Padrões de Design

com aplicações em Java

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

Objetivos

- Apresentar cada um dos 23 padrões clássicos (catálogo do ´Gang of Four´) descrevendo [2][GoF]
  - O problema que solucionam
  - A solução
  - Diagramas UML
  - Exemplos em Java
  - Aplicações típicas
- Apresentar os 9 padrões de atribuições de responsabilidade (GRASP) [4][Larman]
- Apresentar 2 padrões emergentes
  - Injeção de dependências (aplicação do GRASP Indirection)
  - Aspectos

O que é um padrão?

- Maneira testada ou documentada de alcançar um objetivo qualquer
  - Padrões são comuns em várias áreas da engenharia
- Design Patterns, ou Padrões de Design*
  - Padrões para alcançar objetivos na engenharia de software usando classes e métodos em linguagens orientadas a objeto
  - Inspirado em "A Pattern Language" de Christopher Alexander, sobre padrões de arquitetura de cidades, casas e prédios

* Ou ainda Padrões de Projeto, embora algo se perca nesta tradução!!

Responsabilidades

- Booch e Rumbaugh "Responsabilidade é um contrato ou obrigação de um tipo ou classe."
- Dois tipos de responsabilidades dos objetos:
  - De conhecimento (knowing): sobre dados privativos e encapsulados; sobre objetos relacionados; sobre coisas que pode calcular ou derivar.
  - De realização (doing): fazer alguma coisa em si mesmo; iniciar uma ação em outro objeto; controlar e coordenar atividades em outros objetos.
- Responsabilidades são atribuídas aos objetos durante o design

Responsabilidades e Métodos

- *A tradução de responsabilidades em classes e métodos depende da granularidade da responsabilidade*
- *Métodos são implementados para cumprir responsabilidades*
  - *Uma responsabilidade pode ser cumprida por um único método ou uma coleção de métodos trabalhando em conjunto*
- *Responsabilidades do tipo knowing geralmente são inferidas a partir do modelo conceitual (são os atributos e relacionamentos)*

Responsabilidades e Diagramas de Interação

- *Diagramas de interação mostram escolhas ao atribuir responsabilidades a objetos*
  - *No diagrama de colaboração ao lado objetos Order têm a responsabilidade de se prepararem: método prepare()*
  - *O cumprimento dessa responsabilidade requer colaboração com objetos Order Line e Stock Item*

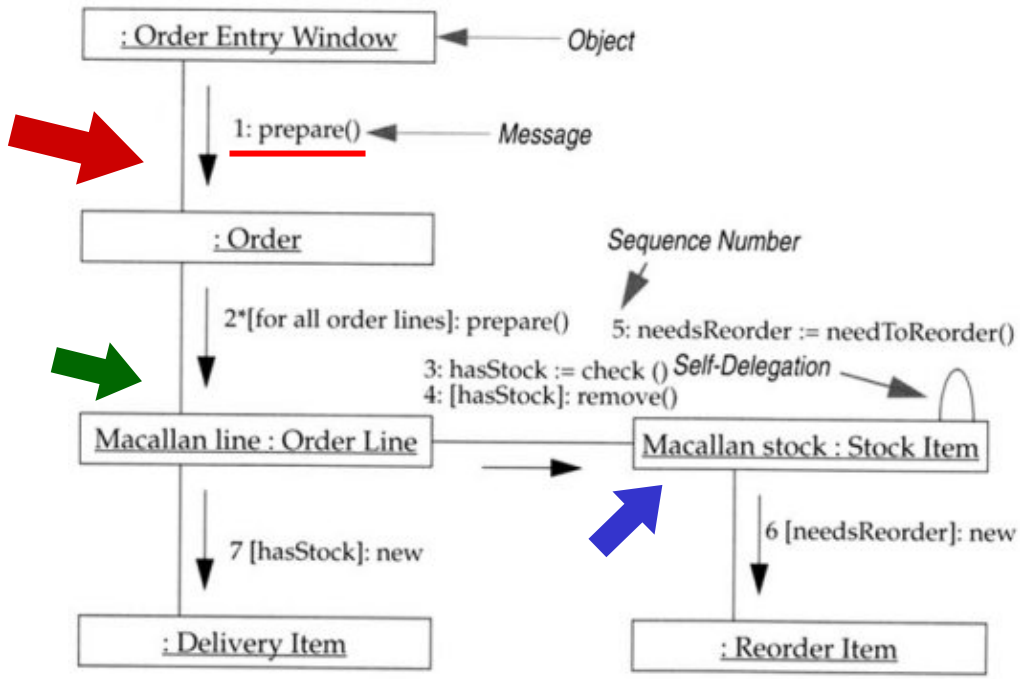


Figure 5-4: *Collaboration Diagram with Simple Numbering*

Fonte: [4] Craig Larman, Applying UML and Patterns, 2nd. Edition

Padrões

- Padrões são um repertório de soluções e princípios que ajudam os desenvolvedores a criar software e que são codificados em um formato estruturado consistindo de
  - Nome
  - Problema que soluciona
  - Solução do problema
- O objetivo dos padrões é codificar conhecimento (knowing) existente de uma forma que possa ser reaplicado em contextos diferentes

Padrões GRASP

- *Introduzidos por Craig Larman em seu livro "Applying UML and Patterns" [4]*
- *GRASP: **G**eneral **R**esponsibility and **A**ssignment **S**oftware **P**atterns*
  - *Os padrões GRASP descrevem os princípios fundamentais da atribuição de responsabilidades a objetos, expressas na forma de padrões*
- *Esses padrões exploram os princípios fundamentais de sistemas OO*
  - *5 padrões fundamentais*
  - *4 padrões avançados*
- *Se você conhece os padrões GRASP, pode dizer que compreende o paradigma orientado a objetos*

Padrões clássicos ou padrões GoF

- *O livro "Design Patterns" (1994) de Erich Gamma, John Vlissides, Ralph Jonhson e Richard Helm, descreve 23 padrões de design [2]*
  - *São soluções genéricas para os problemas mais comuns do desenvolvimeto de software orientado a objetos*
  - *O livro tornou-se um clássico na literatura orientada a objeto e continua atual*
  - *Não são invenções. São documentação de soluções obtidas através da experiência. Foram coletados de experiências de sucesso na indústria de software, principalmente de projetos em C++ e SmallTalk*
  - *Os quatro autores, são conhecidos como "The Gang of Four", ou GoF*

Mais padrões?

- *Há vários catálogos de padrões em software*
  - *Muitos são específicos a uma determinada área (padrões J2EE, padrões de implementação em Java, em C#, padrões para concorrência, sistemas distribuídos, etc.)*
  - *Os padrões apresentados aqui são aplicáveis em Java e outras linguagens*
- *Dois outros padrões serão apresentados*
  - *Dependency Injection: um caso particular de um dos padrões GRASP (Indirection) bastante popular no momento (também conhecido como Inversão de Controle)*
  - *Aspectos: uma extensão ao paradigma orientado a objetos que ajuda a lidar com limitações dos sistemas OO*

Por que aprender padrões?

- Aprender com a experiência dos outros
  - Identificar problemas comuns em engenharia de software e utilizar soluções testadas e bem documentadas
  - Utilizar soluções que têm um nome: facilita a comunicação, compreensão e documentação
- Aprender a programar bem com orientação a objetos
  - Os 23 padrões de projeto "clássicos" utilizam as melhores práticas em OO para atingir os resultados desejados
- Desenvolver software de melhor qualidade
  - Os padrões utilizam eficientemente polimorfismo, herança, modularidade, composição, abstração para construir código reutilizável, eficiente, de alta coesão e baixo acoplamento

Por que aprender padrões?

- Vocabulário comum
  - Faz o sistema ficar menos complexo ao permitir que se fale em um nível mais alto de abstração
- Ajuda na documentação e na aprendizagem
  - Conhecendo os padrões de projeto torna mais fácil a compreensão de sistemas existentes
  - "As pessoas que estão aprendendo POO freqüentemente reclamam que os sistemas com os quais trabalham usam herança de forma complexa e que é difícil de seguir o fluxo de controle. Geralmente a causa disto é que eles não entendem os padrões do sistema" [GoF]
  - Aprender os padrões ajudam um novato a agir mais como um especialista

Elementos de um padrão

- Nome
- Problema
  - Quando aplicar o padrão, em que condições?
- Solução
  - Descrição abstrata de um problema e como usar os elementos disponíveis (classes e objetos) para solucioná-lo
- Conseqüências
  - Custos e benefícios de se aplicar o padrão
  - Impacto na flexibilidade, extensibilidade, portabilidade e eficiência do sistema

Padrões GoF: 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

Padrões GRASP focam neste objetivo

Classificação dos 23 padrões segundo GoF*

		Propósito		
		1. Criação	2. Estrutura	3. Comportamento
Escopo	Classe	Factory Method	Class Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

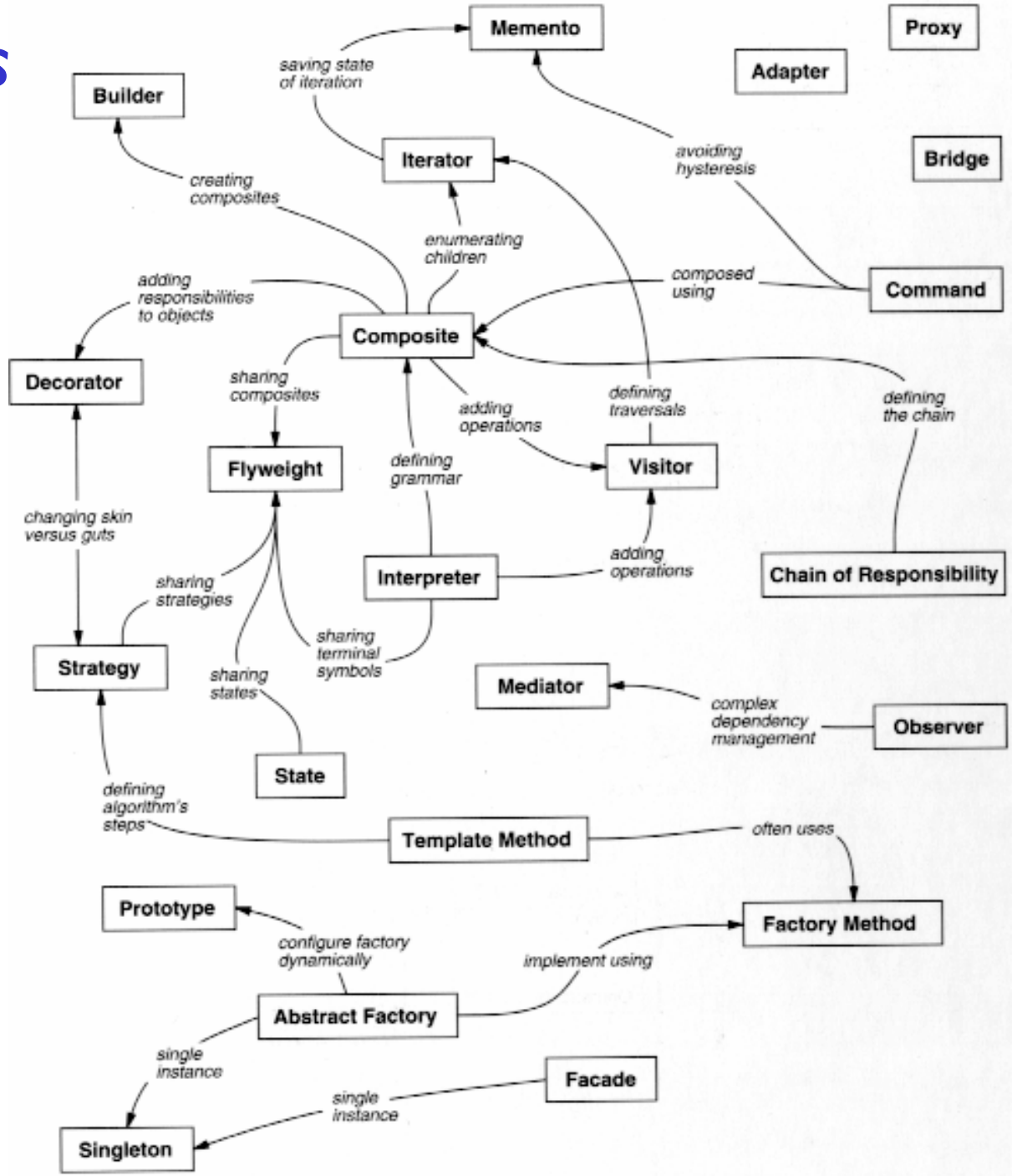
* Padrões "clássicos" selecionados e organizados por Gamma et al. "Design Patterns" [2]

Classificação dos padrões GoF segundo Metsker [1]

Intenção	Padrões
1. Interfaces	*Adapter, Facade, Composite, Bridge*
2. Responsabilidade	*Singleton, Observer, Mediator, Proxy, Chain of Responsibility, Flyweight*
3. Construção	*Builder, Factory Method, Abstract Factory, Prototype, Memento*
4. Operações	*Template Method, State, Strategy, Command, Interpreter*
5. Extensões	*Decorator, Iterator, Visitor*

- *Usaremos esta classificação*

Relacionamentos entre os 23 padrões "clássicos"



Fonte: [2]

Introdução: interfaces

- *Interface: coleção de métodos e dados que uma classe permite que objetos de outras classes acessem*
- *Implementação: código dentro dos métodos*
- *Interface Java: componente da linguagem que representa apenas a interface de um objeto*
  - *Exigem que classe que implementa a interface ofereça implementação para seus métodos*
  - *Não garante que métodos terão implementação que faça efetivamente alguma coisa (chaves vazias): stubs.*

Além das interfaces

- Padrões de design que concentram-se principalmente na adaptação de interfaces
  - Adapter: para adaptar a interface de uma classe para outra que o cliente espera
  - Façade: oferecer uma interface simples para uma coleção de classes
  - Composite: definir uma interface comum para objetos individuais e composições de objetos
  - Bridge: desacoplar uma abstração de sua implementação para que ambos possam variar independentemente

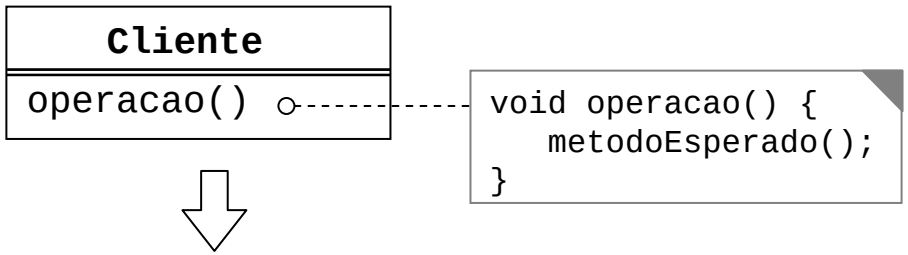
1

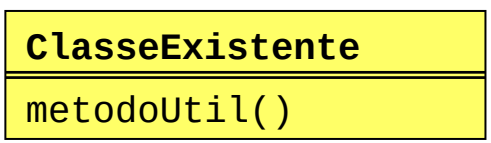
Adapter

"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]

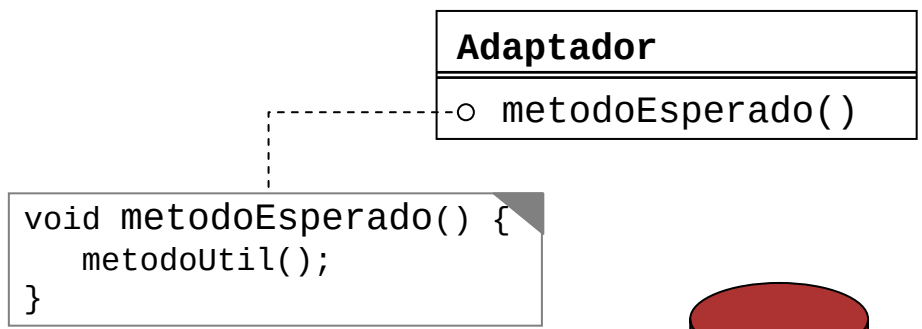
Problema e Solução

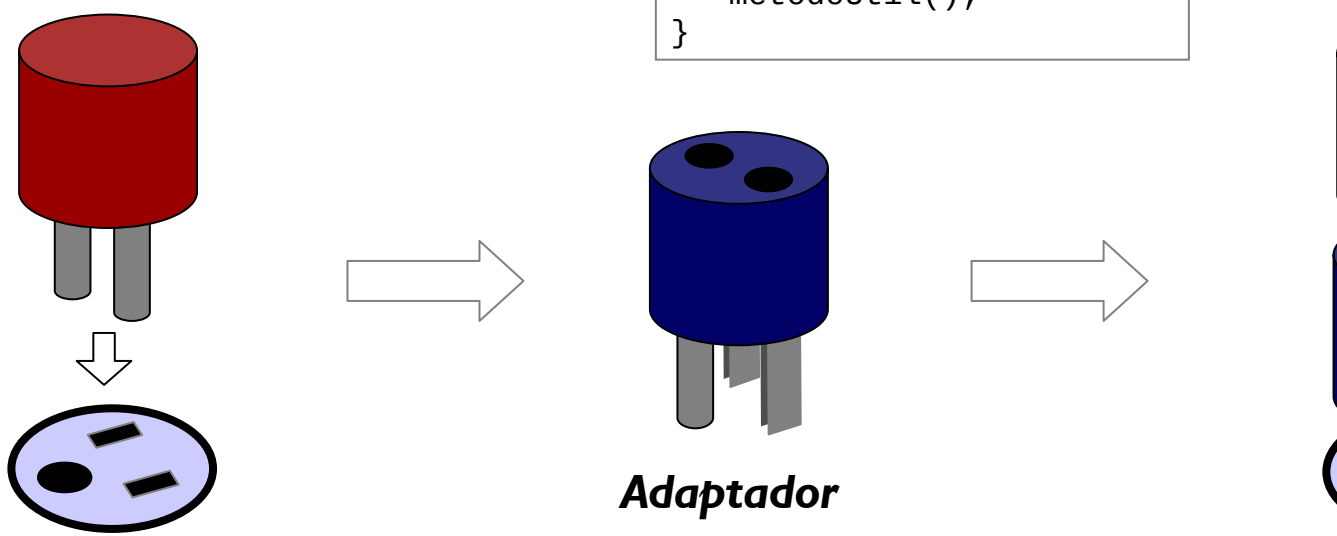
Problema





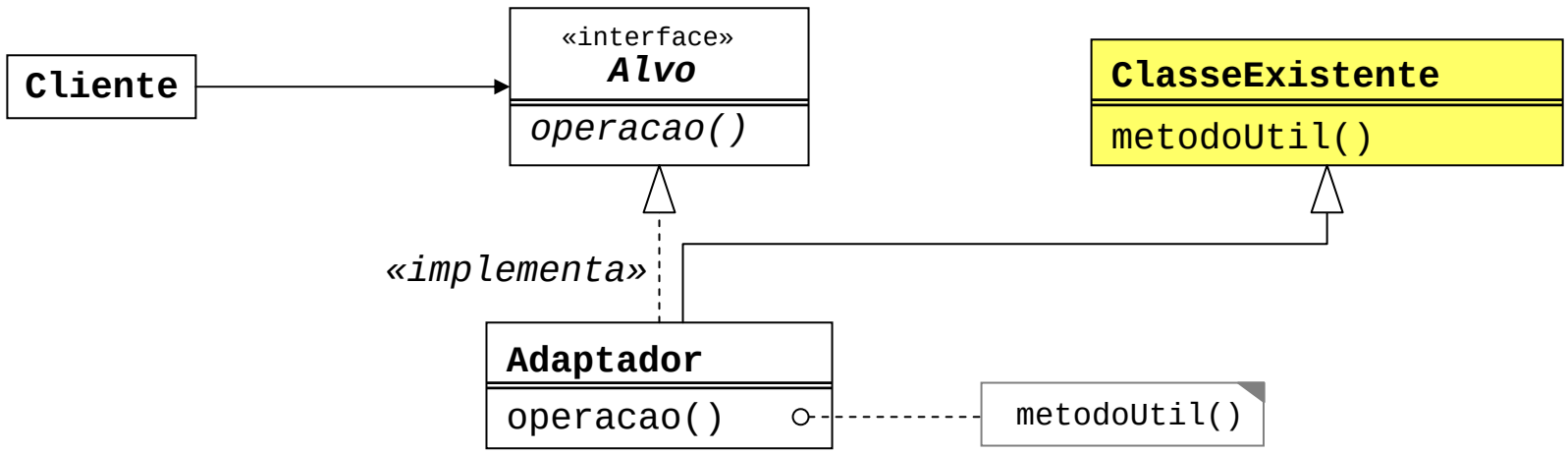
Solução





Duas formas de Adapter

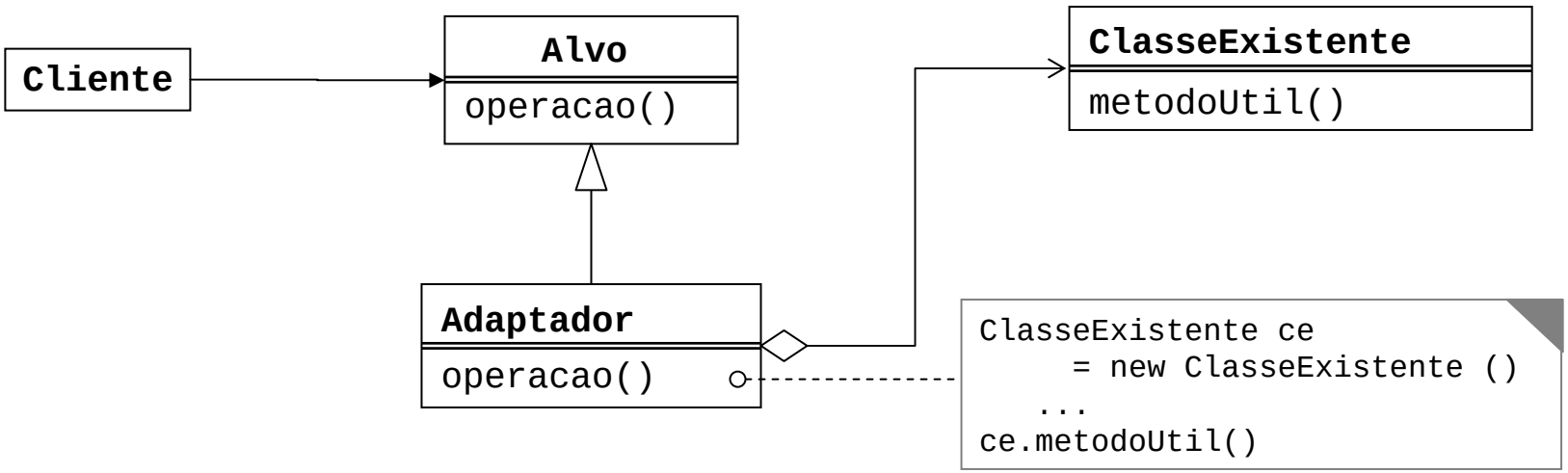
- *Class Adapter: usa herança múltipla*



- *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.*

Class 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++) {
            alvo.operacao();
        }
    }
}
```

```
public interface Alvo {
    void operacao();
}
```

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

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

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();
    }
}
```

Quando usar?

- Sempre que for necessário adaptar uma interface para um cliente
- Class Adapter
  - Quando houver uma interface que permita a implementação estática
- Object Adapter
  - Quando menor acoplamento for desejado
  - Quando o cliente não usa uma interface Java ou classe abstrata que possa ser estendida

Onde estão os adapters?

- *A API do J2SDK tem vários adapters*
- *Você consegue identificá-los?*

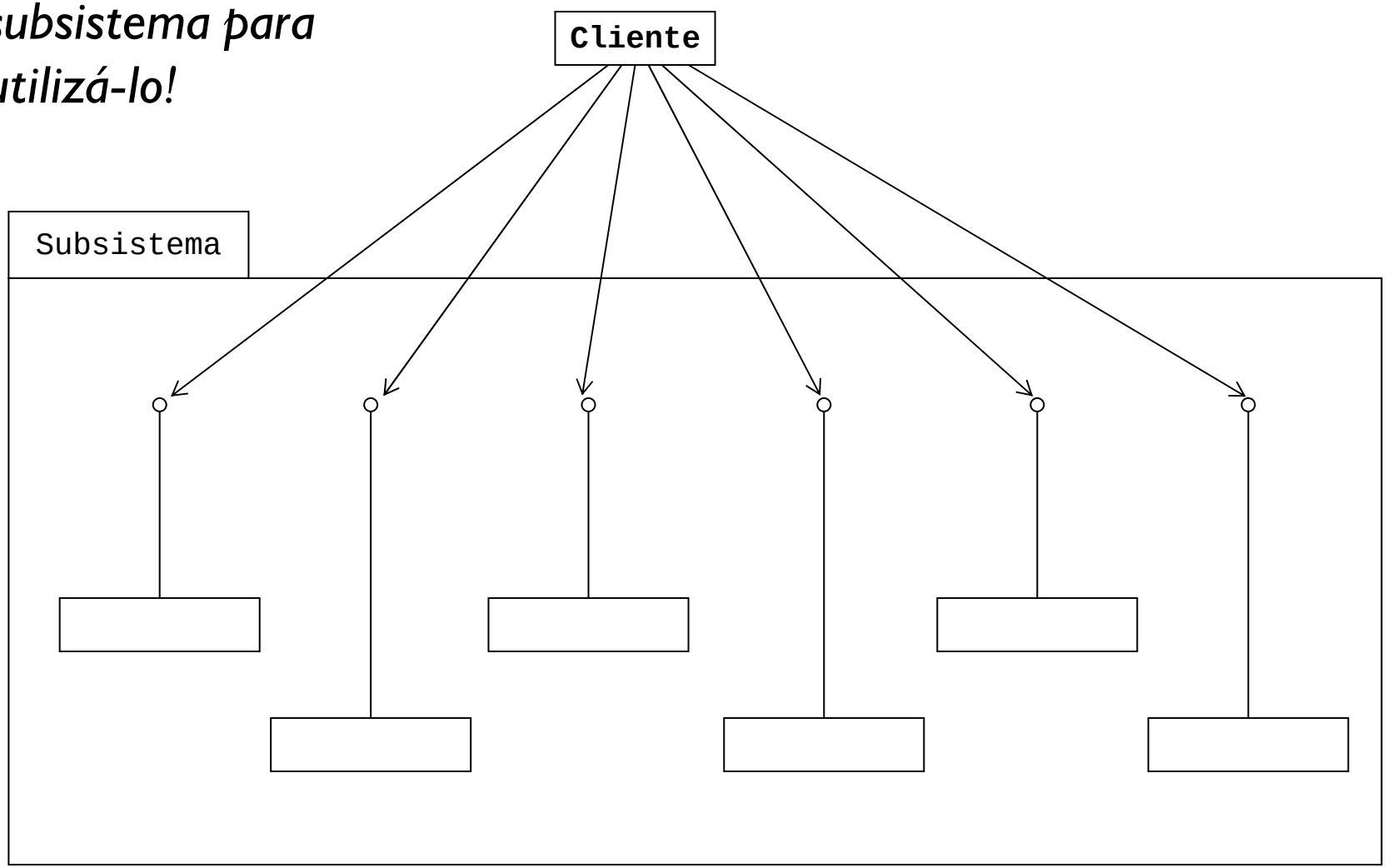
2

Façade

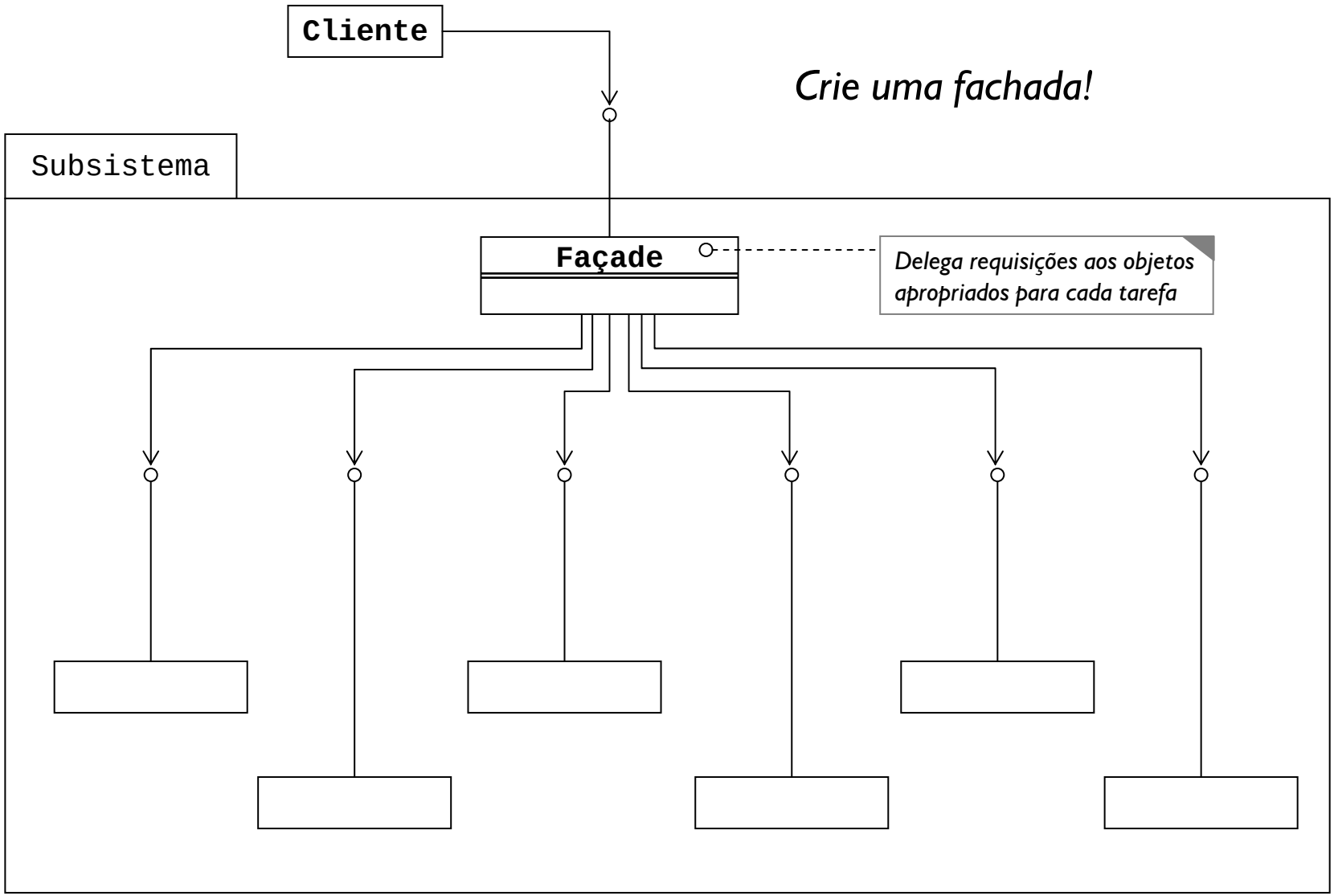
"Oferecer uma interface única para um conjunto de interfaces de um subsistema. Façade define uma interface de nível mais elevado que torna o subsistema mais fácil de usar." [GoF]

Problema

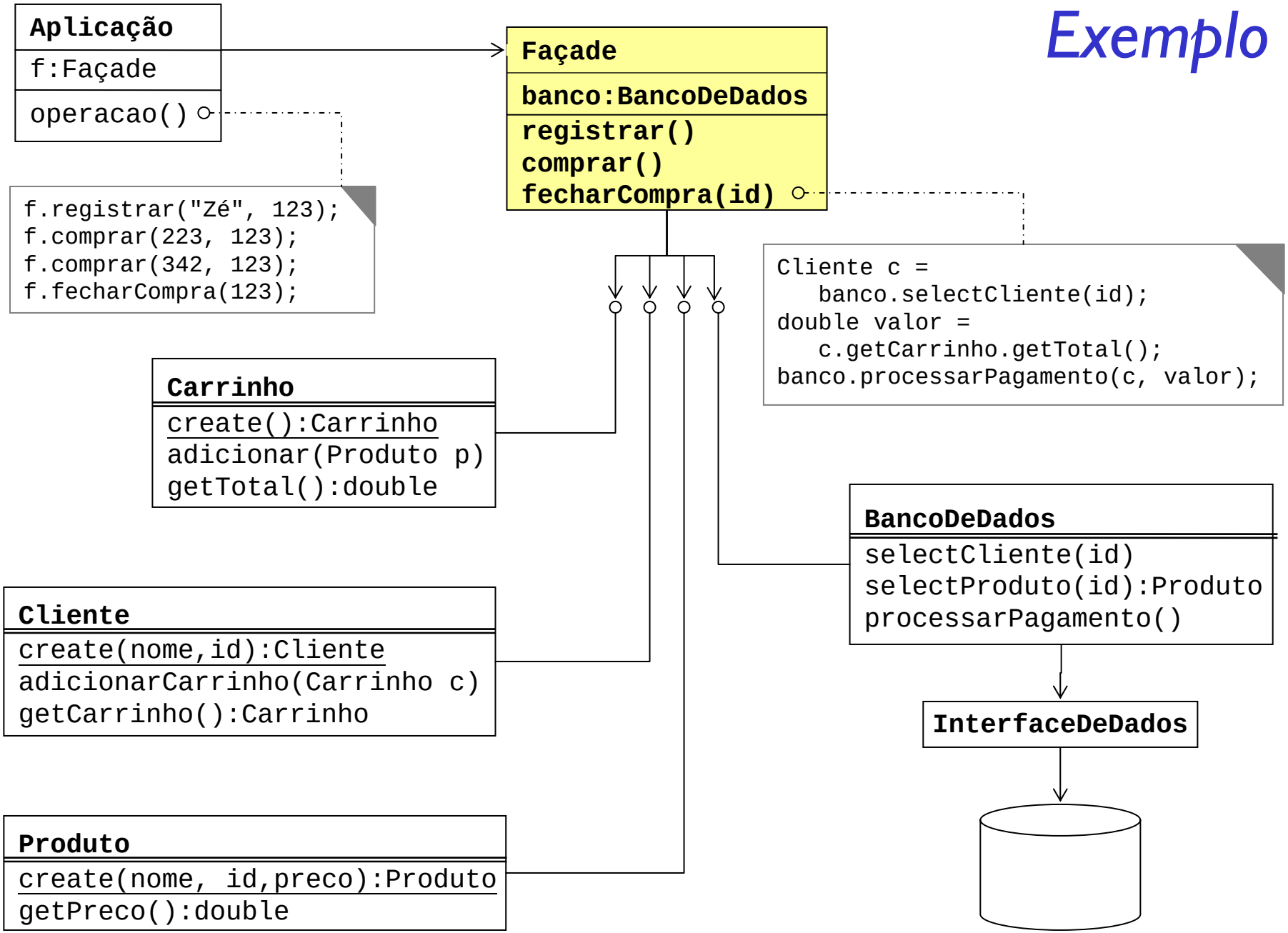
Cliente precisa saber muitos detalhes do subsistema para utilizá-lo!



Estrutura de Façade



Exemplo


Aplicação
f:Façade
operacao()
Façade
banco:BancoDeDados
registrar()
comprar()
fecharCompra(id)
f.registrar("Zé", 123);
f.comprar(223, 123);
f.comprar(342, 123);
f.fecharCompra(123);
Cliente c =
banco.selectCliente(id);
double valor =
c.getCarrinho.getTotal();
banco.processarPagamento(c, valor);
Carrinho
create():Carrinho
adicionar(Produto p)
getTotal():double
BancoDeDados
selectCliente(id)
selectProduto(id):Produto
processarPagamento()
Cliente
create(nome,id):Cliente
adicionarCarrinho(Carrinho c)
getCarrinho():Carrinho
InterfaceDeDados
Produto
create(nome, id,preco):Produto
getPreco():double

Façade em Java

```
class Aplicação {
   ...
   Facade f;
   // Obtem instancia f
   f.registrar("Zé", 123);
   f.comprar(223, 123);
   f.comprar(342, 123);
   f.fecharCompra(123);
   ...
}
```

```
public class Facade {
  BancoDeDados banco = Sistema.obterBanco();
  public void registrar(String nome, int id) {
     Cliente c = Cliente.create(nome, id);
     Carrinho c = Carrinho.create();
     c.adicionarCarrinho();
  }
  public void comprar(int prodID, int clienteID) {
     Cliente c = banco.selectCliente(cliente ID);
     Produto p = banco.selectProduto(prodID) {
     c.getCarrinho().adicionar(p);
   }
   public void fecharCompra(int clienteID) {
      Cliente c = banco.selectCliente(clienteID);
      double valor = c.getCarrinho.getTotal();
      banco.processarPagamento(c, valor);
   }
}
```

```
public class Carrinho {
  static Carrinho create() {...}
  void adicionar(Produto p) {...}
  double getTotal() {...}
}
```

```
public class Produto {
  static Produto create(String nome,
                        int id,double preco) {...}
  double getPreco() {...}
}
```

```
public class Cliente {
 static Cliente create(String nome,
                       int id) {...}
 void adicionarCarrinho(Carrinho c) {...}
 Carrinho getCarrinho() {...}
}
```

```
public class BancoDeDados {
 Cliente selectCliente(int id) {...}
 Produto selectProduto(int id) {...}
 void processarPagamento() {...}
}
```

Quando usar?

- *Sempre que for desejável criar uma interface para um conjunto de objetos com o objetivo de facilitar o uso da aplicação*
  - *Permite que objetos individuais cuidem de uma única tarefa, deixando que a fachada se encarregue de divulgar as suas operações*
- *Fachadas viabilizam a separação em camadas com alto grau de desacoplamento*
- *Existem em várias partes da aplicação*
  - *Fachada da aplicação para interface do usuário*
  - *Fachada para sistema de persistência: Data Access Object*

Nível de acoplamento

- *Fachadas podem oferecer maior ou menor isolamento entre aplicação cliente e objetos*
  - *Nível ideal deve ser determinado pelo nível de acoplamento desejado entre os sistemas*
- *A fachada mostrada como exemplo isola totalmente o cliente dos objetos*

```
Facade f; // Obtem instancia f
f.registrar("Zé", 123);
```

- *Outra versão com menor isolamento (requer que aplicação-cliente conheça objeto Cliente)*

```
Cliente joao = Cliente.create("João", 15);
f.registrar(joao); // método registrar(Cliente c)
```

Questões

- *Você conhece algum exemplo de na API do J2SDK*
- *Onde você criaria fachadas em uma aplicação Web J2EE?*
- *Você saberia dizer qual a diferença entre Façade e Adapter*

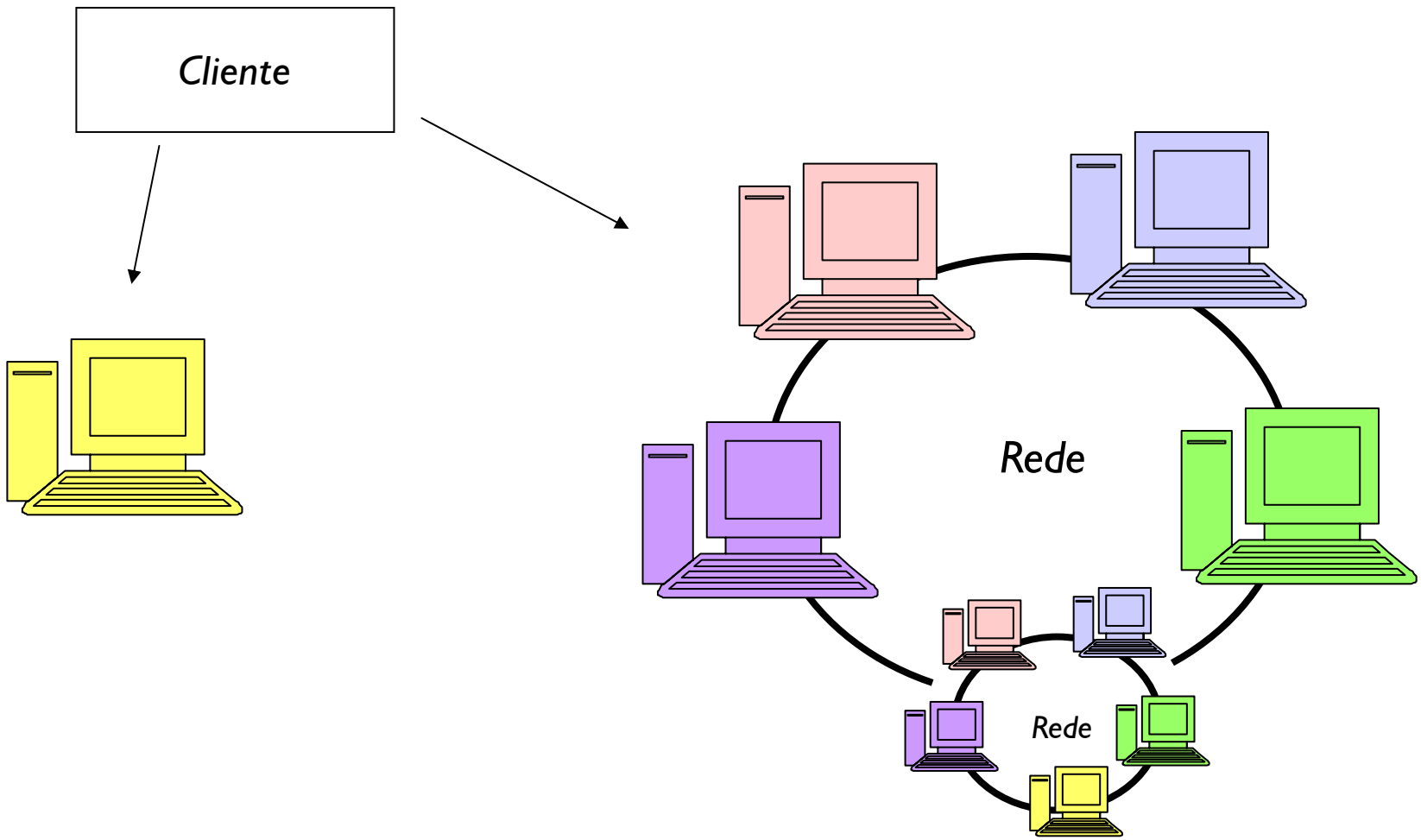
3

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]

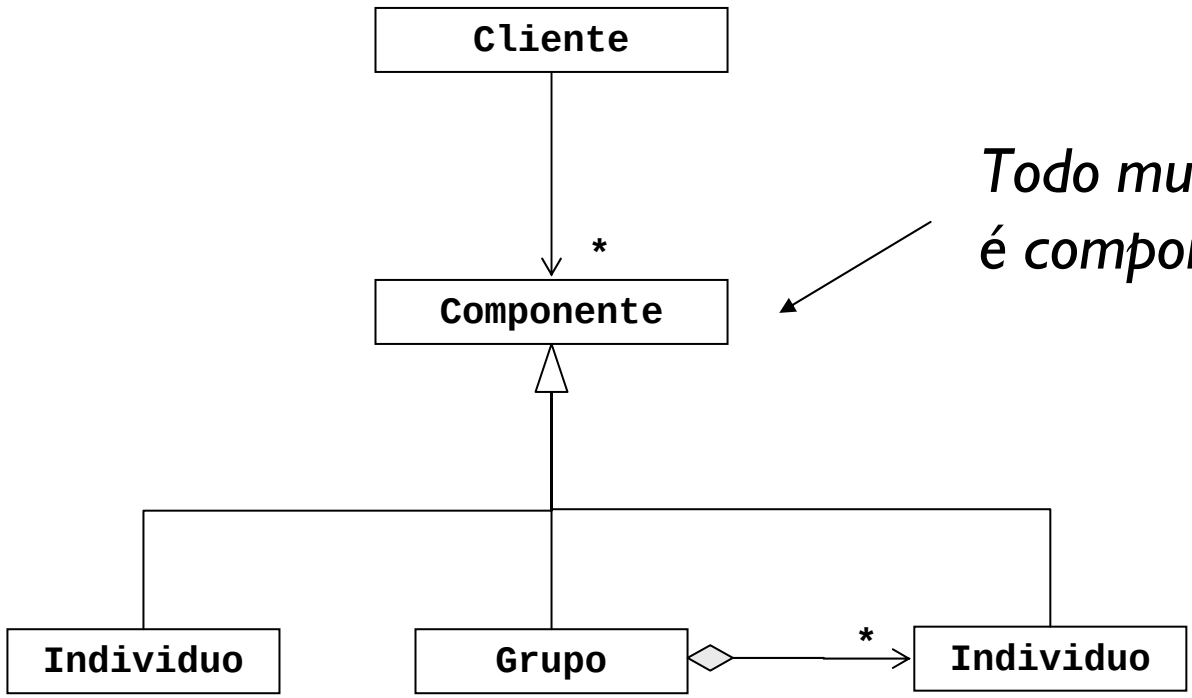
Problema

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

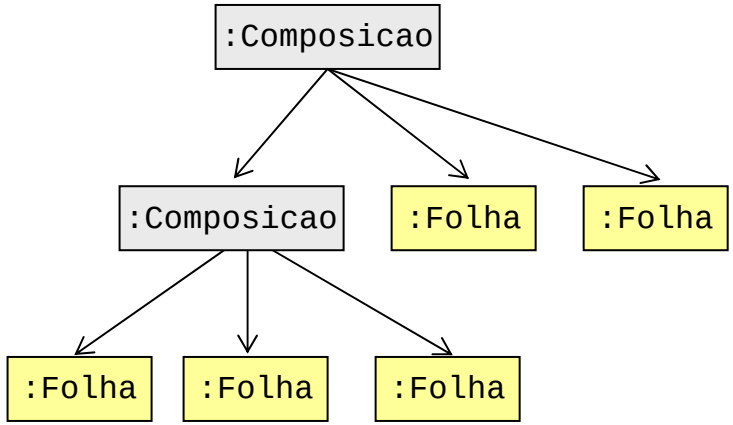


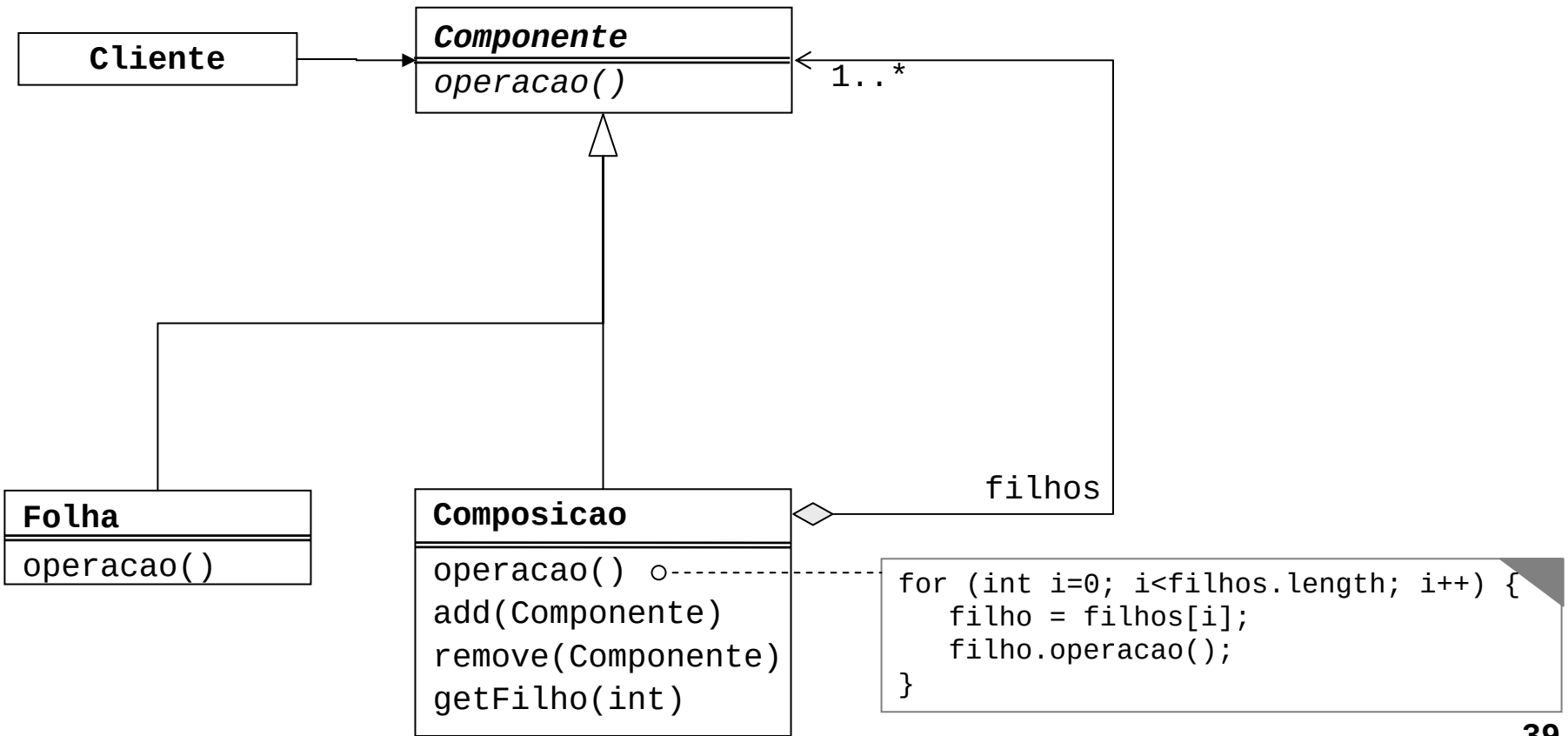
Tratar grupos e indivíduos diferentes através de uma única interface

Solução



Estrutura de Composite





Composite em Java

```
import java.util.*;
public class MachineComposite extends MachineComponent {
   protected List components = new ArrayList();
   public void add(MachineComponent component) {
      components.add(component);
   }
   public int getMachineCount() {
       // Exercício
   }
}
```

```
public abstract class MachineComponent {
   public abstract int getMachineCount();
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
}
```

O que colocar no lugar indicado para implementar corretamente esta classe?

```
public abstract class Machine extends MachineComponent {
   public int getMachineCount() {
      // Exercício
   }
}
```

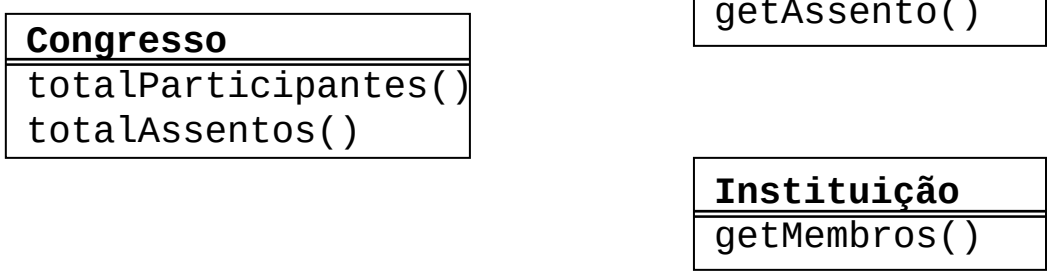
Fonte: [2]

Quando usar?

- *Sempre que houver necessidade de tratar um conjunto como um indivíduo*
- *Funciona melhor se relacionamentos entre os objetos for uma árvore*
  - *Caso o relacionamento contenha ciclos, é preciso tomar precauções adicionais para evitar loops infinitos, já que Composite depende de implementações recursivas*
- *Há várias estratégias de implementação*

Questões

- *Cite exemplos de Composite*
  - *Na API do J2SDK*
  - *Em frameworks que você conhece*
  - *Em aplicações que você conhece*
- *Como poderia ser resolvido o problema abaixo?*
  - *É preciso saber quantas pessoas irão participar do congresso*
  - *Participantes podem ser pessoas ou instituição*



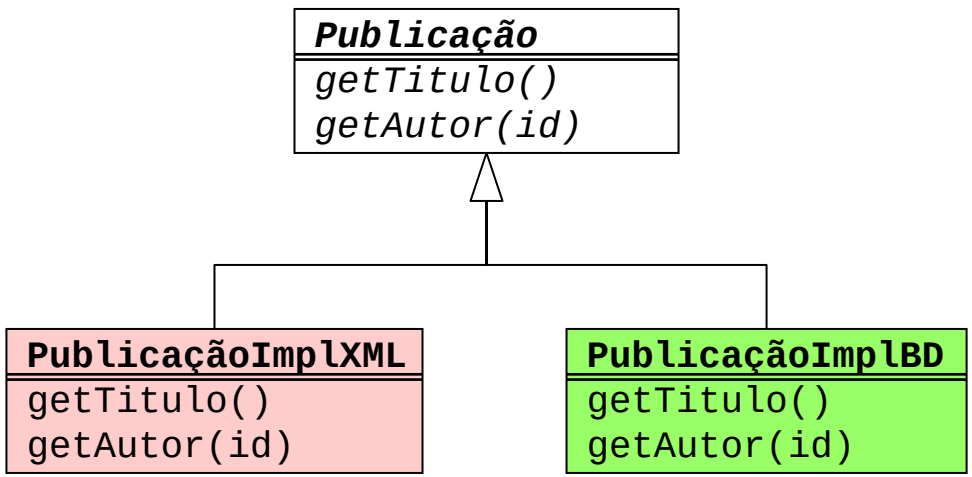
4

Bridge

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

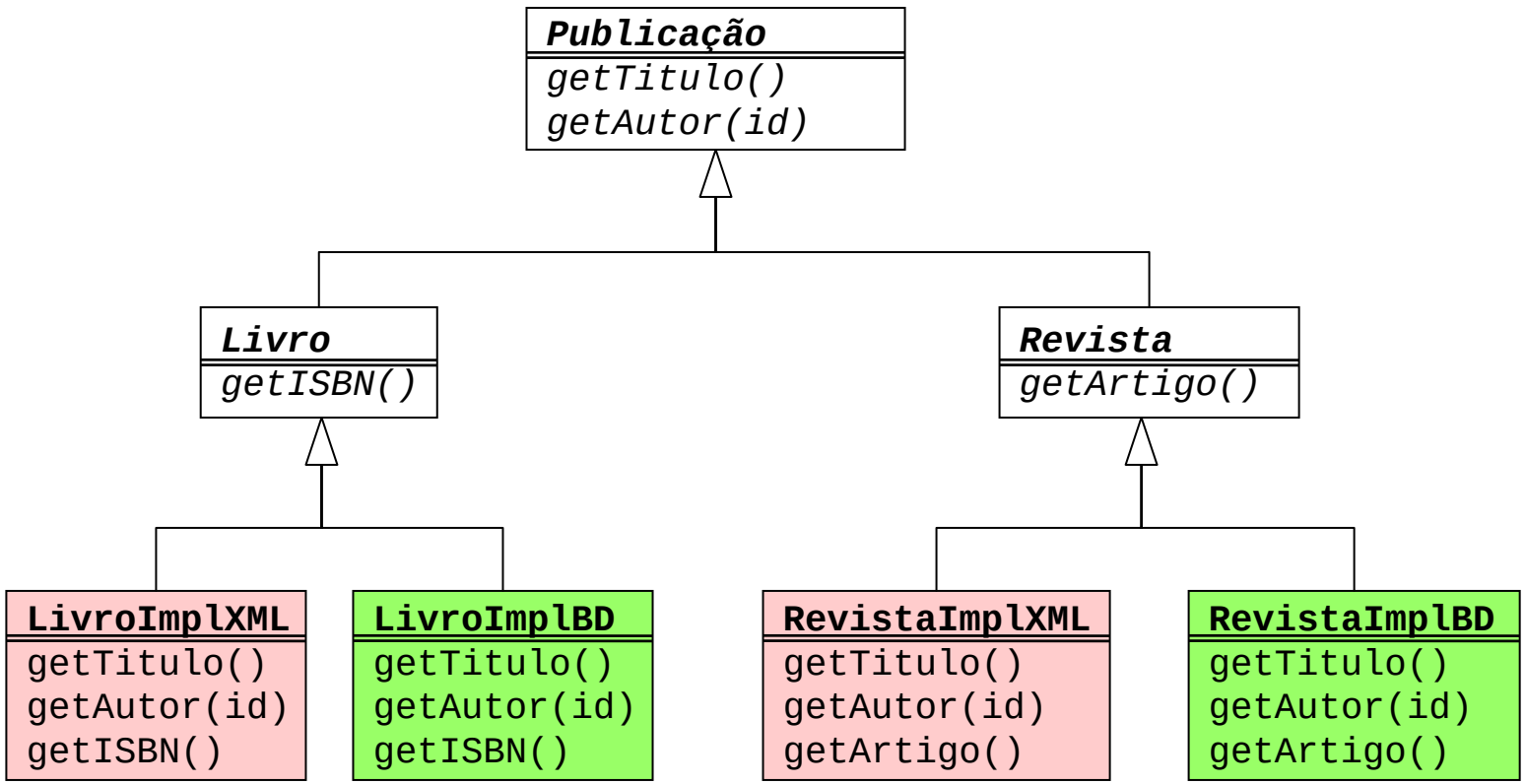
Problema (1)

- *Necessidade de um driver*
- *Exemplo: implementações específicas para tratar objeto em diferentes meios persistentes*

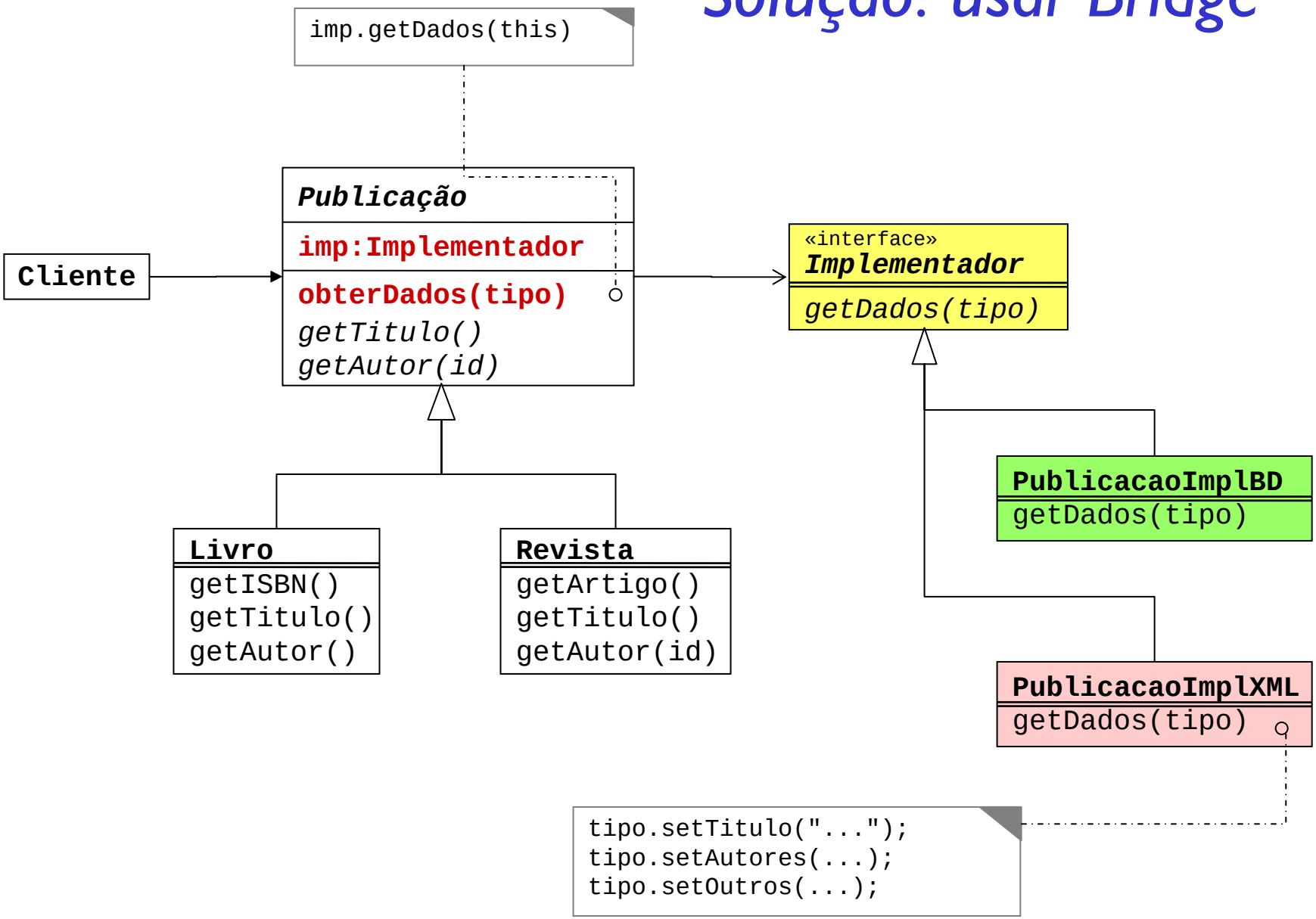


Problema (II)

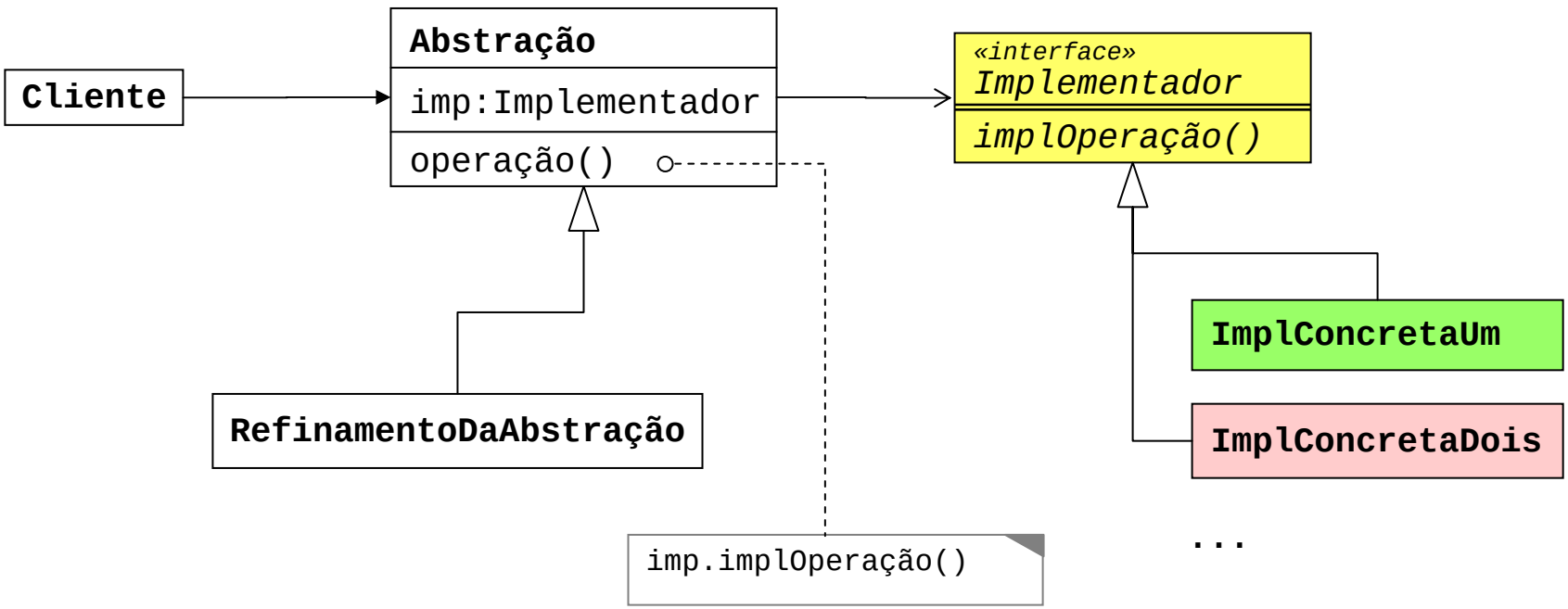
- *Mas herança complica a implementação*



Solução: usar Bridge



Estrutura de Bridge



A Abstração define operações de nivel mais elevado baseadas nas operações primitivas do Implementador

A interface Implementador define operações primitivas

Quando usar?

- Quando for necessário evitar uma ligação permanente entre a interface e implementação
- Quando alterações na implementação não puderem afetar clientes
- Quando tanto abstrações como implementações precisarem ser capazes de suportar extensão através de herança
- Quando implementações são compartilhadas entre objetos desconhecidos do cliente

Questões

- *Você conhece alguma implementação de Bridge na API do J2SDK?*

Resumo: quando usar?

- Adapter
  - Adaptar uma interface existente para um cliente
- Bridge
  - Implementar um design que permita total desacoplamento entre interface e implementação
- Façade
  - Simplificar o uso de uma coleção de objetos
- Composite
  - Tratar composições e unidades uniformemente

Distribuição de responsabilidades

- Os padrões a seguir estão principalmente associados a atribuições especiais de responsabilidade
  - Singleton: centraliza a responsabilidade em uma única instância de uma classe
  - Observer: desacopla um objeto do conhecimento de que outros objetos dependem dele
  - Mediator: centraliza a responsabilidade em uma classe que determina como outros objetos interagem
  - Proxy: assume a responsabilidade de outro objeto (intercepta)
  - Chain of Responsibility: permite que uma requisição passe por uma corrente de objetos até encontrar um que a processe
  - Flyweight: centraliza a responsabilidade em objetos compartilhados de alta granularidade (blocos de montagem)

5

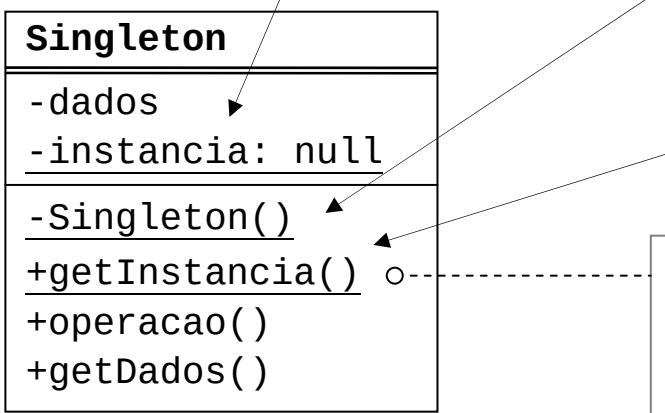
Singleton

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

Problema

- *Garantir que apenas um objeto exista, independente do número de requisições que receber para criá-lo*
- *Aplicações*
  - *Um único banco de dados*
  - *Um único acesso ao arquivo de log*
  - *Um único objeto que representa um vídeo*
  - *Uma única fachada (Façade pattern)*
- *Objetivo: garantir que uma classe só tenha uma instância*

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*

(omitido por falta de espaço!)*

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

Esta classe cria apenas um objeto Highlander

```
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!");
    }
  }
}
```

Prós e contras

- Vantagens
  - Acesso central e extensível a recursos e objetos
- Desvantagens
  - Qualidade da implementação depende da linguagem
  - Difícil de testar (simulações dependem de instância extra)
  - Uso (abuso) como substituto para variáveis globais
  - Inicialização lazy "preguiçosa" é complicada em ambiente multithreaded (é um anti-pattern – veja a seguir)
  - Difícil ou impossível de implementar em ambiente distribuído (é preciso garantir que cópias serializadas refiram-se ao mesmo objeto)

Double-checked lazy Singleton anti-pattern: Não use!

[5] [6]

- *Esse famoso padrão é um truque para suportar inicialização lazy evitando o overhead da sincronização*
  - *Parece uma solução inteligente (evita sincronização no acesso)*
  - *Mas não funciona! Inicialização de resource (null) e instanciamento podem ser reordenados no cache*

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

Alternativas

- *Não usar lazy 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 sincronizacão 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.

Fonte: [5]

Questões

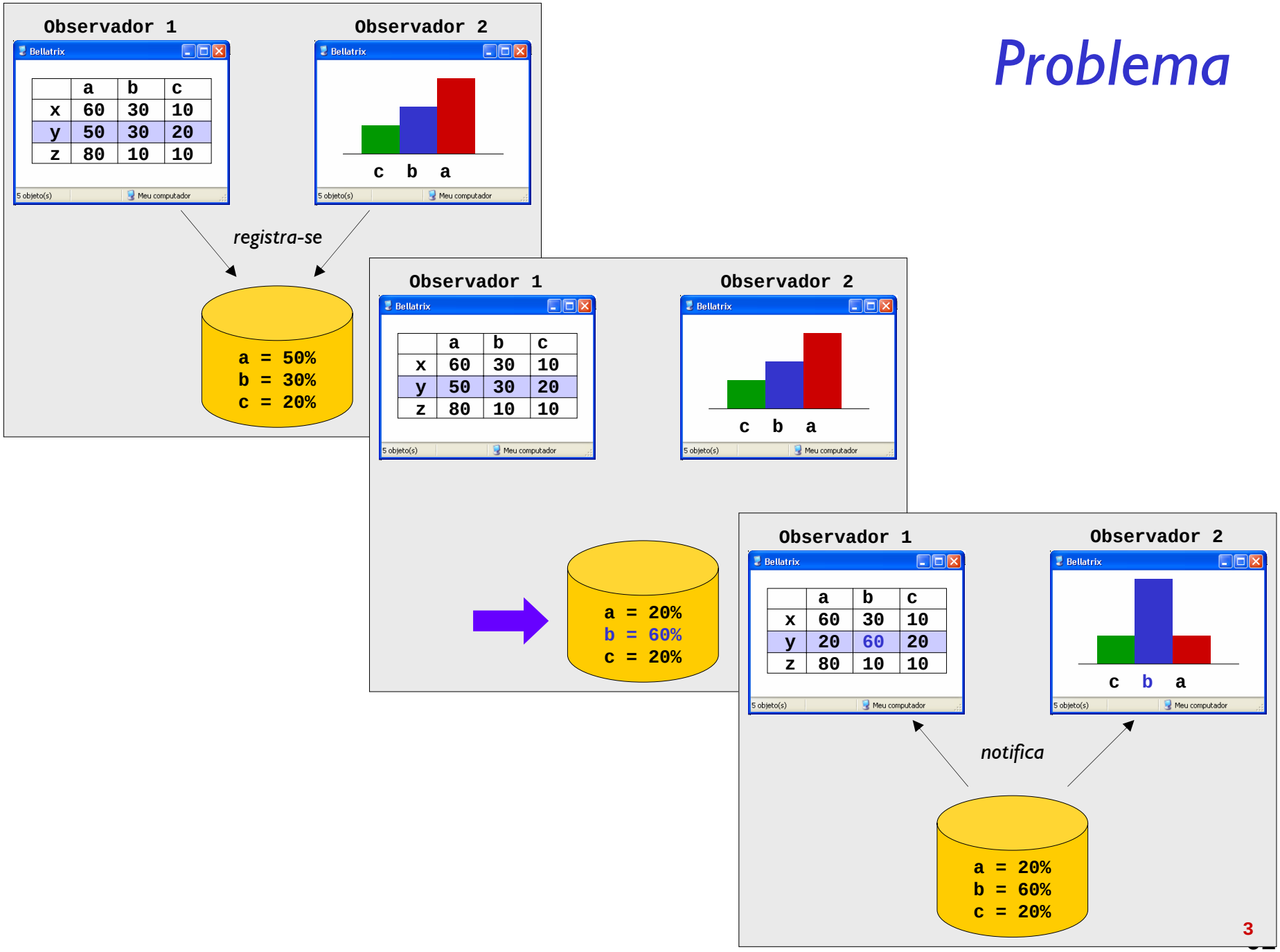
- *Cite aplicações onde seria interessante usar um singleton?*
- *Questões extremas*
  - *Todos os objetos deveriam ser Singletons?*
  - *Singletons são maus e não deviam ser usados?*

6

Observer

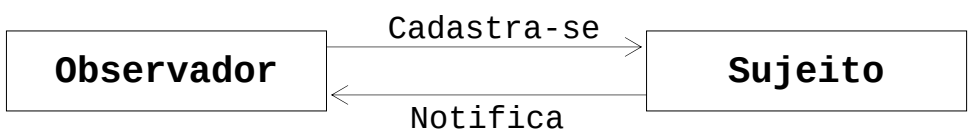
"Definir uma dependência um-para-muitos entre objetos para que quando um objeto mudar de estado, todos os seus dependentes sejam notificados e atualizados automaticamente." [GoF]

Problema

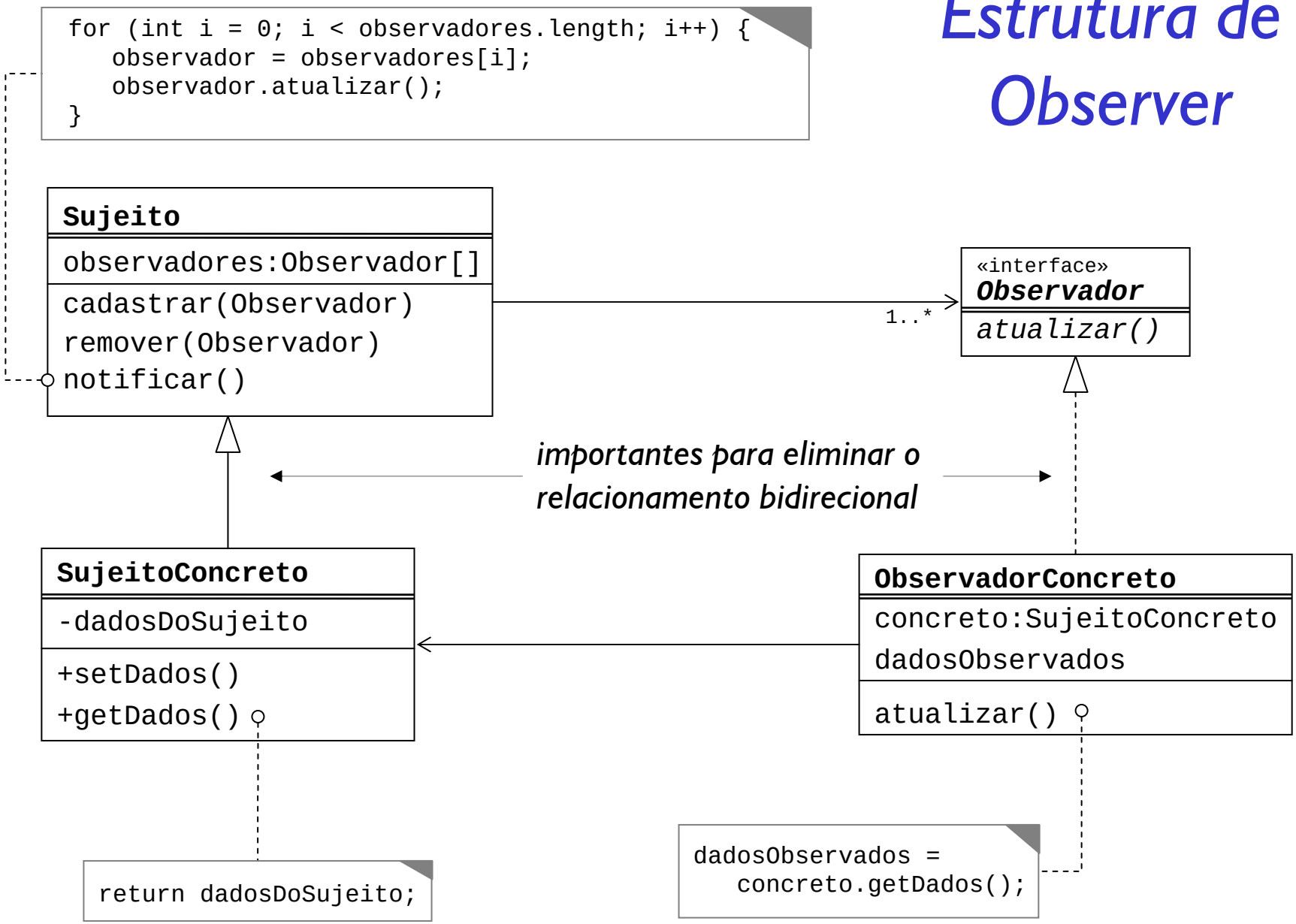

Observador 1
Observador 2
Bellatrix
a b c
x 60 30 10
y 50 30 20
z 80 10 10
c b a
5 objeto(s)
Meu computador
registra-se
a = 50%
b = 30%
c = 20%
a = 20%
b = 60%
c = 20%
notifica

Problema (2)

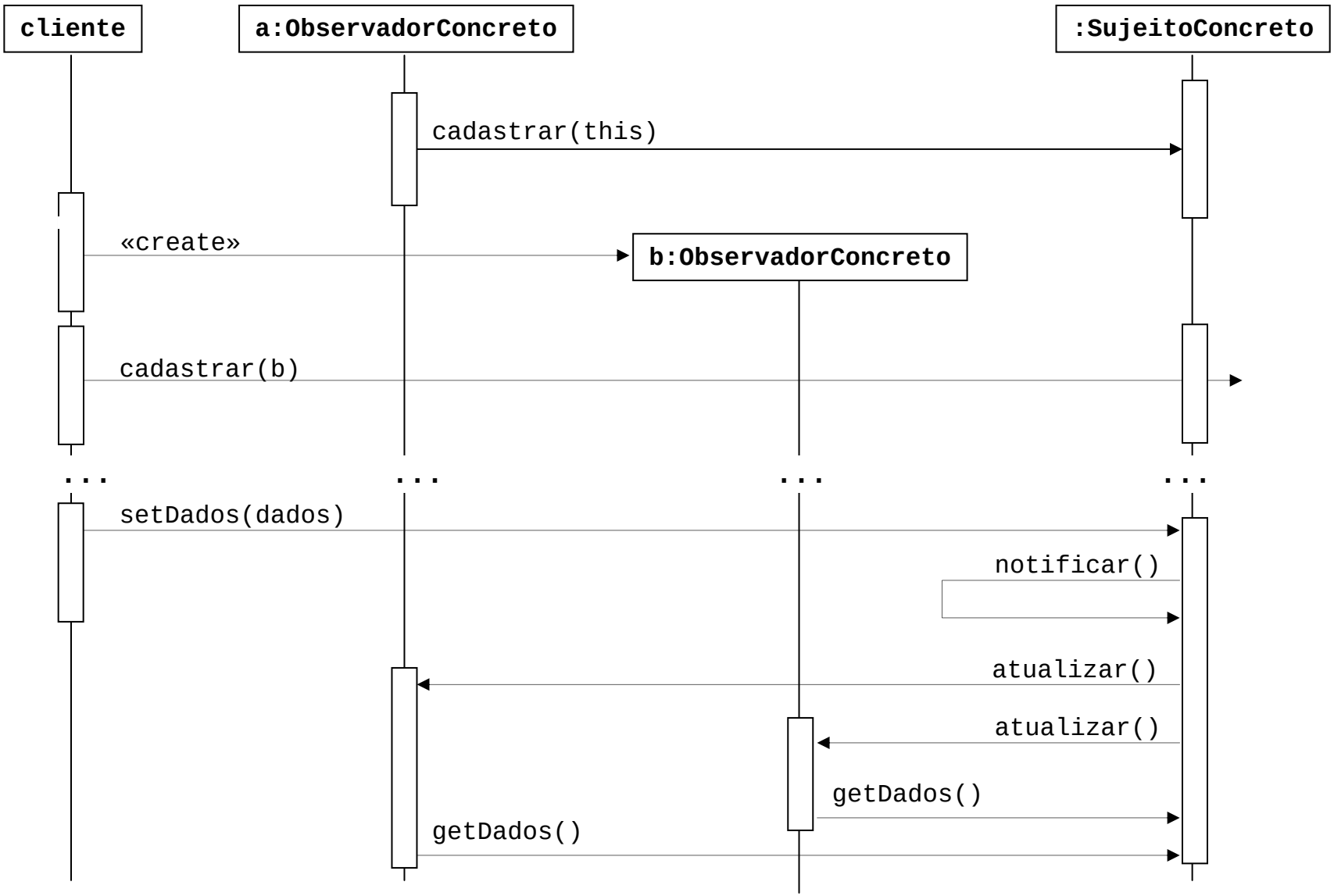
- *Como garantir que objetos que dependem de outro objeto fiquem em dia com mudanças naquele objeto?*
  - *Como fazer com que os observadores tomem conhecimento do objeto de interesse?*
  - *Como fazer com que o objeto de interesse atualize os observadores quando seu estado mudar?*
- *Possíveis riscos*
  - *Relacionamento (bidirecional) implica alto acoplamento. Como podemos eliminar o relacionamento bidirecional?*



Estrutura de Observer



Seqüência de Observer



Observer em Java

```
public class ConcreteObserver
             implements Observer {

  public void update(Observable o) {
    ObservableData data = (ObservableData) o;
    data.getData();
  }
}
```

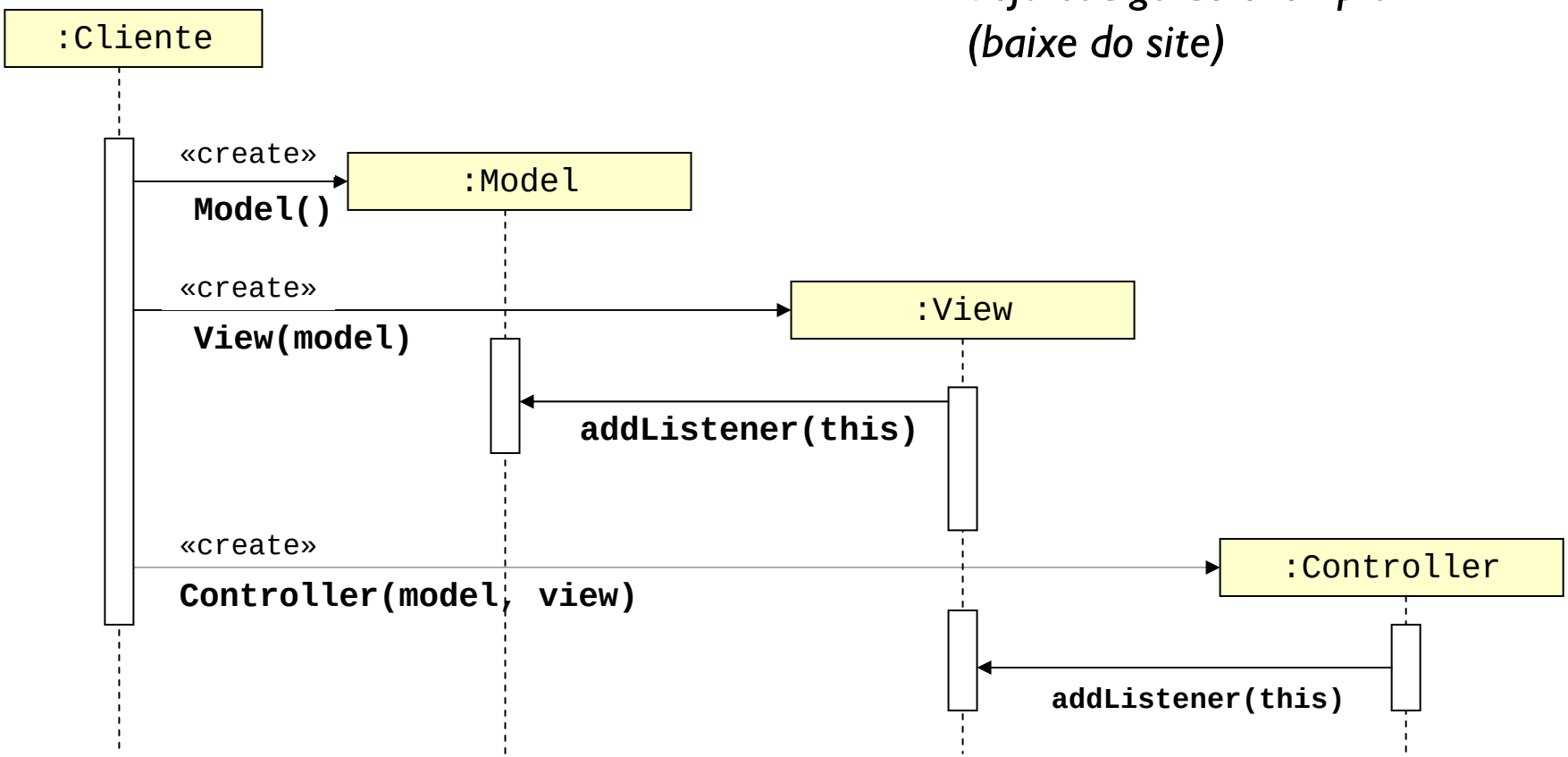
```
public class Observable {
  List observers = new ArrayList();

  public void add(Observer o) {
    observers.add(o);
  }

  public void remove(Observer o) {
    observers.remove(o);
  }

  public void notify() {
    Iterator it = observers.iterator();
    while(it.hasNext()) {
      Observer o = (Observer)it.next();
      o.update(this);
    }
  }
}
```

```
public class ObservableData
             extends Observable {
  private Object myData;

  public void setData(Object myData) {
    this.myData = myData;
    notify();
  }

  public Object getData() {
    return myData();
  }
}
```

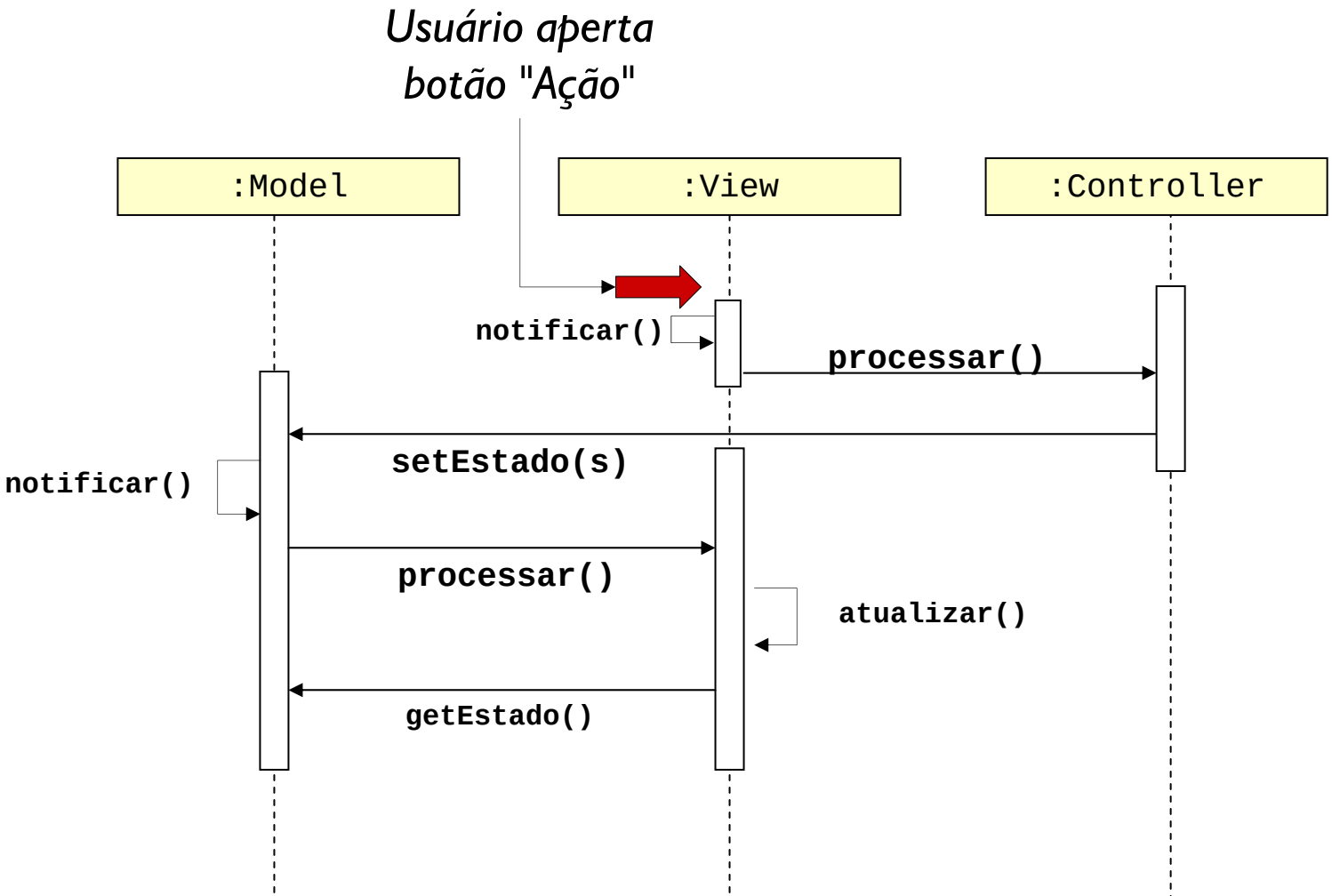
```
public interface Observer {
  public void update(Observable o);
}
```

Seqüência de registro das ligações

Veja código de exemplo (baixe do site)



Seqüência de operação



Questões

- *Cite exemplos de implementações de Observer e MVC*
  - *Na API Java*
  - *Em frameworks que você conhece*

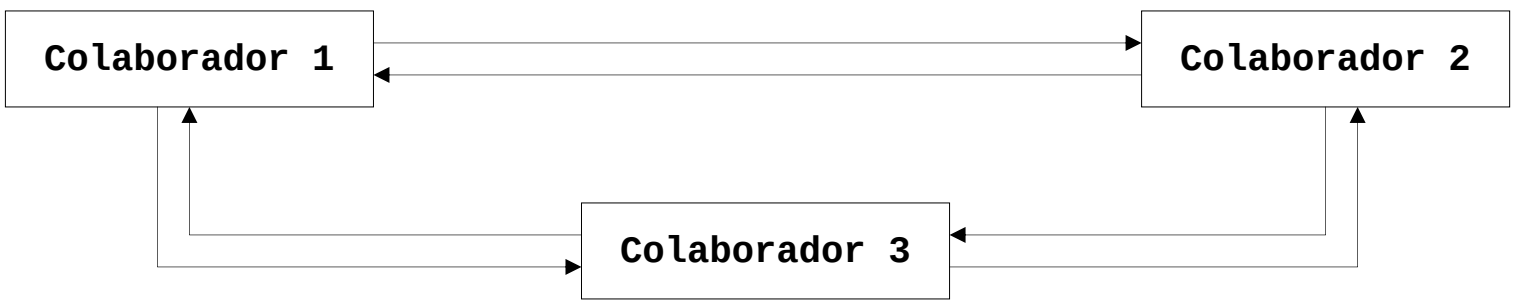
7

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]

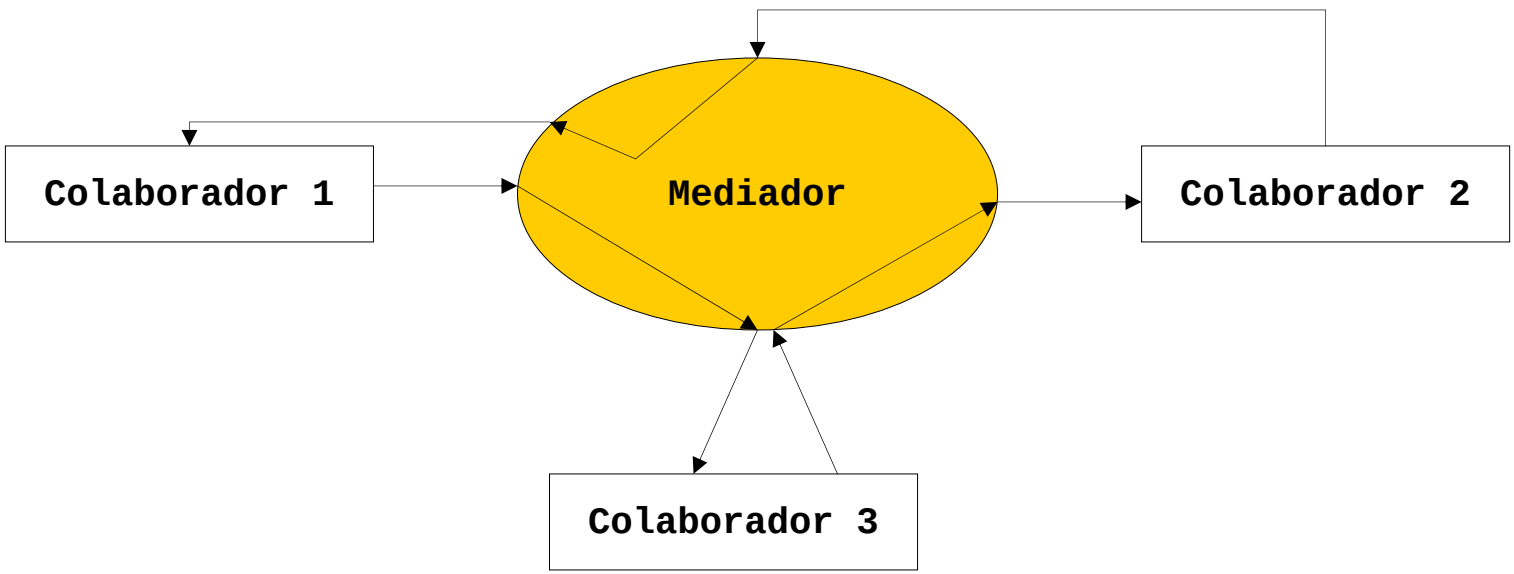
Problema

- *Como permitir que um grupo de objetos se comunique entre si sem que haja acoplamento entre eles?*
- *Como remover o forte acoplamento presente em relacionamentos muitos para muitos?*
- *Como permitir que novos participantes sejam ligados ao grupo facilmente?*

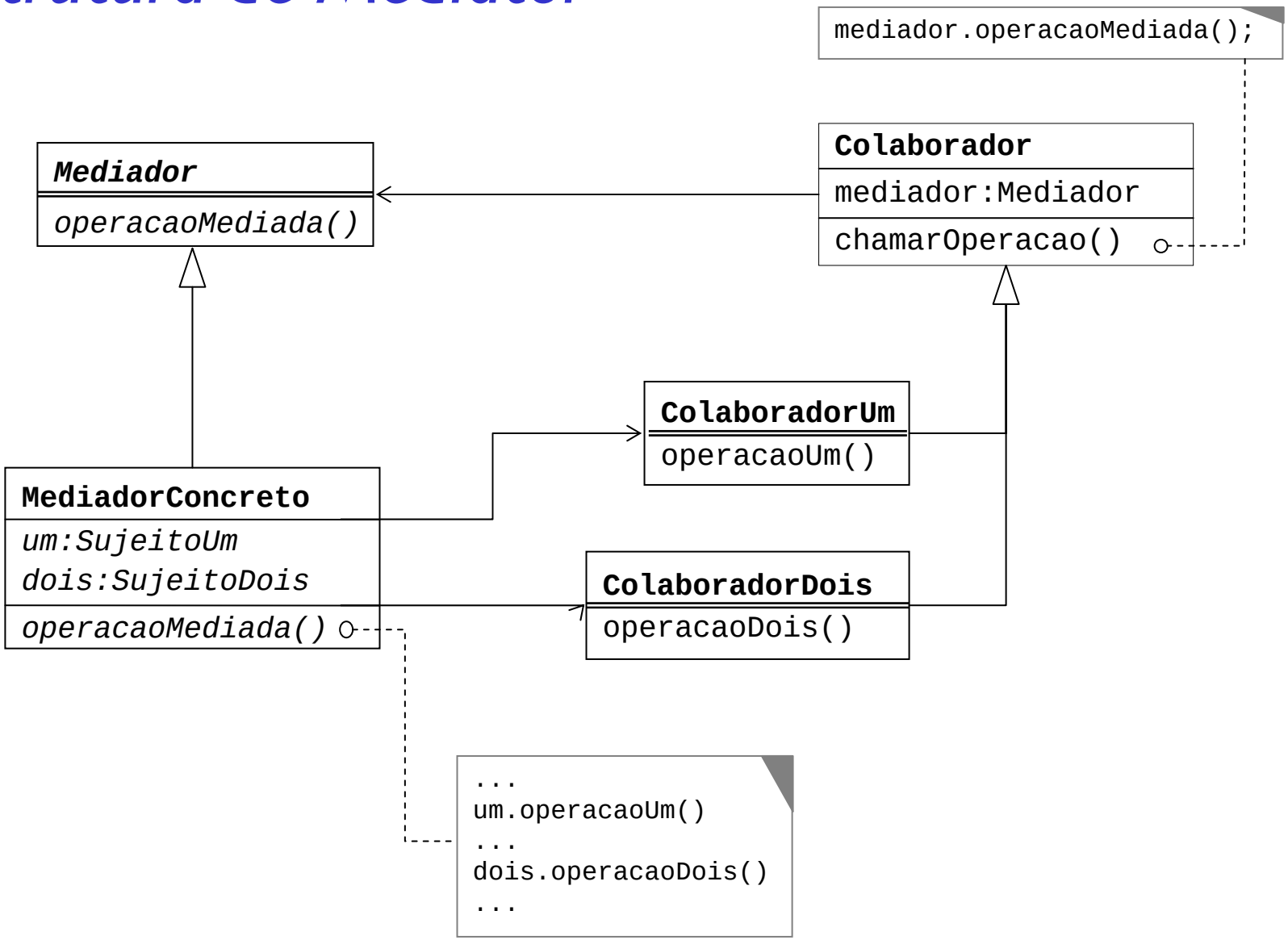


Solução

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



Estrutura de Mediator



Descrição da solução

- *Um objeto Mediador deve encapsular toda a comunicação entre um grupo de objetos*
  - *Cada objeto participante conhece o mediador mas ignora a existência dos outros objetos*
  - *O mediador conhece cada um dos objetos participantes*
- *A interface do Mediador é usada pelos colaboradores para iniciar a comunicação e receber notificações*
  - *O mediador recebe requisições dos remetentes*
  - *O mediador repassa as requisições aos destinatários*
  - *Toda a política de comunicação é determinada pelo mediador (geralmente através de uma implementação concreta do mediador)*

Questões

- *Onde você usaria mediadores?*
- *Você conhece exemplos de mediadores na API Java? Em frameworks?*

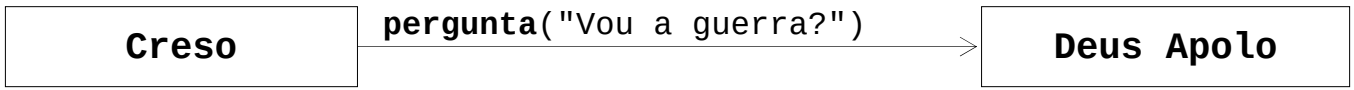
8

Proxy

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

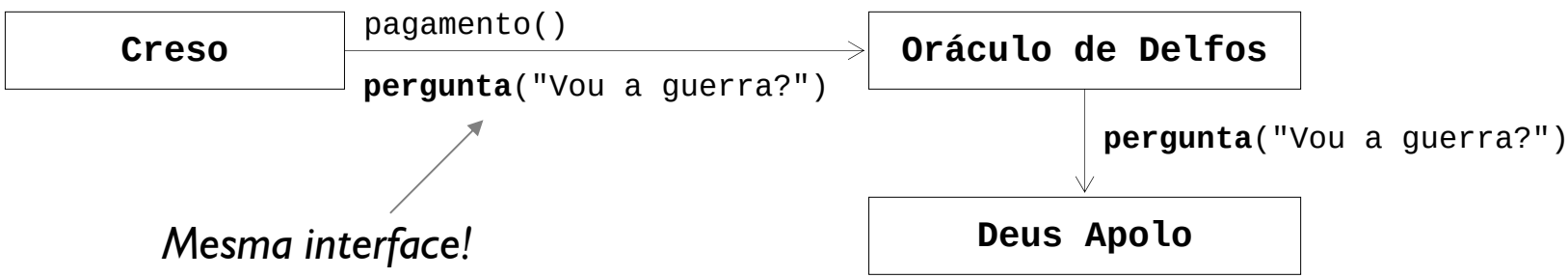
Problema

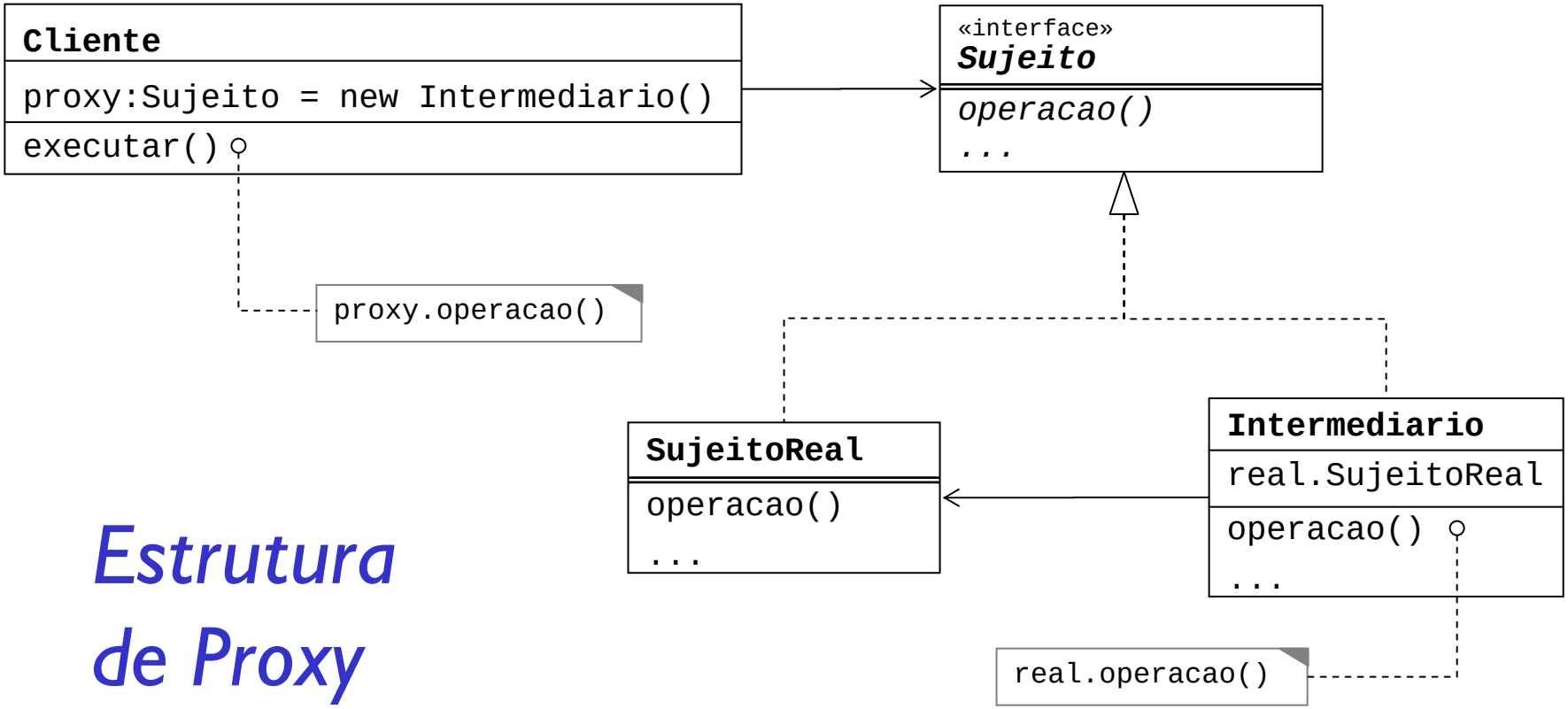
- *Sistema quer utilizar objeto real...*



- *Mas ele não está disponível (remoto, inaccessível, ...)*

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





Estrutura de Proxy

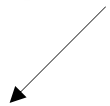
- *Cliente usa intermediário em vez de sujeito real*
- *Intermediário suporta a mesma interface que sujeito real*
- *Intermediário contém uma referência para o sujeito real e repassa chamadas, possivelmente, acrescentando informações ou filtrando dados no processo*

Proxy em Java

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

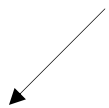
```
public class SujeitoReal implements Sujeito {
  public Object operacao() {
    return coisaUtil;
  }
}
```

inaccessível pelo cliente



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

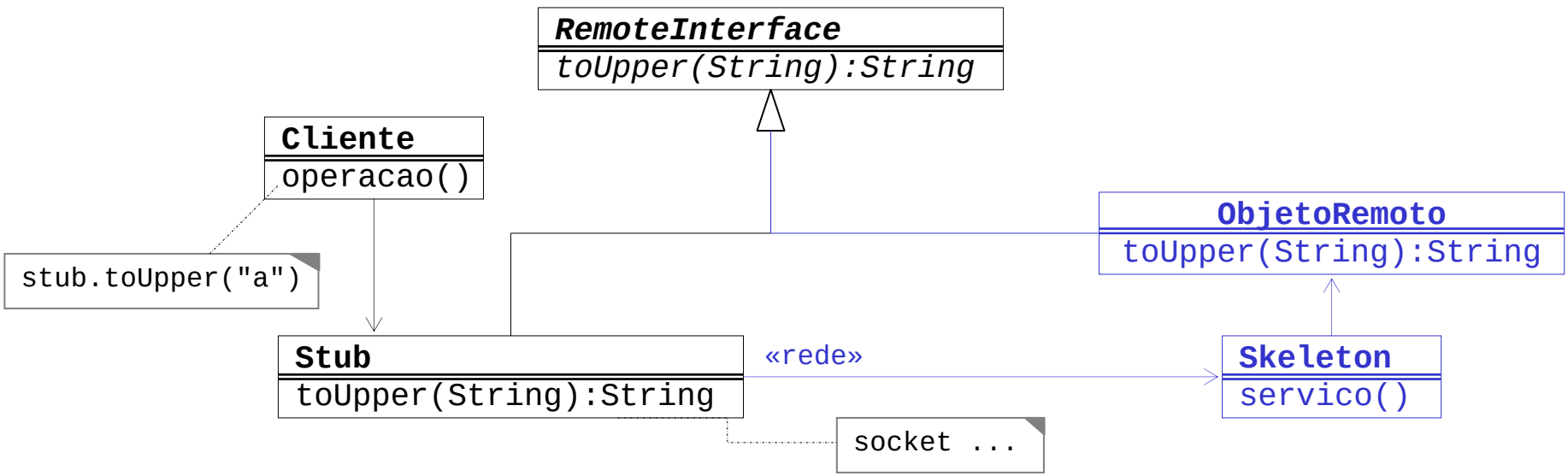
cliente comunica-se com este objeto



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

Quando usar?

- A aplicação mais comum é em objetos distribuídos
- Exemplo: RMI (e EJB)
  - O Stub é proxy do cliente para o objeto remoto
  - O Skeleton é parte do proxy: cliente remoto chamado pelo Stub



- Outras aplicações típicas
  - Image proxy: guarda o lugar de imagem sendo carregada

Questões

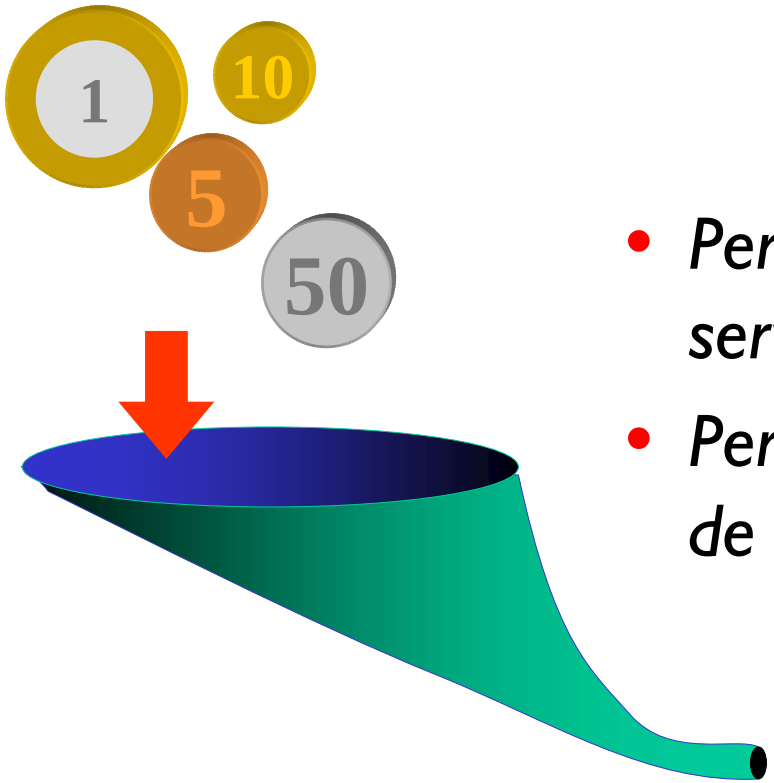
- *Cite exemplos do uso de Proxy*
  - *Em frameworks*
  - *Na API Java*
- *Qual a diferença entre um Proxy e um Adapter?*
- *Qual a diferença entre um Façade e um Proxy?*
  - *E entre um Adapter e um Façade?*

9

Chain of Responsibility

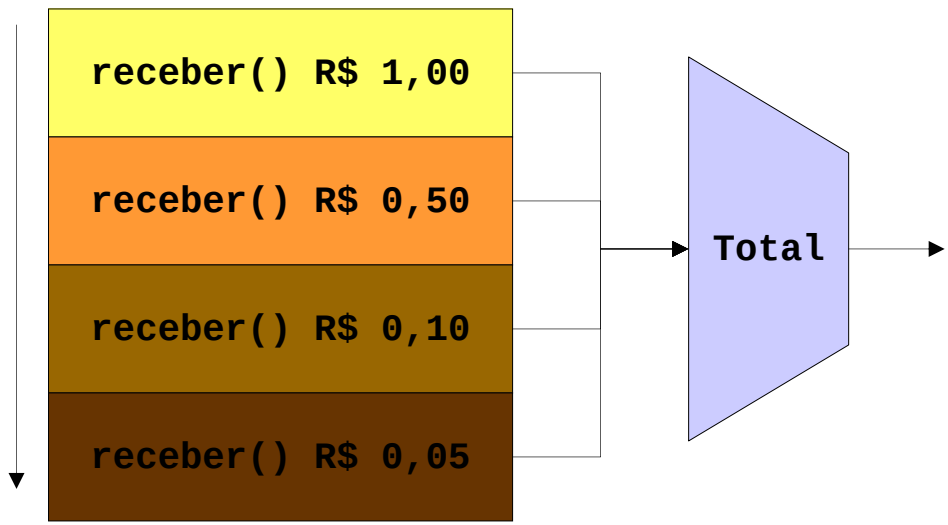
"Evita acoplar o remetente de uma requisição ao seu destinatário ao dar a mais de um objeto a chance de servir a requisição. Compõe os objetos em cascata e passa a requisição pela corrente até que um objeto a sirva." [GoF]

Problema

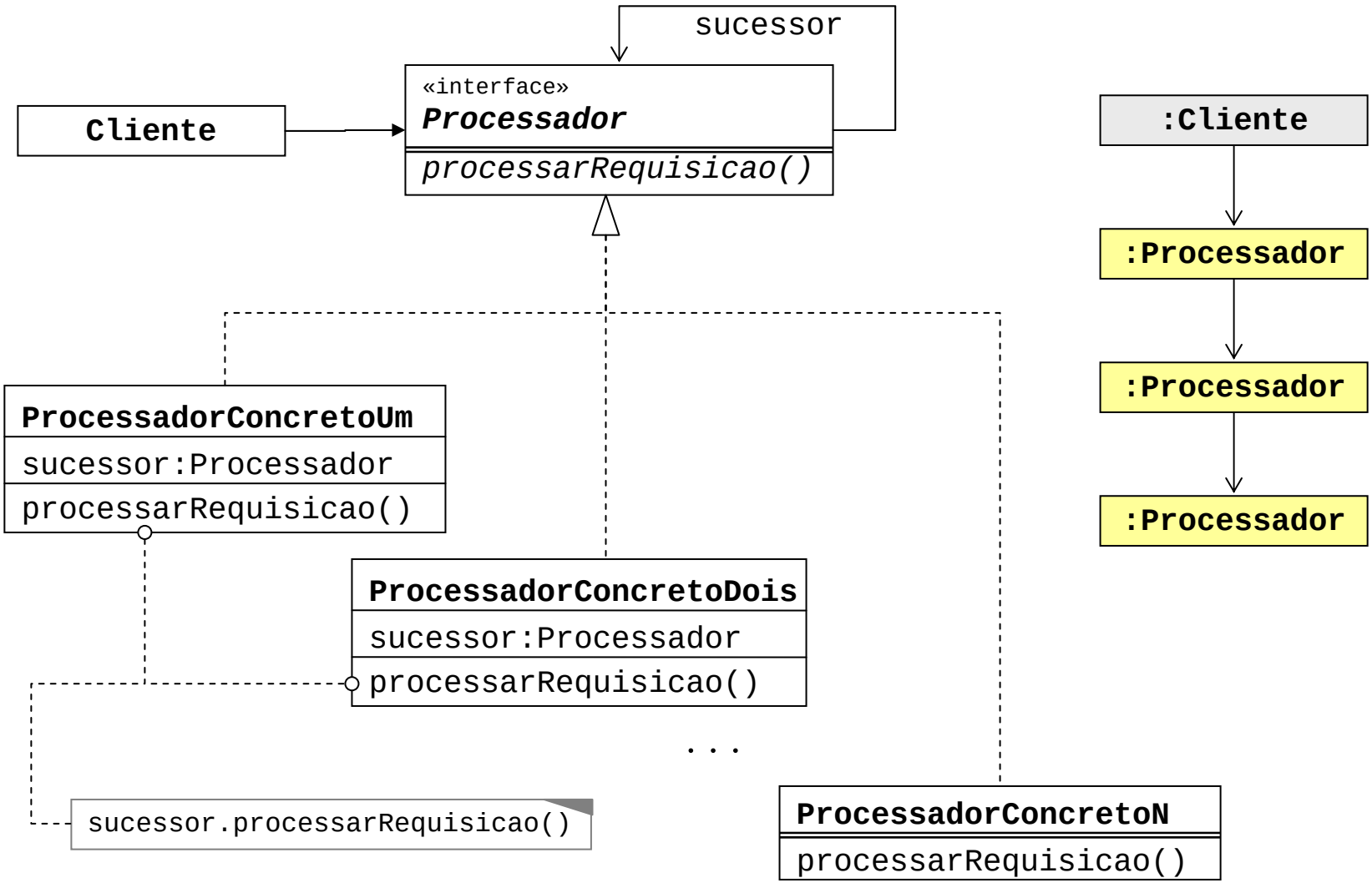


- *Permitir que vários objetos possam servir a uma requisição ou repassá-la*
- *Permitir divisão de responsabilidades de forma transparente*

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



Estrutura de Chain of Responsibility



Chain of Responsibility em Java

```
public class cliente {
   ...
        Processador p1 = ...
        Object resultado = p1.processarRequisicao();
   ...
}
```

```
public class ProcessadorUm implements Processador {
  private Processador sucessor;
  public Object processarRequisicao() {
    ... // codigo um
    return sucessor.processarRequisicao();
  }
}
```

Nesta estratégia cada participante conhece seu sucessor

```
...
```

```
public class ProcessadorFinal implements Processador {
  public Object processarRequisicao() {
    return objeto;
  }
}
```

```
public interface Processador {
  public Object processarRequisicao();
}
```

Estratégias de Chain Of Responsibility

- *Pode-se implementar um padrão de várias formas diferentes. Cada forma é chamada de estratégia (ou idiom*)*
- *Chain of Responsibility pode ser implementada com estratégias que permitem maior ou menor acoplamento entre os participantes*
  - *Usando um mediador: só o mediador sabe quem é o próximo participante da cadeia*
  - *Usando delegação: cada participante conhece o seu sucessor*

** Não são sinônimos: diferem no detalhamento e dependência de linguagem*

Questões

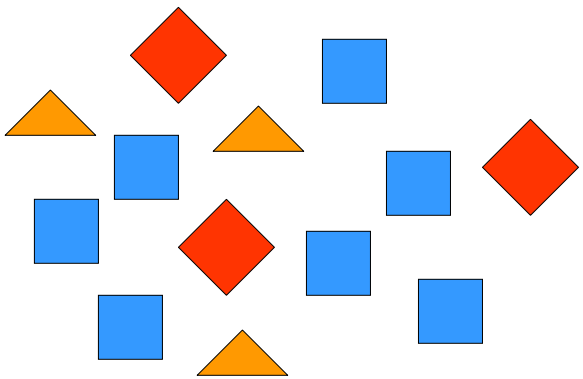
- *Cite exemplos de correntes de responsabilidade em*
  - *Aplicações*
  - *API Java*

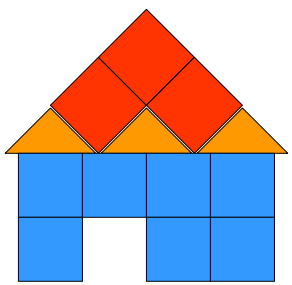
10

Flyweight

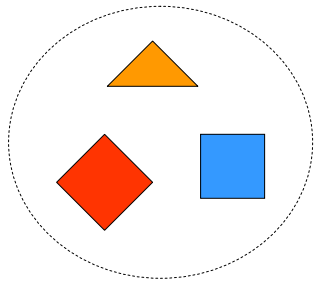
"*Usar compartilhamento para suportar grandes quantidades de objetos refinados eficientemente.*" *[GoF]*

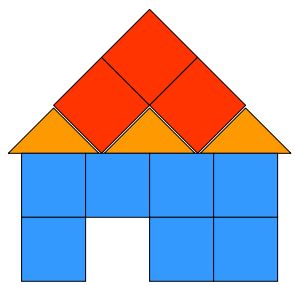
Problema





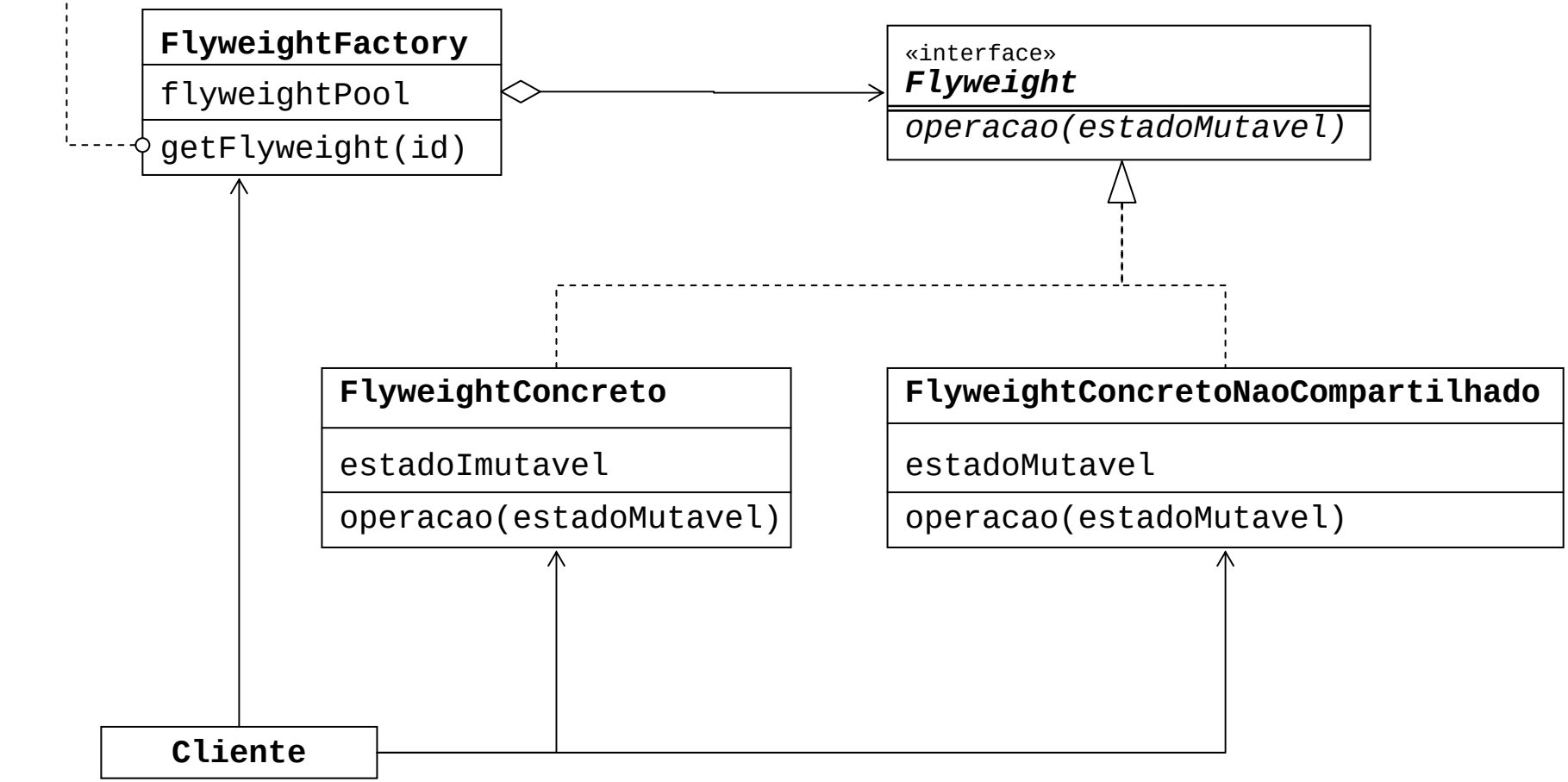
Pool de objetos imutáveis compartilhados





Estrutura de Flyweight

```
if(flyweightPool.containsKey(id)) {
    return (Flyweight)flyweightMap.get(id);
} else {
    Flyweight fly = new FlyweightConcreto( genKey() );
    flyweightPool.put(fly.getKey(), fly);
    return fly;
}
```



Prós e Contras

- *Flyweight é uma solução para construção de aplicações usando objetos imutáveis*
  - *Ideal para objetos que oferecem serviços (guardados em caches e em pools)*
  - *Ideal para objetos que podem ser usados para construir outros objetos*
- *Problemas*
  - *Possível impacto na performance (se houver muitas representações diferentes, elas não podem ser alteradas, e é preciso criar muitos objetos)*

Questões

- *Cite exemplos de Flyweight em Java*
  - *Aplicações*
  - *API Java*
- *Seria uma boa idéia ter principalmente objetos imutáveis em uma aplicação*
  - *Quais as vantagens?*
  - *Quais as desvantagens? Cenários?*
  - *Soluções?*

Resumo: Quando usar?

- *Singleton*
  - *Quando apenas uma instância for permitida*
- *Observer*
  - *Quando houver necessidade de notificação automática*
- *Mediator*
  - *Para controlar a interação entre dois objetos independentes*
- *Proxy*
  - *Quando for preciso um intermediário para o objeto real*
- *Chain of Responsibility*
  - *Quando uma requisição puder ou precisar ser tratada por um ou mais entre vários objetos*
- *Flyweight*
  - *Quando for necessário reutilizar objetos visando performance (cuidado com o efeito oposto!)*

Questões

Qual a diferença entre

- *Observer* e *Mediator*
- *Flyweight* e *Composite*
- *Singleton* e *Façade*
- *Mediator* e *Proxy*
- *Chain of Responsibility* e *Adapter*

Além dos construtores

- Construtores em Java definem maneiras padrão de construir objetos. Sobrecarga permite ampla flexibilidade
- Alguns problemas em depender de construtores
  - Cliente pode não ter todos os dados necessários para instanciar um objeto
  - Cliente fica acoplado a uma implementação concreta (precisa saber a classe concreta para usar new com o construtor)
  - Cliente de herança pode criar construtor que chama métodos que dependem de valores ainda não inicializados (vide processo de construção)
  - Objeto complexo pode necessitar da criação de objetos menores previamente, com certo controle difícil de implementar com construtores
  - Não há como limitar o número de instâncias criadas

Além dos construtores

- *Padrões que oferecem alternativas à construção de objetos*
  - *Builder: obtém informação necessária em passos antes de requisitar a construção de um objeto*
  - *Factory Method: adia a decisão sobre qual classe concreta instanciar*
  - *Abstract Factory: constuir uma família de objetos que compartilham um "tema" em comum*
  - *Prototype: especificar a criação de um objeto a partir de um exemplo fornecido*
  - *Memento: reconstruir um objeto a partir de uma versão que contém apenas seu estado interno*

11

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

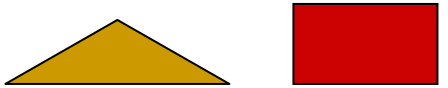
Utilizando as informações passadas pelo cliente, ordena a criação da casa pelo construtor usando uma interface uniforme

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)

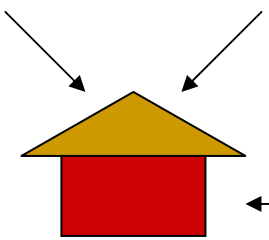
ConstrutorDeCasas

ConstrutorDePredios



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

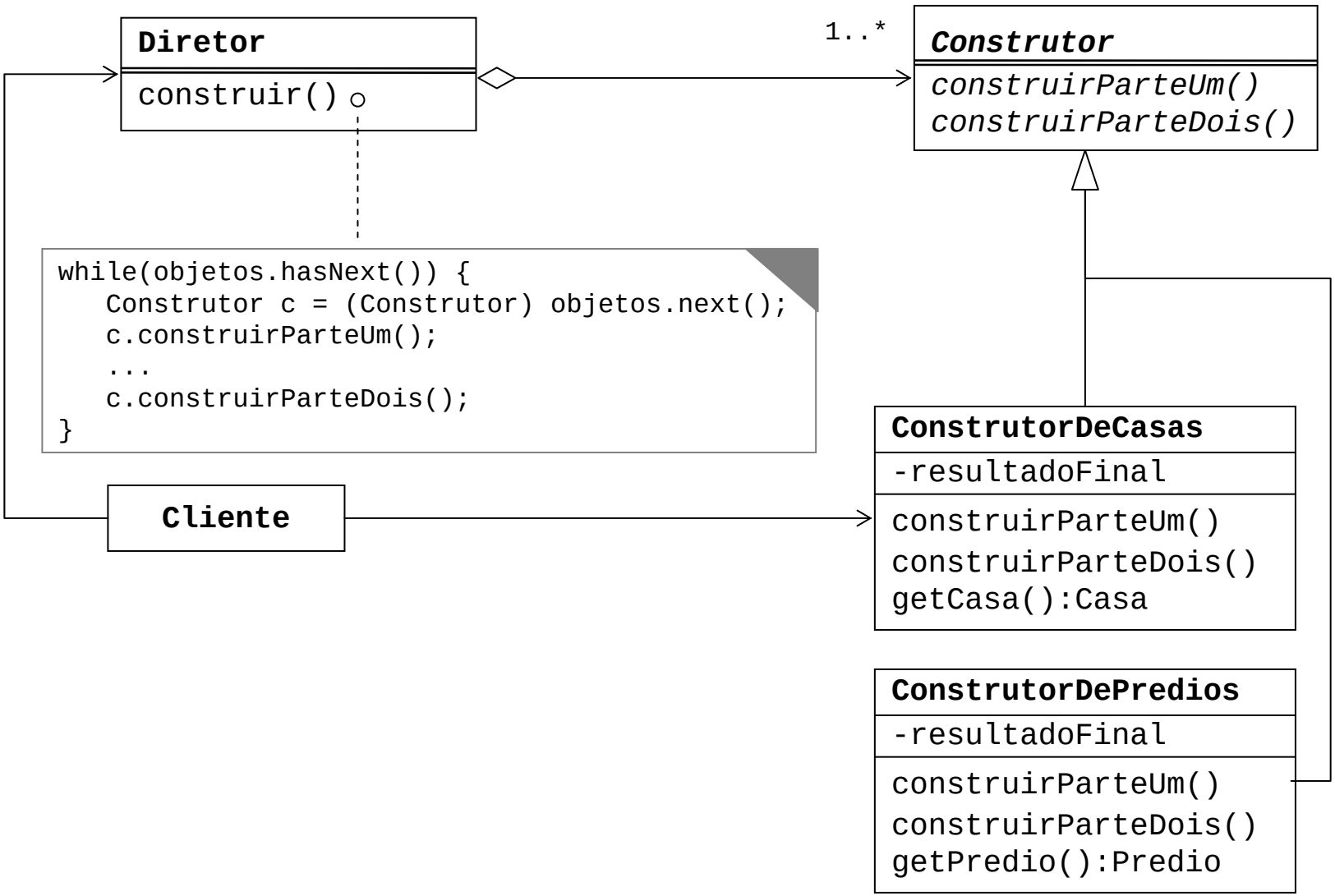
Quando o produto estiver pronto, o cliente pode buscar seu produto diretamente do construtor.



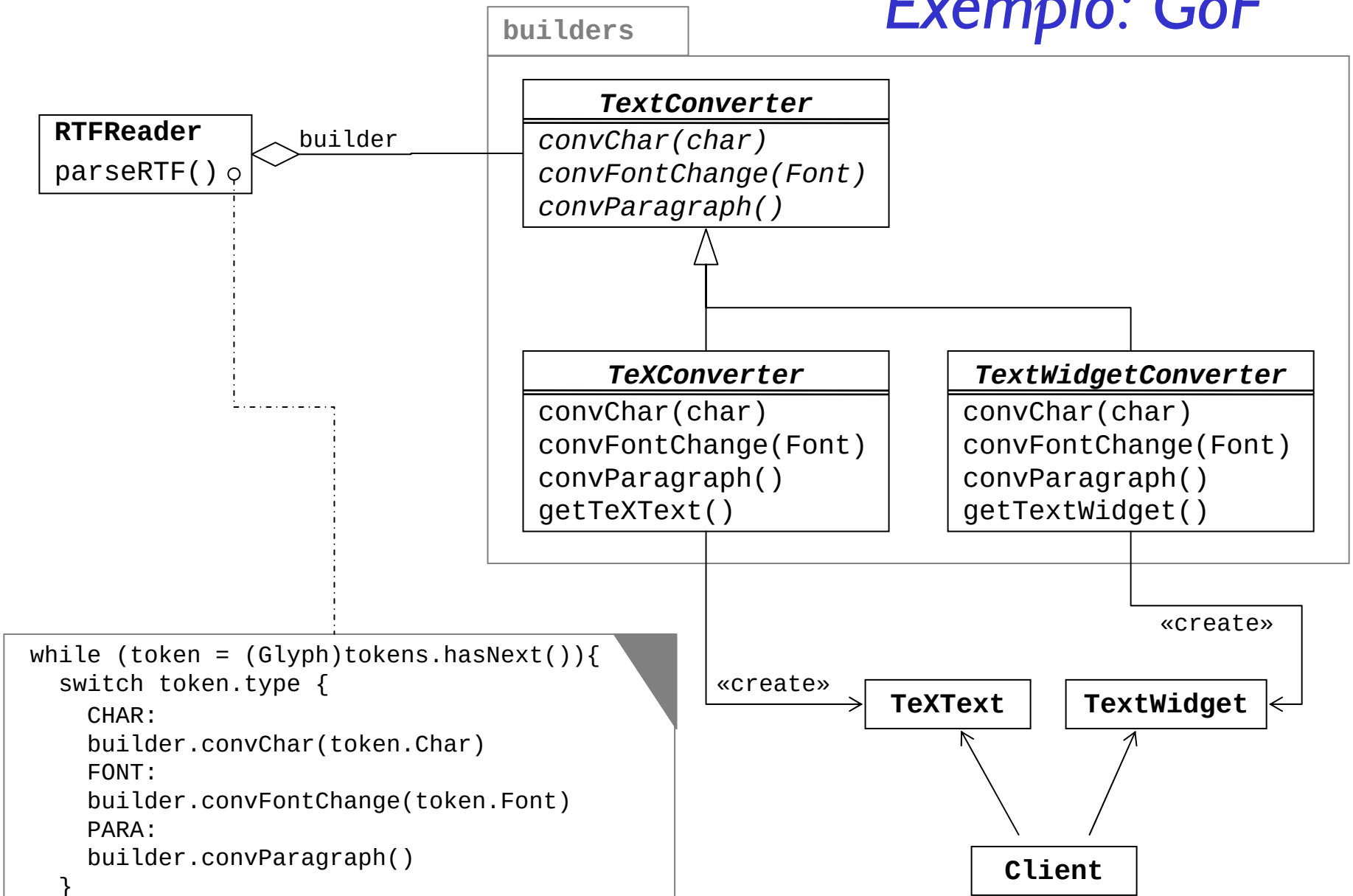
obterProduto()

Cliente

Exemplo

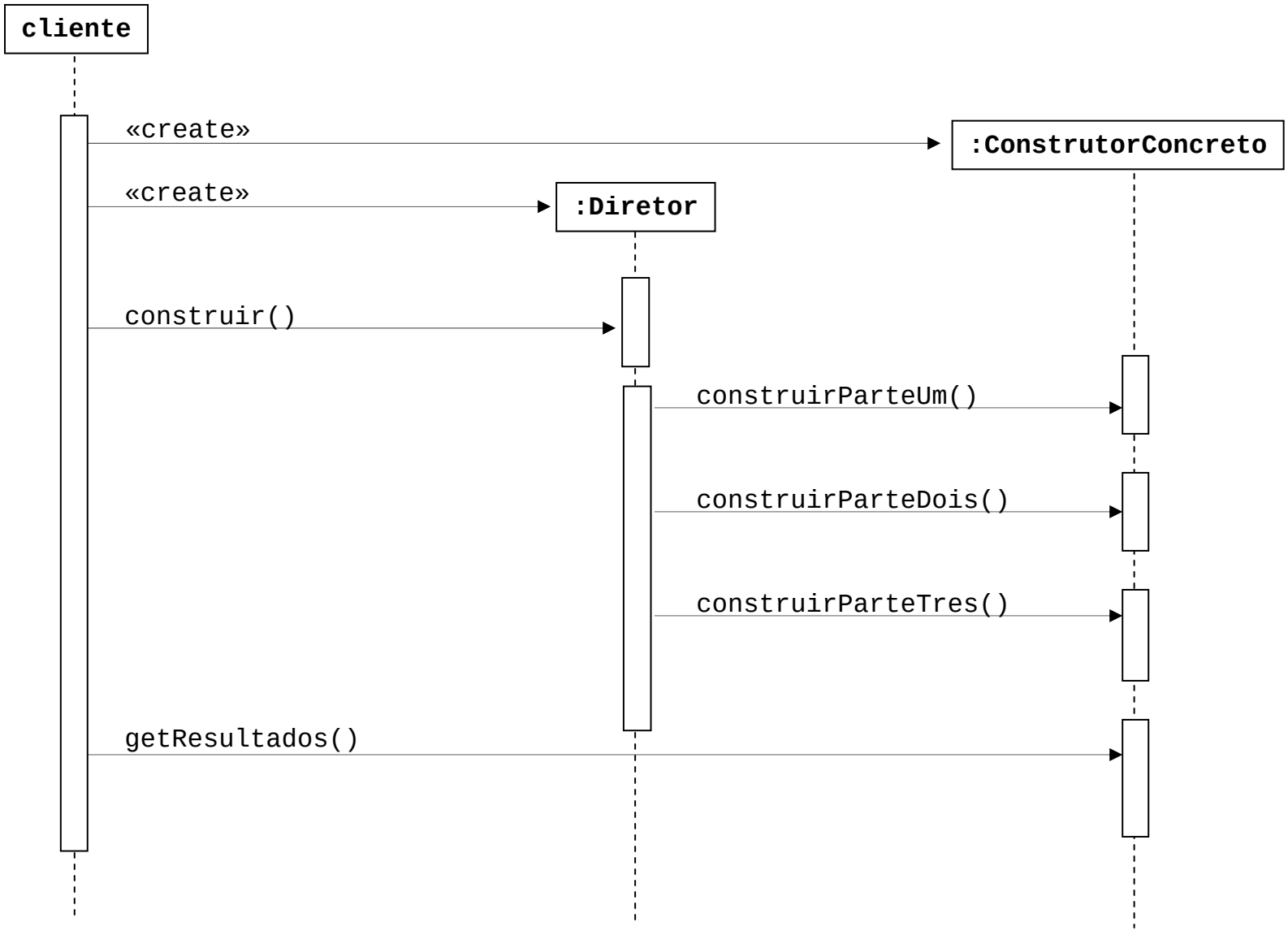

Diretor
construir()
1..*
Construtor
construirParteUm()
construirParteDois()
while(objetos.hasNext()) {
Construtor c = (Construtor) objetos.next();
c.construirParteUm();
...
c.construirParteDois();
}
Cliente
ConstrutorDeCasas
-resultadoFinal
construirParteUm()
construirParteDois()
getCasa():Casa
ConstrutorDePredios
-resultadoFinal
construirParteUm()
construirParteDois()
getPredio():Predio

Exemplo: GoF




```
while (token = (Glyph)tokens.hasNext()){
  switch token.type {
    CHAR:
    builder.convChar(token.Char)
    FONT:
    builder.convFontChange(token.Font)
    PARA:
    builder.convParagraph()
  }
}
```

Seqüência de Builder



Quando usar?

- Builder permite que uma classe se preocupe com apenas uma parte da construção de um objeto. É útil em algoritmos de construção complexos
  - Use-o quando o algoritmo para criar um objeto complexo precisar ser independente das partes que compõem o objeto e da forma como o objeto é construído
- Builder também suporta substituição dos construtores, permitindo que a mesma interface seja usada para construir representações diferentes dos mesmos dados
  - Use quando o processo de construção precisar suportar representações diferentes do objeto que está sendo construído

Questões

- *Cite implementações de Builder na API Java!*

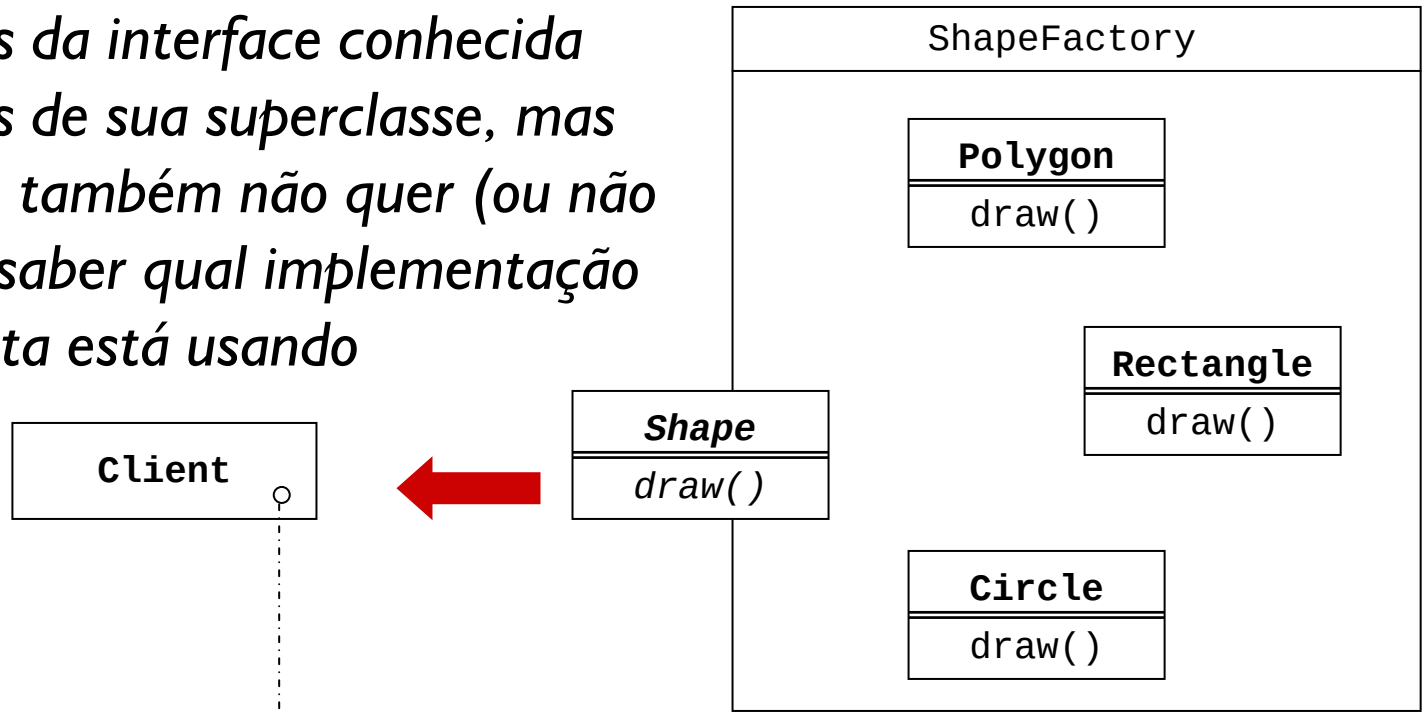
12

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]

Problema

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




```
~~Shape shape = new Rectangle();~~
Shape shape = ShapeFactory.getShape("rect");
shape.draw();
```

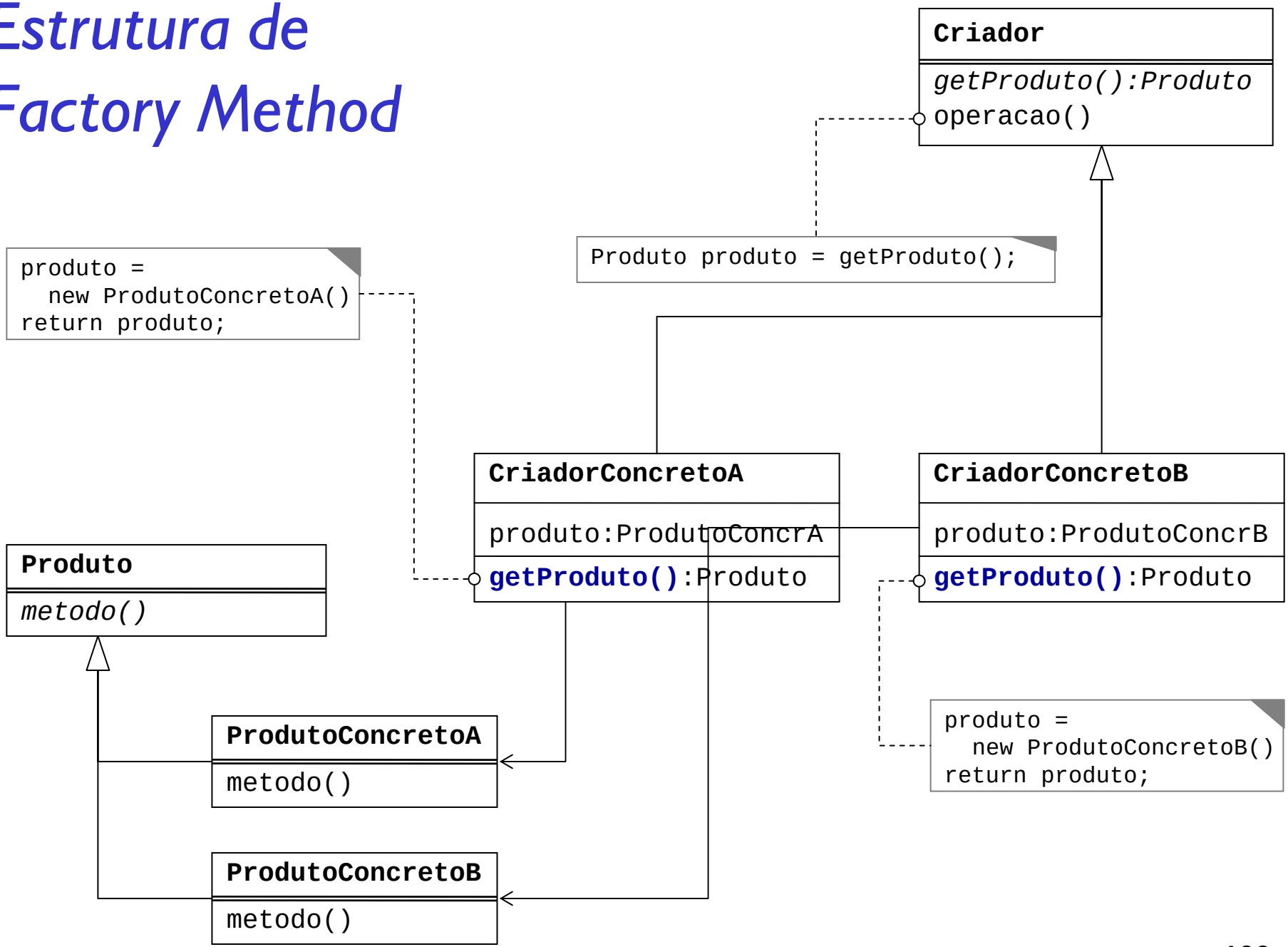
return new Rectangle()
neste contexto

```
public static Shape getShape(String type) {
   ShapeFactory factory = (ShapeFactory)typeMap.get(type);
   return factory.getShape(); // non-static Factory Method
}
```

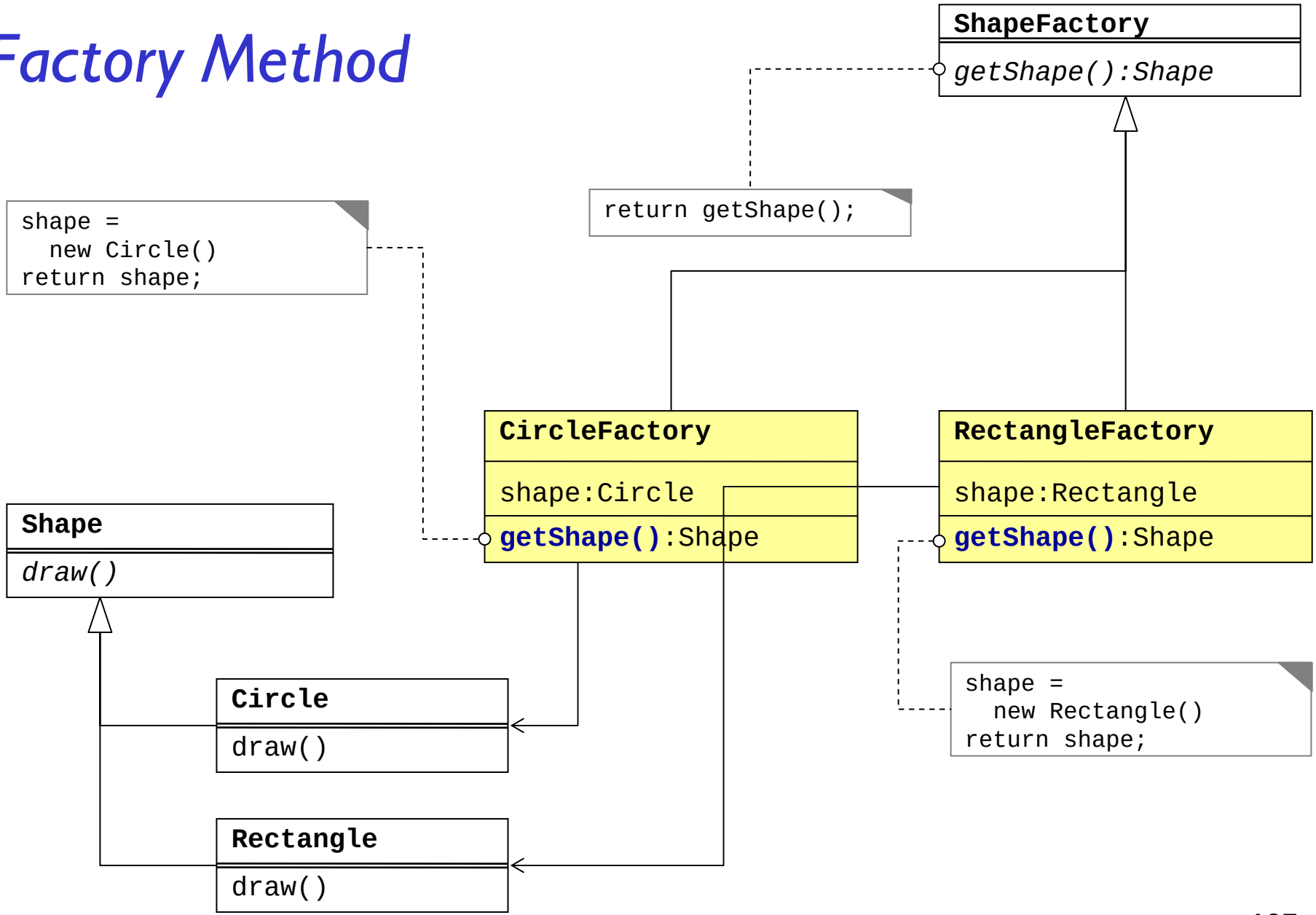
Como implementar?

- É possível criar um objeto sem ter conhecimento algum de sua classe concreta?
  - Esse conhecimento deve estar em alguma parte do sistema, mas não precisa estar no cliente
  - FactoryMethod define uma interface comum para criar objetos
  - O objeto específico é determinado nas diferentes implementações dessa interface
  - O cliente do FactoryMethod precisa saber sobre implementações concretas do objeto criador do produto desejado

Estrutura de Factory Method



Estrutura de Factory Method



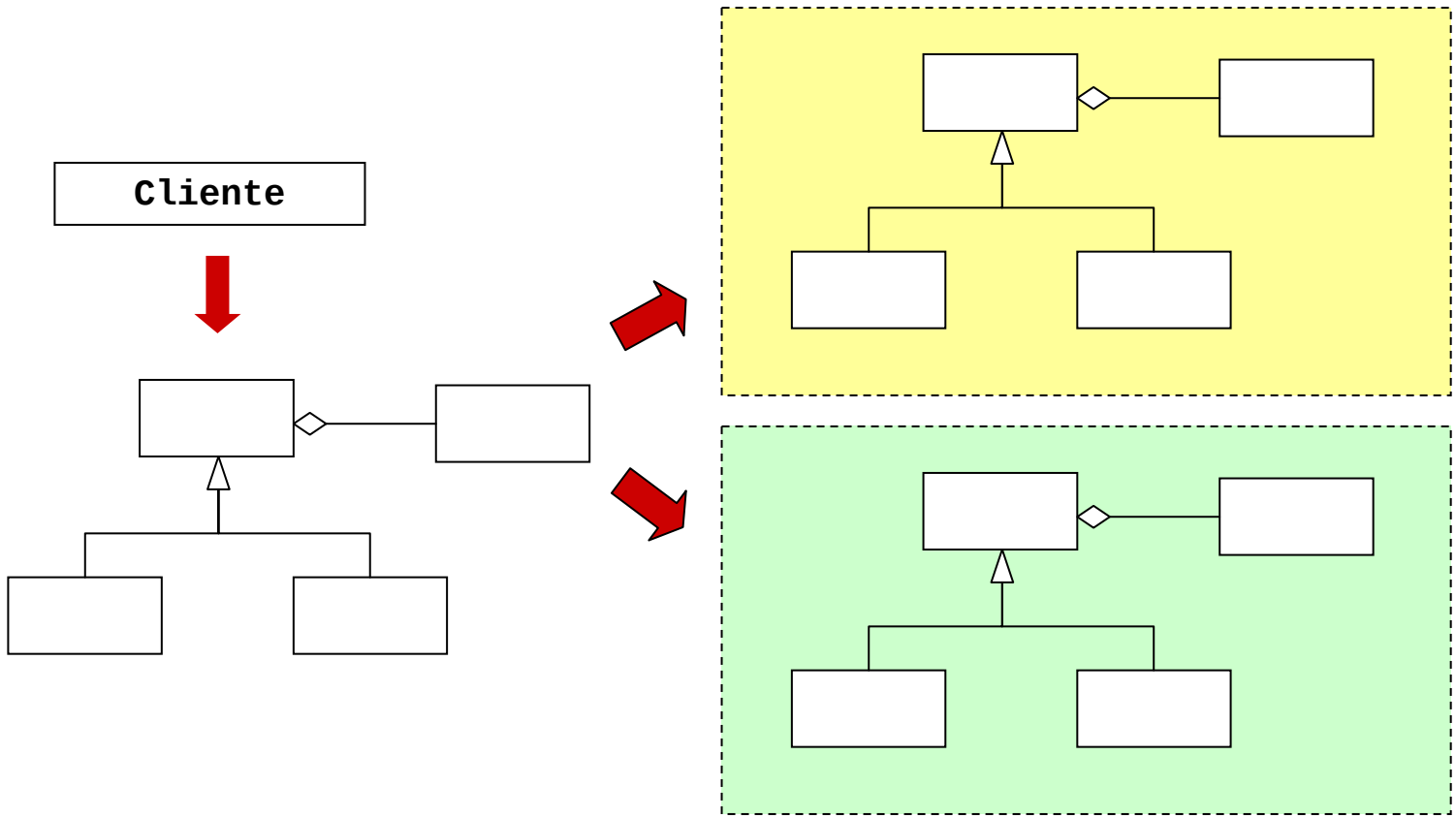
13

Abstract Factory

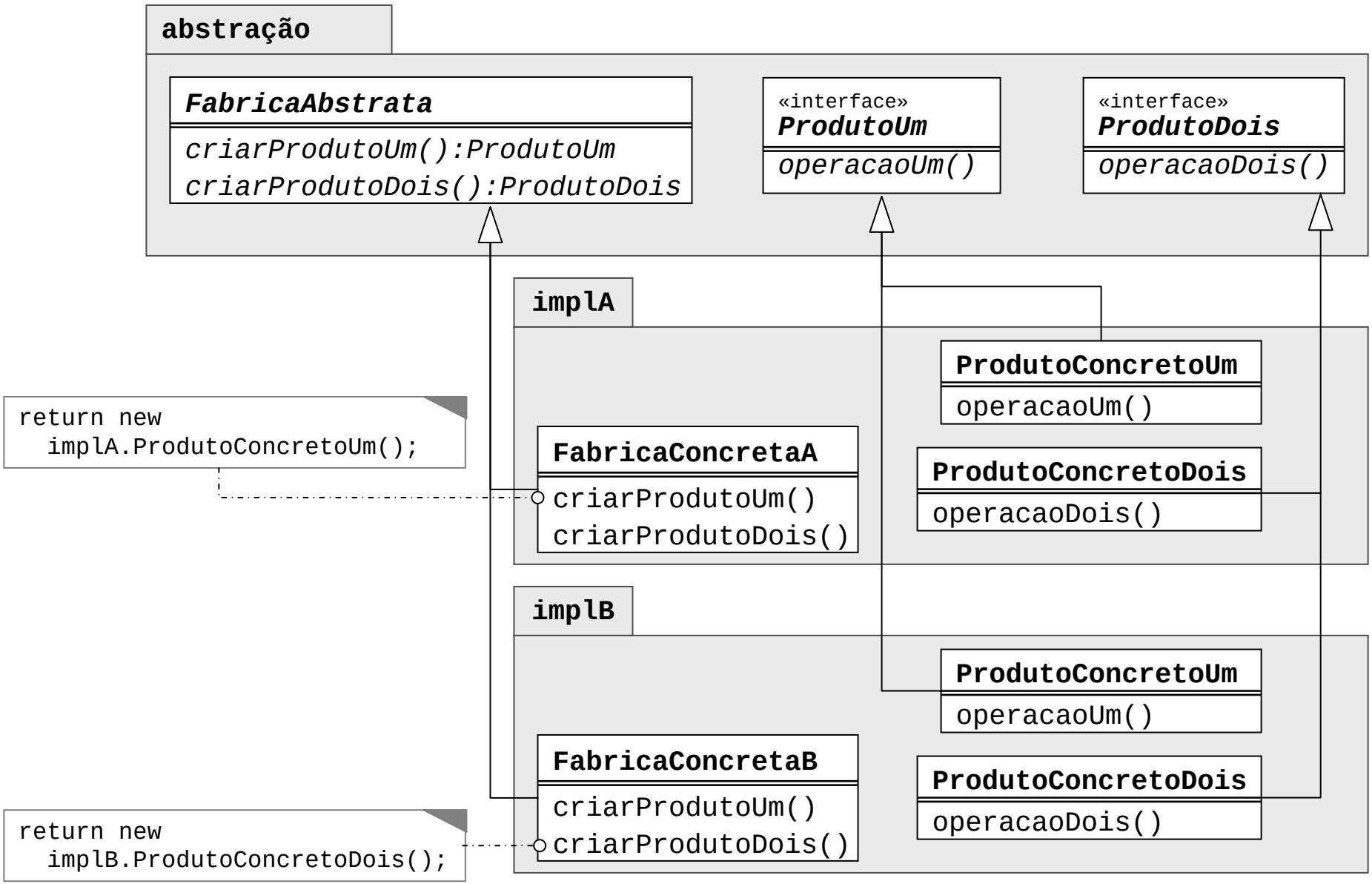
"Prover uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas." [GoF]

Problema

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



Estrutura de Abstract Factory



Questões

- *Cite exemplos de Abstract Factory no J2SDK*
- *Qual a diferença entre Factory Method e Abstract Factory?*

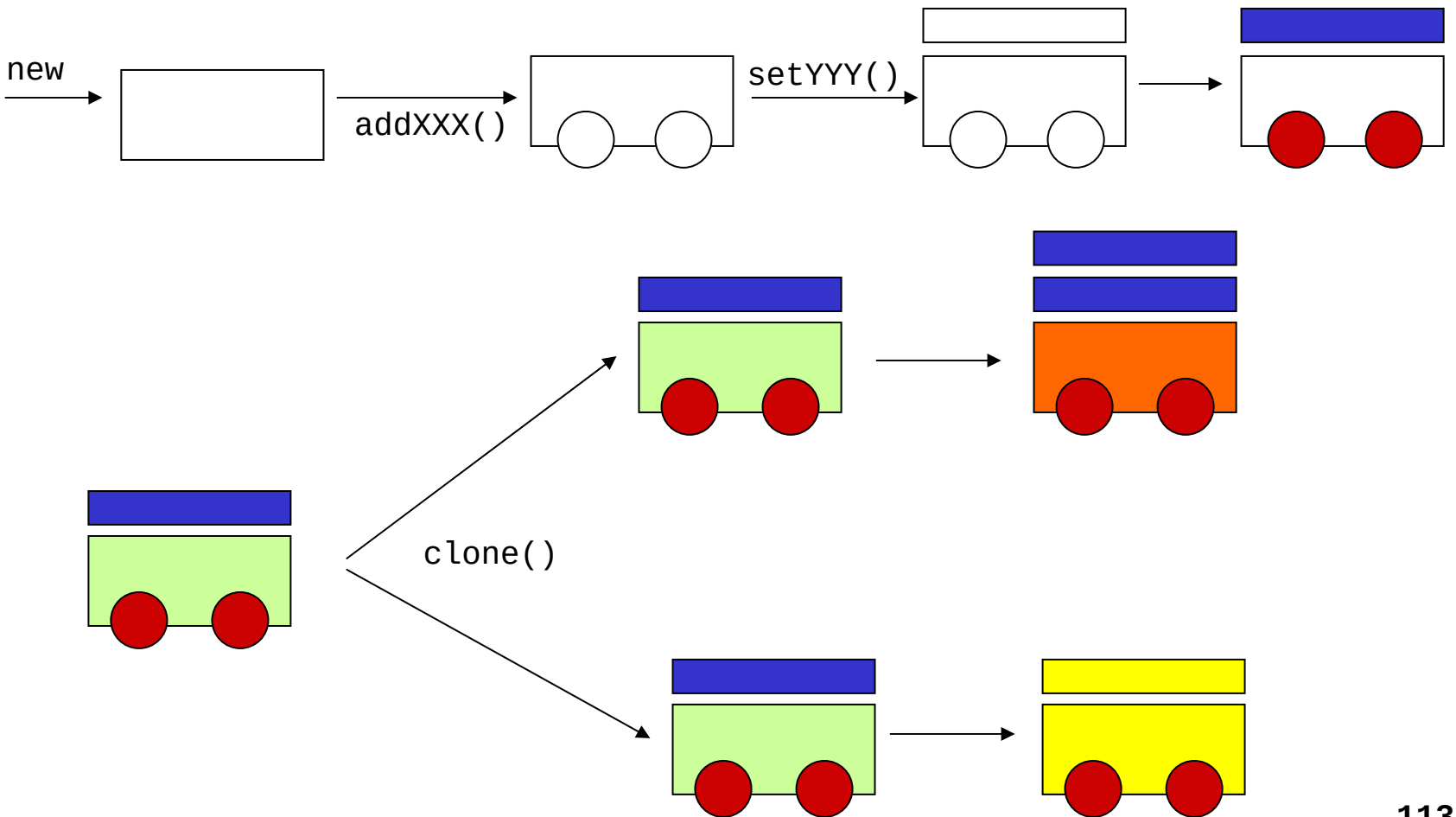
14

Prototype

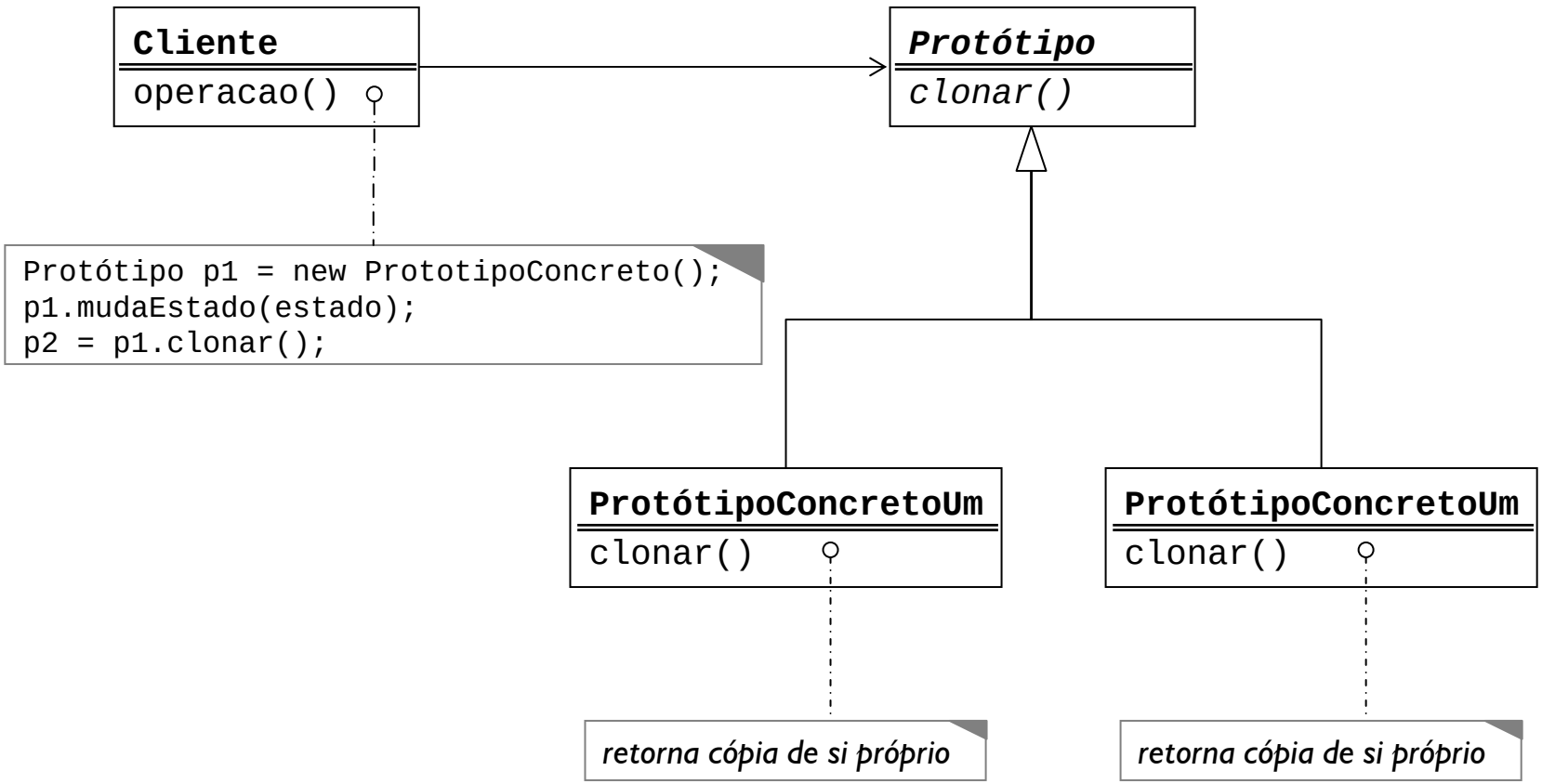
"*Especificar os tipos de objetos a serem criados usando uma instância como protótipo e criar novos objetos ao copiar este protótipo.*" *[GoF]*

Problema

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



Estrutura de Prototype



Resumo

- *O padrão Prototype permite que um cliente crie novos objetos ao copiar objetos existentes*
- *Uma vantagem de criar objetos deste modo é poder aproveitar o estado existente de um objeto*
- *Questão*
  - *Você conhece alguma implementação de Prototype na API Java?*

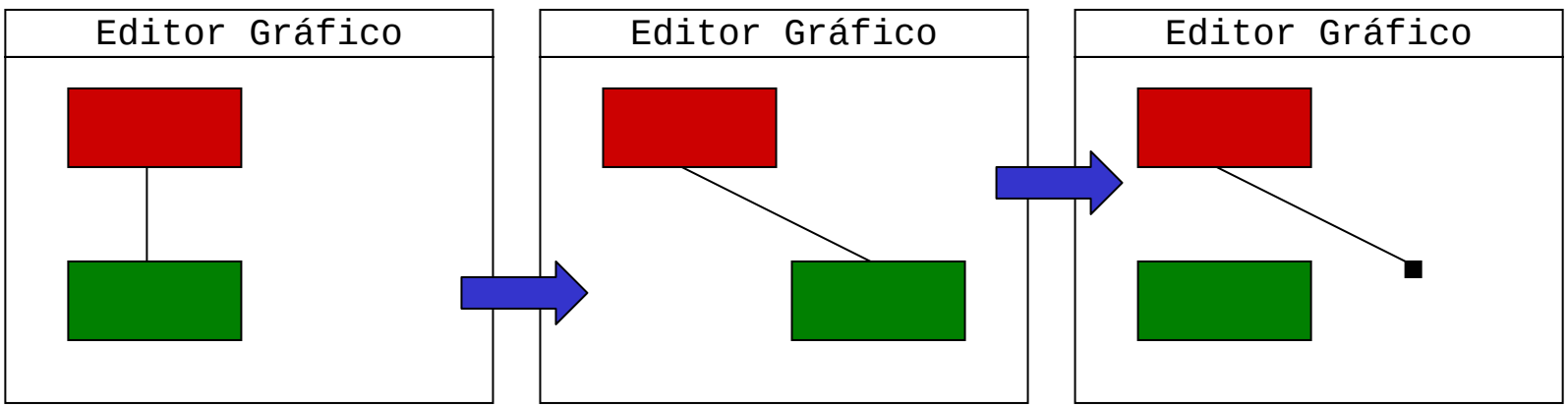
15

Memento

"Sem violar o encapsulamento, capturar e externalizar 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!

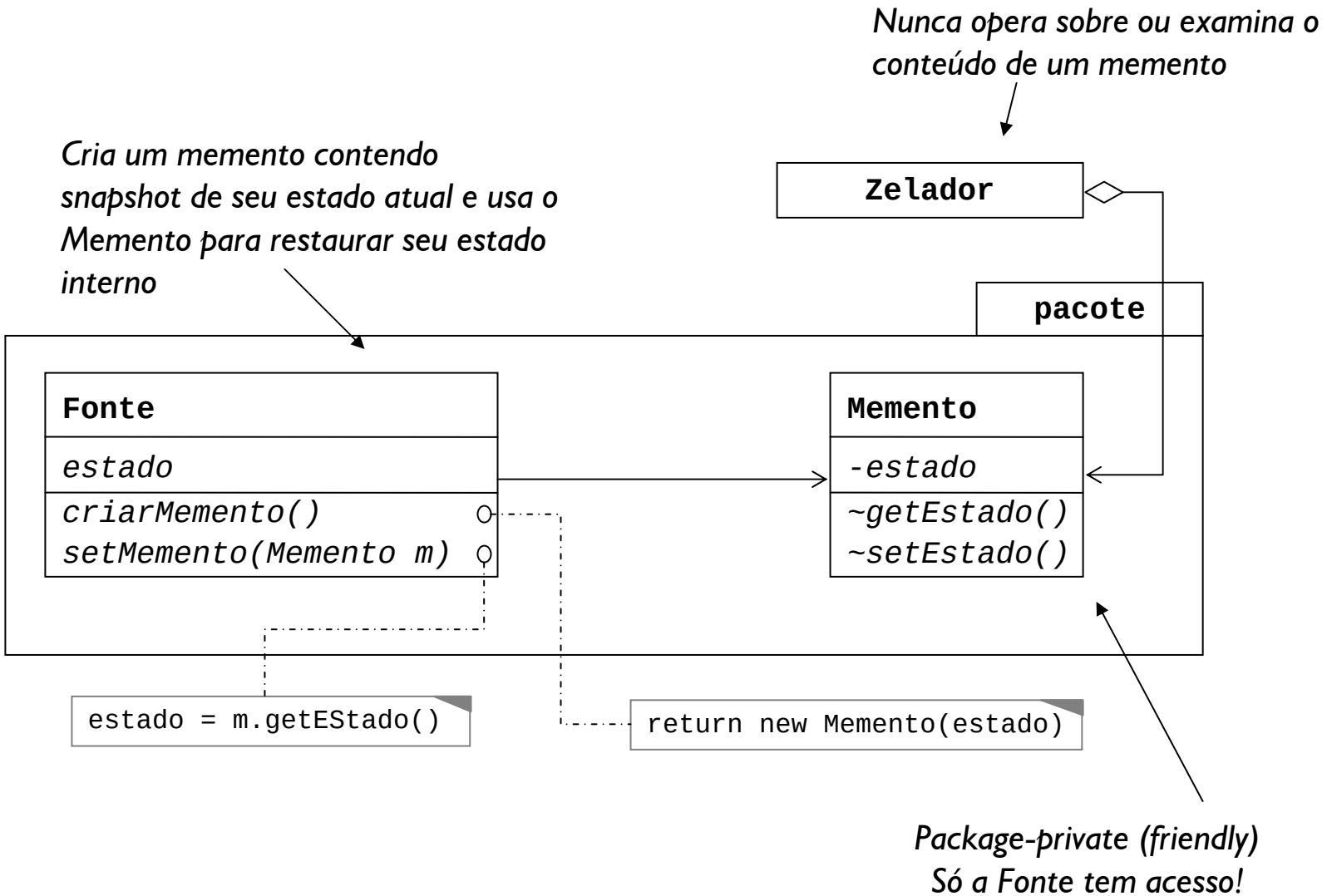
Solução: Memento

- Um memento é um pequeno repositório para guardar estado dos objetos
  - Pode-se usar outro objeto, um string, um arquivo
- Memento guarda um snapshot no estado interno de outro objeto - a Fonte
  - Um mecanismo de Undo irá requisitar um memento da fonte quando ele necessitar verificar o estado desse objeto
  - A fonte reinicializa o memento com informações que caracterizam seu estado atual
  - Só a fonte tem permissão para recuperar informãções do memento (o memento é "opaco" aos outros objetos)

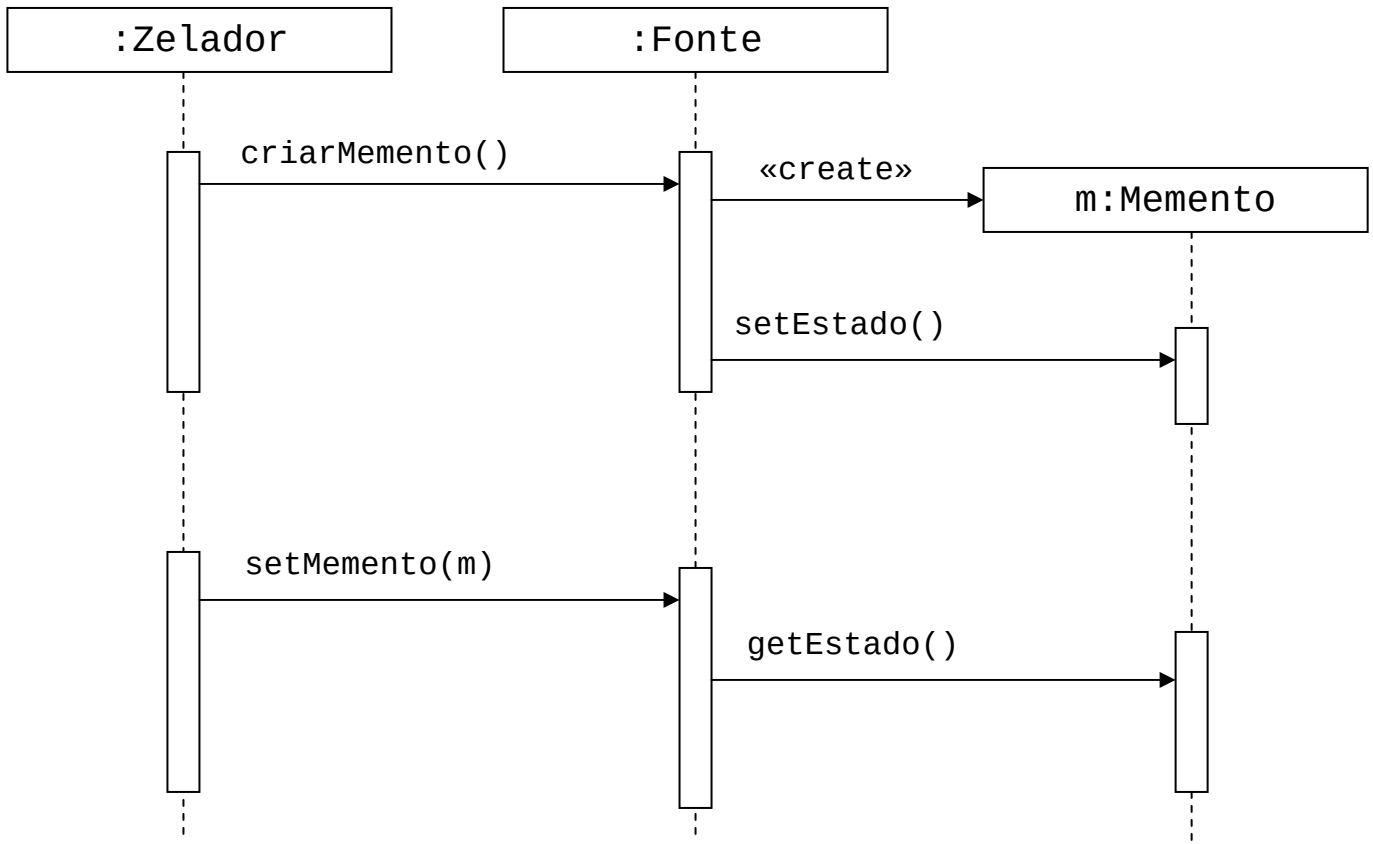
Quando usar?

- *Use Memento quando*
  - *Um snapshot do (parte do) estado de um objeto precisa ser armazenada para que ele possa ser restaurado ao seu estado original posteriormente*
  - *Uma interface direta para se obter esse estado iria expor detalhes de implementação e quebrar o encapsulamento do objeto*

Estrutura de Memento



Seqüência


:Zelador
:Fonte
criarMemento()
«create»
m:Memento
setEstado()
setMemento(m)
getEstado()

Memento em Java

Exemplo genérico

```
package memento;

public class Fonte {
   private Memento memento;
   private Object estado;
   public Memento criarMemento() {
      return new Memento();
   }
   public void setMemento(Memento m) {
      memento = m;
   }
}
```

```
package memento;

public class Memento {
   private Object estado;
   Memento() { }
   void setEstado(Object estado) {
      this.estado = estado;
   }
   Object getEstado() {
      return estado;
   }
}
```

- *Implementar Memento em Java pode ser realizado simplesmente aplicando o encapsulamento de pacotes*
  - *Pacotes pequenos contendo apenas as classes que precisam compartilhar estado*

Resumo: Quando usar?

- *Builder*
  - *Para construir objetos complexos em várias etapas e/ou que possuem representações diferentes*
- *Factory Method*
  - *Para isolar a classe concreta do produto criado da interface usada pelo cliente*
- *Abstract Factory*
  - *Para criar famílias inteiras de objetos que têm algo em comum sem especificar suas interfaces.*
- *Prototype*
  - *Para criar objetos usando outro como base*
- *Memento*
  - *Para armazenar o estado de um objeto sem quebrar o encapsulamento. O uso típico deste padrão é na implementação de operações de Undo.*

Introdução: operações

- *Definições essenciais*
  - *Operação: especificação de um serviço que pode ser requisitado por uma instância de uma classe. Exemplo: operação toString() é implementada em todas as classes.*
  - *Método: implementação de uma operação. Um método tem uma assinatura. Exemplo: cada classe implementa toString() diferentemente*
  - *Assinatura: descreve uma operação com um nome, parâmetros e tipo de retorno. Exemplo: public String toString()*
  - *Algoritmo: uma seqüência de instruções que aceita entradas e produz saída. Pode ser um método, parte de um método ou pode consistir de vários métodos.*

Além das operações comuns

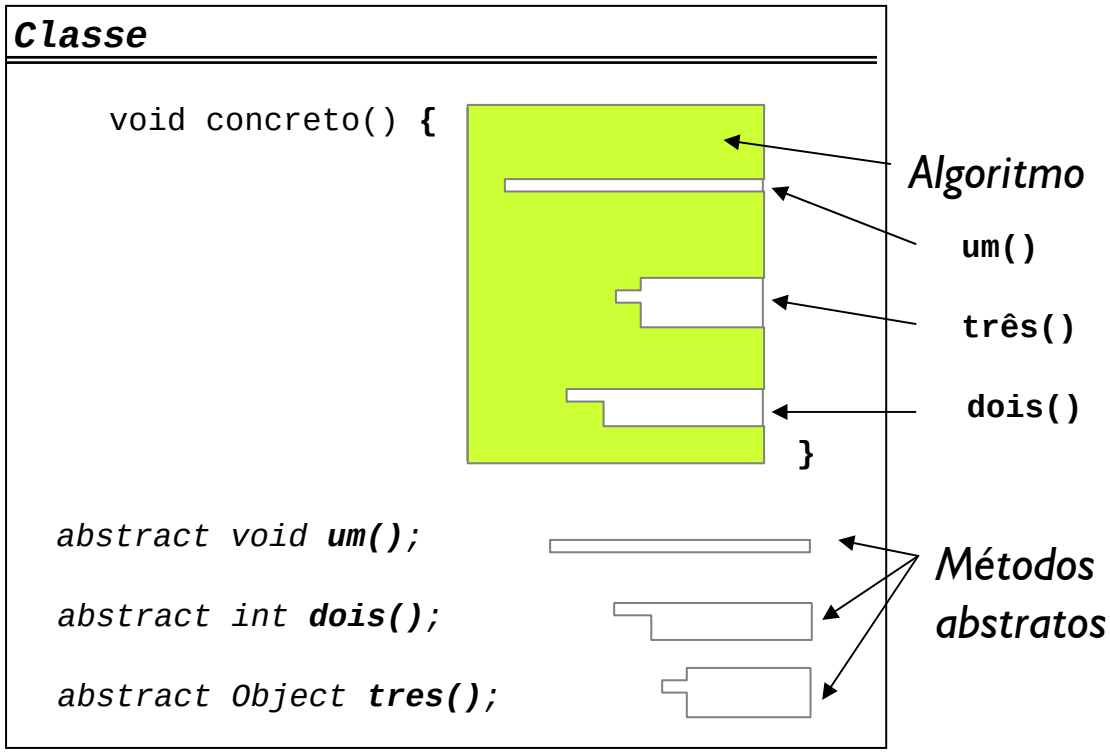
- Vários padrões lidam com diferentes formas de implementar operações e algoritmos
  - Template Method: implementa um algoritmo em um método adiando a definição de alguns passos do algoritmo para que subclasses possam defini-los
  - State: distribui uma operação para que cada classe represente um estado diferente; encapsula **um estado**
  - Strategy: encapsula **um algoritmo** fazendo com que as implementações sejam intercambiáveis
  - Command: encapsula **uma instrução** em um objeto
  - Interpreter: distribui uma operação de tal forma que cada implementação se aplique a um tipo de composição diferente

16

Template Method

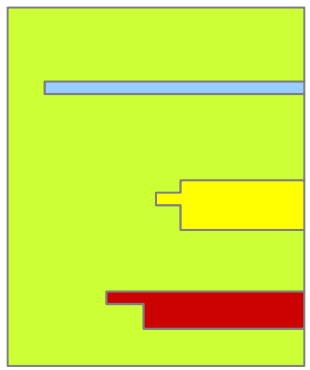
"Definir o esqueleto de um algoritmo dentro de uma operação, deixando alguns passos a serem preenchidos pelas subclasses. Template Method permite que suas subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura." [GoF]

Problema

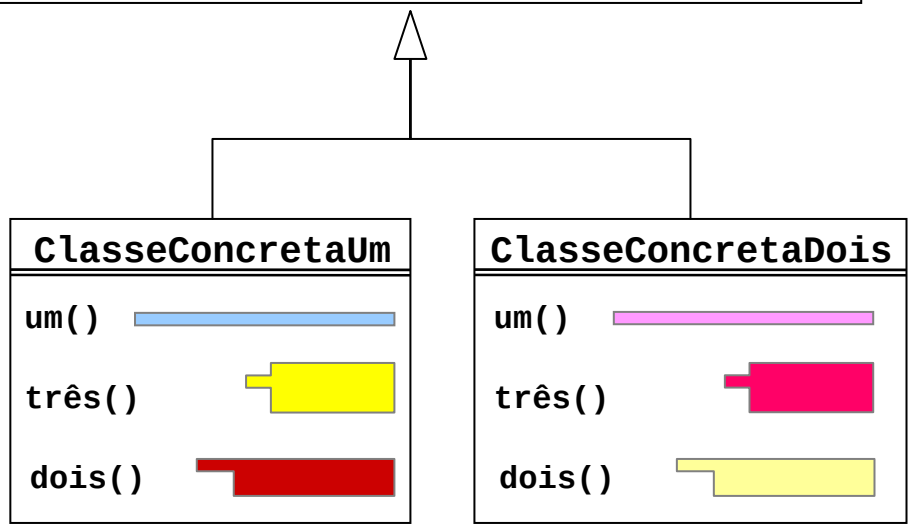


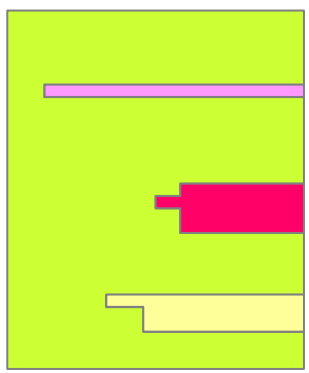
Algoritmos resultantes

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



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





Solução: Template Method

- O que é um Template Method
  - Um Template Method define um algoritmo em termos de operações abstratas que subclasses sobrepõem para oferecer comportamento concreto
- Quando usar?
  - Quando a estrutura fixa de um algoritmo puder ser definida pela superclasse deixando certas partes para serem preenchidos por implementações que podem variar

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();
   }
}
```

Exemplo no J2SDK

- *O método Arrays.sort (java.util) é um bom exemplo de Template Method. Ele recebe como parâmetro um objeto do tipo Comparator que implementa um método compare(a, b) e utiliza-o para definir as regras de ordenação*

```
public class MedeCoisas implements Comparator<Coisa> {

  public int compare(Coisa c1, Coisa c2) {
    if (c1.getID() > c2.getID()) return 1;
    if (c1.getID() < c2.getID()) return -1;
    if (c1.getID() == c2.getID()) return 0;
  }
}
```

Coisa
id: int

Método retorna 1, 0 ou -1 para ordenar Coisas pelo ID

```
...
Coisa coisas[] = new Coisa[10];
coisas[0] = new Coisa("A");
coisas[1] = new Coisa("B");
...
Arrays.sort(coisas, new MedeCoisas());
...
```

Questões

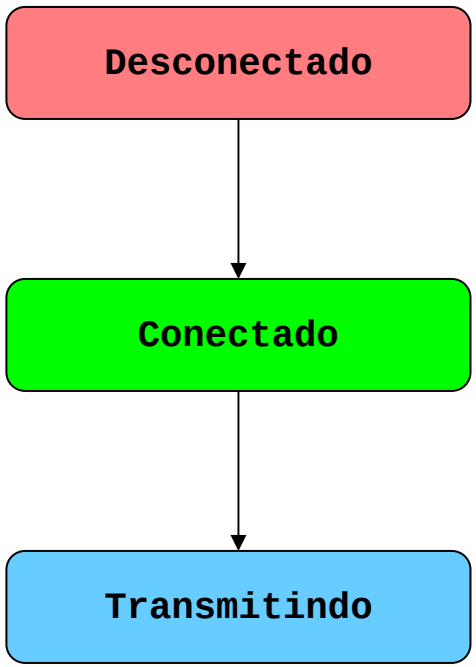
- *Cite outros exemplos de Template Method*
  - *Em Java*
  - *Em frameworks*
- *Que outras aplicações poderiam ser implementadas com Template Method?*

17

State

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

Problema



:Objeto

operação

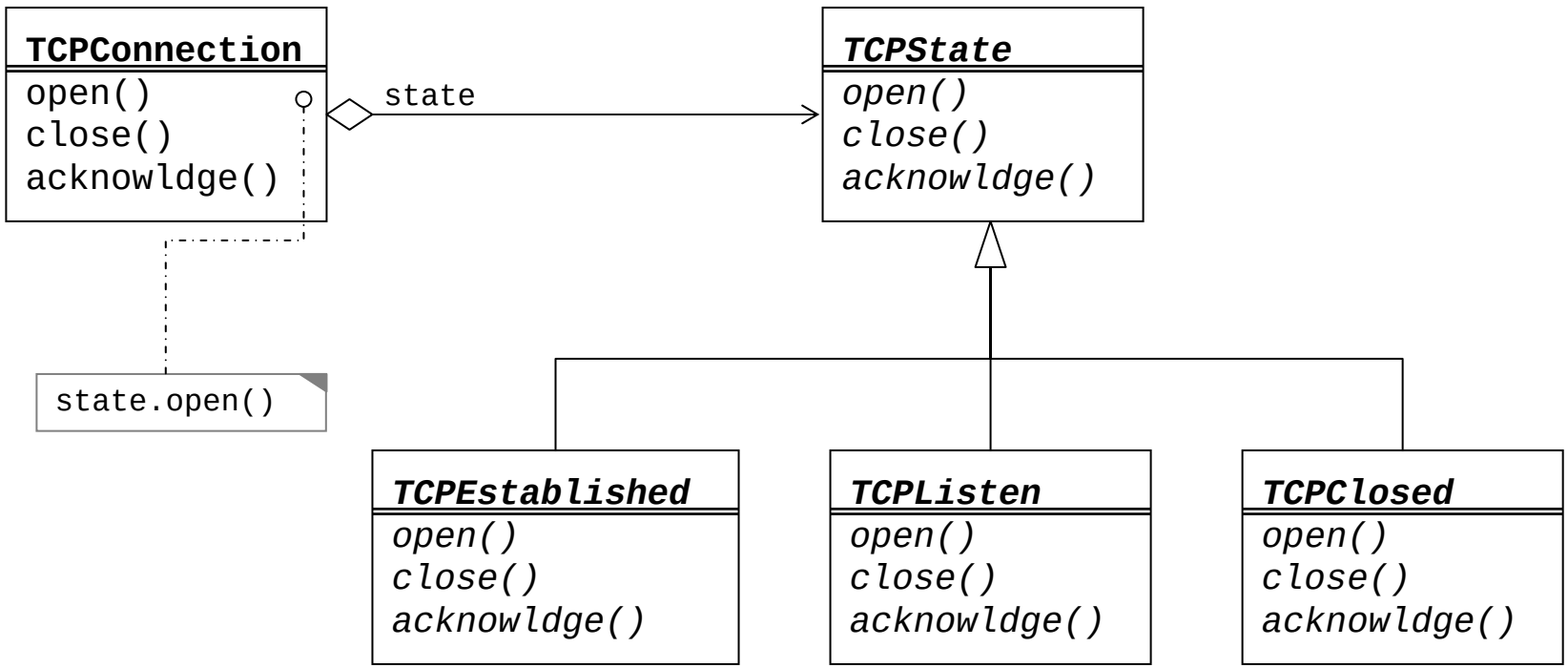
```
if (estado == desconectado) {
  façaIsto();
} else if (estado == conectado) {
  façaAquilo();
} else {
  faça();
}
```

operação

```
estado.faça()
```

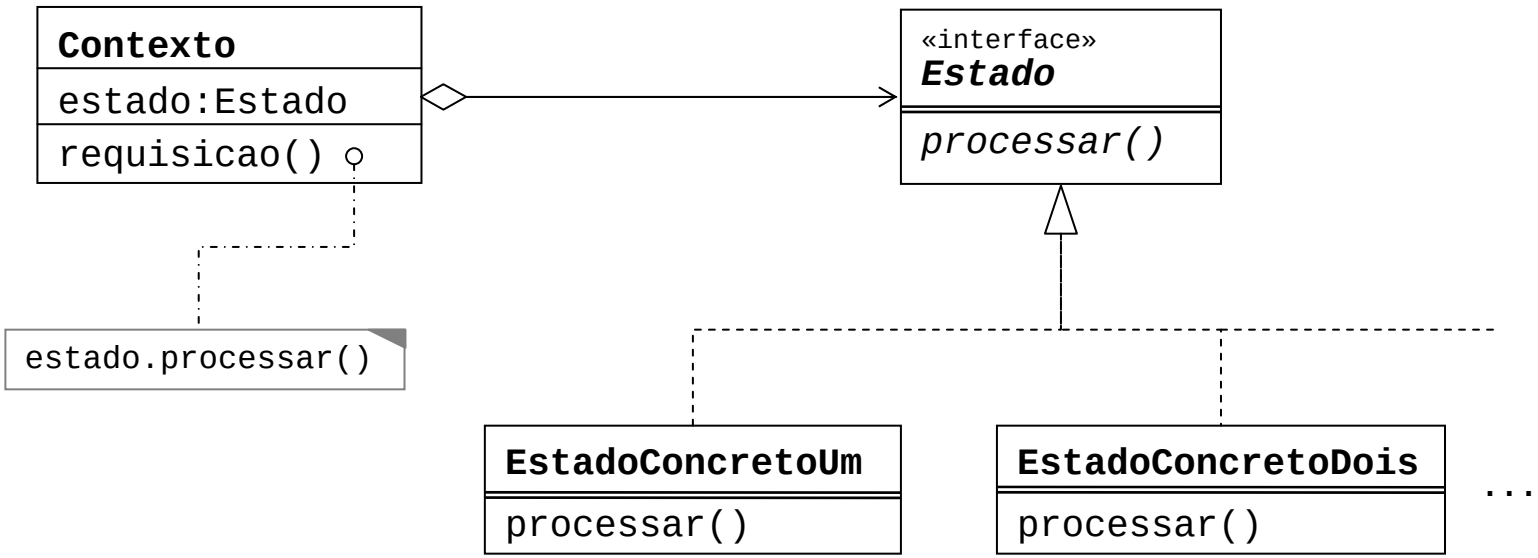
Objetivo: usar objetos para representar estados e ***polimorfismo*** *para tornar a execução de tarefas dependentes de estado transparentes*

Exemplo [GoF]



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

Estrutura de State



- *Contexto:*
  - *define a interface de interesse aos clientes*
  - *mantém uma instância de um EstadoConcreto que define o estado atual*
- *Estado*
  - *define uma interface para encapsular o comportamento associado com um estado particular do contexto*
- *EstadoConcreto*
  - *Implementa um comportamento associado ao estado do contexto*

State em Java

```
public class GatoQuantico {
   public final Estado VIVO = new EstadoVivo();
   public final Estado MORTO = new EstadoMorto();
   public final Estado QUANTICO = new EstadoQuantico();

   private Estado estado;

   public void setEstado(Estado estado) {
      this.estado = estado;
   }

   public void miar() {
      estado.miar();
   }
}
```

```
public class EstadoVivo {
   public void miar() {
      System.out.println("Meaaaooww!!");
   }
}
```

```
public interface Estado {
   void miar();
}
```

```
public class EstadoMorto {
  public void miar() {
     System.out.println("Buu!");
  }
}
```

```
public class EstadoQuantico {
  public void miar() {
     System.out.println("Hello Arnold!");
  }
}
```

18

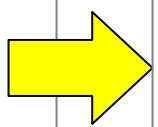
Strategy

"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

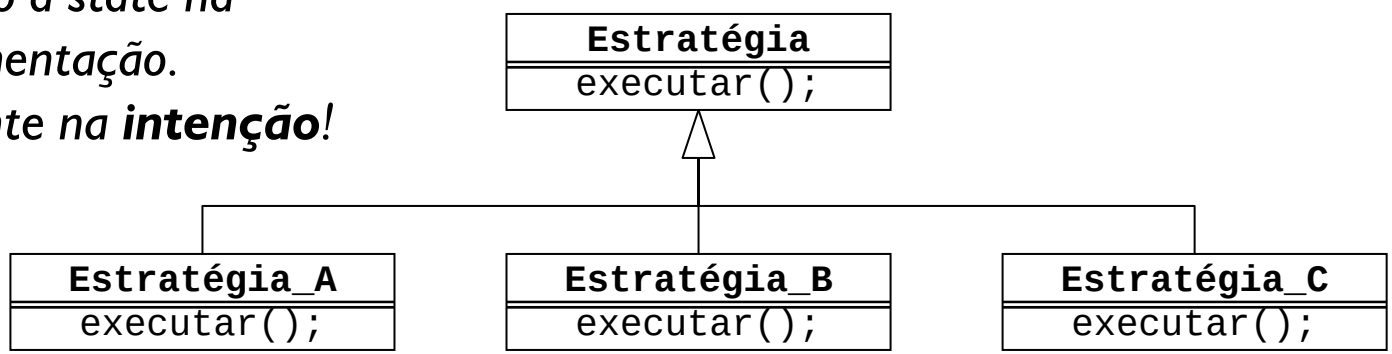
Várias estratégias, escolhidas de acordo com opções ou condições

```
if (guerra && inflação > META) {
   doPlanoB();
else if (guerra && recessão) {
   doPlanoC();
} else {
   doPlanejado();
}
```

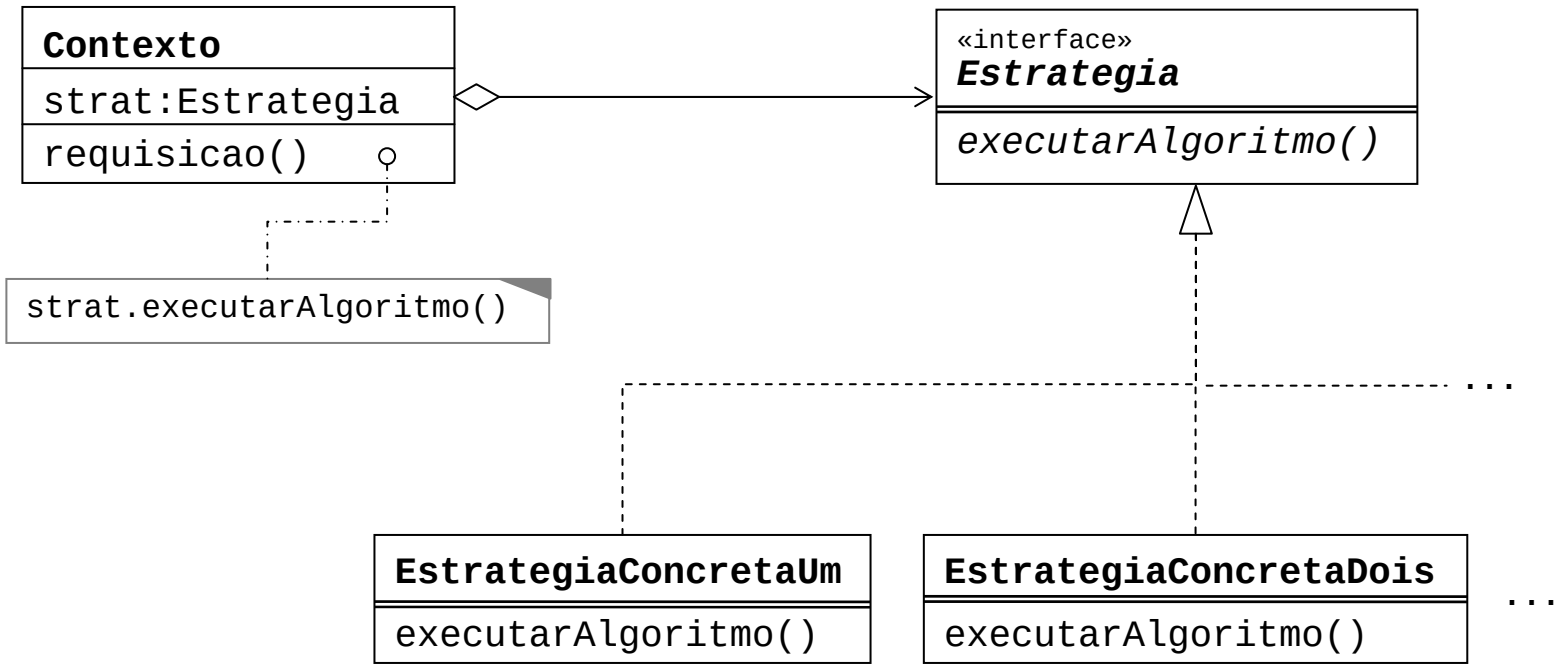


```
if (guerra && inflação > META) {
   plano = new Estrategia_C();
} else if (guerra && recessão) {
   plano = new Estrategia_B();
} else {
   plano = new Estrategia_A();
}

plano.executar();
```

Idêntico a state na implementação.
*Diferente na **intenção**!*



Estrutura de Strategy



- *Um contexto repassa requisições de seus clientes para sua estratégia. Clientes geralmente criam e passam uma EstategiaConcreta para o contexto. Depois, clientes interagem apenas com o contexto*
- *Estrategia e Contexto interagem para implementar o algoritmo escolhido. Um contexto pode passar todos os dados necessários ou uma cópia de si próprio*

Quando usar?

- Quando classes relacionadas forem diferentes apenas no seu comportamento
  - Strategy oferece um meio para configurar a classe com um entre vários comportamentos
- Quando você precisar de diferentes variações de um mesmo algoritmo
- Quando um algoritmo usa dados que o cliente não deve conhecer
- Quando uma classe define muitos comportamentos, e estes aparecem como múltiplas declarações condicionais em suas operações

Strategy 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() {
      vizinhoPeloNorte();
      atacarPeloSul();
      ...
   }
   public void concluir() {
      dividirBeneficios(...);
      dividirReconstrução(...);
   }
}
```

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

Questões

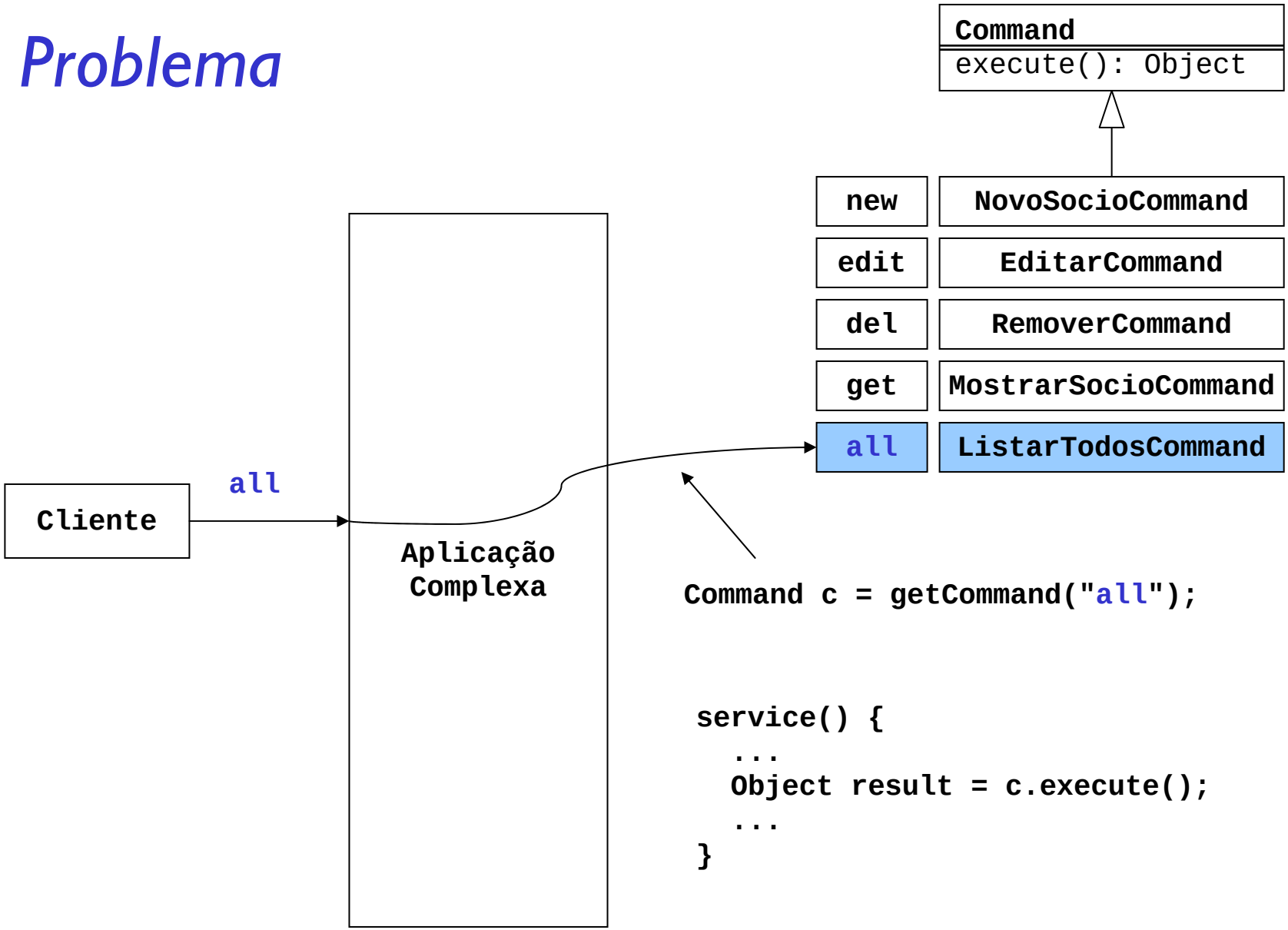
- *Cite exemplos de Strategy*
- *Qual a diferença entre Strategy e State?*

19

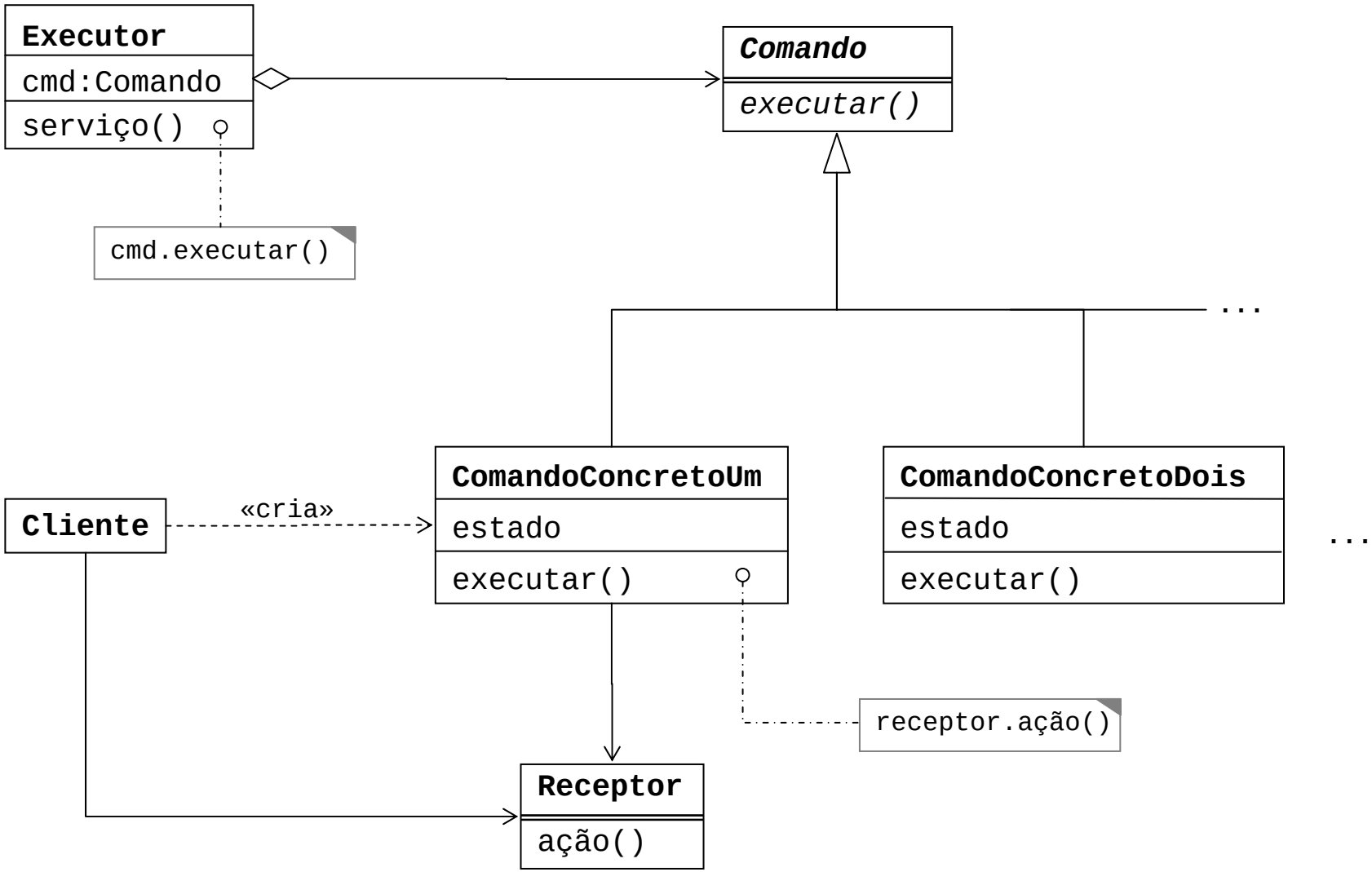
Command

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

Problema


Command
execute(): Object
new
NovoSocioCommand
edit
EditarCommand
del
RemoverCommand
get
MostrarSocioCommand
all
ListarTodosCommand
Cliente
all
Aplicação
Complexa
Command c = getCommand("all");
service() {
...
Object result = c.execute();
...
}

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);
  }
}
```

Questões

- *Cite exemplos de Command*
  - *Na API Java*
  - *Em frameworks*
- *Qual a diferença entre*
  - *Strategy e Command?*
  - *State e Command?*
  - *State e Strategy?*

State, Strategy e Command

- *Diferentes intenções, diagramas e implementações similares (ou idênticas)*
- *Como distinguir?*
  - *State representa um **estado** (substantivo) e geralmente está menos accessível (a mudança de estado pode ser desencadeada por outro estado)*
  - *Strategy representa um **comportamento** (verbo) e é escolhida dentro da aplicação (a ação pode ser desencadeada por ação do cliente ou estado)*
  - *Command representa uma **ação** escolhida e iniciada por um cliente externo (usuário)*

20

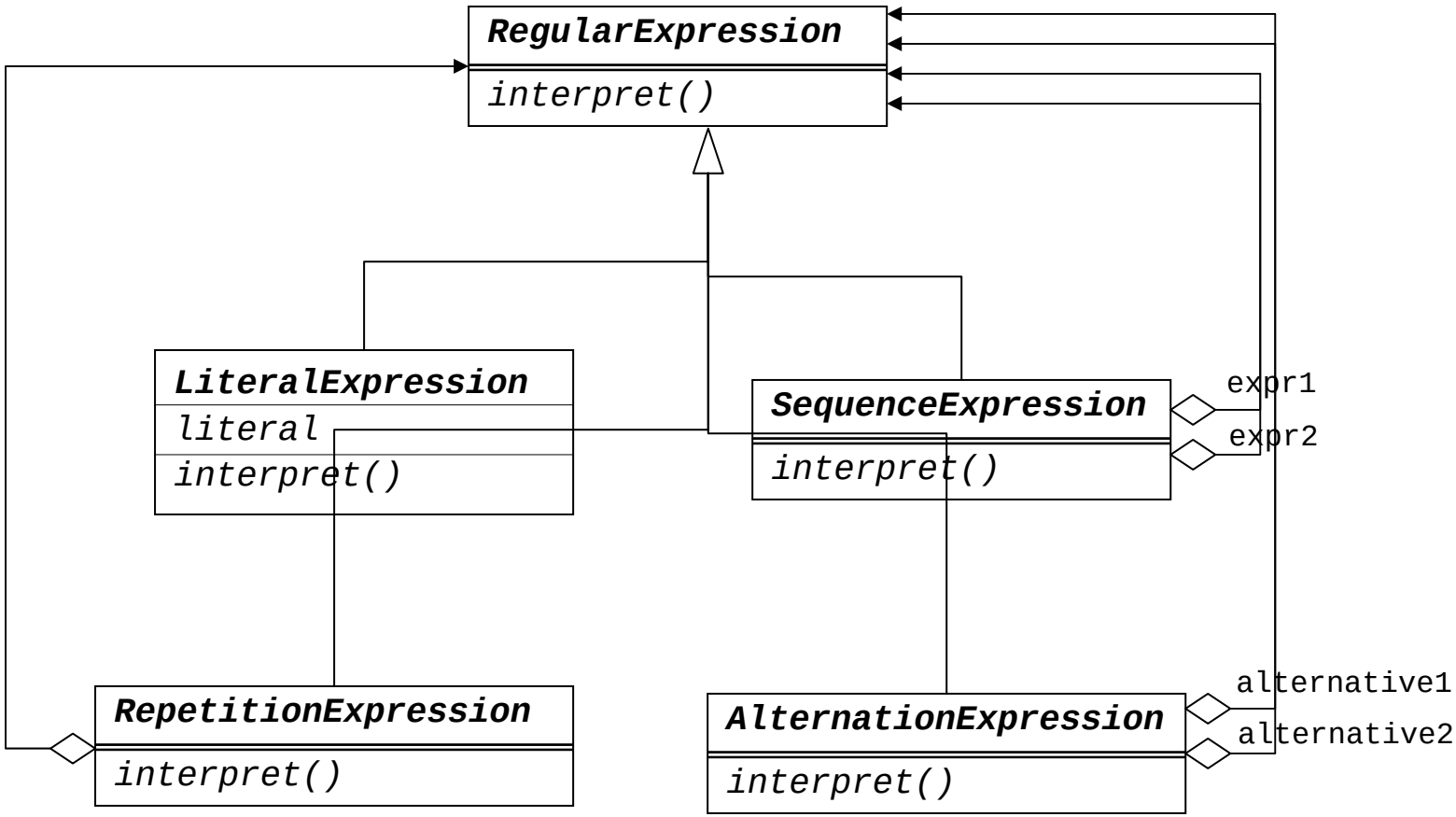
Interpreter

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

Problema

- Se comandos estão representados como objetos, eles poderão fazer parte de algoritmos maiores
  - Vários padrões repetitivos podem surgir nesses algoritmos
  - Operações como iteração ou condicionais podem ser frequentes: representá-las como objetos Command
- Solução em OO: elaborar uma gramática para calcular expressões compostas por objetos
  - Interpreter é uma extensão do padrão Command (ou um tipo de Command; ou uma micro-arquitetura construída com base em Commands) em que toda uma lógica de código pode ser implementada com objetos

Exemplo [GoF]


RegularExpression
interpret()
LiteralExpression
literal
interpret()
SequenceExpression
interpret()
expr1
expr2
RepetitionExpression
interpret()
AlternationExpression
interpret()
alternative1
alternative2

Questões

- *Qual a diferença entre Command e Interpreter?*
- *Cite exemplos de implementações na API Java e em frameworks que você conhece*

Resumo: quando usar?

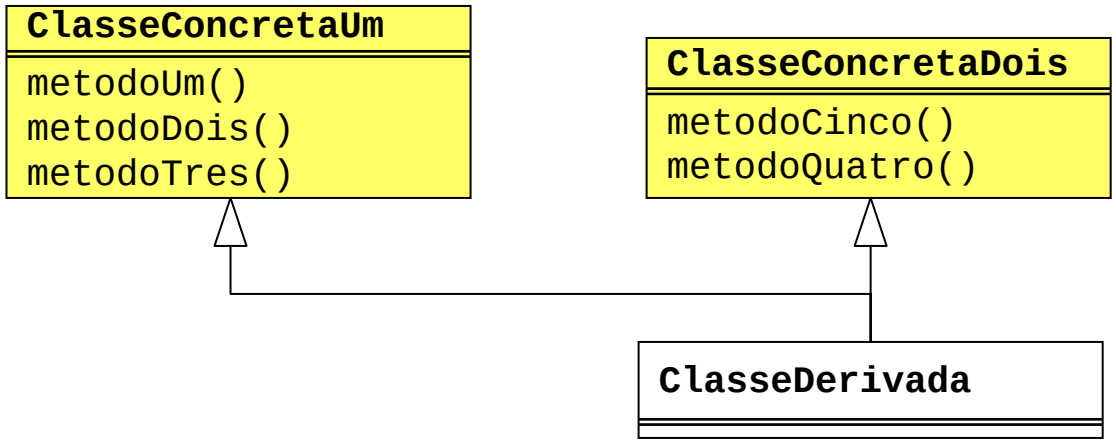
- *Template Method*
  - *Para compor um algoritmo feito por métodos abstratos que podem ser completados em subclasses*
- *State*
  - *Para representar o estado de um objeto*
- *Strategy*
  - *Para representar um algoritmo (comportamento)*
- *Command*
  - *Para representar um comando (ação imperativa do cliente)*
- *Interpreter*
  - *Para realizar composição com comandos e desenvolver uma linguagem de programação usando objetos*

Introdução: Extensão

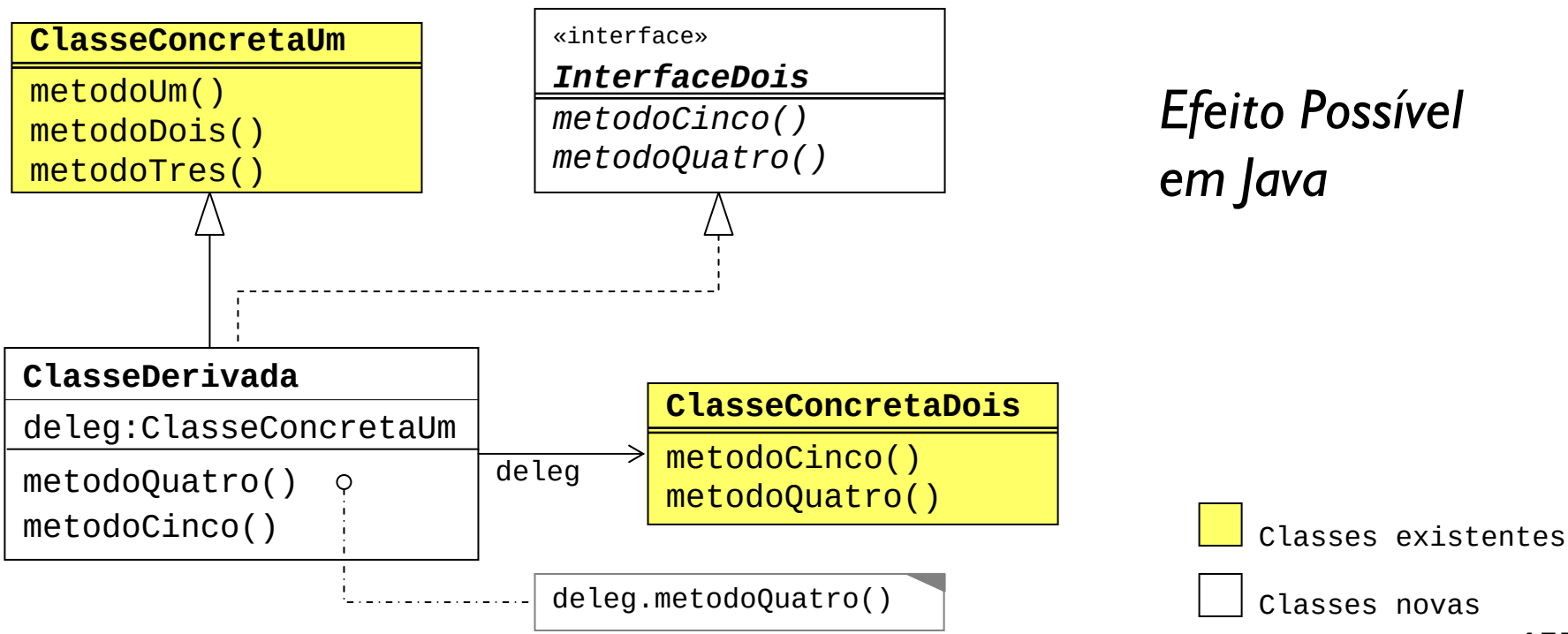
- *Extensão é a adição de uma classe, interface ou método a uma base de código existente* [2]
- *Formas de extensão*
  - *Herança (criação de novas classes)*
  - *Delegação (para herdar de duas classes, pode-se estender uma classe e usar delegação para "herdar" o comportamento da outra classe)*
- *Desenvolvimento em Java é sempre uma forma de extensão*
  - *Extensão começa onde o reuso termina*

Exemplo de extensão por delegação

Efeito Desejado



Efeito Possível em Java



Além da extensão

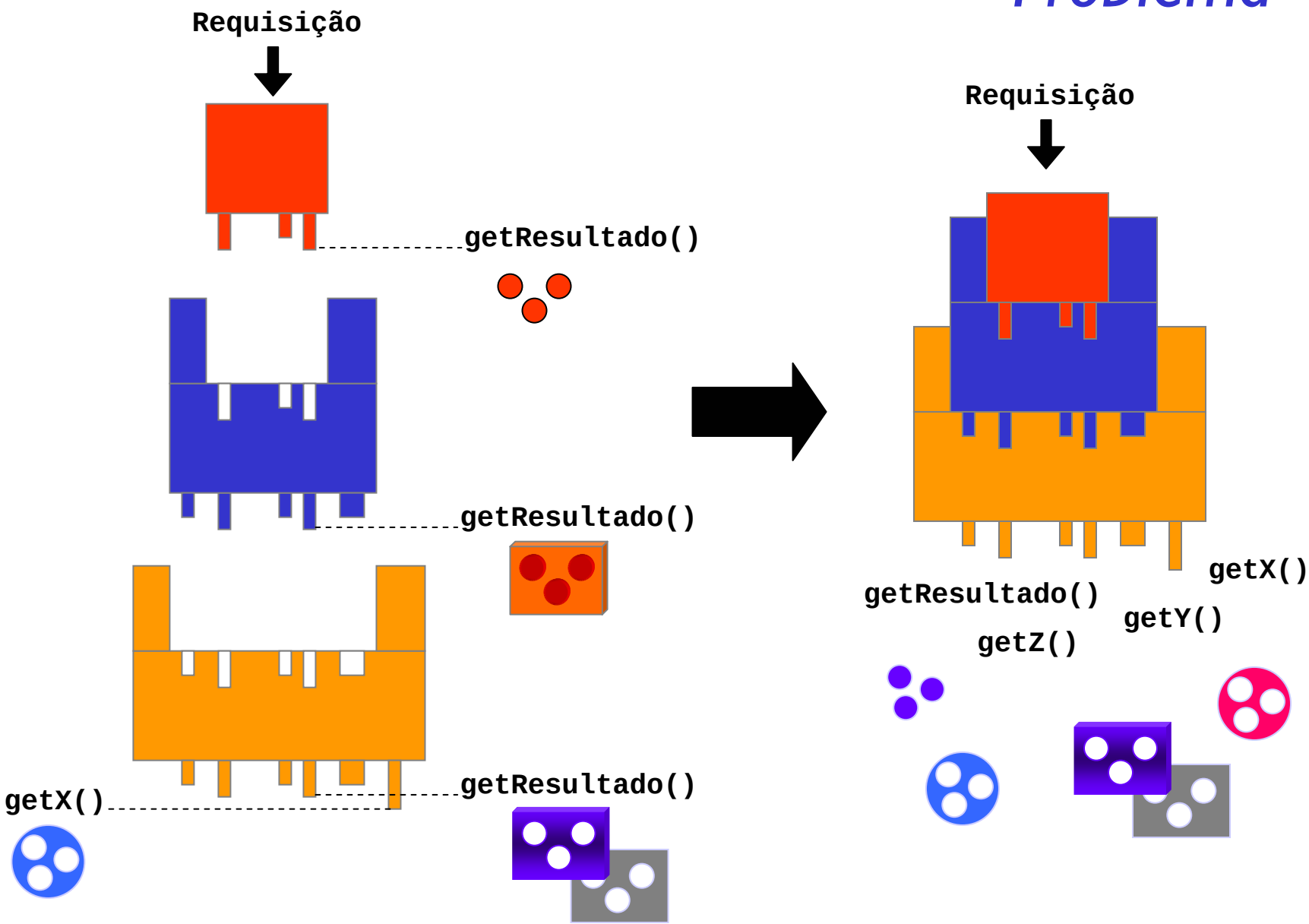
- *Tanto herança como delegação exigem que se saiba, em tempo de compilação, que comportamentos são desejados. Os patterns permitem acrescentar comportamentos em um objeto sem mudar sua classe*
- *Principais classes*
  - *Command (capítulo anterior)*
  - *Template Method (capítulo anterior)*
  - *Decorator: adiciona responsabilidades a um objeto dinamicamente.*
  - *Iterator: oferece uma maneira de acessar uma coleção de instâncias de uma classe carregada.*
  - *Visitor: permite a adição de novas operações a uma classe sem mudar a classe.*

21

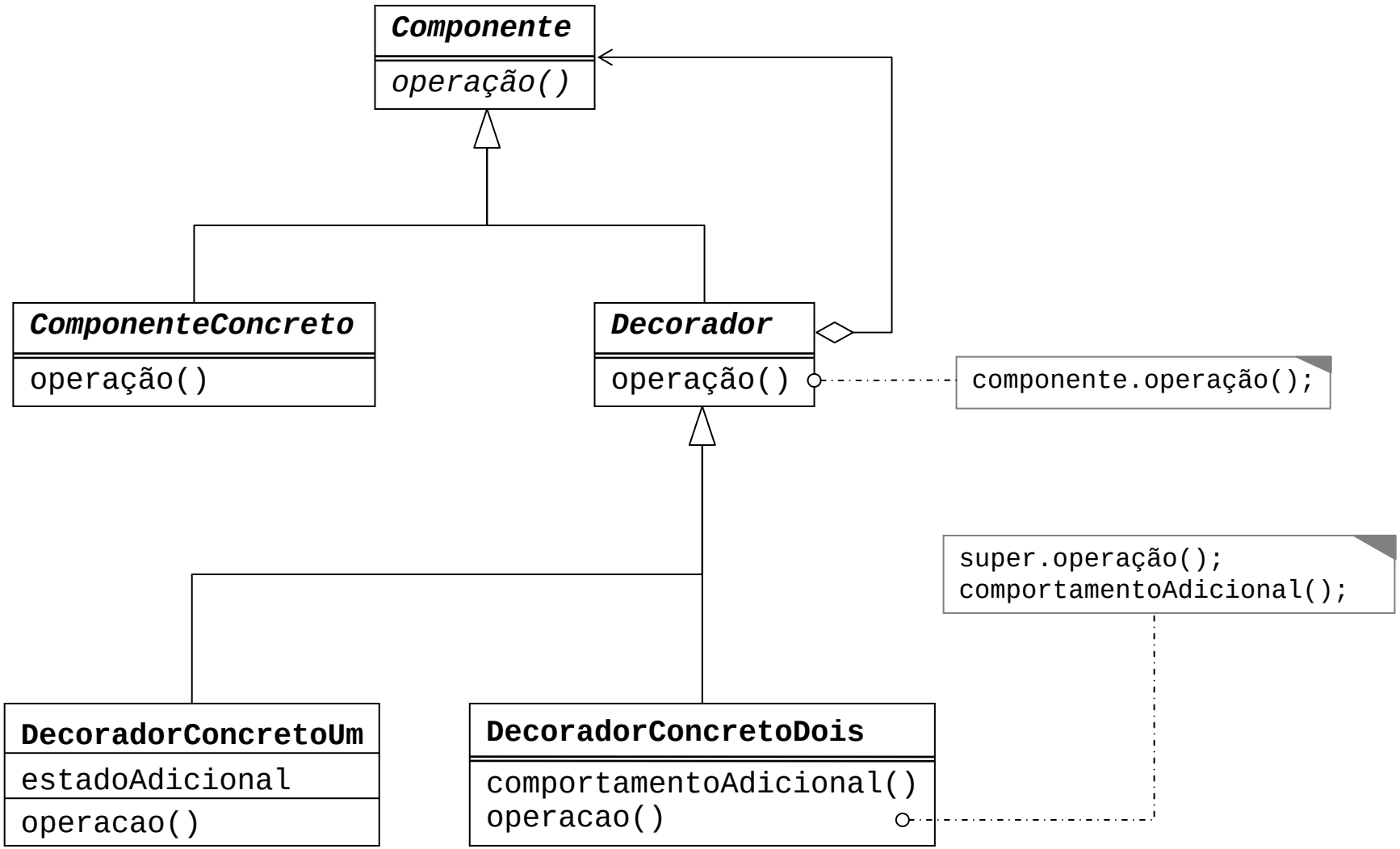
Decorator

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

Problema


Requisição
getResultado()
getResultado()
getX()
getResultado()
Requisição
getResultado()
getX()
getY()
getZ()

Estrutura de Decorator



Decorator em Java

```
public abstract class DecoradorConcretoUm extends Decorador {
   public DecoradorConcretoUm (Componente componente) {
      super(componente);
   }
   public String getDadosComoString() {
      return getDados().toString();
   }
   private Object transformar(Object o) {
      ...
   }
   public Object getDados() {
     return transformar(getDados());
   }
   public void operacao(Object arg) {
     // ... comportamento adicional
     componente.operacao(arg);
   }
}
```

```
public abstract class DecoradorConcretoUm
                                  extends Decorador {
   private Object estado;
   public DecoradorConcretoUm (Componente comp,
                                  Object estado) {
      super(comp);
      this.estado = estado;
   }
   ...
   public void operacao(Object arg) {
     // ... comportamento adicional
     super.operacao(estado);
     // ...
   }
}
```

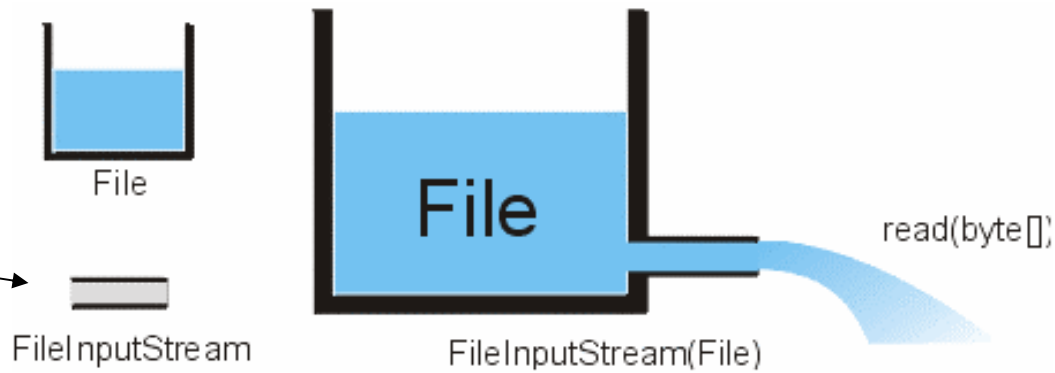
```
public class ComponenteConcreto implements Componente {
   private Object dados;
   public Object getDados() {
     return dados;
   }
   public void operacao(Object arg) {
     ...
   }
}
```

```
public interface Componente {
   Object getDados();
   void operacao(Object arg);
}
```

```
public abstract class Decorador implements Componente {
   private Componente componente;
   public Decorador(Componente componente) {
      this.componente = componente;
   }
   public Object getDados() {
     return componente.getDados();
   }
  public void operacao(Object arg) {
     componente.operacao(arg);
   }
}
```

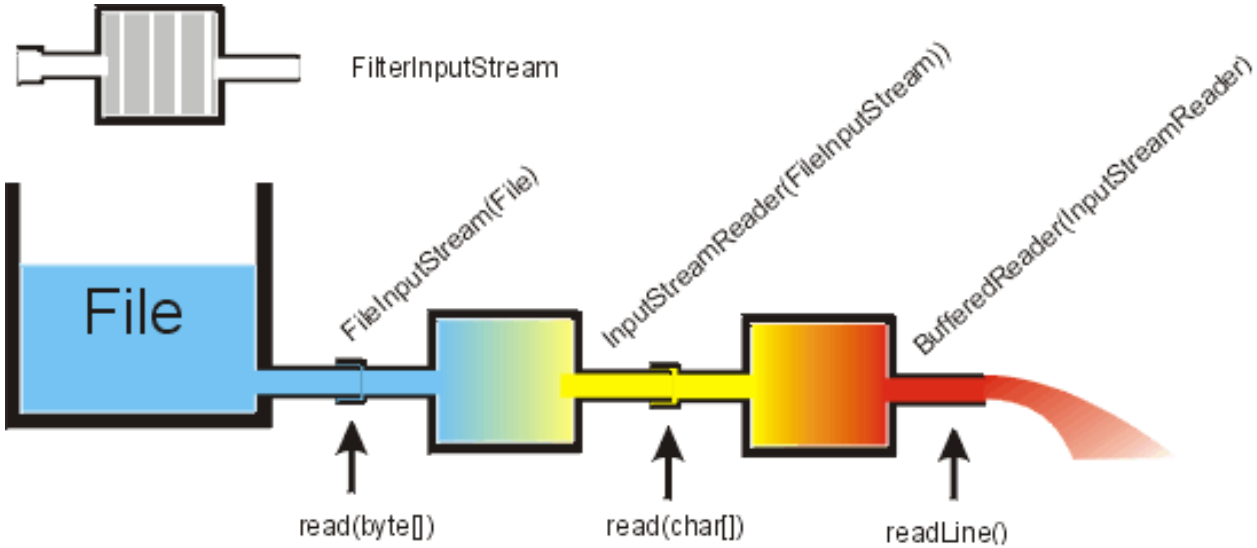
Exemplo: I/O Streams

Um componente concreto




```
// 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



Concatenação do decorador

```
// 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();
```

Uso de método com comportamento alterado

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

Comportamento adicional

Questões

- *Cite outros exemplos de Decorator no J2SDK e em frameworks que você conhece*

22

Iterator

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

Problema

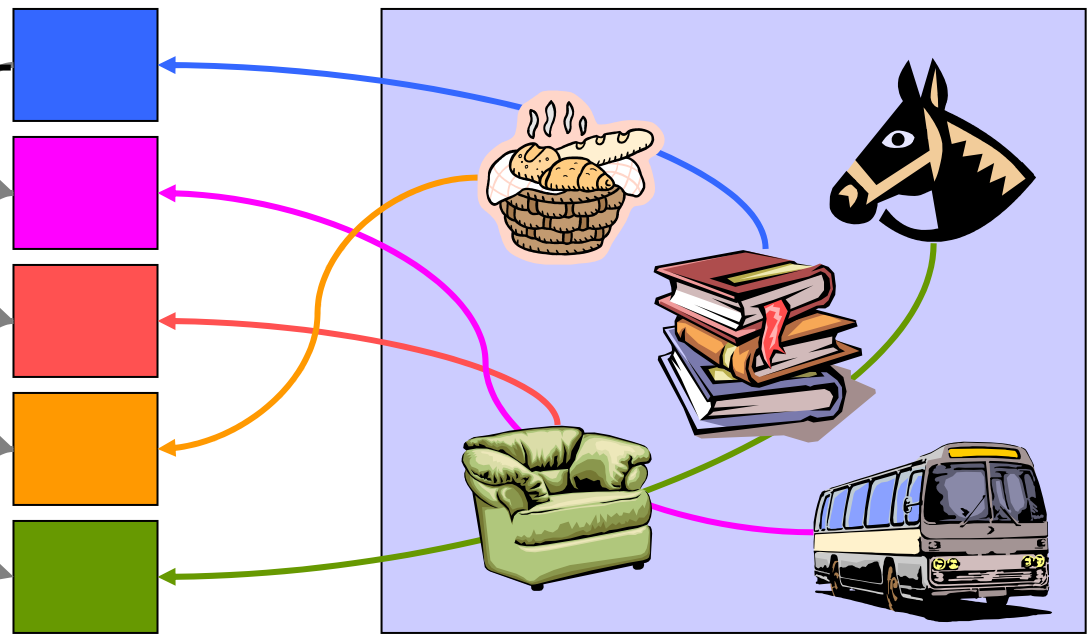
Tipo de referência é genérico

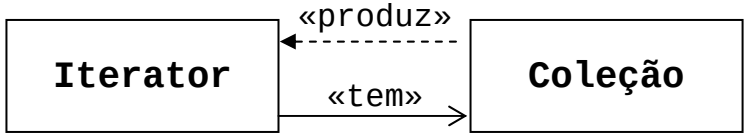
Iterator

Coleção arbitrária de objetos (array, hashmap, lista, conjunto, pilha, tabela, ...)

```
Object o =
iterator.next()
```

```
iterator.hasNext() ?
```





Para que serve?

- *Iterators servem para acessar o conteúdo de um agregado sem expor sua representação interna*
- *Oferece uma interface uniforme para atravessar diferentes estruturas agregadas*
- *Iterators são implementados nas coleções do Java. É obtido através do método iterator() de Collection, que devolve uma instância de java.util.Iterator.*
- *Interface java.util.Iterator:*

```
package java.util;
public interface Iterator<E> {
   boolean hasNext();
   Object next();
   void remove();
}
```

- *iterator() é um exemplo de Factory Method*

Questões

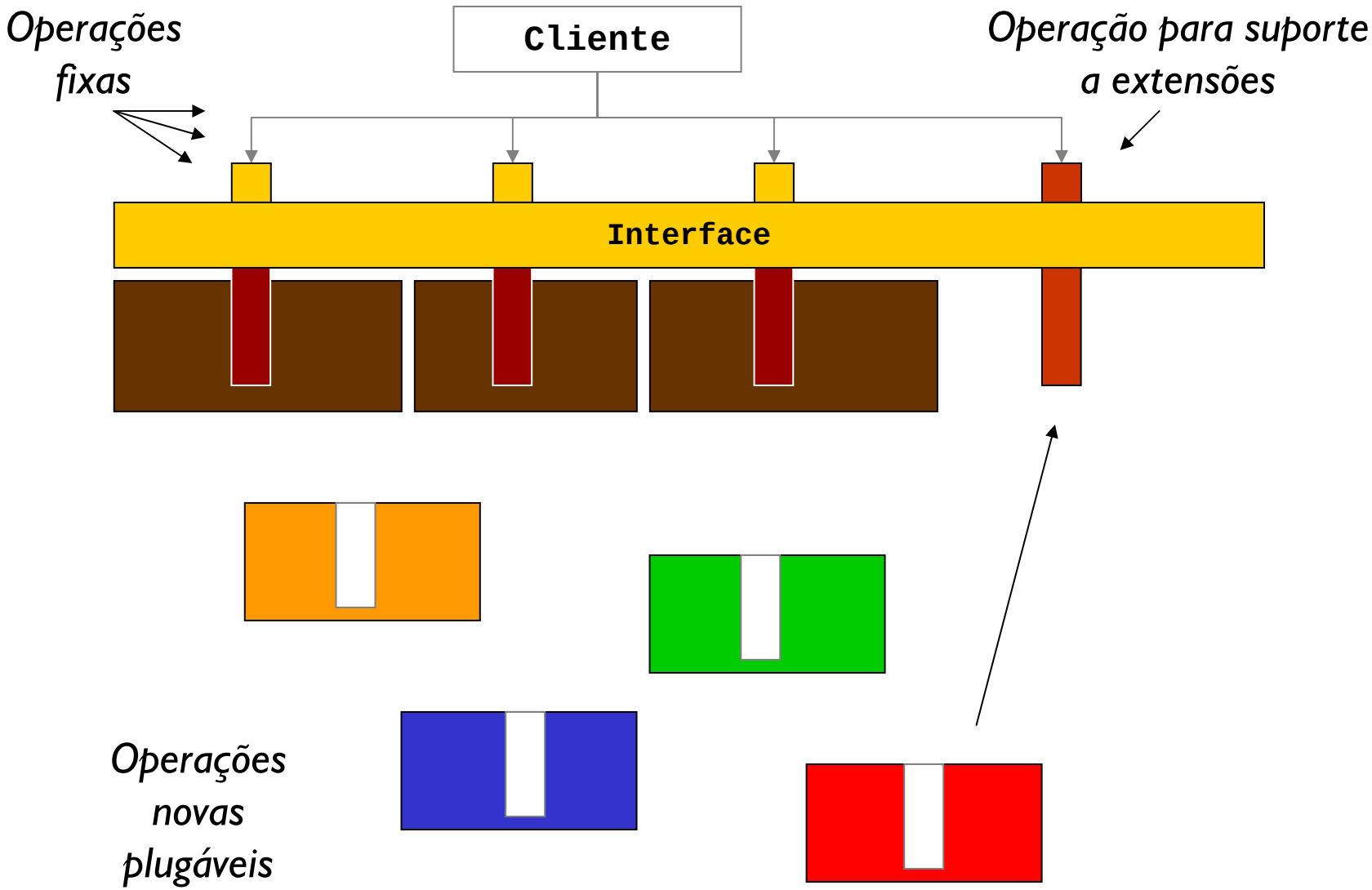
- *Quantos iterators você conhece nas APIs da linguagem Java (além de java.util.Iterator)?*

23

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]

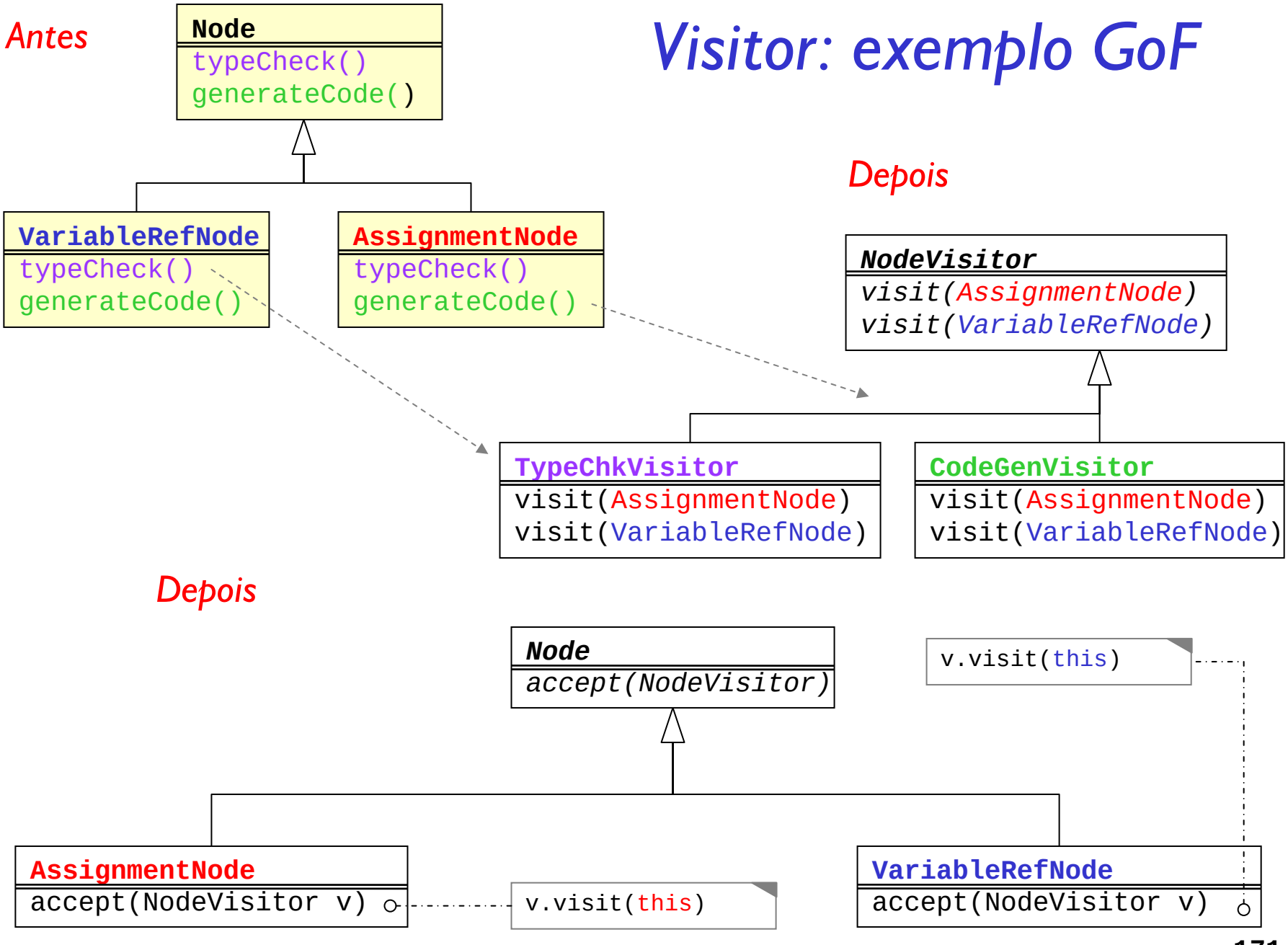
Problema


Operações
fixas
Cliente
Operação para suporte
a extensões
Interface
Operações
novas
plugáveis

Para que serve?

- *Visitor permite*
  - *Plugar nova funcionalidade em objetos sem precisar mexer na estrutura de herança*
  - *Agrupar e manter operações relacionadas em uma classe e aplicá-las, quando conveniente, a outras classes (evitar espalhamento e fragmentação de interesses)*
  - *Implementar um Iterator para objetos não relacionados através de herança*

Visitor: exemplo GoF



Estrutura de Visitor

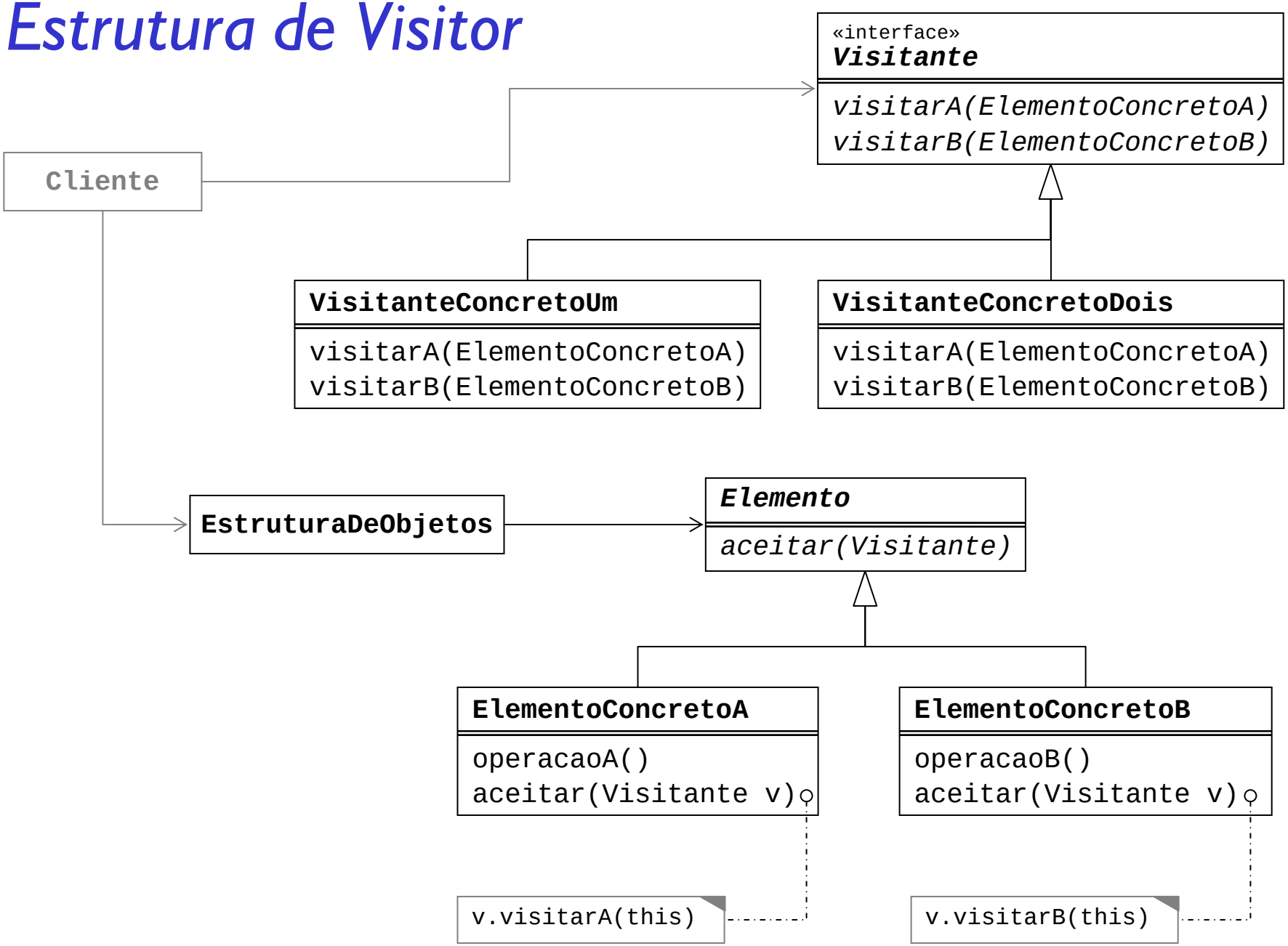
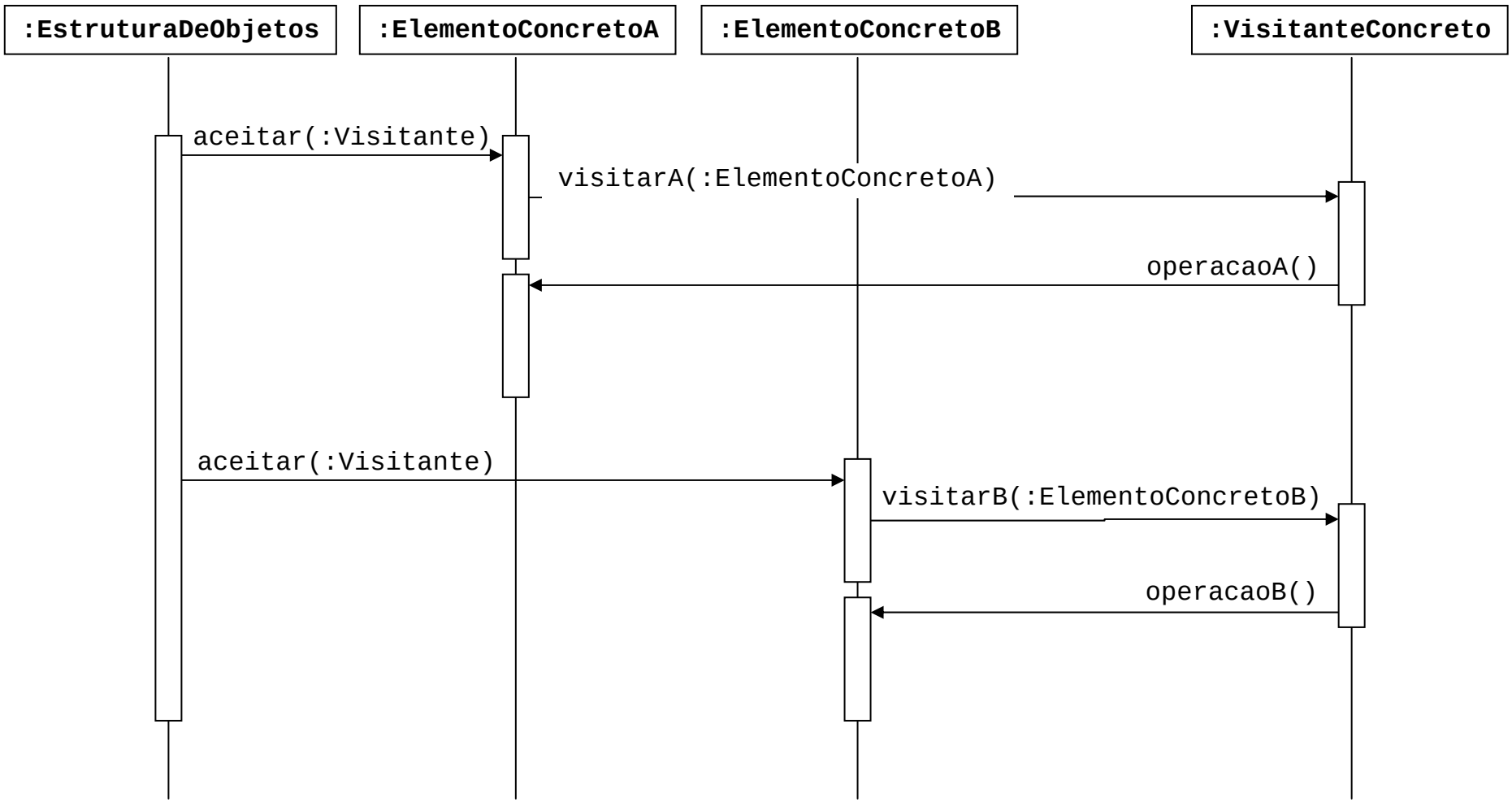


Diagrama de seqüência



Refatoramento para Visitor em Java: ***Antes***

```
public interface Documento_1 {
  public void gerarTexto();
  public void gerarHTML();
  public boolean validar();
}
```

É talvez o mais complexo dos padrões GoF

```
public class Cliente {
  public static void main(String[] args) {
    Documento_1 doc = new Texto_1();
    Documento_1 doc2 = new Grafico_1();
    Documento_1 doc3 = new Planilha_1();
    doc.gerarTexto();
    doc.gerarHTML();
    if (doc.validar())
      System.out.println(doc + " valido!");
    doc2.gerarTexto();
    doc2.gerarHTML();
    if (doc2.validar())
      System.out.println(doc2 + " valido!");
    doc3.gerarTexto();
    doc3.gerarHTML();
    if (doc3.validar())
      System.out.println(doc3 + " valido!");

  }
}
```

```
public class Texto_1
           implements Documento_1 {
  public void gerarTexto() {...}
  public void gerarHTML() {...}
  public boolean validar() {...}
         ...
}
```

```
public class Planilha_1
          implements Documento_1 {
  public void gerarTexto() {...}
  public void gerarHTML() {...}
  public boolean validar() {...}
         ...
}
```

```
public class Grafico_1
             implements Documento_1 {
  public void gerarTexto() {
    System.out.println("Nao impl.");
  }
  public void gerarHTML() {
    System.out.println("HTML gerado");
        }
  public boolean validar() {
    return true;
  }
  public String toString() {
    return "Grafico";
  }
}
```

Visitor em Java (Depois)

```
public interface Visitante {
  public Object visitar(Planilha p);
  public Object visitar(Texto t);
  public Object visitar(Grafico g);
}
```

```
public class GerarHTML implements Visitante {
  public Object visitar(Planilha p) {
    p.gerarHTML(); return null;  }
  public Object visitar(Texto t) {
    t.gerarHTML(); return null;  }
  public Object visitar(Grafico g) {
    g.gerarPNG();   }
}
```

```
public class Validar implements Visitante {
  public Object visitar(Planilha p) {
    return new Boolean(true);     }
  public Object visitar(Texto t) {
    return new Boolean(true);     }
  public Object visitar(Grafico g) {
    return new Boolean(true);     }
}
```

```
public class Cliente {
  public static void main(String[] args) {
    Documento doc = new Texto();
    doc.aceitar(new GerarTexto());
    doc.aceitar(new GerarHTML());
    if (((Boolean)doc.aceitar(
      new Validar())).booleanValue()) {
      System.out.println(doc + " valido!");
    }
  }
}
```

```
public interface Documento {
  public Object aceitar(Visitante v);
}
```

```
public class Planilha implements Documento {
  public Object aceitar(Visitante v) {
    return v.visitar(this);
  }
  public void gerarHTML() {...}
  public void gerarTexto() {...}
  public String toString() {...}
}
```

```
public class Texto implements Documento {
  public Object aceitar(Visitante v) {
    return v.visitar(this);
  }
  public void gerarHTML() {...}
  public void gerarTexto() {...}
  public String toString() {...}
}
```

```
public class Grafico implements Documento {
  public Object aceitar(Visitante v) {
    return v.visitar(this);
  }
  public void gerarPNG() {...}
  public String toString() {...}
}
```

Prós e contras

- Vantagens
  - Facilita a adição de novas operações
  - Agrupa operações relacionadas e separa operações não relacionadas: reduz espalhamento de funcionalidades e embaralhamento
- Desvantagens
  - Dá trabalho adicionar novos elementos na hierarquia: requer alterações em todos os Visitors. Se a estrutura muda com frequência, não use!
  - Quebra de encapsulamento: métodos e dados usados pelo visitor têm de estar acessíveis
- Alternativas ao uso de visitor
  - Aspectos (www.aopalliance.org e www.aspectj.org)
  - Hyperslices (www.research.ibm.com/hyperspace/)

Resumo: quando usar?

- *Decorator*
  - *Para acrescentar recursos e comportamento a um objeto existente, receber sua entrada e poder manipular sua saída.*
- *Iterator*
  - *Para navegar em uma coleção elemento por elemento*
- *Visitor*
  - *Para estender uma aplicação com novas operações sem que seja necessário mexer na interface existente.*

Você sabe distinguir os padrões GoF

- *Faça alguns testes!*

Padrões GRASP

Padrões básicos

- Information Expert
- Creator
- High Cohesion
- Low Coupling
- Controller

Padrões avançados

- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

Padrões GRASP refletem práticas mais pontuais da aplicação de técnicas OO

Padrões GoF exploram soluções mais específicas

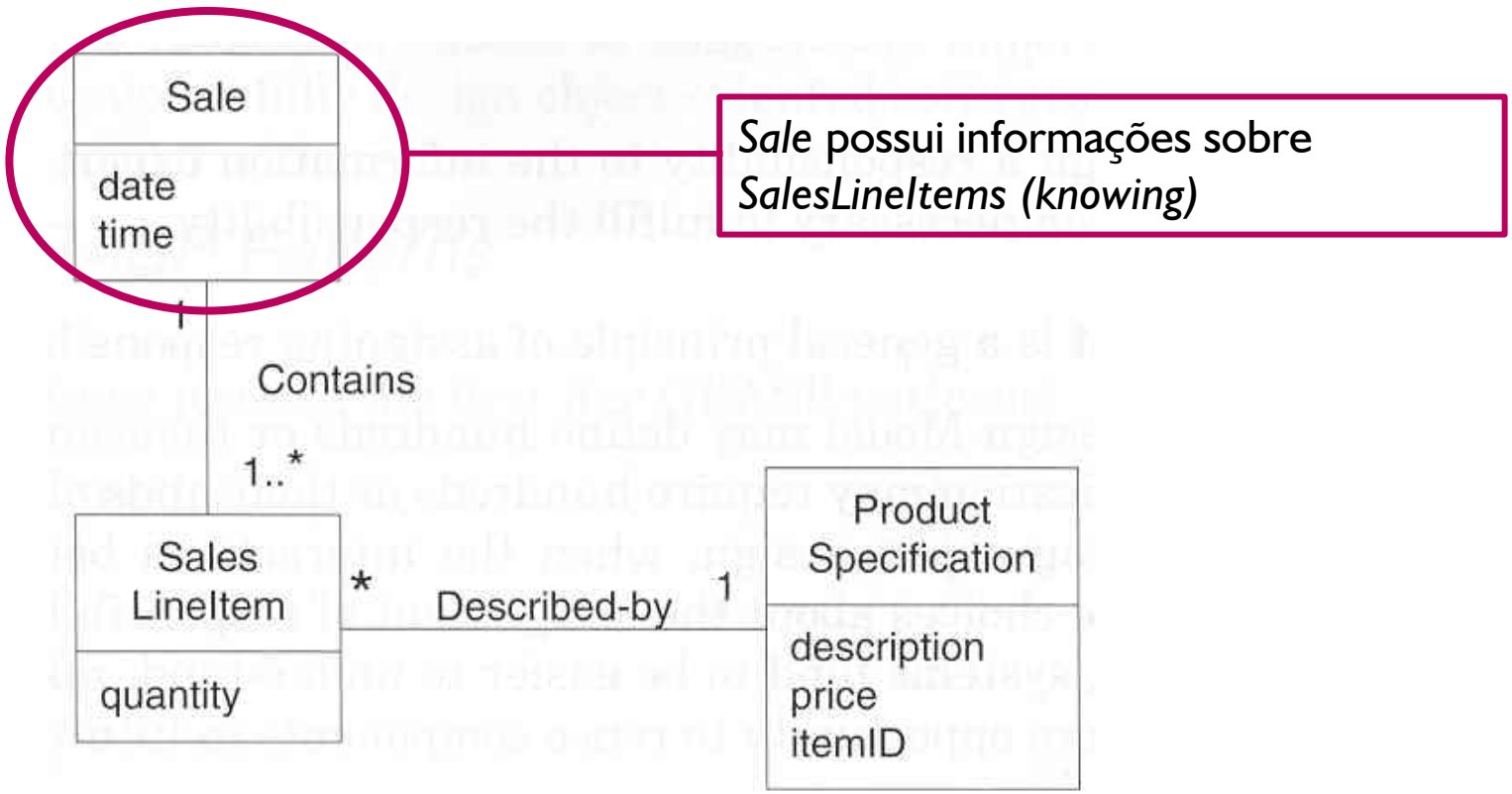
Padrões GRASP ocorrem na implementação de vários padrões GoF

Expert (especialista de informação)

- *Problema*
  - *Precisa-se de um princípio geral para atribuir responsabilidades a objetos*
  - *Durante o design, quando são definidas interações entre objetos, fazemos escolhas sobre a atribuição de responsabilidades a classes*
- *Solução*
  - *Atribuir uma responsabilidade ao especialista de informação: classe que possui a informação necessária para cumpri-la*
  - *Comece a atribuição de responsabilidades ao declarar claramente a responsabilidade*

Expert

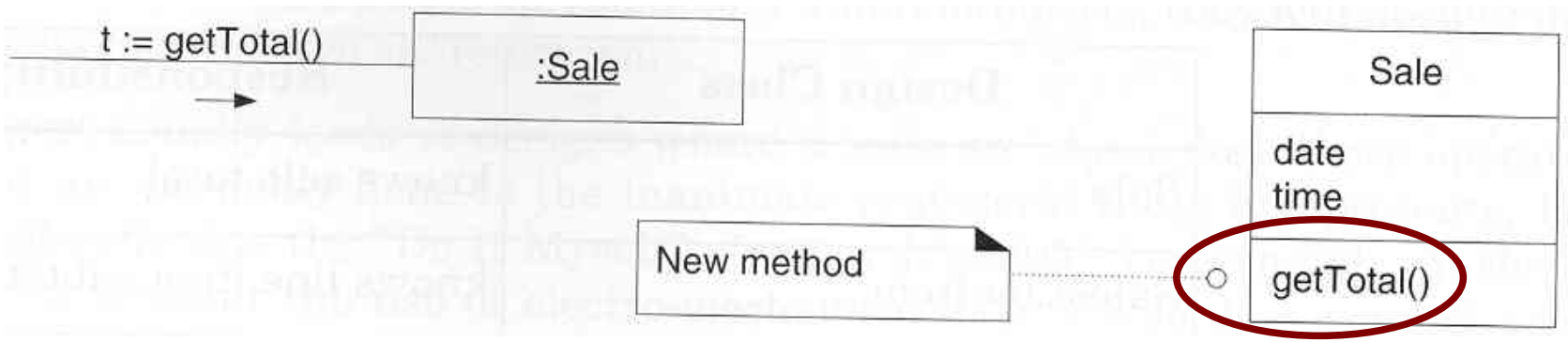
- *No sistema abaixo, uma classe precisa saber o total geral de uma venda (Sale). Que classe deve ser a responsável?*



Fonte: [4]

Expert (2)

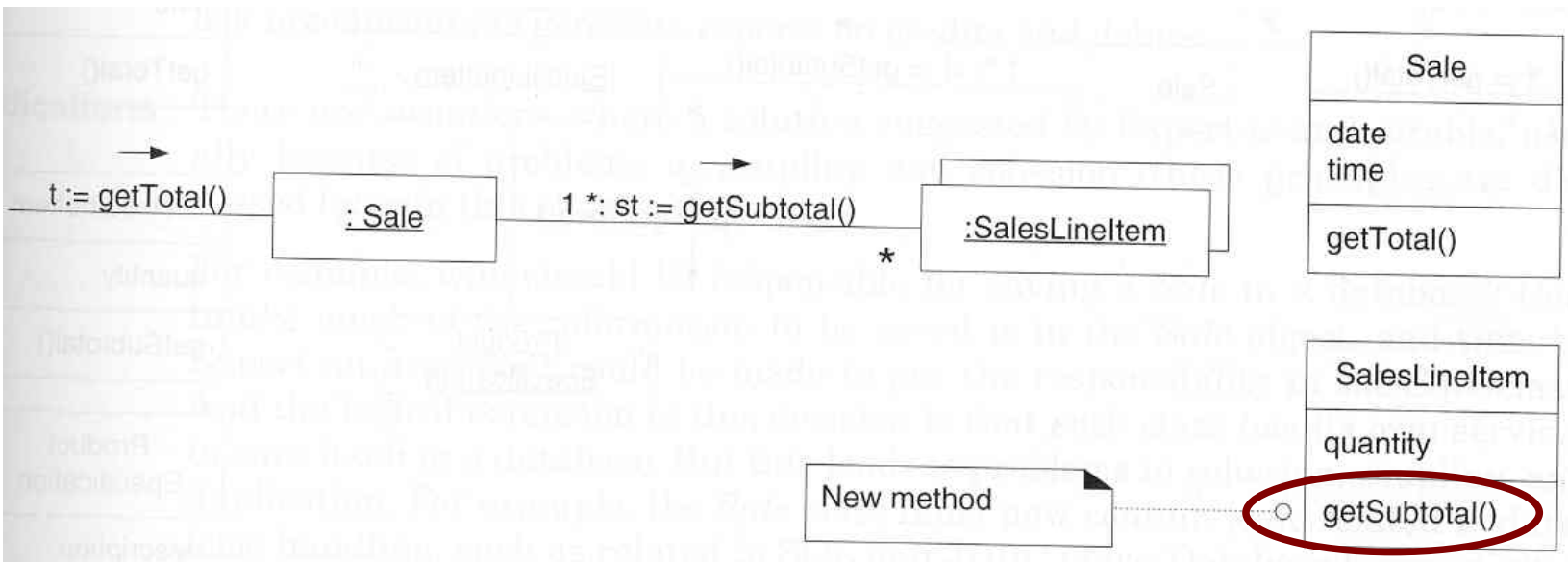
- *A nova responsabilidade é conduzida por uma operação no driagrama de interação*
- *Um novo método é criado*



Fonte: [4]

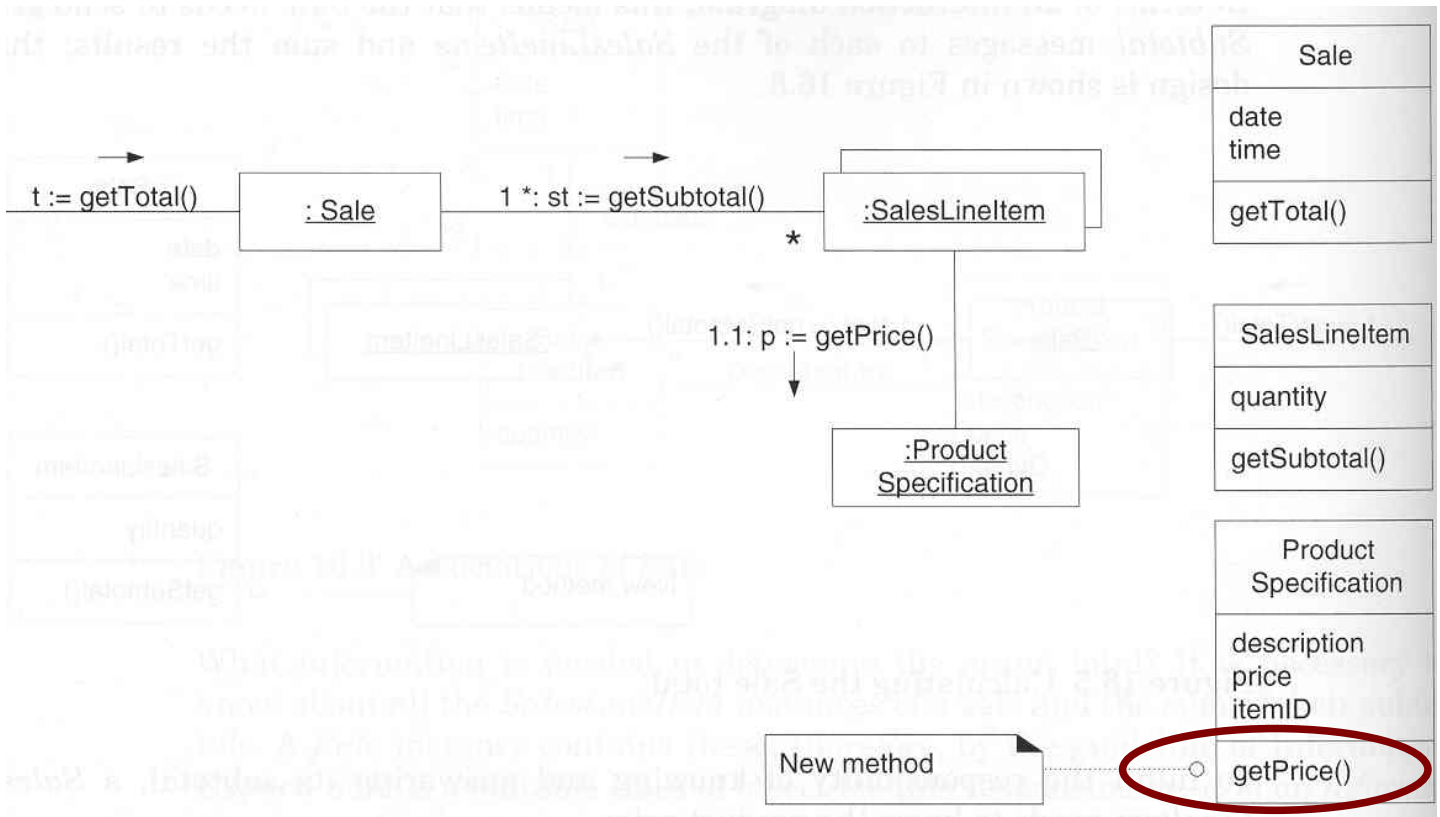
Expert(3)

- *Mas como a classe Sale vai calcular o valor total?*
  - *Somando os subtotais. Mas quem faz isto?*
- *A responsabilidade para cada subtotal é atribuída ao objeto* SalesLineItem (item de linha do pedido)



Expert (4)

- *O subtotal depende do preço. O objeto ProductSpecification é o especialista que conhece o preço, portanto a responsabilidade é dele.*

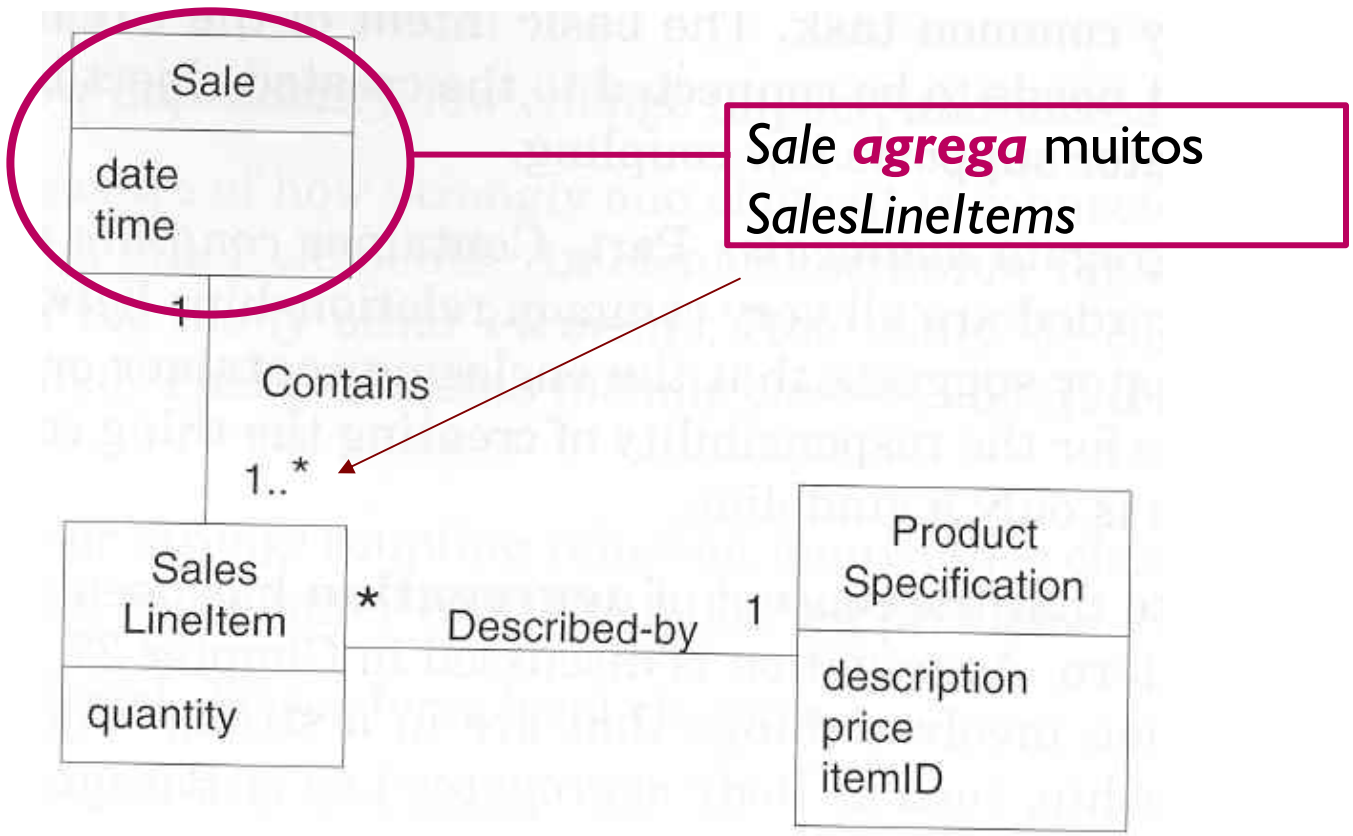


Creator

- ***Problema**: Que classe deve ser responsável pela criação de uma nova instância de uma classe?*
- ***Solução**: Atribua a B a responsabilidade de criar A se:*
  - *B agrega A objetos*
  - *B contém A objetos*
  - *B guarda instâncias de A objetos*
  - *B faz uso de A objetos*
  - *B possui dados para inicialização que será passado para A quando ele for criado.*

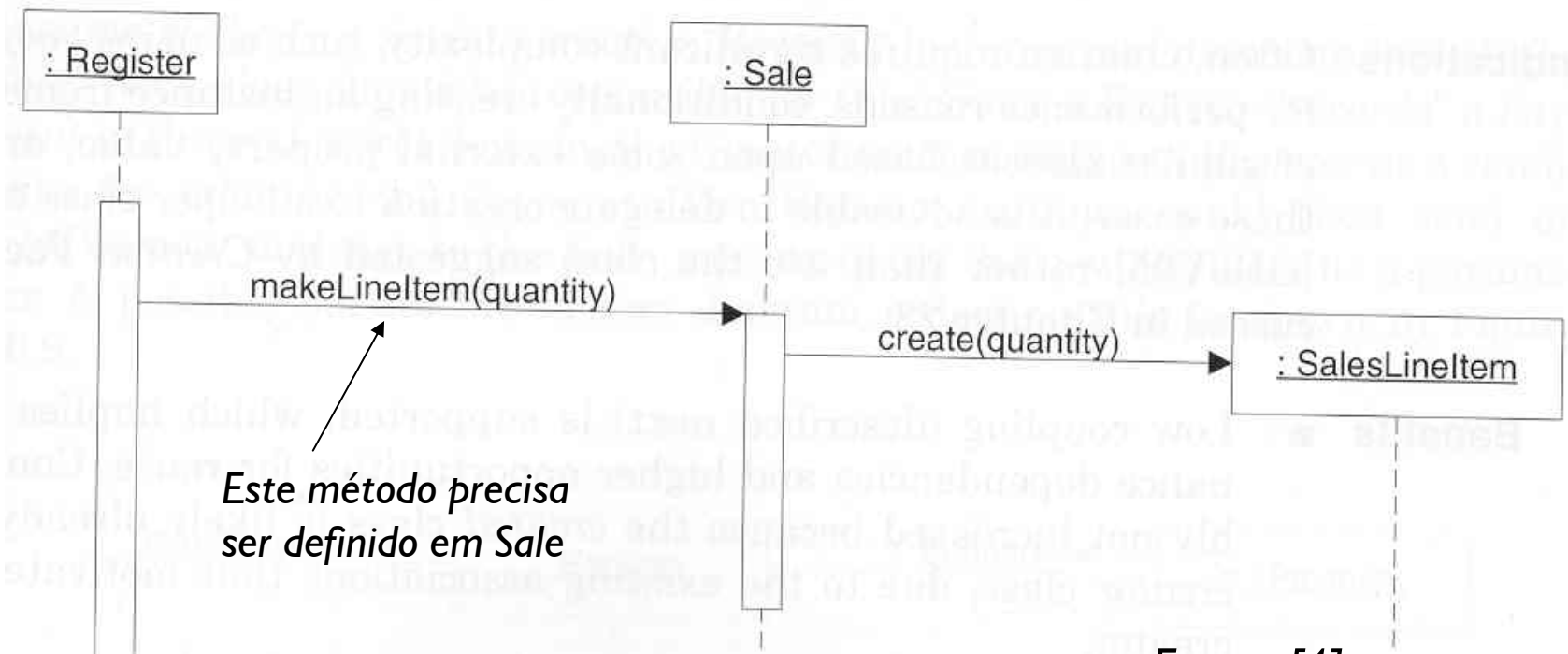
Creator

- *Que classe deve ser responsável por criar uma instância do objeto SalesLineItem abaixo?*



Creator

- *A nova responsabilidade é conduzida por uma operação em um diagrama de interações*
  - *Um novo método é criado na classe de design para expressar isto.*



Fonte: [4]

High Cohesion

- *Problema*
  - *Como manter a complexidade sob controle?*
  - *Classes que fazem muitas tarefas não relacionadas são mais difíceis de entender, de manter e de reusar, além de serem mais vulneráveis à mudança.*
- *Solução*
  - *Atribuir uma responsabilidade para que a coesão se mantenha alta.*

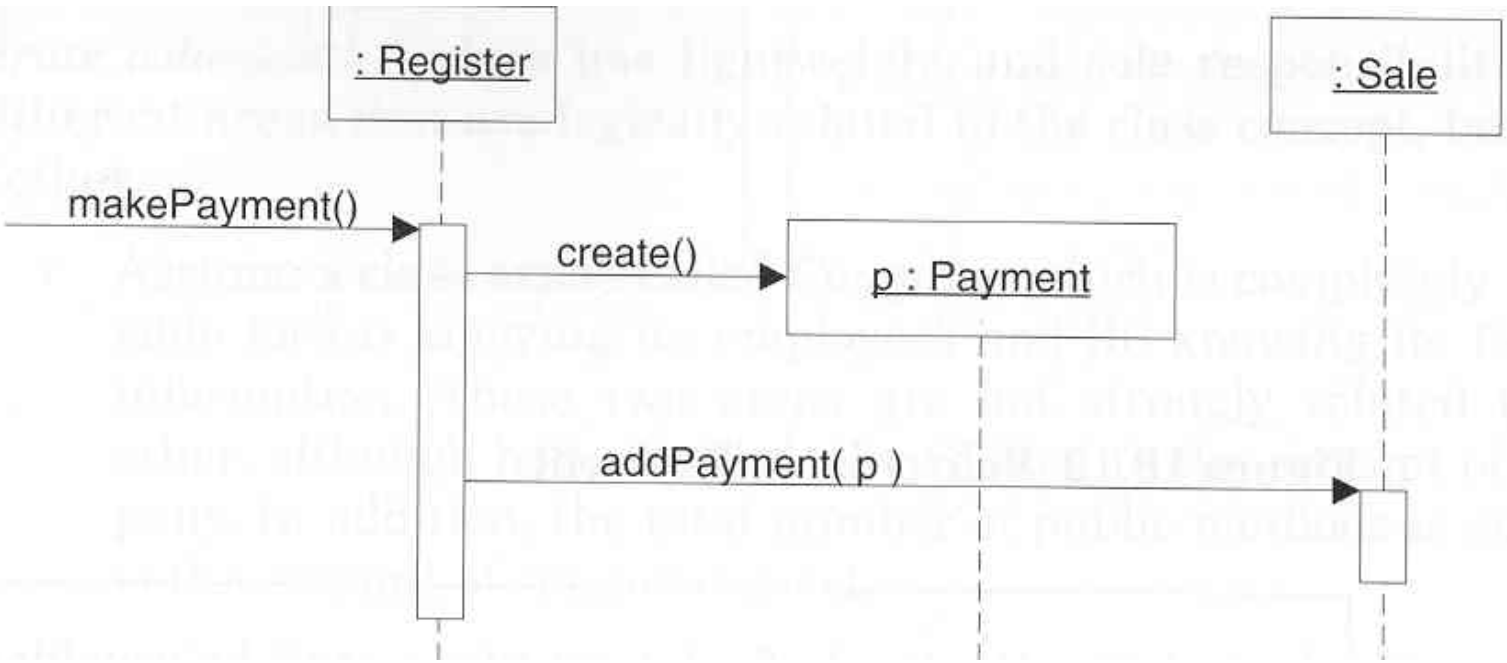
Você sabe o que é coesão?

Coesão

- Coesão [Funcional]
  - Uma medida de quão relacionadas ou focadas estão as responsabilidades de um elemento.
- Exemplo
  - Uma classe Cão é coesa se tem operações relacionadas ao Cão (morder, correr, comer, latir) e apenas ao Cão (não terá por exemplo, validar, imprimirCao, listarCaes)
  - Alta coesão promove design modular

High Cohesion

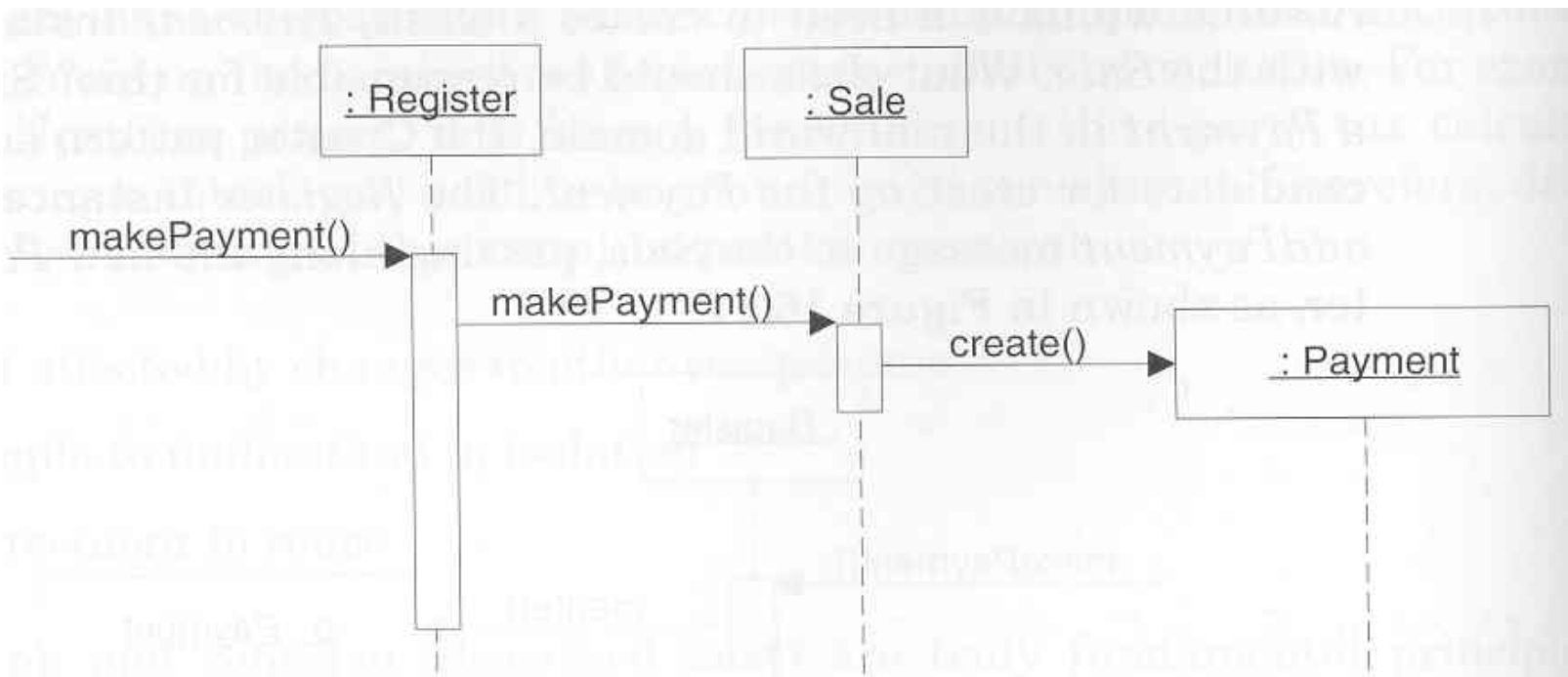
- *Que classe é responsável por criar um pagamento (Payment) e associá-lo a uma venda (Sale)?*



- *Register assumiu responsabilidade por uma coisa que é parte de Sale (fazer um pagamento não é responsabilidade de registrar)*

High Cohesion

- *Nesta solução, Register delega a responsabilidade a Sale, diminuindo aumentando a coesão de Register*
  - *A criação do processo de pagamento agora é responsabilidade da venda e não do registro. Faz mais sentido pois o pagamento é parte de Sale.*



Low Coupling (baixo acoplamento)

- *Problema*
  - *Como suportar baixa dependência, baixo impacto devido a mudanças e reuso constante?*
- *Solução*
  - *Atribuir uma responsabilidade para que o acoplamento mantenha-se fraco.*
- *Acoplamento*
  - *É uma medida de quanto um elemento está conectado a, ou depende de outros elementos*
  - *Uma classe com acoplamento forte depende de muitas outras classes: tais classes podem ser indesejáveis*
  - *O acoplamento está associado à coesão: maior coesão, menor acoplamento e vice-versa.*

Low Coupling

- *Como devemos atribuir uma responsabilidade para criar Payment e associá-lo com Sale?*

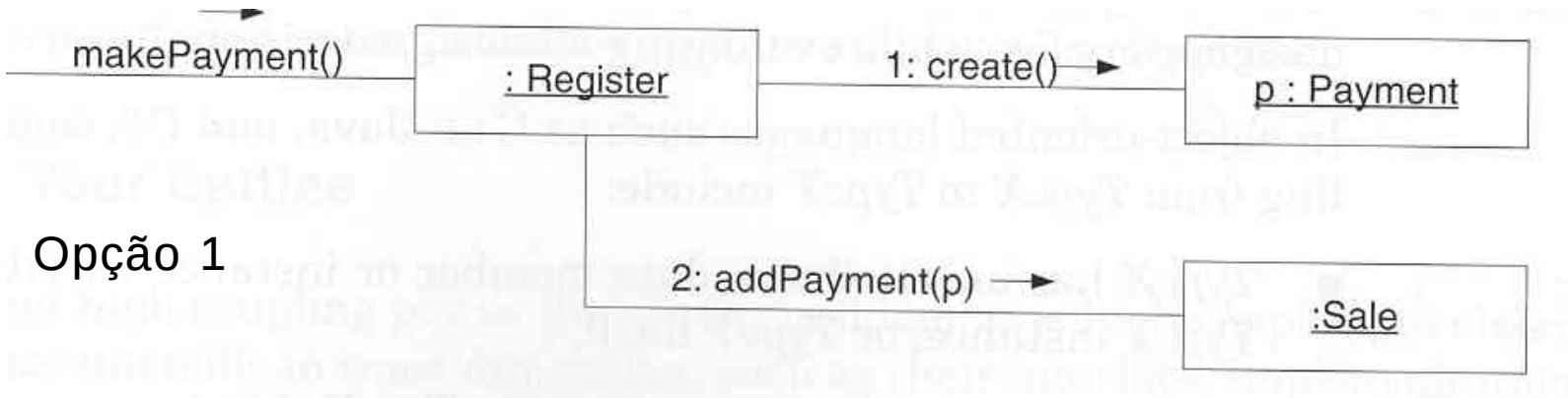
Payment

Register

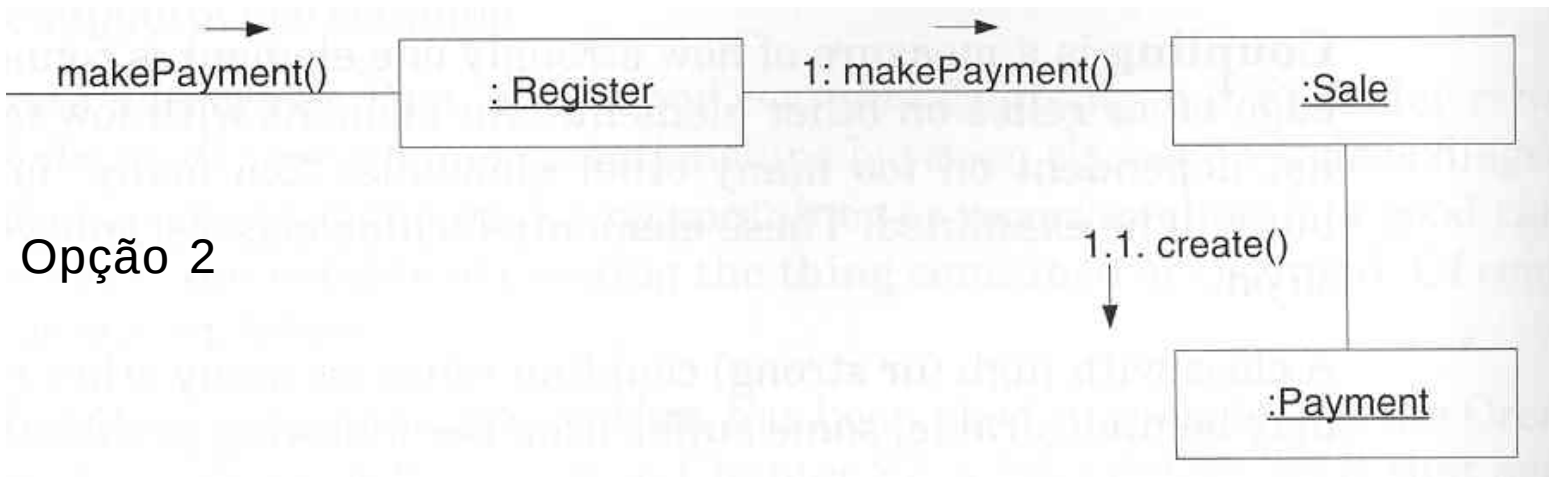
Sale

Low Coupling

- *Qual das opções abaixo suporta o menor acoplamento?*



Opção 1



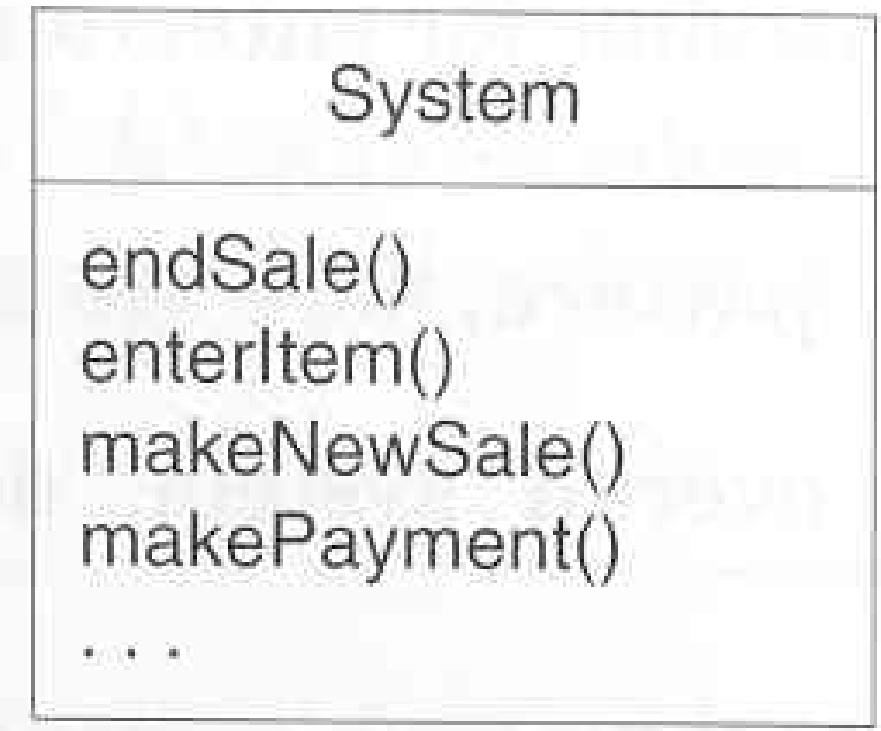
Opção 2

Controller

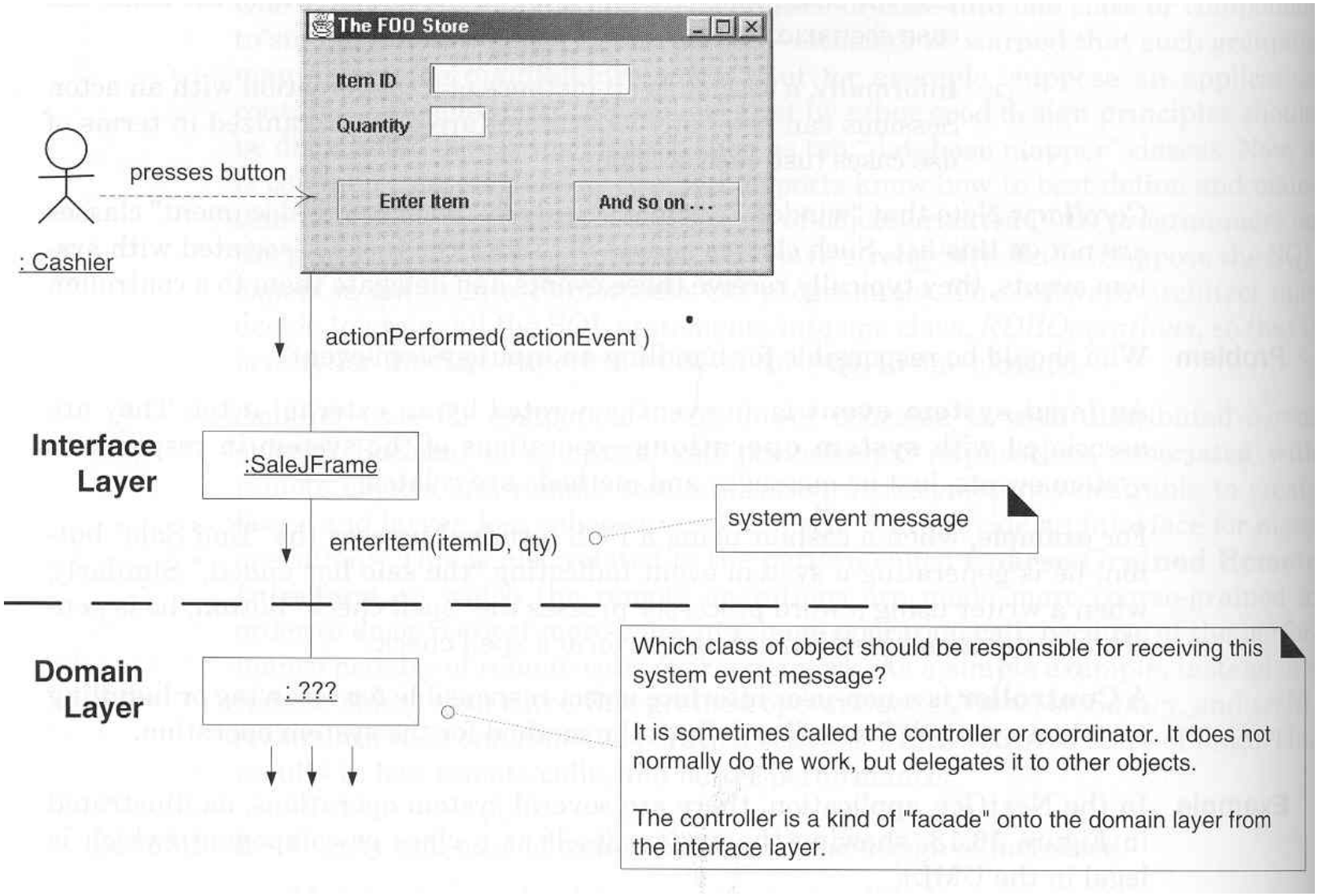
- *Problema*
  - *Quem deve ser o responsável por lidar com um evento de uma interface de entrada?*
- *Solução*
  - *Atribuir responsabilidades para receber ou lidar com um evento do sistema para uma classe que representa todo o sistema (controlador de fachada – front controller), um subsistema e um cenário de caso de uso (controlador de caso de uso ou de sessão).*
- *Este padrão é semelhante (ou equivalente) a um padrão GoF. Qual?*

Controller

- *Um sistema contendo operações "de sistema" associados com eventos do sistema.*

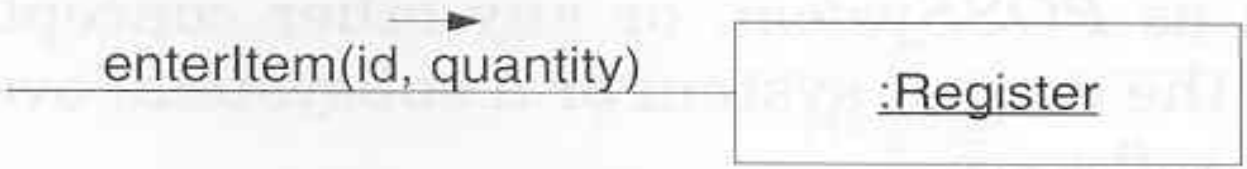


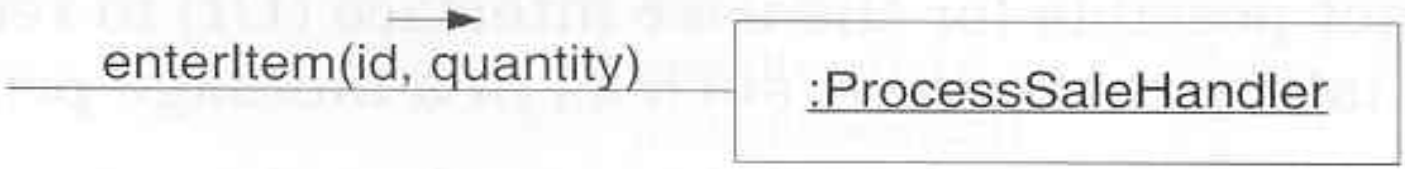
Controller: problema



Controller: solução

- A primeira solução representa o sistema inteiro
- A segunda solução representa o destinatário ou handler de todos os eventos de um caso de uso





O diagrama acima não mostra o Controller, mas a transparência da comunicação (chamada do cliente, objeto que recebe a mensagem)

Outros padrões

- Padrões GRASP avançados
- Domine primeiro os cinco básicos antes de explorar estes quatro:
  - Polymorphism
  - Indirection
  - Pure Fabrication
  - Protected Variations
- Outros padrões
  - Dependency Injection (aplicação de Indirection)
  - Aspectos

GRASP: Polymorphism

- *Problema:*
  - *Como lidar com alternativas baseadas no tipo? Como criar componentes de software plugáveis?*
  - *Deseja-se evitar variação condicional (if-then-else): pouco extensível.*
  - *Deseja-se substituir um componente por outro sem afetar o cliente.*
- *Solução*
  - *Não use lógica condicional para realizar alternativas diferentes baseadas em tipo. Atribua responsabilidades ao comportamento usando operações polimórficas*
  - *Refatore!*

GRASP: Pure Fabrication

- *Problema*
  - *Que objeto deve ter a responsabilidade, quando você não quer violar High Cohesion e Low Coupling, mas as soluções oferecidas por Expert não são adequadas?*
  - *Atribuir responsabilidades apenas para classes do domínio conceitual pode levar a situações de maior acoplamento e menos coesão.*
- *Solução*
  - *Atribuir um conjunto altamente coesivo de responsabilidades a uma classe artificial que não representa um conceito do domínio do problema.*

GRASP: Protected Variations

- *Problema*
  - *Como projetar objetos, subsistema e sistemas para que as variações ou instabilidades nesses elementos não tenha um impacto indesejável nos outros elementos?*
- *Solução*
  - *Identificar pontos de variação ou instabilidade potenciais.*
  - *Atribuir responsabilidades para criar uma interface estável em volta desses pontos.*
  - *Encapsulamento, interfaces, polimorfismo, indireção e padrões; máquinas virtuais e brokers são motivados por este princípio*
  - *Evite enviar mensagens a objetos muito distantes.*

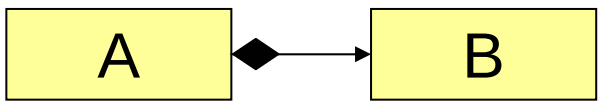
GRASP: Indirection

- *Problema*
  - *Onde atribuir uma responsabilidade para evitar acoplamento direto entre duas ou mais coisas? Como desacoplar objetos para que seja possível suportar baixo acoplamento e manter elevado o potencial de reuso?*
- *Solução*
  - *Atribua a responsabilidade a um objeto intermediário para mediar as mensagens entre outros componentes ou serviços para que não sejam diretamente acoplados.*
  - *O objeto intermediário cria uma camada de indireção entre os dois componentes que não mais dependem um do outro: agora ambos dependem da indireção.*
- *Veja uma aplicação em: Dependency Injection*

Injeção de dependências

(inversão de controle)

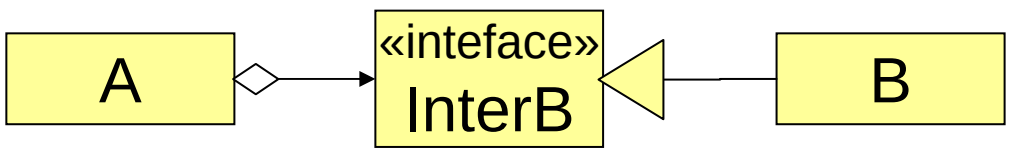
- *Um dos benefícios de usar interfaces*
- *Em vez de*



A depende de B
A precisa achar ou criar B

```
class A {
    B b = new B();
}
```

- *Use*

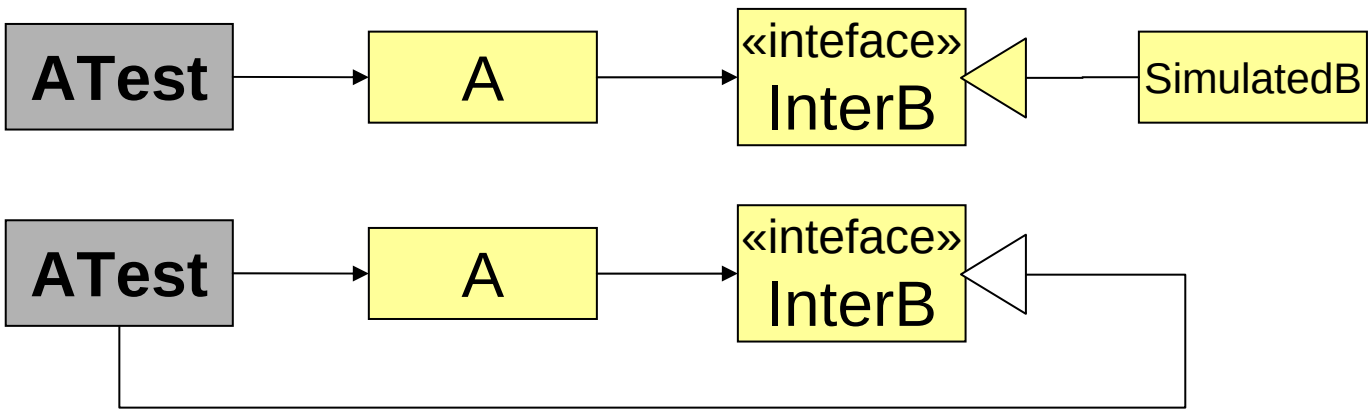


A depende de uma interface IB
B depende de uma interface IB
A recebe implementação de B quando é criado

```
class A {
    InterB b;
    A(InterB b) {
        this.b = b;
    }
}
```

Injeção de dependências (2)

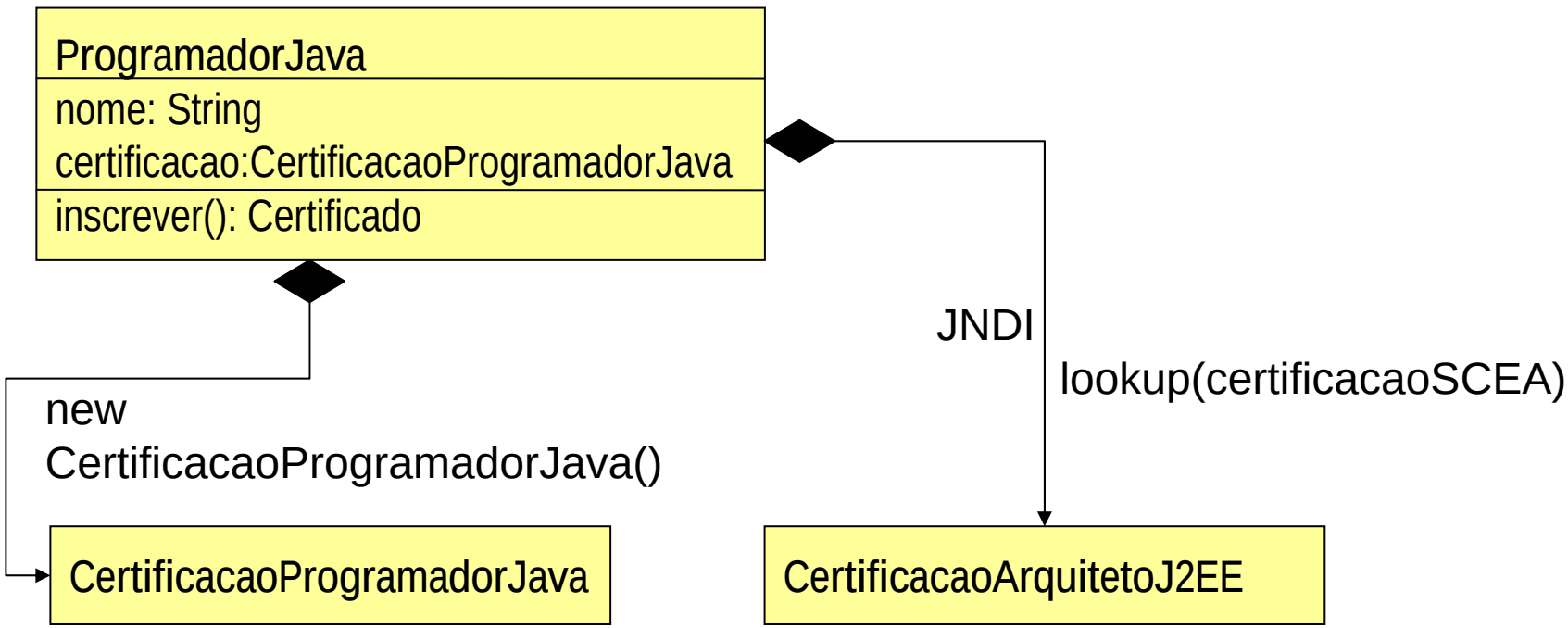
- *Se A depende de InterB, fica fácil testar ou medir A usando uma implementação/simulação de B para testes*



- *Interfaces devem ser pequenas*
  - *Um objeto pode implementar várias interfaces*
  - *Facilita evolução (interfaces grandes publicam um contrato grande que não pode ser revogado)*
- *Planeje interfaces em todos os objetos*
  - *Principalmente objetos que oferecem serviços, singletons, etc.*

Dependency Injection

- *Problema*
  - *Controle convencional: o próprio objeto cria ou localiza suas dependências: acoplamento!*



Exemplo: controle convencional

- Linha 9: dependência fortemente acoplada
  - Difícil de testar objeto sem testar dependência

```
 1:package faculdade;
 2:
 3:public class ProgramadorJava {
 4:    private String nome;
 5:    private CertificacaoProgramadorJava certificacao;
 6:
 7:    public ProgramadorJava(String nome) {
 8:        this.nome = nome;
 9:        certificacao = new CertificacaoProgramadorJava();
10:    }
11:
12:    public Certificado inscrever() throws NaoAprovadoException {
13:        return certificacao.iniciarTeste();
14:    }
15:}
```

A dependência

```
 1:package faculdade;
 2:
 3:public class CertificacaoProgramadorJava {
 4:    public CertificacaoProgramadorJava() {}
 5:    public Certificado iniciarTeste()
                           throws NaoAprovadoException {
 6:        Certificado certificado = null;
 7:        // Realizar questoes
 8:
 9:        return certificado;
10:    }
11:}
```

Diminuindo o acoplamento

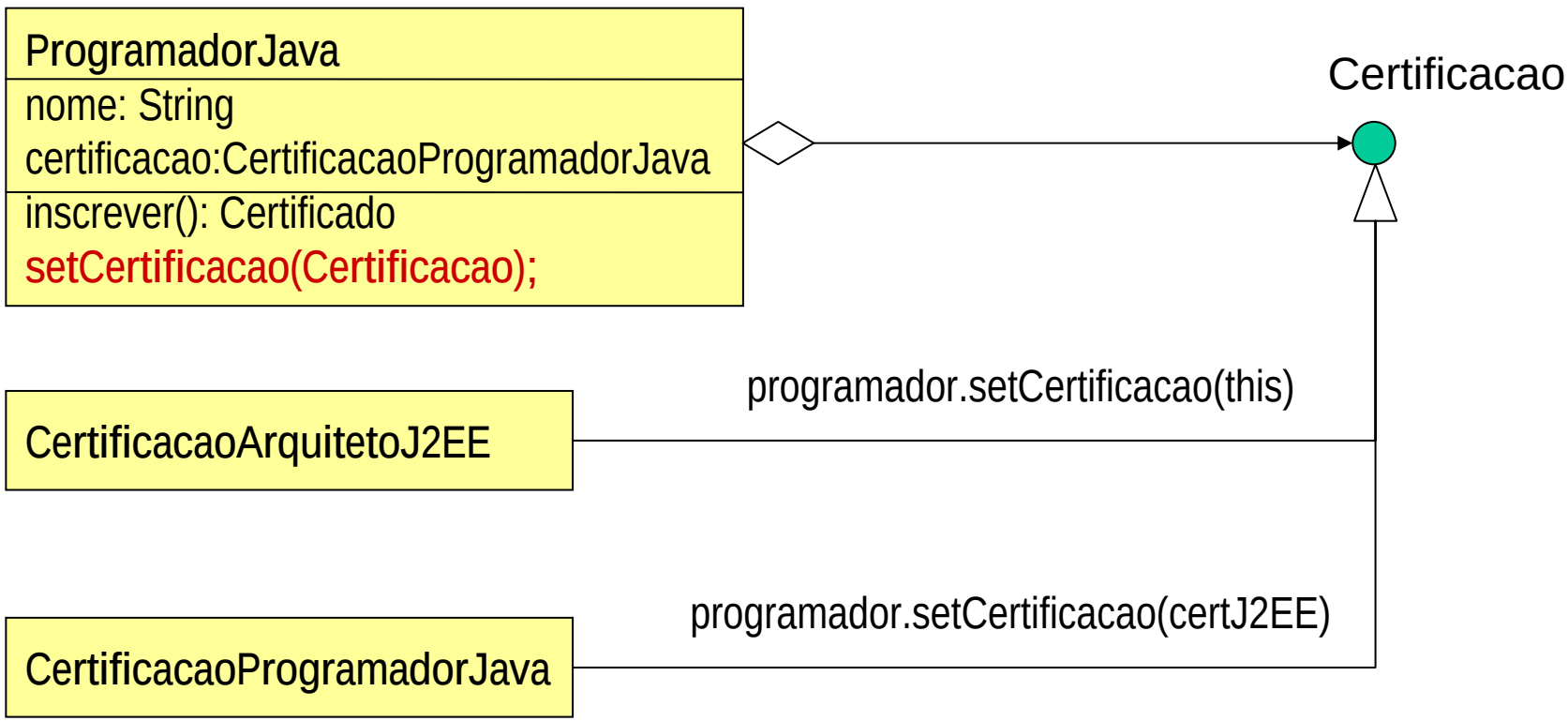
- *Melhor forma de eliminar acoplamento é usar interfaces*
  - *Implementação é dependente da interface*
  - *Cliente depende da interface, e não mais da implementação*

```
3:public interface Certificacao {
4:    public Object iniciarTeste() throws CertificacaoException;
5:}
```

```
3:public class CertificacaoProgramadorJava implements Certificacao {
4:    public CertificacaoProgramadorJava() {}
5:    public Object iniciarTeste() throws NaoAprovadoException {
6:        Certificado certificado = new Certificado();
7:        // Realizar questoes
8:        certificado.setNota(90);
9:
10:        return certificado;
11:    }
12:}
```

Solução

- *Em vez do programador buscar ou criar sua própria certificação, ela é atribuída a ele*
  - *Para isto, deve haver método para criar a associação*



Inversão de controle

```
public interface Programador {
    public Object inscrever() throws NaoAprovadoException;
}
```

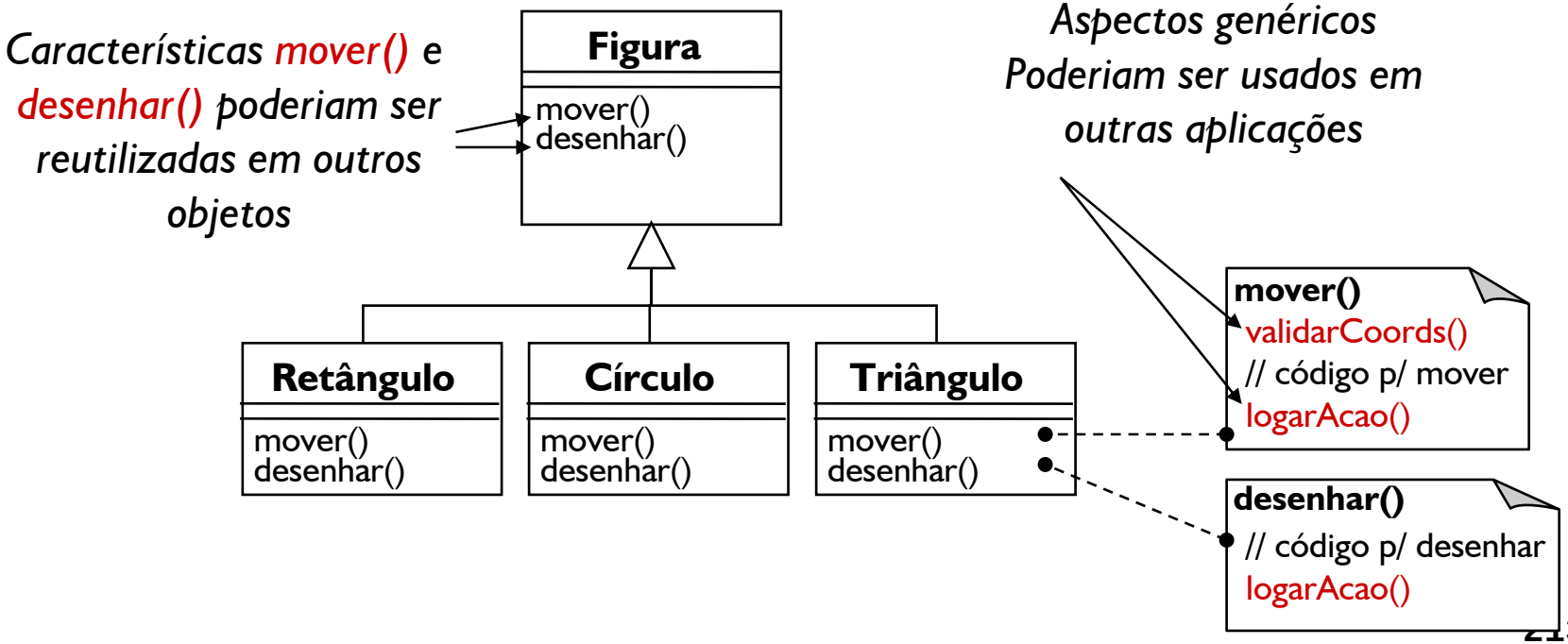
```
3:public class ProgramadorJava implements Programador {
4:    private String nome;
5:    private CertificacaoProgramadorJava certificacao;
6:
7:    public ProgramadorJava(String nome) {
8:        this.nome = nome;
9:    }
10:
11:    public Object inscrever() throws NaoAprovadoException {
12:        return certificacao.iniciarTeste();
13:    }
14:
15:    public void setCertificacao(Certificacao certificacao) {
16:        this.certificacao = certificacao;
17:    }
18:}
```

Aspectos: Separação de interesses

- A separação de interesses é objetivo essencial do processo de decomposição da solução de um problema
  - Decomposição deve continuar até que cada unidade da solução possa ser compreendida e construída
  - Cada unidade deve lidar com apenas um interesse
- Separação de interesses eficiente promove código de melhor qualidade
  - Maior modularidade
  - Facilita atribuição de responsabilidades entre módulos
  - Promove o reuso
  - Facilita a evolução do software
  - Viabiliza análise do problema dentro de domínios específicos

Tipos de interesse

- *Usando uma linguagem orientada a objetos, pode-se representar de forma eficiente e modular as classes e procedimentos*
- *Outros interesses são implementados como partes de classes e partes ou composição de procedimentos*



Problema: scattering e tangling

```
public class VeiculoPassageiro
             extends Veiculo {
  public void abastecer() {
    super.abastecer() + gasolina;
    Logger.log(nome() +
      " abastecido com gasolina");
  } ...
}
```

Scattering: *acrescentar* ***um*** *requerimento causa espalhamento do código em várias classes*

```
public class VeiculoCarga
             extends Veiculo {
  public void abastecer() {
    super.abastecer() + diesel;
    Logger.log(nome() +
      " abastecido com diesel");
  } ...
}
```

```
public class VeiculoAereo
             extends Veiculo {
  public void abastecer() {
    // código
    Logger.log(nome() +
      " abastecido com " +
      x + " litros");
    // código
  }  ...
}
```

Tangling: *código para implementar o requerimento se mistura com lógica do código existente*

Inclusão de nova funcionalidade

- *Também sujeita a scattering e tangling*
  - *Funcionalidade espalhada por várias classes (funcionalidade consiste de métodos contidos em várias classes)*
  - *Funcionalidade misturada com outros recursos (não é representado por uma entidade separada)*

```
public class VeiculoPassageiro
             extends Veiculo {
  public void abastecer() {...}
  public void carregar() {...}
  public Apolice segurar() {
    Apolice a =
            new Apolice(...);
    // preenche apolice basica
    return a;
  }
  ...
}
```

```
public class VeiculoCarga
             extends Veiculo {
  public void abastecer() {...}
  public Apolice segurar() {
    Apolice a = new Apolice();
    // preenche apolice p/
    // segurar carro e carga
    return a;
  }
  public void carregar() {...}
  ...
}
```

Como reduzir tangling e scattering?

1. Criar subclasse que implemente recursos, sobreponha métodos, etc.

   Problema: classes cliente têm que ser alteradas para que saibam como criar a nova subclasse:

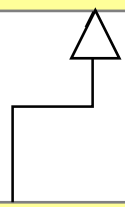
```
VeiculoCarga v =
      // new VeiculoCarga();
      new VeiculoCarga2();
```

2. Design patterns como factory method resolveriam o problema

```
VeiculoCarga v =
    VeiculoCarga.create();
```

   mas sua interface precisaria ser planejada antes!

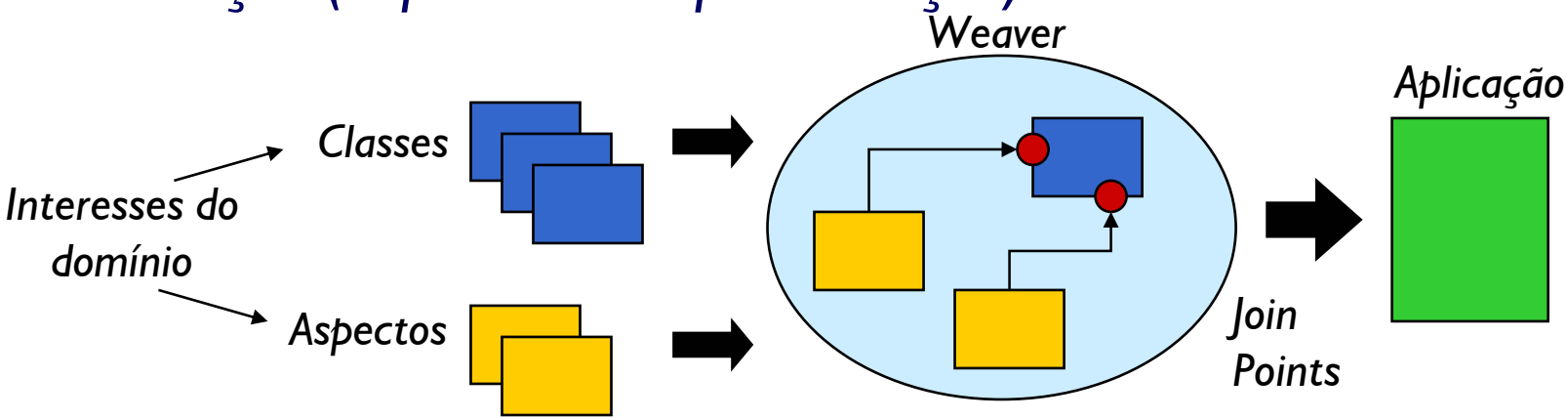
```
package veiculos;
public class VeiculoCarga
        extends Veiculo {
  public void abastecer() {
    super.abastecer()
                + diesel;
  }
}
```



```
package veiculos.versao2;
public class VeiculoCarga2
  extends veiculos.VeiculoCarga {
  public void abastecer() {
    super.abastecer() + diesel;
    Logger.log(nome() +
    " abastecido com diesel");
  }
  public Apolice segurar() {...}
}
```

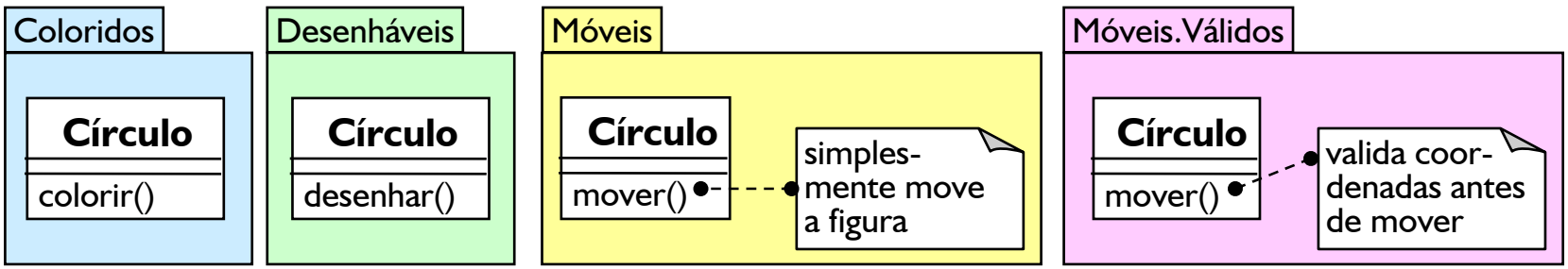
Solução: Aspect-Oriented Programming

- AOP divide interesses em dois grupos
  - Componentes: quando interesse é encapsulável em unidades OO (classe, objeto, método, procedimento).
  - Aspectos: interesses que não são representados como componentes mas representar propriedades que interferem no seu funcionamento ou estrutura.
- Aspectos são interesses ortogonais
  - Costurados ao código principal durante compilação ou execução (depende da implementação)

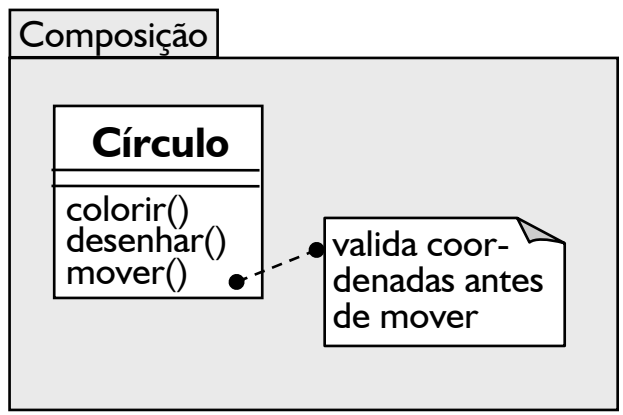


Solução: Subject-Oriented Programming

- *Sujeitos (subjects) são abstrações diferentes de um mesmo conceito dentro de um domínio específico*

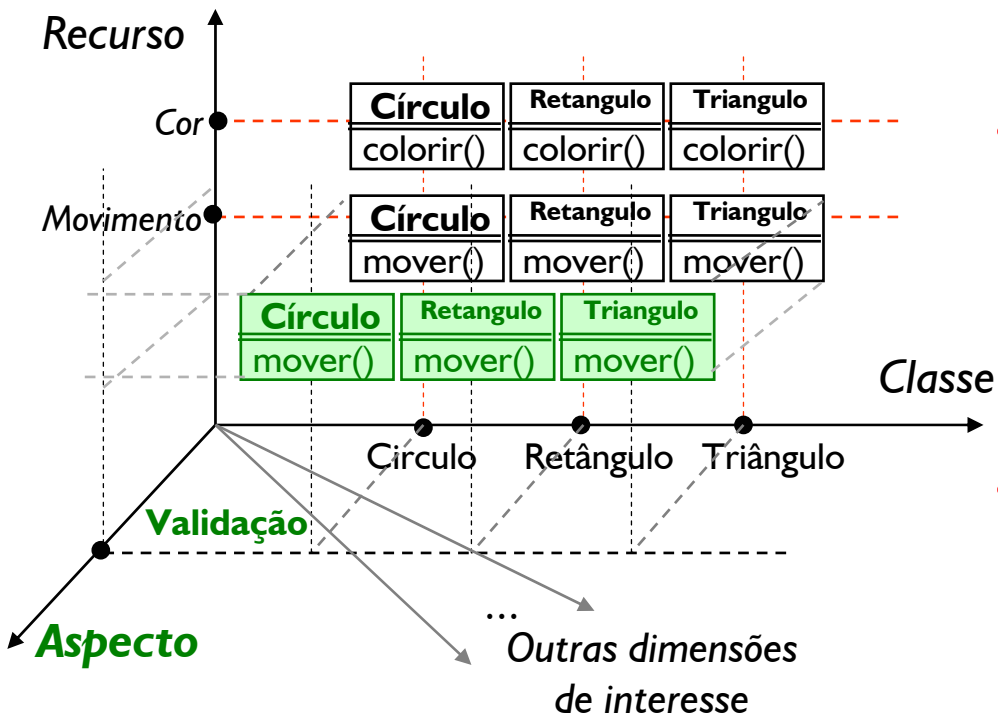


- *Para implementar a aplicação, sujeitos são compostos gerando um objeto*
  - *Regras definem forma como os sujeitos serão compostos*
- *Pode-se trabalhar com conceitos simples (domain-specific)*



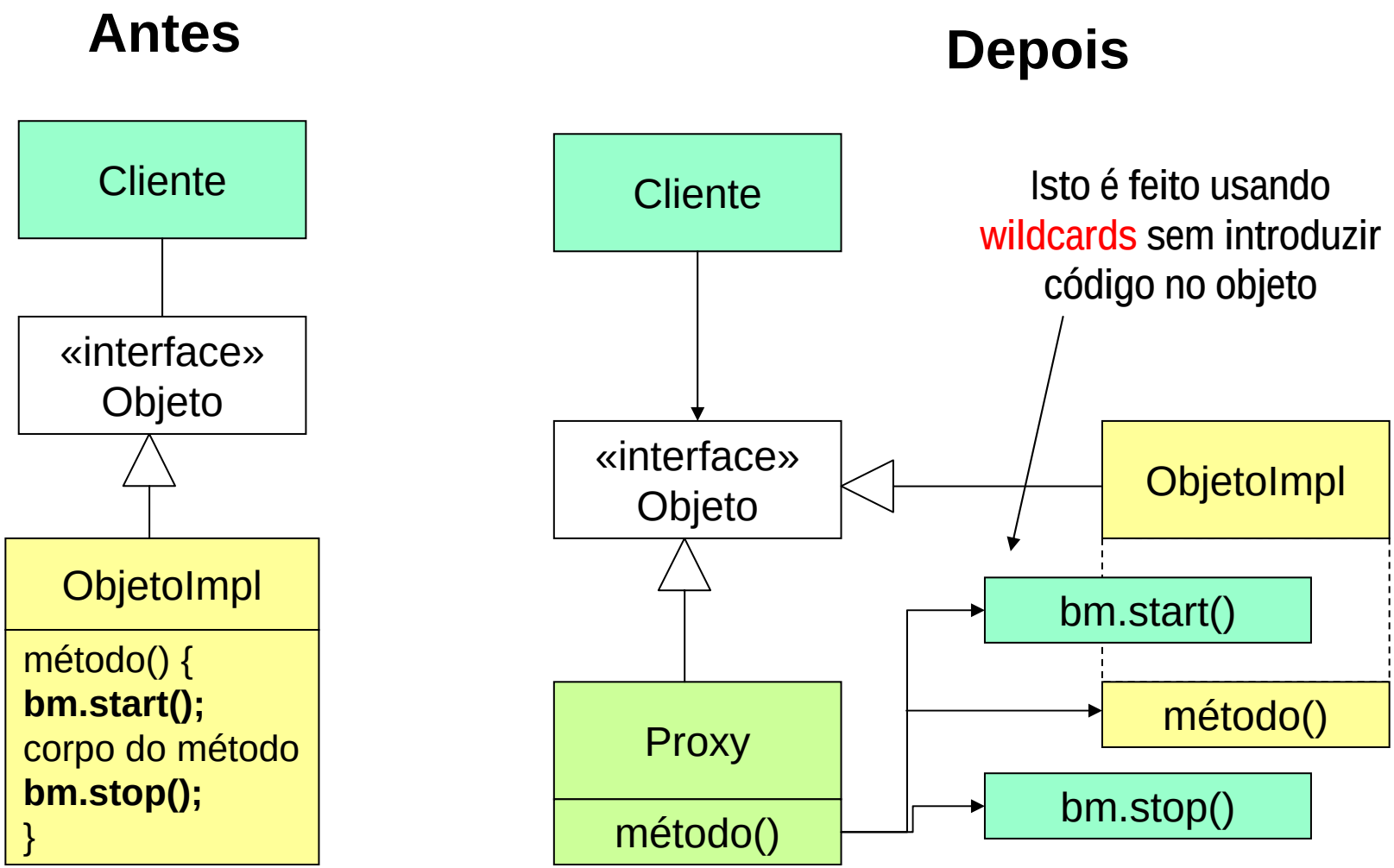
Solução: MDSoC: HyperSpaces

- *Representação de unidades de software em múltiplas dimensões de interesse*
  - *Eixos representam dimensões de interesse*
  - *Pontos nos eixos representam interesses*
  - *Unidades de software são representadas no espaço*



- *Hyperslices*
  - *Encapsulam interesses*
- *Hypermodule*
  - *Conjunto de hyperslices e relacionamentos de integração (informam como hyperslices se relacionam entre si)*
- *Neste modelo, aspecto é apenas mais uma dimensão de interesse*

Uso de aspectos



Conclusões

- *Neste minicurso, introduzimos diversos padrões usados em aplicações OO*
  - *Padrões clássicos GoF, que descrevem soluções para problemas comuns, elaborados*
  - *Padrões GRASP, que descrevem aplicação de princípios OO*
  - *Padrões e práticas emergentes como injeção de dependências (aplicação de uma prática GRASP) e aspectos (extensão do OO)*
- *Aprenda a usar os padrões clássicos e encurte o tempo para ganhar tornar-se um programador experiente!*
- *Vários outros padrões existem e serão inventados*
  - *Alguns sobreviverão por muito tempo, outros não*
  - *Catalogue suas soluções e crie seus próprios padrões!*

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*

[6][JMM FAQ] *Jeremy Manson and Brian Goetz, JSR 133 (Java Memory Model) FAQ, Feb 2004*

Curso J930: Design Patterns

Versão 2.1

www.argonavis.com.br

(helder.darocha@gmail.com)