

# Padrões de Projeto

Prof. Marum Simão Filho

# Agenda

- Padrão Command
- Padrão Decorator
- Padrão Facade

# Padrão Command

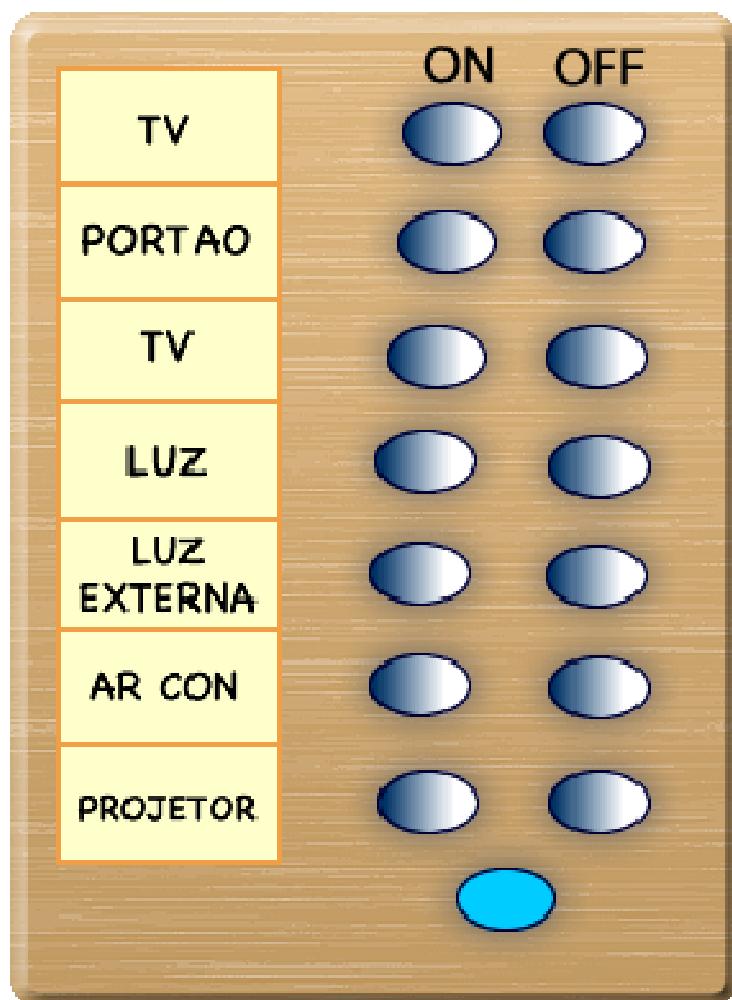


# O padrão Command

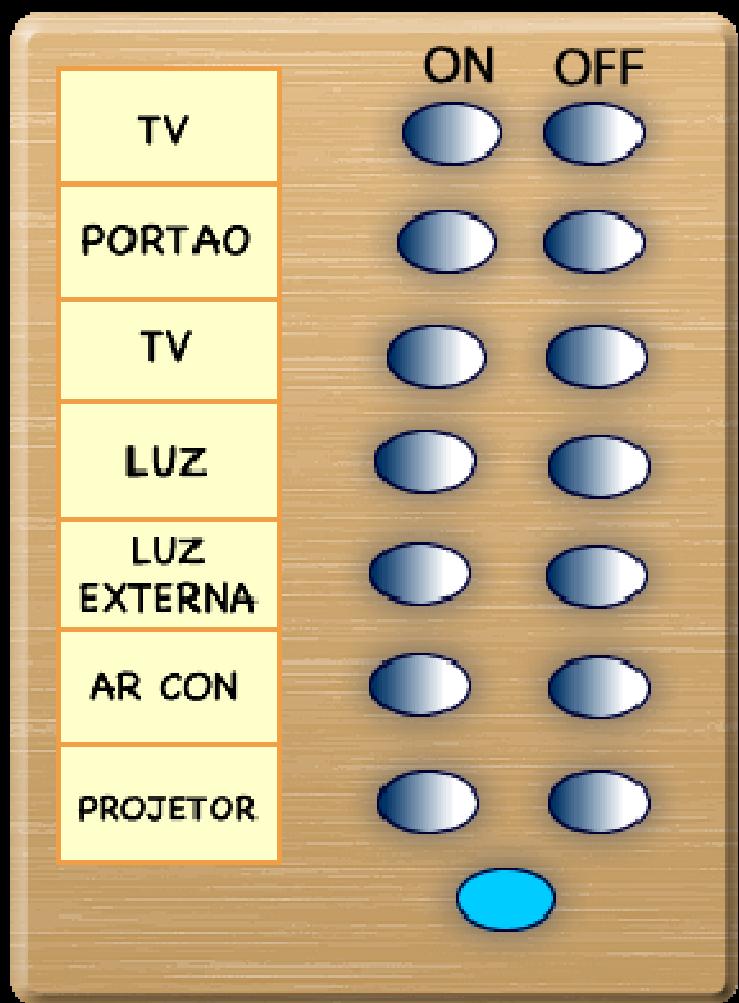
- Nossa motivação:

- Algumas vezes é necessário emitir solicitações para objetos sem nada saber sobre a operação que está sendo solicitada ou sobre o receptor da mesma.

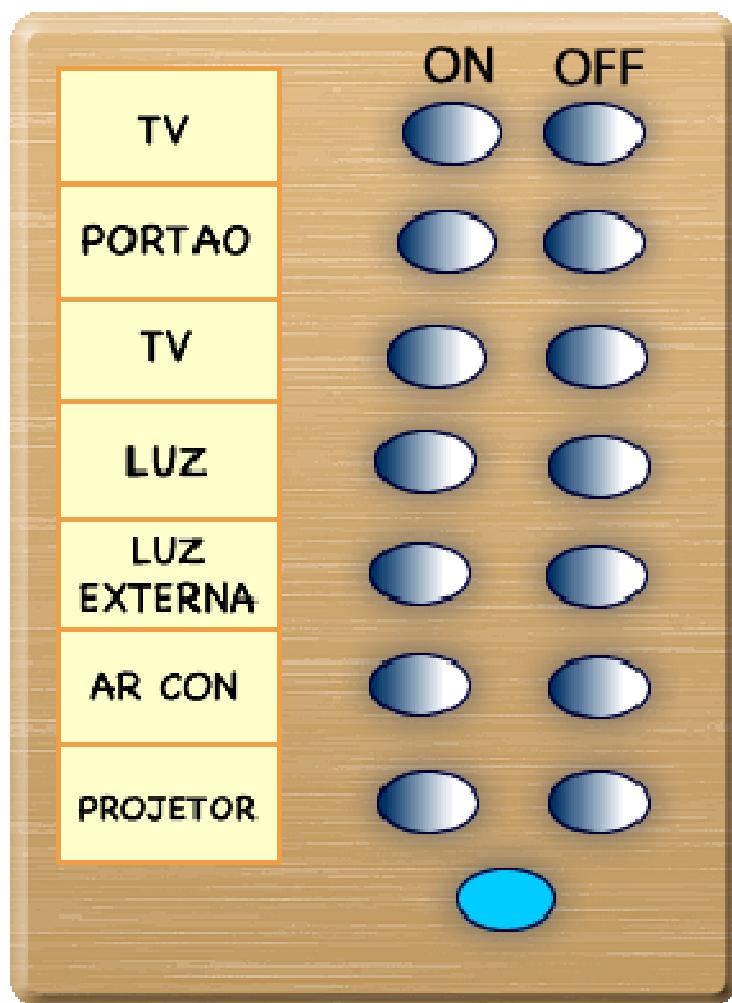
# Motivação



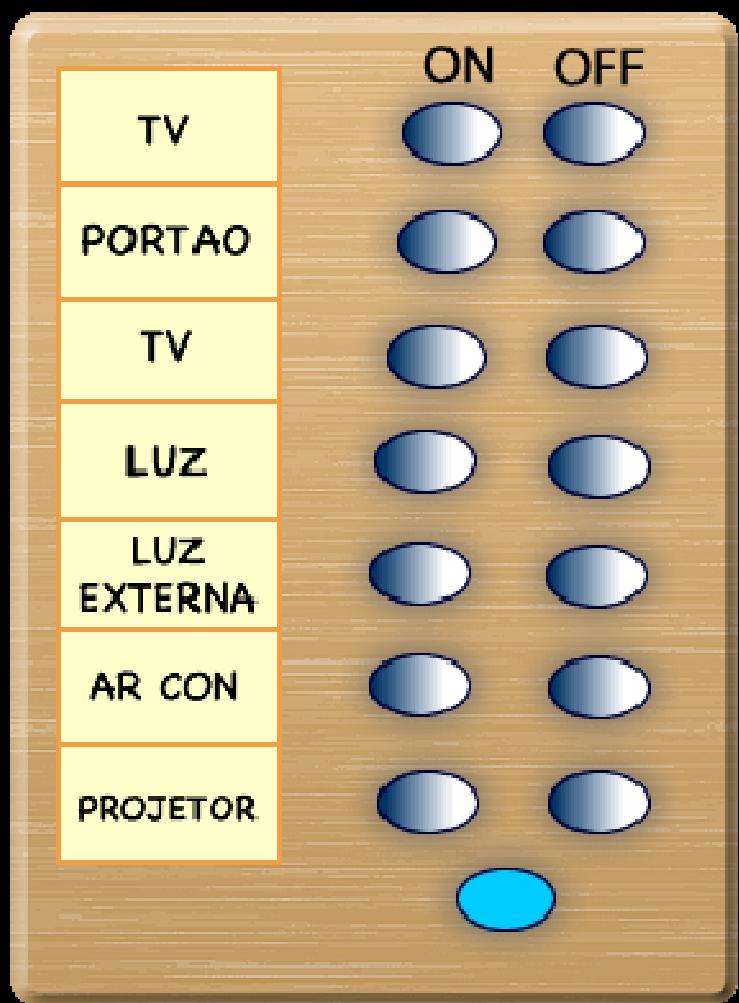
# Motivação



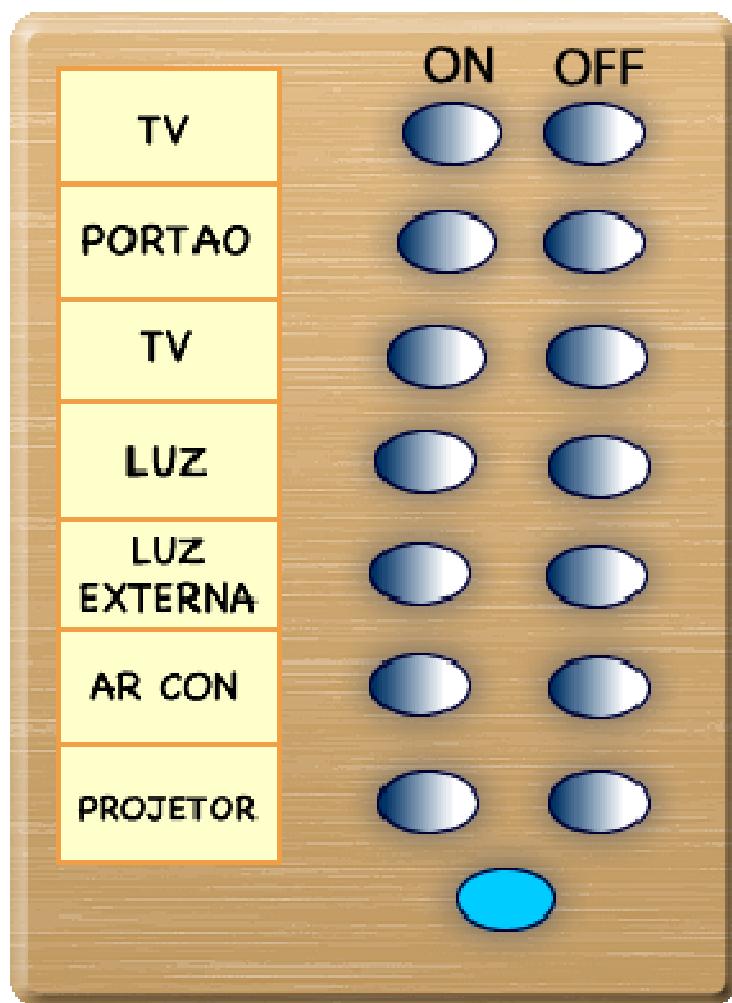
# Motivação



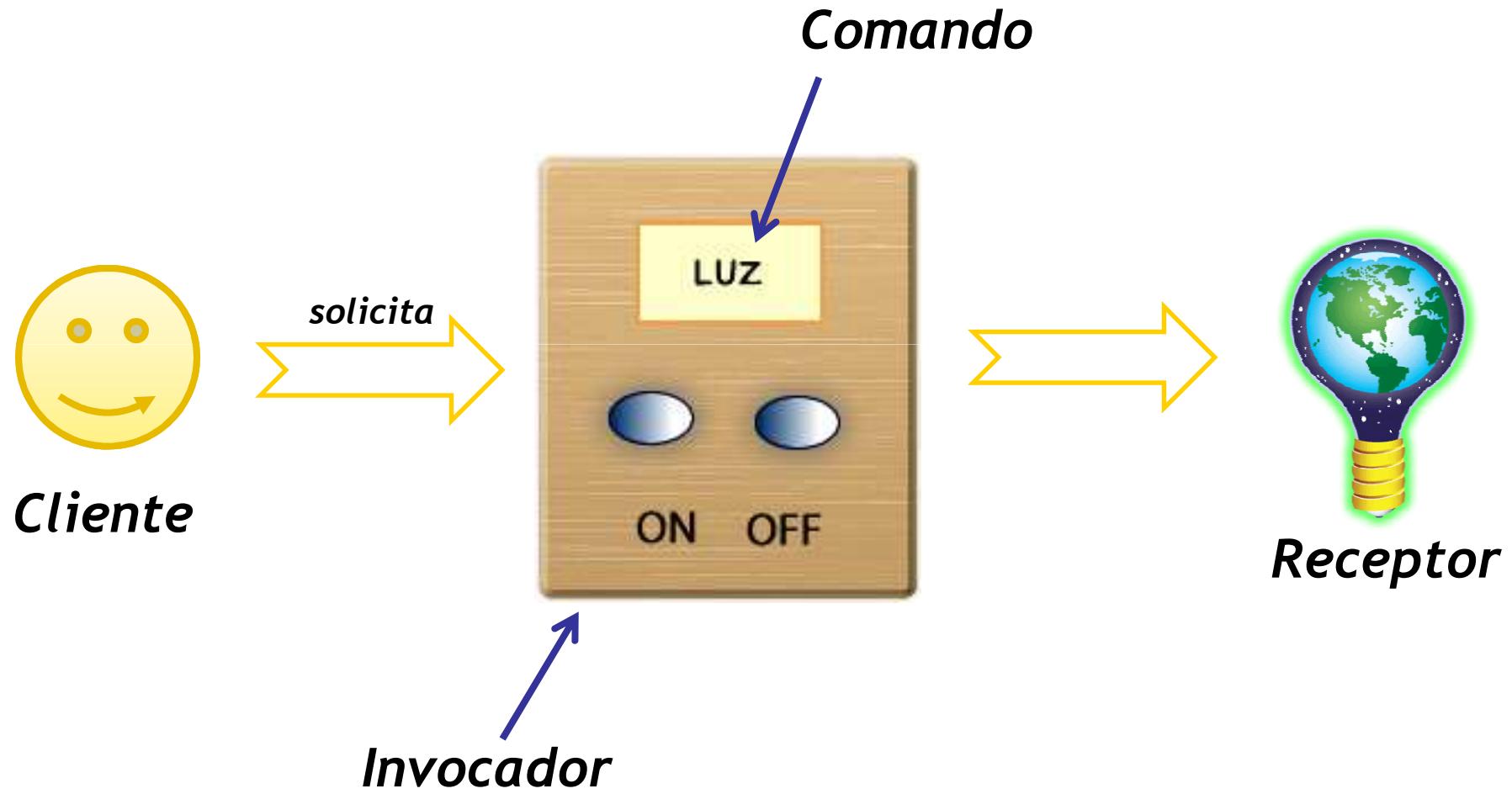
# Motivação



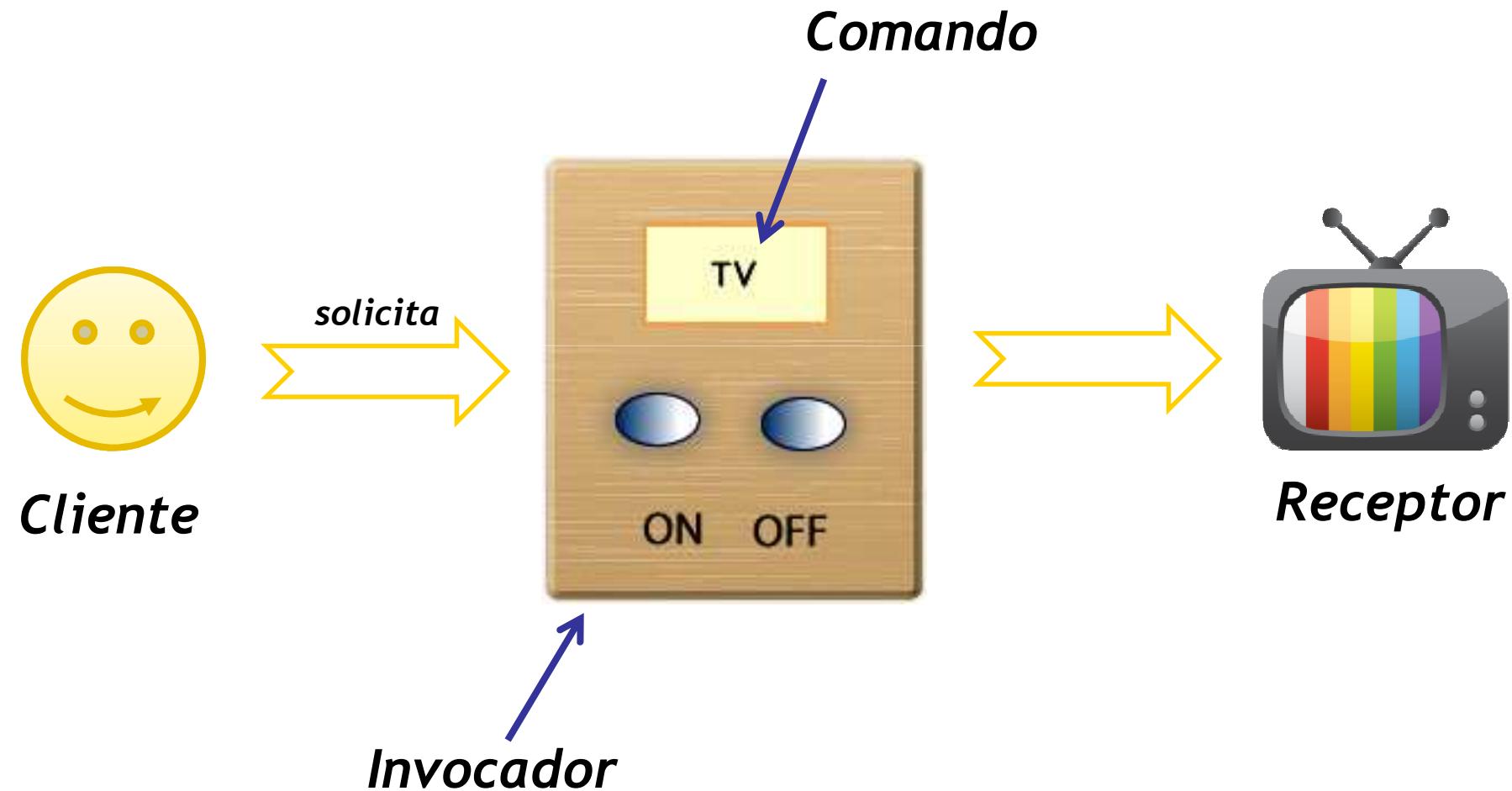
# Motivação



# Simplificando



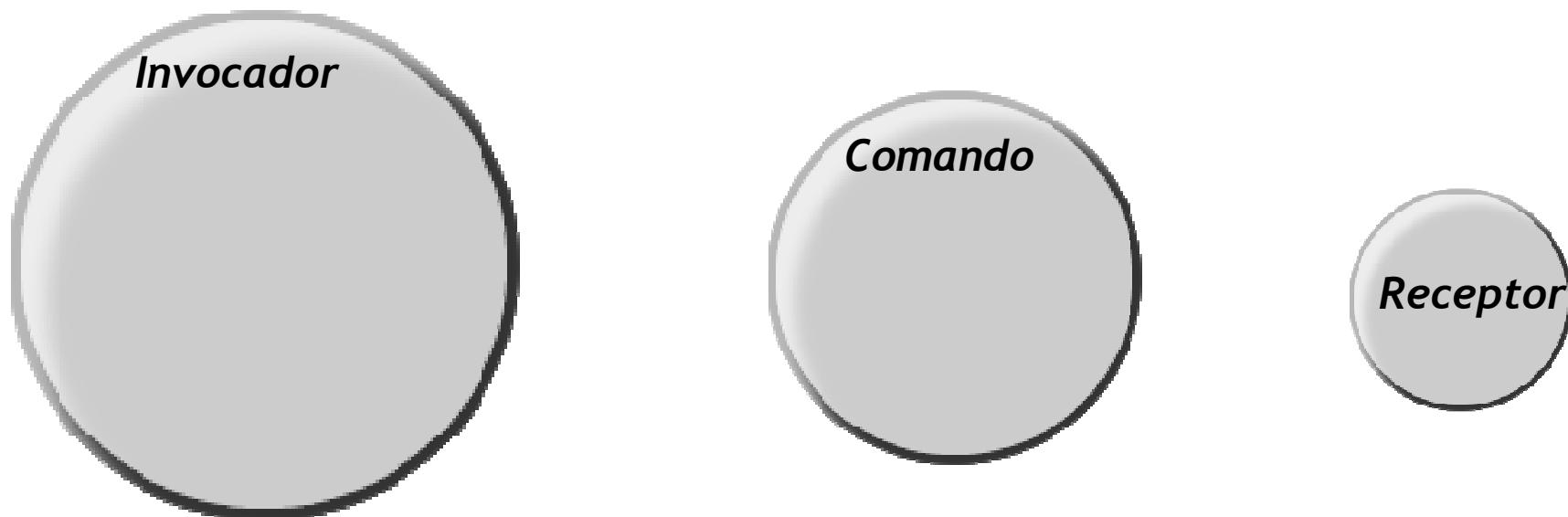
# Simplificando



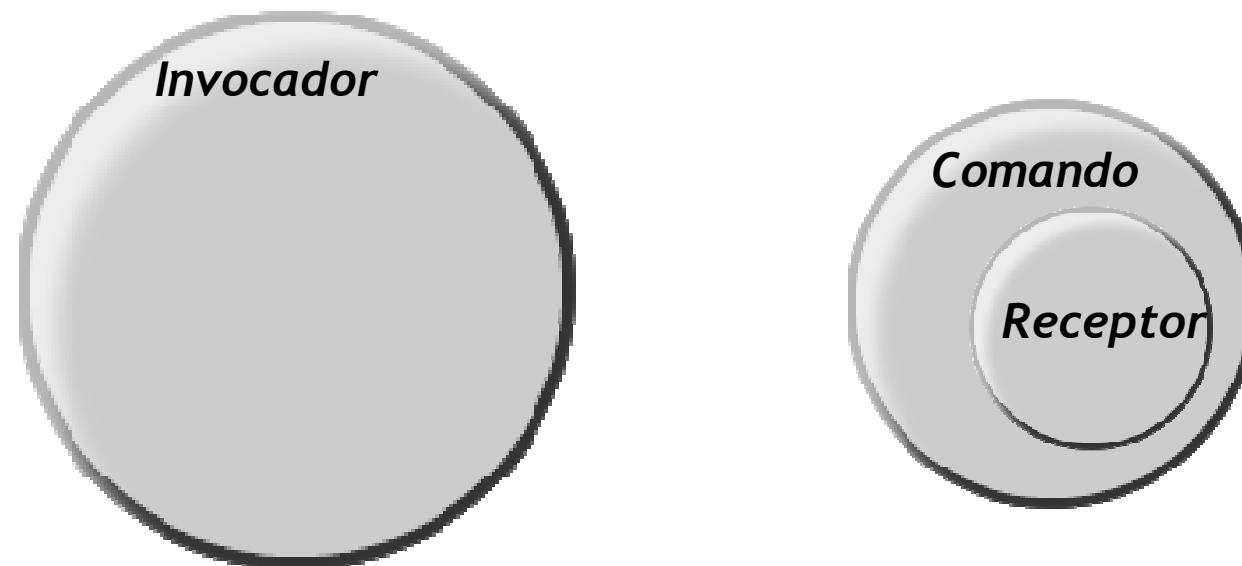
# Padrão Comando Participantes



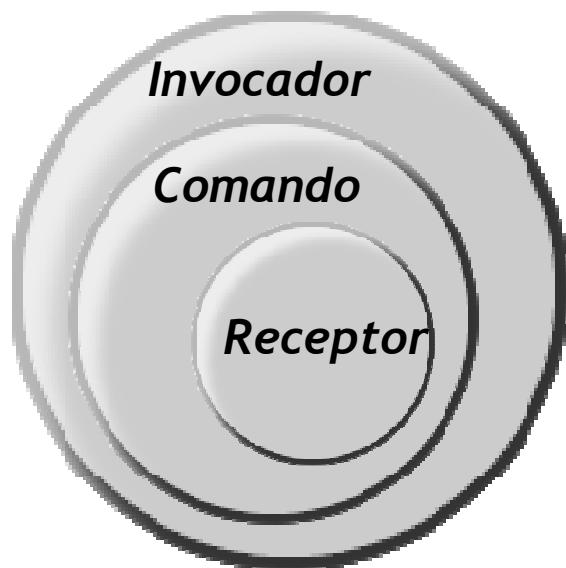
# Estrutura



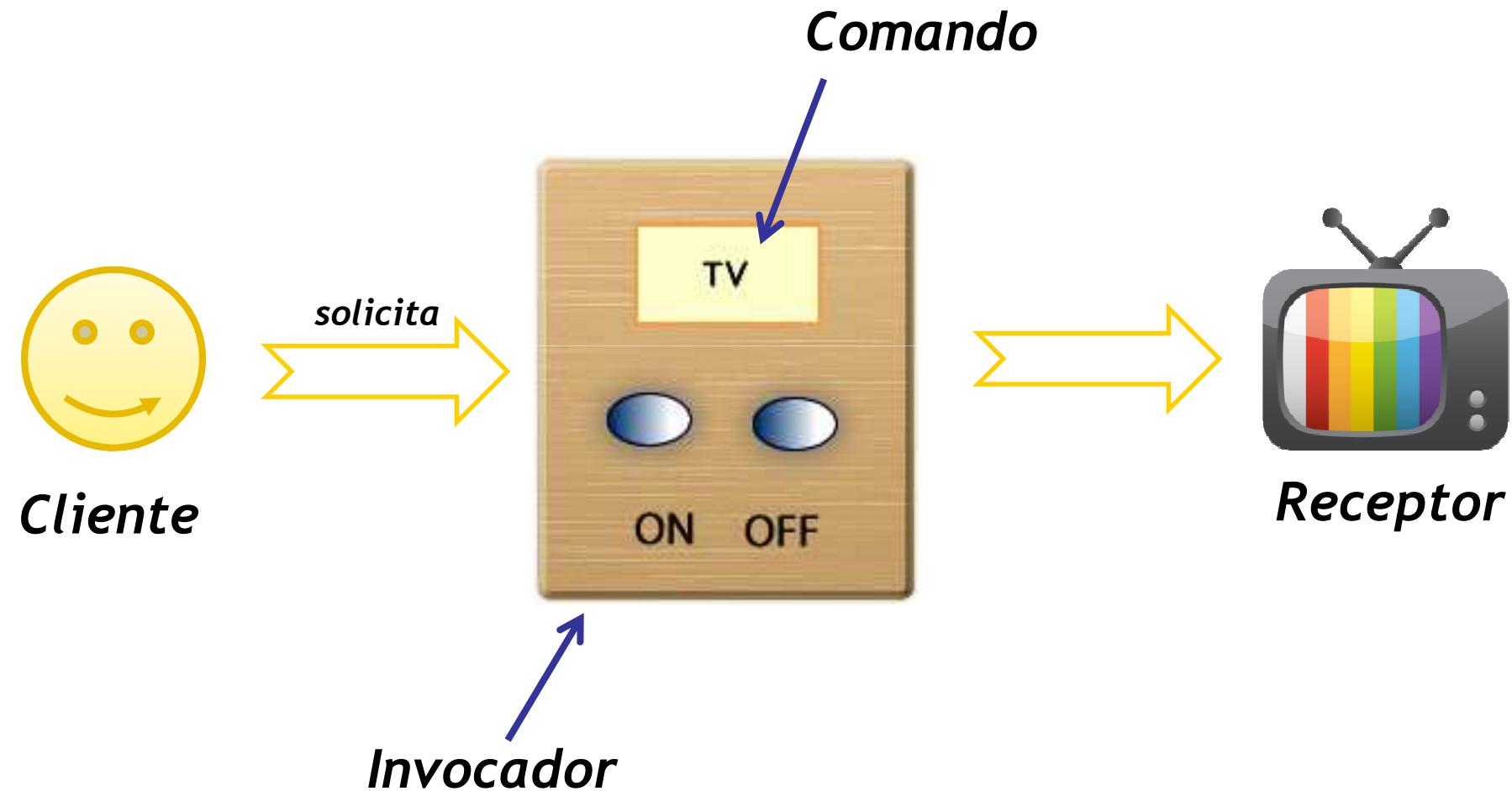
# Estrutura



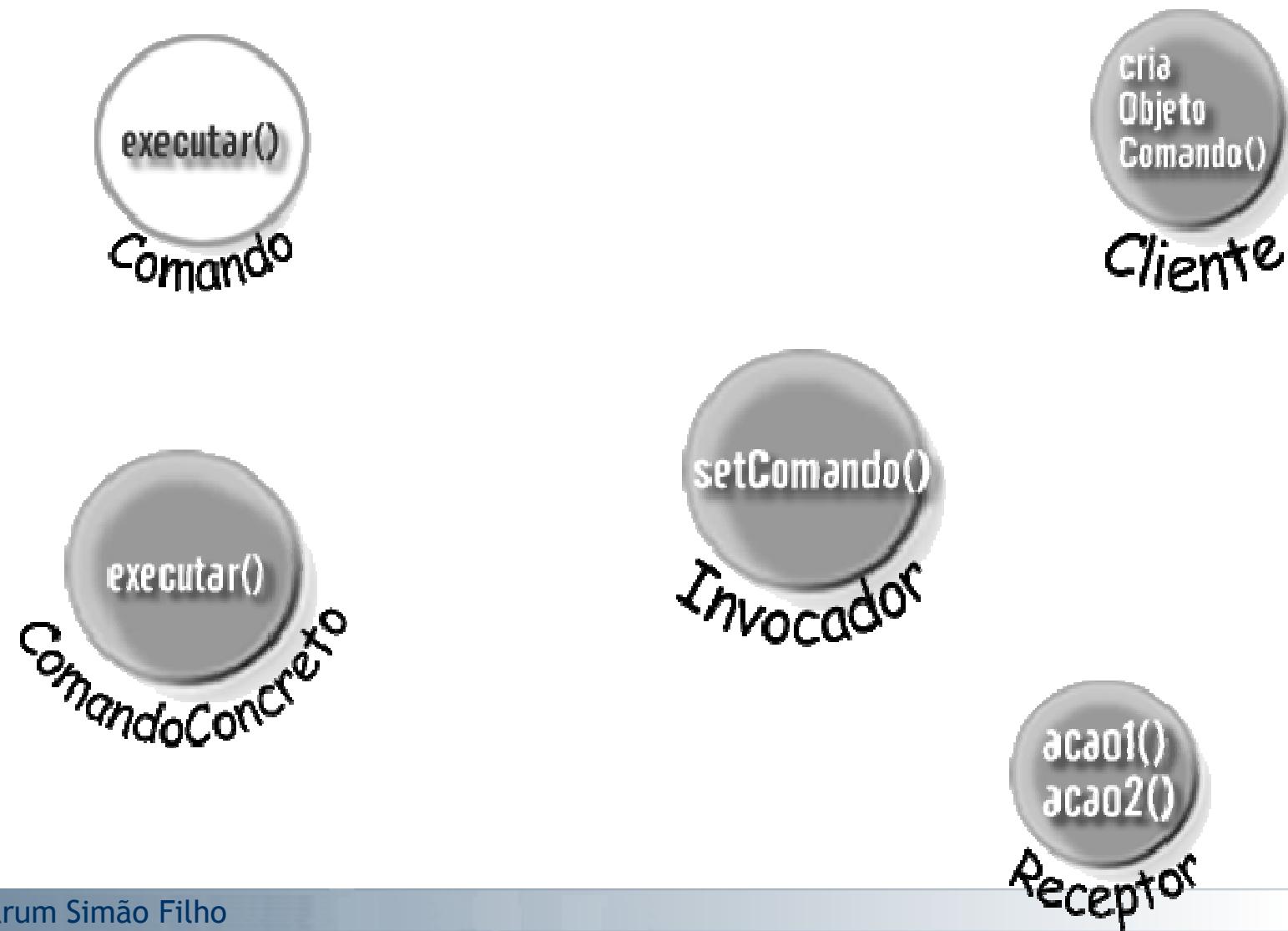
# Estrutura



# Simplificando



# Padrão Comando Participantes

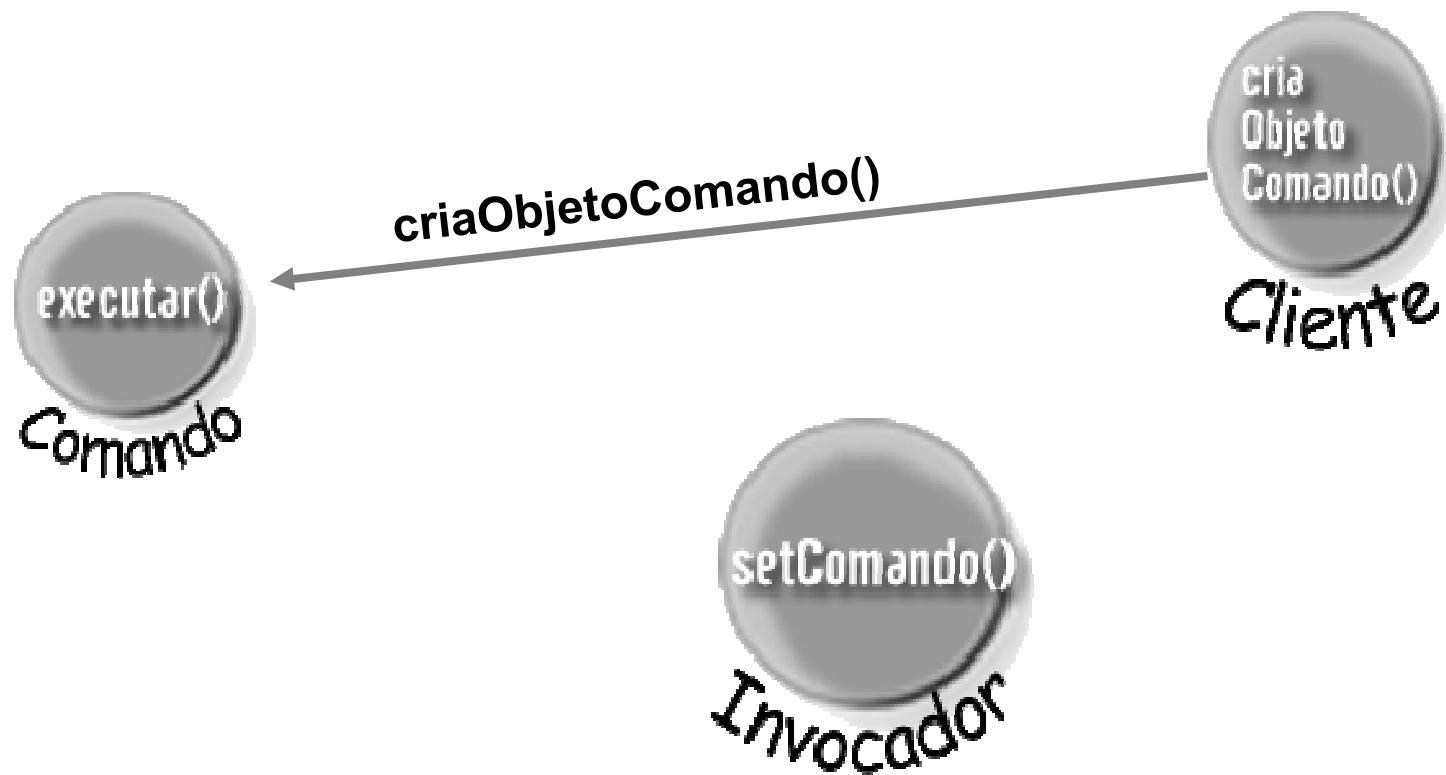


# O Comando

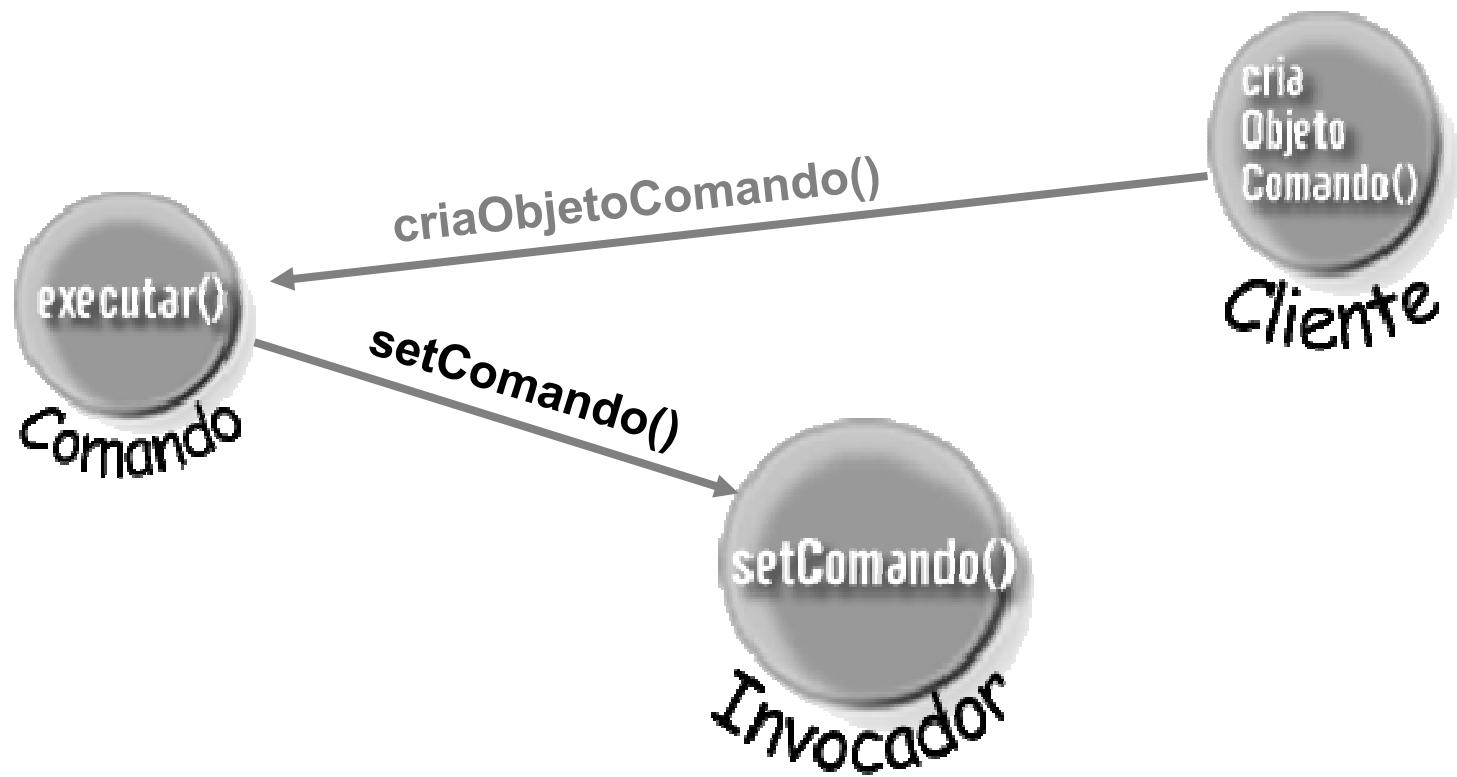
## Um conjunto de Ações no Receptor



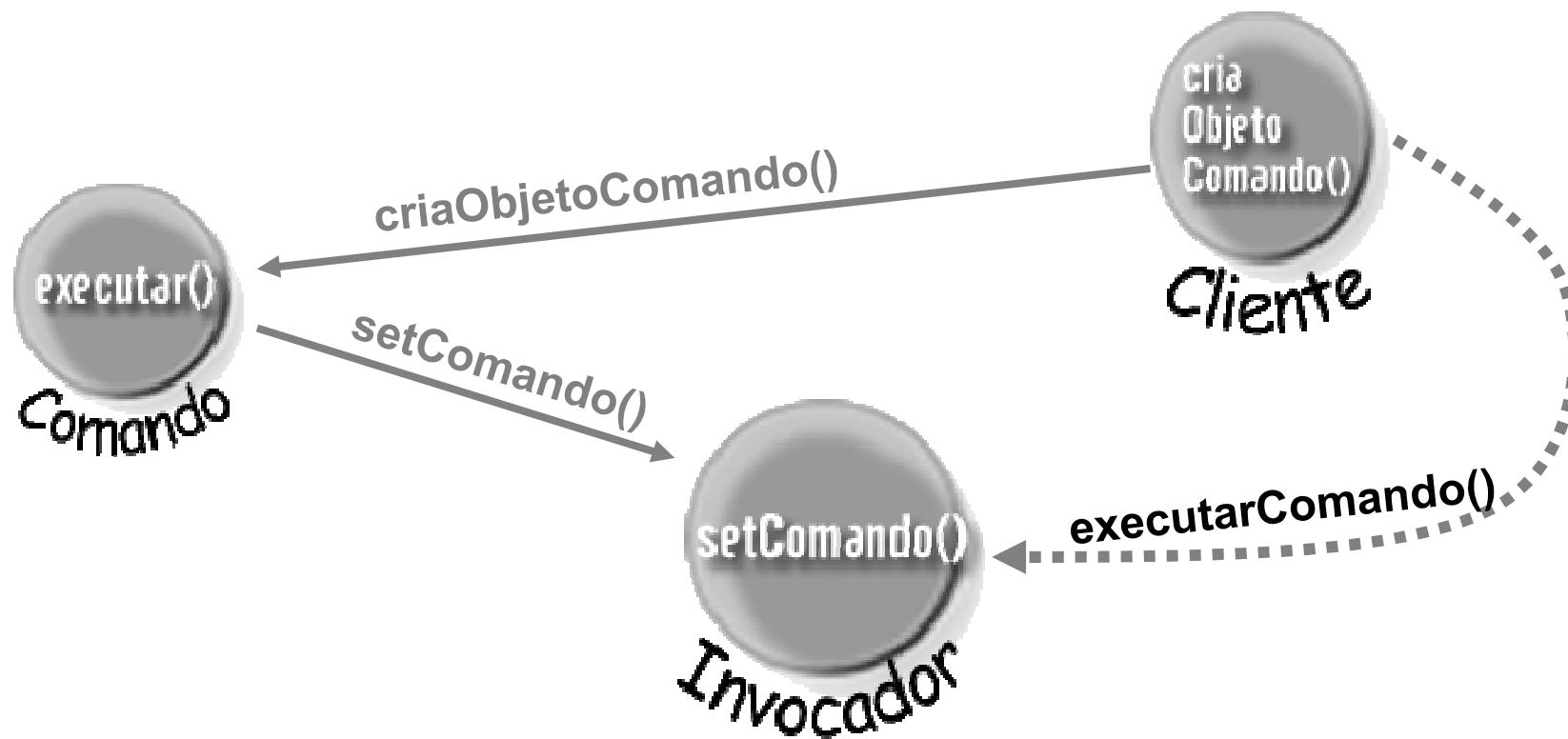
# 1. O cliente cria um objeto do tipo Comando



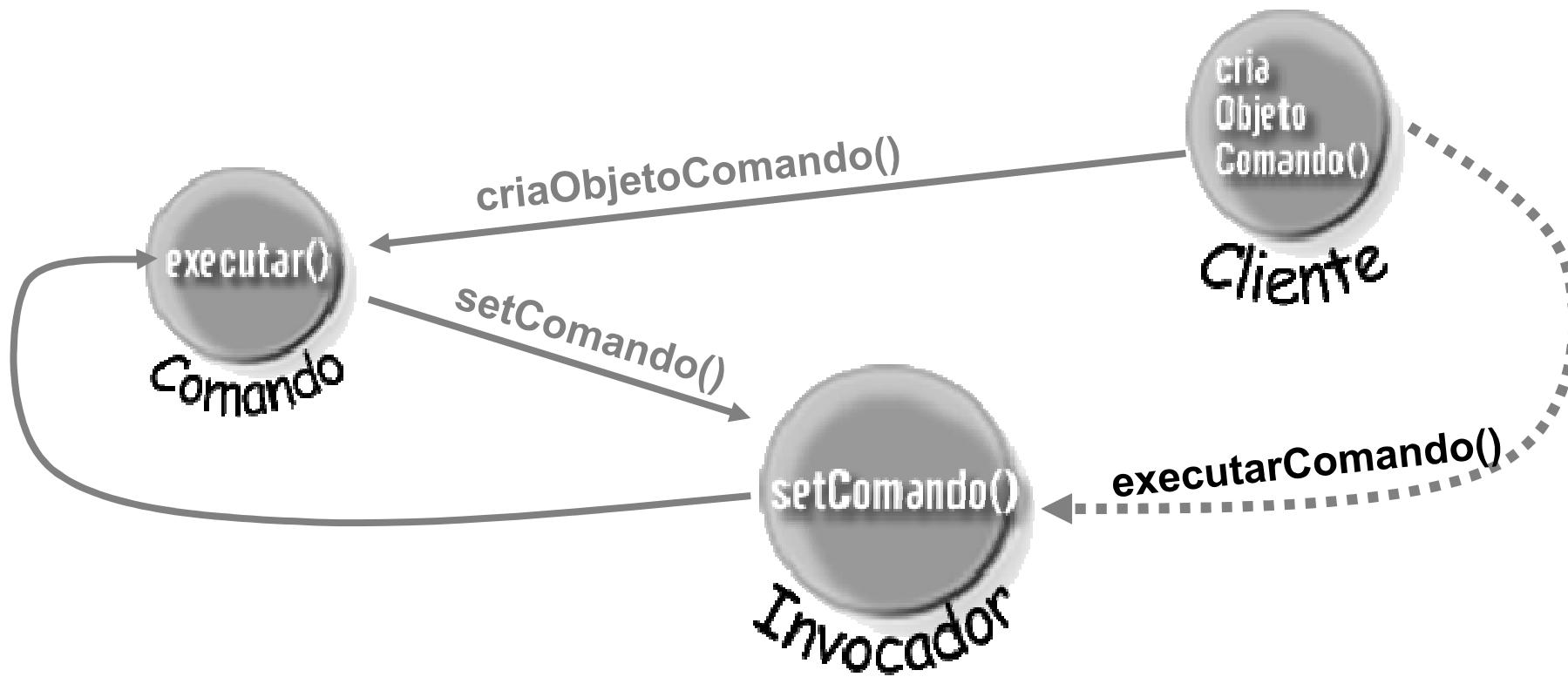
## 2. O cliente executa um `setComando()` para armazenar o objeto de comando no Invocador



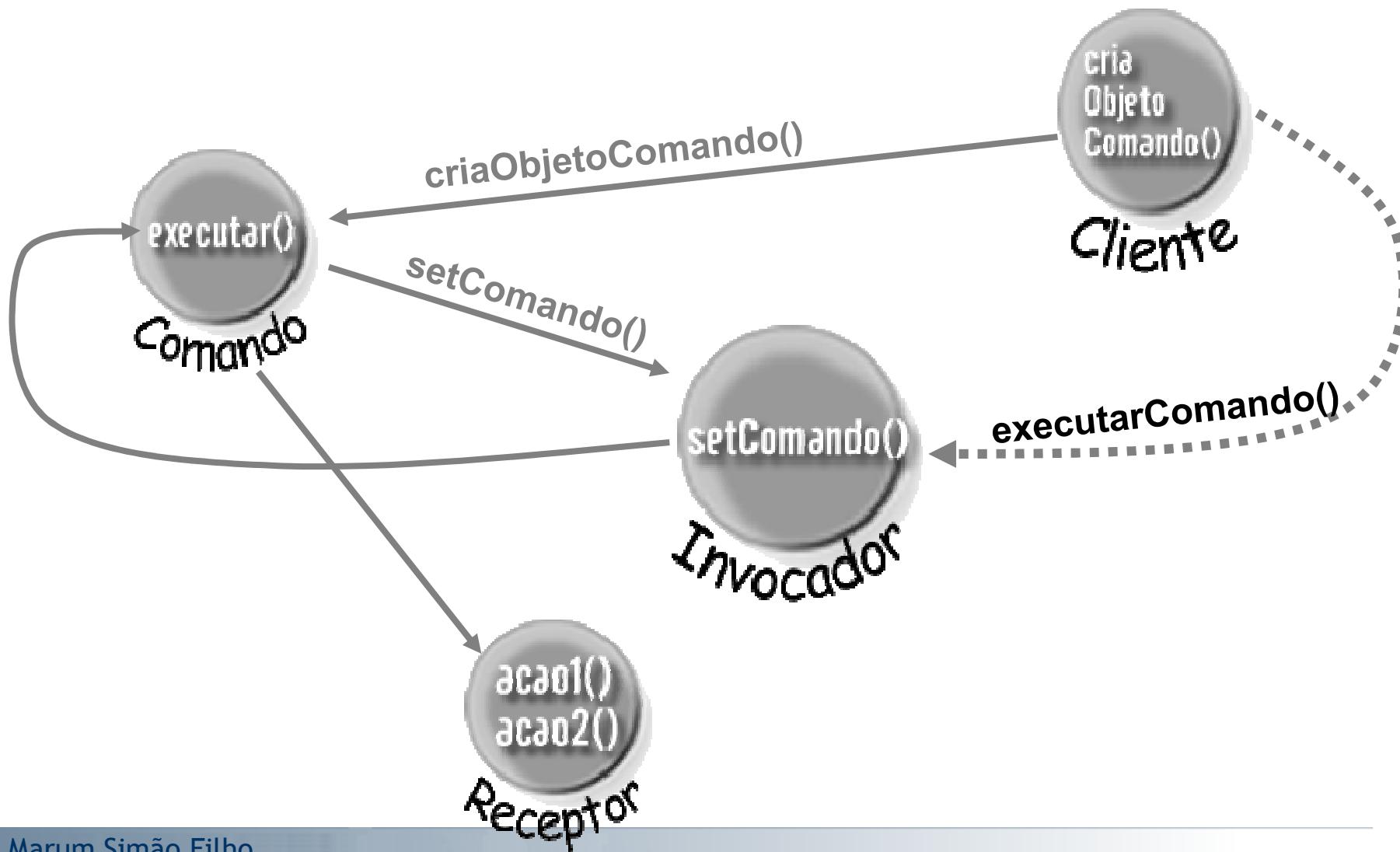
### 3. O cliente pede ao invocador para executar o comando

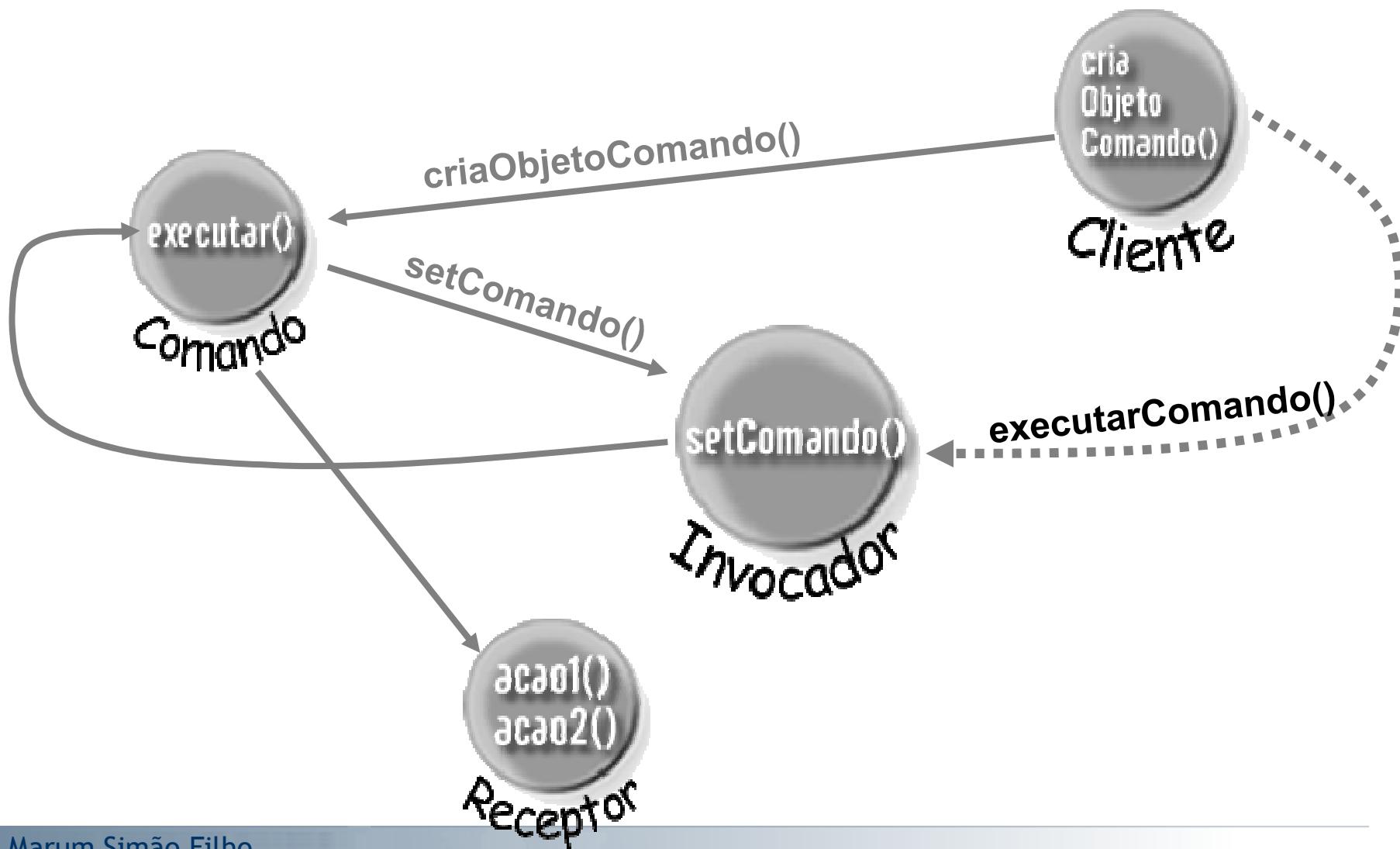


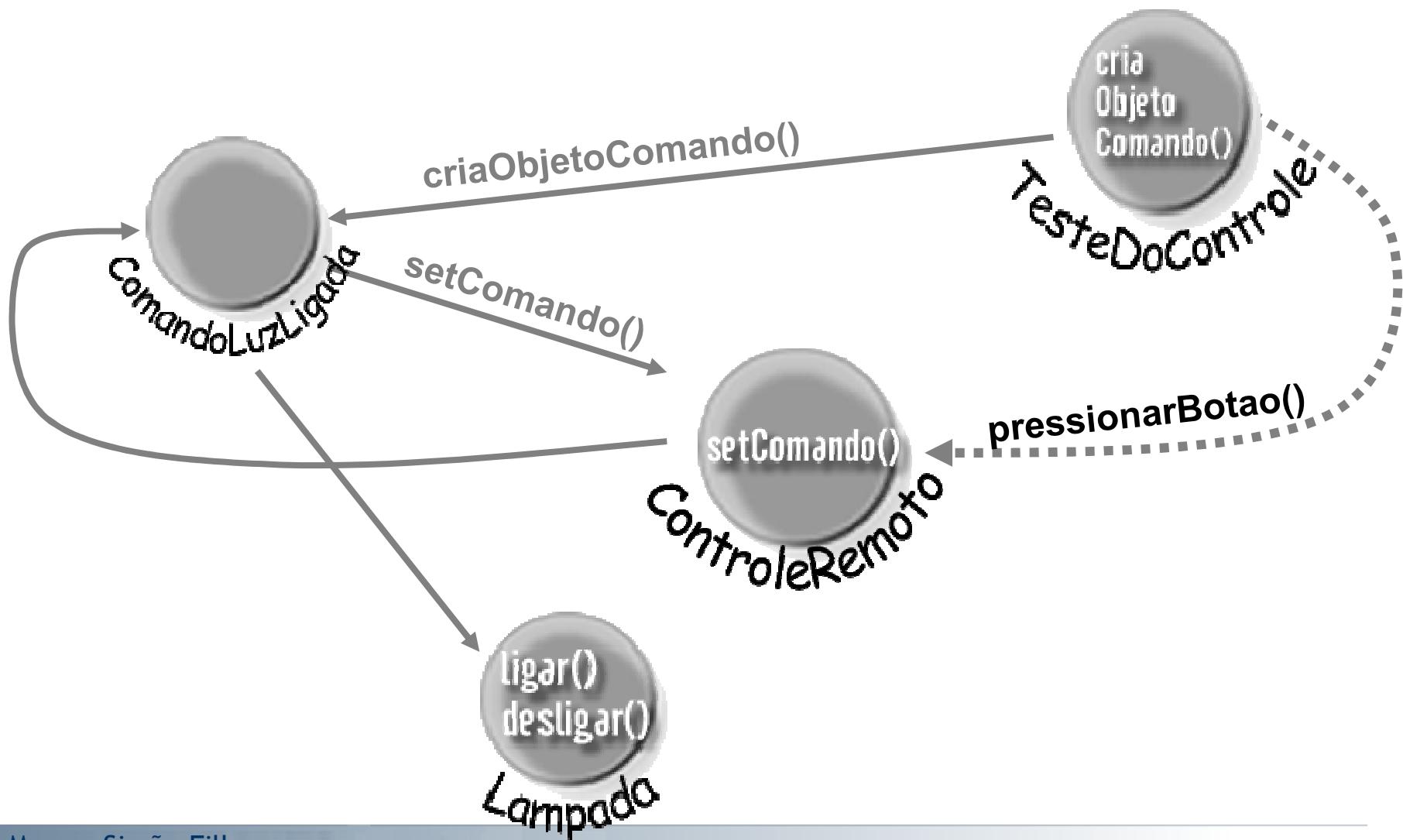
## 4. O Invocador chama o método executar do Comando



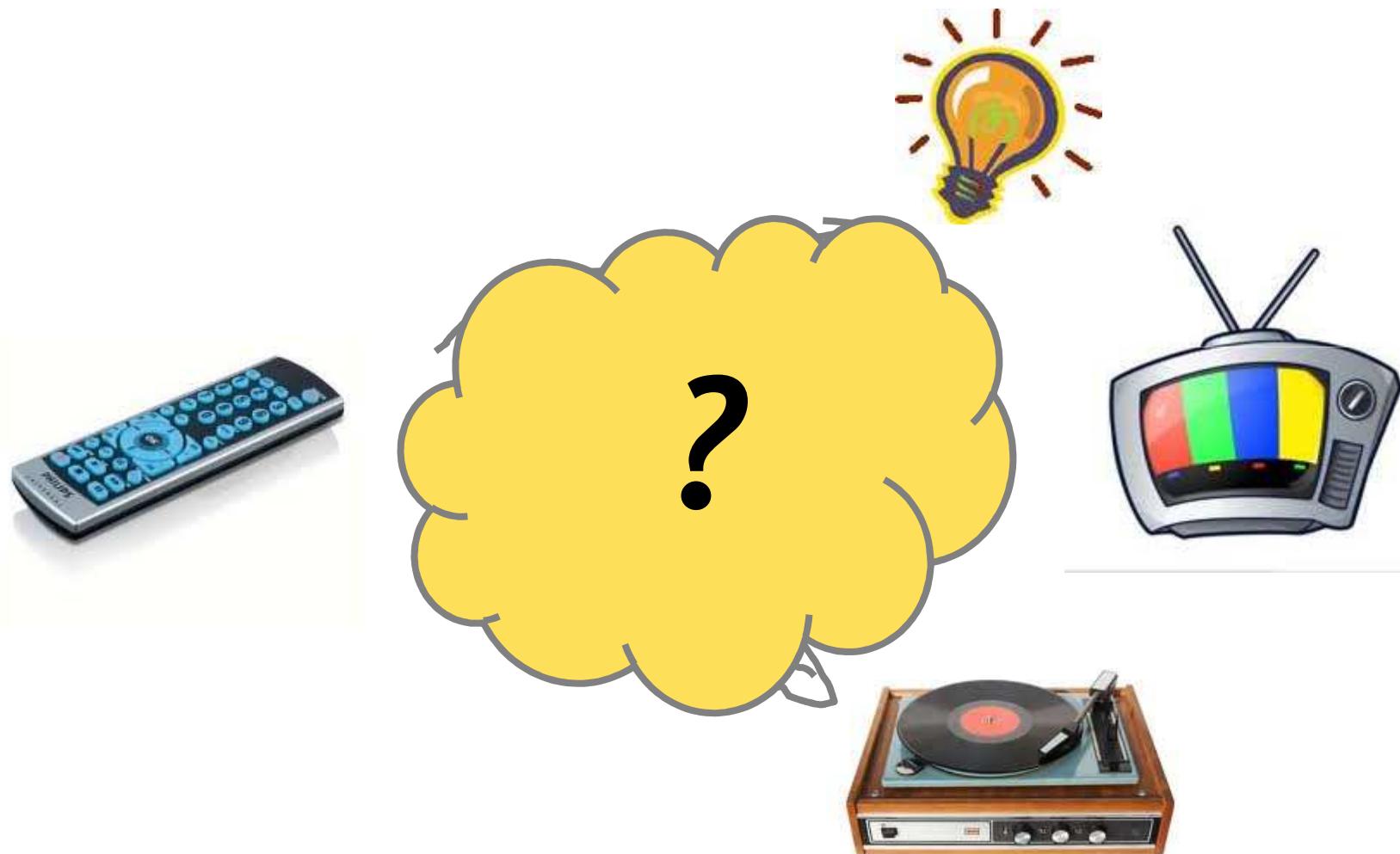
## 4. O Comando executa os métodos do Receptor







# O controle remoto universal

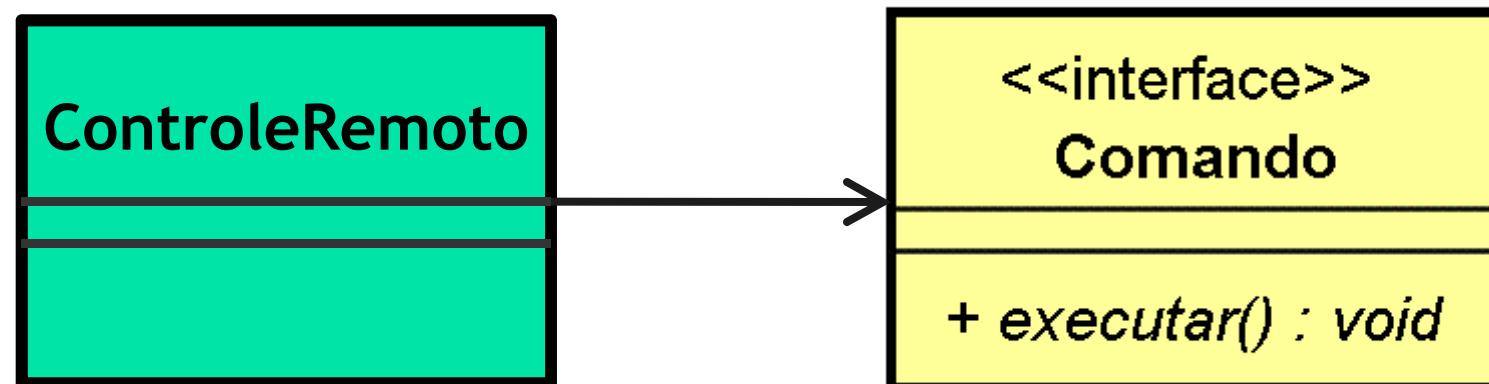


# O controle remoto universal

- Vamos construir um **controle remoto** que funcionará como um **invocador**.
- Ele acionará diversos métodos de várias classes.
- Todas as classes que representam um comando implementarão uma mesma interface.

# Interface Comando

```
public interface Comando {  
    public void executar();  
}
```

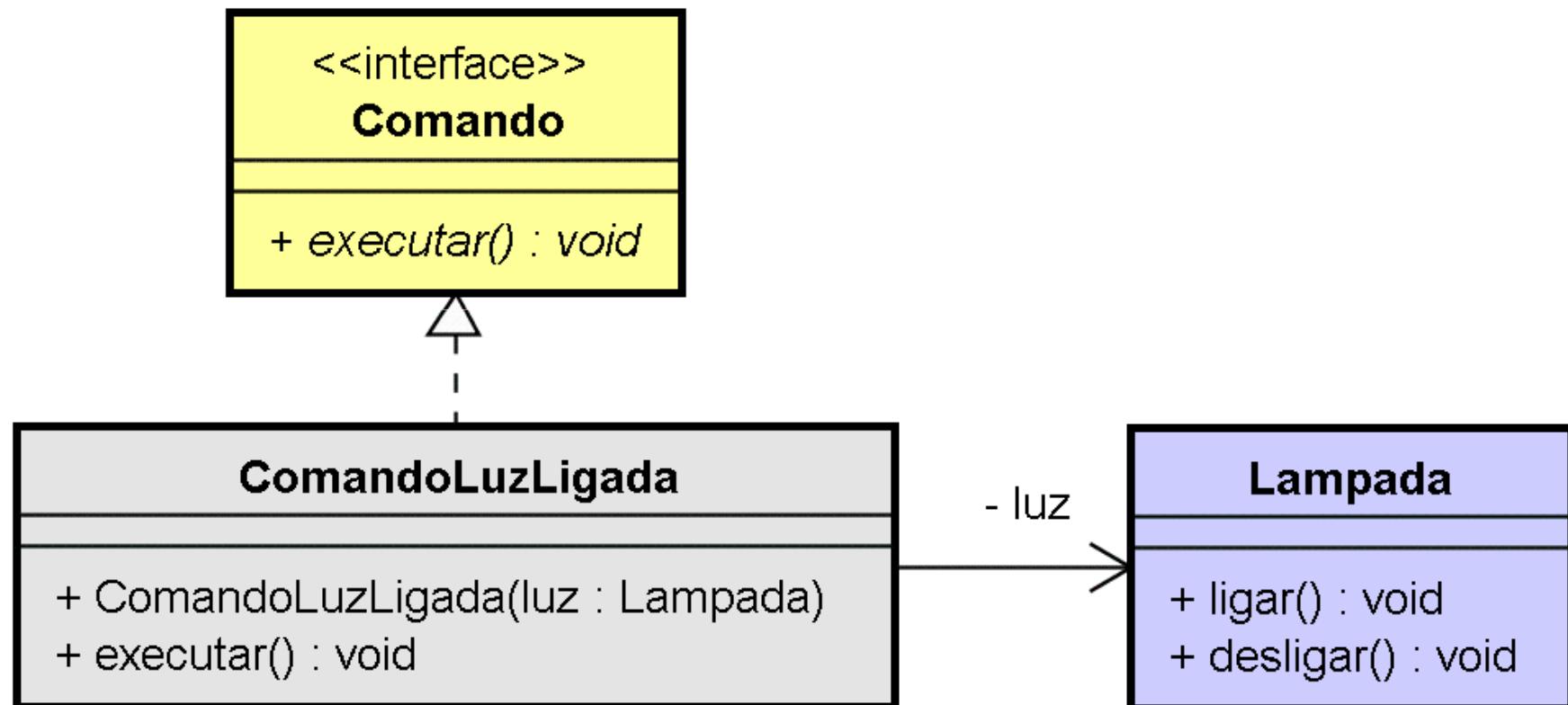


Lampada
+ ligar() : void
+ desligar() : void

Digamos que exista  
um objeto Lâmpada (receptor)

```
public class Lampada {  
    public void ligar() {  
        System.out.println("Luz ligada");  
    }  
  
    public void desligar() {  
        System.out.println("Luz desligada");  
    }  
}
```

# Vamos agora implementar um Comando para ligar a Lâmpada(receptor)



# Vamos agora implementar um Comando para ligar a Lâmpada(receptor)

```
public class ComandoLuzLigada implements Comando
{
    private Lampada luz;

    public ComandoLuzLigada(Lampada luz) {
        this.luz = luz;
    }

    public void executar() {
        luz.ligar();
    }
}
```

# O Controle Remoto (invocador)

```
public class ControleRemotoSimples {  
    private Comando slot;  
  
    public ControleRemotoSimples( ) {  
    }  
  
    public void setComando(Comando comando) {  
        this.slot = comando;  
    }  
  
    public void pressionarBotao( ) {  
        slot.executar();  
    }  
}
```

# Testando o Controle

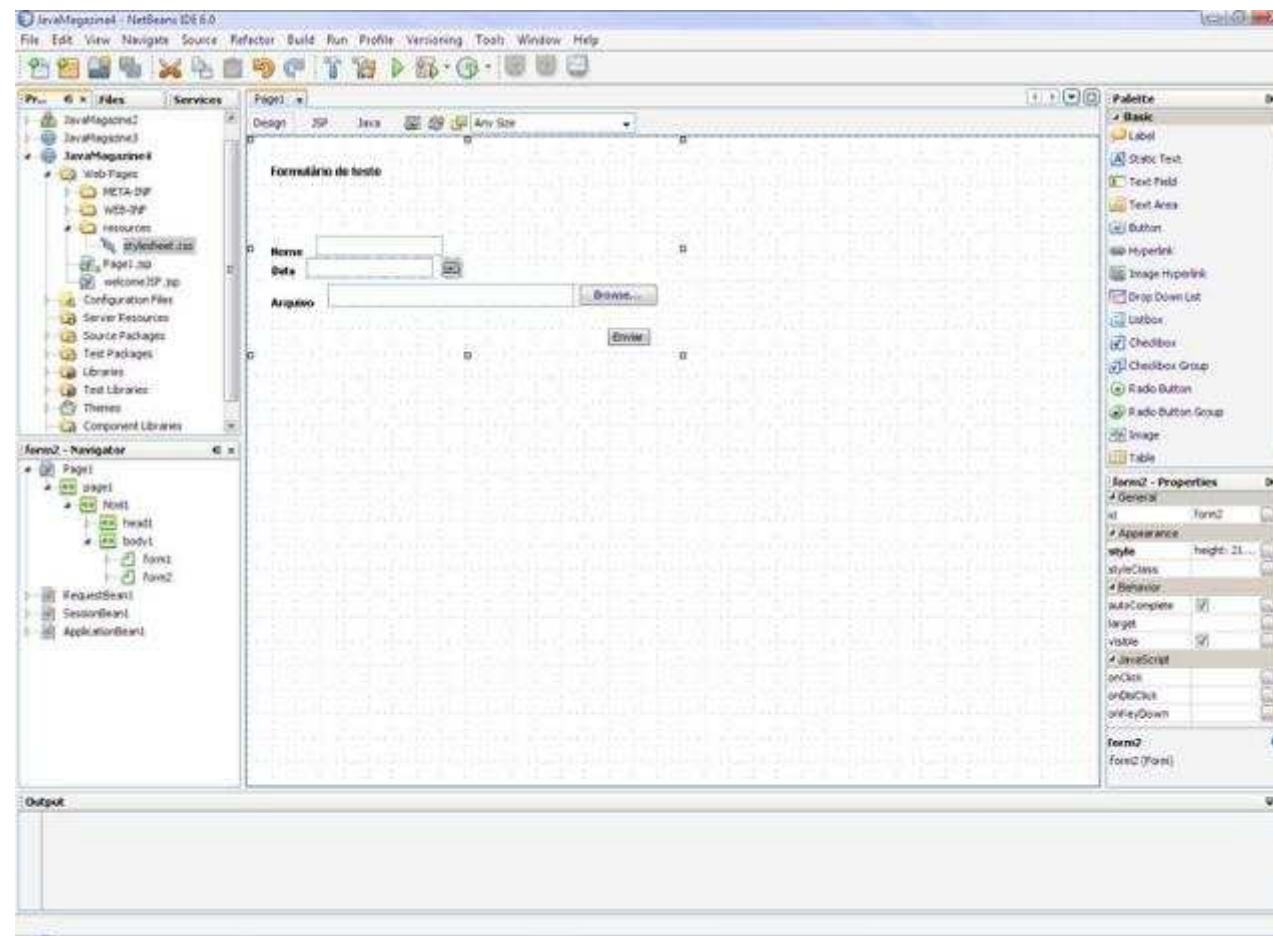
```
public class TesteDoControleRemoto {  
    public static void main(String[] args) {  
  
        // invocador  
        ControleRemotoSimples controle = new ControleRemotoSimples();  
  
        // receptor  
        Lampada lampada = new Lampada();  
  
        // comando  
        ComandoLuzLigada ligarLuz = new ComandoLuzLigada(lampada);  
  
        controle.setComando(ligarLuz);  
  
        controle.pressionarBotao();  
    }  
}
```

# Um exemplo mais concreto...

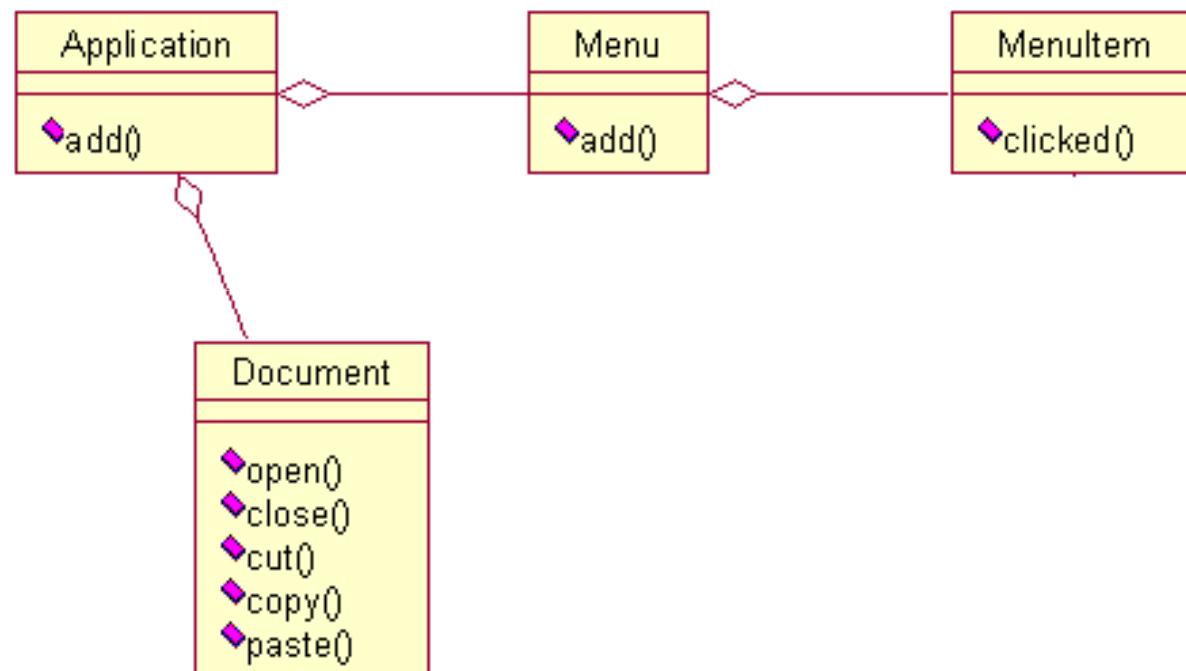
## ■ Motivação:

- *Toolkits* (ferramentas) para construção de interfaces de usuário incluem objetos como botões e 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



# A Toolkit



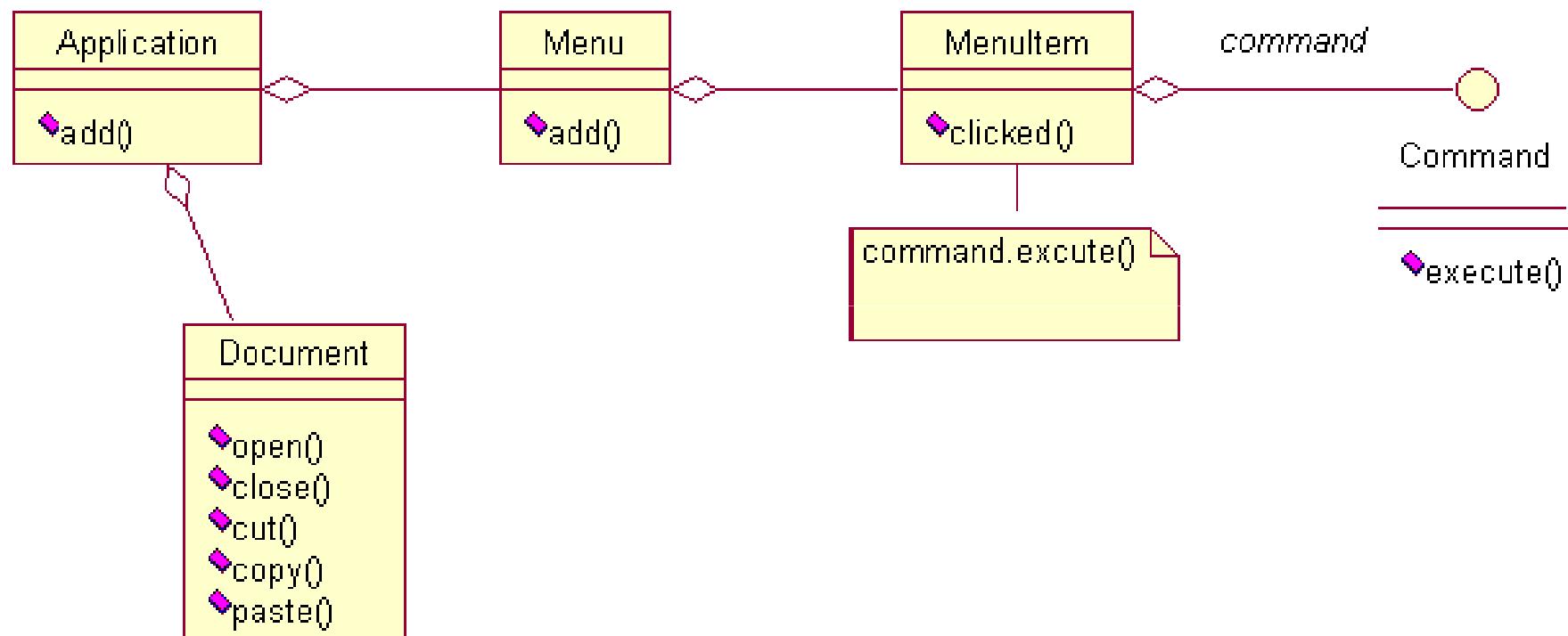
# A Toolkit

- 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 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.

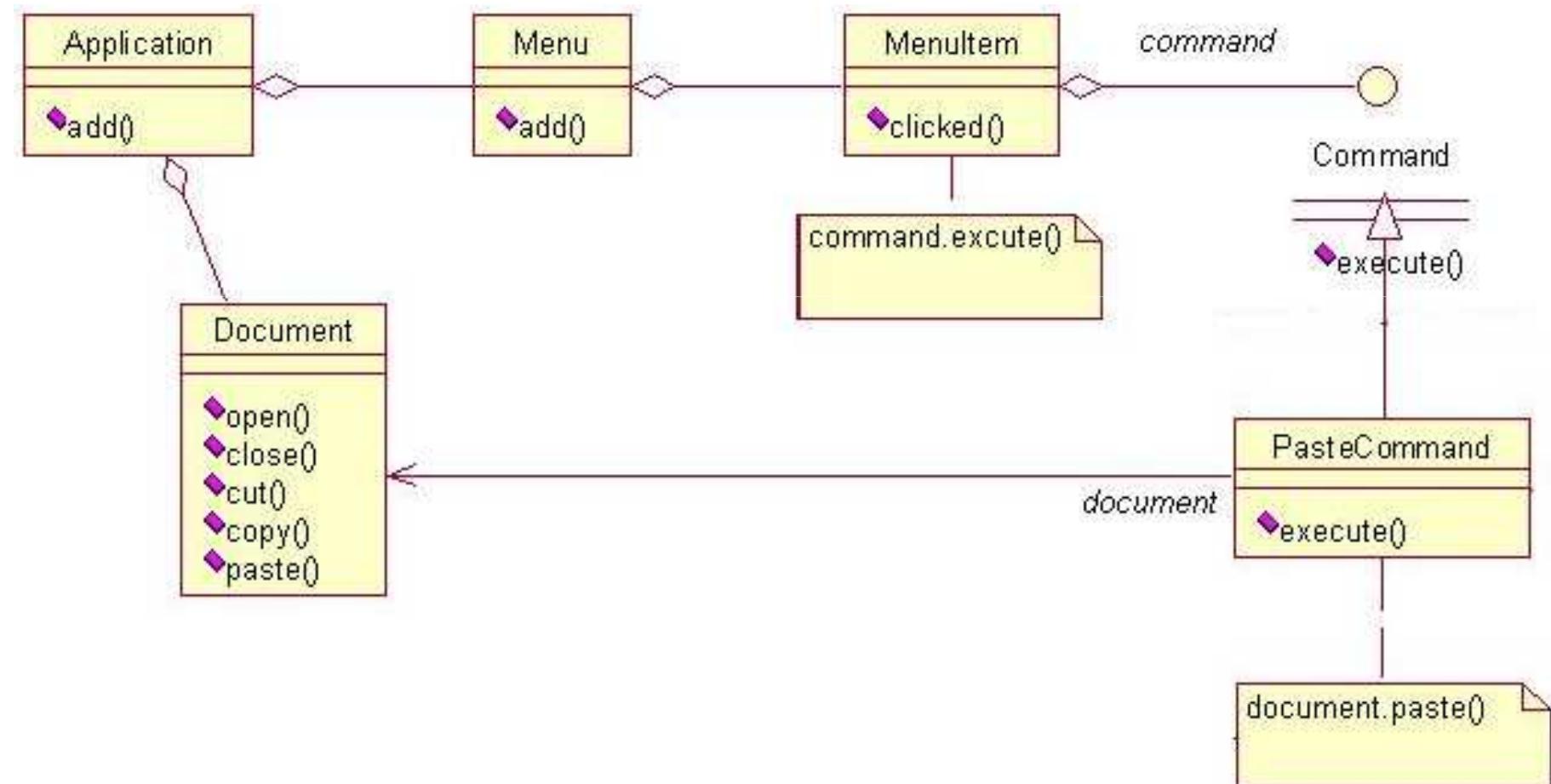
# A Toolkit

- Note que:
  - MenuItem's 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.

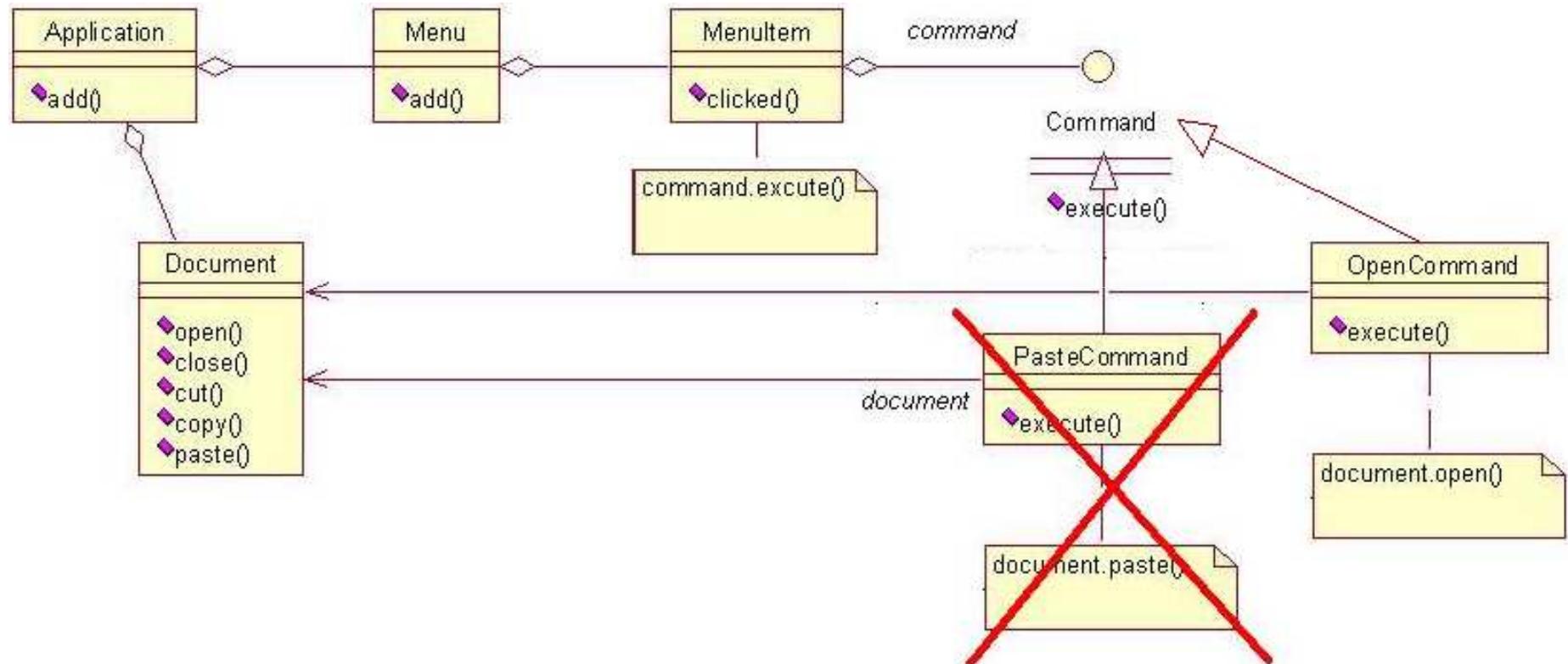
# A solução...



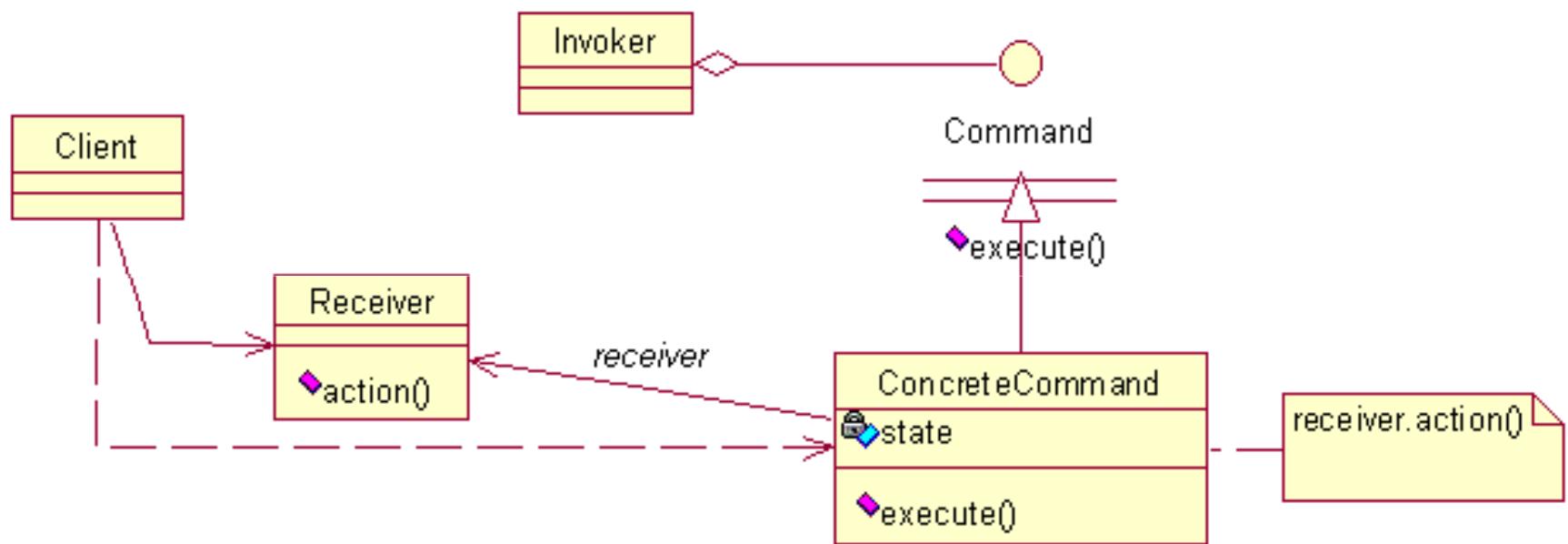
# A solução...



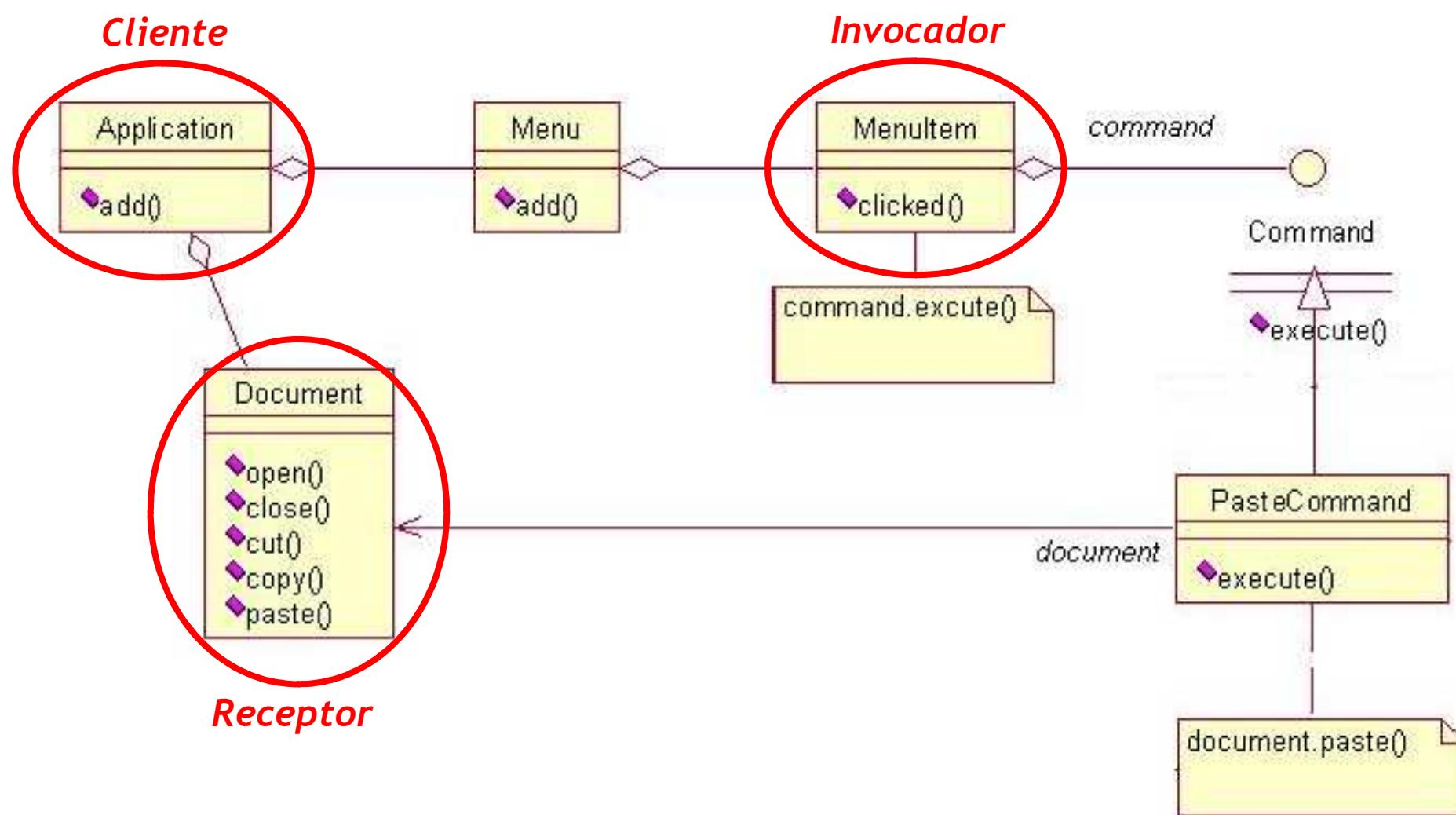
# Trocando um comando...



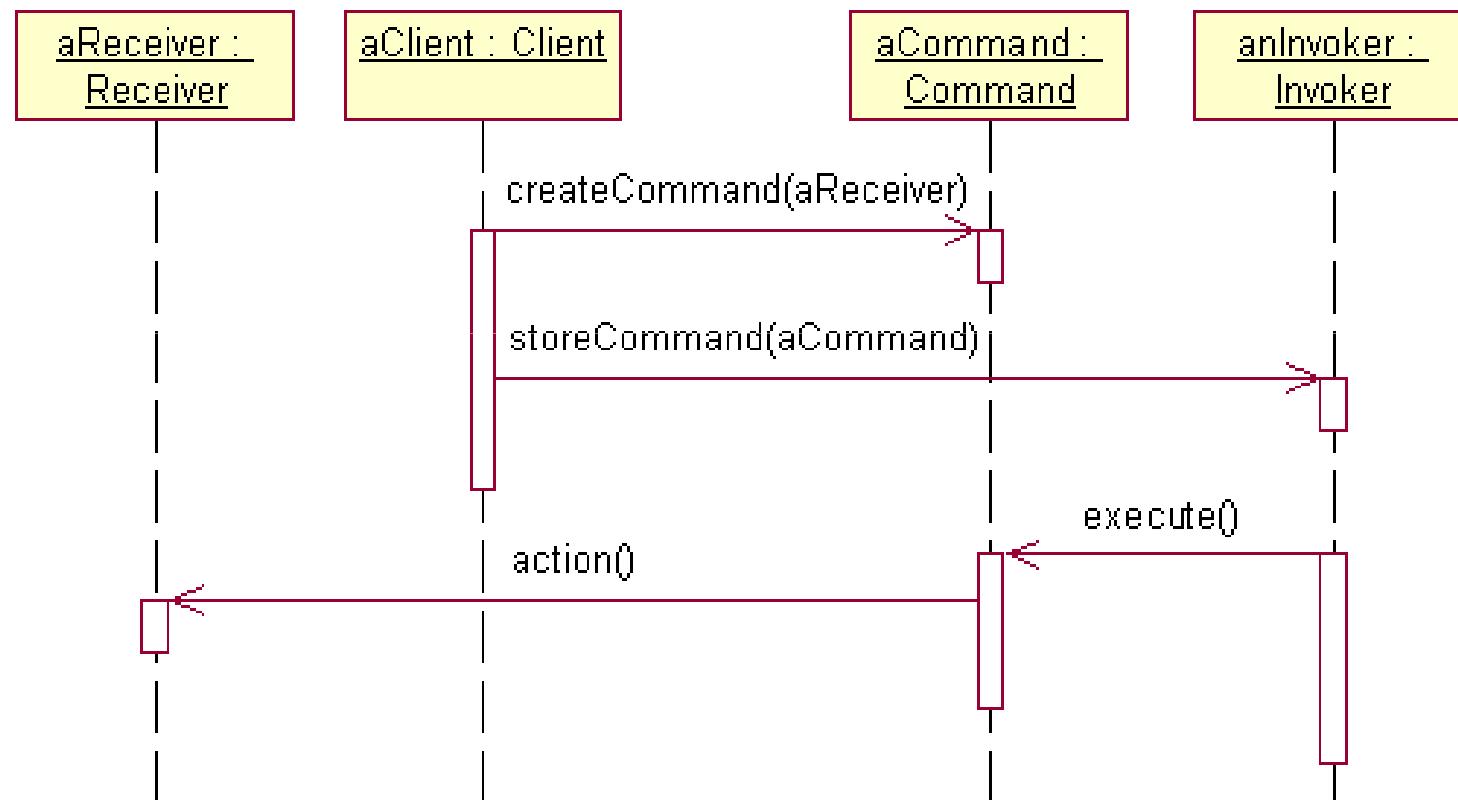
# Estrutura - Diagrama de Classes



# A solução...



# Colaborações - Diagrama de Seqüência



# Um pouco de código

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

//classe PasteCommand
public class PasteCommand implements Command{
    private Document document; // receptor
    public PasteCommand(Document document){
        this.document = document;
    }
    public void execute(){
        document.paste();
    }
}
```

# Um pouco de código

```
//class Document - receptor
public class Document {
    private String name;
    public Document(String name) {this.name = name; }
    public void open(){
        System.out.println("Documento "+ this.name + " foi
aberto."); }
    public void paste(){
        System.out.println("Algo colado no Documento " +
this.name + "."); }
    public void close(){...}
    public void cut(){...}
    public void copy(){...}
}
```

# Um pouco de código

```
//classe Menu
public class Menu {
    private Collection menuItens;
    public Menu() {
        this.menuItens = new ArrayList();
    }
    public void add (MenuItem menuItem) {
        this.menuItens.add(menuItem);
    }
}
```

# Um pouco de código

```
//classe MenuItem - invocador
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();
    }
}
```

# Um pouco de código

```
//classe Application - cliente
public class Application {
    private Collection docs;
    public static void main(String[] args){
        Application application = new Application();
        Menu menu = application.createMenu();
        Command pasteCommand = new PasteCommand(
            new Document("File1.doc"));
        MenuItem item = application.createMenuItem
            ("Paste", pasteCommand);
        menu.add(item);
        item.clicked();
    }
}
```

# +1 Padrão **COMMAND**

O Padrão **Command** encapsula uma **solicitação** como um **objeto**, o que lhe permite **parametrizar** clientes com diferentes solicitações, enfileirar ou registrar solicitações (log) e implementar recursos para **desfazer** operações.

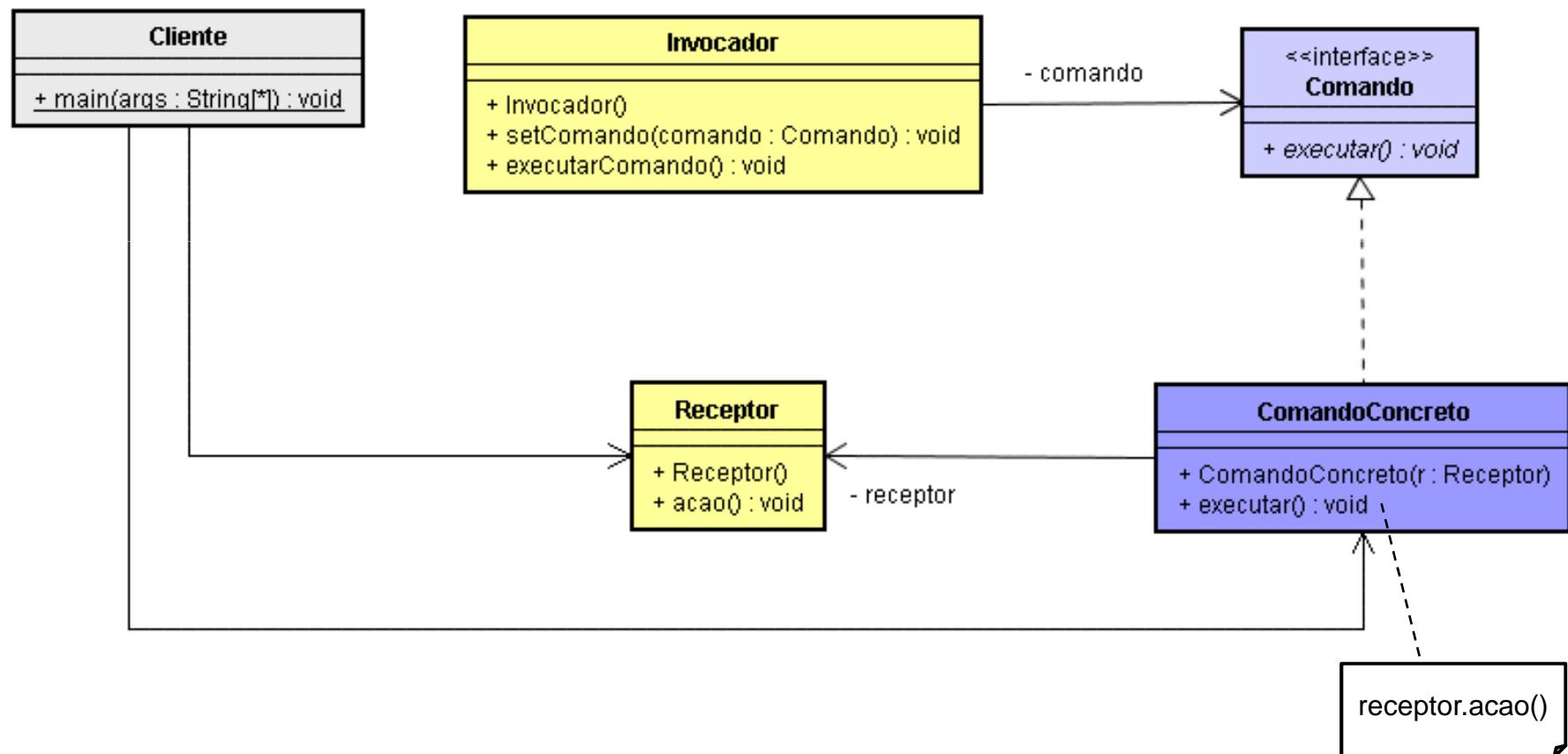
# Aplicabilidade

- Parametrizar objetos por uma ação a ser executada.
- Especificar, enfileirar e executar solicitações em tempos diferentes.
  - Um objeto *Command* pode ter um tempo de vida independente de sua solicitação original.
- Suportar desfazer operações.
  - A operação *execute* de *Command* pode armazenar estados para reverter seus efeitos no próprio comando.

# Aplicabilidade

- Suportar o registro (*logging*) de mudanças de maneira que possam ser reaplicadas no caso de uma queda de sistema.
  - Requer expansão da interface de *Command*, acrescentando métodos como *armazenar* e *carregar*.
- Estruturar um sistema em torno de operações de alto nível construídas sobre operações primitivas.
  - Isso é comum em sistemas que suportam *transações*.

# Diagrama de classes



# Participantes

## ■ Comando

- Declara uma interface para a execução de uma operação.

## ■ ComandoConcreto

- Define uma vinculação entre um objeto *Receptor* e uma ação.
- Implementa *execute* através da invocação da(s) correspondente(s) operação(ões) no *Receptor*.

## ■ Cliente

- Cria um objeto ComandoConcreto e estabelece o seu *Receptor*.

# Participantes

## ■ Invocador

- Requisita ao Comando a execução da solicitação.

## ■ Receptor

- Sabe como executar as operações associadas a uma solicitação. Qualquer classe pode funcionar como um *Receptor*.

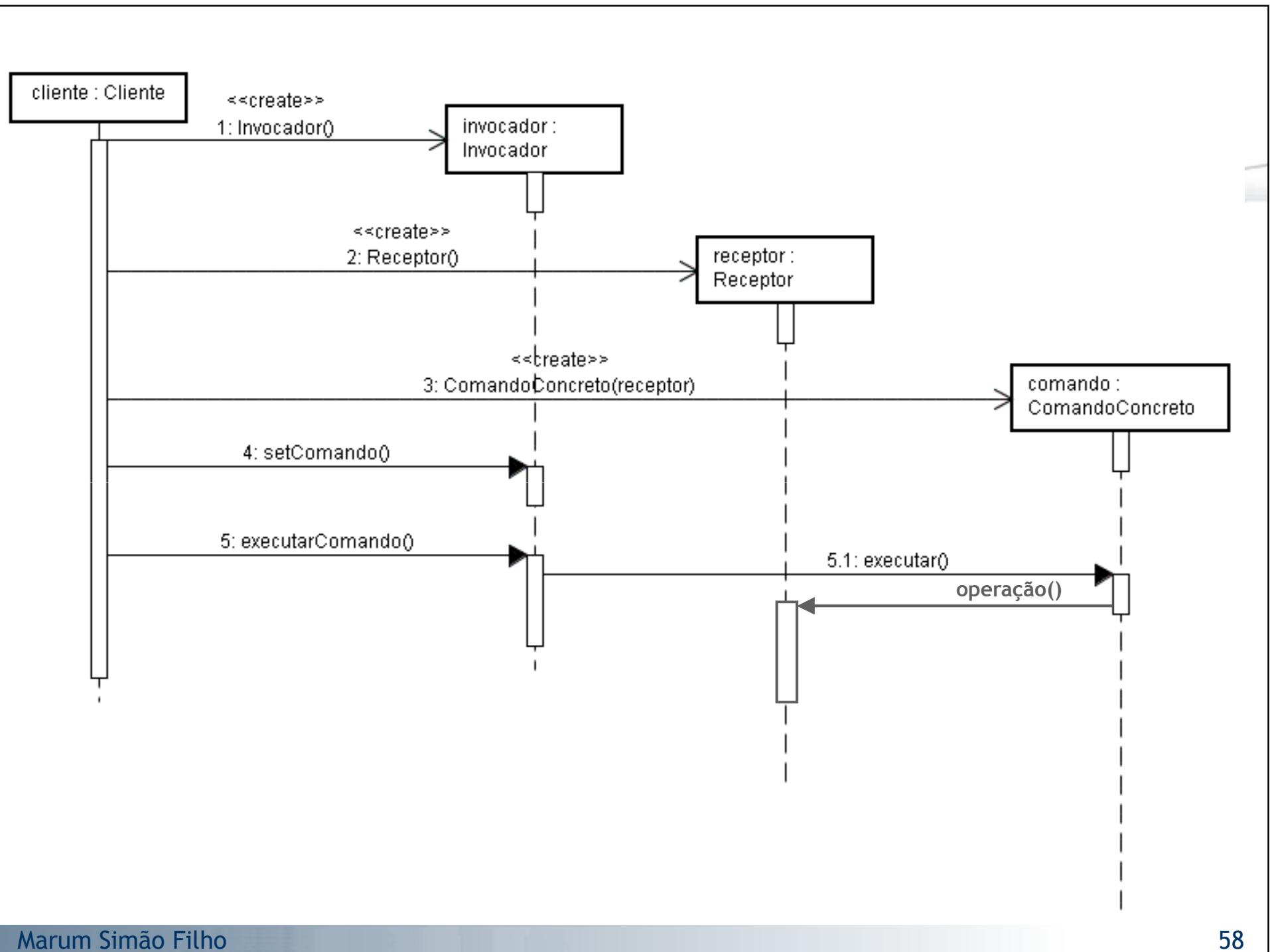
# Descrevendo

## ■ Um objeto Comando...

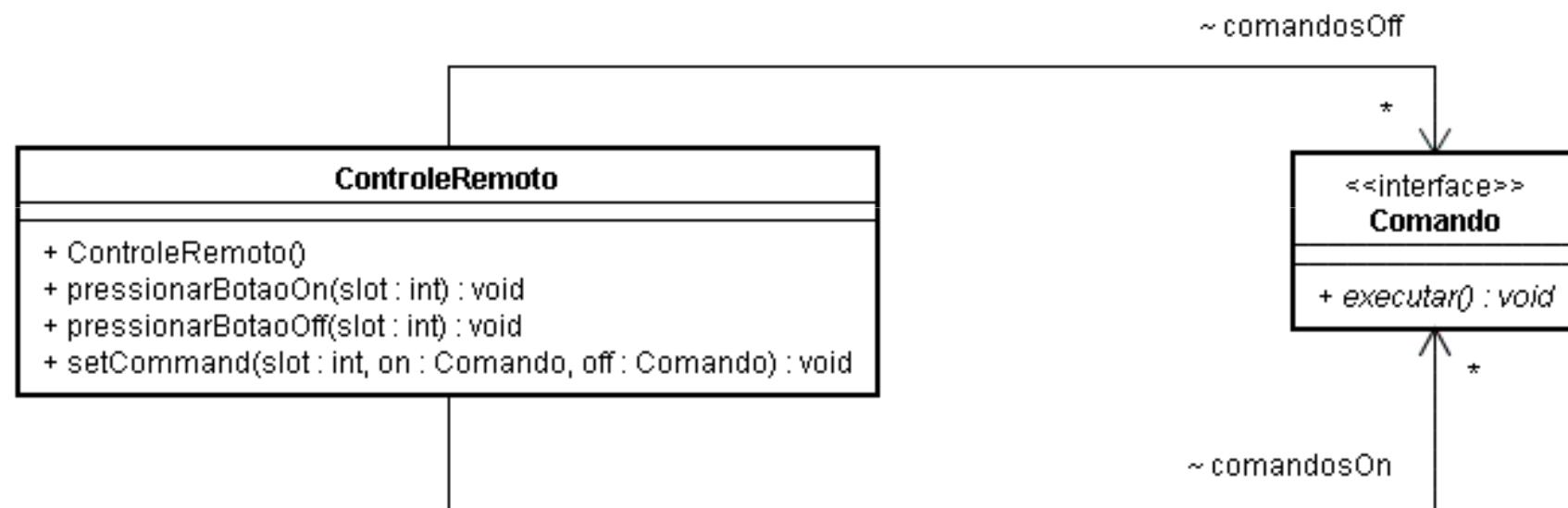
- Encapsula uma solicitação vinculando um conjunto de ações em um receptor específico;
- Empacota as ações e o objeto em um objeto que expõe um único método → executar();
- Ao executar() → Ações são realizadas no receptor;
- Não se sabe quais ações estão sendo realizadas.

# Colaborações

- Um Cliente cria um objeto ComandoConcreto e especifica seu Receptor.
- Um objeto Invocador armazena o objeto ComandoConcreto.
- O Invocador emite uma solicitação chamando *execute* no ComandoConcreto.
- O objeto ComandoConcreto invoca operações no Receptor para executar a solicitação.



# ControleRemoto.java



# ControleRemoto.java

```
public class ControleRemoto {  
    Comando[ ] comandosOn;  
    Comando[ ] comandosOff;  
  
    public ControleRemoto() { }  
  
    public void pressionarBotaoOn(int slot) {  
        comandosOn[slot].executar();  
    }  
  
    public void pressionarBotaoOff(int slot) {  
        comandosOff[slot].executar();  
    }  
  
    public void setCommand(int slot, Comando on, Comando off) {  
        comandosOn[slot] = on;  
        comandosOff[slot] = off;  
    }  
}
```

# Há um problema

- E quando não houver comandos?
- Teremos que escrever?

```
if (comandosOn[slot] != null) {  
    comandosOn[slot].executar();  
}
```

Criar um Comando que não faz nada!

# NoCommand

```
public class ComandoNenhum  
implements Comando {  
  
    public void executar() {  
  
    }  
  
}
```

# O Construtor do ControleRemoto

```
public ControleRemoto() {  
    comandosOn = new Comando[ 7 ];  
    comandosOff = new Comando[ 7 ];  
  
    Comando semComando = new ComandoNenhum();  
    for (int i = 0; i < 7; i++) {  
        comandosOn[i] = semComando;  
        comandosOff[i] = semComando;  
    }  
}
```

# Comando Com undo

```
public class ComandoLuzLigada implements Comando {  
    private Lampada luz;  
  
    public ComandoLuzLigada(Lampada luz) {  
        this.luz = luz;  
    }  
  
    public void executar() {  
        luz.ligar();  
    }  
  
    public void undo() {  
        luz.desligar();  
    }  
}
```

# Controle Remoto Com Undo

```
public class ControleRemotoComUndo {  
    Comando[ ] comandosOn;  
    Comando[ ] comandosOff;  
    Comando comandoUndo;  
  
    public ControleRemotoComUndo( ) {  
        comandosOn = new Comando[ 7 ];  
        comandosOff = new Comando[ 7 ];  
  
        Comando semComando = new ComandoNenhum( );  
        for ( int i = 0; i < 7; i++ ) {  
            comandosOn[ i ] = semComando;  
            comandosOff[ i ] = semComando;  
        }  
        comandoUndo = semComando;  
    }  
    // ...
```

# Controle Remoto Com Undo

```
public void pressionarBotaoOn(int slot) {  
    comandosOn[slot].executar();  
    comandoUndo = comandosOn[slot];  
}  
public void pressionarBotaoOff(int slot) {  
    comandosOff[slot].executar();  
    comandoUndo = comandosOff[slot];  
}  
public void setCommand(int slot, Comando on, Comando off) {  
    comandosOn[slot] = on;  
    comandosOff[slot] = off;  
}  
public void botaoUndoPressionado() {  
    comandoUndo.undo();  
}  
}
```

# Consequências

- Comando desacopla o objeto que invoca a operação daquele que sabe como executá-la.
- Comandos são objetos de primeira classe, ou seja, podem ser manipulados e estendidos como qualquer outro objeto.
- Pode-se montar Comandos para formar um Comando composto.
- É fácil acrescentar novos Comandos porque não é necessário mudar classes existentes.

# Padrão Decorator



# O padrão Decorator

- Suporta todas as características da POO
  - Herança
  - Tipos de objetos customizados
  - Coleções
- Provê uma abstração para o SQL
  - HQL – Hibernate Query Language
  - Suporte para coleções, índices de objetos, consultas nomeadas

Eduardo Mendes de Oliveira - edumendes@gmail.com



# O padrão Decorator



# O padrão Decorator

- Suporta todas as características da POO
  - Herança
  - Tipos de objetos customizados
  - Coleções
- Provê uma abstração para o SQL
  - HQL – Hibernate Query Language
  - Suporte para coleções, índices de objetos, consultas nomeadas



# O padrão Decorator

- Suporta todas as características da POO
  - Herança
  - Tipos de objetos customizados
  - Coleções
- Provê uma abstração para o SQL
  - HQL – Hibernate Query Language
  - Suporte para coleções, índices de objetos, consultas nomeadas

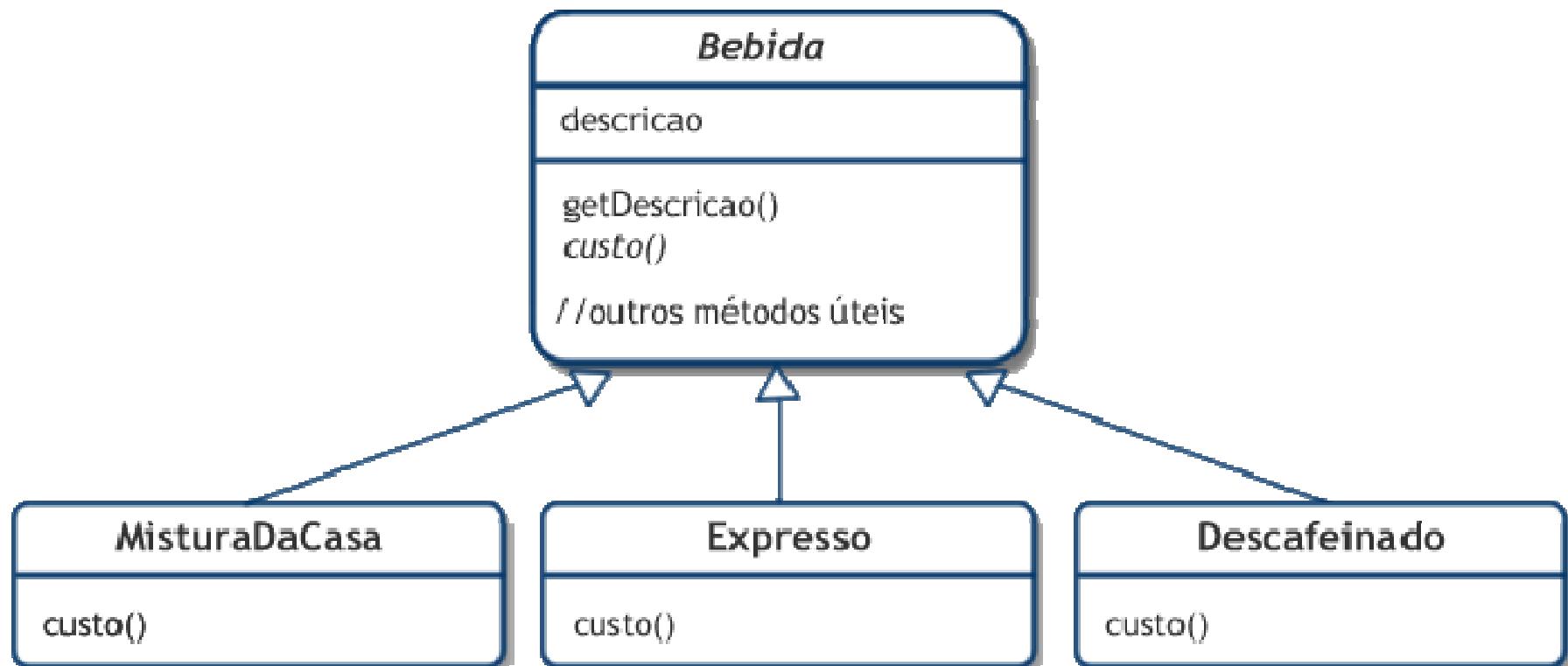
Eduardo Mendes de Oliveira - edumendes@gmail.com

Hibernate

8,50 x 11,00 in



# Estudo de caso JavaCoffe



# Podem ser adicionados Ingredientes Extras

- Chocolate



- Soja



- Leite



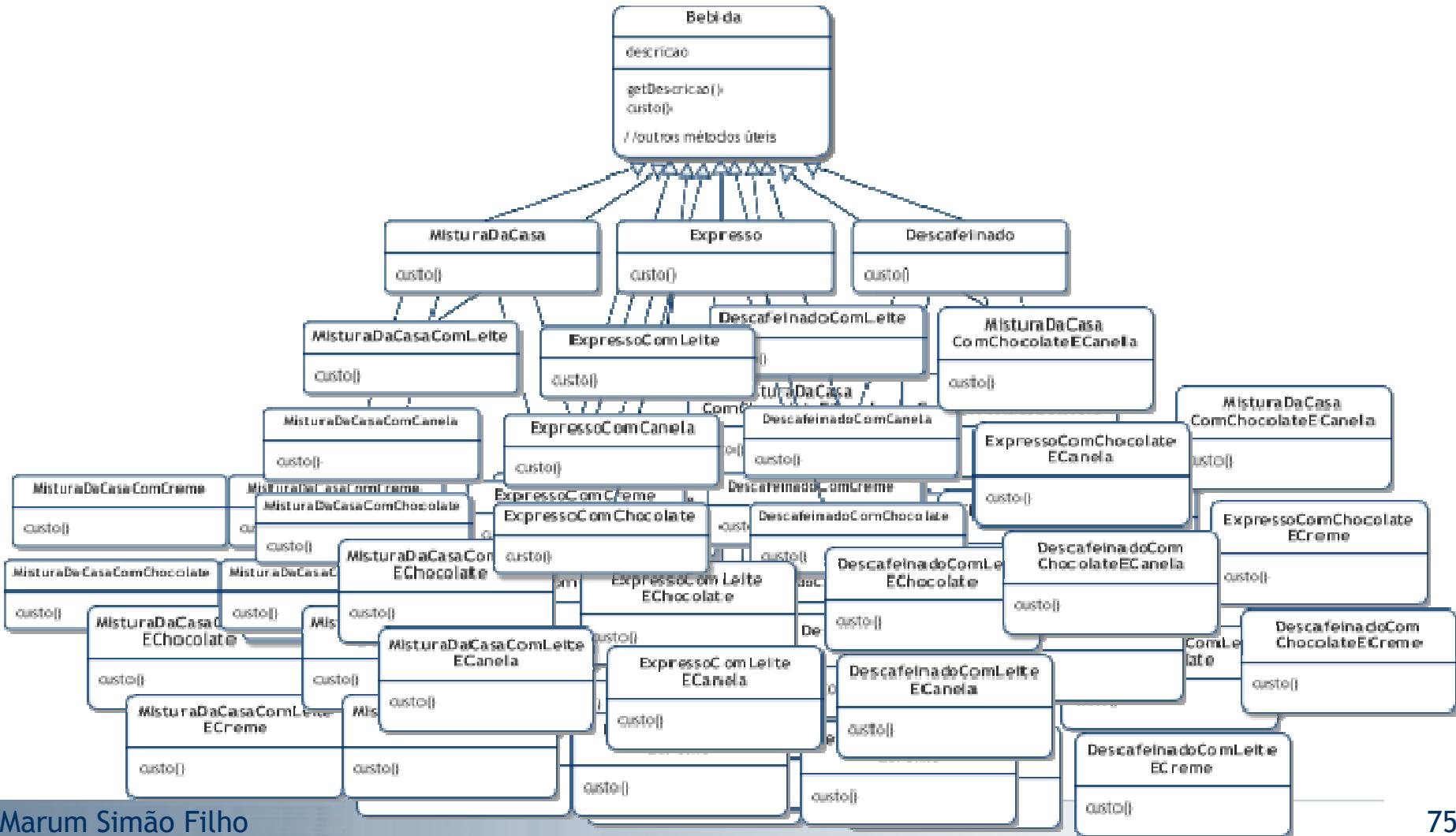
- Creme



- Cada classe precisaria alterar o método custo para adicionar o valor dos ingredientes

# Ops...

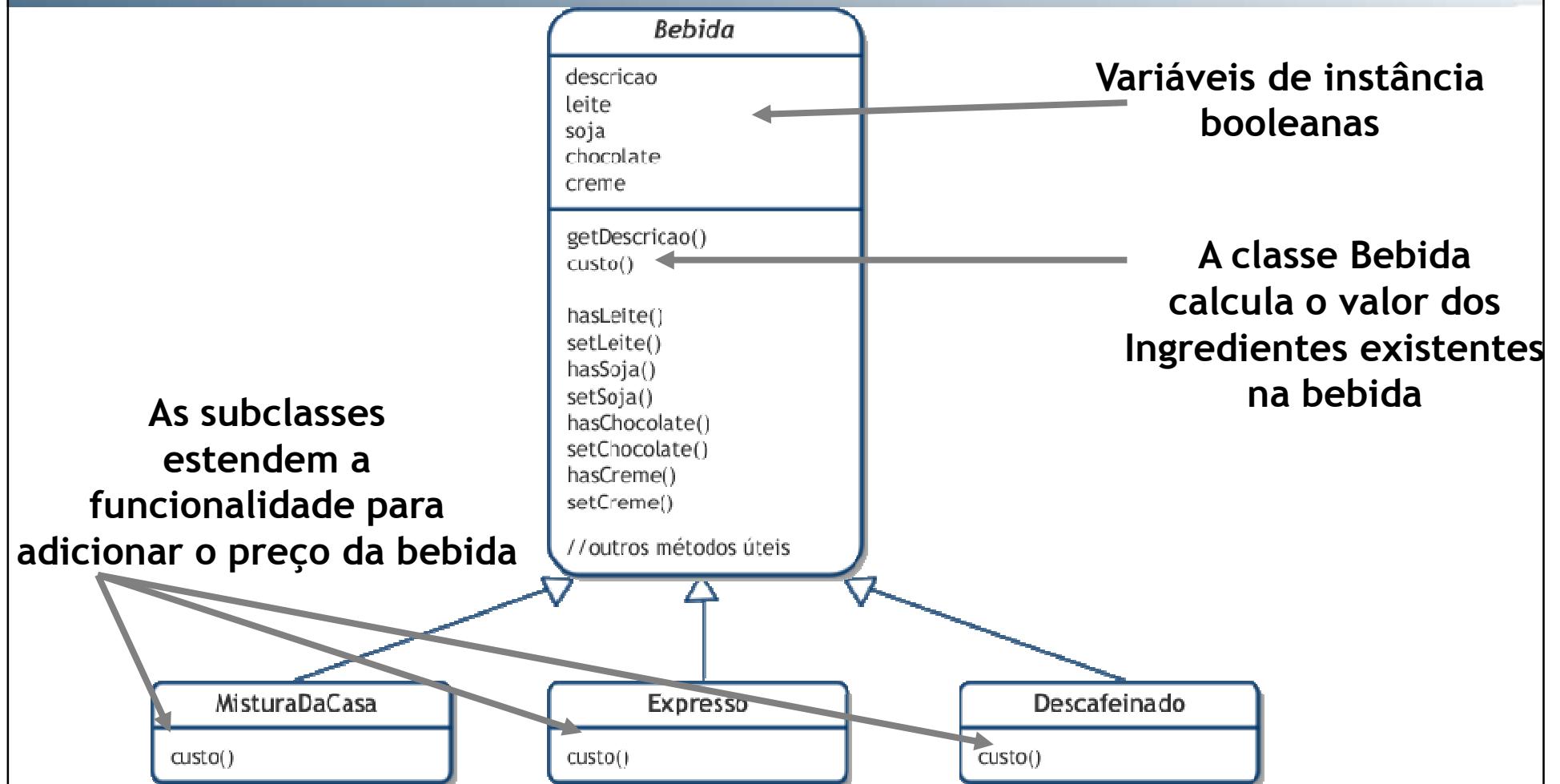
## Número muito grande de subclasses



# Alternativa

- Adicionar variáveis de instâncias booleanas na superclasse para monitorar os ingredientes
  - Classe: Bebida
    - Variáveis de Instância

# Bebida + Variáveis de instância



## Cafés

MisturaDaCasa	0,89
Expresso	1,99
Descafeinado	1,49

# Exercício

- Codifique os métodos custo() das seguinte classes:

```
public class Bebida {  
    public double custo() {  
        //...  
    }  
}
```

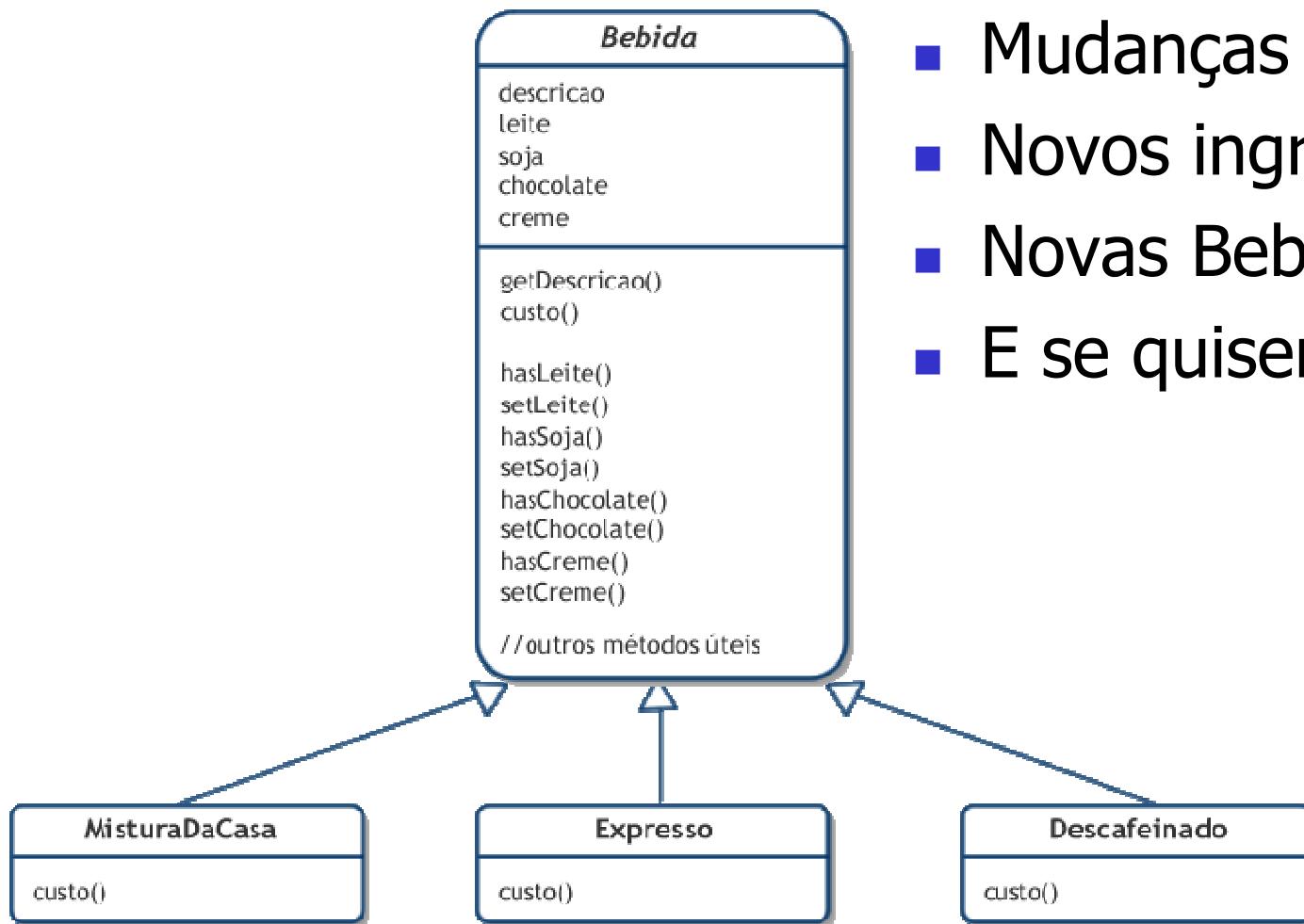
## Ingredientes

Leite	0,10
Soja	0,15
Chocolate	0,20
Creme	0,15



```
public class Expresso  
    extends Bebida {  
  
    public Expresso() {  
        descricao = "Expresso";  
    }  
    public double custo() {  
        //...  
    }  
}
```

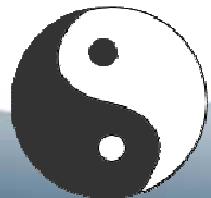
# Quais os problemas?



- Mudanças de preços
- Novos ingredientes
- Novas Bebidas
- E se quiser 2x chocolate?

# Quais os problemas?

- Herança nem sempre leva a designs flexíveis e fáceis de manter.
  - Quais outros meios de “herdar” comportamento?
    - Composição e Delegação
  - Compondo objetos de forma dinâmica é possível adicionar **novas** funcionalidades através da criação de um código novo, **ao invés de alterar** o já existente.



# Princípio de Design OPEN-CLOSED

- As classes devem estar **abertas** para extensão, mas **fechadas** para modificação.



- Adicione novos comportamentos através da extensão

- Não altere o código existente

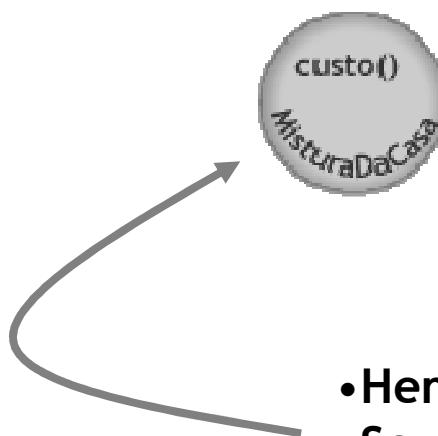


- Permite que a classe seja facilmente estendida sem alterar seu comportamento.

# Qual a idéia?

- O design inicial de bebidas não funciona.
  - Começaremos com uma bebida.
    - E iremos “decorá-la” com ingredientes em tempo de execução.
  - Passos:
    - Pegar um objeto MisturaDaCasa
    - Decorá-lo com um objeto Chocolate
    - Decorá-lo com um objeto Leite
    - Chamar o método *custo* e através da delegação calcular o valor total da bebida.

# 1. Pegar o objeto MisturaDaCasa

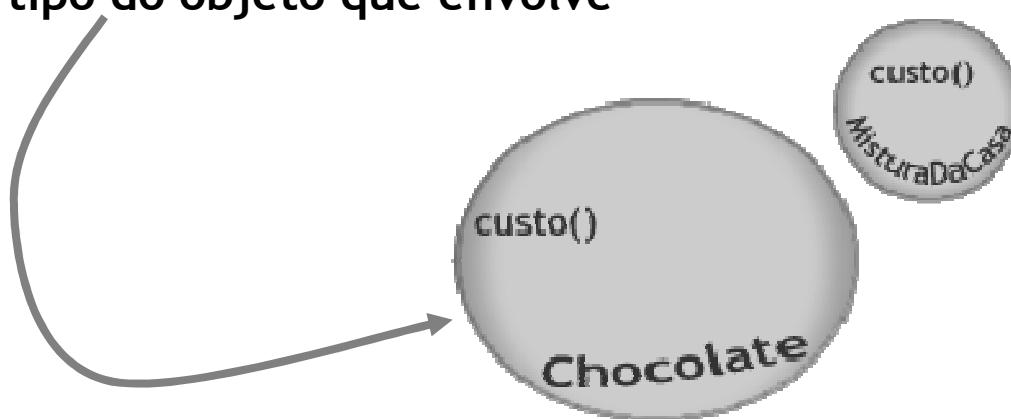


- Herda de Bebida
- Seu método custo() calcula o valor do drinque

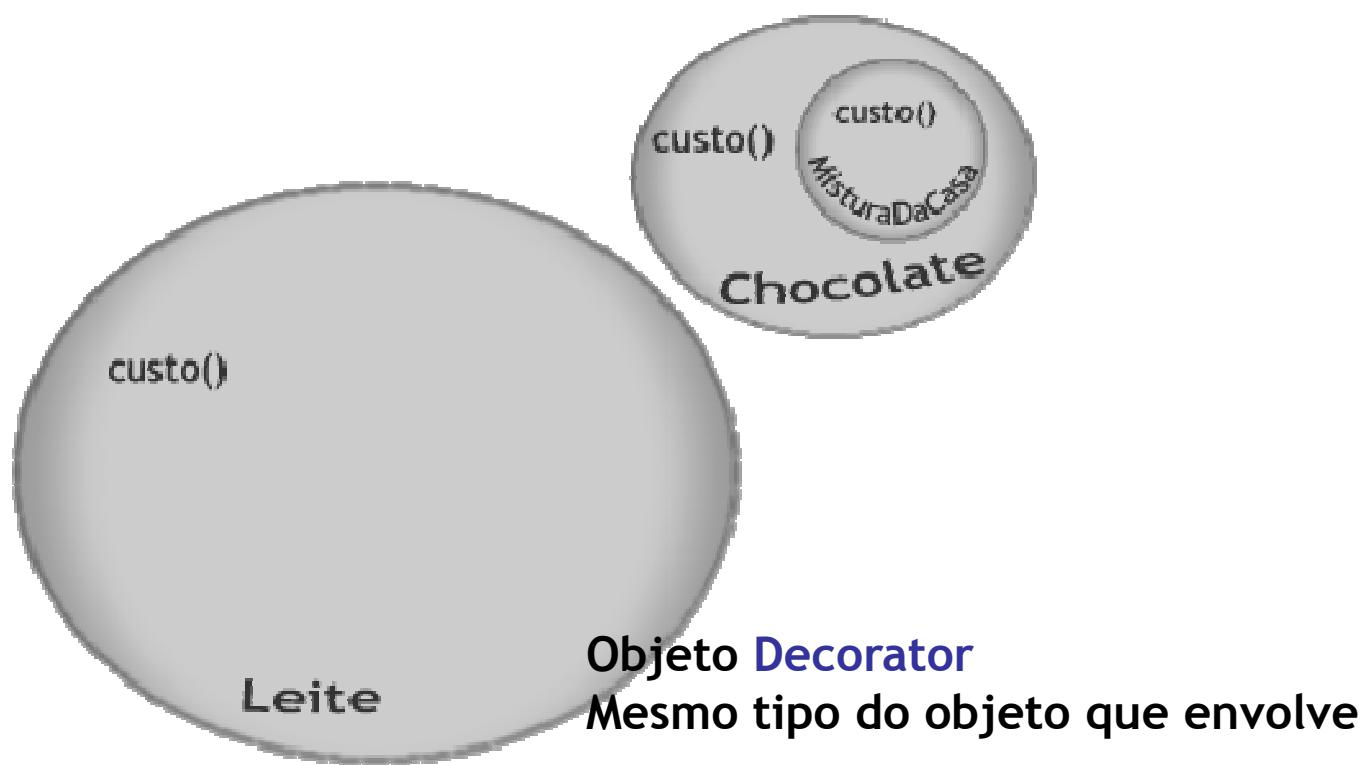
## 2. Criamos um objeto Chocolate e inserimos o MisturaDaCasa nele

Objeto **Decorator**

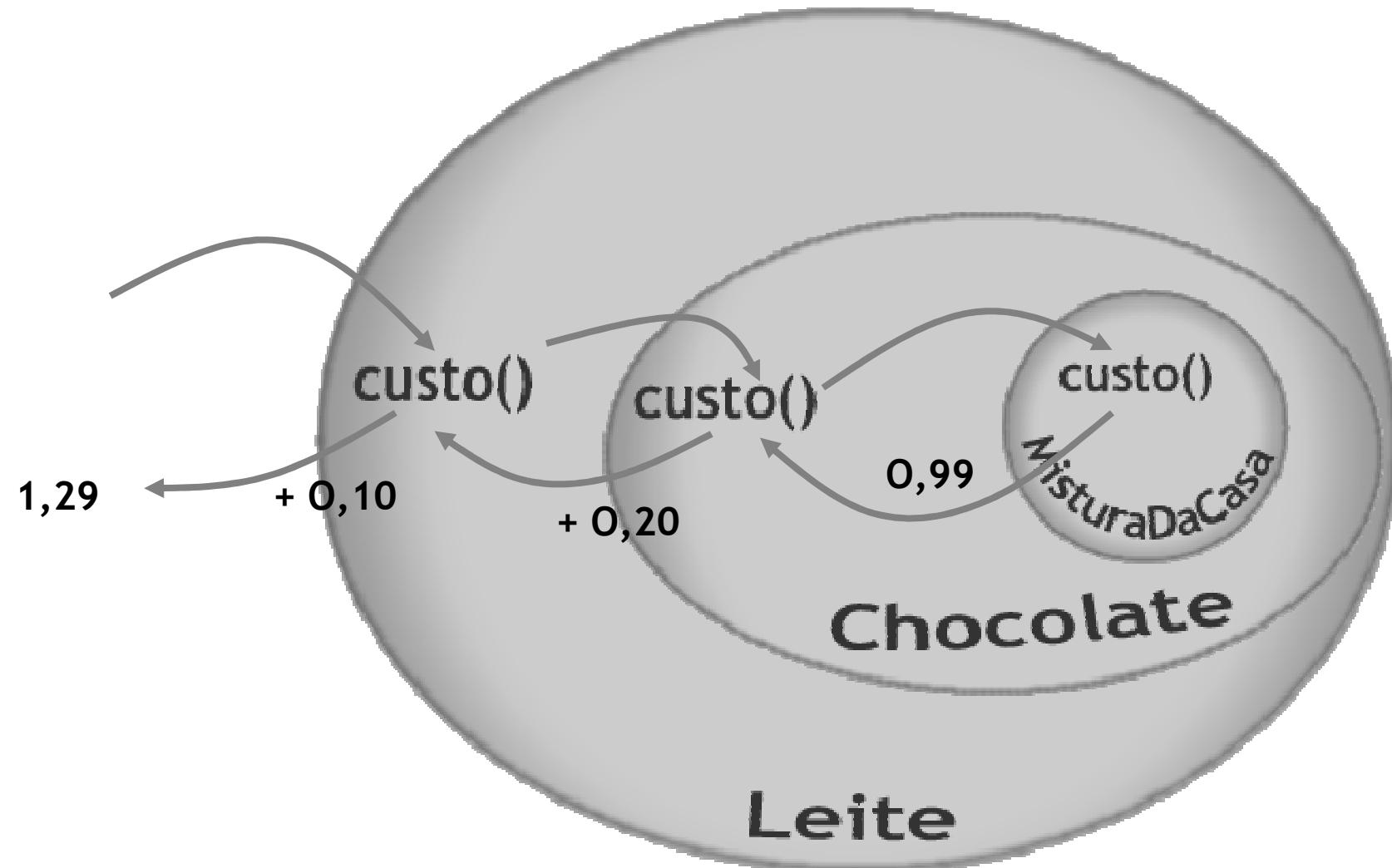
Mesmo tipo do objeto que envolve

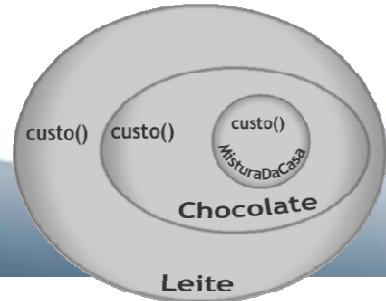


### 3. O cliente também quer Leite Criamos o leite e colocamos tudo nele



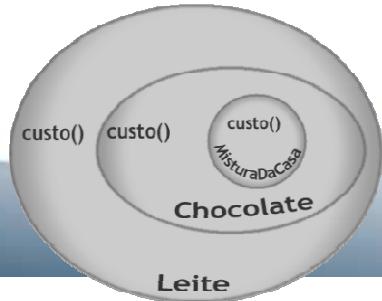
## 4. Calculo total da bebida





## O que temos até então

- **Decorators** têm o **mesmo supertipo** do objeto que **envolvem** (do objeto que “decoram”).
- É possível **envolver** um objeto com **mais de um** Decorator.
- Como o Decorator tem o mesmo tipo do objeto envolvido, podemos passar um **objeto “decorado” no lugar do objeto original**.



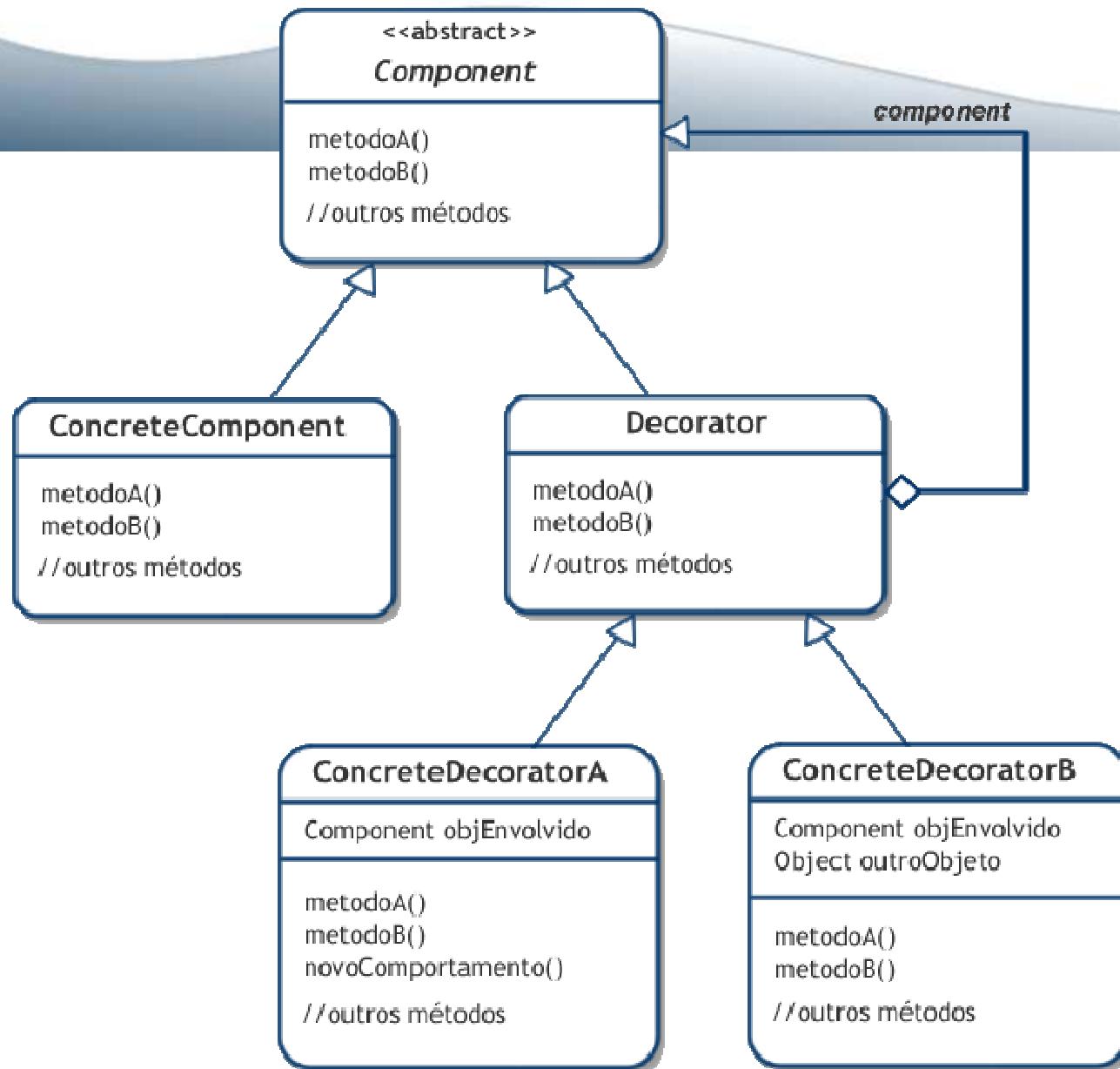
## O que temos até então

- O Decorator **adiciona** seu próprio **comportamento** antes e/ou depois de **delegar** para o objeto que ele decora o resto do trabalho.
- Objetos podem ser decorados a qualquer momento, então podemos **decorar os objetos** de **forma dinâmica** em **tempo de execução** com quantos *decorators* desejarmos.

# Padrão DECORATOR

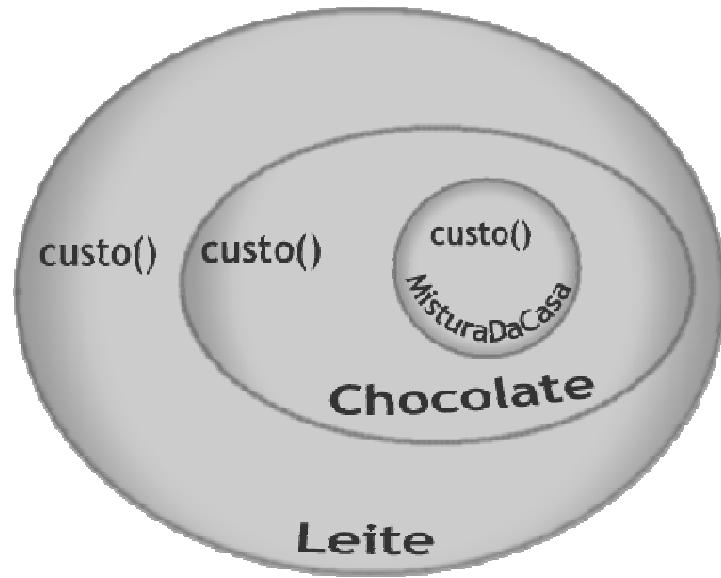
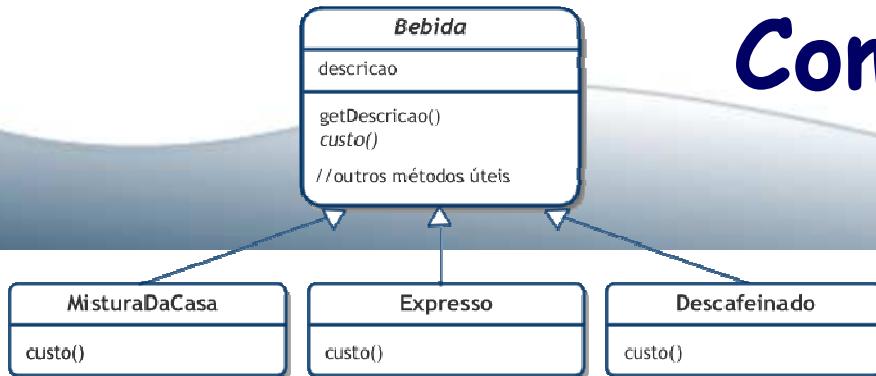
O Padrão Decorator agrega **responsabilidades adicionais** a um objeto dinamicamente. Os **decorators** (decoradores) fornecem uma **alternativa flexível** ao uso de **subclasse** para **extensão** de funcionalidades.

# Diagrama de classes



O Padrão Decorator agrega responsabilidades adicionais a um objeto dinamicamente. Os decorators (decoradores) fornecem uma alternativa flexível ao uso de subclasse para extensão de funcionalidades.

# Como adaptar o padrão para as bebidas?

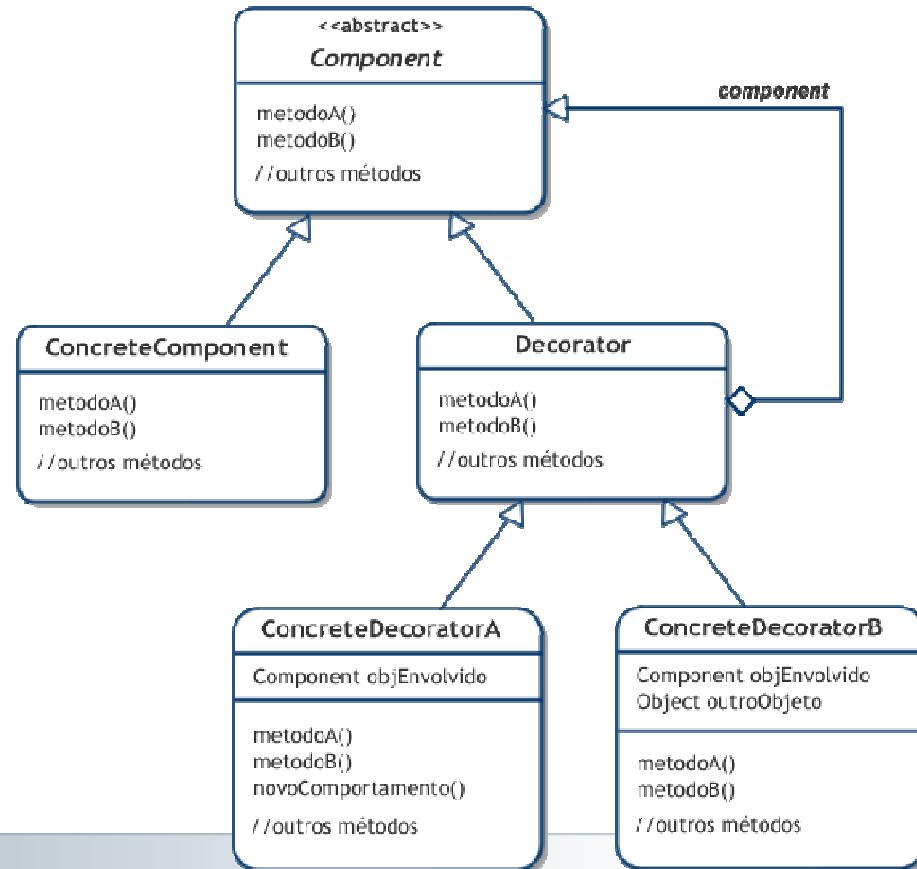


## Ingredientes

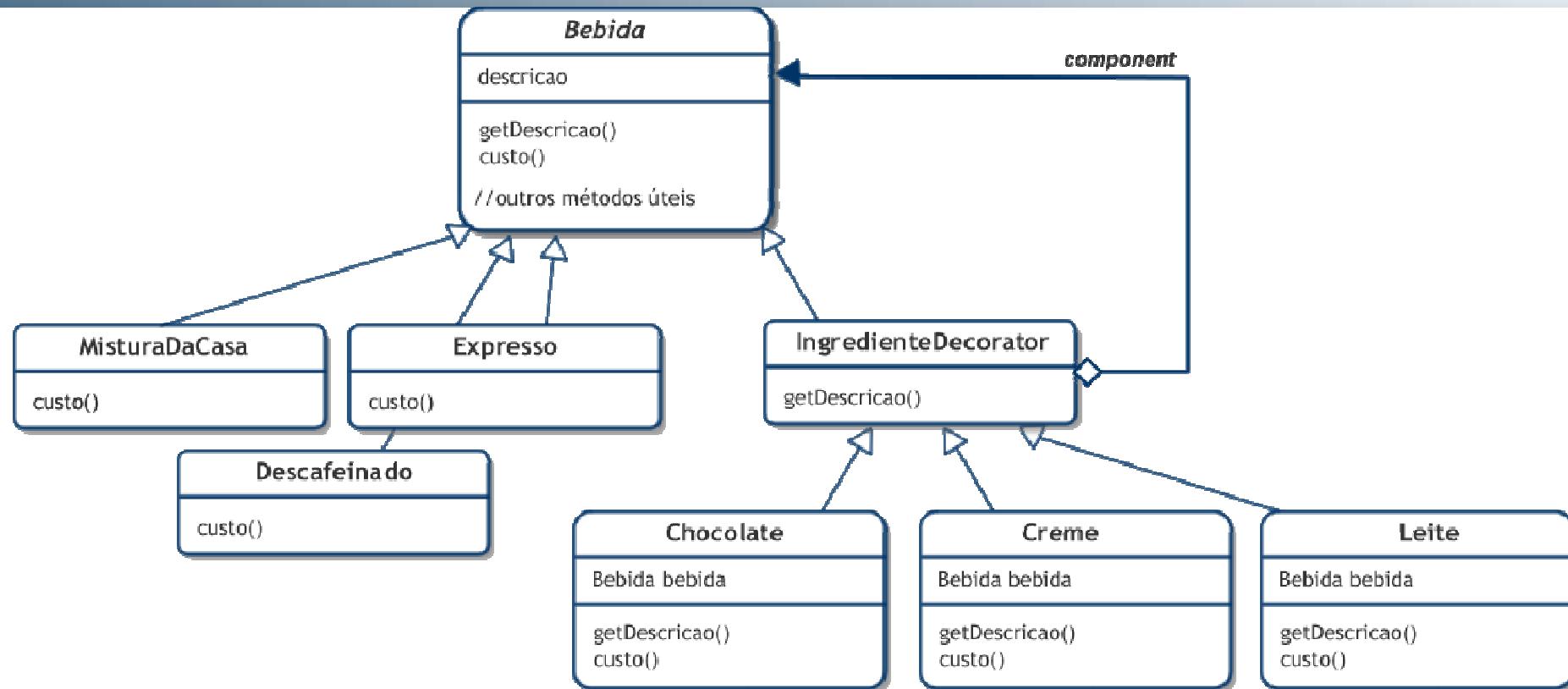
Leite	0,10
Soja	0,15
Chocolate	0,20
Creme	0,15

## Cafés

MisturaDaCasa	0,89
Expresso	1,99
Descafeinado	1,49

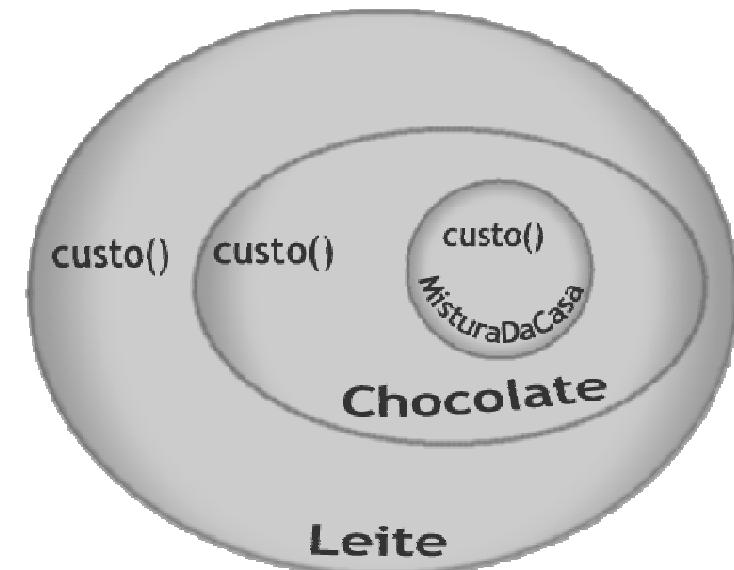


# Diagrama de classes para Bebidas com Decorator



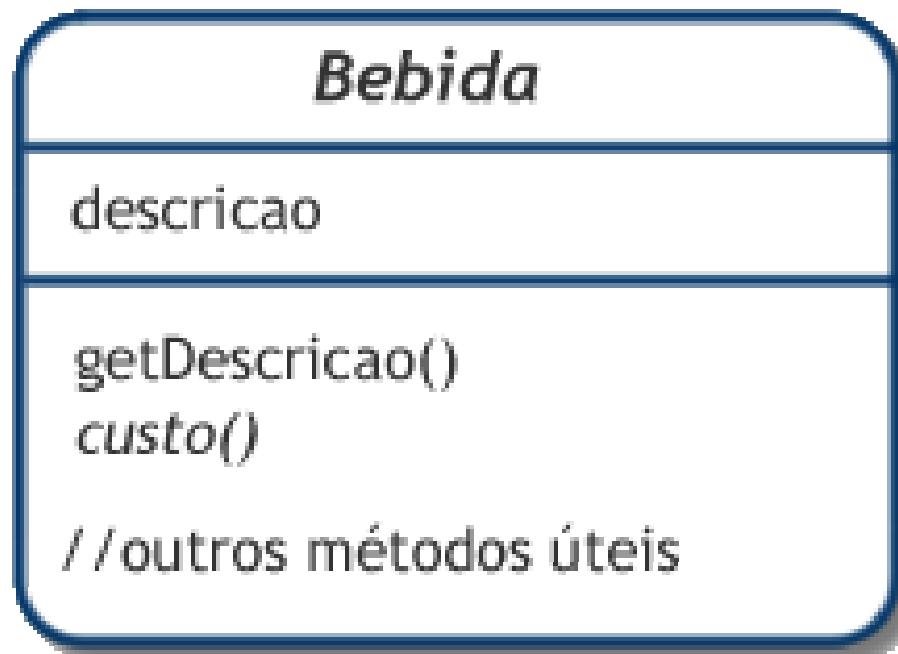
# Exercício

- Como seria representação gráfica do pedido de um café Espresso com 2 doses de chocolate e 1 dose de soja?



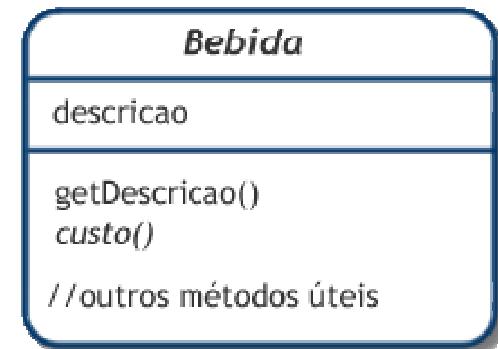
# Codificando a Bebida

## ■ Classe Abstrata

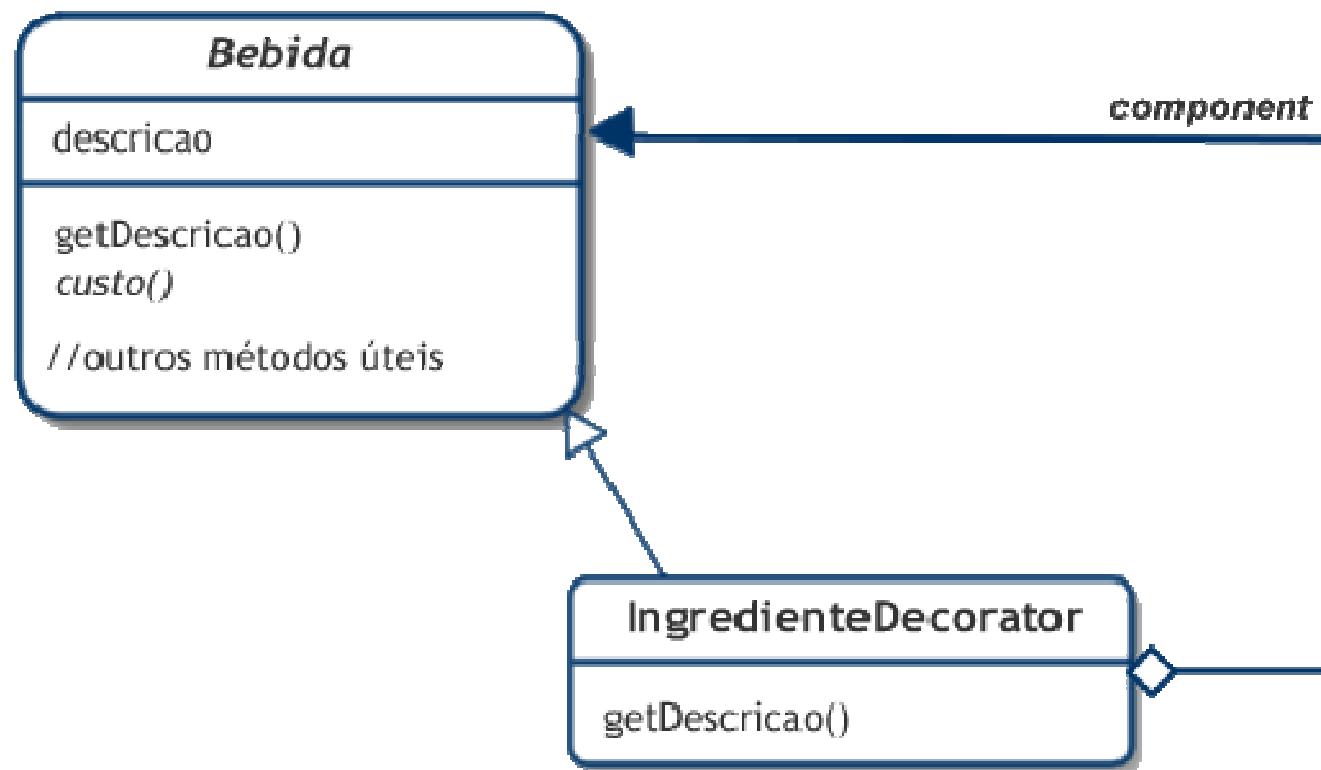


# Bebida.java

```
public abstract class Bebida {  
    String descricao = "Bebida Desconhecida";  
  
    /**  
     * @return the descricao  
     */  
    public String getDescricao() {  
        return descricao;  
    }  
    public abstract double custo();  
}
```



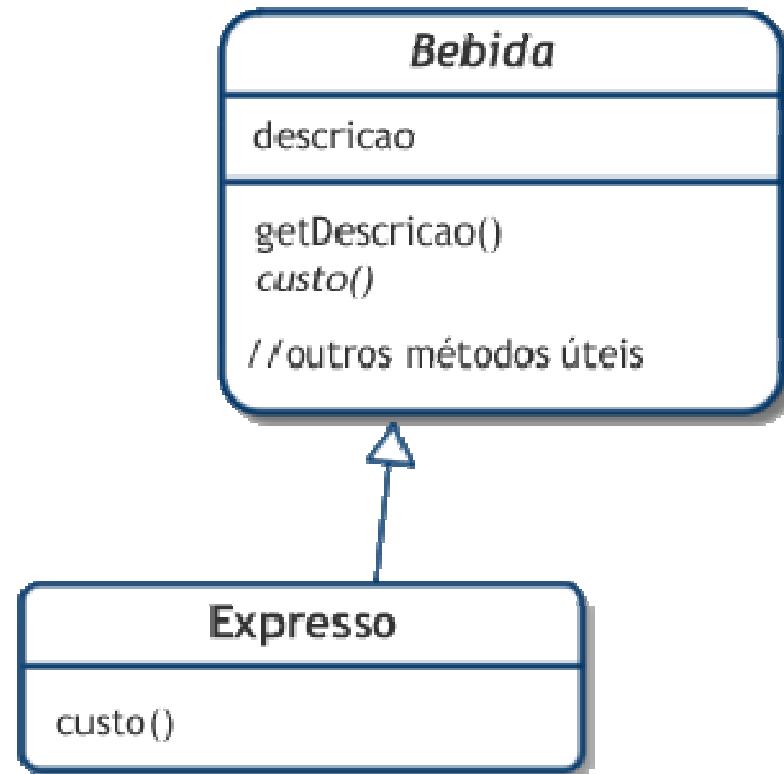
# Codificando IngredienteDecorator



# IngredienteDecorator.java

```
public abstract class IngredienteDecorator  
    extends Bebida {  
  
    Bebida bebida;  
  
    public abstract String getDescricao();  
  
}
```

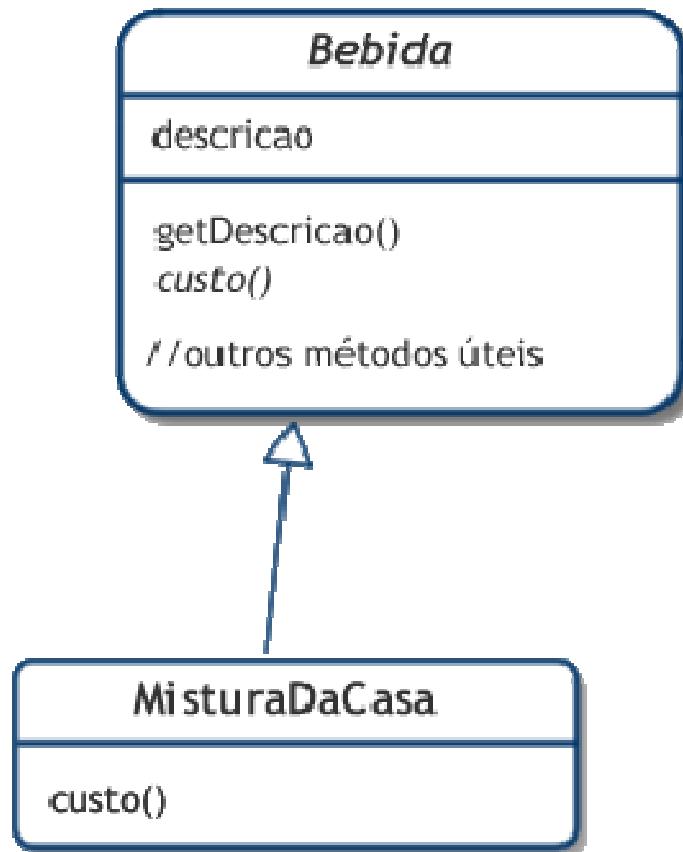
# Codificando o Expresso



# Expresso.java

```
public class Expresso extends Bebida {  
  
    public Expresso() {  
        descricao = "Café Expresso";  
    }  
  
    public double custo() {  
        return 1.99;  
    }  
}
```

# Codificano MisturaDaCasa



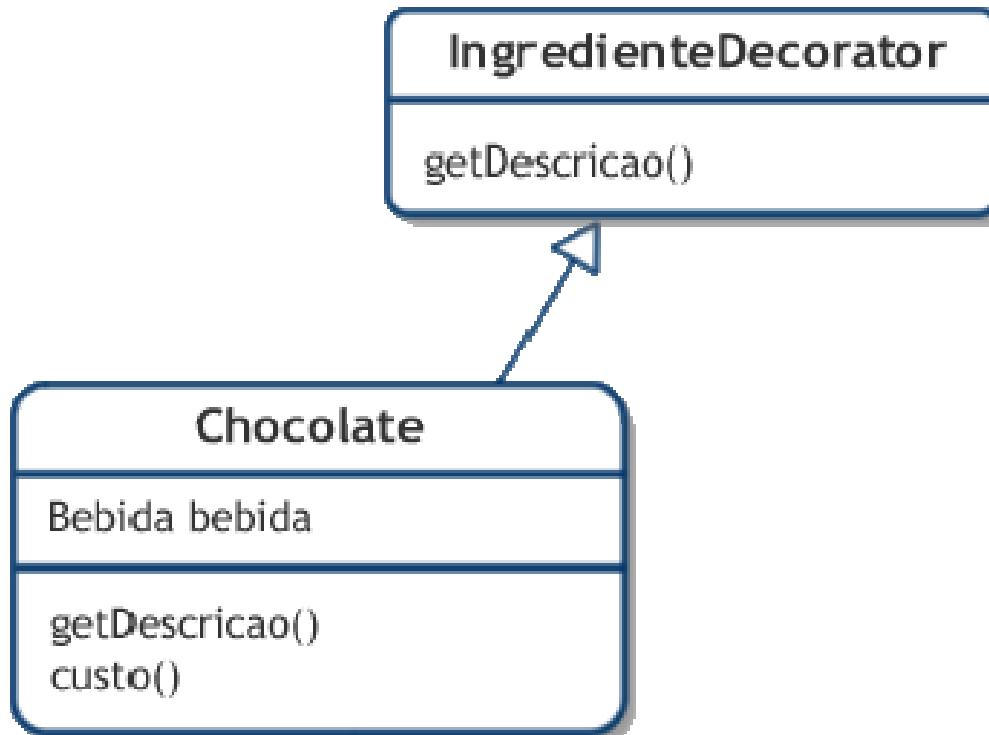
# MisturaDaCasa.java

```
public class MisturaDaCasa extends Bebida {  
  
    public MisturaDaCasa() {  
        descricao = "Café Mistura da Casa";  
    }  
  
    public double custo() {  
        return 0.99;  
    }  
}
```

# Mais Componentes

- Descafeinado.java
- Capuccino.java

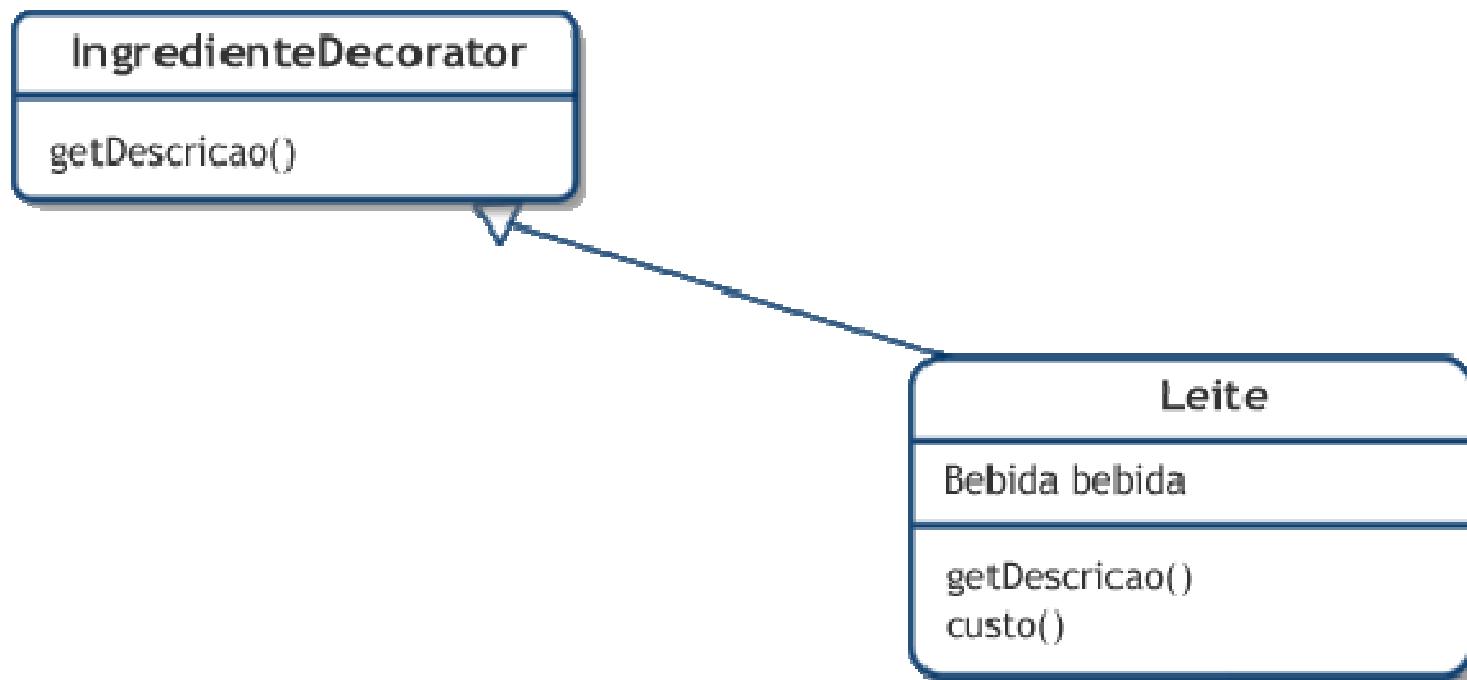
# Codificando Chocolate.java



# Chocolate.java

```
public class Chocolate extends IngredienteDecorator {  
  
    public Chocolate(Bebida bebida) {  
        this.bebida = bebida;  
    }  
  
    public String getDescricao() {  
        return bebida.getDescricao() + ", com Chocolate";  
    }  
  
    public double custo() {  
        return 0.5 + bebida.custo();  
    }  
}
```

# Codificando Leite.java



# Leite.java

```
public class Leite extends IngredienteDecorator {  
    Bebida bebida;  
  
    public Leite(Bebida bebida) {  
        this.bebida = bebida;  
    }  
  
    public String getDescricao() {  
        return bebida.getDescricao() + ", com Leite";  
    }  
  
    public double custo() {  
        return 0.5 + bebida.custo();  
    }  
}
```

# Mais Decorators

- Creme.java
- Soja.java

# Como codificar a Cafeteria JavaCoffe?

- A Classe de testes



# JavaCoffe.java

```
public class JavaCoffe {
    public static void main(String[] args) {
        Bebida bebida = new Expresso();
        System.out.println(bebida.getDescricao()
            + " R$ " + bebida.custo());

        Bebida bebida2 = new MisturaDaCasa();
        bebida2 = new Chocolate(bebida2);
        bebida2 = new Chocolate(bebida2);
        bebida2 = new Leite(bebida2);
        System.out.println(bebida2.getDescricao()
            + " R$ " + bebida2.custo());

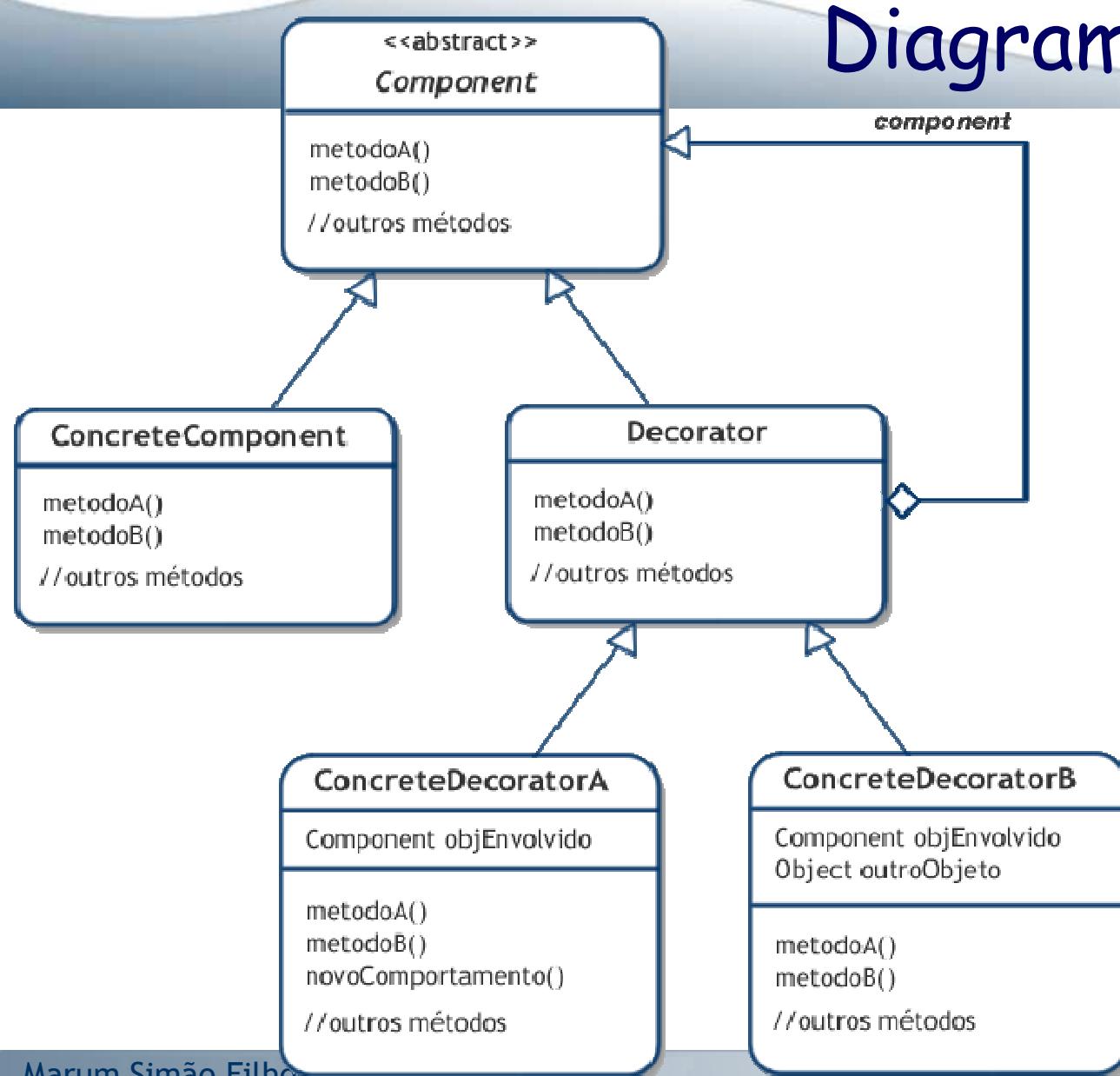
        Bebida bebida3 = new Expresso();
        bebida3 = new Creme(bebida3);
        bebida3 = new Chocolate(bebida3);
        bebida3 = new Leite(bebida3);
        System.out.println(bebida3.getDescricao()
            + " R$ " + bebida3.custo());
    }
}
```

# PONTOS IMPORTANTES

- Herança nem sempre produz designs flexíveis.
- O design deveria permitir adição de comportamento sem afetar o que já existe.
- Composição e delegação podem sempre ser usadas para adicionar novos comportamentos em tempo de execução.
- O padrão Decorator fornece uma alternativa ao uso de subclasses para adicionar comportamento.
- O padrão Decorator é constituído de um conjunto de classes Decorator que são usadas para envolver componentes concretos.
- As classes Decorator são do mesmo tipo das classes que envolvem.
- Decorators mudam o comportamento de seus componentes adicionando novas funcionalidades.
- Você pode envolver componentes com a quantidade de Decorators que desejar.
- O padrão Decorator pode resultar em muitos pequenos objetos, e o uso exagerado pode se tornar complexo.

# Decorator

## Diagrama de Classes



# Aplicabilidade

- Acrescentar responsabilidades a objetos individuais de forma dinâmica e transparente (sem afetar outros objetos).
- Para responsabilidades que podem ser removidas.
- Quando a extensão através de subclasses não é prática.

# Participantes

## ■ Component

- Define uma interface para objetos que podem ter responsabilidades acrescentadas aos mesmos dinamicamente.

## ■ ConcreteComponent

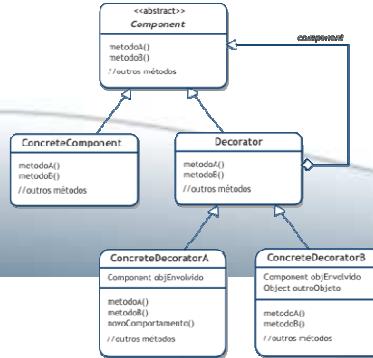
- Define um objeto para o qual responsabilidades adicionais podem ser atribuídas.

## ■ Decorator

- Mantém uma referência para um objeto **Component** e define uma interface que segue a interface de **Component**.

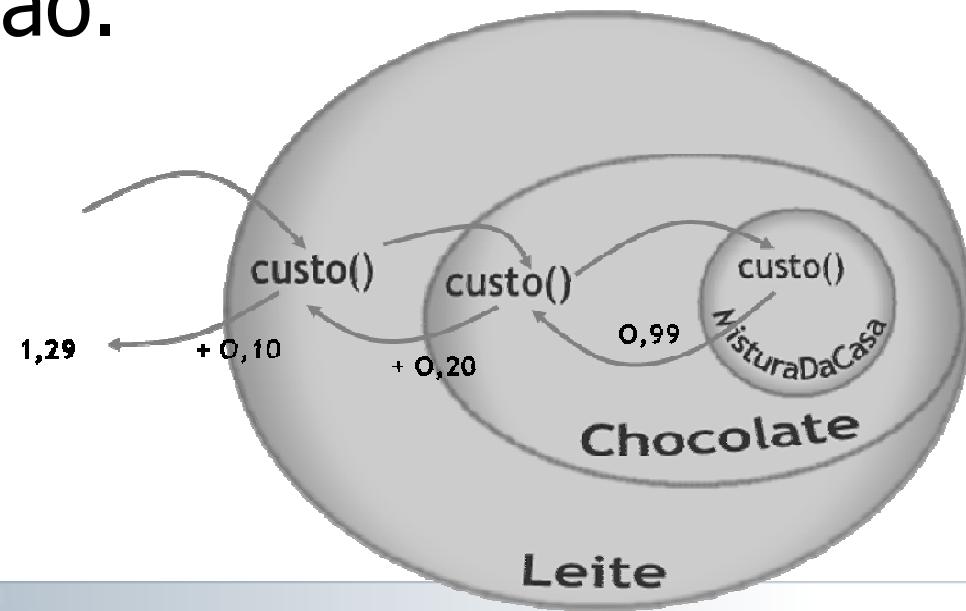
## ■ ConcreteDecorator

- Acrescenta responsabilidades ao **componente**.



# Colaborações

- Decorator repassa solicitações para o seu objeto Component.
- Opcionalmente, o Decorator pode executar operações adicionais antes e depois de repassar a solicitação.

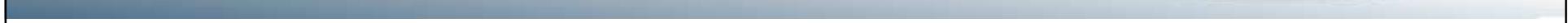


# Consequências

- Maior flexibilidade do que a herança estática.
  - Responsabilidades removidas e acrescentadas em tempo de execução através de associação e dissociação enquanto o mecanismo da herança cria uma nova subclasse para cada funcionalidade.
  - Decorators permitem que propriedades sejam adicionadas 2 ou mais vezes.
- Evita classes sobre carregadas de características na parte superior da hierarquia
  - Use quando for necessário.

# Consequências

- Grande quantidade de pequenos objetos.
  - Sistemas compostos por uma grande quantidade de objetos parecidos.
  - Objetos diferem na maneira com que são interconectados.
  - Sistemas fáceis de customizar por quem os comprehende.
    - Difíceis de aprender e depurar.

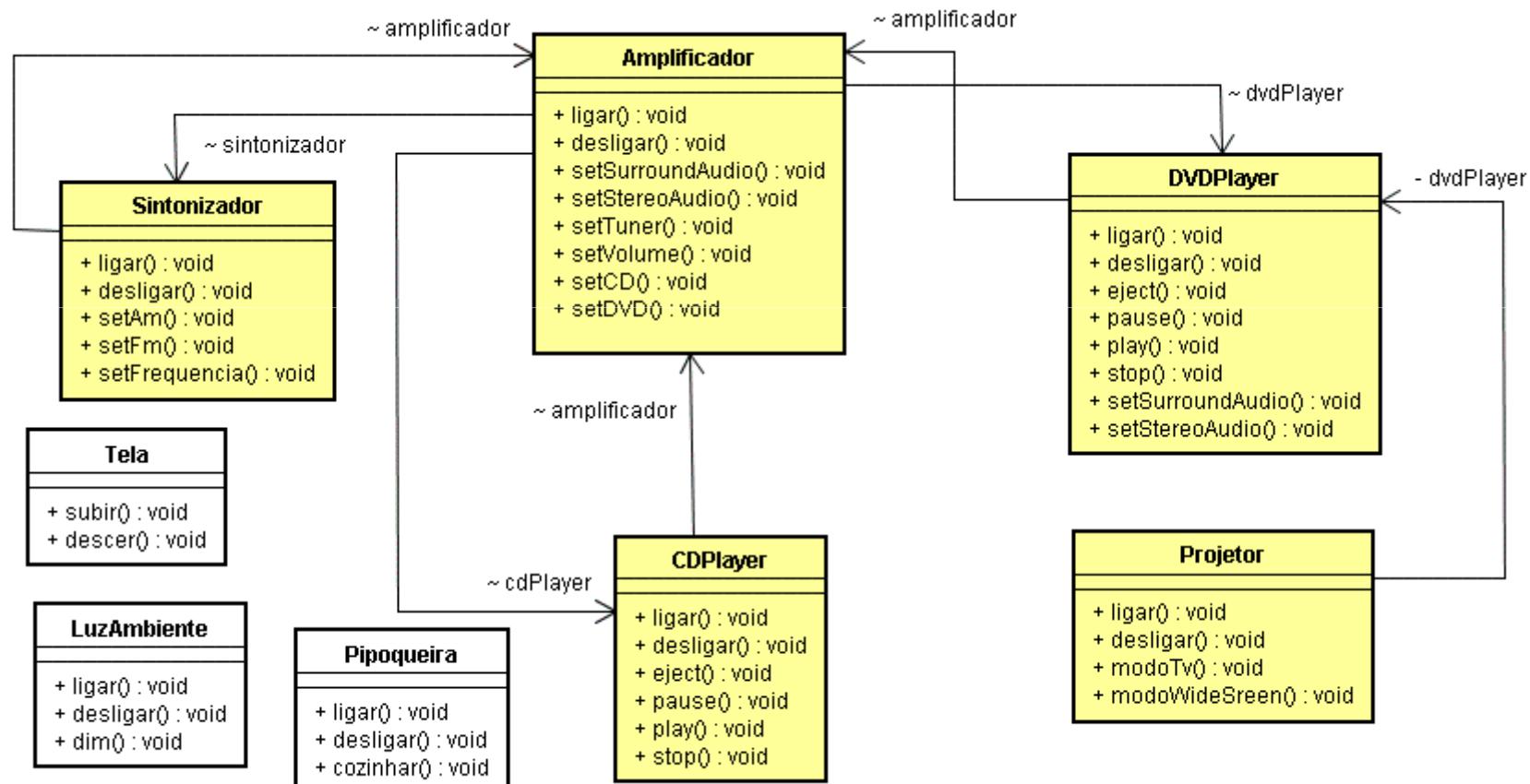


# Padrão Fachada

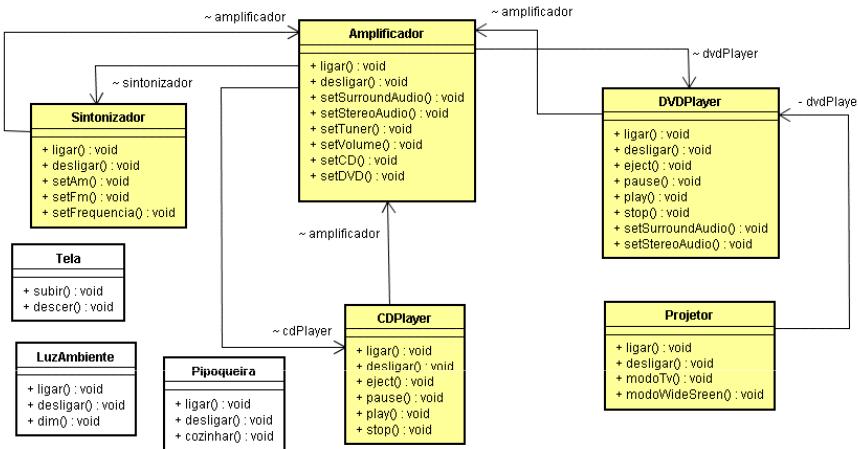
# Fachada

- Alteração de um interface com a intenção de simplificá-la.
- Oculta a complexidade de um conjunto de classes.

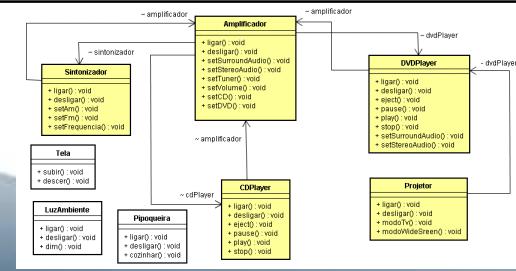
# Imagine um HomeTheater



# HomeTheater

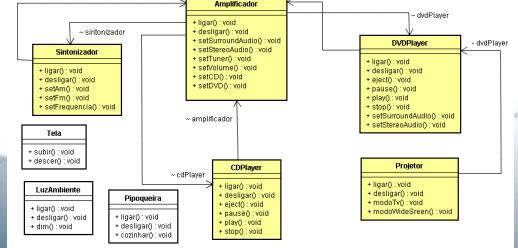


- Muitas classes.
- Muitas interações.
- Você precisa saber o funcionamento de tudo para saber usar corretamente.



# Para assistir um filme

1. Ligar a máquina de pipoca.
2. Colocar a máquina de pipoca em funcionamento.
3. Reduzir as luzes.
4. Baixar a tela.
5. Ligar o projetor.
6. Configurar a entrada do projetor para DVD.
7. Colocar o projetor em modo wide-screen
8. Ligar o amplificador de áudio.
9. Configurar a entrada do amplificador para DVD.
10. Configurar o amplificador para surround.
11. Ajustar o amplificador para o volume médio (5).
12. Ligar o DVD Player.
13. Acionar o play no DVD Player.



# Para assistir um filme Em Java

1. Ligar a máquina de pipoca
2. Colocar a máquina de pipoca em funcionamento
3. Reduzir as luzes
4. Baixar a tela
5. Ligar o projetor
6. Configurar a entrada do projeto para DVD
7. Colocar o projetor em modo wide-screen
8. Ligar o amplificador de áudio
9. Configurar o amplificador para DVD
10. Configurar o amplificador para surround
11. Ajustar o amplificador para o volume médio
12. Ligar o DVD Player
13. Acionar o play no DVD Player

```

pipoqueira.ligar();
pipoqueira.cozinhar();

luz.dim(10);

tela.descer();

projetor.ligar();
projetor.setInput(dvdPlayer);
projetor.modoWideSreen();

amplificador.ligar();
amplificador.setDVD(dvdPlayer);
amplificador.setSurroundAudio();
amplificador.setVolume(5);

dvdPlayer.ligar();
dvdPlayer.play(filme);

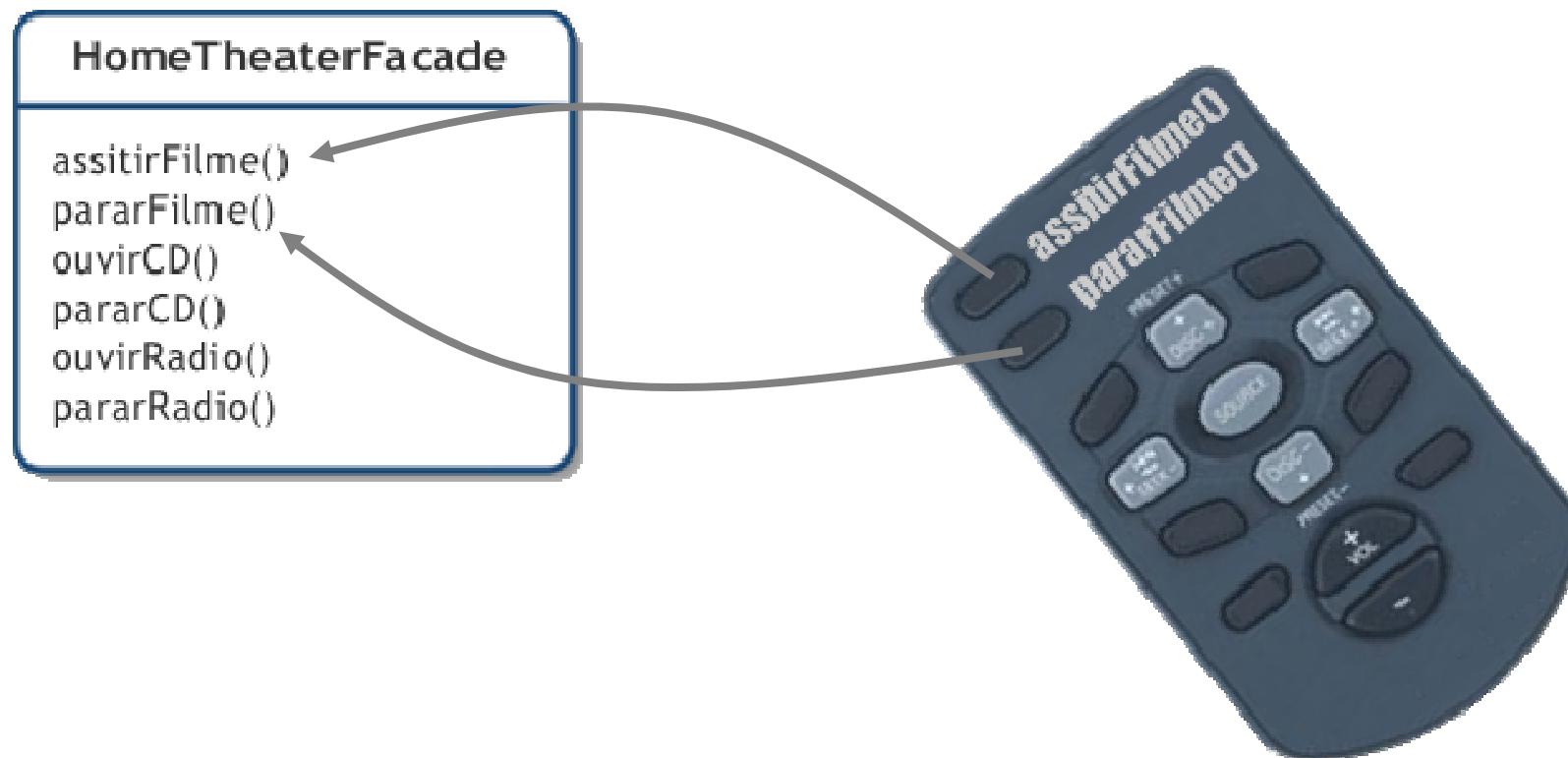
```

## + Problemas

- Neste tipo de situação, ainda há uma ordem para fazer tudo.
- Ao final, pode-se precisar fazer as ações em ordem inversa.
- E para ouvir um CD ou rádio?
- Se precisar atualizar o sistema, terá que aprender um novo algoritmo.

# Solução para a complexidade

## ■ A FACHADA



## HomeTheaterFacade

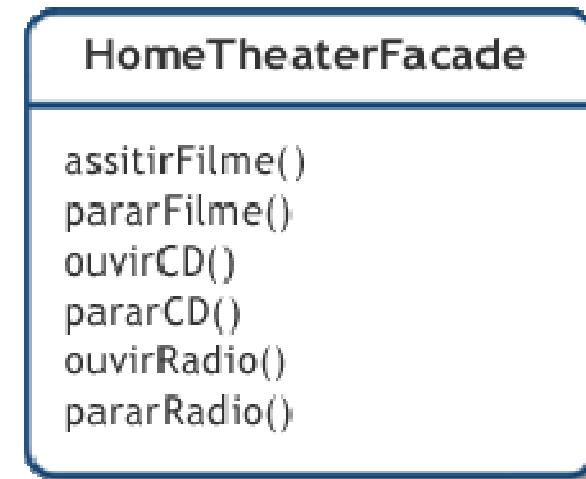
assitirFilme()  
pararFilme()  
ouvirCD()  
pararCD()  
ouvirRadio()  
pararRadio()

# A Fachada

- Uma classe que exponha somente alguns métodos simples:
  - assistirFilme()
  - pararFilme()
- Trata os componentes do Home Theater como um subsistema.
- Aciona as diversas partes necessárias para implementar o método assistirFilme(), por exemplo.

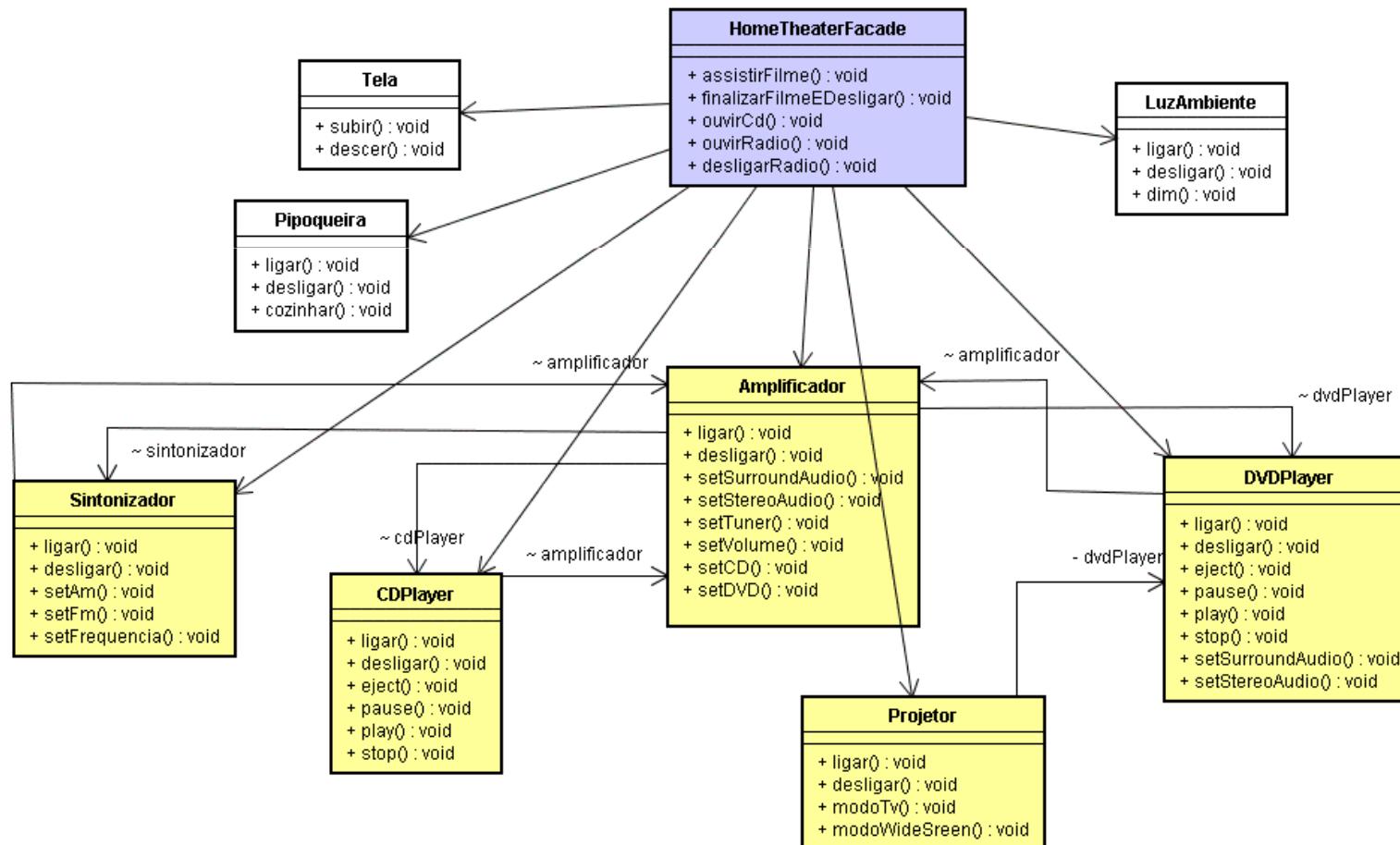
# A Fachada

- O cliente, assim, pode usar os métodos da Fachada e não do subsistema.
- Para assistir a um filme, basta acionar um método que se comunica com todo o subsistema.



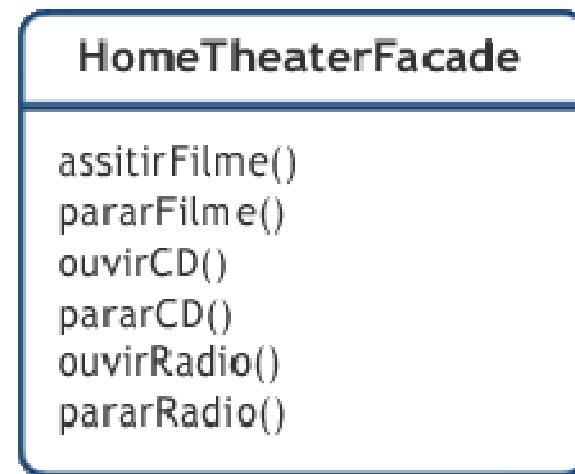
# Solução para a complexidade

## ■ A FACHADA



# A Fachada

- Preserva o acesso direto ao subsistema.
- Você ainda pode usar os métodos de cada aparelho.



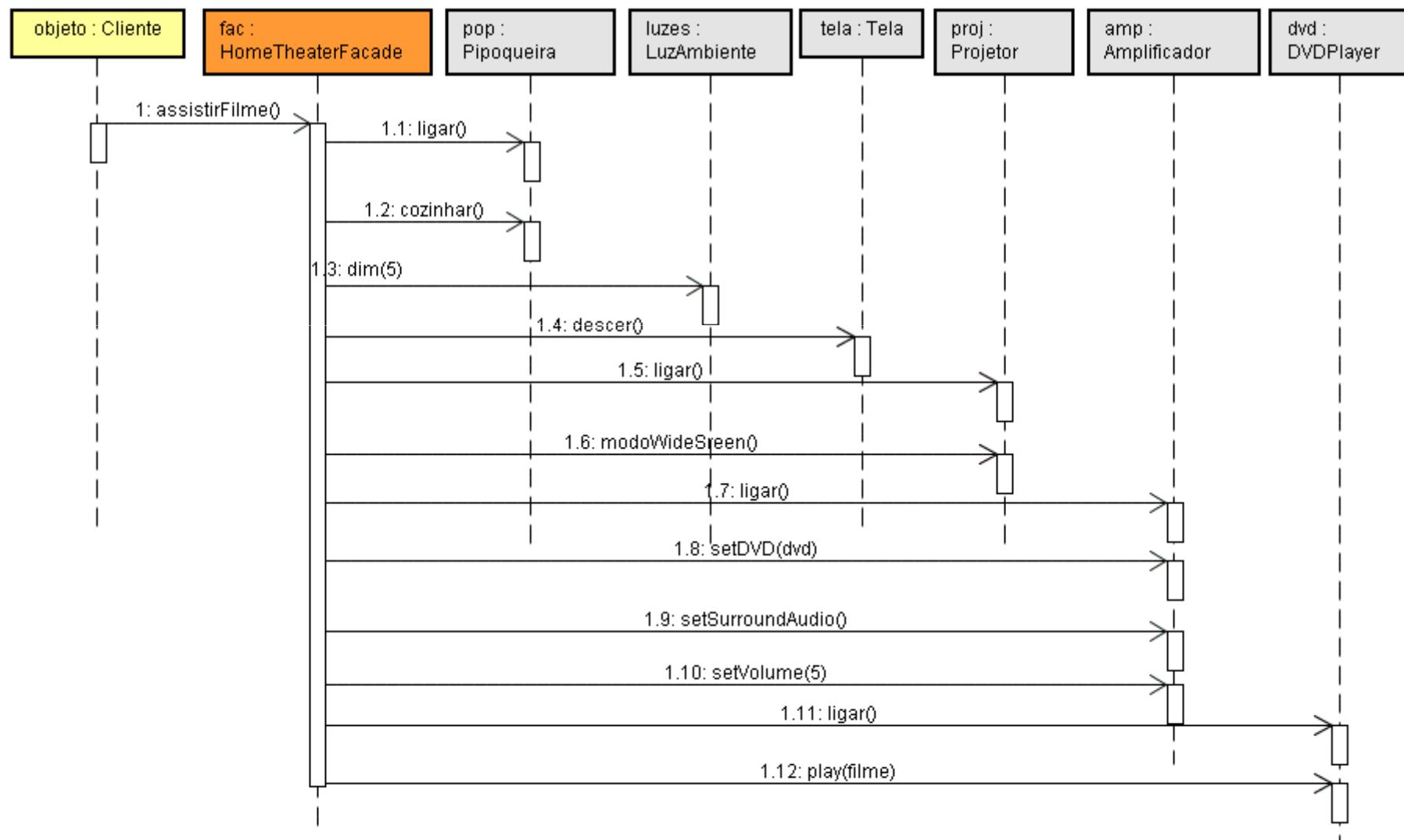
# Implementando a Fachada

```
public class HomeTheaterFacade {  
    Amplificador amplificador;  
    Sintonizador sintonizador;  
    CDPlayer cdPlayer;  
    DVDPlayer dvdPlayer;  
    Projetor projetor;  
    LuzAmbiente luz;  
    Tela tela;  
    Pipoqueira pipoqueira;  
  
    /**  
     * Construtor  
     */  
    public HomeTheaterFacade(  
        Amplificador amplificador,  
        Sintonizador sintonizador,  
        CDPlayer cdPlayer,  
        DVDPlayer dvdPlayer,  
        Projetor projetor,  
        LuzAmbiente luz,  
        Tela tela,  
        Pipoqueira pipoqueira) {  
  
        this.amplificador = amplificador;  
        this.sintonizador = sintonizador;  
        this.cdPlayer = cdPlayer;  
        this.dvdPlayer = dvdPlayer;  
        this.projetor = projetor;  
        this.luz = luz;  
        this.tela = tela;  
        this.pipoqueira = pipoqueira;  
    }  
}
```

# Implementando a Fachada

```
/**  
 * Método que simplifica o uso de vários objetos  
 * @param filme  
 */  
  
public void assistirFilme(Filme filme) {  
    System.out.println(  
        "Tudo ok para assistir um filme!");  
    pipoqueira.ligar();  
    pipoqueira.cozinhar();  
  
    luz.dim(10);  
  
    tela.descer();  
  
    projetor.ligar();  
    projetor.setInput(dvdPlayer);  
    projetor.modoWideSreen();  
  
    amplificador.ligar();  
    amplificador.setDVD(dvdPlayer);  
    amplificador.setSurroundAudio();  
    amplificador.setVolume(5);  
  
    dvdPlayer.ligar();  
    dvdPlayer.play(filme);  
}  
/**  
 * Outro método que simplifica  
 * um conjunto de tarefas  
 */  
  
public void pararFilme(){  
    System.out.println("Desligando o  
home...");  
    pipoqueira.desligar();  
    luz.ligar();  
    tela.subir();  
    projetor.desligar();  
    amplificador.desligar();  
    dvdPlayer.parar();  
    dvdPlayer.eject();  
    dvdPlayer.desligar();  
}}
```

# Fachada



# Para assistir um filme de maneira simplificada

```
public class TesteHomeTheater {  
    public static void main(String[] args) {  
  
        //Criação das instâncias dos componentes  
  
        HomeTheaterFacade ht =  
            new HomeTheaterFacade(amplificador,  
                sintonizador, cdPlayer,dvdPlayer, projetor,  
                luz, tela, pipoqueira);  
  
        ht.assistirFilme();  
        ht.pararFilme();  
    }  
}
```

# Questões sobre a fachada

- Encapsula as classes do sistema?
- Acrescenta funcionalidade?
- Cada subsistema possui somente um fachada?
- Qual o benefício da fachada além de fornecer uma interface simples?
- Qual a diferença da Fachada para o Adaptador?

## Um pouco mais

- Além de simplificar uma interface...
  - Desconecta o cliente de um subsistema de componentes.
- Uma fachada simplifica e um adapter converte uma interface para algo diferente.

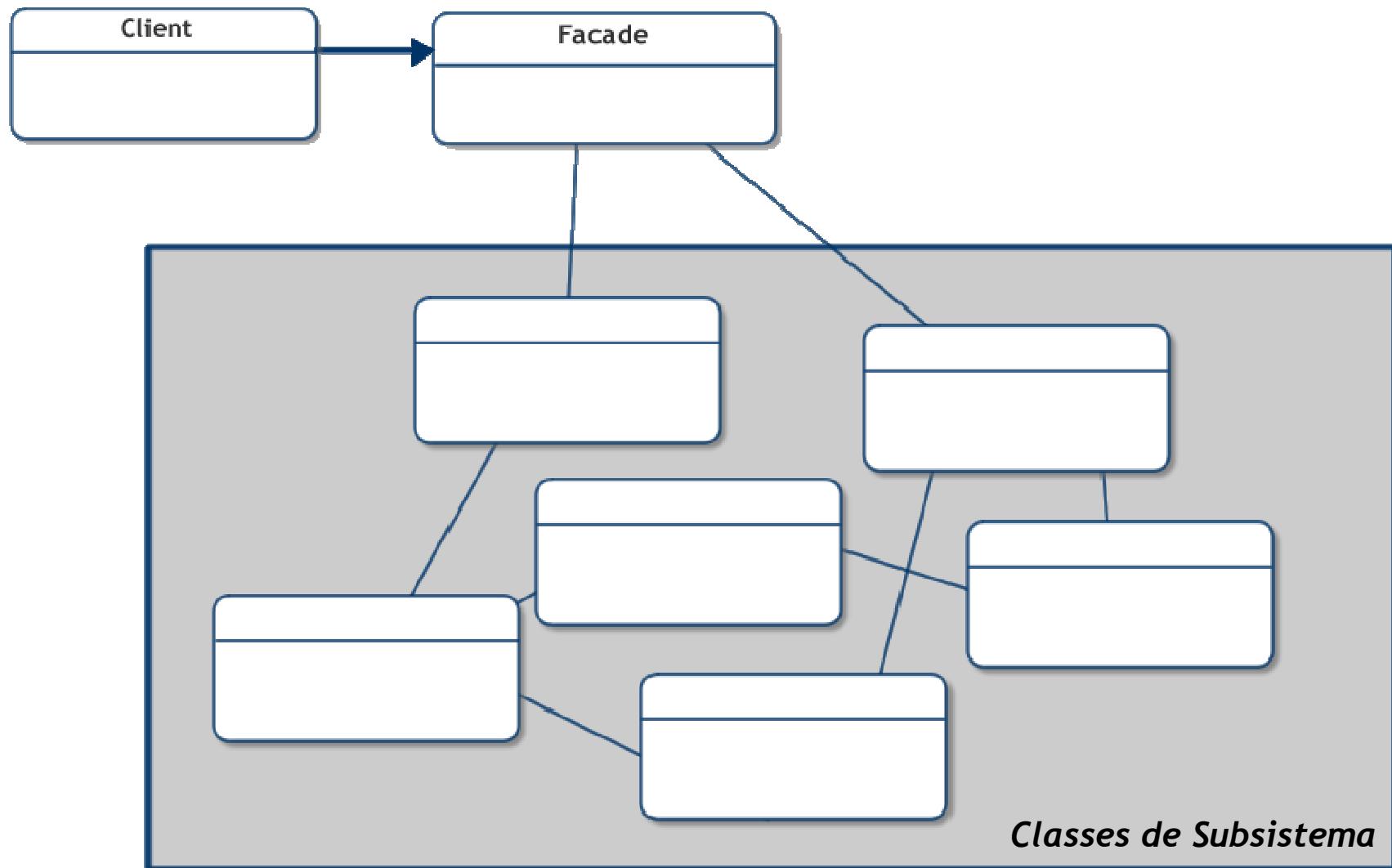
## +1 Padrão **FACADE**

O Padrão **Facade** fornece uma **interface unificada** para um conjunto de interfaces em um subsistema. A **Fachada** define uma interface de **nível mais alto** que **facilita** a utilização de um subsistema.

# Aplicabilidade

- Desejar fornecer uma interface simples para um subsistema complexo.
- Existirem muitas dependências entre os clientes e as classes de implementação de uma abstração.
- Desejar estruturar em camadas seus subsistemas.
  - Use uma Façade para definir o ponto de entrada de cada nível do subsistema.

# Diagrama de classes



# Participantes

## ■ Fachada

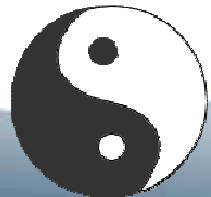
- Conhece quais as classes do subsistema são responsáveis pelo atendimento de uma solicitação.

## ■ Classes de Subsistema

- Implementam a funcionalidade do subsistema.
- Encarregam-se do trabalho atribuído a elas pelo objeto Fachada.
- Não têm conhecimento da Fachada, ou seja, não possuem referência para a mesma.

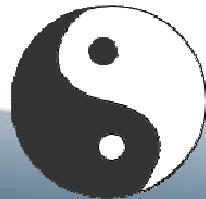
# Colaborações

- Os clientes se comunicam com um subsistema através do envio de solicitações para a Fachada, a qual repassa para os objetos apropriados do subsistema.
- Embora os objetos do subsistema executem o trabalho real, a fachada pode ter que efetuar trabalho próprio dela para traduzir a sua interface para as interfaces do subsistema.
- Os clientes que usam a fachada não têm que acessar os objetos do subsistema diretamente.



# Princípio de Design

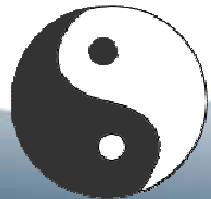
- “Princípio do **Mínimo Conhecimento**: só interaja com seus amigos mais próximos.”
  - Ter cuidado com o número de classes com que qualquer objeto interage e também com a forma como essa interação ocorre.
  - Minimizar o acoplamento entre classes:
    - Evita um efeito em cascata quando houver alterações em um sistema.



# Princípio de Design

## ■ **Mínimo Conhecimento** (Lei de Demétrio)

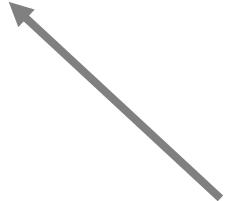
- Em um determinado objeto, a partir de qualquer método do mesmo, só podemos invocar métodos que pertençam:
  - Ao próprio objeto;
  - A objetos que tenham sido passados como parâmetros para o método;
  - A qualquer objeto que seja criado ou instanciado pelo método;
  - A qualquer componente do objeto.

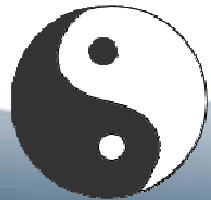


# Princípio de Design

## ■ SEM o uso do princípio

```
public float getTemperatura() {  
    Termometro t = estacao.getTermometro();  
    return t.getTemperatura();  
}  
  
estacao.getTermometro().getTemperatura();
```

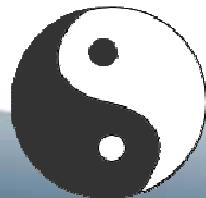




# Princípio de Design

## ■ COM o uso do princípio

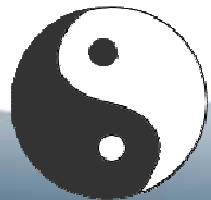
```
public float getTemperatura() {  
    return estacao.getTemperatura();  
}
```



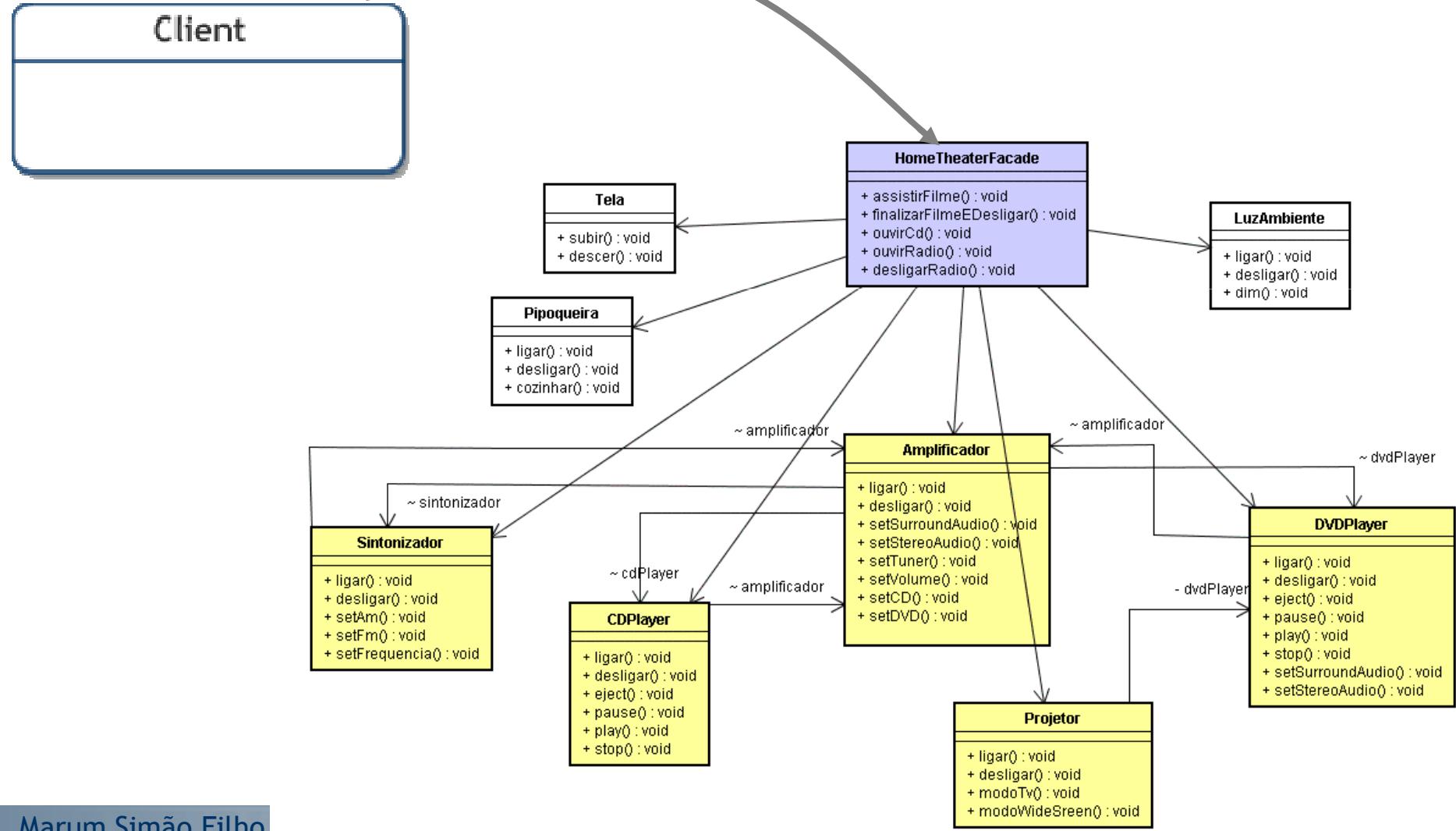
# Princípio de Design

## ■ Uma classe em conformidade com o princípio

```
public class Carro {  
    Motor motor;           ← Componente da classe. Podemos chamar seus métodos.  
  
    public Carro() { }  
  
    public void ligar(Chave chave) {  
        boolean autorizado = chave.virar();  
        motor.ligar();           ← Método de um componente da classe.  
        atualizarVisorDoCarro(); ← Método local. Podemos chamá-lo.  
  
        Portas portas = new Portas();  
        portas.trancar();         ← Objeto local. Podemos chamar  
        seus métodos  
    }  
    private void atualizarVisorDoCarro() {  
        // TODO Auto-generated method stub  
    }  
}
```

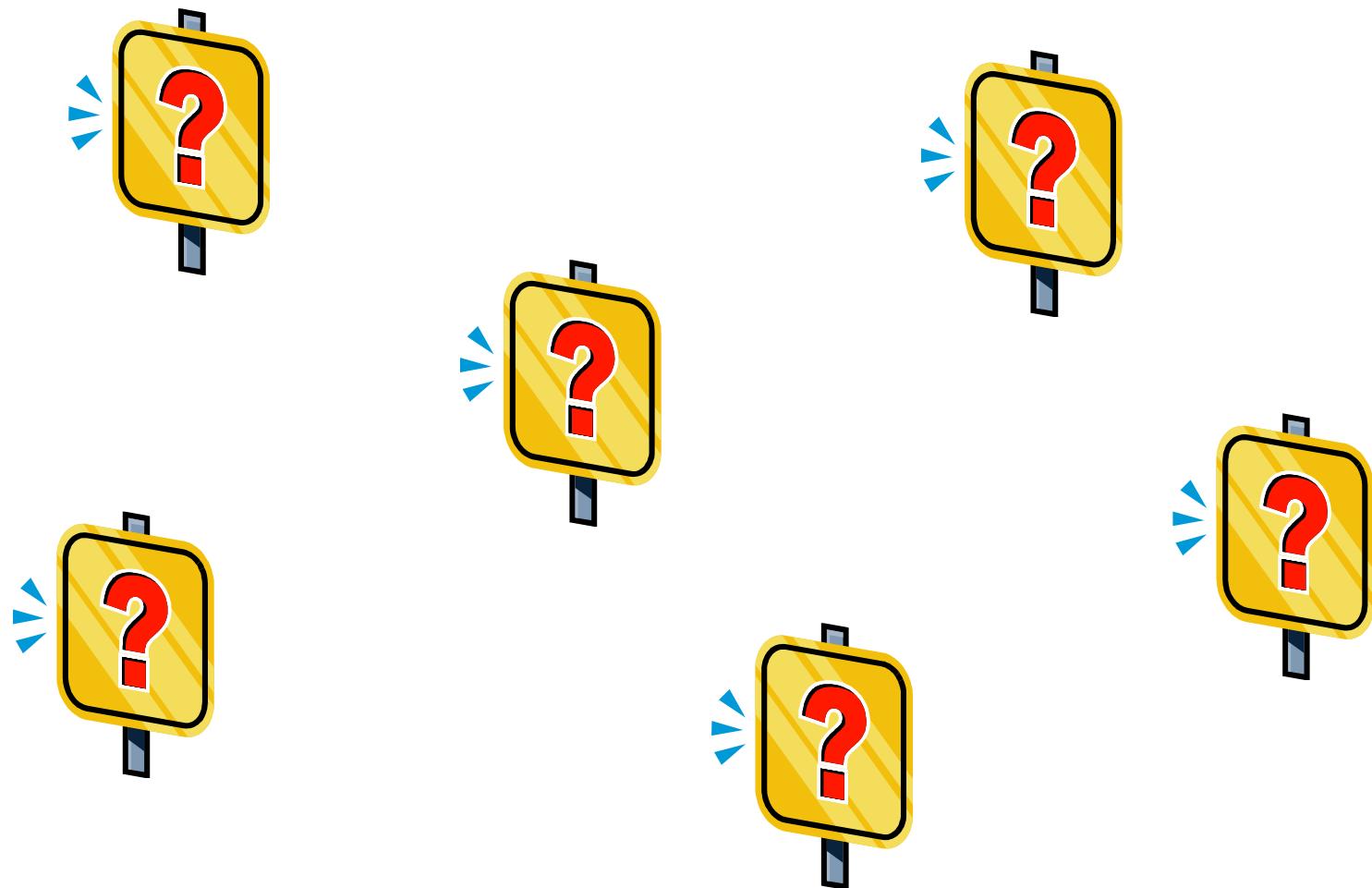


# Princípio de Design X Fachada



# Consequências

- Introduz simplicidade e facilidade para o cliente.
- Permite estruturar melhor a aplicação.
- Maior número de classes que envolvem outras classes, isto pode gerar:
  - Aumento de complexidade;
  - Aumento no tempo de desenvolvimento de software;
  - Redução de desempenho.





***Obrigado!!!***

**Colaborações:**

Prof. Eduardo Mendes

Prof. Régis Simão

**Faculdade 7 de Setembro**