

# Uma introdução a

# Padrões

# de

# Design

com exemplos em Java

*Helder da Rocha (helder.darocha@gmail.com)*



# Objetivos desta palestra

- Explicar o que são padrões de design
- Apresentar uma introdução aos principais padrões de design clássicos (GoF) usando exemplos em código Java
  - Serão explorados em mais detalhe apenas os padrões mais importantes
- Esta palestra tem como objetivo motivá-lo(a) a estudar e aprender os padrões

# O que é um padrão?

- Maneira testada ou documentada de alcançar um objetivo qualquer
  - Padrões são comuns em várias áreas da engenharia
  - Padrões **não são invenções originais**
- Design Patterns, ou Padrões de Design/Projeto
  - Padrões para alcançar objetivos na engenharia de software
  - Inspirado em "**A Pattern Language**" de Christopher Alexander, sobre padrões de arquitetura de cidades, casas e prédios
  - "**Design Patterns**" de Erich Gamma, John Vlissides, Ralph Jonhson e Richard Helm, conhecidos como "The Gang of Four", ou GoF, descreve 23 padrões de projeto úteis.

# O que é um padrão?

*"Cada padrão descreve um problema que ocorre repetidas vezes em nosso ambiente, e então descreve o núcleo da solução para aquele problema, de tal maneira que pode-se usar essa solução milhões de vezes **sem nunca fazê-la da mesma forma duas vezes**"*

Christopher Alexander, sobre padrões em Arquitetura

*"Os padrões de projeto são descrições de objetos que se comunicam e classes que são customizadas para resolver **um problema genérico de design em um contexto específico**"*

Gamma, Helm, Vlissides & Johnson, sobre padrões em software

# Formas de classificação

- Há várias formas de classificar os padrões.  
Gamma et al [2] os classifica de duas formas
  - Por propósito: (1) **criação** de classes e objetos, (2) alteração da **estrutura** de um programa, (3) controle do seu **comportamento**
  - Por escopo: **classe** ou **objeto**
- Metsker [1] os classifica em 5 grupos, por intenção (problema a ser solucionado):
  - (1) oferecer uma **interface**,
  - (2) atribuir uma **responsabilidade**,
  - (3) realizar a **construção** de classes ou objetos
  - (4) controlar formas de **operação**
  - (5) implementar uma **extensão** para a aplicação

# Por que aprender padrões?

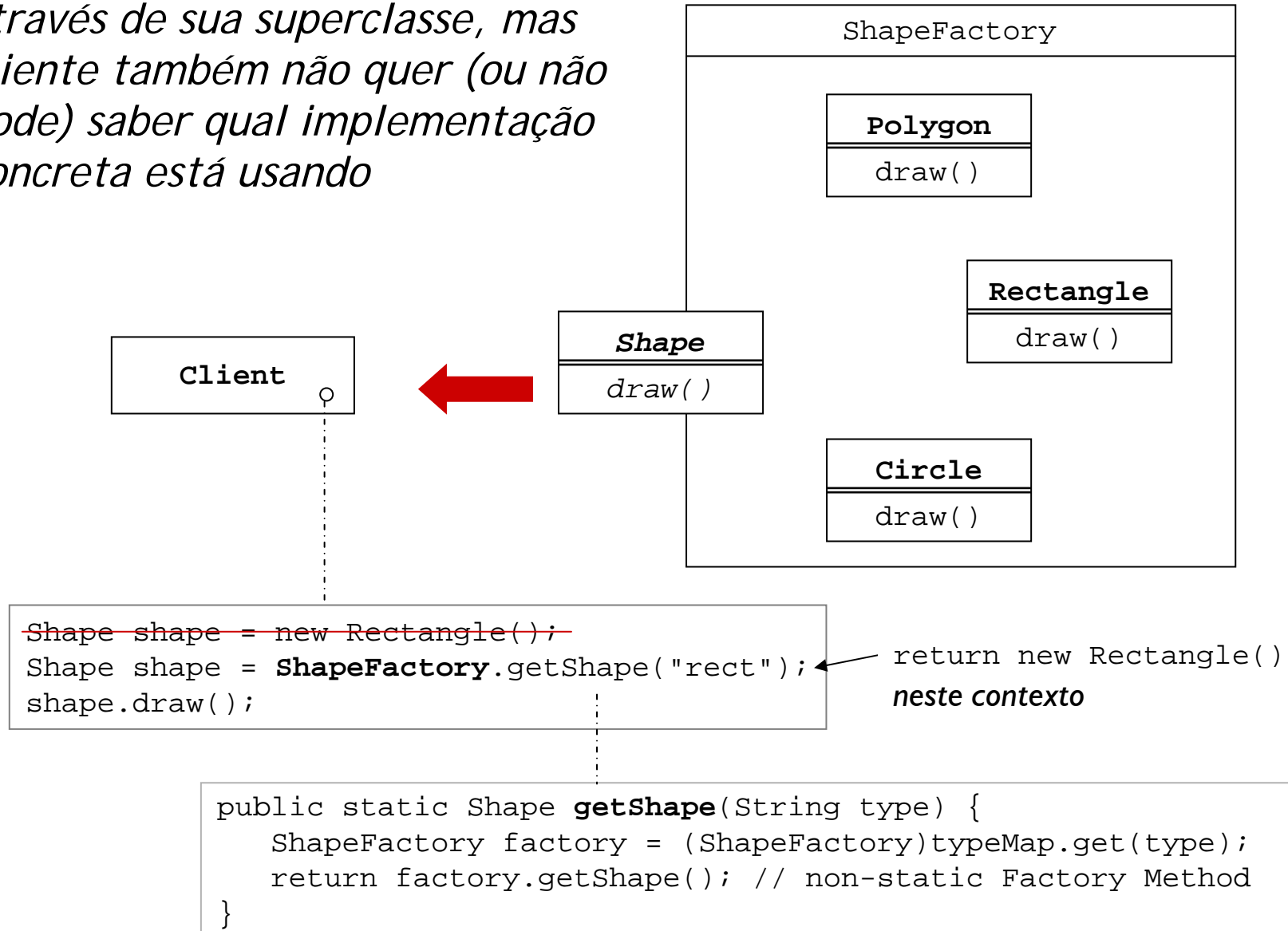
- Aprender com a **experiência** dos outros
  - Aprender os padrões ajudam um novato a agir mais como um **especialista**
- Programar melhor com **orientação a objetos**
- Desenvolver software de melhor **qualidade**
- Aprender um **vocabulário** comum
- Compreender sistemas existentes
- Saber a melhor forma de converter um modelo de análise em um modelo de implementação
- Ter um caminho e um alvo para refatoramento

# 1

## Factory Method

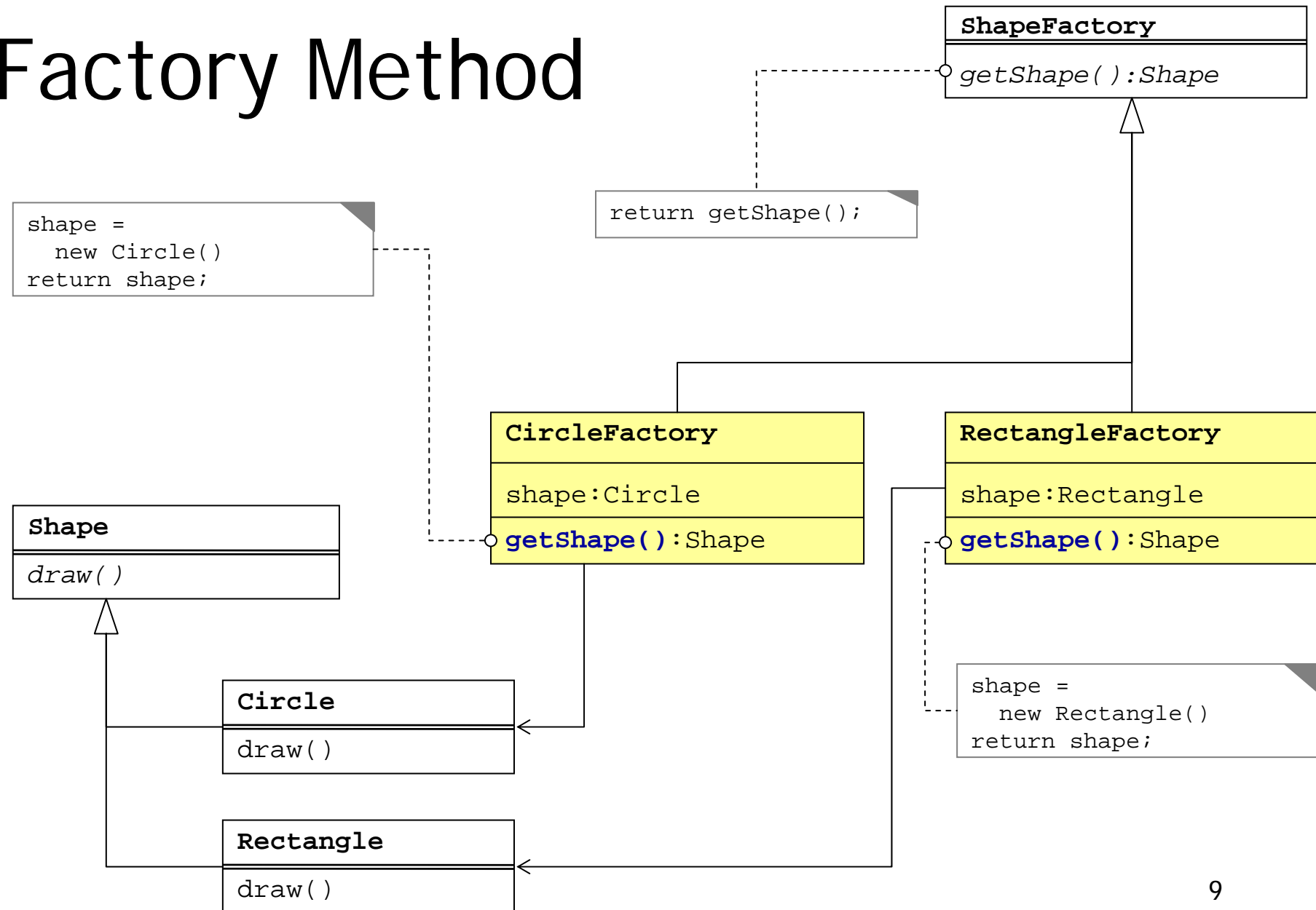
*"Definir uma **interface para criar um objeto** mas deixar que subclasses decidam que classe instanciar. Factory Method permite que uma classe delegue a responsabilidade de instanciamento às subclasses." [GoF]*

*O acesso a um objeto concreto será através da interface conhecida através de sua superclasse, mas cliente também não quer (ou não pode) saber qual implementação concreta está usando*





# Estrutura de Factory Method



# 2

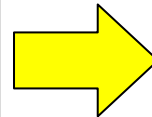
## Strategy

*"Definir uma **família de algoritmos**, encapsular cada um, e fazê-los intercambiáveis. Strategy permite que algoritmos mudem independentemente entre clientes que os utilizam."*  
*[GoF]*

# Problema

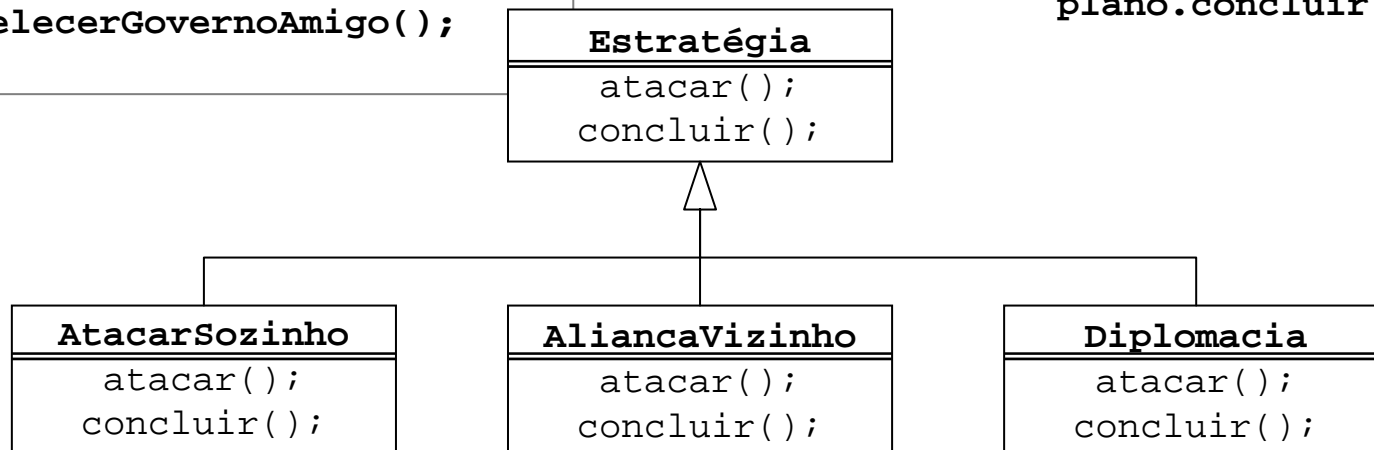
*Várias estratégias, escolhidas de acordo com opções ou condições*

```
if (inimigo.exercito() > 10000) {  
    fazerAlianca();  
    vizinhoAtacaPeloNorte();  
    nosAtacamosPeloSul();  
    dividirBeneficios(...);  
    dividirReconstrução(...);  
}  
else if (inimigo.isNuclear()) {  
    recuarTropas();  
    proporCooperacaoEconomica();  
    desarmarInimigo();  
}  
else if (inimigo.hasNoChance()) {  
    plantarEvidenciasFalsas();  
    soltarBombas();  
    derrubarGoverno();  
    estabelecerGovernoAmigo();  
}
```



```
if (inimigo.exercito() > 10000) {  
    plano = new AliancaVizinho();  
}  
else if (inimigo.isNuclear()) {  
    plano = new Diplomacia();  
}  
else if (inimigo.hasNoChance())  
    plano = new AtacarSozinho();  
}
```

```
plano.atacar();  
plano.concluir();
```



# Em Java

```
public class Guerra {
    Estrategia acao;
    public void definirEstrategia() {
        if (inimigo.exercito() > 10000) {
            acao = new AliancaVizinho();
        } else if (inimigo.isNuclear()) {
            acao = new Diplomacia();
        } else if (inimigo.hasNoChance()) {
            acao = new AtacarSozinho();
        }
    }
    public void declararGuerra() {
        acao.atacar();
    }
    public void encerrarGuerra() {
        acao.concluir();
    }
}
```

```
public interface Estrategia {
    public void atacar();
    public void concluir();
}
```

```
public class AtacarSozinho
    implements Estrategia {
    public void atacar() {
        plantarEvidenciasFalsas();
        soltarBombas();
        derrubarGoverno();
    }
    public void concluir() {
        estabelecerGovernoAmigo();
    }
}
```

```
public class AliancaVizinho
    implements Estrategia {
    public void atacar() {
        vizinhoAtacaPeloNorte();
        nosAtacamosPeloSul();
        ...
    }
    public void concluir() {
        dividirBeneficios(...);
        dividirReconstrução(...);
    }
}
```

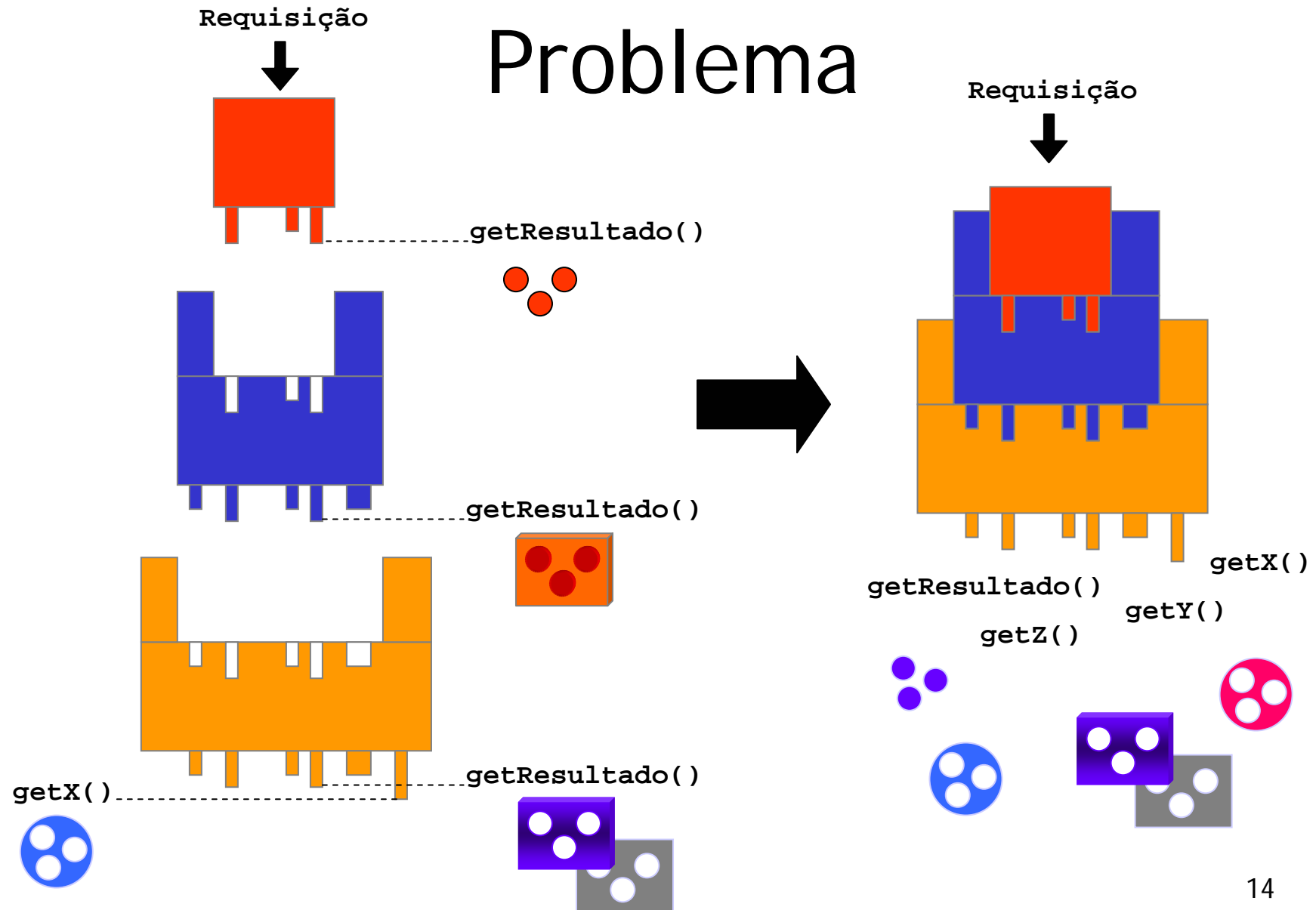
```
public class Diplomacia
    implements Estrategia {
    public void atacar() {
        recuarTropas();
        proporCooperacaoEconomica();
        ...
    }
    public void concluir() {
        desarmarInimigo();
    }
}
```

# 3

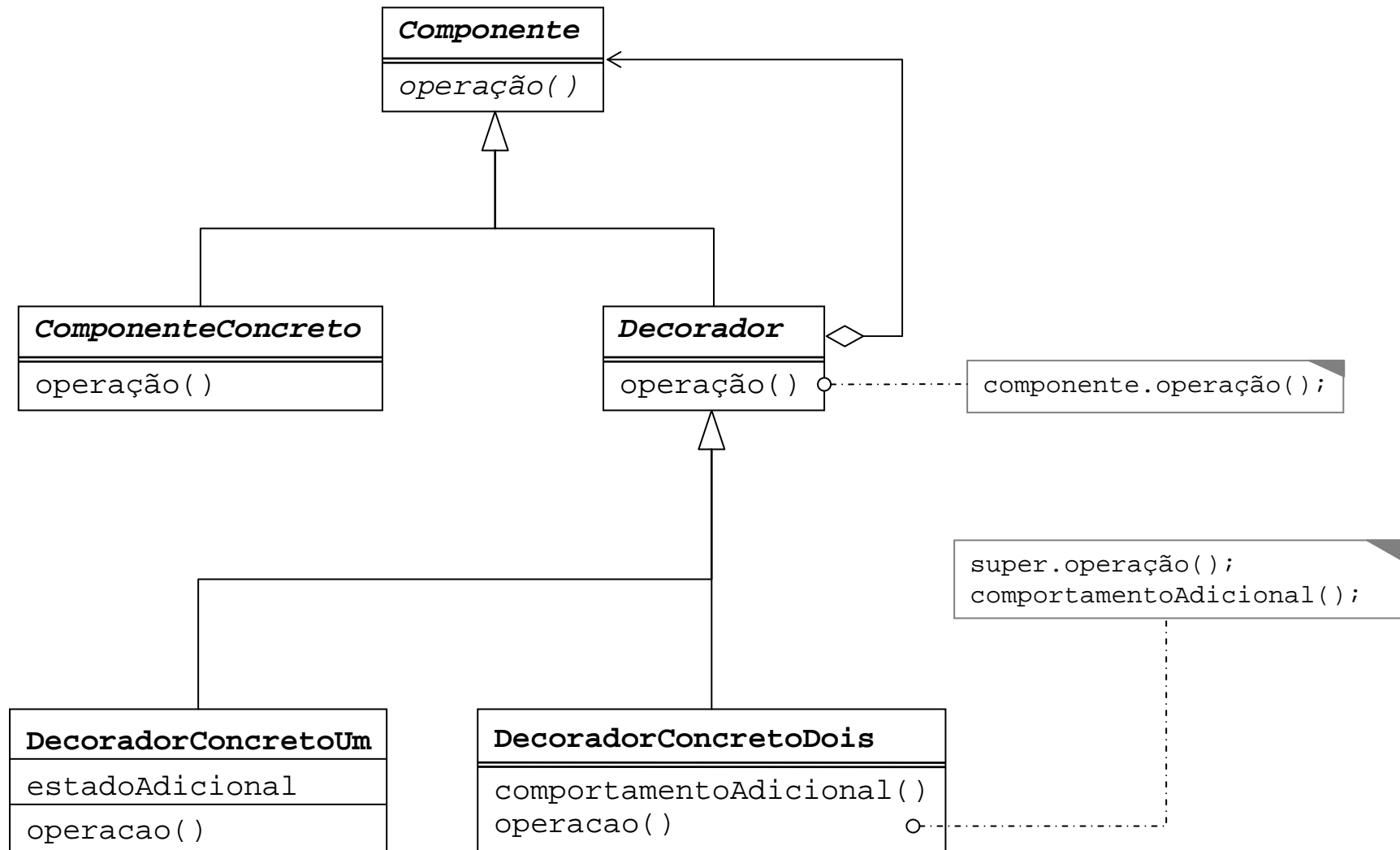
## Decorator

*"Anexar **responsabilidades adicionais** a um objeto dinamicamente. Decorators oferecem uma alternativa flexível ao uso de herança para estender uma funcionalidade." [GoF]*

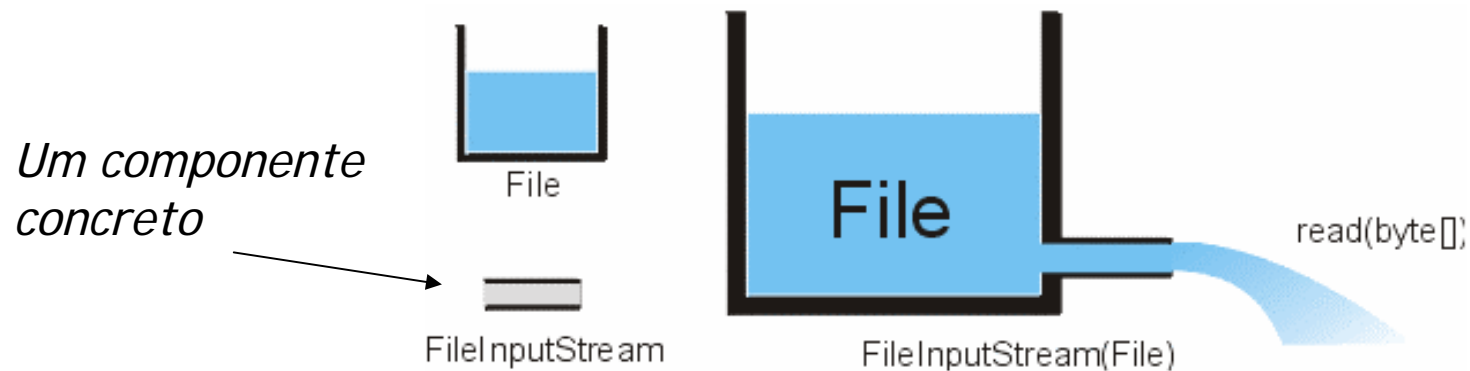
# Problema



# Estrutura de Decorator



# Exemplo: I/O Streams

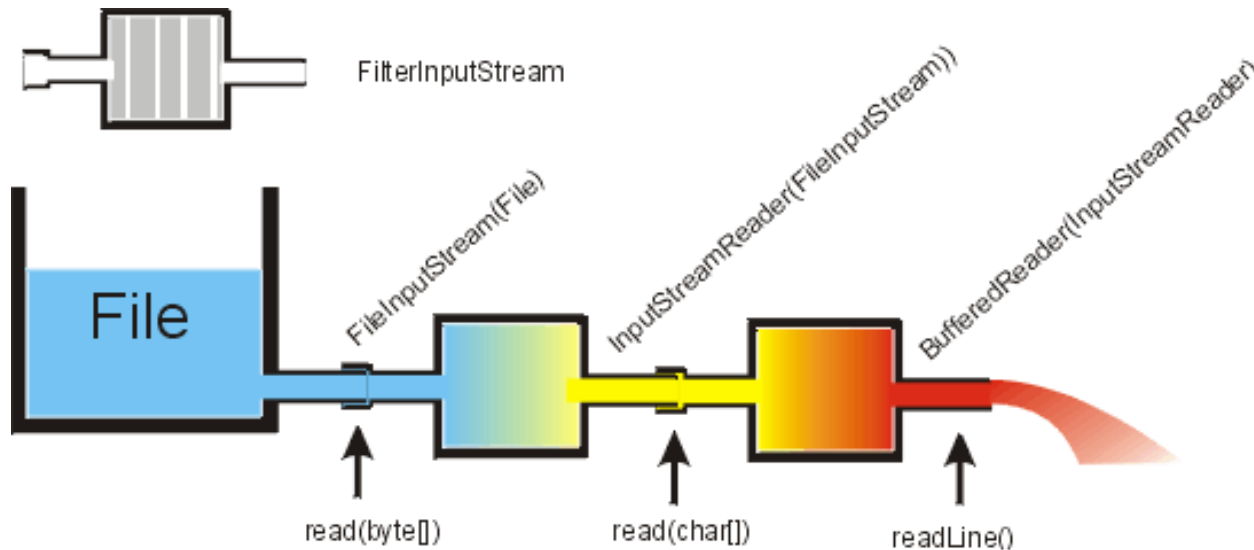


```
// objeto do tipo File
File tanque = new File("agua.txt");

// componente FileInputStream
// cano conectado no tanque
FileInputStream cano =
    new FileInputStream(tanque);

// read() lê um byte a partir do cano
byte octeto = cano.read();
```





# Concatenação de I/O streams

```
// partindo do cano (componente concreto)
FileInputStream cano = new FileInputStream(tanque);

// decorador chf conectado no componente
InputStreamReader chf = new InputStreamReader(cano);
```

```
// pode-se ler um char a partir de chf (mas isto impede que
// o char chegue ao fim da linha: há um vazamento no cano!)
char letra = chf.read();
```

```
// decorador br conectado no decorador chf
BufferedReader br = new BufferedReader (chf);
// lê linha de texto a de br
String linha = br.readLine();
```

Concatenação do decorador

Uso de método com comportamento alterado

Comportamento adicional

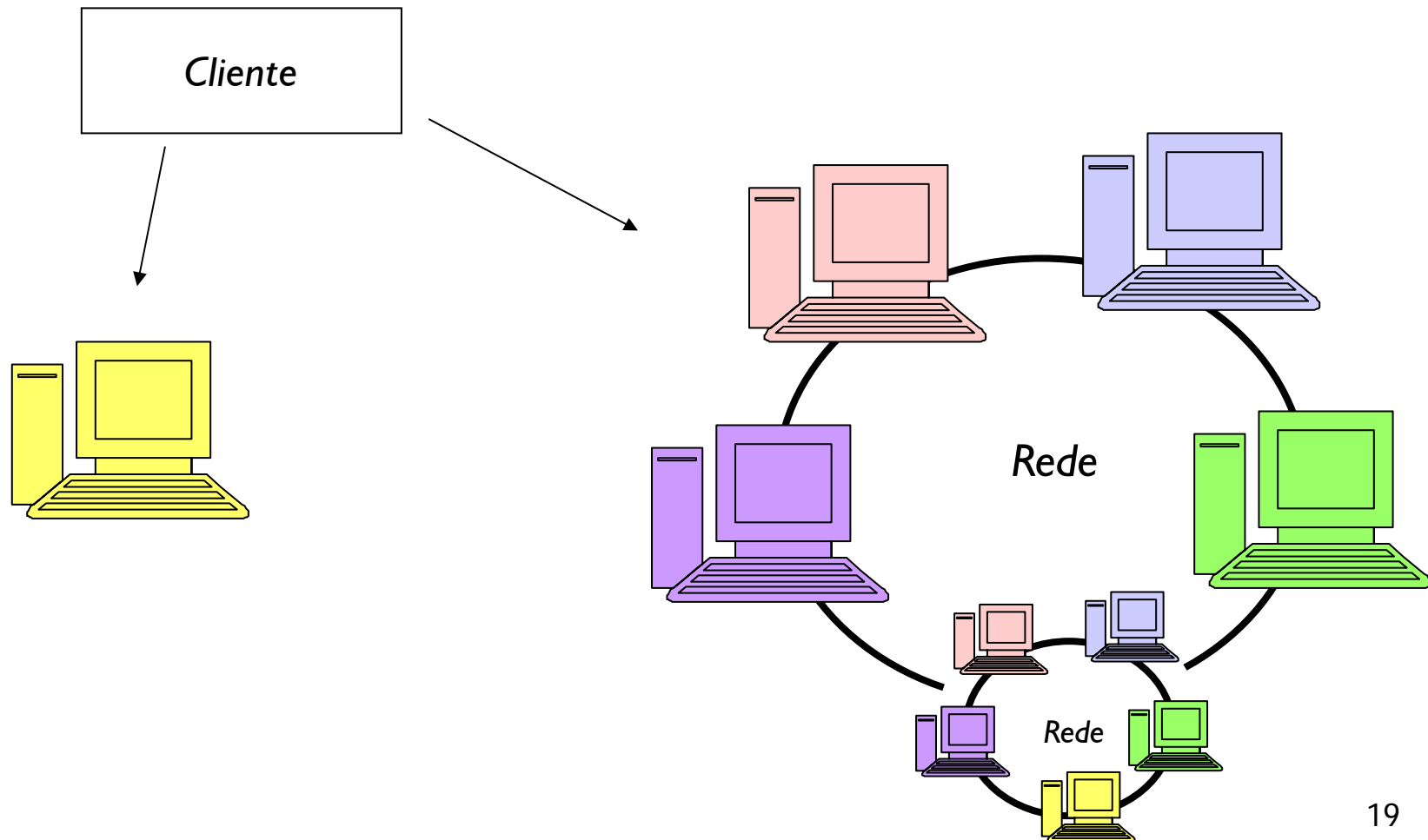
# 4

## Composite

*"Compor objetos em estruturas de árvore para representar **hierarquias todo-parte**. Composite permite que clientes tratem objetos individuais e composições de objetos de maneira uniforme." [GoF]*

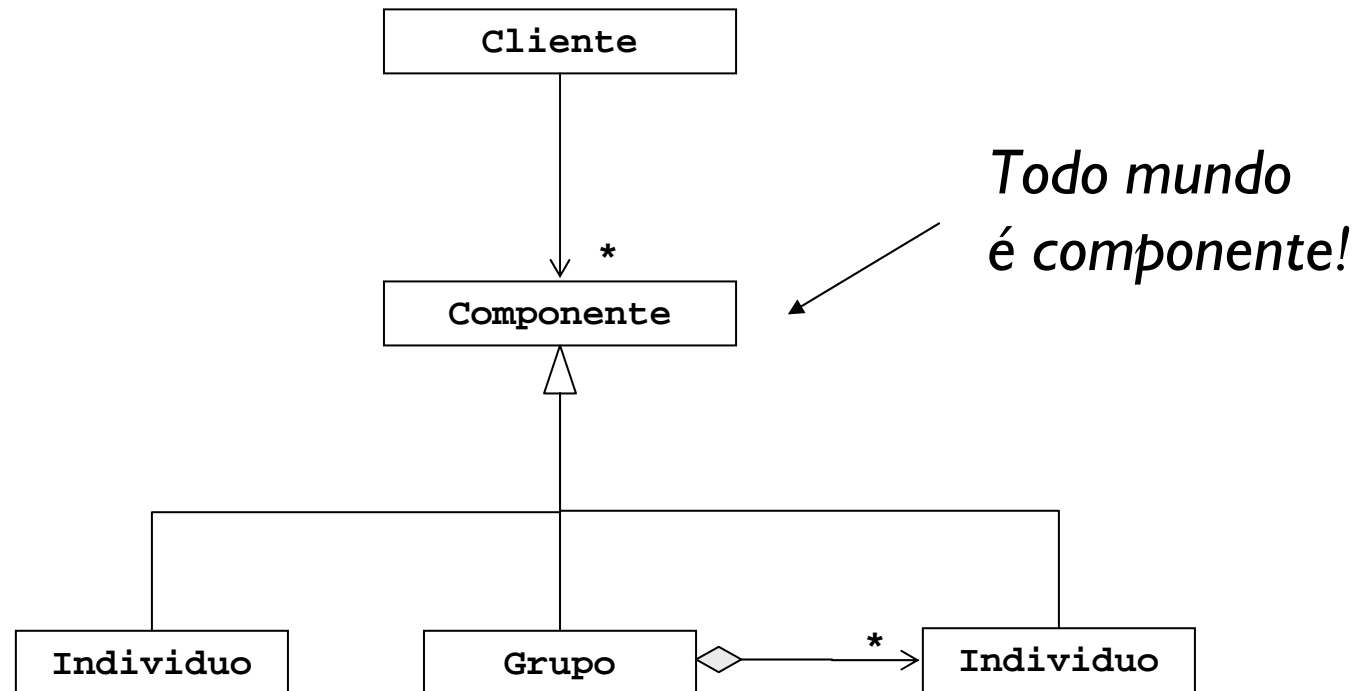
# Problema

*Cliente precisa tratar de maneira uniforme  
objetos individuais e composições desses objetos*

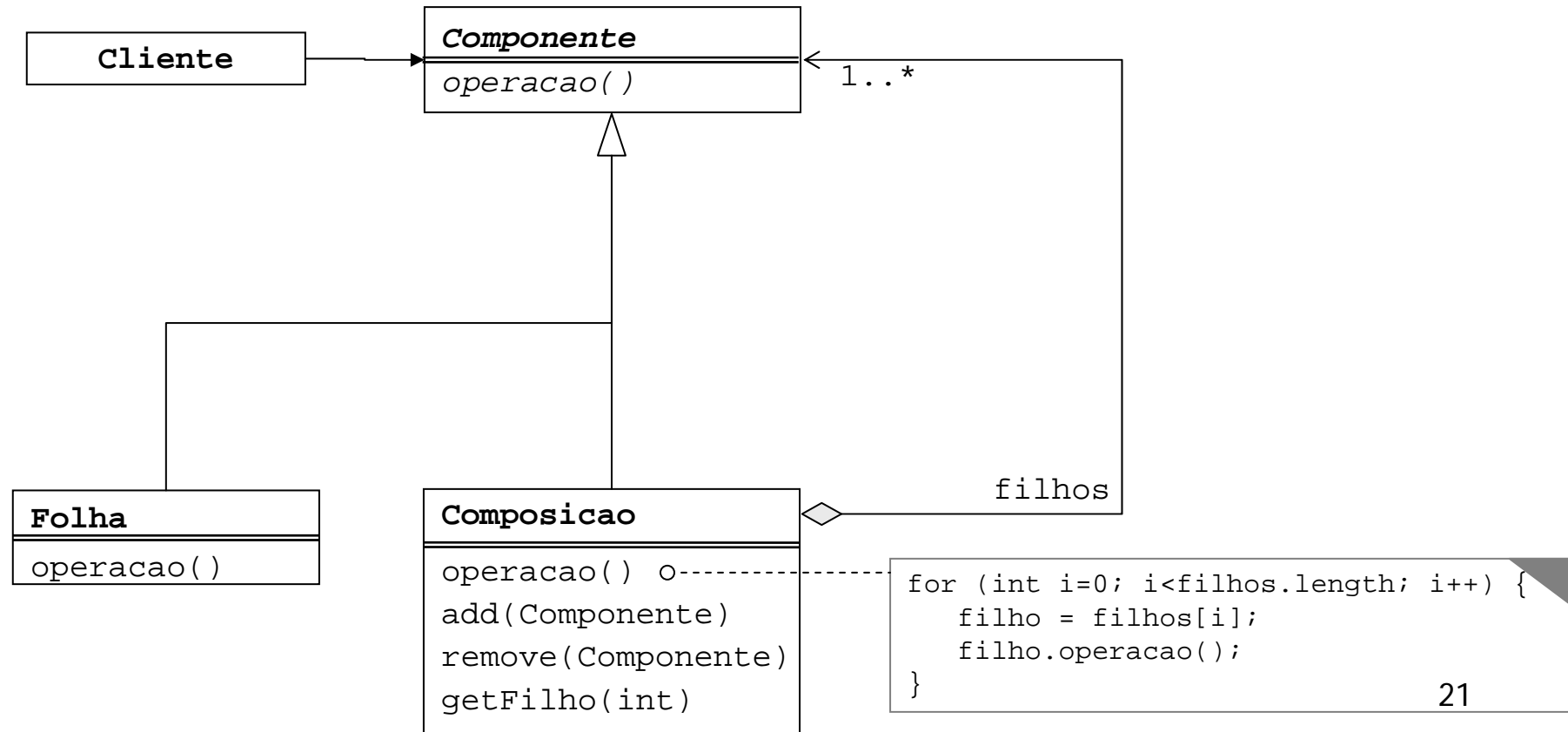
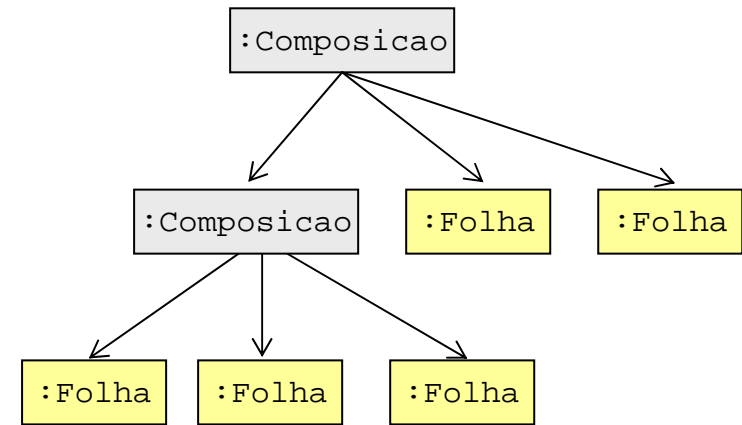


*Tratar grupos e indivíduos  
diferentes através de uma  
única interface*

# Solução



# Estrutura

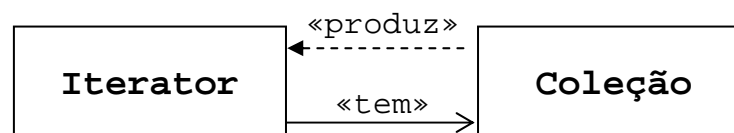
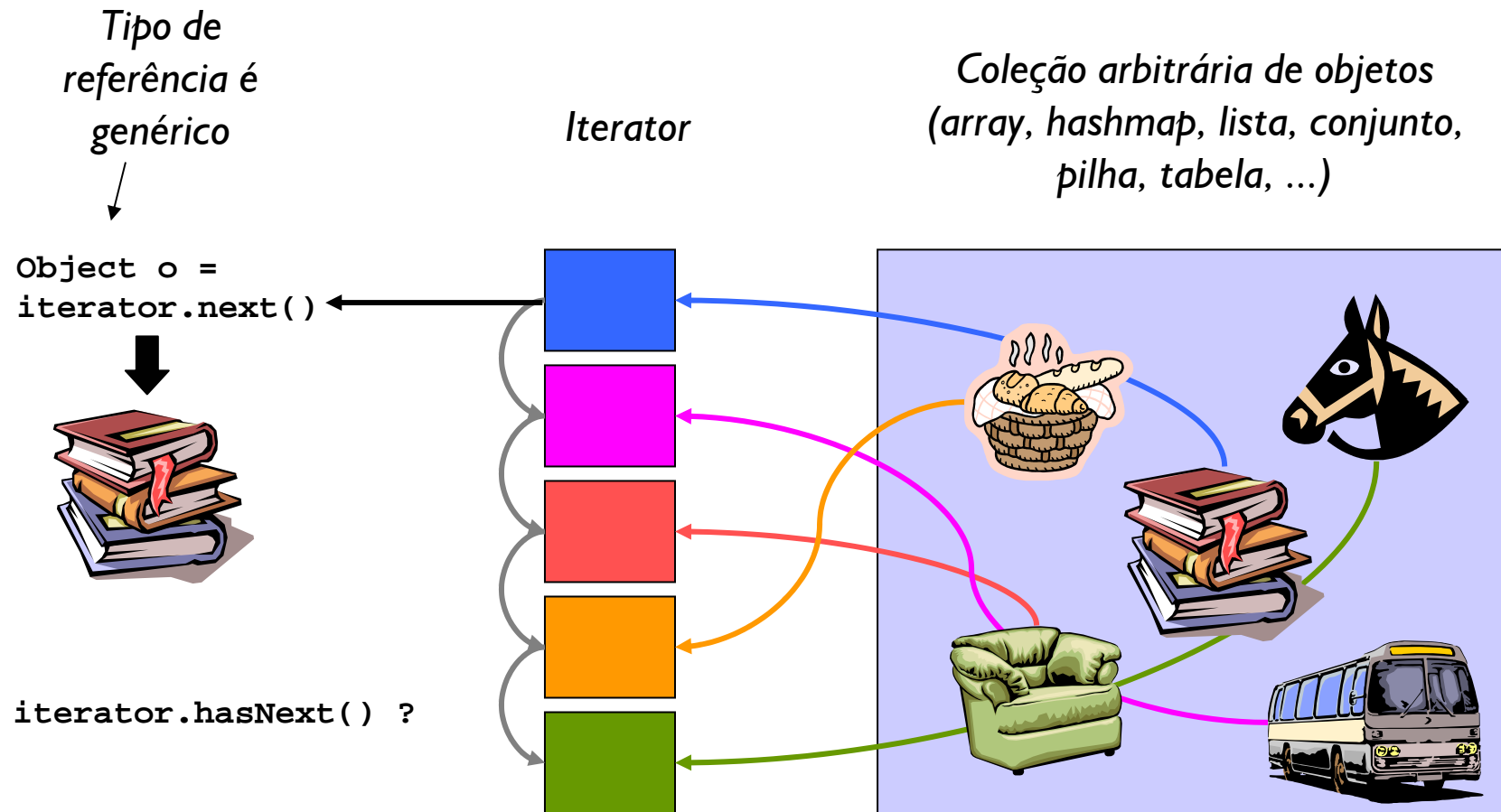


# 5

## Iterator

*"Prover uma maneira de **acessar seqüencialmente os elementos de um objeto agregado** sem expor sua representação interna." [GoF]*

# Iterator



# Iterators em Java

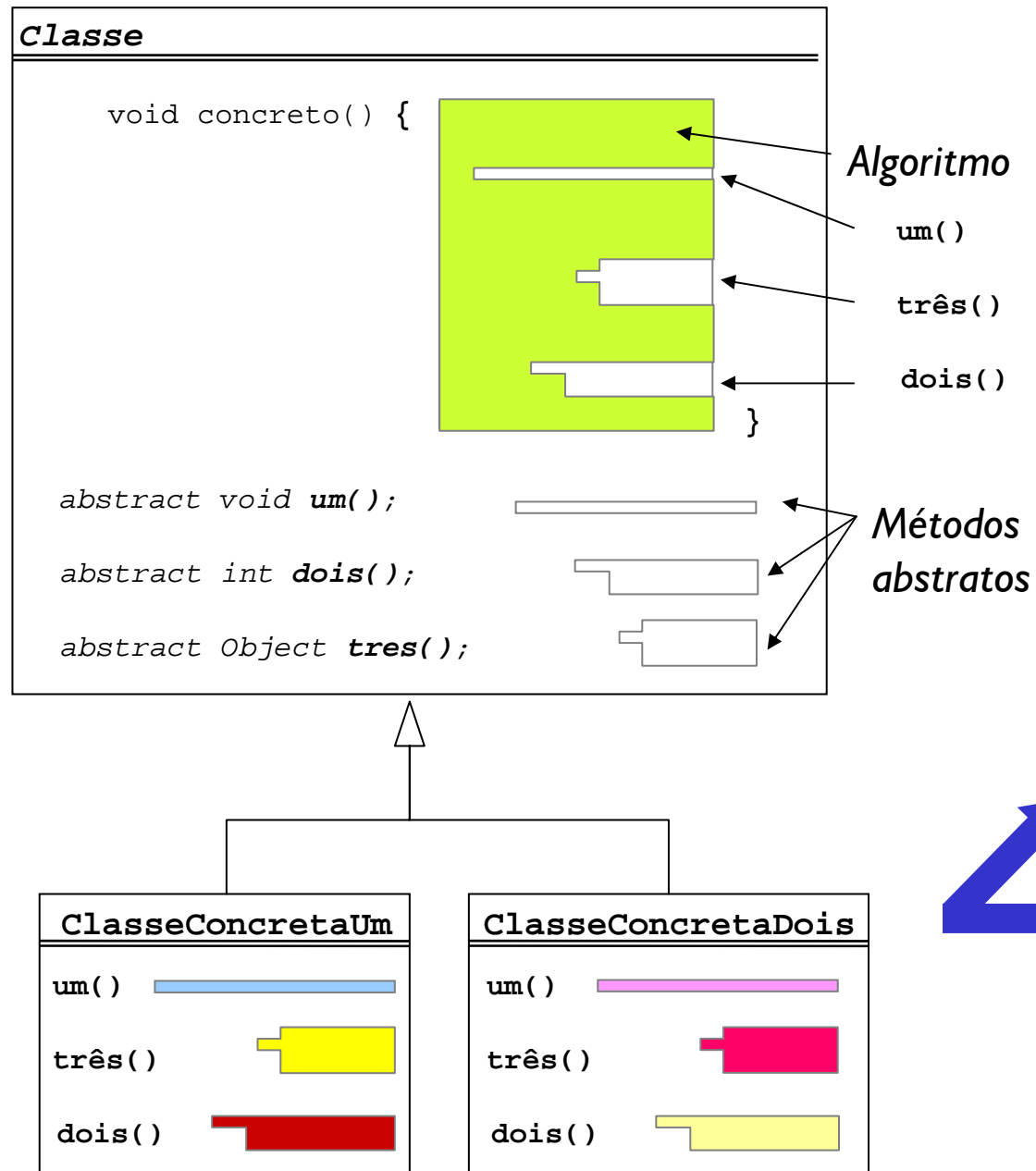
- `java.util.Iterator`
- `java.sql.ResultSet`
- `java.util.Enumeration`
- ...



# Template Method

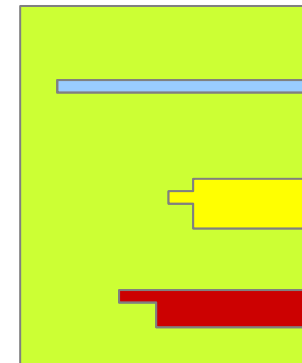
*"Definir o **esqueleto de um algoritmo** dentro de uma operação, deixando alguns passos a serem preenchidos pelas subclasses. Template Method permite que suas subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura." [GoF]*

# Problema

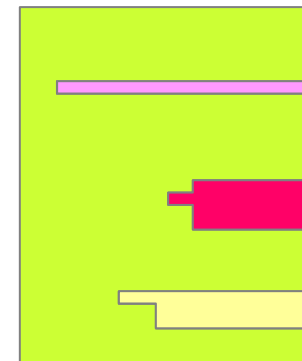


## Algoritmos resultantes

Classe x =  
new ClasseConcretaUm()  
x.concreto()



Classe x =  
new ClasseConcretaDois()  
x.concreto()



# Template Method em Java

```
public abstract class Template {  
    protected abstract String link(String texto, String url);  
    protected String transform(String texto) { return texto; }  
    public final String templateMethod() {  
        String msg = "Endereço: " + link("Empresa", "http://www.empresa.com");  
        return transform(msg);  
    }  
}
```

```
public class XMLData extends Template {  
    protected String link(String texto, String url) {  
        return "<endereco xlink:href='"+url+"'>" + texto + "</endereco>";  
    }  
}
```

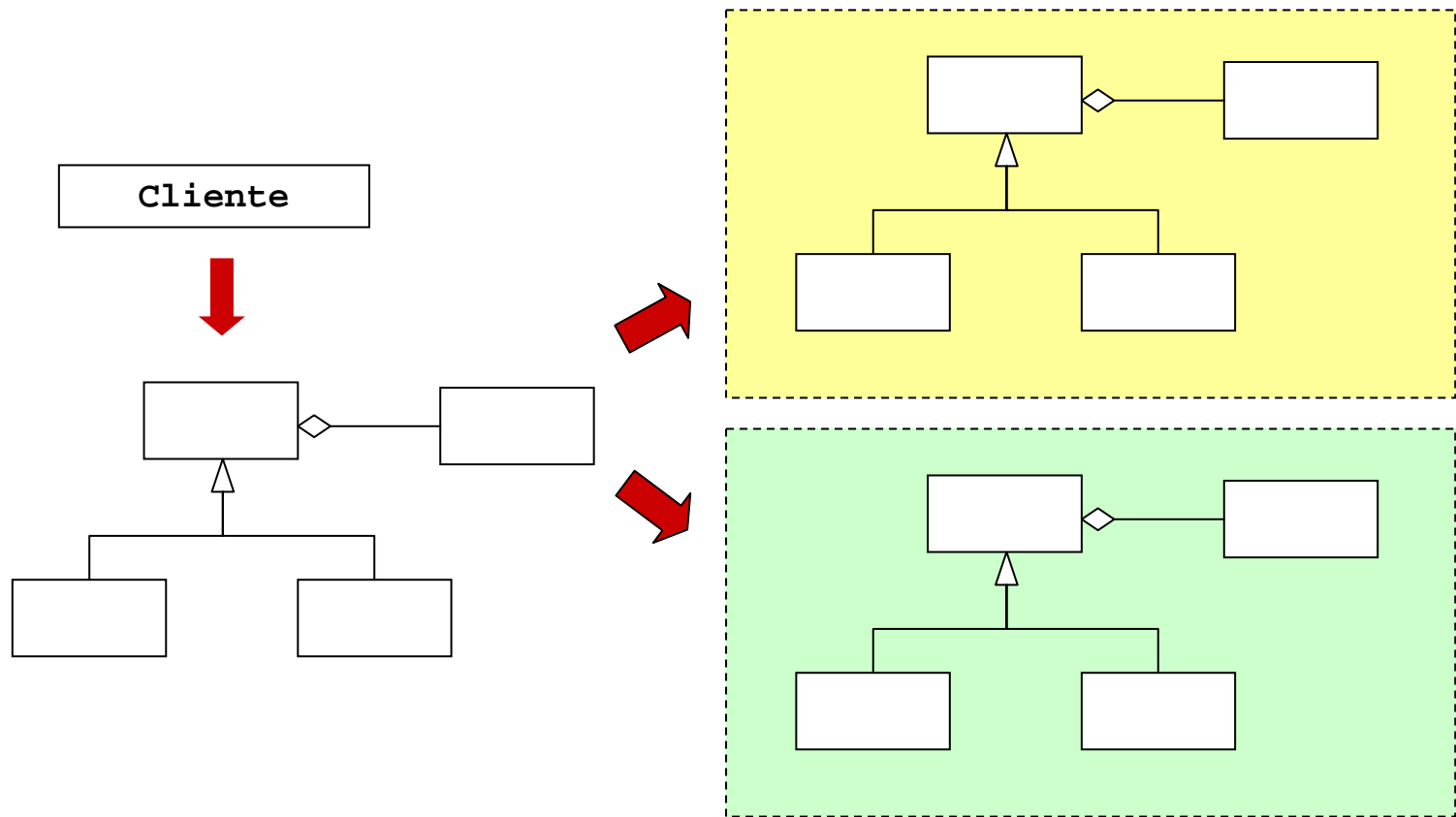
```
public class HTMLData extends Template {  
    protected String link(String texto, String url) {  
        return "<a href='"+url+"'>" + texto + "</a>";  
    }  
    protected String transform(String texto) {  
        return texto.toLowerCase();  
    }  
}
```

# Abstract Factory

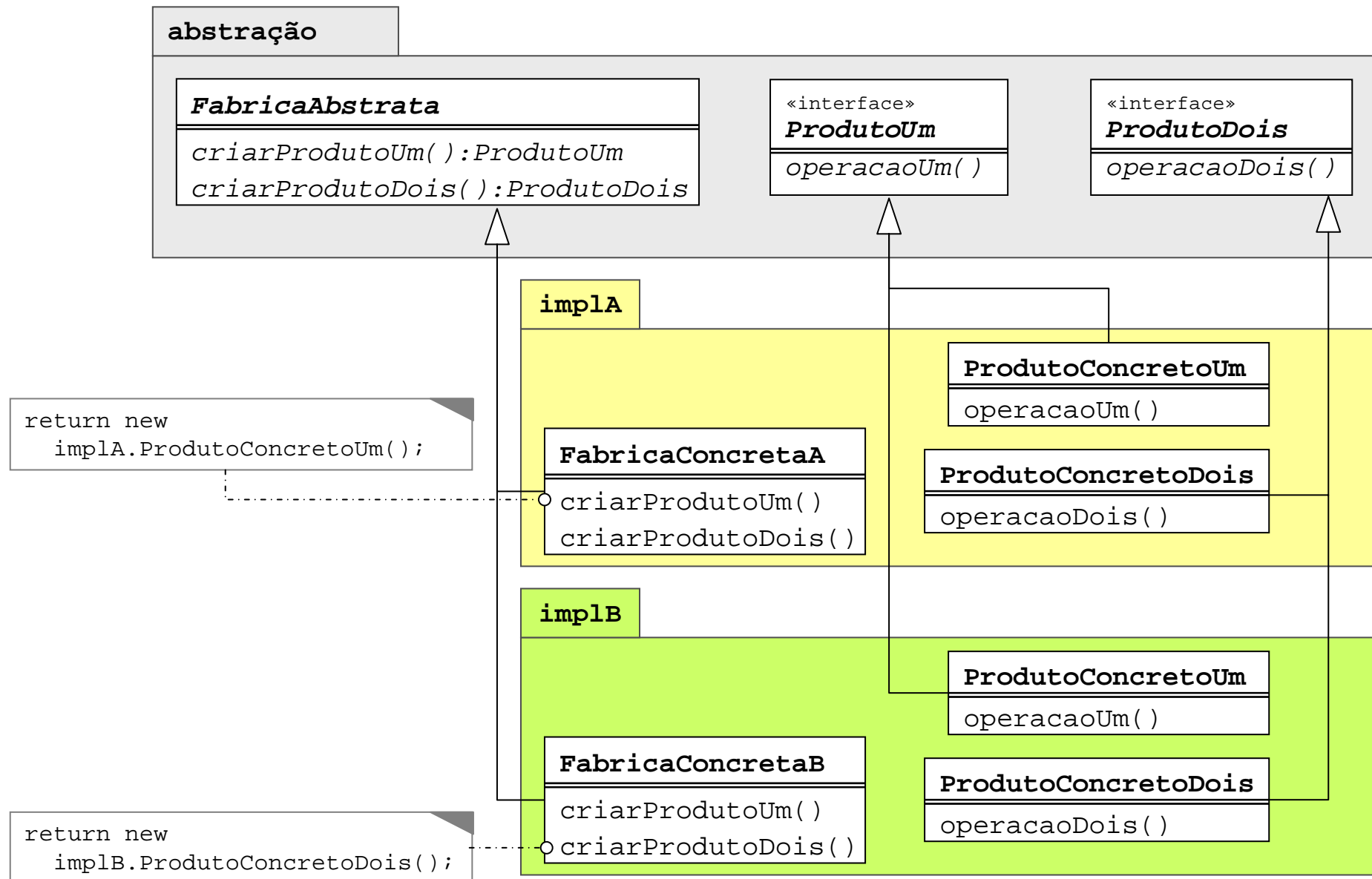
*"Prover uma **interface para criar famílias de objetos relacionados** ou dependentes sem especificar suas classes concretas." [GoF]*

# Problema

- Criar uma família de objetos relacionados sem conhecer suas classes concretas



# Abstract Factory



# 8

## Builder

*"Separar a construção de um objeto complexo de sua representação para que o mesmo processo de construção possa criar representações diferentes." [GoF]*

# Problema

Cliente

*Cliente precisa de uma casa. Passa as informações necessárias para seu diretor*

Diretor

*Utilizando as informações passadas pelo cliente, ordena a criação da casa pelo construtor usando uma interface uniforme*

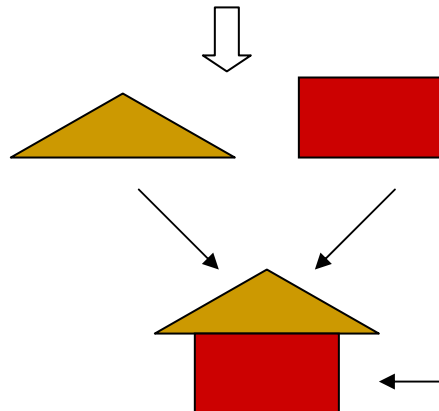
Construtor

passoUm()  
passoDois()  
obterProduto()

*O construtor é habilitado para construir qualquer objeto complexo (poderia, por exemplo, construir um prédio em vez de uma casa, caso o cliente tivesse indicado esse desejo)*



*O Diretor selecionou um construtor de casas e chamou os passos necessários da construção*



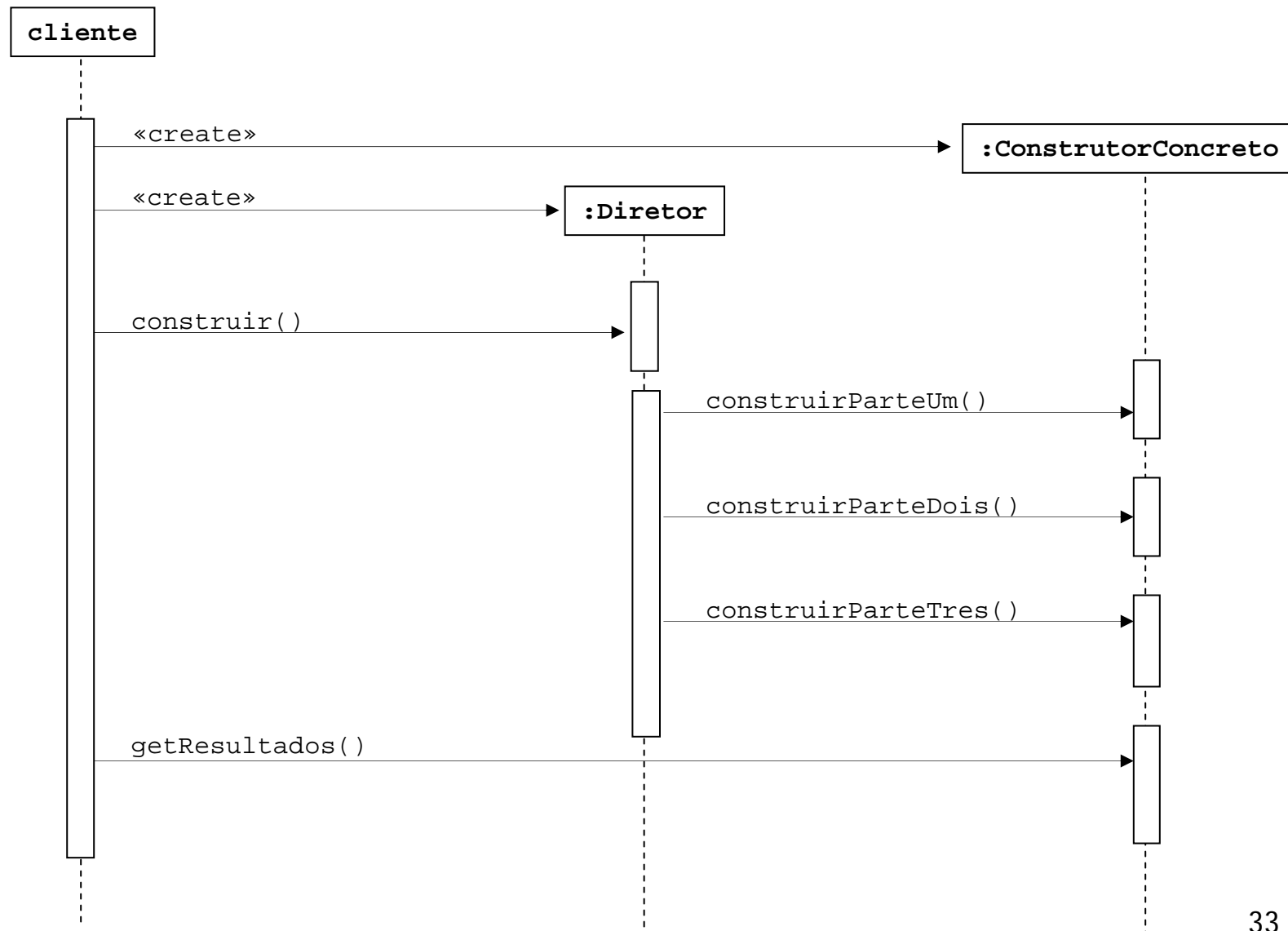
*Quando o produto estiver pronto, o cliente pode buscar seu produto diretamente do construtor.*

obterProduto()

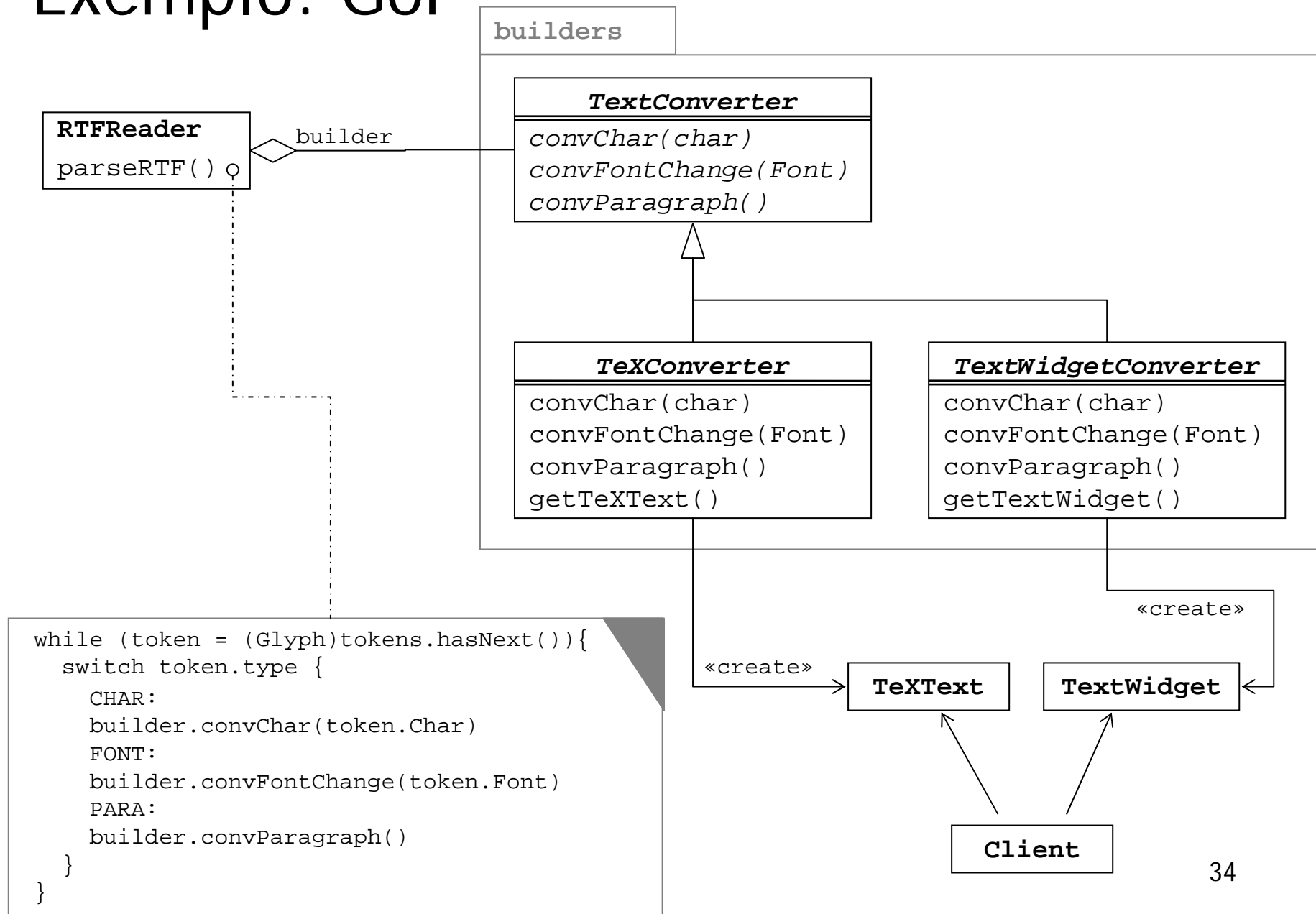
Cliente



# Seqüência de Builder



# Exemplo: GoF



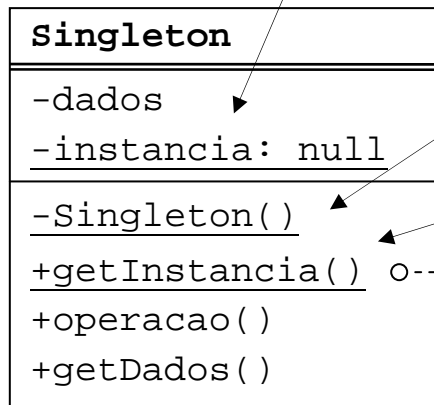
# 9

## Singleton

*"Garantir que uma classe só tenha **uma única instância**, e prover um ponto de acesso global a ela." [GoF]*

# Estrutura de Singleton

*Objeto com acesso privativo*



*Construtor privativo (nem subclasses têm acesso)*

*Ponto de acesso simples, estático e global*

```
public static Singleton getInstancia() {  
    if (instancia == null) {  
        instancia = new Singleton();  
    }  
    return instancia;  
}
```

*Lazy initialization idiom*

*Bloco deve ser **synchronized** para evitar que dois objetos tentem criar o objeto ao mesmo tempo*

# Singleton em Java

```
public class Highlander {  
    private Highlander() {}  
    private static Highlander instancia = new Highlander();  
    public static synchronized Highlander obterInstancia() {  
        return instancia;  
    }  
}
```

*Esta classe  
implementa o  
design pattern  
Singleton*

```
public class Fabrica {  
    public static void main(String[] args) {  
        Highlander h1, h2, h3;  
        //h1 = new Highlander(); // nao compila!  
        h2 = Highlander.obterInstancia();  
        h3 = Highlander.obterInstancia();  
        if (h2 == h3) {  
            System.out.println("h2 e h3 são mesmo objeto!");  
        }  
    }  
}
```

*Esta classe  
cria apenas  
um objeto  
Highlander*

# Implementações

- Eager instantiation

- Melhor alternativa (deixar otimizações para depois)

```
private static final Resource resource = new Resource();  
public static Resource getResource() {  
    return resource;  
}
```



- Instanciamento lazy corretamente sincronizado

- Há custo de sincronização em cada chamada

```
private static Resource resource = null;  
public static synchronized Resource getResource() {  
    if (resource == null)  
        resource = new Resource();  
    return resource;  
}
```



- Initialize-on-demand holder class idiom

```
private static class ResourceHolder {  
    static final Resource resource = new Resource();  
}  
public static Resource getResource() {  
    return ResourceHolder.resource;  
}
```



Esta técnica explora a garantia de que uma classe não é inicializada antes que seja usada.

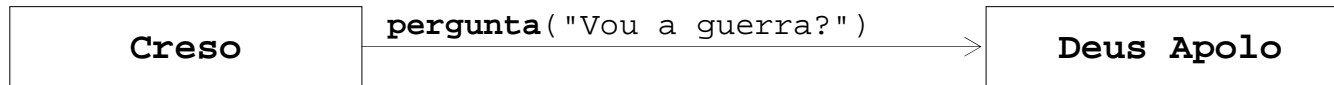
10

# Proxy

*"Prover um **substituto** ou ponto através do qual um objeto possa controlar o acesso a outro." [GoF]*

# Problema

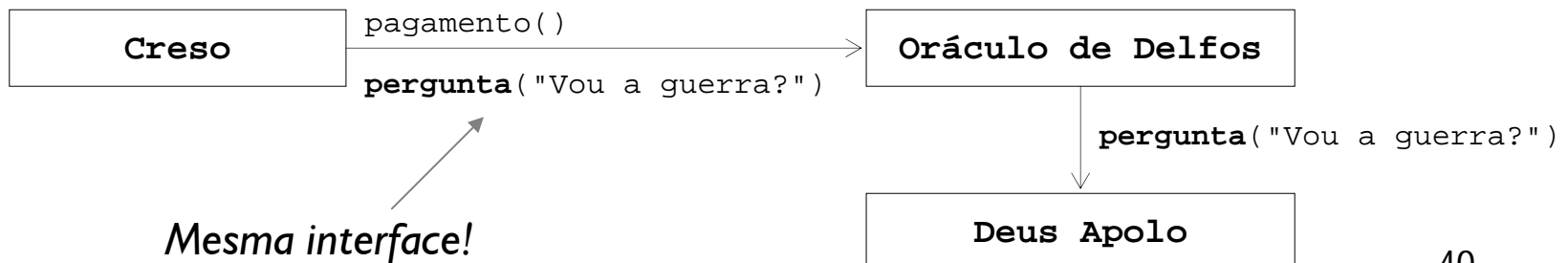
- Sistema quer utilizar objeto real...



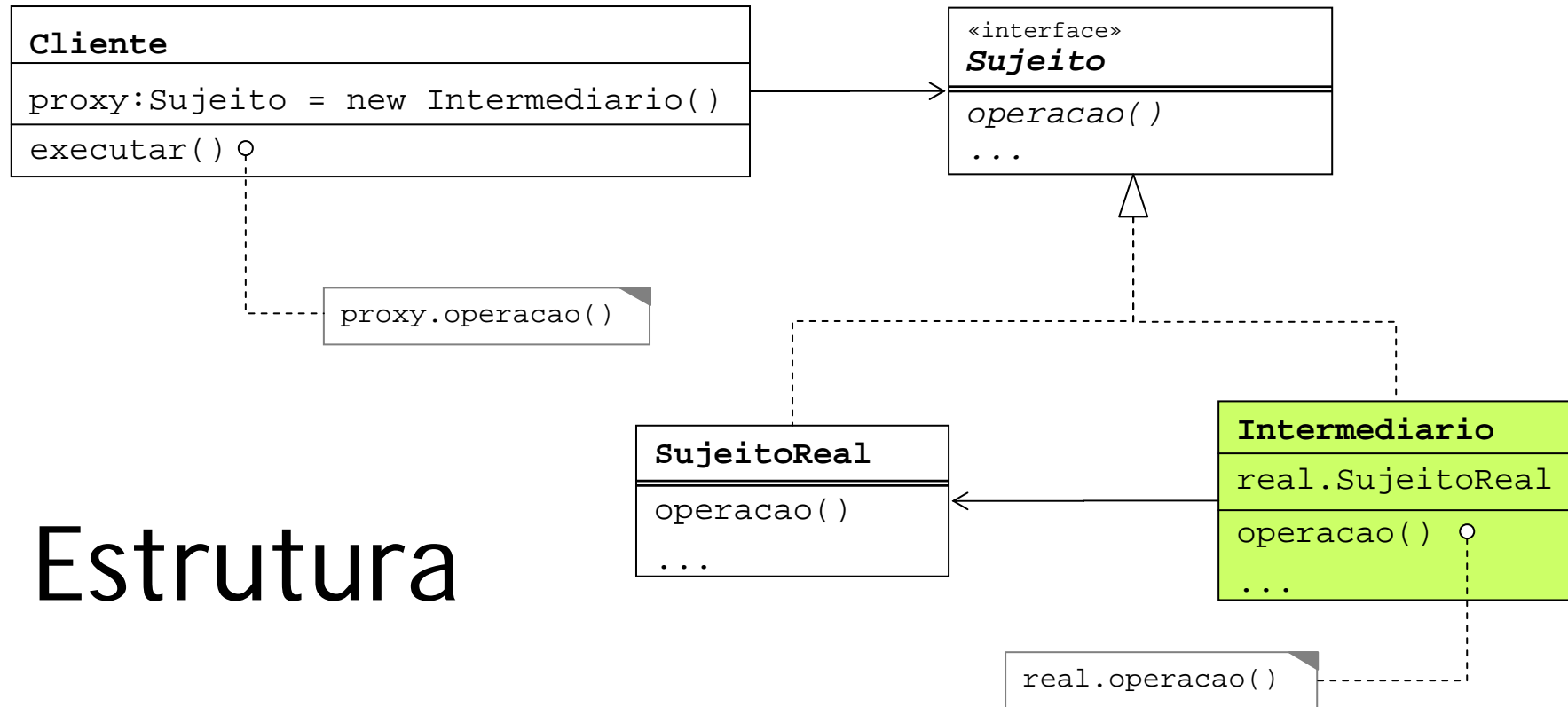
- Mas ele não está disponível (remoto, inacessível, ...)



- Solução: arranjar um intermediário que saiba se comunicar com ele eficientemente








# Estrutura

- Cliente usa **intermediário** em vez de sujeito real
- Intermediário suporta a **mesma interface** que sujeito real
- Intermediário geralmente delega chamadas a sujeito

# Proxy em Java

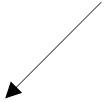
```
public class Creso {  
    ...  
    Sujeito apolo = Fabrica.getSujeito();  
    apolo.operacao();  
    ...  
}
```

*retorna objeto  
que pode ser  
um proxy!*



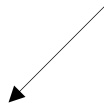
```
public class SujeitoReal implements Sujeito {  
    public Object operacao() {  
        return coisaUtil;  
    }  
}
```

*inacessível  
pelo cliente*



```
public class Intermediario implements Sujeito {  
    private SujeitoReal real;  
    public Object operacao() {  
        cobraTaxa();  
        return real.operacao();  
    }  
}
```

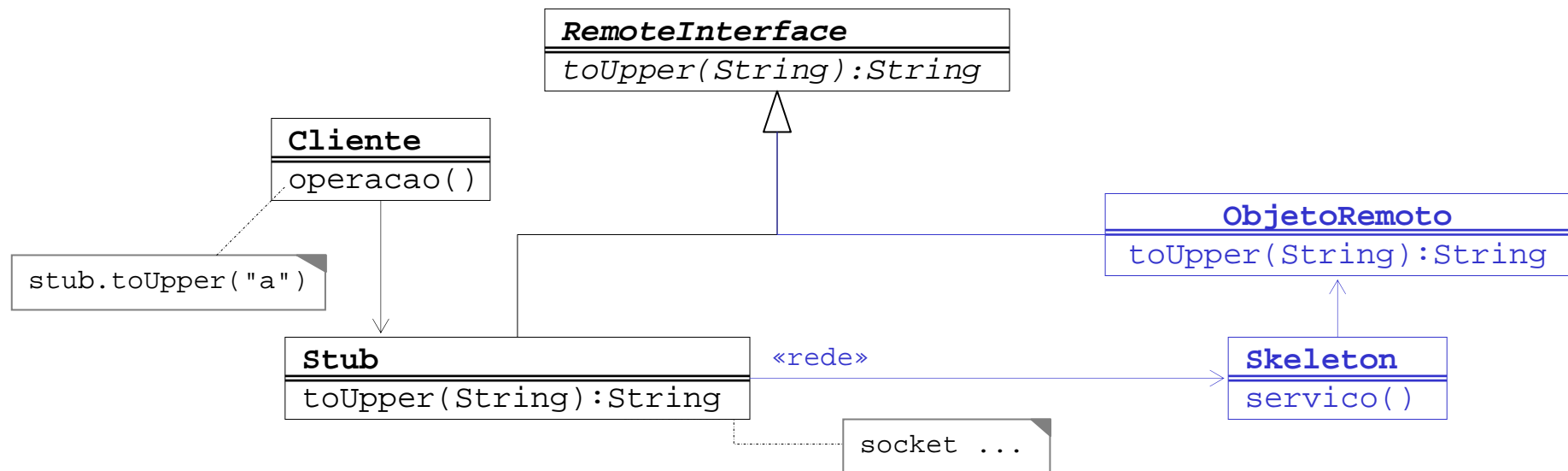
*cliente comunica-se  
com este objeto*



```
public interface Sujeito {  
    public Object operacao();  
}
```

# Exemplo: aplicações distribuídas

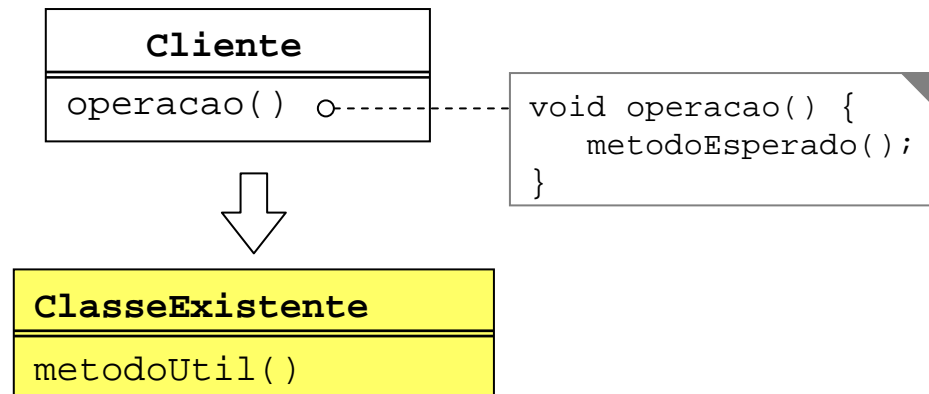
- RMI (e EJB)
  - O Stub é proxy do cliente para o objeto remoto
  - O Skeleton é parte do proxy: cliente remoto chamado pelo Stub



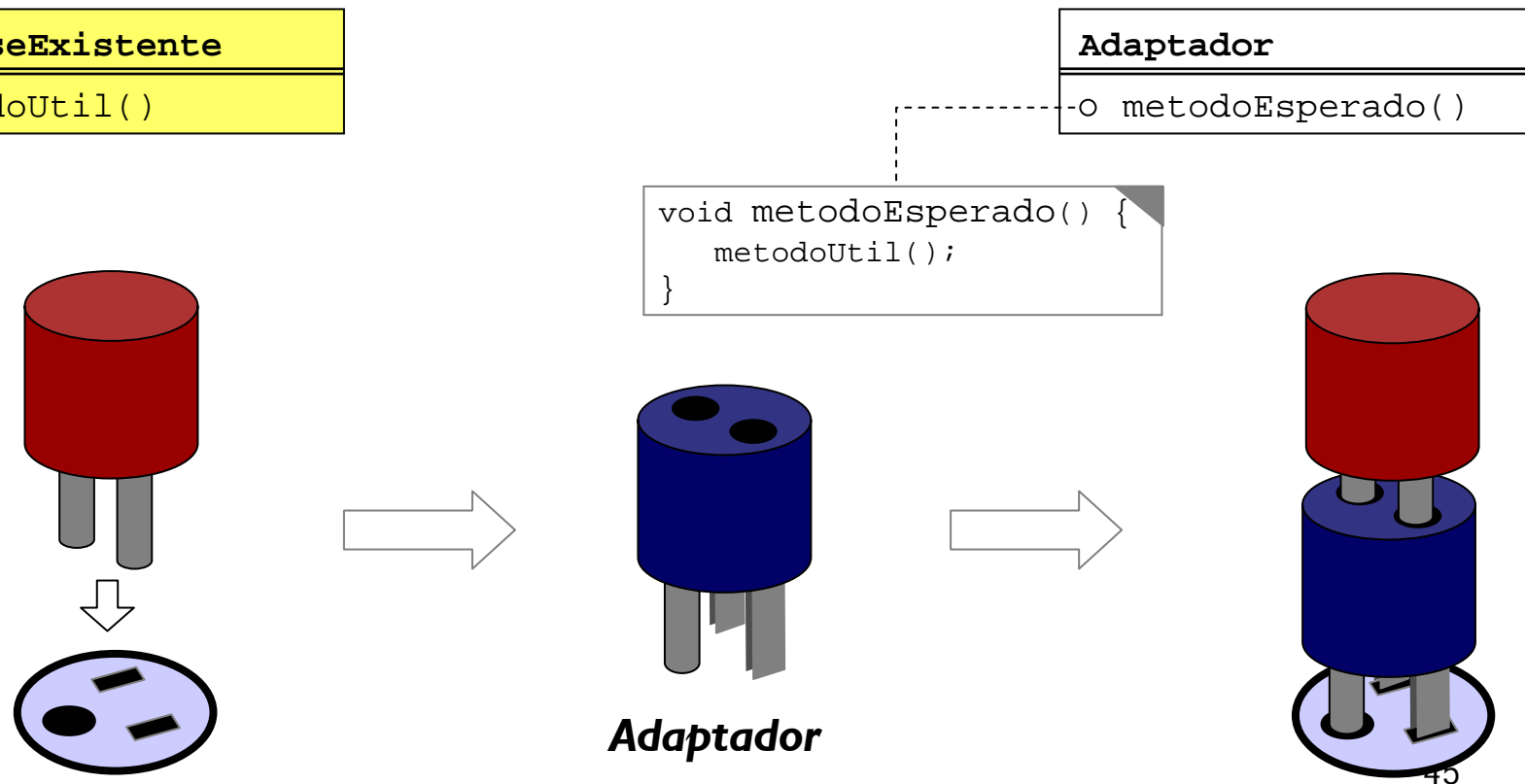
# Adapter

*"Objetivo: **converter a interface** de uma classe em outra interface esperada pelos clientes. Adapter permite a comunicação entre classes que não poderiam trabalhar juntas devido à incompatibilidade de suas interfaces." [GoF]*

## Problema

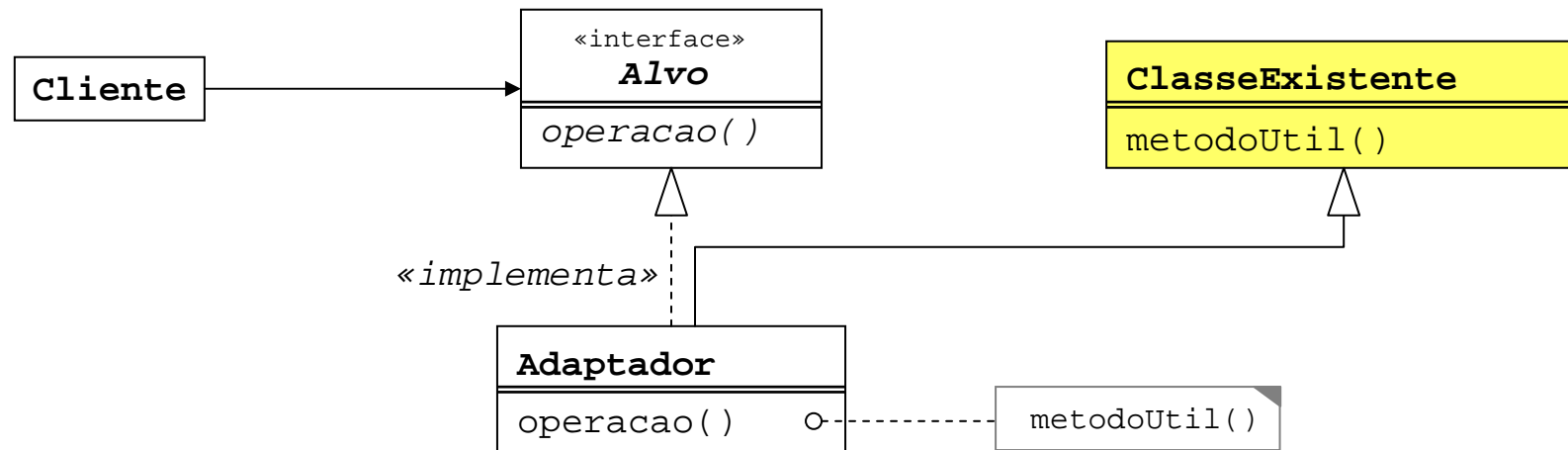


## Solução



# Duas formas de Adapter

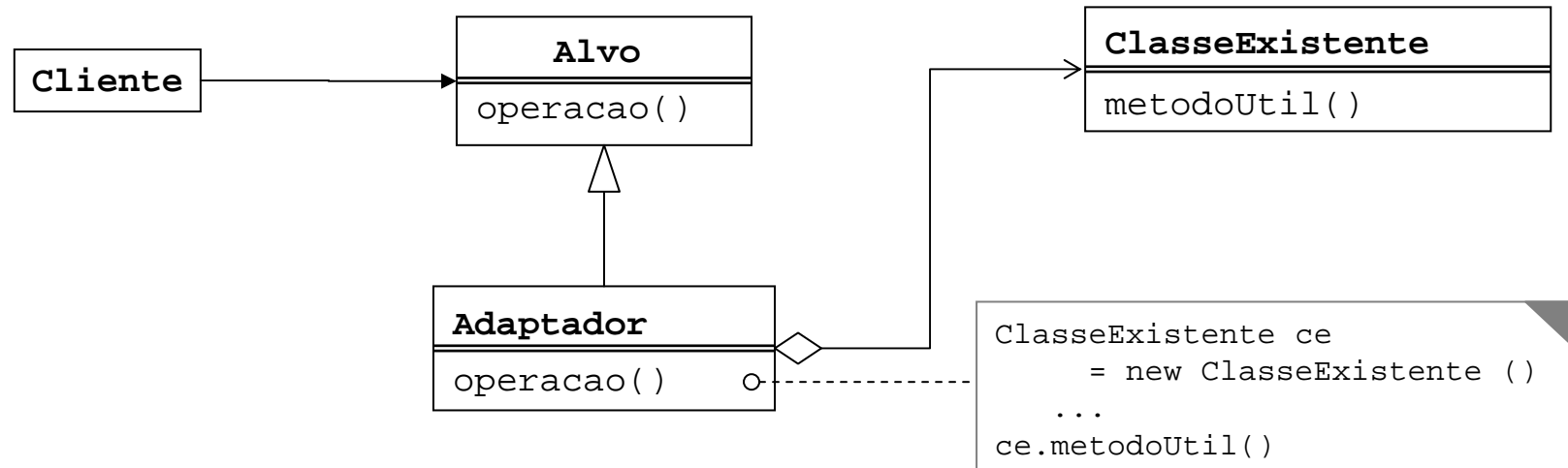
- **Class Adapter**: usa herança múltipla



- **Cliente**: aplicação que colabora com objetos aderentes à interface **Alvo**
- **Alvo**: define a interface requerida pelo **Cliente**
- **ClasseExistente**: interface que requer adaptação
- **Adaptador** (Adapter): adapta a interface do **Recurso** à interface **Alvo**

# Duas formas de Adapter

- **Object Adapter**: usa composição



- *Única solução se Alvo não for uma interface Java*
- *Adaptador possui referência para objeto que terá sua interface adaptada (instância de ClasseExistente).*
- *Cada método de Alvo chama o(s) método(s) correspondente(s) na interface adaptada.*

# Object Adapter em Java

```
public class ClienteExemplo {
    Alvo[] alvos = new Alvo[10];
    public void inicializaAlvos() {
        alvos[0] = new AlvoExistente();
        alvos[1] = new Adaptador();
        // ...
    }
    public void executaAlvos() {
        for (int i = 0; i < alvos.length; i++) {
            alvos[i].operacao();
        }
    }
}
```

```
public abstract class Alvo {
    public abstract void operacao();
    // ... resto da classe
}
```

```
public class Adaptador extends Alvo {
    ClasseExistente existente = new ClasseExistente();
    public void operacao() {
        String texto = existente.metodoUtilDois("Operação Realizada.");
        existente.metodoUtilUm(texto);
    }
}
```

```
public class ClasseExistente {
    public void metodoUtilUm(String texto) {
        System.out.println(texto);
    }
    public String metodoUtilDois(String texto) {
        return texto.toUpperCase();
    }
}
```

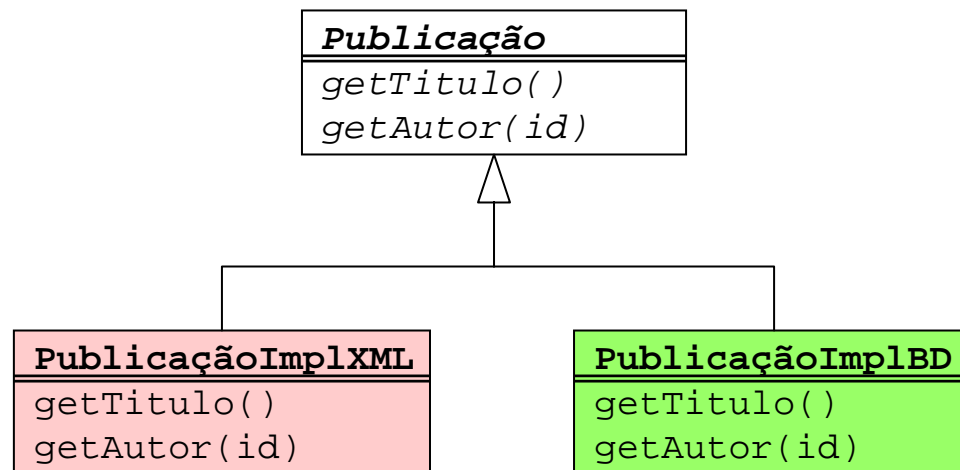


# Bridge

*"Desacoplar uma abstração de sua implementação  
para que os dois possam variar  
independentemente." [GoF]*

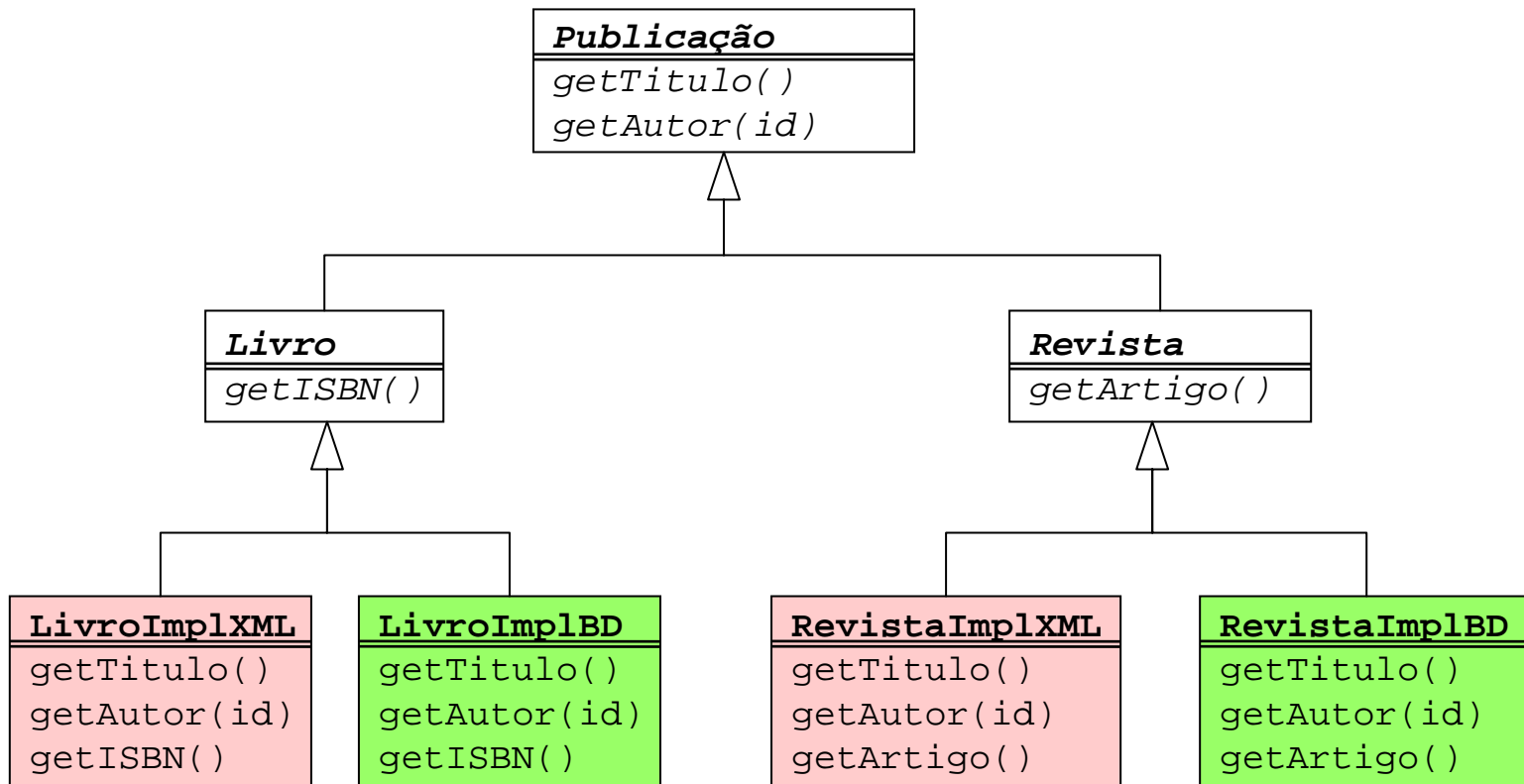
# Problema (1)

- Exemplo: implementações específicas para tratar objeto em diferentes meios persistentes

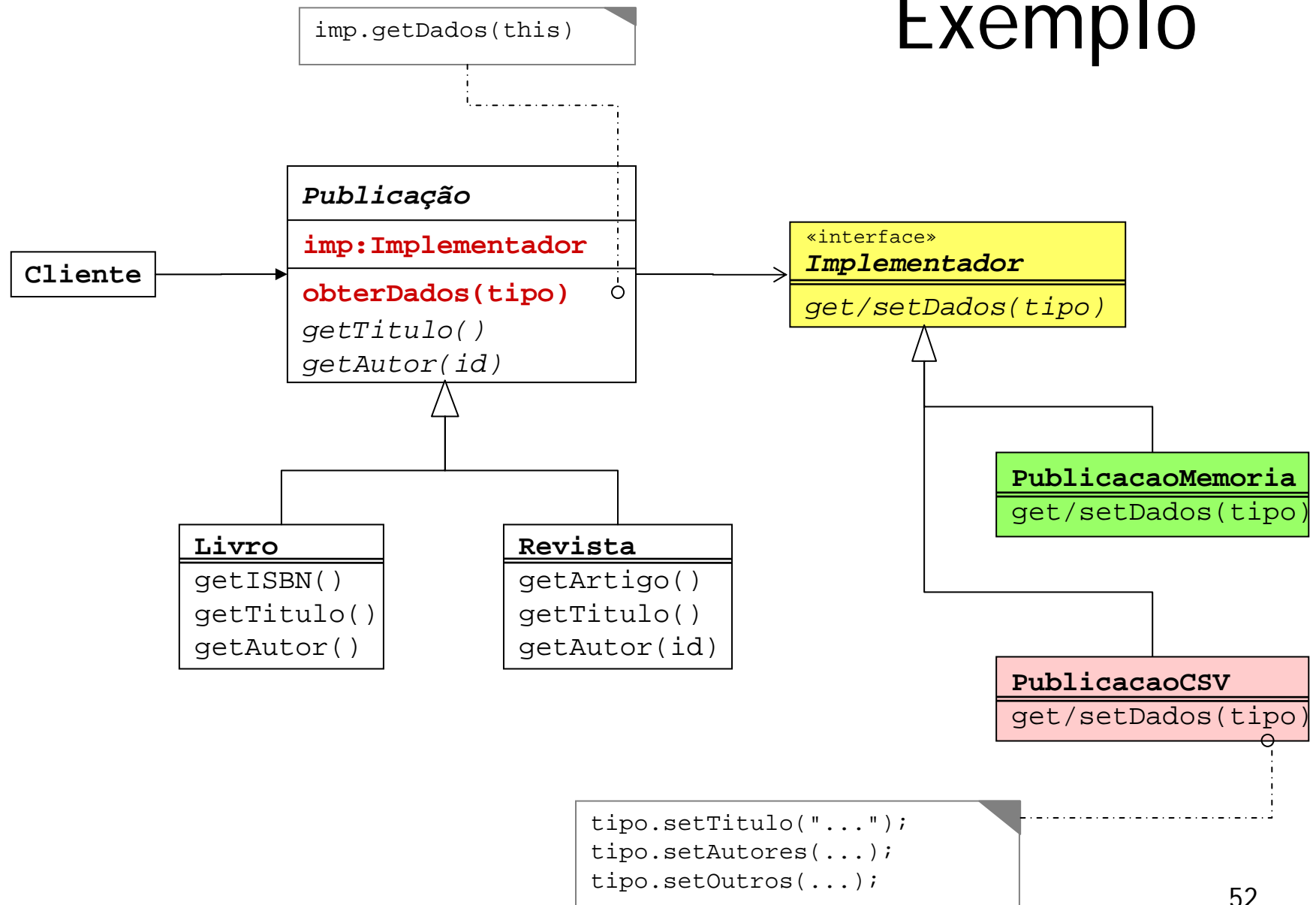


# Problema (II)

- Mas herança complica a implementação



# Exemplo

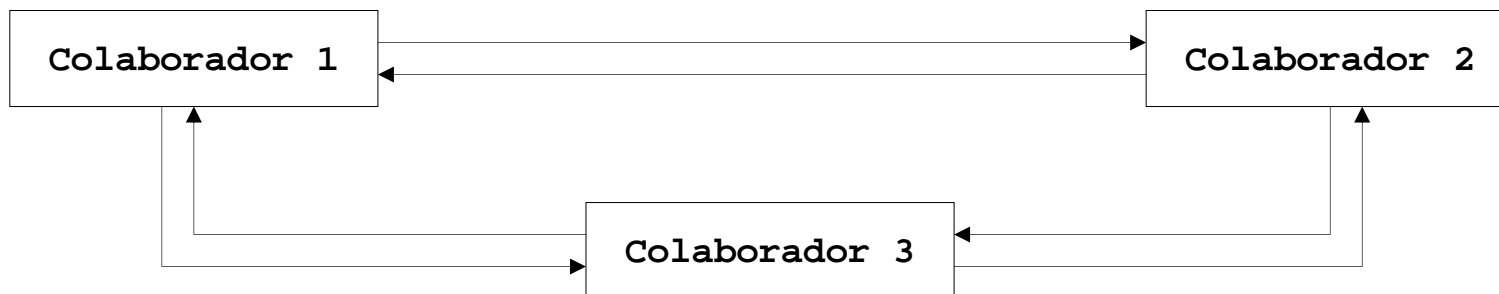


# Mediator

*"Definir um objeto que **encapsula como um conjunto de objetos interagem**. Mediator promove acoplamento fraco ao manter objetos que não se referem um ao outro explicitamente, permitindo variar sua interação independentemente." [GoF]*

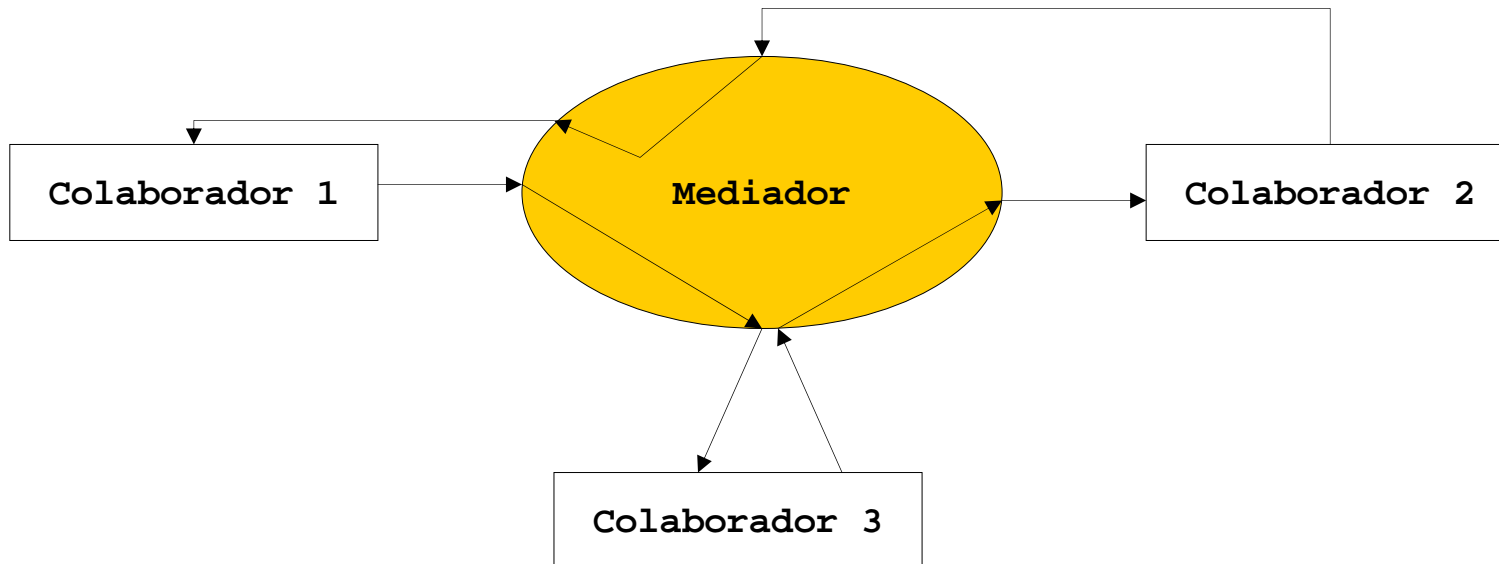
# Problema

- Como permitir que um grupo de objetos se comunique entre si sem que haja acoplamento entre eles?
- Como permitir que novos participantes sejam ligados ao grupo facilmente?



# Solução

- Introduzir um mediador
  - Objetos podem se comunicar sem se conhecer

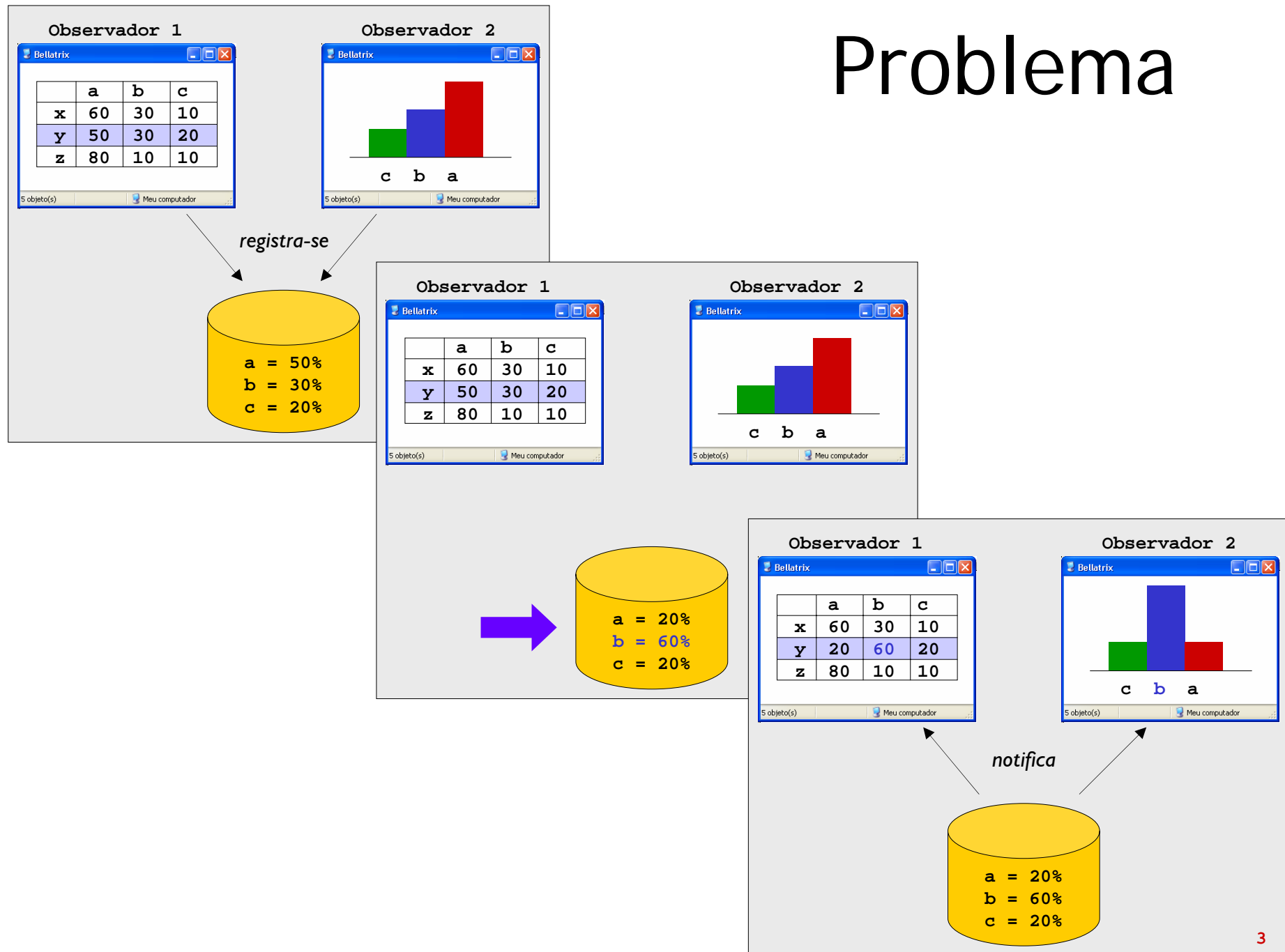


# Observer

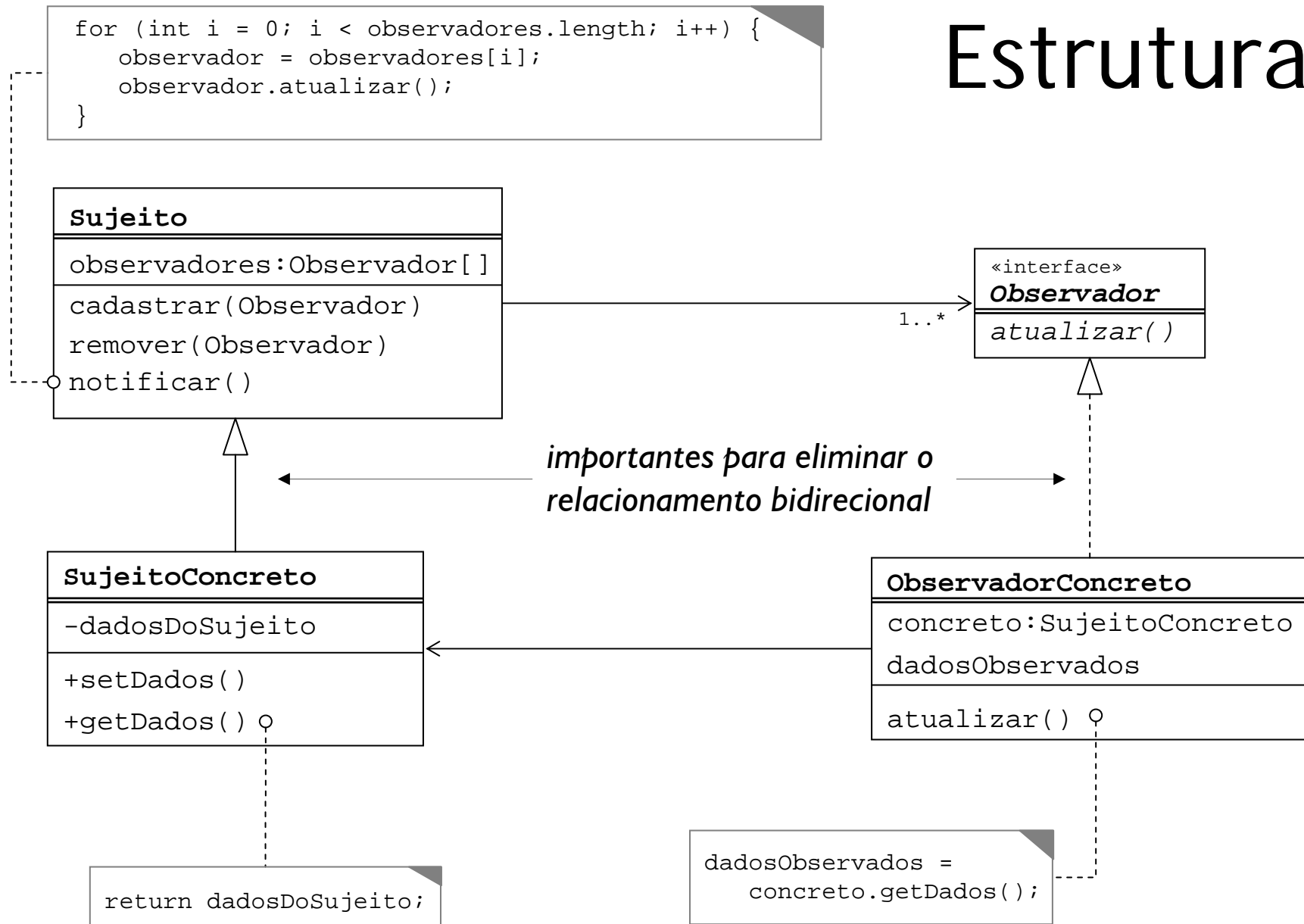
*"Definir uma dependência um-para-muitos entre objetos para que **quando um objeto mudar de estado, todos os seus dependentes sejam notificados** e atualizados automaticamente." [GoF]*



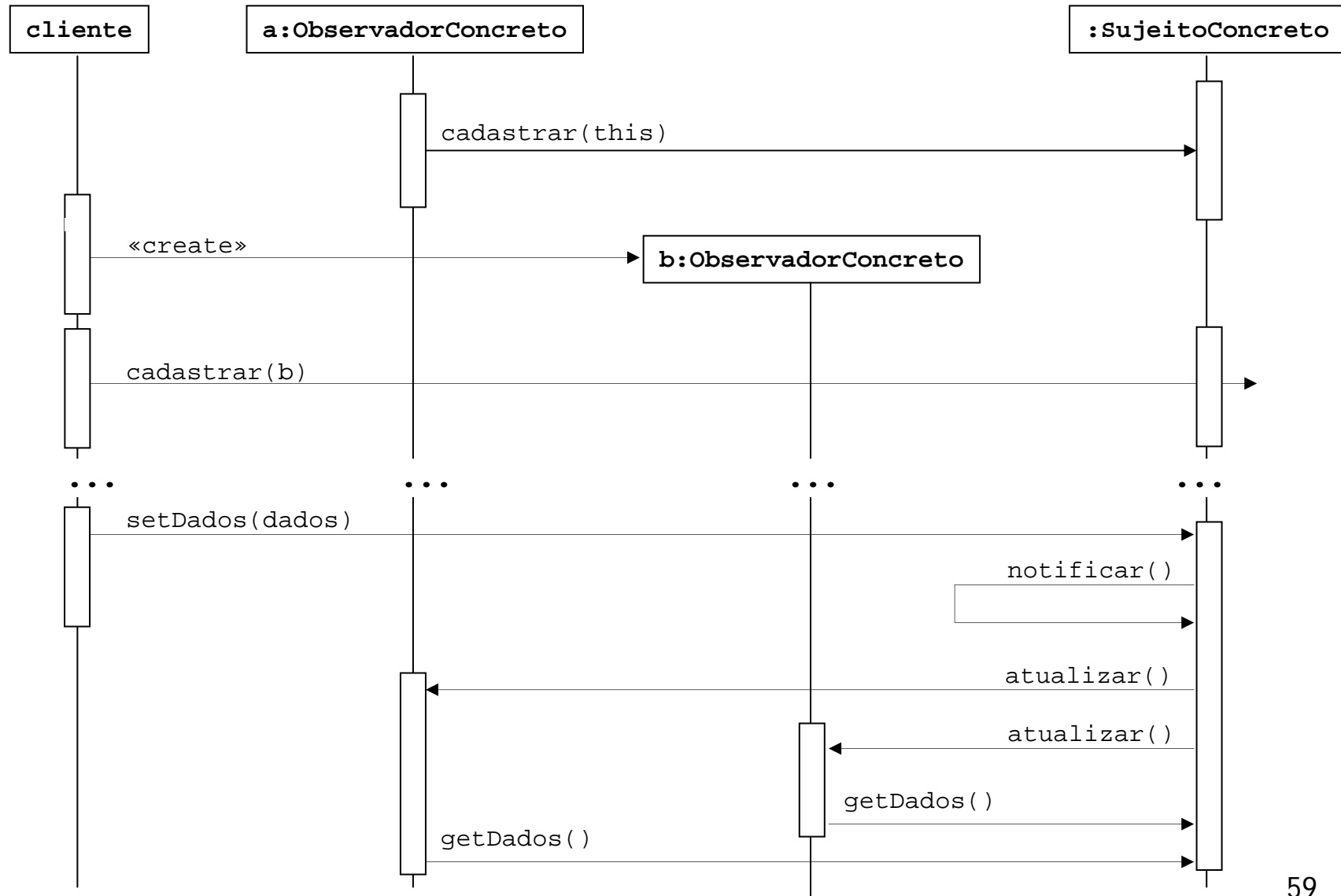
# Problema



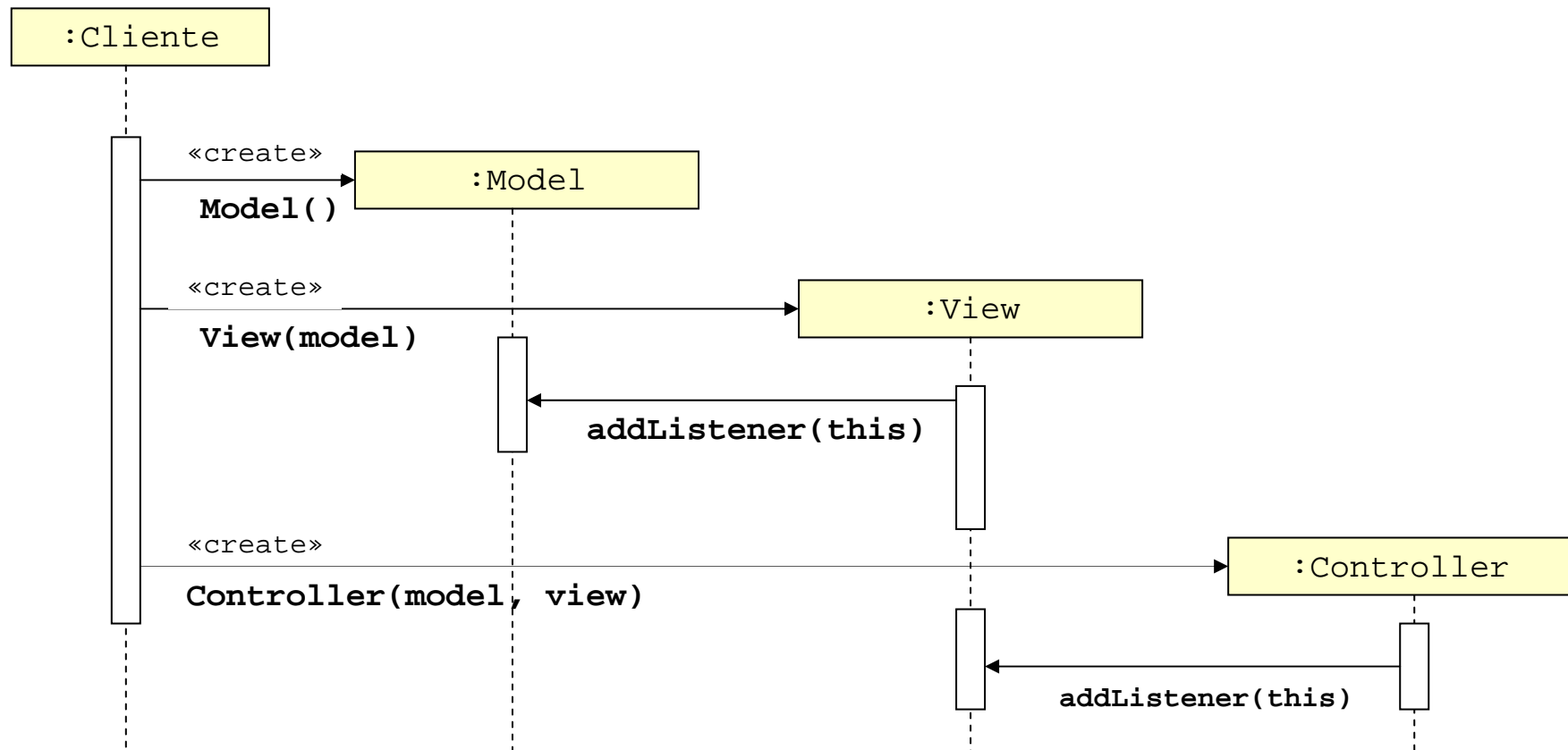
# Estrutura



# Seqüência de Observer

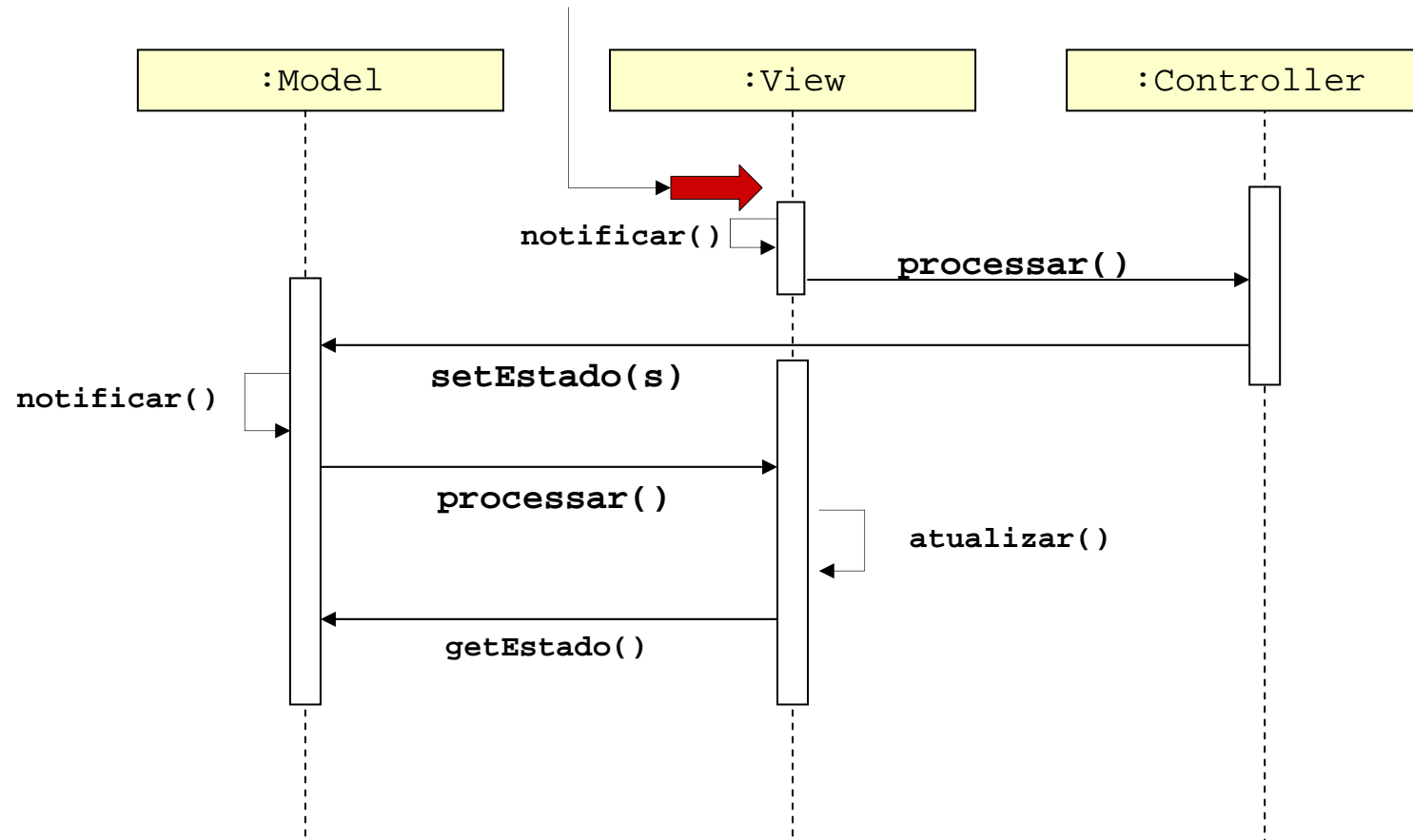


# MVC: registro



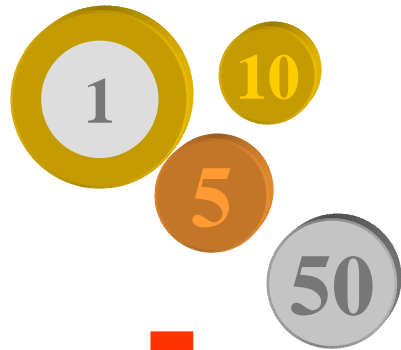
# MVC: operação

*Usuário aperta  
botão "Ação"*



# Chain of Responsibility

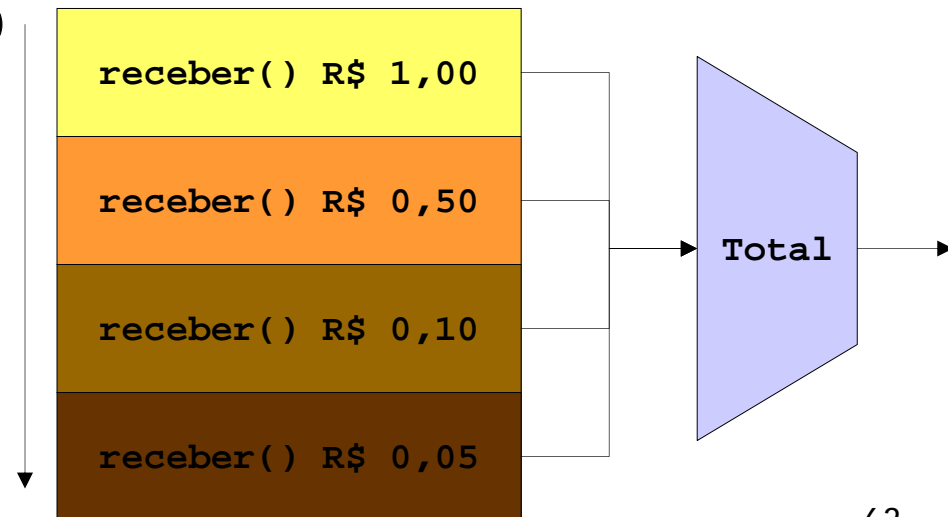
*"Evita acoplar o remetente de uma requisição ao seu destinatário ao dar a mais de um objeto a chance de servir a requisição. Compõe os **objetos em cascata** e passa a requisição pela corrente até que um objeto a sirva." [GoF]*



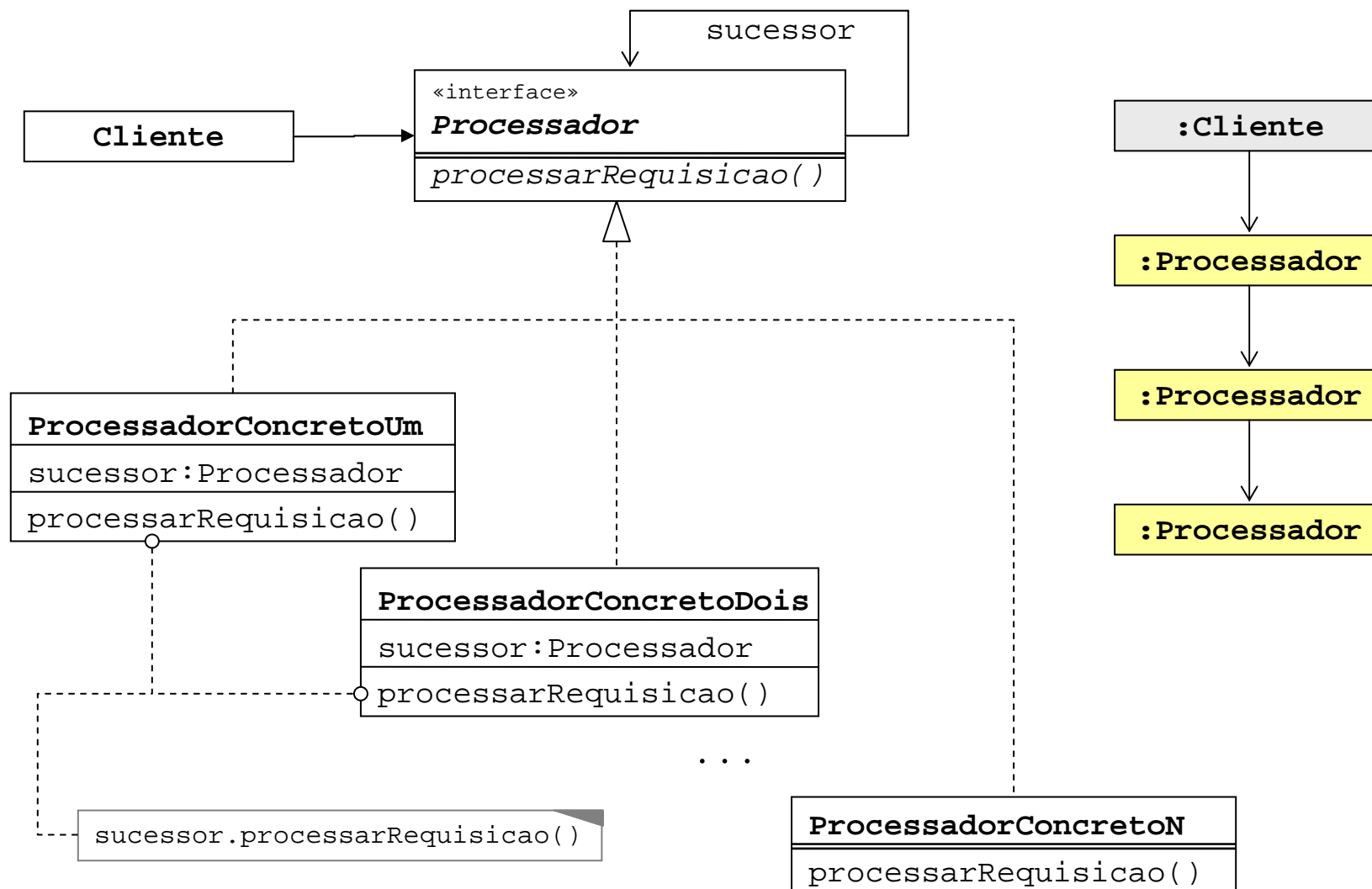
# Problema

- Permitir que vários objetos possam servir a uma requisição ou repassá-la
- Permitir divisão de responsabilidades de forma transparente

*Um objeto pode ser uma **folha** ou uma **composição** de outros objetos*



# Estrutura





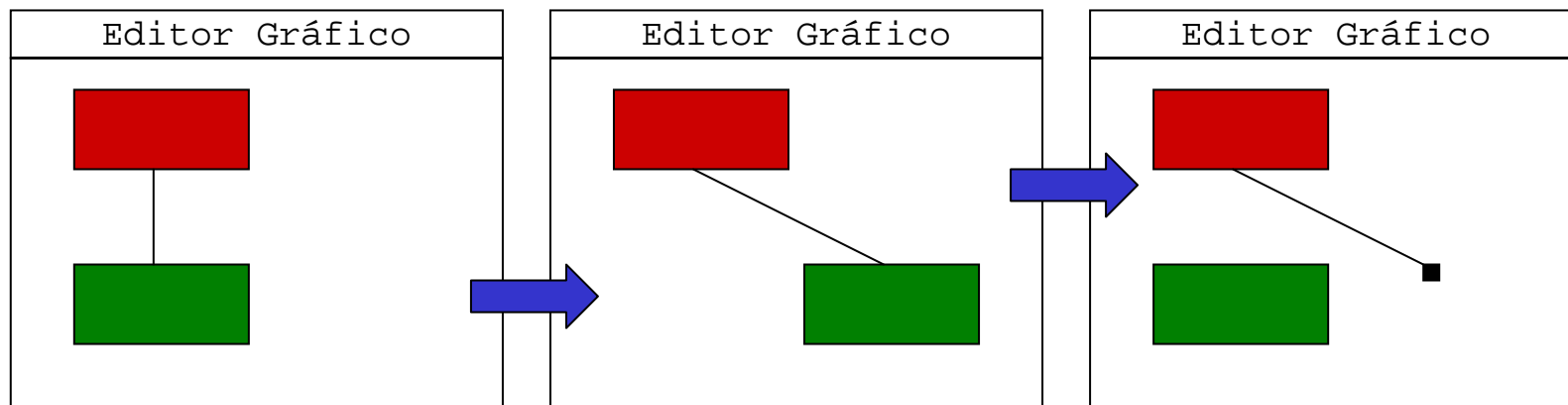
16

# Memento

*"Sem violar o encapsulamento, capturar e expor o estado interno de um objeto **para que o objeto possa ter esse estado restaurado** posteriormente." [GoF]*

# Problema

- É preciso guardar informações sobre um objeto suficientes para desfazer uma operação, mas essas informações não devem ser públicas



*Antes*

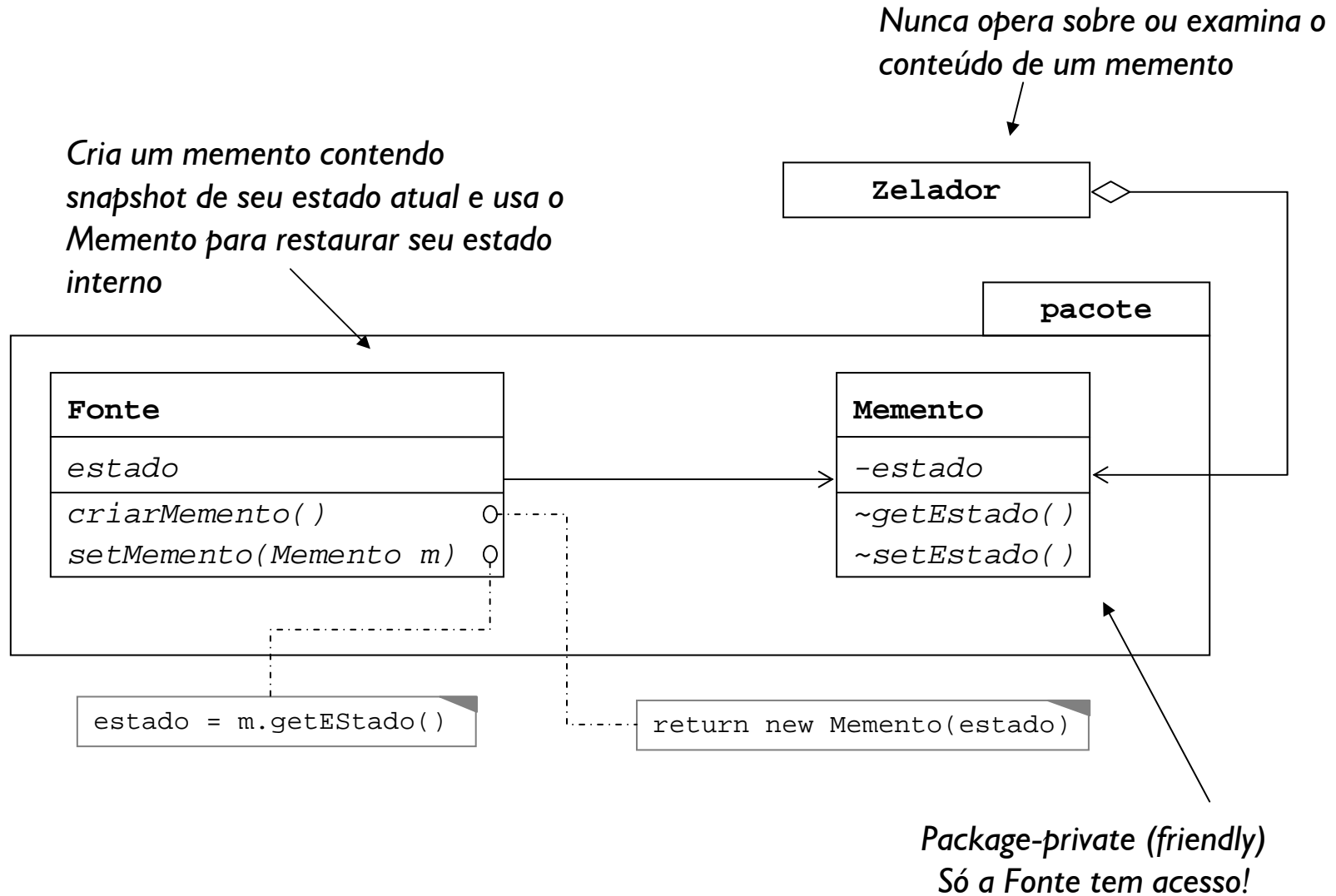
*Ação*

*Undo!*

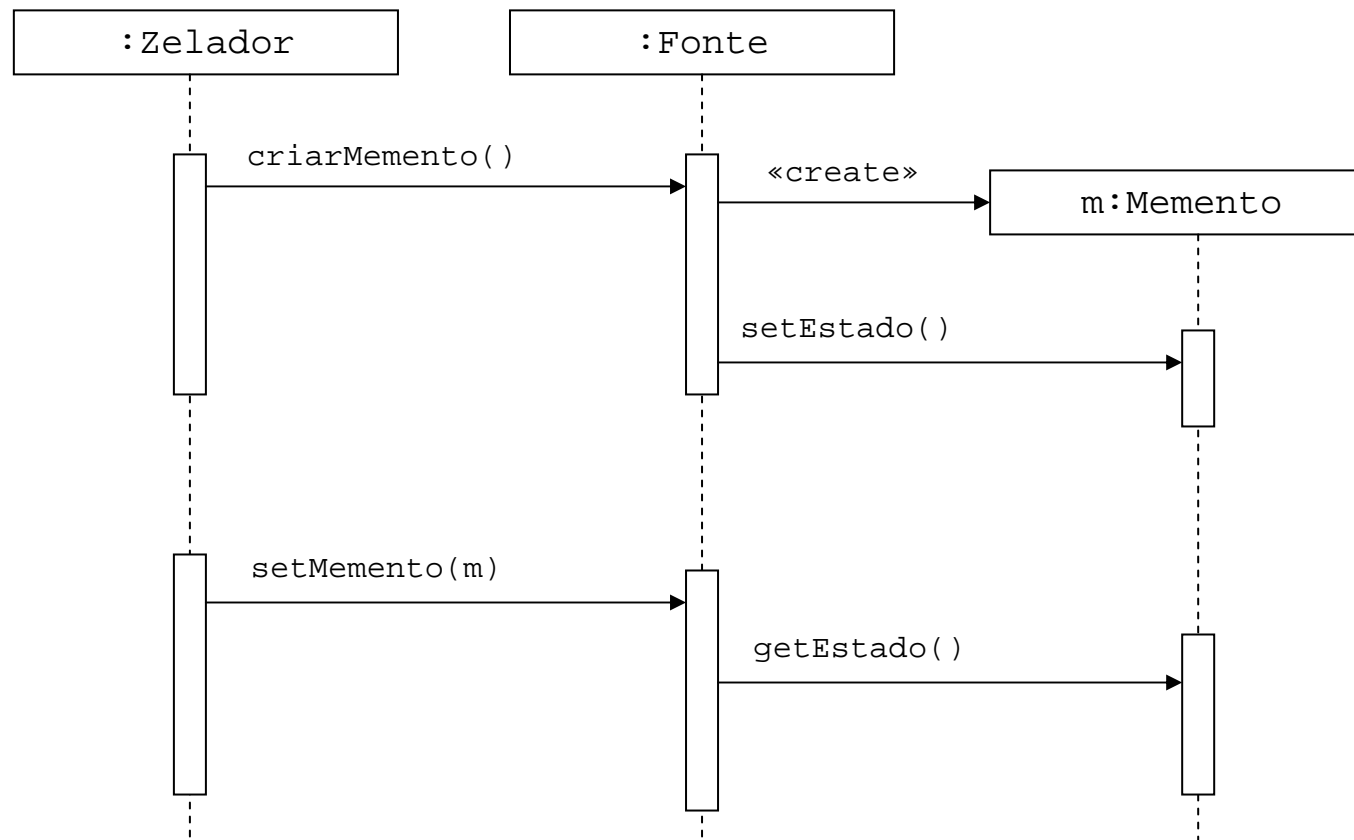
*Não funcionou!*

*Preciso de mais  
informação!*

# Estrutura de Memento



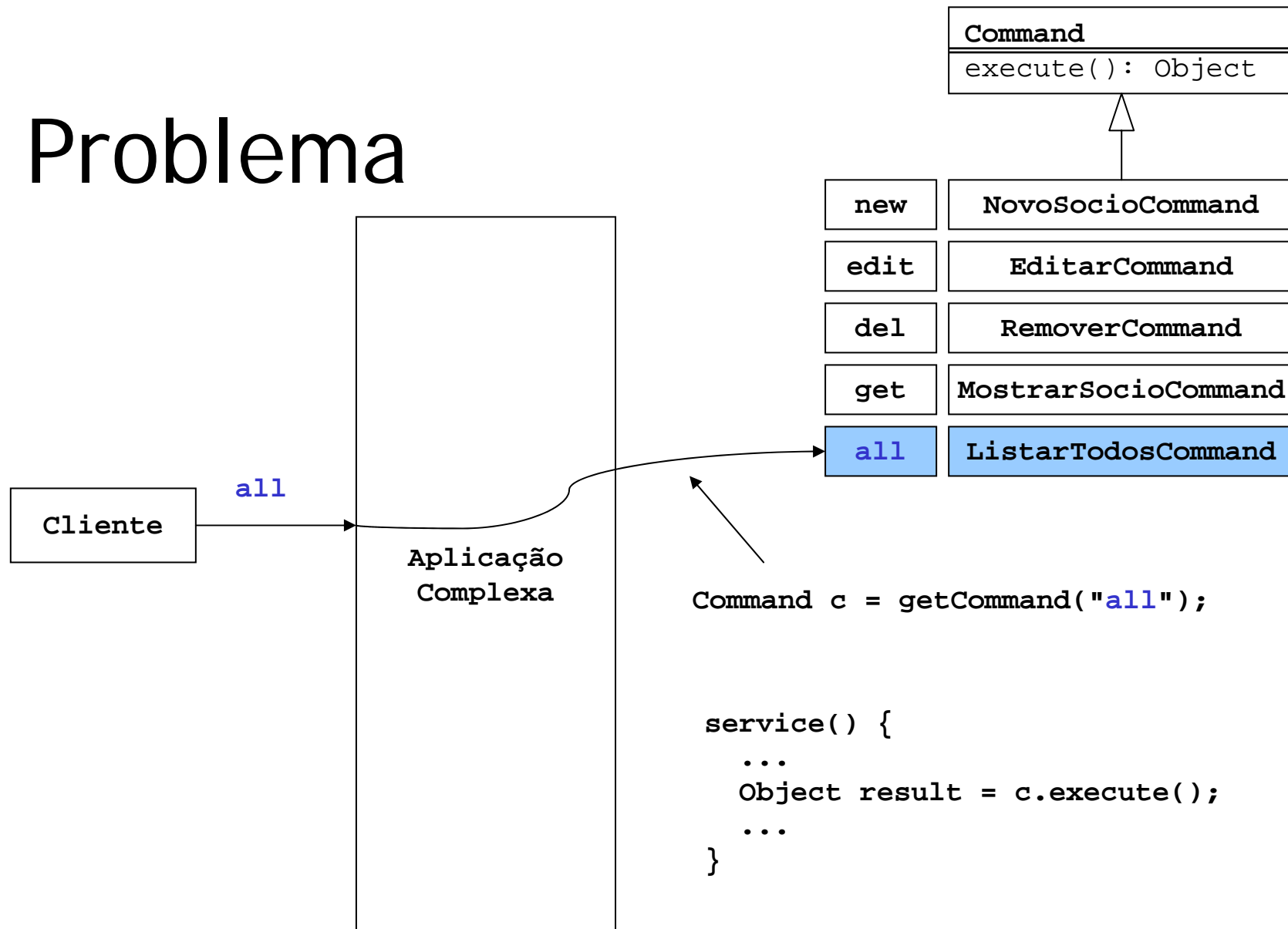
# Seqüência



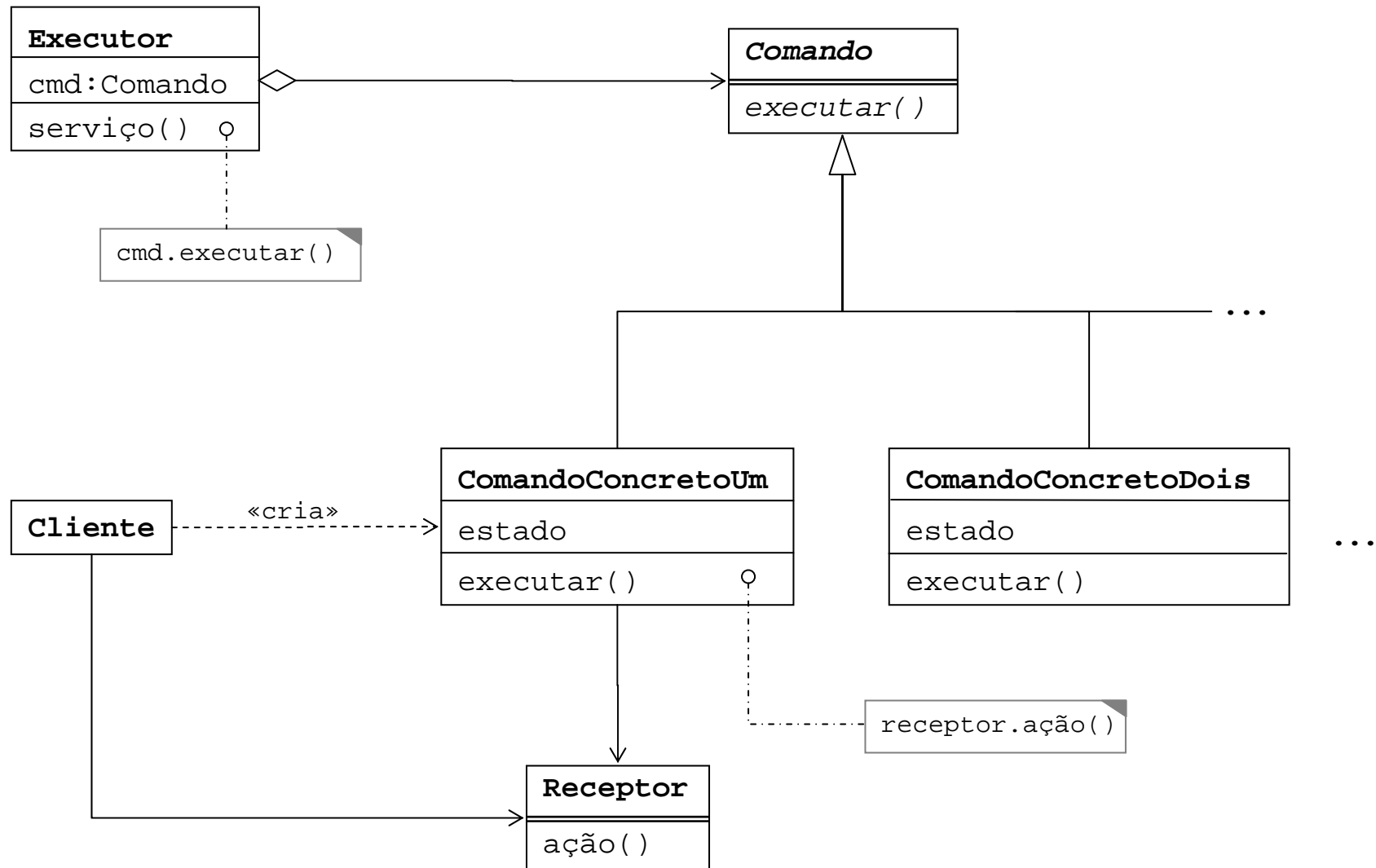
# Command

*"Encapsular **uma requisição como um objeto**, permitindo que clientes parametrizem diferentes requisições, filas ou requisições de log, e suportar operações reversíveis." [GoF]*

# Problema



# Estrutura de Command



# Command em Java

```
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);  
    }  
}
```

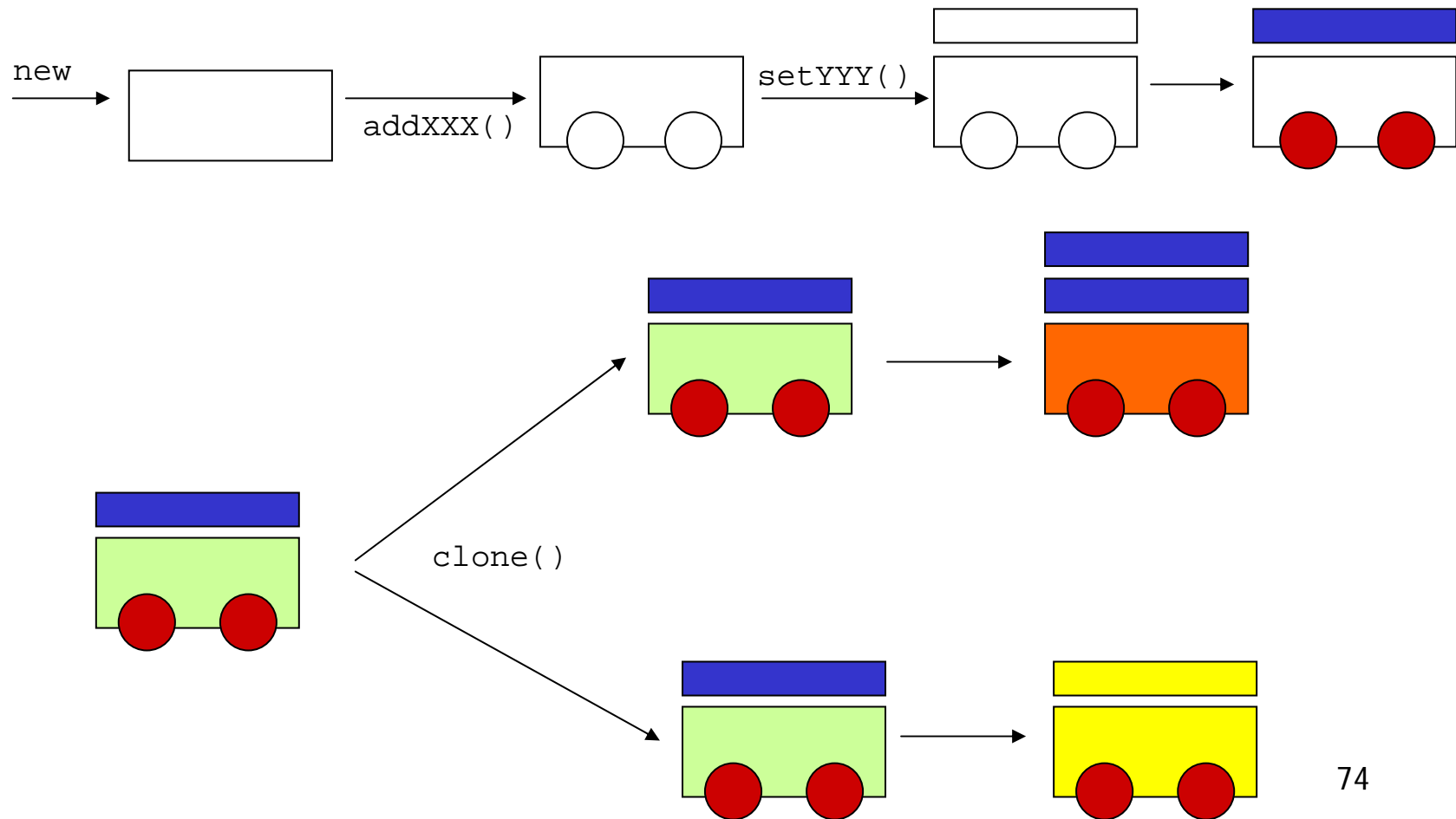


# Prototype

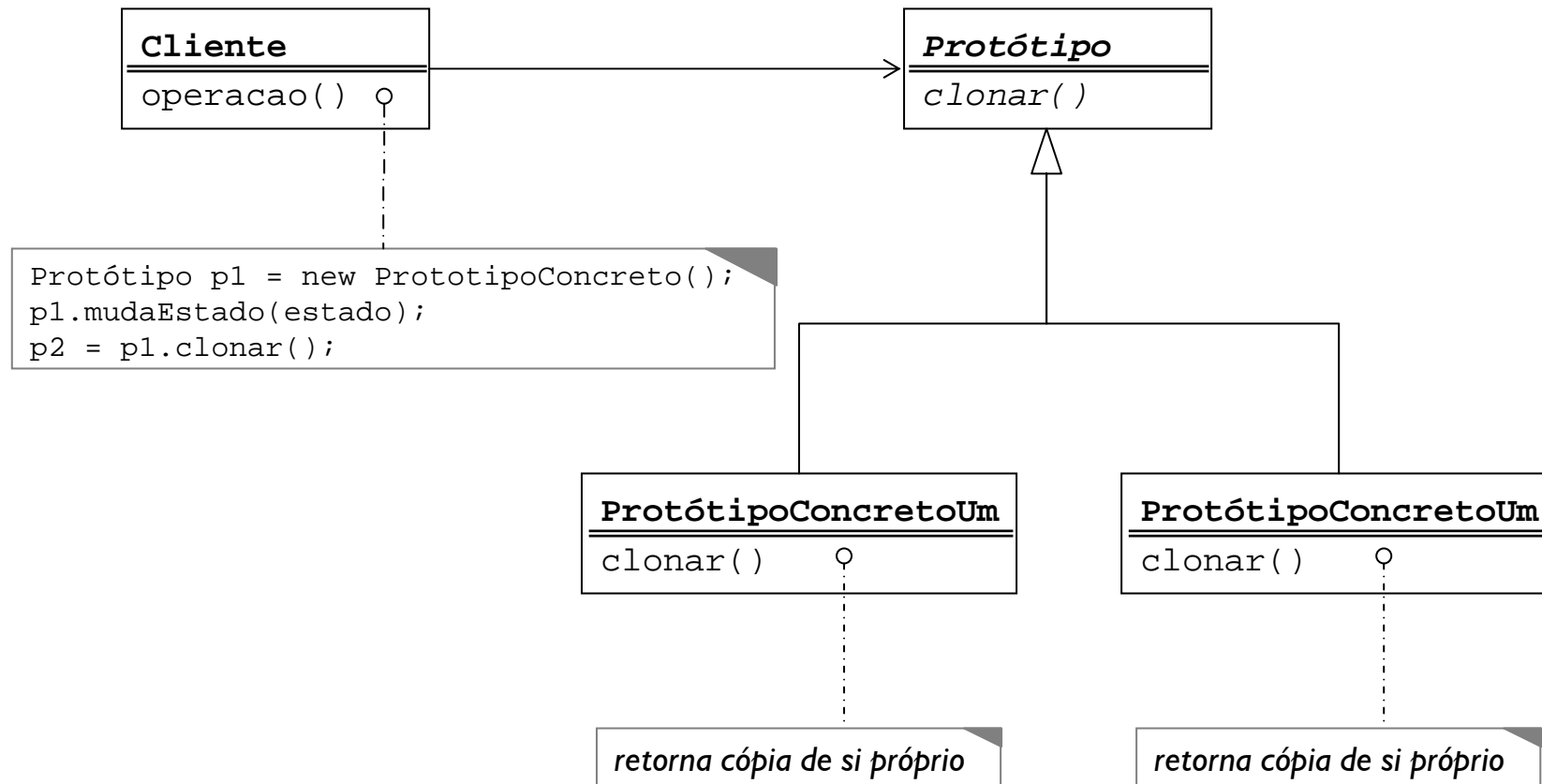
*"Especificar os tipos de objetos a serem criados usando uma **instância como protótipo** e criar novos objetos ao copiar este protótipo." [GoF]*

# Problema

- Criar um objeto novo, mas aproveitar o estado previamente existente em outro objeto



# Estrutura de Prototype

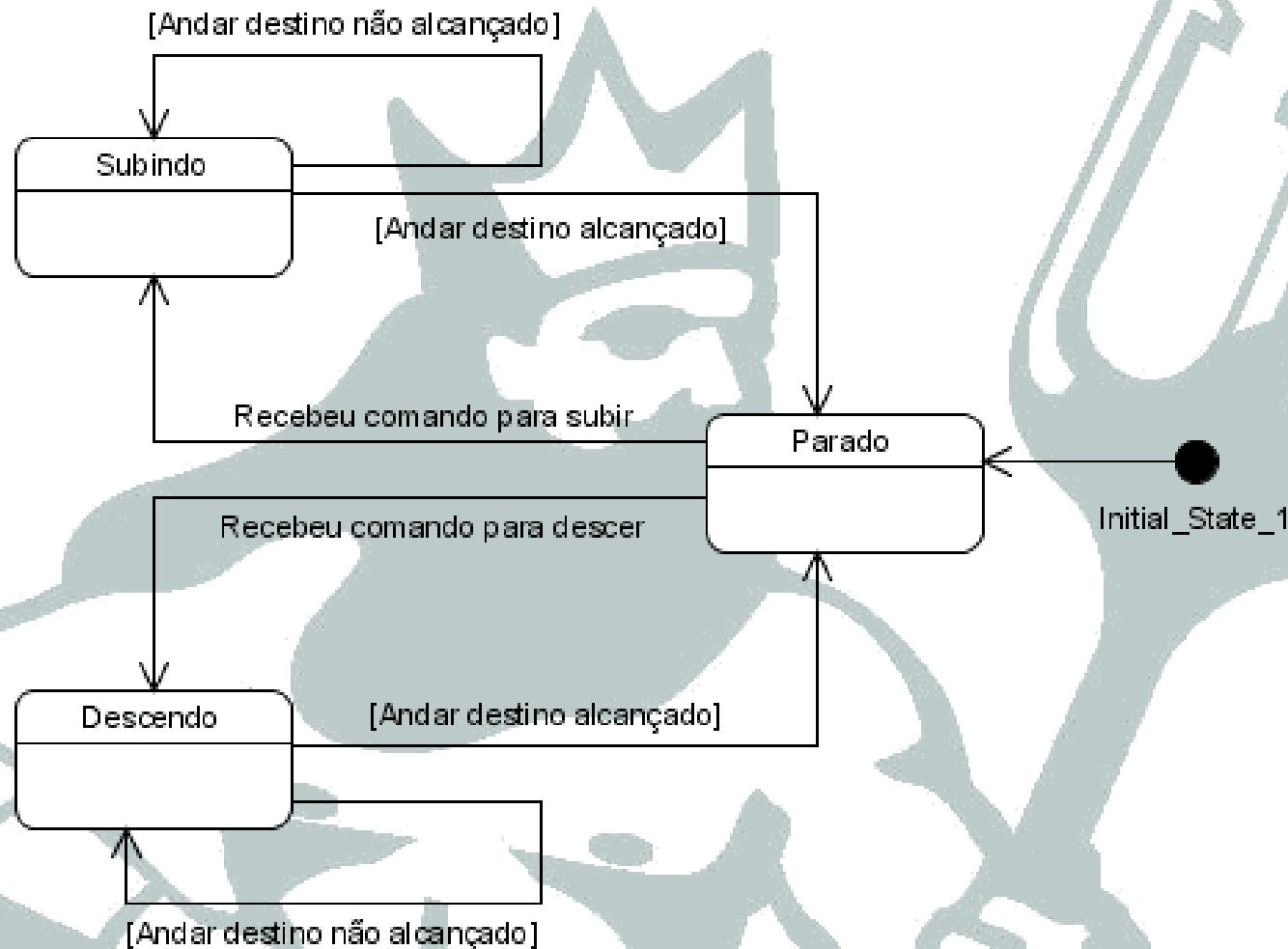


# 19

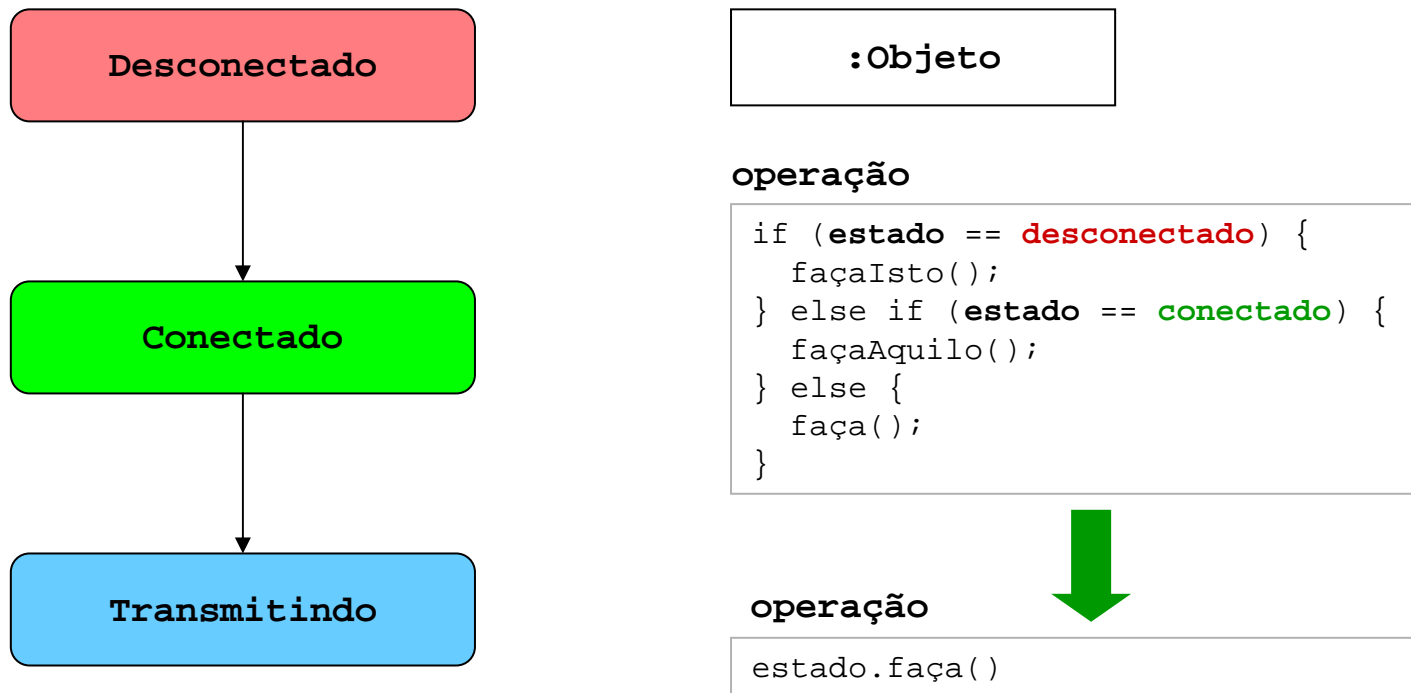
## State

*"Permitir a um objeto **alterar o seu comportamento quanto o seu estado interno mudar**. O objeto irá aparentar mudar de classe." [GoF]*

# Cenário típico

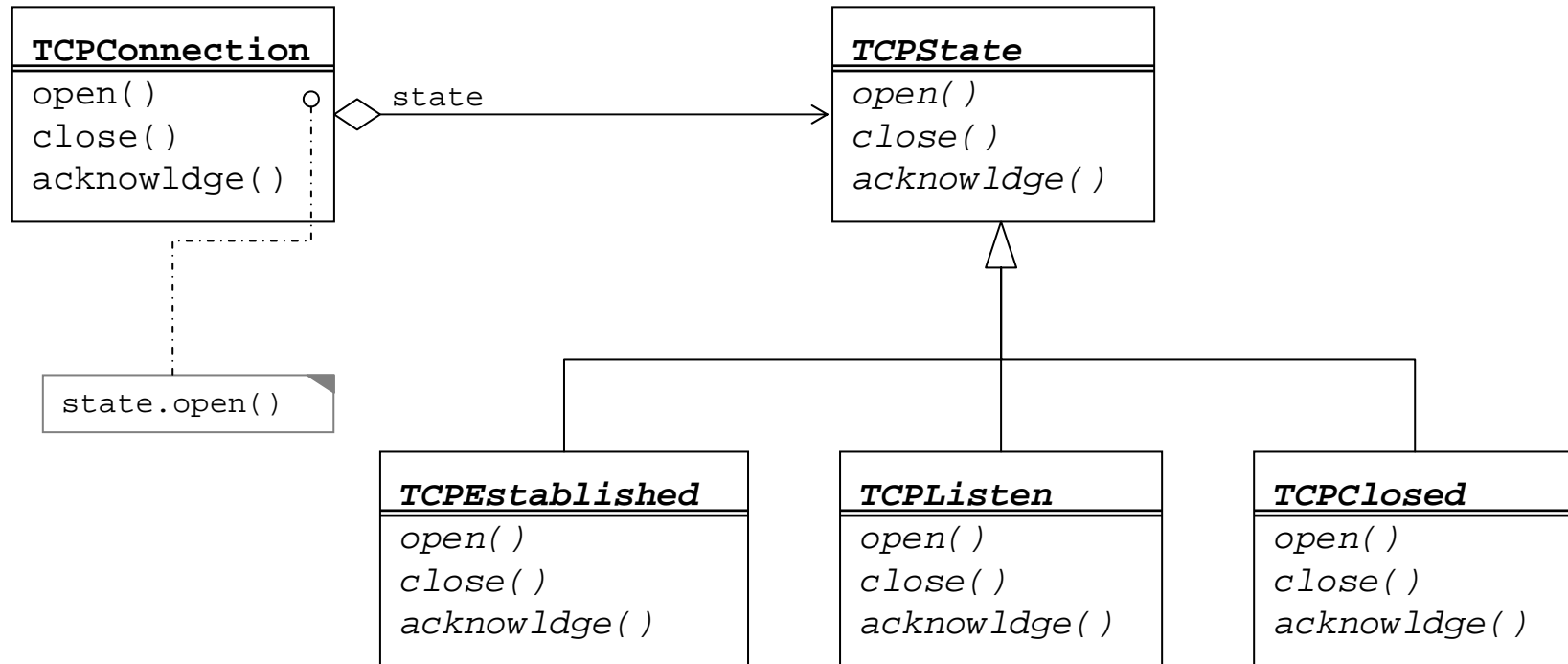


# Problema



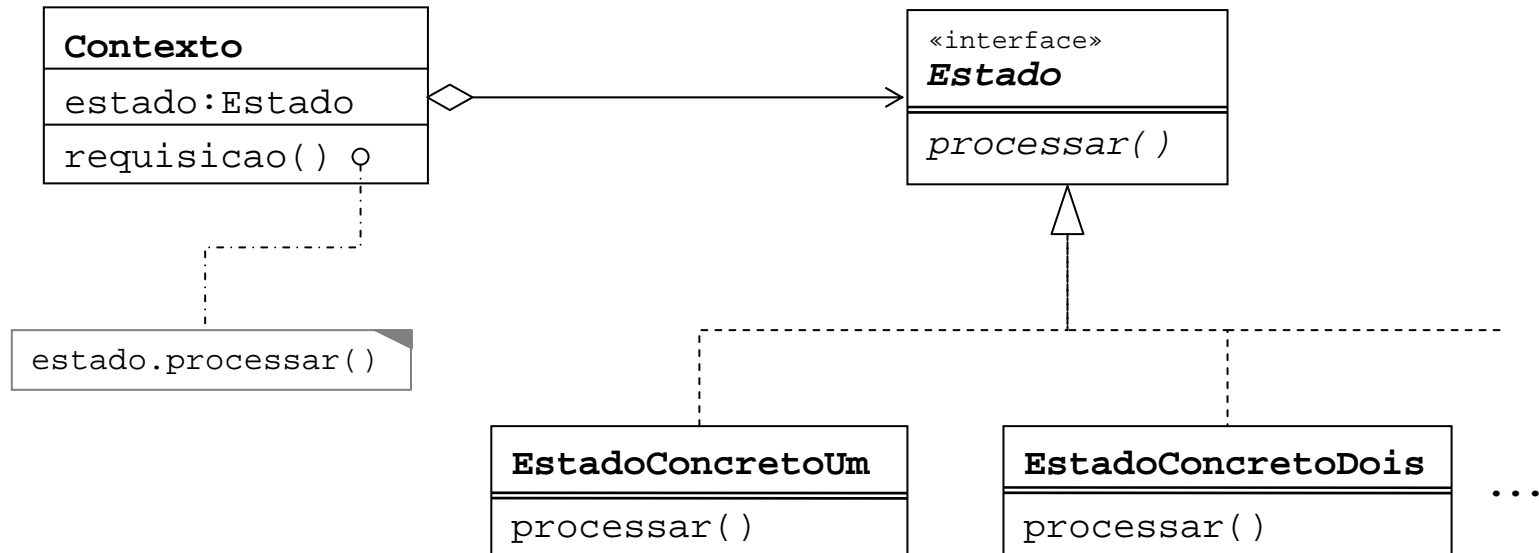
*Objetivo: usar objetos para representar **estados** e **polimorfismo** para tornar a execução de tarefas dependentes de estado transparentes*

# Exemplo [GoF]



*Sempre que a aplicação mudar de estado, o objeto TCPConnection muda o objeto TCPState que está usando*

# Estrutura



- State é um tipo de Strategy cuja mudança de algoritmo é totalmente encapsulada



20

# Visitor

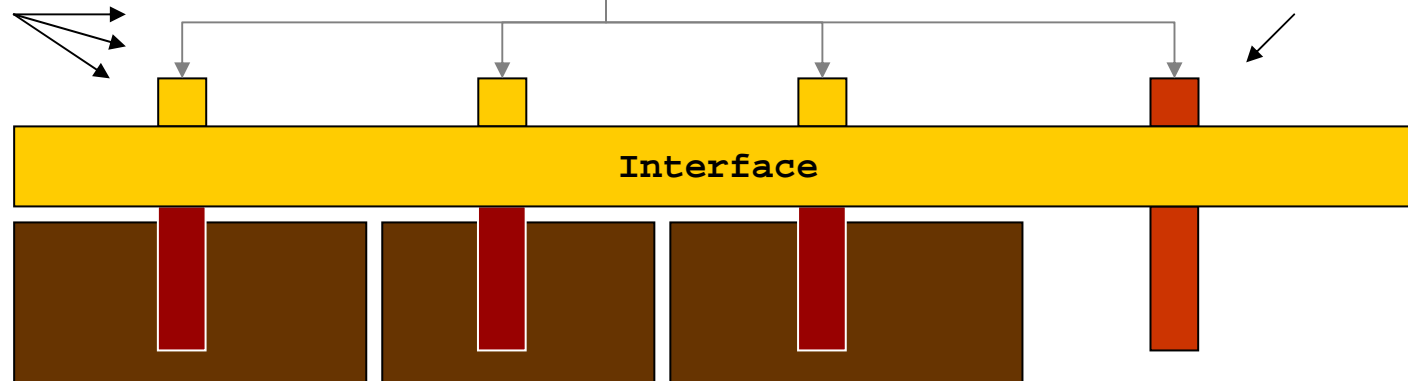
*"Representar uma operação a ser realizada sobre os elementos de uma estrutura de objetos. Visitor permite **definir uma nova operação sem mudar as classes dos elementos** nos quais opera." [GoF]*

# Problema

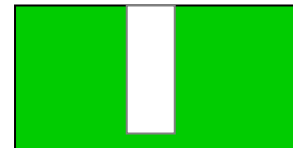
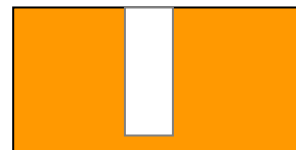
*Operações  
fixas*

Cliente

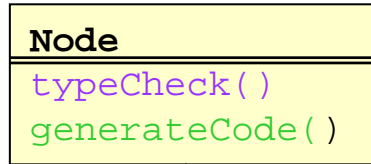
*Operação para suporte  
a extensões*



*Operações  
novas  
plugáveis*

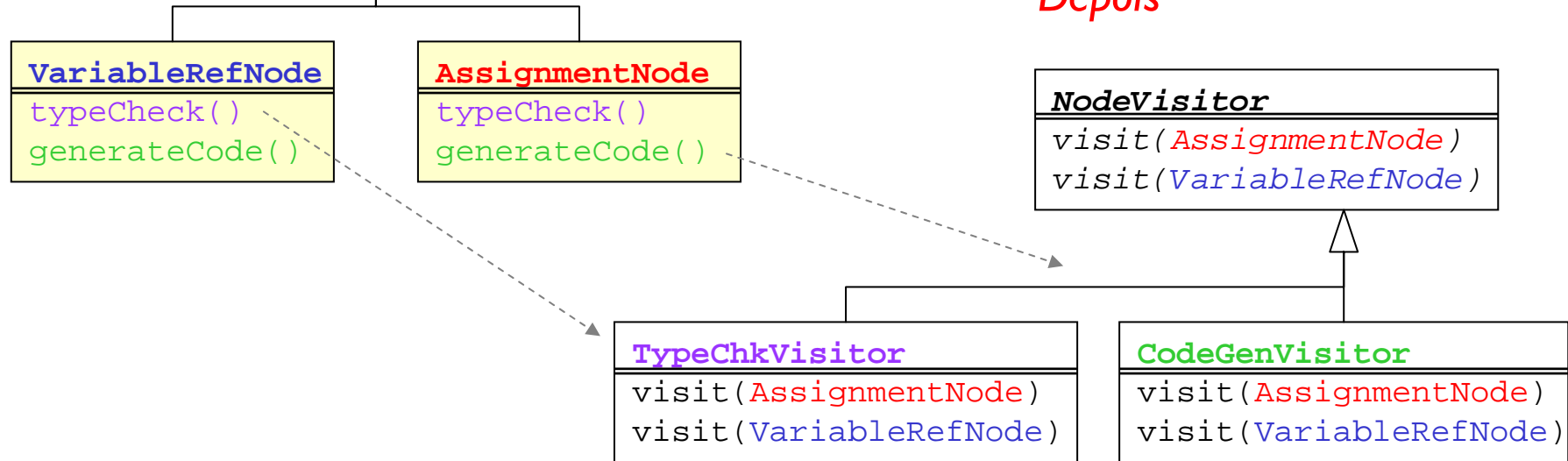


Antes

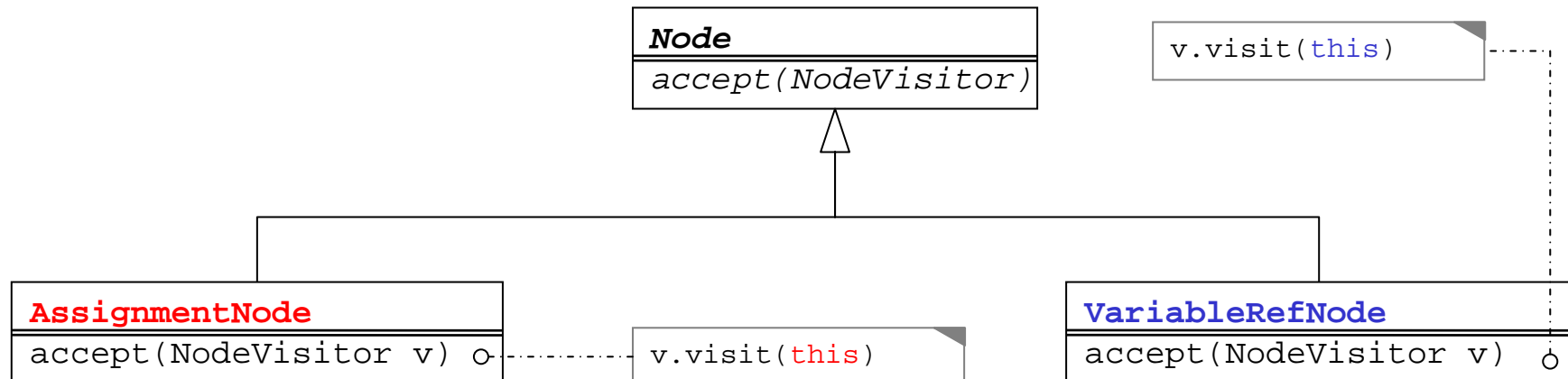


# visitor: exemplo GoF

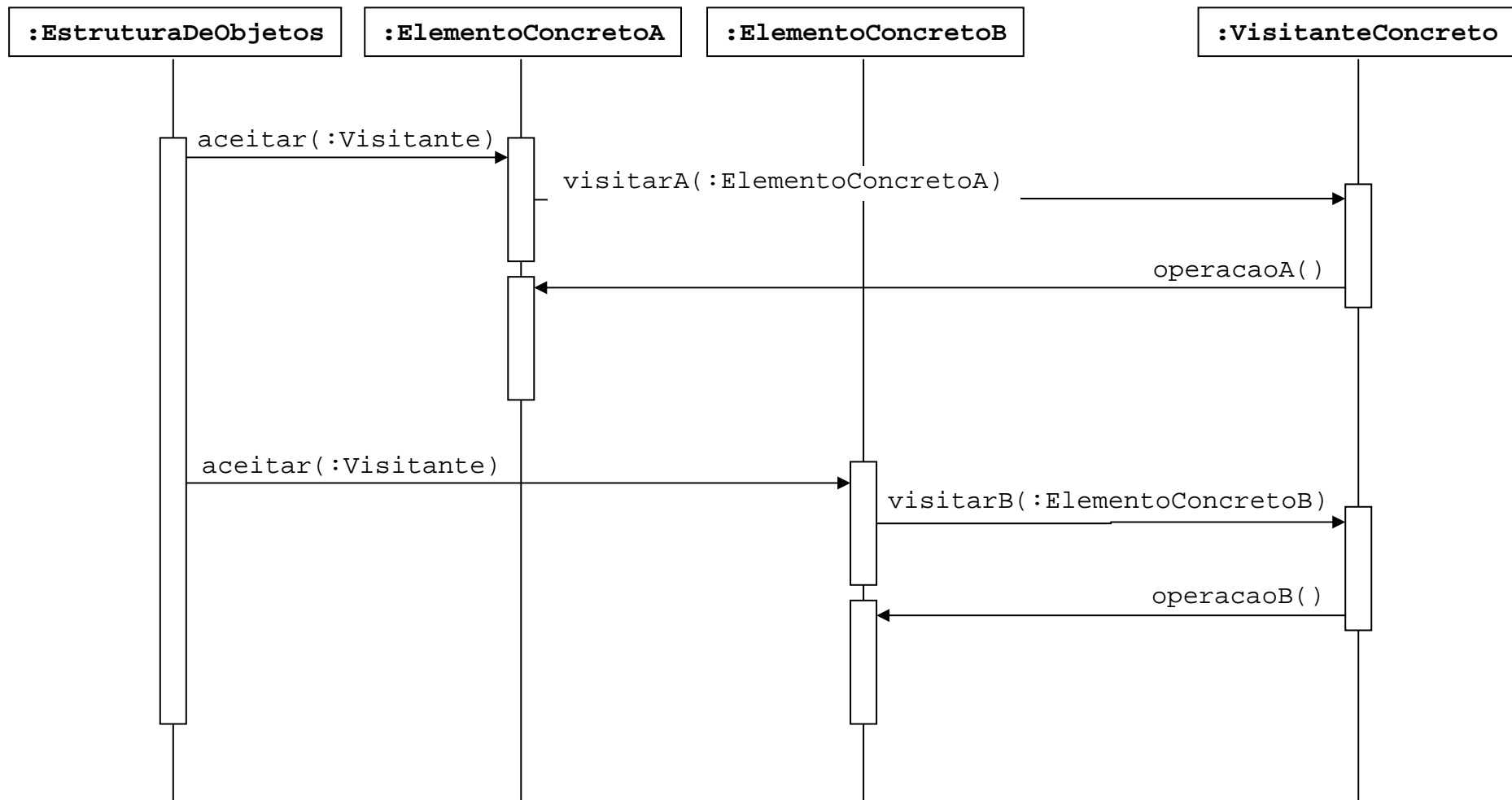
Depois



Depois



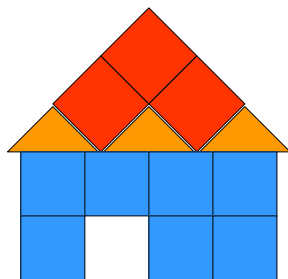
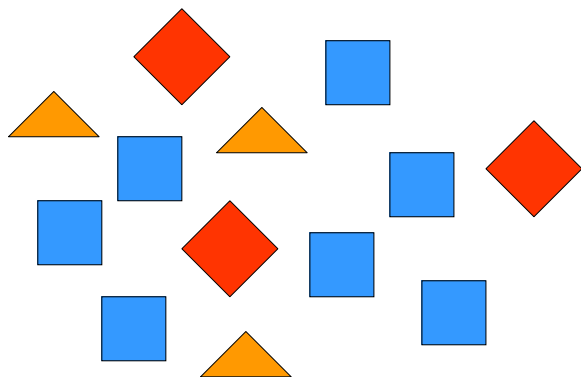
# Diagrama de seqüência



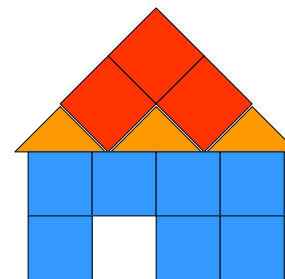
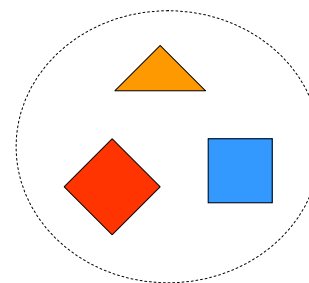
# Flyweight

*"Usar **compartilhamento** para suportar grandes quantidades de objetos refinados eficientemente." [GoF]*

# Problema



*Pool de objetos  
imutáveis  
compartilhados*



# Interpreter

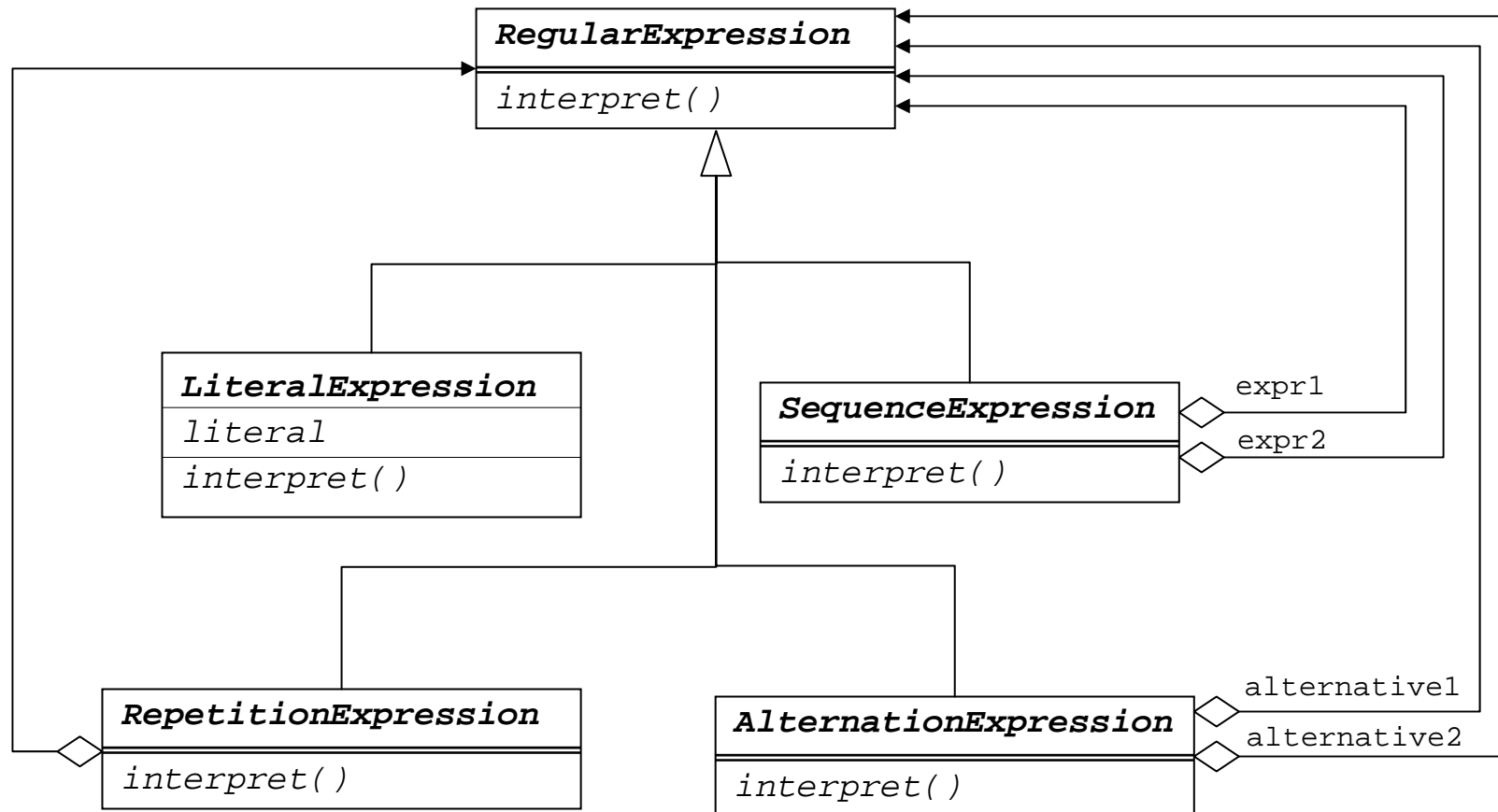
*"Dada uma linguagem, definir uma **representação para sua gramática** junto com um interpretador que usa a representação para interpretar sentenças na linguagem."  
[GoF]*

# Interpreter

- Se comandos estão representados como objetos, eles poderão fazer parte de **algoritmos** maiores
  - Use objetos Command para encapsulá-los
- Interpreter
  - Uma extensão do padrão **Command** (uma micro-arquitetura construída com base em Commands) em que toda uma lógica de código pode ser implementada com objetos



# Exemplo [GoF]

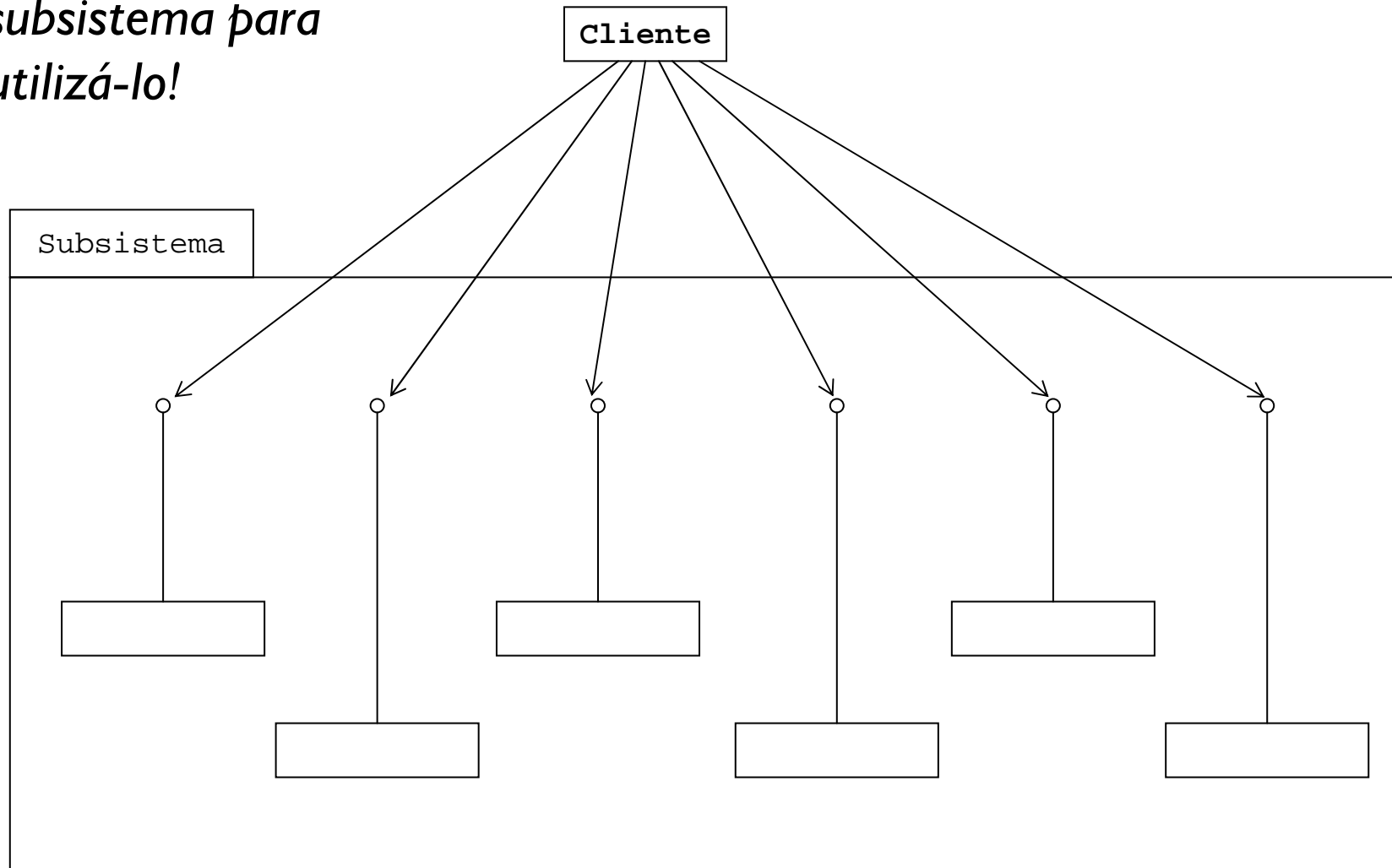


# Façade

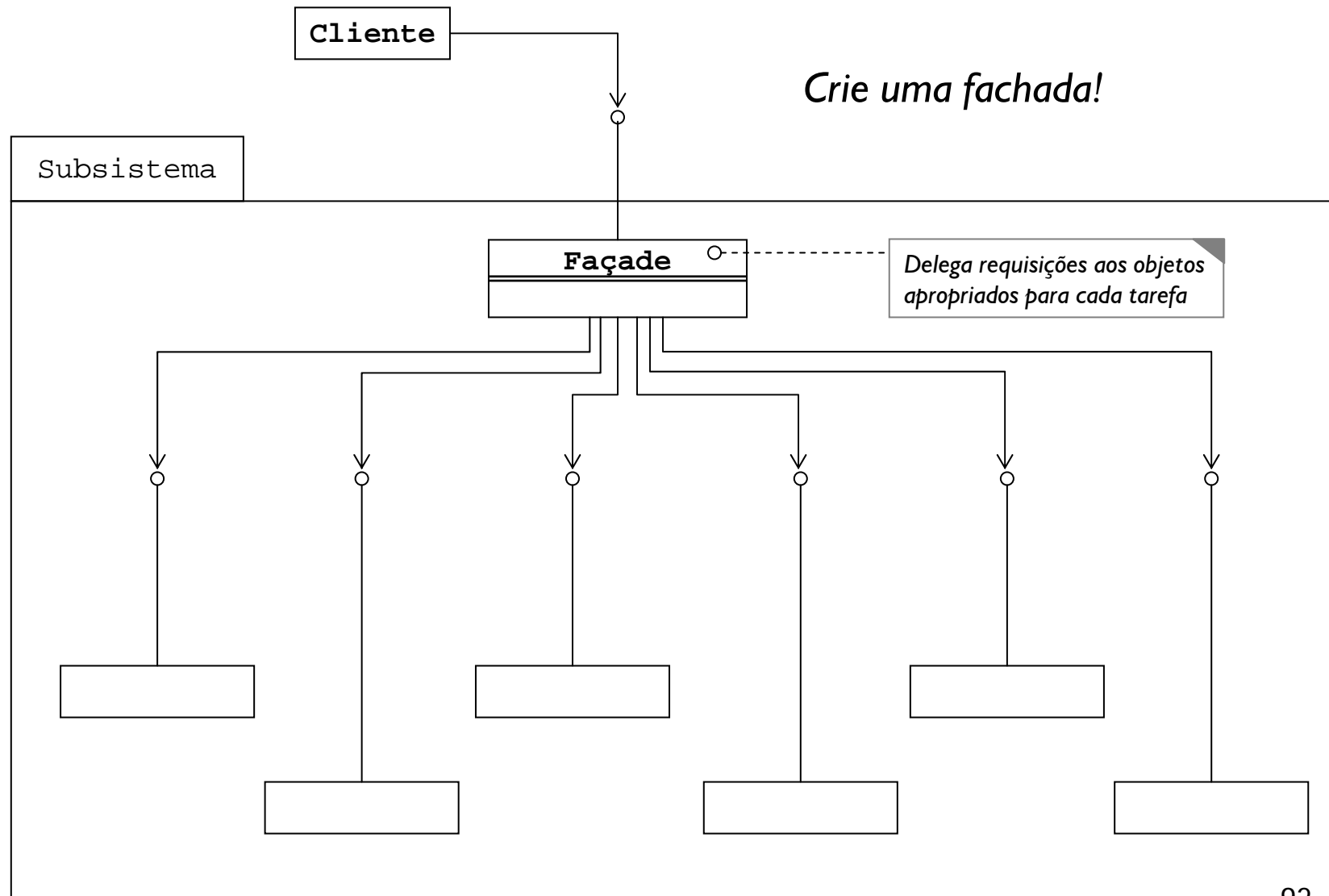
*"Oferecer uma interface única para um conjunto de interfaces de um subsistema. Façade define uma interface de nível mais elevado que torna o subsistema **mais fácil de usar**." [GoF]*

*Cliente precisa saber  
muitos detalhes do  
subsistema para  
utilizá-lo!*

# Problema



# Estrutura de Façade



# Conclusões

- Esta palestra apresentou superficialmente os 23 padrões de design clássicos (GoF)
  - Foram destacados os mais importantes
- Esses são os principais padrões que surgem em aplicações OO
  - Estude-os e aprenda a identificá-los e distingui-los para tornar-se um especialista em OO!
- Veja uma abordagem mais detalhada, fontes, código e links para mais informação em
  - [www.argonavis.com.br/designpatterns/](http://www.argonavis.com.br/designpatterns/)  
(em breve, antes do dia 02/12)

# Fontes

- [1][Metsker] Steven John Metsker, [Design Patterns Java Workbook](#). Addison-Wesley, 2002,
- [2][GoF] Erich Gamma et al. [Design Patterns: Elements of Reusable Object-oriented Software](#). Addison-Wesley, 1994
- [3] James W. Cooper. [The Design Patterns Java Companion](#).  
<http://www.patterndepot.com/put/8/JavaPatterns.htm>
- [4][Larman] Craig Larman, [Applying UML and Patterns](#), 2nd. Edition, Prentice-Hall, 2002
- [5][EJ] Joshua Bloch, [Effective Java Programming Guide](#), Addison-Wesley, 2001

Extraído do  
**Curso J930: Design Patterns**  
Versão 2.1

[www.argonavis.com.br](http://www.argonavis.com.br)

© 2003, 2005, Helder da Rocha  
(helder.darocha@gmail.com)