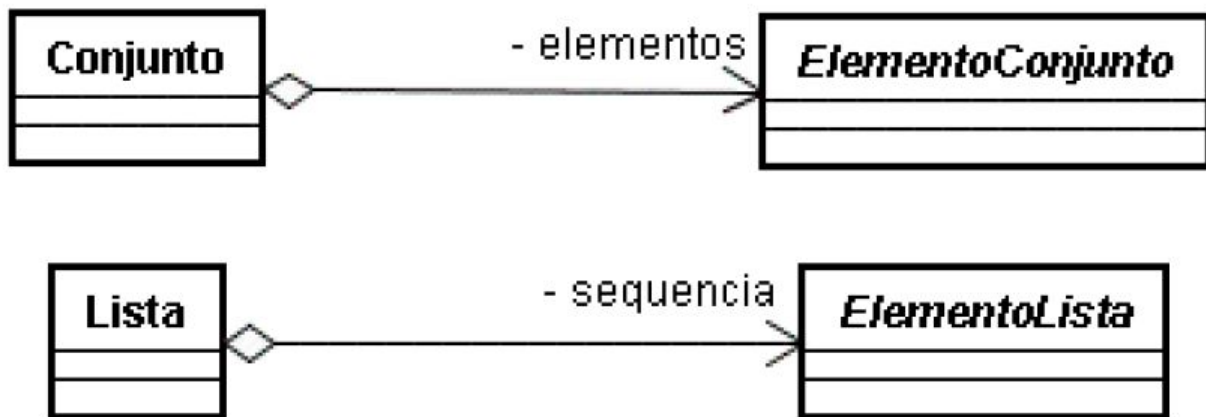
“PROVER UMA MANEIRA DE ACESSAR OS ELEMENTOS DE UM OBJETO AGREGADO SEQUENCIALMENTE SEM EXPOR SUA REPRESENTAÇÃO INTERNA.”

Iterator

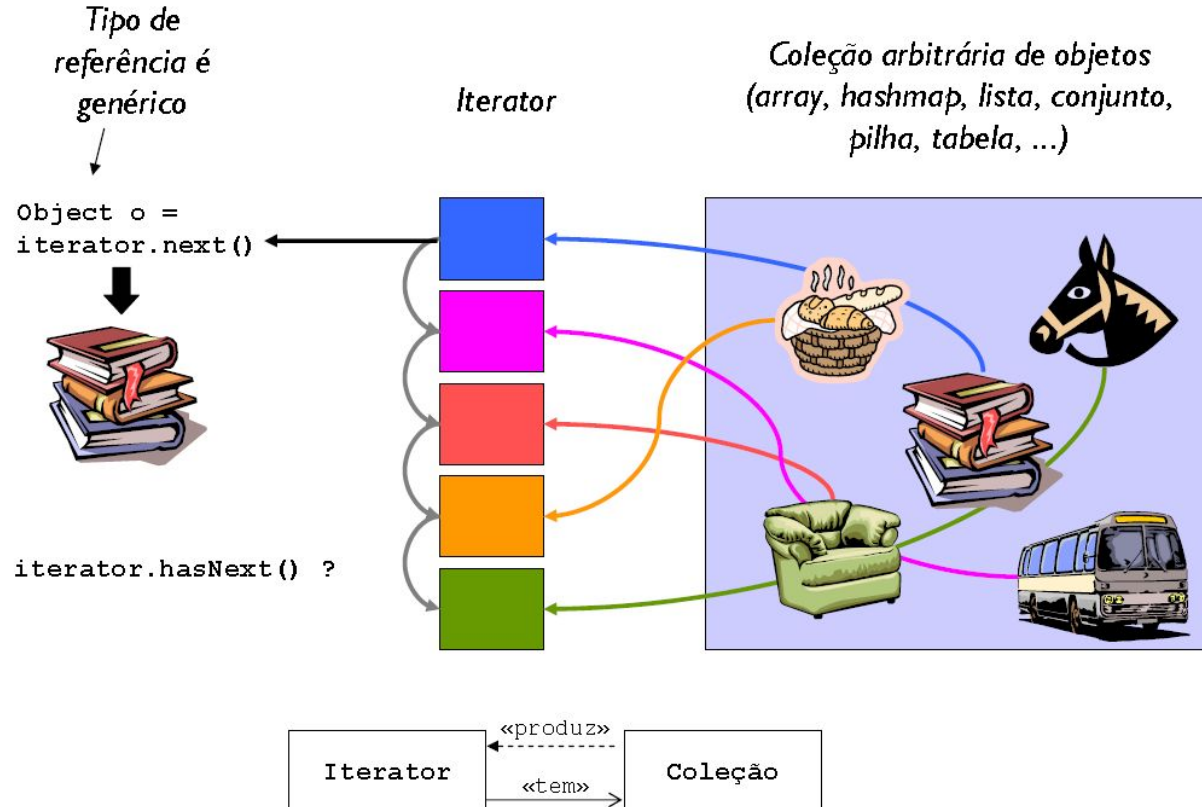


Iterator - Problema

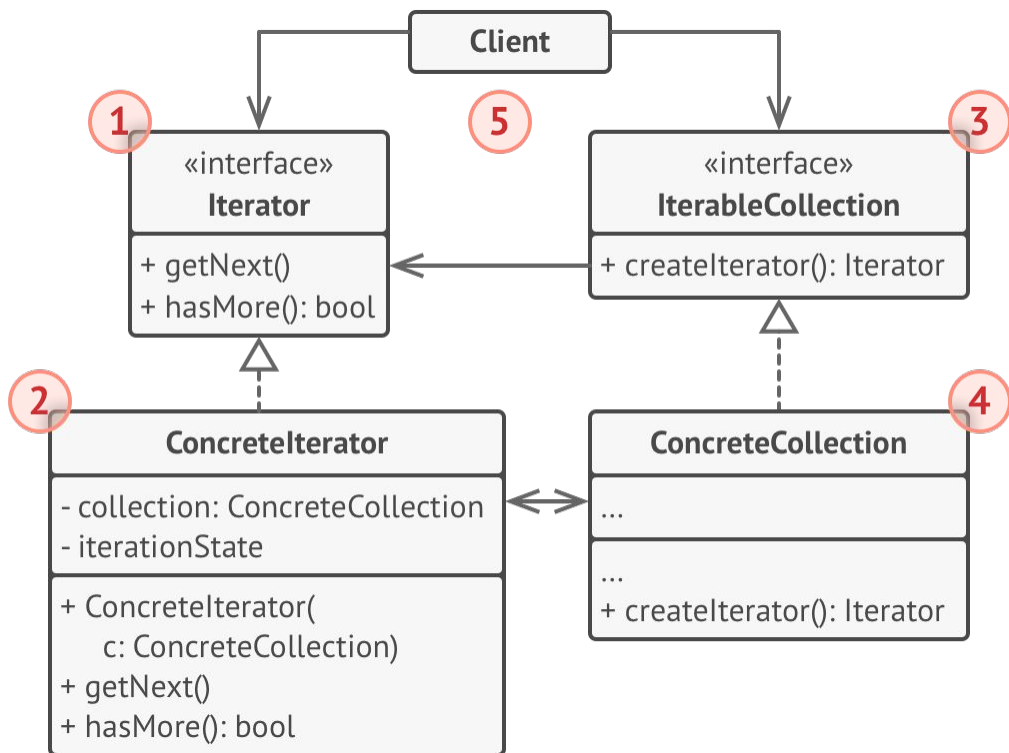
1. Cliente precisa acessar os elementos;
2. Cada coleção é diferente e não queremos expor a estrutura interna de cada um para o Cliente.



Iterator - Problema

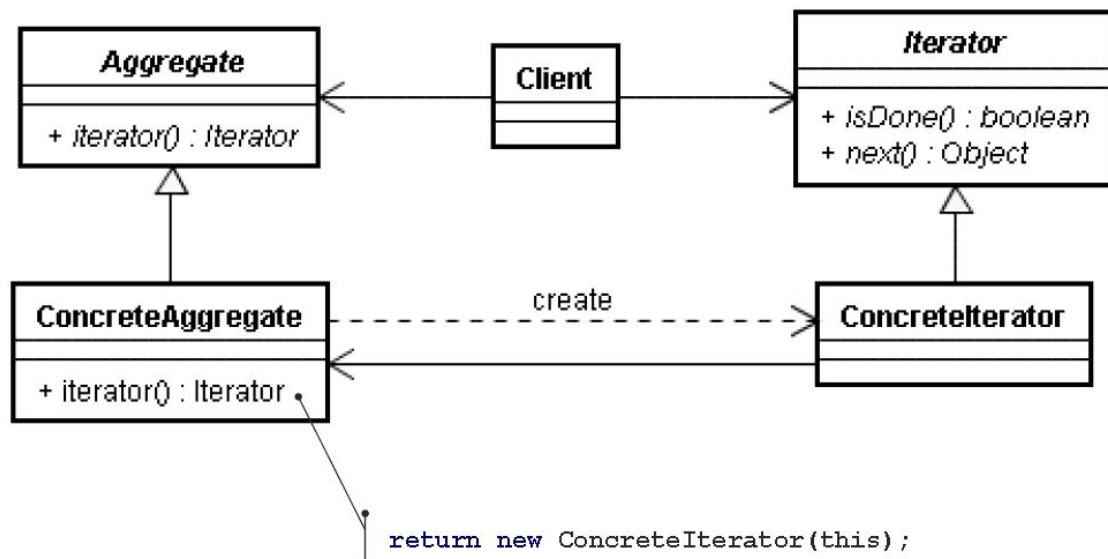


Iterator - Solução



1. A interface **Iterator** declara as operações necessárias para percorrer uma coleção: buscando o próximo elemento, recuperando a posição atual, reiniciando a iteração, etc.
2. O **ConcreteIterator** implementa algoritmos específicos para percorrer uma coleção. O objeto iterador deve rastrear o progresso de percurso por conta própria. Isso permite que vários iteradores percorram a mesma coleção independentemente um do outro.
3. A interface **IterableCollection** declara um ou vários métodos para obter iteradores compatíveis com a coleção.
4. **ConcreteCollection** retorna novas instâncias de uma determinada classe de agente iterativo toda vez que o cliente solicita uma.
5. O **Cliente** trabalha com coleções e iteradores por meio de suas interfaces. Dessa forma, o cliente não é acoplado a classes concretas, permitindo que você use várias coleções e iteradores com o mesmo código de cliente.

Iterator - Estrutura



Iterator - Exemplo

Implemente um iterador de nomes a partir de um array simples.



Quando Usar?

Iterators servem para acessar o conteúdo de um agregado sem expor sua representação interna

- Oferece uma interface uniforme para atravessar diferentes estruturas agregadas
- Iterators são implementados nas coleções do Java. É obtido por meio do método `iterator()` de `Collection`, que devolve uma instância de `java.util.Iterator`.
- Interface `java.util.Iterator`:
- `iterator()` é um exemplo de FactoryMethod

```
package java.util;  
public interface Iterator<E> {  
    boolean hasNext();  
    Object next();  
    void remove();  
}
```

Quando Usar?

- Quiser acessar objetos agregados (coleções) sem expor a estrutura interna;
- Quiser prover diferentes meios de acessar tais objetos;
- Quiser especificar uma interface única e uniforme para este acesso.

Iterator - Vantagens e Desvantagens

Múltiplas formas de acesso:

- Basta implementar um novo iterador com uma nova lógica de acesso.

Interface simplificada:

- Acesso é simples e uniforme para todos os tipos de coleções.

Mais de um iterador:

- É possível ter mais de um acesso à coleção em pontos diferentes.

Iterator - Exercício

Imagine que você está trabalhando no sistema da NET.

Você precisará lidar com coleções armazenadas de diferentes formas: coleção de canais de filmes, de canais de esporte, etc. Que podem ser armazenadas em diferentes estruturas (array, arrayList).

Melhore a implementação deste problema com o uso de Iterator

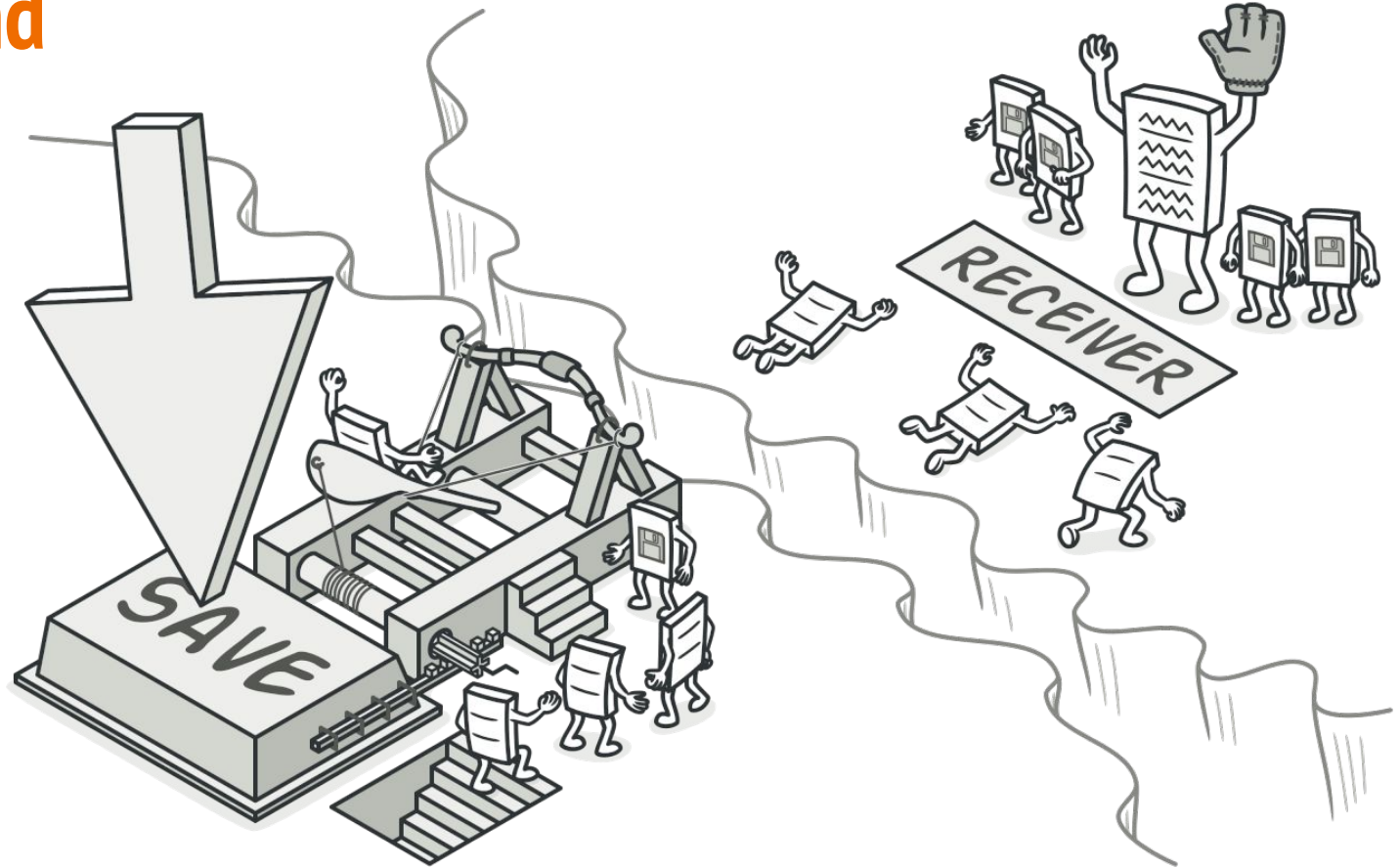
Iterator - Exercício

```
1  ArrayList<Canal> arrayListDeCanais = new ArrayList<Canal>();
2  Canal[] matrizDeCanais = new Canal[5];
3
4  for (Canal canal : arrayListDeCanais) {
5      System.out.println(canal.nome);
6  }
7
8  for (int i = 0; i < matrizDeCanais.length; i++) {
9      System.out.println(matrizDeCanais[i].nome);
10 }
```

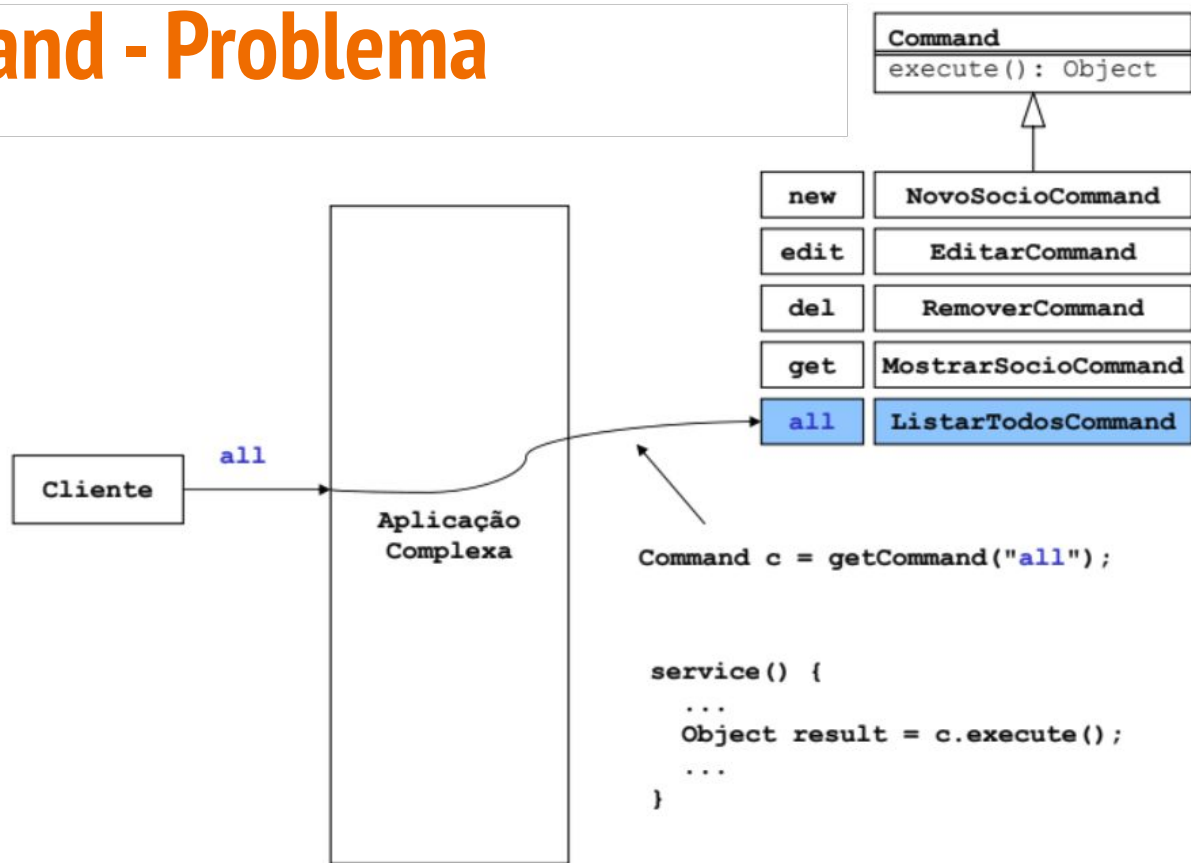

Command

"ENCAPSULAR UMA REQUISIÇÃO COMO UM OBJETO, PERMITINDO QUE CLIENTES PARAMETRIZEM DIFERENTES REQUISIÇÕES, FILAS OU REQUISIÇÕES DE LOG, E SUPORTAR OPERAÇÕES REVERSÍVEIS."

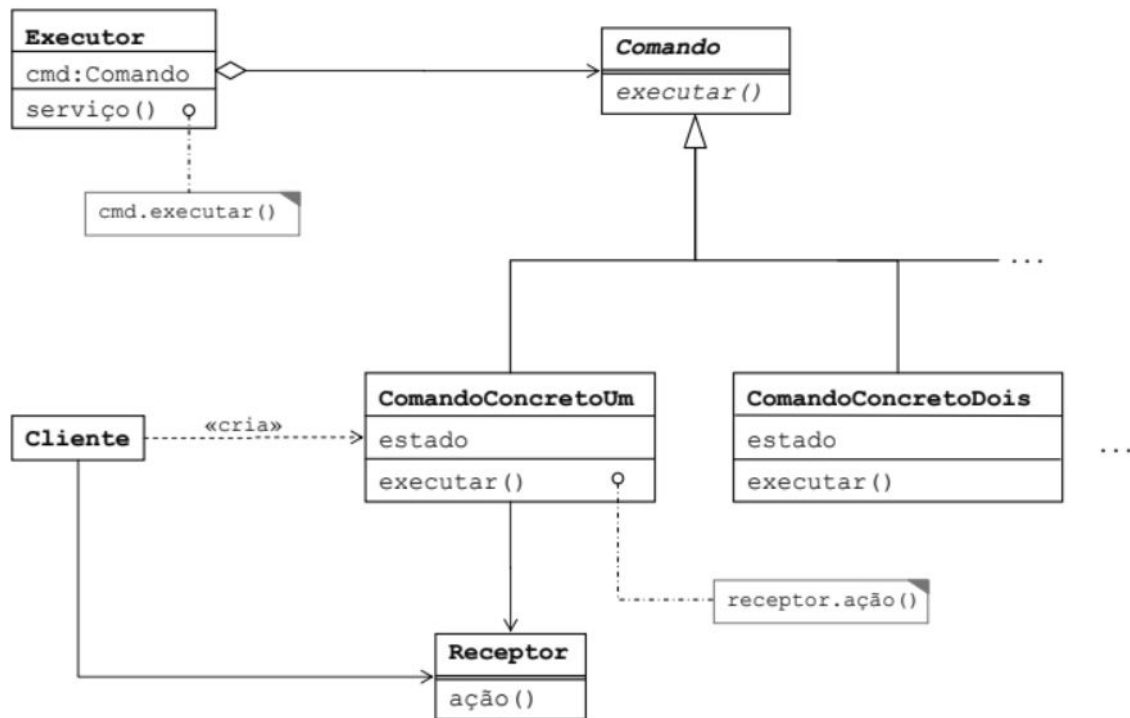
Command



Command - Problema



Estrutura de Command



```
public interface Command {  
    public Object execute(Object arg);  
}
```

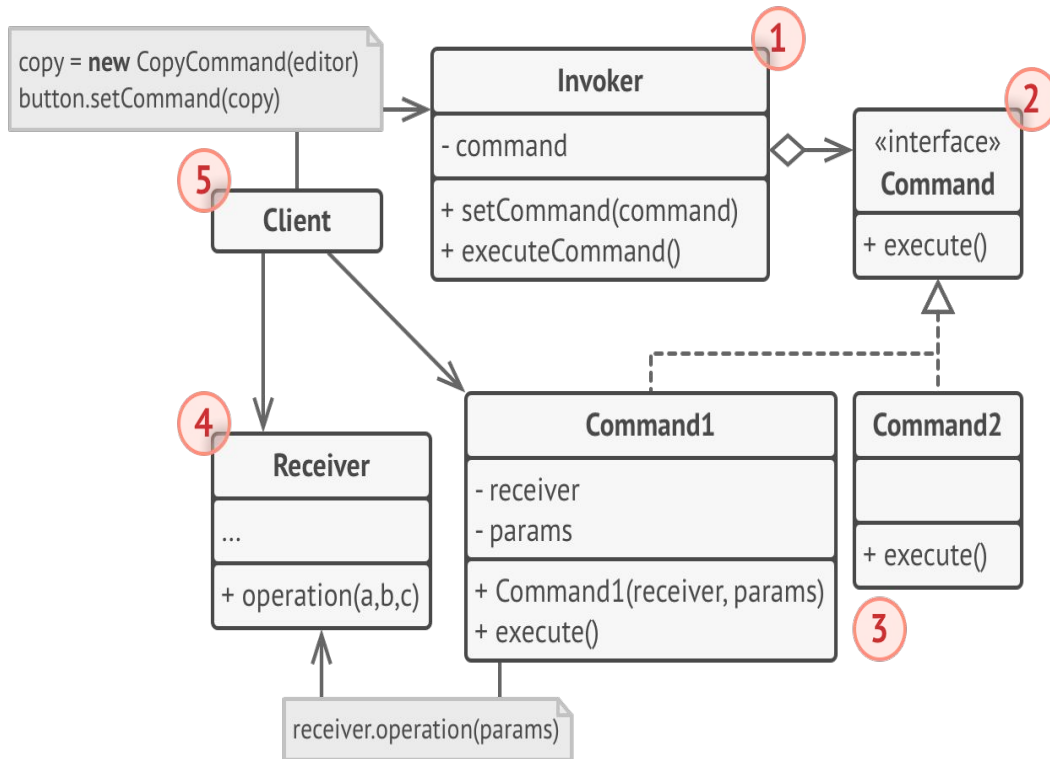
```
public class Server {  
    private Database db = ...;  
    private HashMap cmds = new HashMap();  
  
    public Server() {  
        initCommands();  
    }  
  
    private void initCommands() {  
        cmds.put("new", new NewCommand(db));  
        cmds.put("del",  
            new DeleteCommand(db));  
        ...  
    }  
}
```

```
    public void service(String cmd,  
        Object data) {  
        ...  
        Command c = (Command)cmds.get(cmd);  
        ...  
        Object result = c.execute(data);  
        ...  
    }  
}
```

```
public interface NewCommand implements Command {  
  
    public NewCommand(Database db) {  
        this.db = db;  
    }  
  
    public Object execute(Object arg) {  
        Data d = (Data)arg;  
        int id = d.getArg(0);  
        String nome = d.getArg(1);  
        db.insert(new Member(id, nome));  
    }  
}
```

```
public class DeleteCommand implements Command {  
  
    public DeleteCommand(Database db) {  
        this.db = db;  
    }  
  
    public Object execute(Object arg) {  
        Data d = (Data)arg;  
        int id = d.getArg(0);  
        db.delete(id);  
    }  
}
```

Command



1. O **Invoker** (ou Sender) é responsável por iniciar solicitações. Esta classe deve ter um campo para armazenar uma referência a um objeto de comando. O remetente aciona esse comando em vez de enviar a solicitação diretamente ao receptor. Observe que o remetente não é responsável pela criação do objeto de comando. Geralmente, ele recebe um comando pré-criado do cliente por meio do construtor.
2. A interface de **Command** geralmente declara apenas um único método para executar o comando.
3. **Commands** concretos implementam vários tipos de solicitações. Um comando concreto não deve realizar o trabalho sozinho, mas sim passar a chamada para um dos objetos da lógica de negócios.
4. A **Receiver** contém alguma lógica de negócios. A maioria dos comandos manipula apenas os detalhes de como uma solicitação é passada para o receptor, enquanto o próprio receptor faz o trabalho real.
5. O **Cliente** cria e configura objetos de comando concretos. O cliente deve passar todos os parâmetros da solicitação, incluindo uma instância do receptor, para o construtor do comando. Depois disso, o comando resultante pode estar associado a um ou vários remetentes.

Command - Exemplo

A loja virtual Alana Surf Wear precisa processar suas compras em seu sistema de modo a tornar este processo transparente para as diversas formas de pagamento possíveis.

A loja virtual pode vender por meio de boletos, cartões de crédito e débito.

Resolva este problema por meio do padrão Command.

Command

- Vantagens

- Princípio da responsabilidade única. Você pode separar classes que invocam operações de classes que executam essas operações.
- Princípio Aberto / Fechado. Você pode introduzir novos comandos no aplicativo sem quebrar o código do 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.

- Desvantagens

- O código pode se tornar mais complicado, já que você está introduzindo uma nova camada entre remetentes e destinatários.

Dúvidas?

alanamm.prof@gmail.com