

**Universidade Federal de Uberlândia**  
**Disciplina: POO2**  
**Prof. Fabiano Dorça**

Padrões de Projeto

Padrão Command

## Padrão Command

- O padrão Command encapsula um comando em um objeto.
- Tem como premissa desacoplar o objeto cliente e o objeto que executa a operação.
- O objetivo é tornar mais flexível e poderoso os serviços de trocas de mensagens entre objetos.
- Permite que clientes configurem comandos conforme necessário.

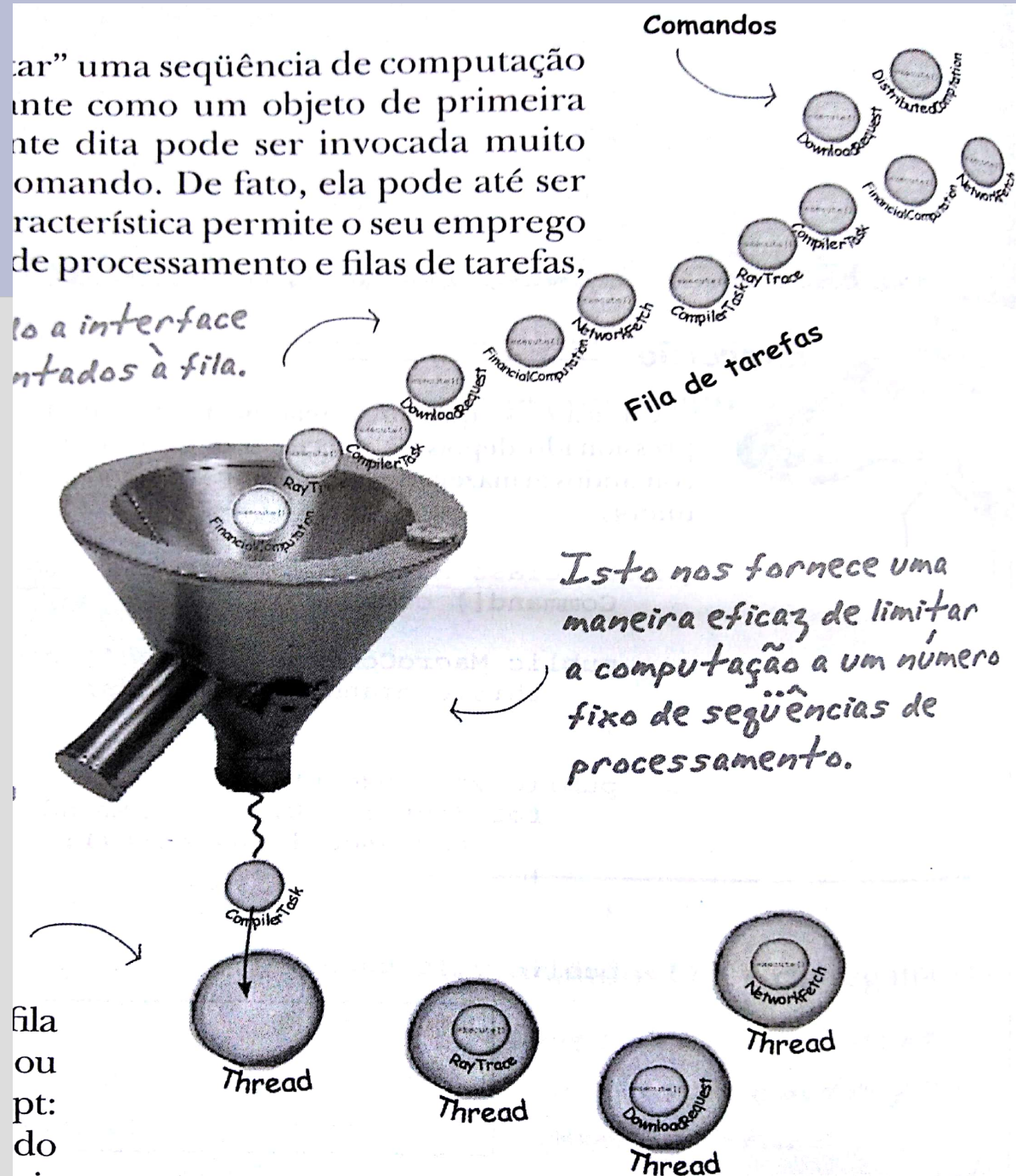
## Padrão Command

Desta forma, é possível:

- Configurar diferentes clientes com diferentes comandos,
- Enfileirar comandos,
- Fazer log de comandos,
- Dar suporte a operações de *undo*,
- Criar macros,
- Dar suporte a comunicação assíncrona.

car” uma seqüência de computação  
 ante como um objeto de primeira  
 te dita pode ser invocada muito  
 comando. De fato, ela pode até ser  
 racterística permite o seu emprego  
 de processamento e filas de tarefas,

to a interface  
 ntados à fila.



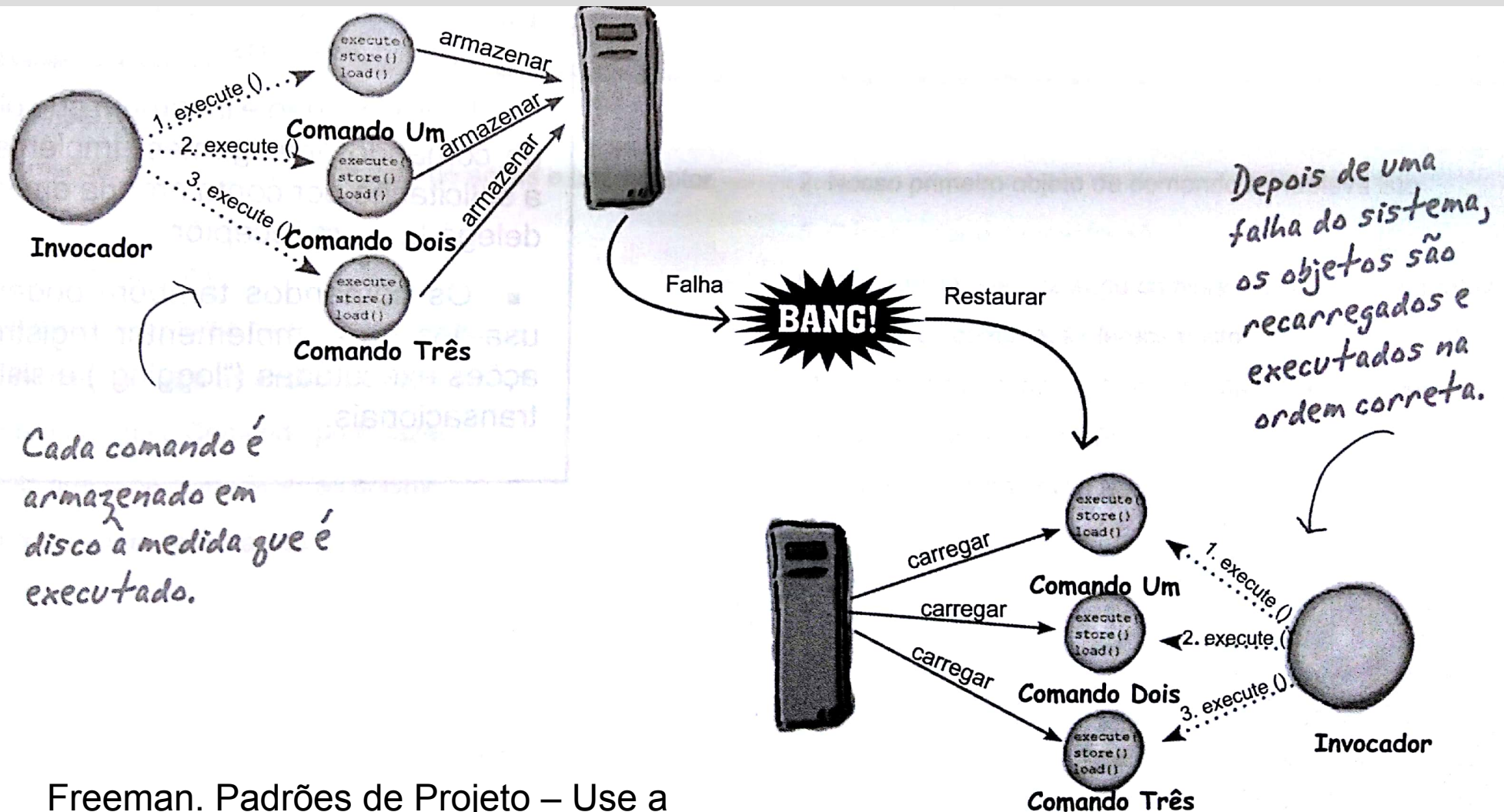
Isto nos fornece uma  
 maneira eficaz de limitar  
 a computação a um número  
 fixo de seqüências de  
 processamento.

fila  
 ou  
 pt:  
 do  
 ois

Sequências de processamento  
 computando tarefas

Freeman. Padrões de Projeto – Use a  
 Cabeça. Altabooks, 2010.

# Padrão Command



Freeman. Padrões de Projeto – Use a Cabeça. Altabooks, 2010.

## Padrão Command

- A chave deste padrão é uma classe abstrata **Command**, a qual declara uma interface para execução de operações.
- Esta interface inclui uma operação abstrata **Execute**.
- As subclasses concretas de **Command** especificam um par receptor-ação
  - Receptor: variável de instância
  - Ação: implementação de Execute para invocar a solicitação.
- O receptor tem o conhecimento necessário para executar a solicitação.

# Padrão Command

## Participantes:

As classes e que participam no padrão são:

### Command

- Declara uma interface para implementação de comandos.

### Client

- Utiliza comandos concretos através de um *invoker*, que pode ser configurado com os comandos necessários a um client específico.

### ConcreteCommand

- Implementa o método Execute, evocando as correspondentes operações no Receiver.

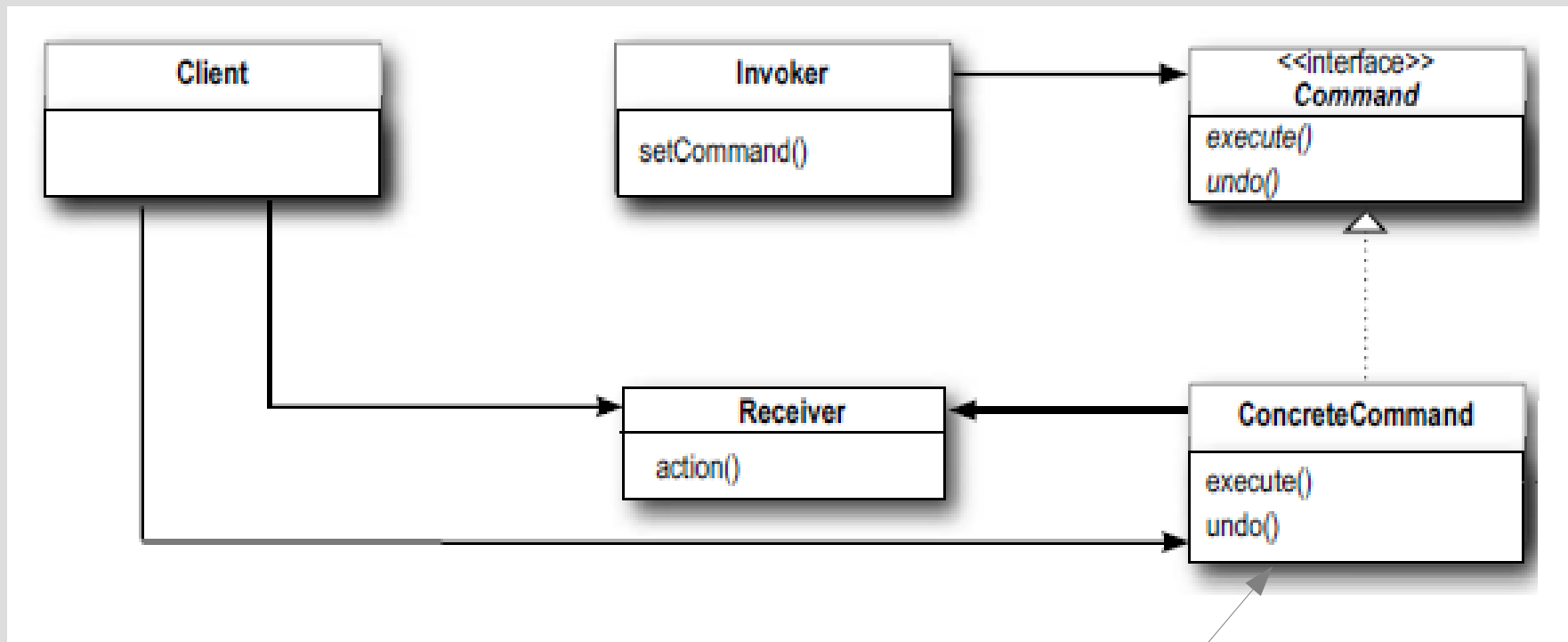
### Invoker

- Solicita ao ConcreteCommand a execução da solicitação. Pode armazenar macros, filas, log e implementar operações de UNDO e REDO.

### Receiver

- Executa as operações associadas à realização do pedido. Qualquer classe pode funcionar como um Receiver.

# Padrão Command



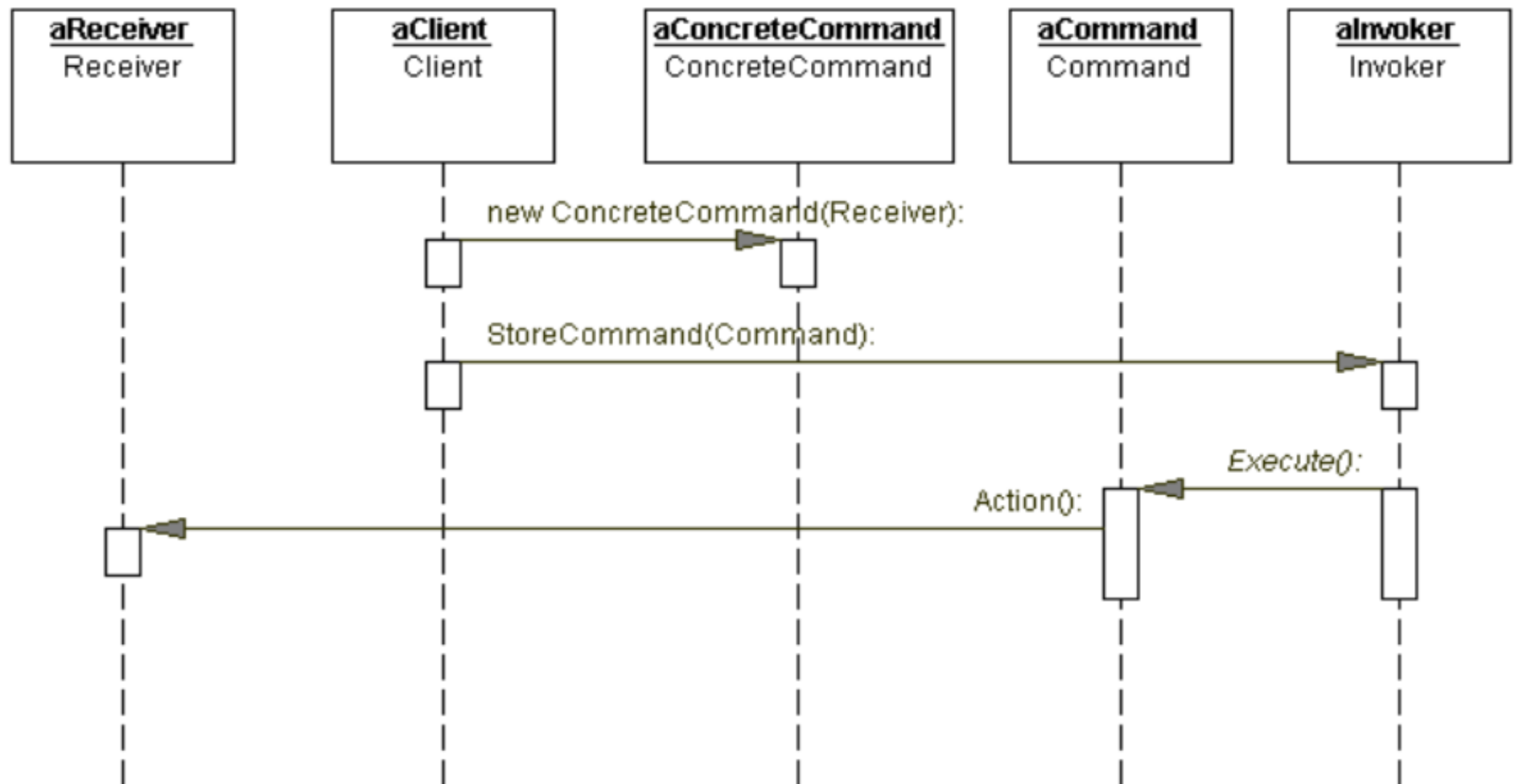
Fonte: <http://www.devmedia.com.br/>

Fachada (facade)



# Padrão Command

- Funcionamento do padrão command



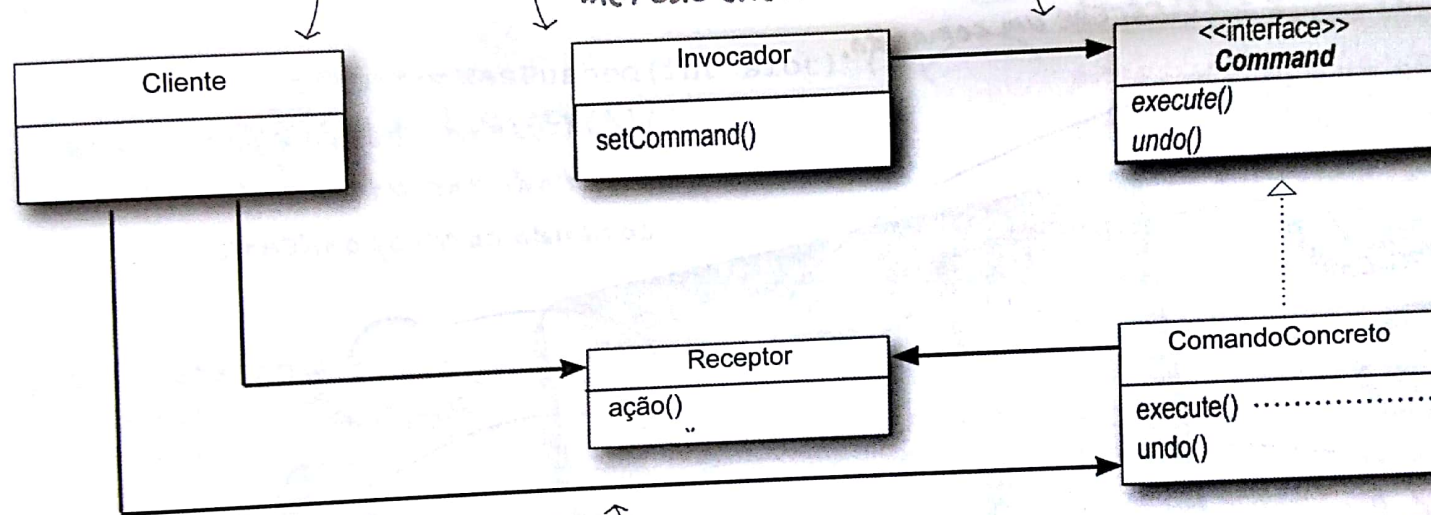
# Padrão Command

## Definição do Padrão Command: o diagrama de classes

O cliente é responsável pela criação de um ComandoConcreto e pela definição do seu Receptor.

O Invocador contém um comando e, em algum momento, pede ao comando para atender uma solicitação chamando o seu método execute().

Command declara uma interface para todos os comandos. Como vimos anteriormente, um comando é invocado através do seu método execute(), que pede a um receptor para executar uma ação. Como você pode perceber, essa interface também possui um método undo(), que será discutido mais adiante neste capítulo.



O método execute() invoca uma ou mais ações no receptor que são necessárias para atender a solicitação.

```
public void execute() {
    receiver.action()
}
```

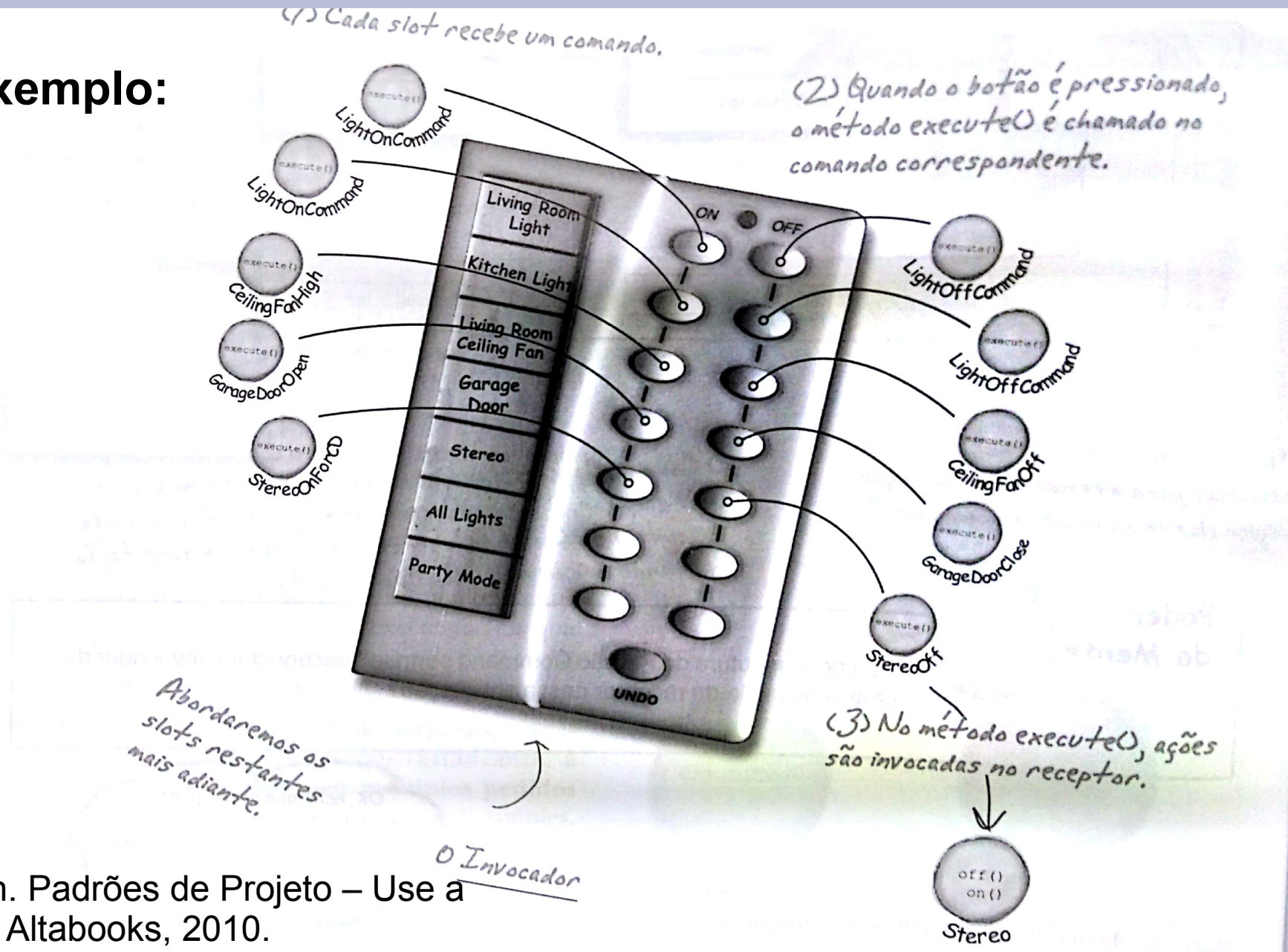
Freeman. Padrões de Projeto – Use a Cabeça. Altabooks, 2010.

O Receptor sabe como executar as tarefas necessárias para atender a solicitação. Qualquer classe pode atuar como um Receptor.

O ComandoConcreto define um vínculo entre uma ação e um Receptor. O Invocador faz uma solicitação chamando execute() e o ComandoConcreto executa essa solicitação chamando uma ou mais ações no Receptor.

# Padrão Command

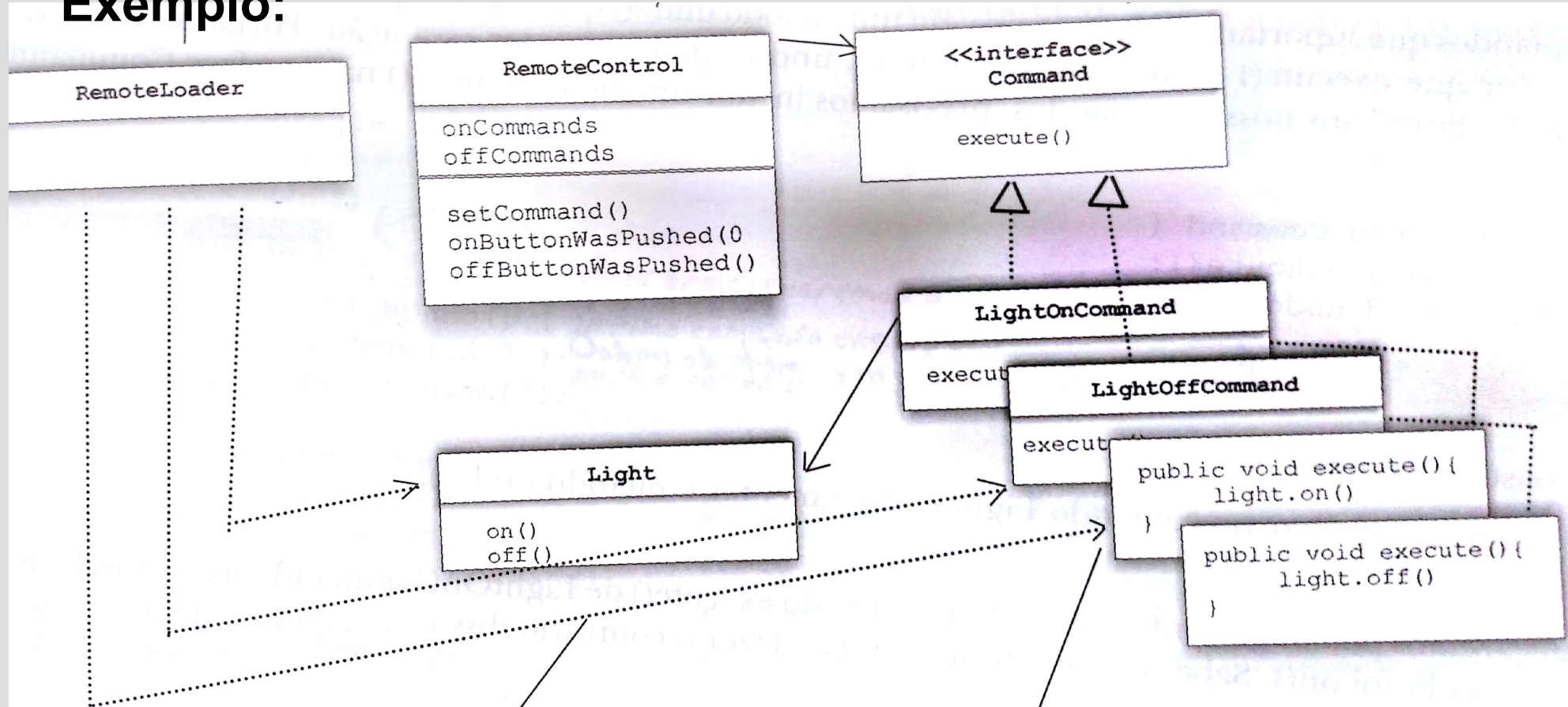
## Exemplo:





# Padrão Command

## Exemplo:



# Padrão Command

- Exemplo:

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

```
public class LightOnCommand implements Command {  
    Light light;
```

```
    public LightOnCommand(Light light) { ← Receiver informado  
        this.light = light;                pelo client  
    }
```

```
    public void execute() {  
        this.light.on();  
    }  
}
```

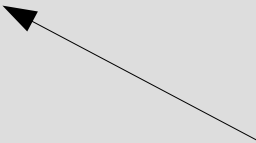
# Padrão Command

- `public class LightOnCommand implements Command {  
    Light light = new light();`

```
    public void execute() {  
        this.light.on();  
    }
```

```
}
```

Receiver já conhecido  
pelo concrete  
command



## Padrão Command

- Outro concrete command

```
public class LightOffCommand implements Command {
```

```
    Light light;
```

```
    public LightOffCommand(Light light) {
```

```
        this.light = light;
```

```
    }
```

```
    public void execute() {
```

```
        light.off();
```

```
    }
```

```
}
```

## Padrão Command

- Outro concrete command:

```
public class StereoOn implements Command {  
    Stereo stereo;  
    public StereoOn(Stereo stereo){  
        this.stereo = stereo;  
    }  
    public void execute() {  
        stereo.on();  
        stereo.setCD();  
        stereo.setVolume(11);  
    }  
}
```



## Padrão Command

- Exemplo de invoker

Cada cliente pode ter uma configuração de invoker diferente

```
public class ControleRemoto {  
    Command slot[10] = new Command[10];  
  
    public void setCommand(Command command, int n) {  
        slot[n] = command; }  
  
    public void botaoPressionado(int n) {  
        slot[n].execute(); }  
}
```

# Padrão Command

## Invoker separando ligar e desligar

```
public class ControleRemoto {  
    Command slot_on[10] = new Command[10];  
    Command slot_off[10] = new Command[10];  
  
    public void setOnCommand(Command command, int n) {  
        slot_on[n] = command; }  
  
    public void setOffCommand(Command command, int n) {  
        slot_off[n] = command; }  
  
    public void pressOn(int n) {  
        slot_on[n].execute(); }  
  
    public void pressOff(int n) {  
        slot_off[n].execute(); }  
}
```

## Padrão Command

- **Observações:**
  - O atributo slot armazena o comando que irá controlar o dispositivo específico;
  - O método setCommand() serve para definir os comandos que o slot irá “controlar”;
  - O pode-se mudar o comportamento dos botões (slots) em tempo de execução;
  - As classes que utilizarão o *invoker* ficam completamente desacopladas de quem executa os comandos e de como são executados.

# Padrão Command

- Como fazer um **undo**

```
public class ControleRemoto {  
  
    Command slot[10] = new Command[10];  
    Command undoCommand;  
  
    public void setCommand(Command command, int n) {  
        slot[n] = command; }  
  
    public void botaoPressionado(int n) {  
        slot[n].execute();  
        undoCommand = slot[n]; }  
  
    public void undo(){  
        undoCommand.unexecute(); }  
}
```

## Padrão Command

- Criando um macro

```
public class MacroCommand implements Command {  
    ...  
    ArrayList<Command> macro = new...  
  
    public void addCommand(Command c){  
        macro.add(c); }  
  
    public void executeMacro() {  
        for (Command c: macro) {  
            c.execute();  
        }  
    }  
}
```

# Padrão Command

## Vantagens da utilização:

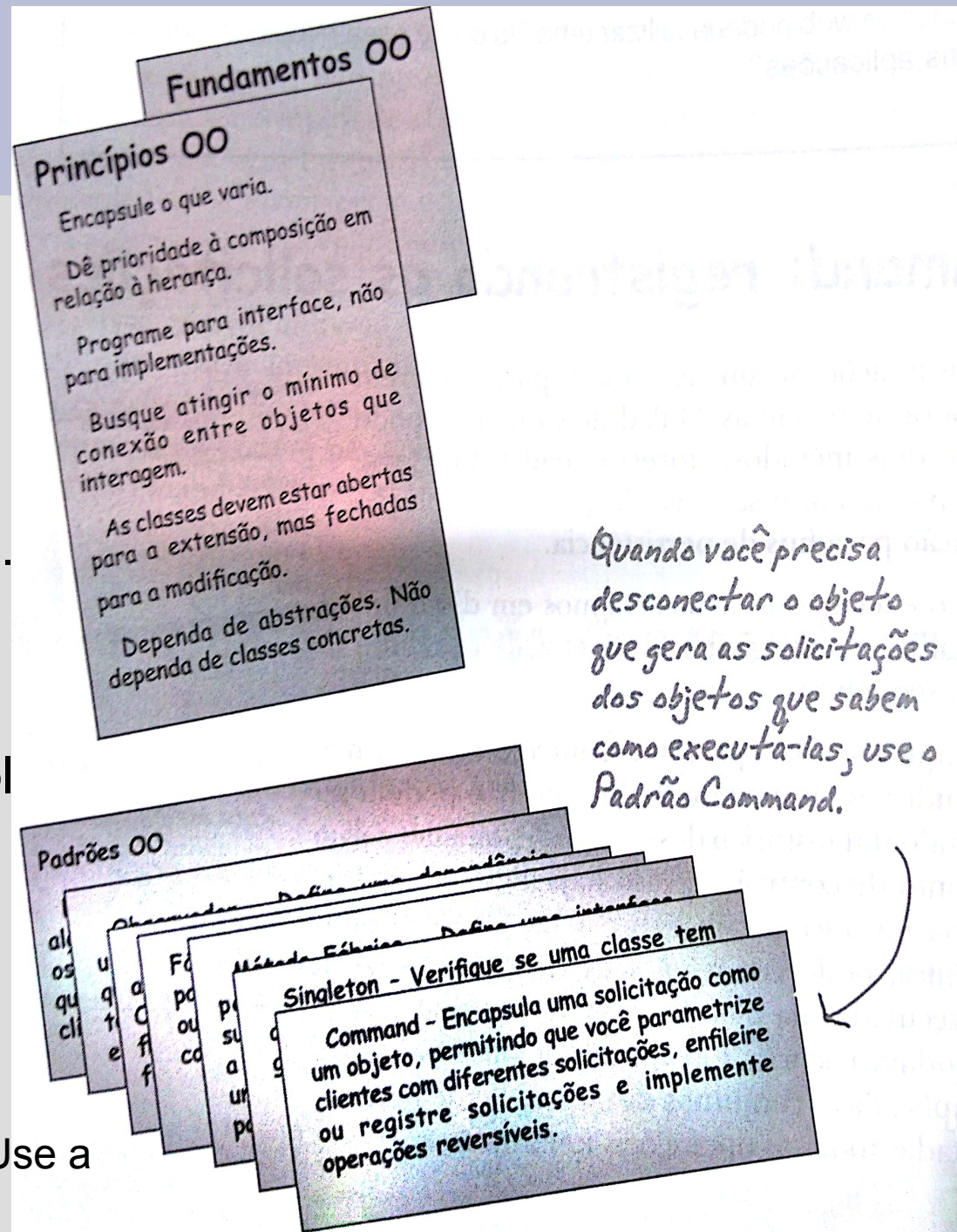
- Desacopla o objeto que invoca (controlador) a operação daquele que tem o conhecimento para executá-la (cria uma fachada entre sub-sistemas).
- Pode-se implementar uma fila de comandos no invoker, que pode executá-los não imediatamente, mas a medida do possível (**assíncrono**).
- Podemos substituir comandos dinamicamente, o que poderia ser útil para a implementação de menus sensíveis ao contexto, e adicionar novos comandos.
- Também podemos suportar scripts de comandos compondo comandos em comandos maiores (macros).
- Possibilidade de implementar Undo e Redo
- 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.

# Padrão Command

## Extensibilidade:

- Comandos são objetos, passíveis de extensão, composição, decoração, etc.
- Pode ser usado junto com Decorator ou outros padrões para formar comandos complexos.
- É possível definir novos comandos sem alterar nada existente.

Freeman. Padrões de Projeto – Use a Cabeça. Altabooks, 2010.



## Padrão Command

- Exercício:
  - Implemente o exemplo do controle remoto.
  - Implemente um log das operações executadas.
  - Implemente uma funcionalidade para desfazer operações.
  - Implemente uma funcionalidade que permita a criação de um lote de operações (macro command) que devem ser executadas em sequência.



# Padrão Command

- Dicas

- A interface *Command* deve ter acrescentada uma operação *Unexecute*, ou *Undo*, que o reverte efeitos de uma chamada anterior de *Execute*. Cada comando deve saber como desfazer a si próprio.
- Os comandos executados são armazenados em uma lista histórica no *invoker*.
- O nível ilimitado de desfazer e refazer operações é obtido percorrendo esta lista para trás e para frente, chamando operações *Unexecute* e *Execute*, respectivamente.

# Padrão Command

- Exemplo:

```
interface Command
    void execute();
    void unexecute();
}
```

- Desta forma, os commands concretos devem implementar ambos os métodos, sendo que o unexecute() deve dar suporte a operação desfazer.

# Padrão Command

- Fim.