



Modelos, Métodos e Técnicas de Engenharia de Software Visão e análise de projeto Padrões Prática 3 – Command (14)

Prof. Osmar de Oliveira Braz Junior

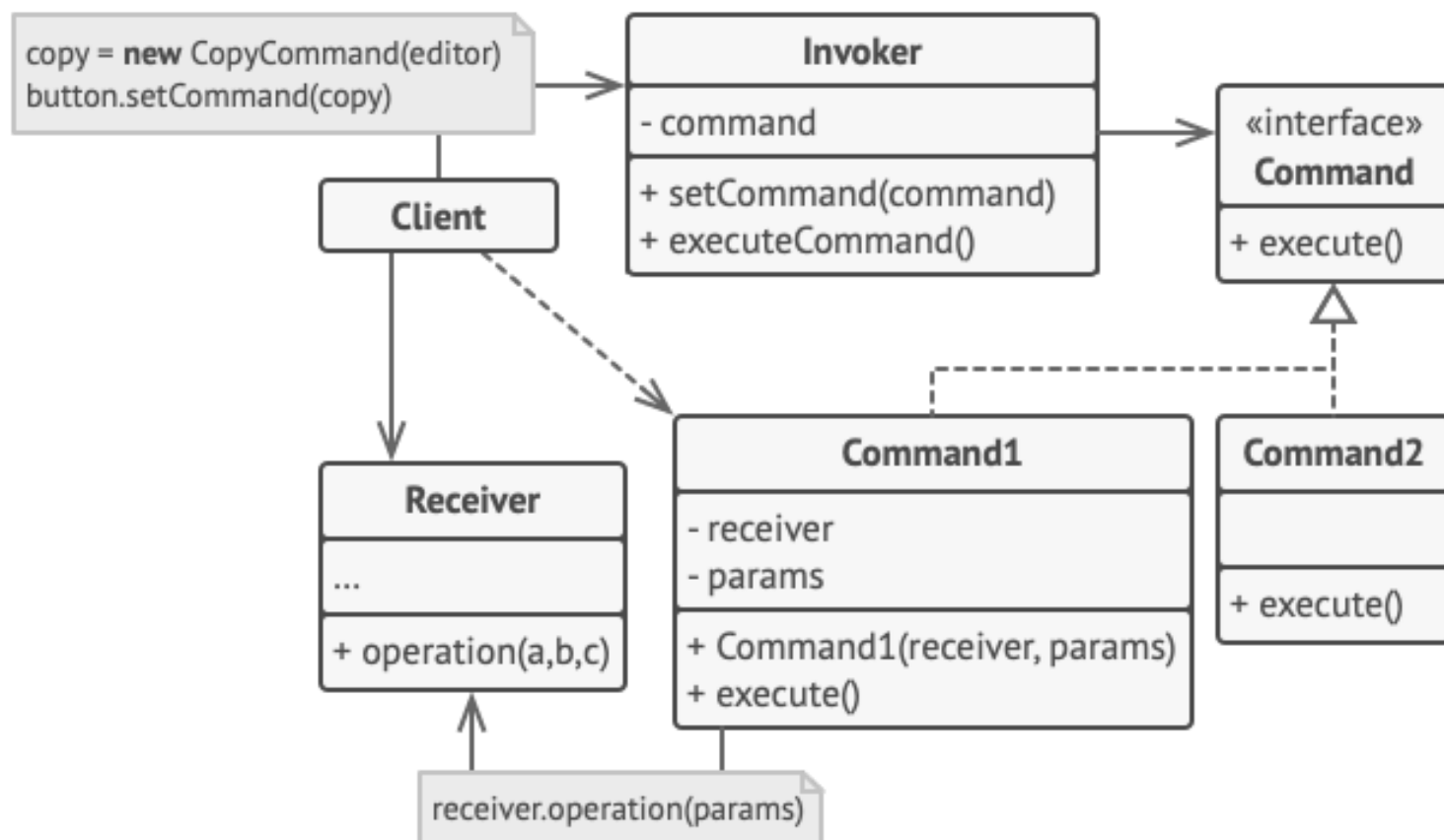
Prof. Richard Henrique de Souza

Objetivos

- Aplicar padrão comportamental ***Command*** em situação problema.

14. Command

Estrutura:



Importante

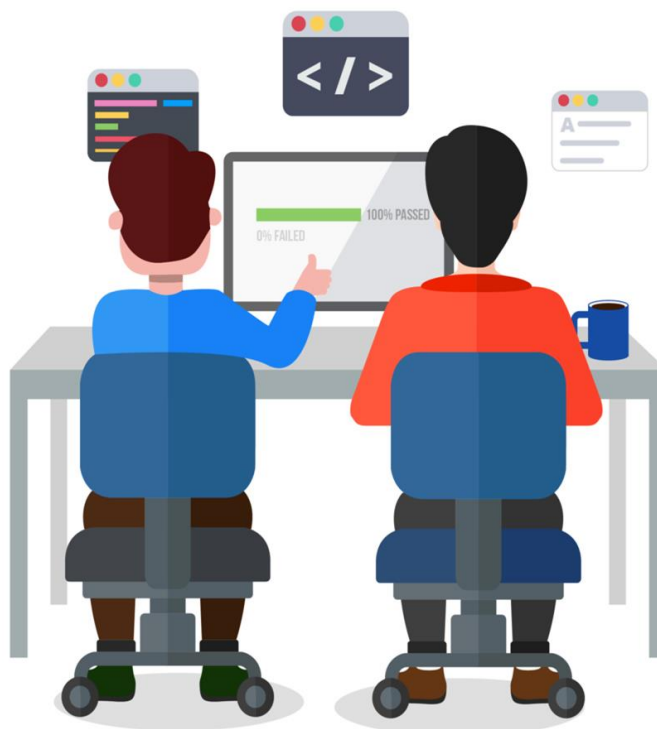
Siga os ROTEIROS !!!



Atividade em Grupo

Para esta atividade crie grupos de 2 alunos, para desenvolver a atividade segundo ***Pair Programming***.

Navegador



Piloto

Pair Programming

- Um é o **piloto**, responsável por escrever o código, o outro o navegador, acompanha a escrita de código e verificar se está de acordo com os **padrões do projeto** e de encontro à solução necessária.
- A intenção desta técnica é **evitar** erros de lógica, e ter um código mais confiável e melhor estruturado, utilizando-se para isso a máxima de que “**duas cabeças pensam melhor do que uma**”.

Preparação do ambiente



- Acesso a ferramenta **draw.io**(<https://app.diagrams.net/>) para realizar a modelagem.
- Escolha a sua linguagem de programação de preferência
- Escolha uma IDE ou o **git.dev**
- Crie um repositório no github(<https://github.com/>) para que todos os membros da equipe possam colaborar no desenvolvimento.



Atividade prática

1



14. Command

Propósito

- Encapsular uma requisição como um objeto, permitindo parametrização, enfileiramento, suporte a histórico, etc.
- Também conhecido como: Comando, Ação, Action, Transação, Transaction

14. Command

Intenção

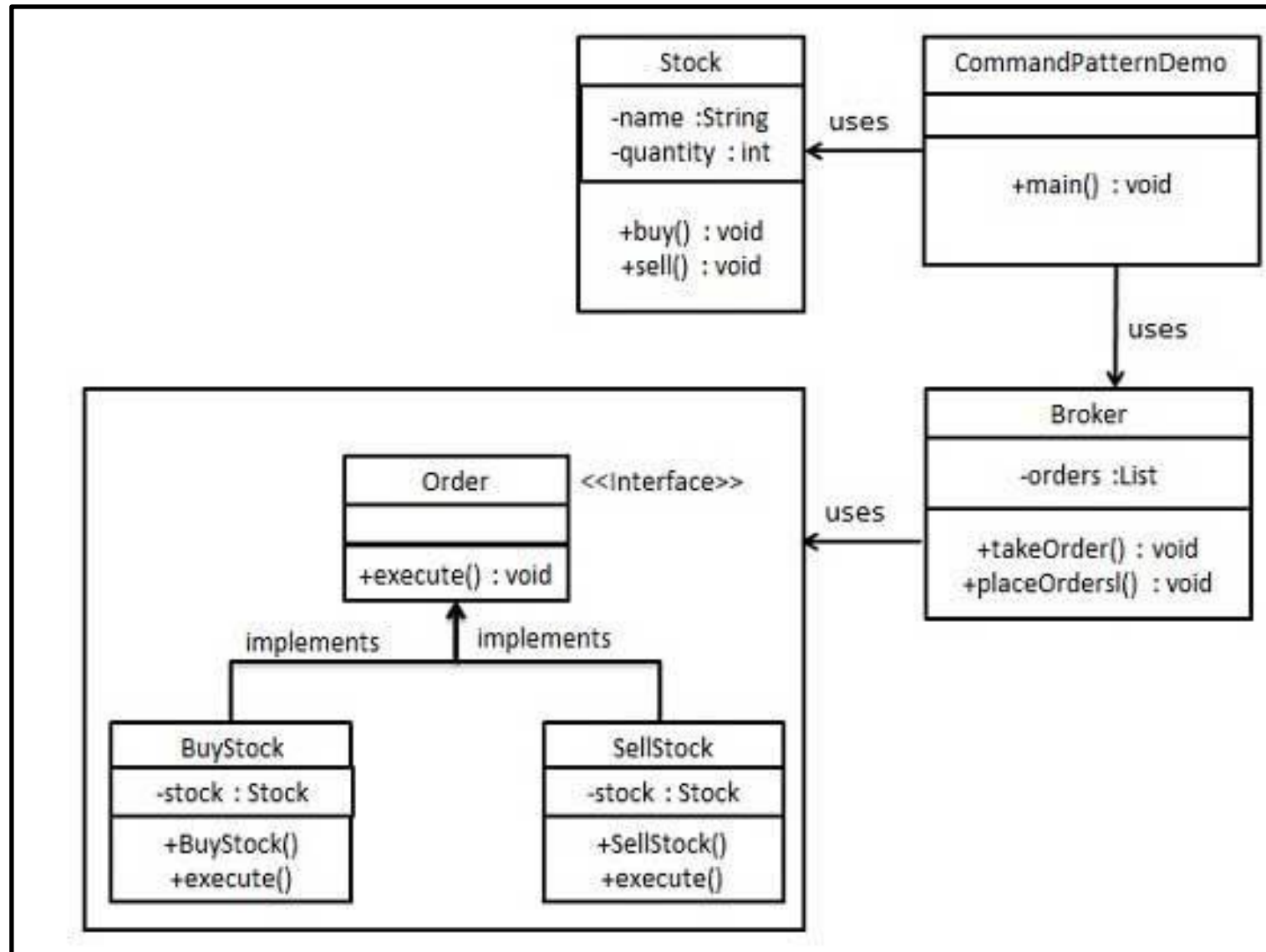
- Usar este padrão quando...
 - Quiser parametrizar ações genéricas;
 - Quiser enfileirar e executar comandos de forma assíncrona, em outro momento;
 - Quiser permitir o undo de operações, dando suporte a históricos;
 - Quiser fazer log dos comandos para refazê-los em caso de falha de sistema;
 - Quiser estruturar um sistema em torno de operações genéricas, como transações.

14. Command

Intenção

- Vantagens e desvantagens
 - Desacoplamento:
 - Objeto que evoca a operação e o que executa são desacoplados.
 - Extensibilidade:
 - Comandos são objetos, passíveis de extensão, composição, etc.;
 - Pode ser usado junto com Composite para formar comandos complexos;
 - É possível definir novos comandos sem alterar nada existente.

14. Command



14. Command

Passo 1

Crie uma interface de comando.

Order.java

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

14. Command

Passo 2

Crie uma classe de solicitação.

Stock.java

```
public class Stock {  
  
    private String name = "ABC";  
    private int quantity = 10;  
  
    public void buy(){  
        System.out.println("Stock [ Name: "+name+",  
            Quantity: " + quantity + " ] bought");  
    }  
    public void sell(){  
        System.out.println("Stock [ Name: "+name+",  
            Quantity: " + quantity + " ] sold");  
    }  
}
```

14. Command

Passo 3

Crie classes concretas implementando a interface Order.

BuyStock.java

```
public class BuyStock implements Order {  
    private Stock abcStock;  
  
    public BuyStock(Stock abcStock){  
        this.abcStock = abcStock;  
    }  
  
    public void execute() {  
        abcStock.buy();  
    }  
}
```

14. Command

Passo 3 - Continuação

SellStock.java

```
public class SellStock implements Order {  
    private Stock abcStock;  
  
    public SellStock(Stock abcStock){  
        this.abcStock = abcStock;  
    }  
  
    public void execute() {  
        abcStock.sell();  
    }  
}
```


14. Command

Passo 4

Criar classe de chamador de comando.

Broker.java

```
import java.util.ArrayList;
import java.util.List;

public class Broker {
    private List<Order> orderList = new ArrayList<Order>();

    public void takeOrder(Order order){
        orderList.add(order);
    }

    public void placeOrders(){
        for (Order order : orderList) {
            order.execute();
        }
        orderList.clear();
    }
}
```

14. Command

Passo 5

Use a classe Broker para obter e executar comandos.

CommandPatternDemo.java

```
public class CommandPatternDemo {  
    public static void main(String[] args) {  
        Stock abcStock = new Stock();  
  
        BuyStock buyStockOrder = new BuyStock(abcStock);  
        SellStock sellStockOrder = new SellStock(abcStock);  
  
        Broker broker = new Broker();  
        broker.takeOrder(buyStockOrder);  
        broker.takeOrder(sellStockOrder);  
  
        broker.placeOrders();  
    }  
}
```

14. Command

Passo 6

- Terminamos
 - Teste sua implementação



Compile e **Mostre** o código para o professor

- Pense, o que você fez aqui ?



Lembre de salvar no seu github



Atividade prática

2



14. Command

Restaurante da Objetolândia

- 1 - você, o freguês, transmite o seu Pedido à Garçonete.
- 2- A garçonete recebe o Pedido, coloca-o no balcão de pedidos e grita “Novo pedido!”.
- 3 - O cozinheiro prepara a sua refeição com base no Pedido.



14. Command

Agora, vamos examinar mais detalhadamente a interação...

e, já que o Restaurante fica em Objetolândia, vamos considerar também os objetos e as chamadas de métodos utilizadas!

14. Command

Freguês: “Quero um cheeseburger e um Milkshake”

O freguês sabe o que quer e gera o pedido:

`createOrder()`

O Pedido consiste em uma comanda na qual são escritos os itens de menu solicitados pelo freguês.

`takeOrder()`

...

14. Command

A garçonete recebe o pedido e, se estiver de bom humor, chama o método `orderUp()` para dar início à preparação do pedido `orderUp()`

O pedido contém todas as instruções necessárias para preparar a refeição e orienta o Cozinheiro usando métodos como `makeBurger()`.

`makeBurger()` , `makeShake()`

O cozinheiro segue as instruções do Pedido e prepara a refeição

14. Command

Papeis e Responsabilidade no Restaurante Objetolândia

Uma Comanda de Pedido encapsula a solicitação para preparar uma refeição

```
public void orderUp() {  
    cook.makeBurgue();  
    cook.makeShake();  
}
```

14. Command

Pense na Comanda de Pedido como um objeto que atua como uma solicitação para preparar uma refeição.

Como qualquer outro objeto, ele pode ser passado de um ponto para outro (da Garçonete para o balcão de pedidos ou para a Garçonete encarregada do próximo turno. Sua interface possui comente um método, `orderUp()`, que encapsula as ações necessárias para preparar a refeição.

Além disso, ele possui uma referência ao objeto que deverá executar a tarefa (no caso, o Cozinheiro).

Dizemos que ele está encapsulado porque a Garçonete não precisa saber qual é o conteúdo do pedido ou sequer quem preparará a refeição; tudo que ela precisa fazer é colocar a comanda no balcão de pedidos e gritar “novo pedido!”.

14. Command

Está bem, na vida real uma garçonete provavelmente se interessante em saber o que está na Comanda de Pedido e quem deverá cozinhar a refeição, mas vamos desconsiderar isso... afinal, estamos em Objetolândia !

14. Command

Vamos aplicar o padrão **Command** no Restaurante:
O cliente é responsável pela criação do objeto de comando, que consiste em um conjunto de ações num receptor.

`createCommandObject()`

As ações e o Receptor são vinculados no objeto de comando.

O objeto de comando fornece um método, `execute()`, que encapsula as ações e pode ser chamado para invocar as ações no Receptor

`execute()`

14. Command

O cliente chama `setCommand()` em um objeto invocador e passa o objeto de comando é armazenado no objeto invocador até se tornar necessário

`setCommand()`

Em algum momento no futuro, o Invocador chamará o método `execute()` do objeto de comando...

`execute()`

14. Command

... a qual resulta em ações que serão invocadas pelo Receptor

`action1(), action2()`

14. Command

a) Associe os objetos e métodos do restaurante aos nomes correspondentes no Padrão Comando.

Restaurante	Padrão Command
Garçone	Comando
Cozinheiro	execute()
orderUp()	Cliente
Pedido	Invocador
Freguês	Receptor
takeOrder()	setCommand()

14. Command

Chegou o momento de construirmos o nosso primeiro objeto de comando.

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


14. Command

Vamos simplificar

Agora, digamos que você queira implementar um comando para acender uma lâmpada.

A classe Light possui dois métodos: on() e off()

```
public class Light {  
    public void on() {  
        //LIGAR  
    }  
  
    public void off() {  
        //DESLIGAR  
    }  
}
```

14. Command

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

Implementação do comando:

```
public class LightOnCommand implements Command {  
    Light light;  
  
    public LightOnCommand (Light light) {  
        this.light = light;  
    }  
  
    public void execute() {  
        light.on();  
    }  
}
```

Como isto é um comando, temos que implementar a **interface** Command

O construtor recebe a luz específica que este comando deverá controlar (digamos, a luz da sala) e a armazena na variável de instância light. Quando o método `execute()` é chamado, este é o objeto light que será o Receptor da solicitação

O método `execute()` chama o método `on()` no objeto receptor, que é a luz que estamos controlando.

14. Command

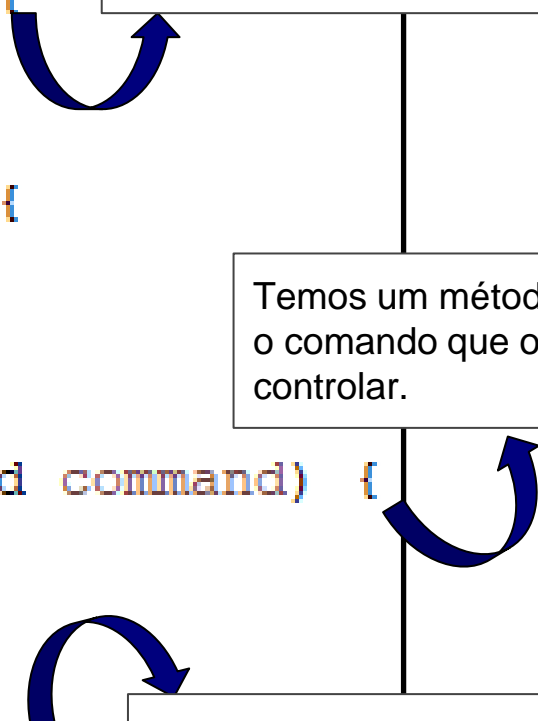
Vamos tentar manter as coisas simples. Nosso controle remoto agora tem apenas um botão e um slot correspondente para armazenar o que queremos controlar.

```
public class SimpleRemoteControl {  
    Command slot;  
  
    public SimpleRemoteControl() {  
  
    }  
  
    public void setCommand(Command command) {  
        slot = command;  
    }  
  
    public void buttonWasPressed() {  
        slot.execute();  
    }  
}
```

14. Command

```
public class SimpleRemoteControl {  
    Command slot;  
  
    public SimpleRemoteControl() {  
  
    }  
  
    public void setCommand(Command command) {  
        slot = command;  
    }  
  
    public void buttonWasPressed() {  
        slot.execute();  
    }  
}
```

Temos um slot para armazenar
nosso comando, que controlará
um dispositivo



Temos um método para definir
o comando que o slot deverá
controlar.

Este método é chamado quando o
botão é pressionado. tudo o que
fazemos é chamar o método
`execute()` do comando que
atualmente está associado ao slot

14. Command

Vamos testar?

b) Termine o código para fazer a luz “ligar”

Código visto até agora:

<https://github.com/richardunisul/Command.git>

```
public class RemoteControlTest {  
  
    public static void main(String[] args) {  
  
    }  
  
}
```

Vamos testar?

```
public class RemoteControlTest {  
  
    public static void main(String[] args) {  
  
        SimpleRemoteControl remote = new SimpleRemoteControl();  
        Light light = new Light();  
        LightOnCommand lightOn = new LightOnCommand(light);  
        remote.setCommand(lightOn);  
        remote.buttonWasPressed()  
  
    }  
}
```

Problems @ Javadoc Declaration Console

<terminated> RemoteControlTest [Java Application] C:\Program
ON -->A LUZ do Quarto foi LIGADA

```
public class Light {  
    public void on() {  
        System.out.println("ON -->A LUZ do Quarto foi LIGADA");  
    }  
  
    public void off() {  
        System.out.println("OFF -->A LUZ do Quarto foi DESLIGADA");  
    }  
}
```



Vamos ver o que está acontecendo

No Jargão do Padrão Command, este é o nosso Cliente

O Controle é o nosso Invocador: ele receberá um objeto de comando que poderá ser utilizado para fazer solicitações

Criamos um objeto Light, que será o receptor da solicitação

Aqui criamos o Comando e o passamos para o Receptor

Aqui passamos o comando para o Invocador

E, finalmente, nós simulamos o botão sendo pressionado

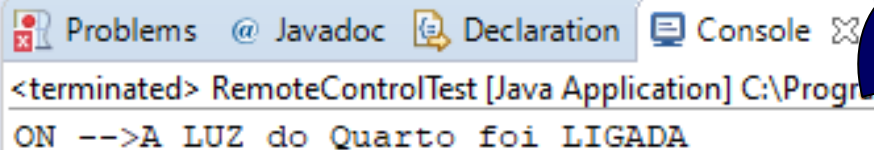
```
public class RemoteControlTest {  
  
    public static void main(String[] args) {  
  
        SimpleRemoteControl remote = new SimpleRemoteControl();  
        Light light = new Light();  
        LightOnCommand lightOn = new LightOnCommand(light);  
        remote.setCommand(lightOn);  
        remote.buttonWasPressed();  
  
    }  
}
```



Vamos ver o que está acontecendo

```
public class Light {  
    public void on() {  
        System.out.println("ON -->A LUZ do Quarto foi LIGADA");  
    }  
  
    public void off() {  
        System.out.println("OFF -->A LUZ do Quarto foi DESLIGADA");  
    }  
}
```

Vamos “Recheiar” o métodos `on()` e `off()`



Problems @ Javadoc Declaration Console

<terminated> RemoteControlTest [Java Application] C:\Program Files\Java\jdk-8.0.60\bin\java.exe
ON -->A LUZ do Quarto foi LIGADA

Aqui está o resultado da execução do código Teste

Controle Remoto

```
public class RemoteControl {
    Command[] onCommands;
    Command[] offCommands;

    public RemoteControl() {
        onCommands = new Command[7];
        offCommands = new Command[7];
        Command noCommand = new NoCommand();

        for(int i = 0; i < 7; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
    }

    public void setCommand (int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
    }

    public void offButtonWasPushed(int slot) {
        offCommands[slot].execute();
    }

    public String toString() {
        StringBuffer stringBuffer = new StringBuffer();
        stringBuffer.append("\n ----- Remote Control ----- \n");
        for(int i=0; i<onCommands.length; i++) {
            stringBuffer.append("[slot " + i + "] " + onCommands[i].getClass().getName() +
                " " + offCommands[i].getClass().getName() + " \n");
        }
        return stringBuffer.toString();
    }
}
```

c) Plano:

Vamos associar cada *slot* a um comando no controle remoto. Isto transformará o controle remoto no nosso invocador.

Controle Remoto

```
public class RemoteControl {  
    Command[] onCommands;  
    Command[] offCommands;  
  
    public RemoteControl() {  
        onCommands = new Command[7];  
        offCommands = new Command[7];  
        Command noCommand = new NoCommand();  
  
        for(int i = 0; i < 7; i++) {  
            onCommands[i] = noCommand;  
            offCommands[i] = noCommand;  
        }  
    }  
  
    public void setCommand (int slot, Command onCommand, Command offCommand) {  
        onCommands[slot] = onCommand;  
        offCommands[slot] = offCommand;  
    }  
  
    public void onButtonWasPushed(int slot) {  
        onCommands[slot].execute();  
    }  
  
    public void offButtonWasPushed(int slot) {  
        offCommands[slot].execute();  
    }  
  
    public String toString() {  
        StringBuffer stringBuffer = new StringBuffer();  
        stringBuffer.append("\n ----- Remote Control ----- \n");  
        for(int i=0; i<onCommands.length; i++) {  
            stringBuffer.append("[slot " +i+"] "+onCommands[i].getClass().getName()+  
                " "+ offCommands[i].getClass().getName() + "\n");  
        }  
        return stringBuffer.toString();  
    }  
}
```

Desta vez o controle remoto terá que lidar com sete comandos On e Off, que armazenaremos nos vetores correspondentes

Tudo o que precisamos fazer no construtor é criar instâncias e inicializar os vetores **on** e **off**

O método **setCommand()** recebe uma posição de slot e o comando On e Off para serem armazenados nesse slot. Ele armazena esses comandos nas matrizes on e off para uso posterior

Quando um botão On ou Off é pressionado, o hardware encarrega-se de chamar os métodos correspondentes, **onButtonWasPushed()** ou **offButtonWasPushed()**

Controle Remoto

```
public class RemoteControl {
    Command[] onCommands;
    Command[] offCommands;

    public RemoteControl() {
        onCommands = new Command[7];
        offCommands = new Command[7];
        Command noCommand = new NoCommand();

        for(int i = 0; i < 7; i++) {
            onCommands[i] = noCommand;
            offCommands[i] = noCommand;
        }
    }

    public void setCommand (int slot, Command onCommand, Command offCommand) {
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    public void onButtonWasPushed(int slot) {
        onCommands[slot].execute();
    }

    public void offButtonWasPushed(int slot) {
        offCommands[slot].execute();
    }

    public String toString() {
        StringBuffer stringBuff = new StringBuffer();
        stringBuff.append("\n ----- Remote Control ----- \n");
        for(int i=0; i<onCommands.length; i++) {
            stringBuff.append("[slot " +i+"] "+onCommands[i].getClass().getName()+
                " "+ offCommands[i].getClass().getName() + "\n");
        }
        return stringBuff.toString();
    }
}
```

Sobrescrevemos `toString()` para imprimir cada slot e o respectivo comando. Você verá isso em operação quando testarmos o controle remoto.

Implementando os comandos

- Já implementamos o LightOnCommand para o Controle Remoto Simples. Agora podemos conectar esse mesmo código aqui.

```
3 public class LightOffCommand implements Command {  
4     Light light;  
5  
6  
7     public LightOffCommand (Light light) {  
8         this.light = light;  
9     }  
10  
11     public void execute() {  
12         light.off();  
13     }  
14 }
```

O **LightOffCommand** funciona de maneira idêntica ao **LightOnCommand** exceto que nesse caso estamos vinculando o receptor a uma ação diferente: método **off()**.

Adicionando + 1

- Vamos escrever comandos para ligar e desligar o aparelho de som.

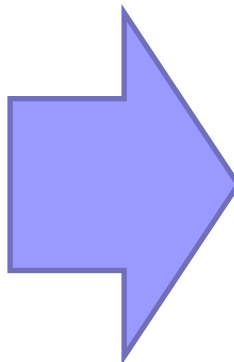
Desligar é fácil, basta vincular Stereo ao método **off()** em StereoOffCommand. Ligar é mais complicado, se tivermos várias opções como **StereoOnWithCDCommand....**

Stereo	
+	off()
+	on():
+	setCd()
+	setDVD()
+	setRadio()
+	setVolume()

Stereo

Stereo

- + off()
- + on()
- + setCd()
- + setDVD()
- + setRadio()
- + setVolume()



```
public class Stereo {  
  
    public void on() {  
        System.out.println("Stereo ON");  
    }  
  
    public void off() {  
        System.out.println("Stereo -- OFF");  
    }  
  
    public void setCD() {  
        System.out.println("Stereo -- CD mode");  
    }  
    public void setVolume(int v) {  
        System.out.println("Stereo -- VOLUME em  "+ v);  
    }  
  
    public void setDVD() {  
        System.out.println("Stereo -- DVD");  
    }  
  
    public void setRadio() {  
        System.out.println("Stereo -- Radio");  
    }  
  
}
```

Implementando...

```
public class StereoOnWithCDCommand implements Command{
    Stereo stereo;

    public StereoOnWithCDCommand(Stereo stereo) {
        this.stereo = stereo;
    }

    public void execute() {
        stereo.on();
        stereo.setCD();
        stereo.setVolume(11);
    }
}
```

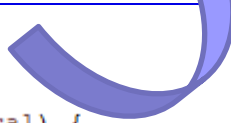
Como em **LightOnCommand**, recebemos a instância do aparelho de som que pretendemos controlar e a armazenamos numa variável de instância local

Para executar essa solicitação, precisamos chamar três métodos no aparelho de som: para ligá-lo, depois configurá-lo para tocar o CD e , finalmente, para ajustar o volume em 11. (Observação: 11 foi aleatório, o ideal é armazenar o volume e recuperar o último ajuste de volume....)

Vamos Testar....

```
public class StereoOffCommand implements Command {  
    Stereo stereo;  
  
    public StereoOffCommand (Stereo stereo) {  
        this.stereo = stereo;  
    }  
  
    public void execute() {  
        stereo.off();  
    }  
}
```

Mudamos a Classe Light para dizer qual é o ambiente (Local) da lâmpada.



```
public class Light {  
    private String local;  
  
    public Light(String local) {  
        this.local = local;  
    }  
  
    public void on() {  
        System.out.println("ON -->A LUZ do "+local+" foi LIGADA" );  
    }  
  
    public void off() {  
        System.out.println("OFF -->A LUZ do "+local+" foi DESLIGADA");  
    }  
}
```



```
public class RemoteLoader {
```

```
    public static void main(String[] args) {  
        RemoteControl remoteControl = new RemoteControl();  
  
        Light livingRoomLight = new Light("Living Room");  
        Light kitchenLight = new Light("Kitchen");  
        Stereo stereo = new Stereo();  
  
        LightOnCommand livingRoomLightOn = new LightOnCommand(livingRoomLight);  
        LightOffCommand livingRoomLightOff = new LightOffCommand(livingRoomLight);  
  
        LightOnCommand kitchenLightOn = new LightOnCommand(kitchenLight);  
        LightOffCommand kitchenLightOff = new LightOffCommand(kitchenLight);  
  
        StereoOnWithCDCommand stereoOnWithCD = new StereoOnWithCDCommand(stereo);  
        StereoOffCommand stereoOff = new StereoOffCommand(stereo);  
  
        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);  
        remoteControl.setCommand(1, kitchenLightOn, kitchenLightOff);  
        remoteControl.setCommand(2, stereoOnWithCD, stereoOff);  
  
        System.out.println(remoteControl);  
  
        remoteControl.onButtonWasPushed(0);  
        remoteControl.offButtonWasPushed(0);  
        remoteControl.onButtonWasPushed(1);  
        remoteControl.offButtonWasPushed(1);  
        remoteControl.onButtonWasPushed(2);  
        remoteControl.offButtonWasPushed(2);  
    }  
}
```

Vamos Testar....

Vamos Testar....

```
public class RemoteLoader {
```

```
    public static void main(String[] args) {
```

```
        RemoteControl remoteControl = new RemoteControl();
```

```
        Light livingRoomLight = new Light("Living Room");
```

```
        Light kitchenLight = new Light("Kitchen");
```

```
        Stereo stereo = new Stereo();
```

```
        LightOnCommand livingRoomLightOn = new LightOnCommand(livingRoomLight);
```

```
        LightOffCommand livingRoomLightOff = new LightOffCommand(livingRoomLight);
```

```
        LightOnCommand kitchenLightOn = new LightOnCommand(kitchenLight);
```

```
        LightOffCommand kitchenLightOff = new LightOffCommand(kitchenLight);
```

```
        StereoOnWithCDCommand stereoOnWithCD = new StereoOnWithCDCommand(stereo);
```

```
        StereoOffCommand stereoOff = new StereoOffCommand(stereo);
```

```
        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);
```

```
        remoteControl.setCommand(1, kitchenLightOn, kitchenLightOff);
```

```
        remoteControl.setCommand(2, stereoOnWithCD, stereoOff);
```

```
        System.out.println(remoteControl);
```

```
        remoteControl.onButtonWasPushed(0);
```

```
        remoteControl.offButtonWasPushed(0);
```

```
        remoteControl.onButtonWasPushed(1);
```

```
        remoteControl.offButtonWasPushed(1);
```

```
        remoteControl.onButtonWasPushed(2);
```

```
        remoteControl.offButtonWasPushed(2);
```

```
    }
```

```
}
```

Crie todos os dispositivos nos locais adequados

Crie todos os objetos de comando Light

Crie todos os objetos de comando para o aparelho de som

Agora que já temos todos os comandos, podemos carregá-los nos slots do controle remoto

Vamos Testar....

```
public class RemoteLoader {  
  
    public static void main(String[] args) {  
        RemoteControl remoteControl = new RemoteControl();  
  
        Light livingRoomLight = new Light("Living Room");  
        Light kitchenLight = new Light("Kitchen");  
        Stereo stereo = new Stereo();  
  
        LightOnCommand livingRoomLightOn = new LightOnCommand(livingRoomLight);  
        LightOffCommand livingRoomLightOff = new LightOffCommand(livingRoomLight);  
  
        LightOnCommand kitchenLightOn = new LightOnCommand(kitchenLight);  
        LightOffCommand kitchenLightOff = new LightOffCommand(kitchenLight);  
  
        StereoOnWithCDCommand stereoOnWithCD = new StereoOnWithCDCommand(stereo);  
        StereoOffCommand stereoOff = new StereoOffCommand(stereo);  
  
        remoteControl.setCommand(0, livingRoomLightOn, livingRoomLightOff);  
        remoteControl.setCommand(1, kitchenLightOn, kitchenLightOff);  
        remoteControl.setCommand(2, stereoOnWithCD, stereoOff);  
  
        System.out.println(remoteControl);  
  
        remoteControl.onButtonWasPushed(0);  
        remoteControl.offButtonWasPushed(0);  
        remoteControl.onButtonWasPushed(1);  
        remoteControl.offButtonWasPushed(1);  
        remoteControl.onButtonWasPushed(2);  
        remoteControl.offButtonWasPushed(2);  
    }  
}
```

Aqui usamos o nosso método `toString()` para imprimir cada slot do controle remoto e o comando associado a ele

Muito bem, estamos prontos para começar! Agora avançamos através de cada slot individual, pressionando seus botões **On** e **Off**

Vamos Testar....



```
Console Problems Debug Shell
<terminated> RemoteLoader [Java Application] C:\Program Files\Java\jre1.8.0_333\bin\javaw.exe (3 de mai de 2022 13:02:47)

----- Remote Control -----
[slot 0] c.LightOnCommand      c.LightOffCommand
[slot 1] c.LightOnCommand      c.LightOffCommand
[slot 2] c.StereoOnWithCDCommand c.StereoOffCommand
[slot 3] c.NoCommand           c.NoCommand
[slot 4] c.NoCommand           c.NoCommand
[slot 5] c.NoCommand           c.NoCommand
[slot 6] c.NoCommand           c.NoCommand

ON -->A LUZ do Living Room foi LIGADA
OFF -->A LUZ do Living Room foi DESLIGADA
ON -->A LUZ do Kitchen foi LIGADA
OFF -->A LUZ do Kitchen foi DESLIGADA
Stereo ON
Stereo -- CD mode
Stereo -- VOLUME em 11
Stereo -- OFF
```

Slots On e Off

Execução dos Comandos

14. Command

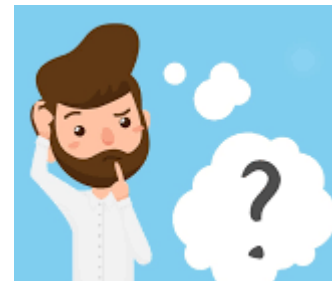
- Terminamos

- ☐ Teste sua implementação



Compile e **Mostre** o código para o professor

- ☐ Pense, o que você fez aqui ?



Lembre de salvar no seu github





Conclusão

Os padrões comportamentais tem como principal função designar responsabilidades entre objetos.

Referências

- PRESSMAN, Roger; MAXIM, Bruce. Engenharia de software: uma abordagem profissional. 8.ed. Bookman, 2016. E-book. Disponível em: <https://integrada.minhabiblioteca.com.br/books/9788580555349>
- SOMMERVILLE, Ian. Engenharia de software. 9. ed. São Paulo: Pearson Prentice Hall, 2011. E-book. Disponível em: <https://plataforma.bvirtual.com.br/Leitor/Publicacao/2613/epub/0>
- LARMAN, Craig. Utilizando UML e padrões: uma introdução à análise e ao projeto orientados a objetos e desenvolvimento iterativo. 3. ed Porto Alegre: Bookman, 2007. E-book. Disponível em: <https://integrada.minhabiblioteca.com.br/books/9788577800476>





Fim