Command

Padrão de Projeto

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Instituto Federal de Educaçãoo, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Campus Rio Grande
Divisão de Computação

Agenda

Introdução

- Como **encapsular uma invocação de um método**
- Como fazer com que o **objeto que invoca uma computação não se preocupe com os detalhes de como o processo é realizado**
- Como **salvar as invocações dos métodos para implementar do e undos em nossa aplicação**

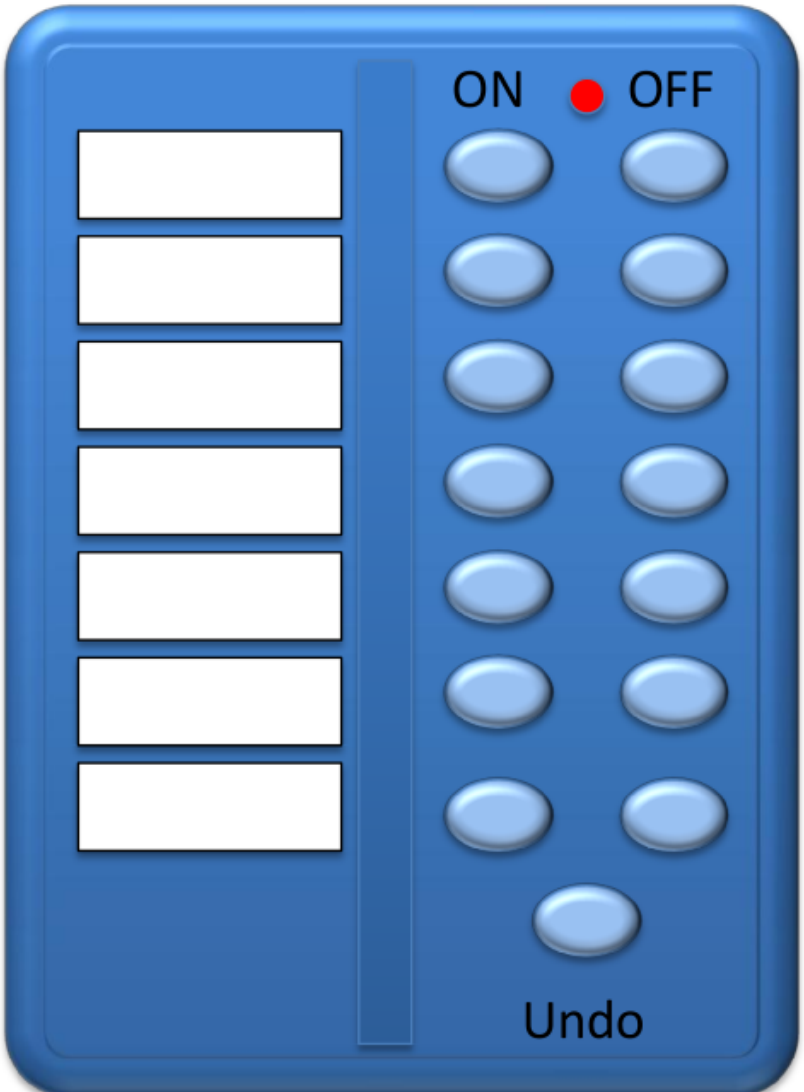
Problema

Considere a implementação de uma API para um controlador Universal com 7 slots programáveis, cada um como botões ON e OFF

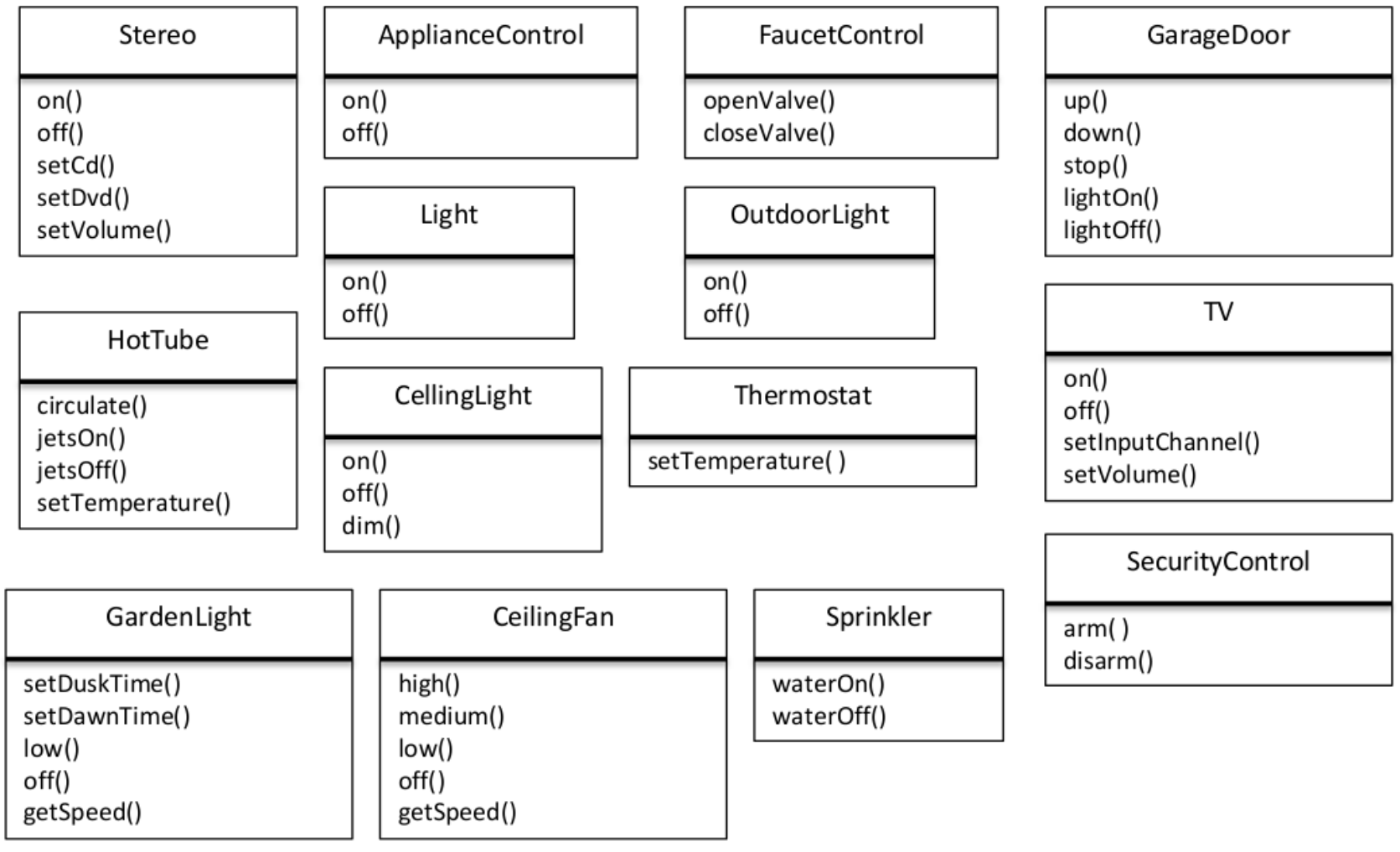
Requisitos Funcionais:

- RF1 - O controlador deve permitir a associação de diferentes dispositivos em cada slot
- RF2 - O controlador deve ser capaz de inicialmente controlar um conjunto pré-definido de dispositivos como luzes, ventiladores, equipamento de áudio e outros que estão especificados
- O controlador deve ser extensível, isto é, poder controlar novos dispositivos que os distribuidores possam a vir distribuir

Problema



Problema


Stereo
on()
off()
setCd()
setDvd()
setVolume()
ApplianceControl
on()
off()
FaucetControl
openValve()
closeValve()
GarageDoor
up()
down()
stop()
lightOn()
lightOff()
Light
on()
off()
OutdoorLight
on()
off()
HotTube
circulate()
jetsOn()
jetsOff()
setTemperature()
CellingLight
on()
off()
dim()
Thermostat
setTemperature( )
TV
on()
off()
setInputChannel()
setVolume()
SecurityControl
arm( )
disarm()
GardenLight
setDuskTime()
setDawnTime()
low()
off()
getSpeed()
CeilingFan
high()
medium()
low()
off()
getSpeed()
Sprinkler
waterOn()
waterOff()

Problema

Problemas

- Cada dispositivo tem sua própria interface
- As **interfaces variam muito** de dispositivo para dispositivo
- Impossível pré-determinar as interfaces dos novos dispositivos que os vendedores podem introduzir

Problema

Objetivo

- **Desacoplar quem requisita uma ação (o controle) de quem de fato executa a ação (o receptor, no caso, cada dispositivo)**

Como?

- Encapsular as operações e o receptor na ideia abstrata de comando com uma interface *execute()*
- O controlador passa a ser apenas um invocador do comando; não sabe como fazer, nem quem de fato faz

Problema

Refinamento da abordagem

- Encapsular as operações (ligar e desligar) de um dispositivo em um commando (*command*) que conhece quem sabe executar (*receiver*)
- Tornar o controlador apenas um invocador (*invoker*) de comandos (ele passa a não conhecer como e quem faz o comando)
- O cliente cria o comando, o configura e passa ao invocador (*invoker*)
- Quando acionado, o invocador dispara (*execute()*) o comando que é de fato executando em detalhes pelo *receiver*, efetuando as operações descritas no comando

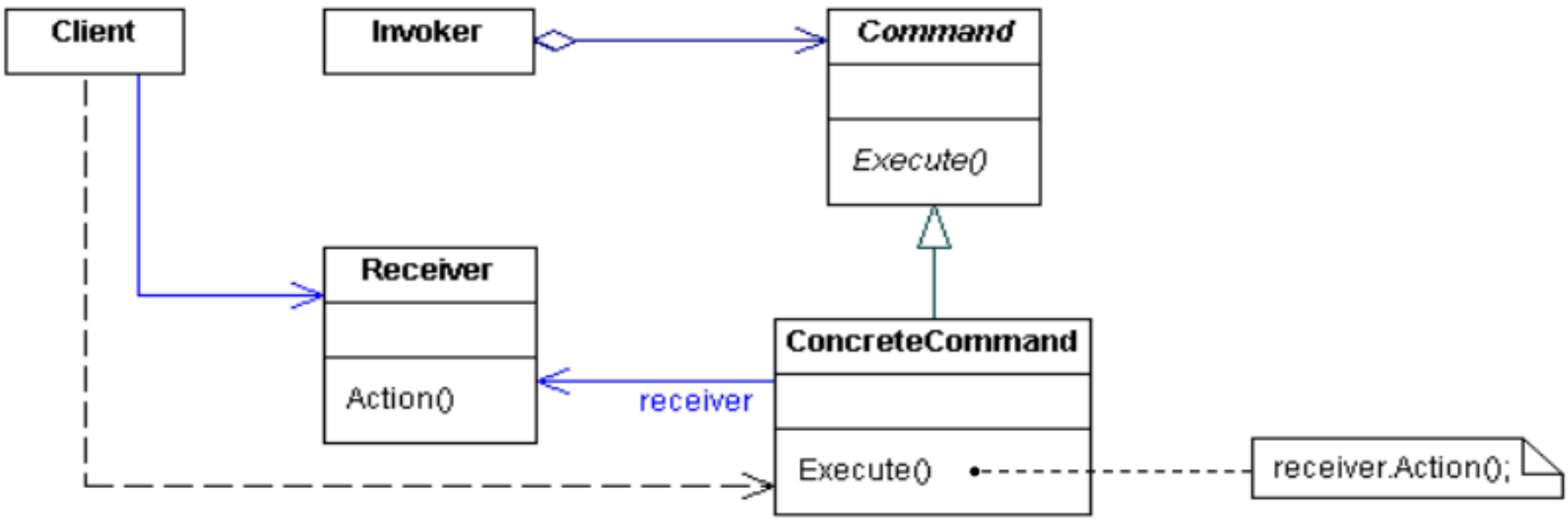
O Padrão Command

O Padrão Command

O Padrão Command encapsula uma requisição como um objeto, permitindo parametrizar outros objetos com diferentes requisições, filas ou log de requisições, dando suporte a operações que possam ser desfeitas

O Padrão Command

Diagrama de Classes



O Padrão Command

- **Command**
  - Define a interface para a execução de uma operação
- **ConcreteCommand**
  - Define uma vinculação entre o objeto Receiver e uma ação
  - Implementa *execute()* através da invocação da(s) correspondente(s) operação(ões) no Receiver
- **Client**
  - Cria um objeto **ConcreteCommand** e estabelece o seu receptor (Receiver)
- **Invoker**
  - Solicita ao Command a execução da solicitação
- **Receiver**
  - Sabe como executar as operações associadas a uma solicitação. Qualquer classe pode funcionar como um receiver

Solução para o Controle Remoto usando o Padrão Command

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

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

  public LightOnCommand(Light light) {
    this.light = light;
  }

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

```
public class SimpleRemoteControl {

  Command slot;

  public SimpleRemoteControl() {}

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

Solução para o Controle Remoto usando o Padrão Command

```
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();
}
```

Solução para o Controle Remoto usando o Padrão Command

Vários Slots

```
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();
  }
}
```

Solução para o Controle Remoto usando o Padrão Command

Vários Slots

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

  public LightOffCommand(Light light) {
    this.light = light;
  }

  public void execute() {
    light.off();
  }
}
```

```
public class StereoOnWithCDCommand implements
Command {
  Stereo stereo;
  public StereoOnWithCDCommand(Stereo   stereo){
     this.stereo = stereo;
  }

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

Solução para o Controle Remoto usando o Padrão Command

Como fazer um undo (desfazer)

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

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

  public LightOffCommand(Light light) {
    this.light = light;
  }

  public void execute() {
    light.off();
  }

  public void undo() {
    light.on();
  }
}
```

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

  public LightOnCommand(Light light) {
    this.light = light;
  }

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

  public void undo() {
    light.off();
  }
}
```

Solução para o Controle Remoto usando o Padrão Command

Como fazer um undo (desfazer)

```
public class RemoteControlWithUndo {
  Command[] onCommands;
  Command[] offCommands;
  Command undoCommand;

  public RemoteControlWithUndo() {
    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();
    undoCommand = onCommands[slot];
  }

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

  public void undoButtonWasPushed(){
    undoCommand.execute();
  }

  public String toString() {
   // toString code here...
  }
}
```

Solução para o Controle Remoto usando o Padrão Command

Como fazer um undo para objetos com estados

```
public class CeilingFan {
  public static final int HIGH = 3;
  public static final int MEDIUM = 2;
  public static final int LOW = 1;
  public static final int OFF = 0;
  String location; int speed;

  public CeilingFan(String location) {
     this.location = location;
     speed = OFF;
  }
  public void high() {
    speed = HIGH;
  }
  public void medium() {
   speed = MEDIUM;
  }
  public void low() {
   speed = LOW;
  }
  public void off() {
    speed = OFF;
  }
  public int getSpeed() {
    return speed;
  }
```

Solução para o Controle Remoto usando o Padrão Command

Como fazer um undo para objetos com estados

```
public class CeilingFanHighCommand implements Command {
  CeilingFan ceilingFan;
  int prevSpeed;

  public CeilingFanHighCommand(CeilingFan ceilingFan) {
    this.ceilingFan = ceilingFan;
  }

  public void execute() {
    prevSpeed = ceilingFan.getSpeed();
    ceilingFan.high();
  }

  public void undo() {
    if (prevSpeed == CeilingFan.HIGH) {
      ceilingFan.high();
    } else if (prevSpeed == CeilingFan.MEDIUM) {
      ceilingFan.medium();
    } else if (prevSpeed == CeilingFan.LOW) {
      ceilingFan.low();
    } else if (prevSpeed == CeilingFan.OFF) {
      ceilingFan.off();
    }
  }
}
```

Solução para o Controle Remoto usando o Padrão Command

Criando uma Macro

```
public class MacroCommand implements Command {
  Command[] commands;

  public MacroCommand(Command[] commands) {
     this.commands = commands;
  }

  public void execute() {
    for (int i = 0; i < commands.length; i++) {
        commands[i].execute();
    }
  }
}
```

Solução para o Controle Remoto usando o Padrão Command

Exemplo de código para o Padrão 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);
  }
}
```

Outro Problema

Suponha uma loja que vende produtos e oferece várias formas de pagamento.

Ao executar uma compra o sistema registra o valor total e, dada uma forma de pagamento, por exemplo, cartão de crédito, emite o valor total da compra para o cartão de crédito do cliente.

Outro Problema

Para este caso então vamos supor as seguintes classes para simplificar o exemplo: **Loja** e **Compra**.

A classe Loja representa a loja que está efetuando a venda.

Outro Problema

Vamos deixar a classe Loja bem simples, como mostra o seguinte código:

```
1  public class Loja {
2      protected String nomeDaLoja;
3
4      public Loja(String nome) {
5          nomeDaLoja = nome;
6      }
7
8      public void executarCompra(double valor) {
9          Compra compra = new Compra(nomeDaLoja);
10         compra.setValor(valor);
11     }
12 }
```

Outro Problema

Para focar apenas no que é necessário para entender o padrão, vamos utilizar a classe Compra, que representa o conjunto de produtos que foram vendidos com o seu valor total:

Outro Problema

```
1  public class Compra {
2      private static int CONTADOR_ID;
3      protected int idNotaFiscal;
4      protected String nomeDaLoja;
5      protected double valorTotal;
6
7      public Compra(String nomeDaLoja) {
8          this.nomeDaLoja = nomeDaLoja;
9          idNotaFiscal = ++CONTADOR_ID;
10     }
11
12     public void setValor(double valor) {
13         this.valorTotal = valor;
14     }
15
16     public String getInfoNota() {
17         return new String("Nota fiscal nº: " + idNotaFiscal + "'
18                 + nomeDaLoja + "\nValor: " + valorTotal);
19     }
20 }
```

Pronto, agora precisamos alterar a classe Loja para que ela, ao executar uma compra saiba qual a forma de pagamento.

Outro Problema

Uma maneira interessante de fazer esta implementação seria adicionando um parâmetro a mais no método executar que nos diga qual forma de pagamento deve ser usada.

Poderíamos então utilizar um Enum para identificar a forma de pagamento e daí passar a responsabilidade ao objeto específico.

Outro Problema

Veja o exemplo que mostra o código do método executar compra utilizando uma enumeração:

```
1  public void executarCompra(double valor, FormaDePagamento forma
2      Compra compra = new Compra(nomeDaLoja);
3      compra.setValor(valor);
4      if(formaDePagamento == FormaDePagamento.CartaoDeCredito){
5          new PagamentoCartaoCredito().processarCompra(compra);
6      } else if(formaDePagamento == FormaDePagamento.CartaoDeDebi
7          new PagamentoCartaoDebito().processarCompra(compra);
8      } else if(formaDePagamento == FormaDePagamento.Boleto){
9          new PagamentoBoleto().processarCompra(compra);
10     }
11 }
```

Outro Problema

O problema desta solução é que, caso seja necessário incluir ou remover uma forma de pagamento precisaremos fazer várias alterações, alterando tanto a enumeração quando o método que processa a compra.

Veja também a quantidade de ifs aninhados, isso é um sintoma de um design mal feito.

Outro Problema

Poderíamos passar o objeto que faz o pagamento como um dos parametros, ao invés de utilizar a enumeração, assim não teríamos mais problemas com os ifs aninhados e as alterações seriam locais.

Ok, está é uma boa solução, mas ainda não está boa, pois precisaríamos de um método diferente pra cada tipo de objeto.

A saída obvia então é utilizar uma classe comum a todos as formas de pagamento, e no parâmetro passar um objeto genérico! Essa é justamente a ideia do Padrão Command.

Outro Problema

Para resolver o exemplo acima vamos encapsular as solicitações de pagamento de uma compra em objetos para parametrizar os clientes com as diferentes solicitações.

Outro Problema

Perceba no código com os vários ifs é um local suscetível a refatoração de código.

Dentro de cada um dos if, a ação é a mesma: criar um objeto para processar o pagamento e realizar uma chamada ao método de processamento da compra.

Outro Problema

Então vamos primeiro definir a interface comum aos objetos que processam um pagamento.

Todos eles possuem um mesmo método, processar pagamento, que toma como parâmetro uma compra e faz o processamento dessa compra de várias formas.

A classe interface seria a seguinte:

```
1  public interface PagamentoCommand {
2      void processarCompra(Compra compra);
3  }
```

Outro Problema

Uma possível implementação seria a de processar um pagamento via boleto.

Vamos apenas emitir uma mensagem no terminal para saber que tudo foi executado como esperado:

```
1  public class PagamentoBoleto implements PagamentoCommand {
2
3      @Override
4      public void processarCompra(Compra compra) {
5          System.out.println("Boleto criado!\n" + compra.getInfoNo
6      }
7
8  }
```

Outro Problema

Agora o método de execução da compra na classe Loja seria assim:

```
1  public void executarCompra(double valor, PagamentoCommand formaDe
2      Compra compra = new Compra(nomeDaLoja);
3      compra.setValor(valor);
4      formaDePagamento.processarCompra(compra);
5  }
```

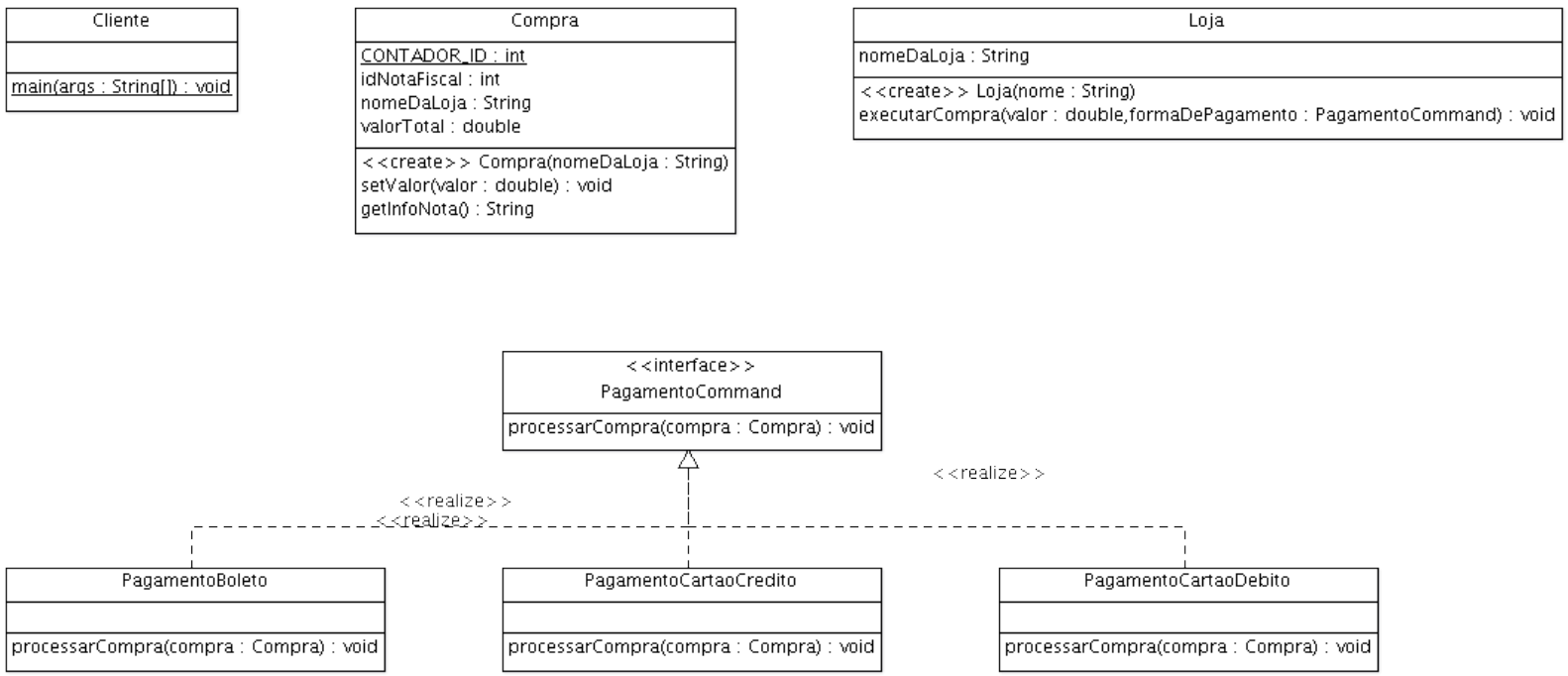
Outro Problema

O código cliente que usaria o padrão Command seria algo do tipo:

```
1  public static void main(String[] args) {
2      Loja lojasAfricanas = new Loja("Afriacanas");
3      lojasAfricanas.executarCompra(999.00, new PagamentoCartaoCr
4      lojasAfricanas.executarCompra(49.00, new PagamentoBoleto())
5      lojasAfricanas.executarCompra(99.00, new PagamentoCartaoDeb
6
7      Loja exorbitante = new Loja("Exorbitante");
8      exorbitante.executarCompra(19.00, new PagamentoCartaoCredit
9
10 }
```

Ao executar uma compra nós passamos o comando que deve ser utilizado, neste caso, a forma de pagamento utilizado.

Outro Problema



Trabalho

Trata-se de uma abstração de um joystick.

O joystick utiliza o padrão command para interagir com diferentes jogos.

- O joystick possui 2 botões:
  - A
  - B
- Os receptores são os jogos, que executarão diferentes ações para a mesma tecla pressionada.
  - Exemplo:
    - o botão A faz correr no jogo de futebol, acelera no jogo de corrida e chuta alto no jogo de luta

Trabalho

Figura: Controle Xbox

- A = Pode funcionar como um chute (futebol) ou soco (luta)
- B = Pode funcionar como passe (futebol) ou tiro (primeira pessoa)

Command

Padrão de Projeto

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Instituto Federal de Educaçãoo, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Campus Rio Grande
Divisão de Computação