



# *Programação Orientada a Objetos II*

## **Padrões de Projeto - b**

Prof. Dr. Fábio Fagundes Silveira

`fsilveira@unifesp.br`

`http://fabiosilveira.net`

UNIFESP – Universidade Federal de São Paulo



# Créditos



- Grande parte destes slides foram baseados:
  - no curso de Padrões de Projeto, preparado e ministrado por Helder Rocha, da empresa Argonavis; e
  - no livro: Design Patterns: Elements of Reusable Object-oriented Software - Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

# 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 (diferente de nada): stubs.*

# Além das interfaces

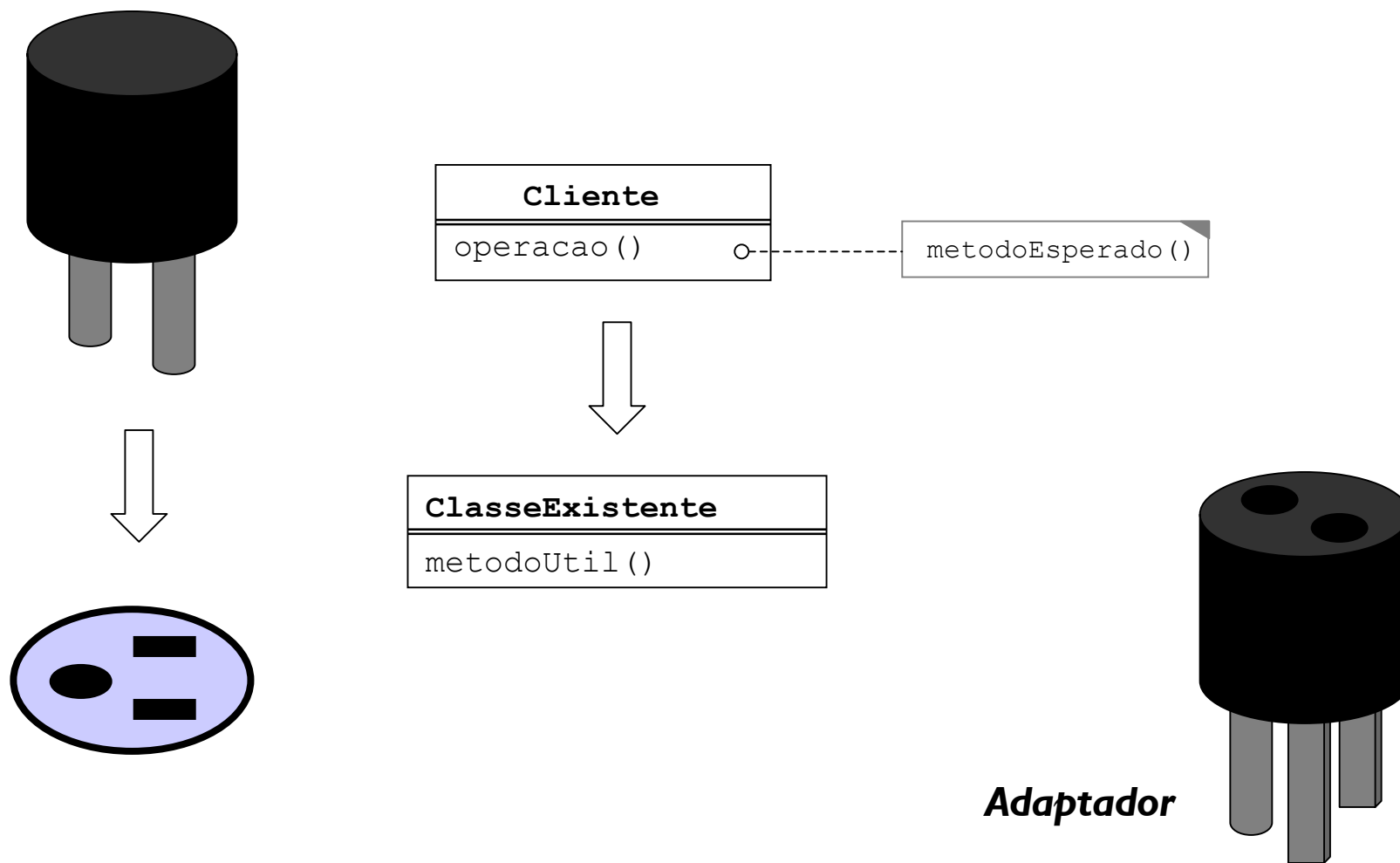
- *Design patterns oferecem aplicações específicas de interfaces com regras definidas*
  - *Adapter*: para adaptar a interface de uma classe para outra que o cliente espera
  - *Façade*: oferecer uma interface simples para uma coleção de classes
  - *Composite*: definir uma interface comum para objetos individuais e composições de objetos
  - *Bridge*: desacoplar uma abstração de sua implementação para que ambos possam variar independentemente

# 1

## Adapter

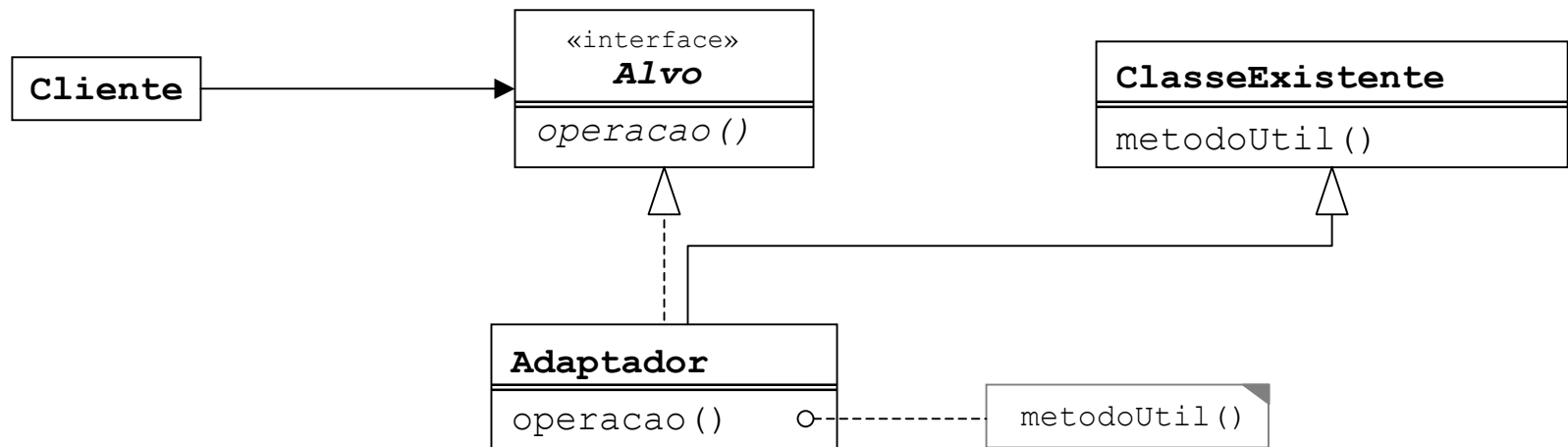
*"Objetivo: converter a interface de uma classe em outra interface esperada pelos clientes. Adapter permite a comunicação entre classes que não poderiam trabalhar juntas devido à incompatibilidade de suas interfaces." [GoF]*

# Problema



# Duas formas de Adapter

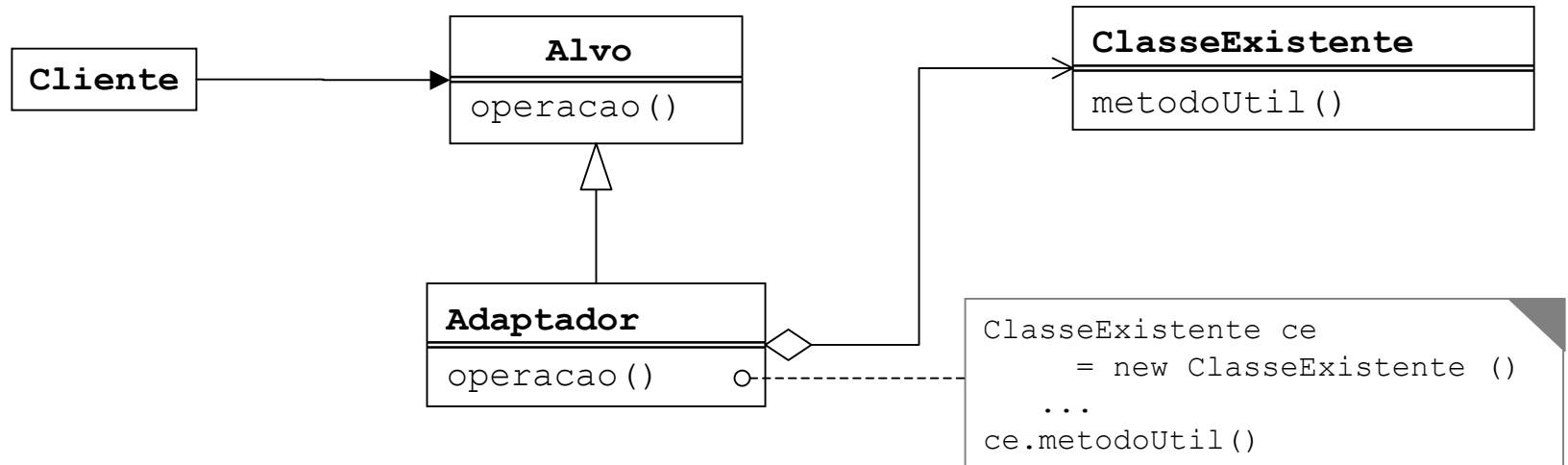
- **Class Adapter**: usa herança múltipla



- **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();
    }
}
```

# Exemplos de Adapter

- *JSDK API: Tratamento de eventos (java.awt.event)*
  - *MouseAdapter, WindowAdapter, etc. são stubs para implementação de adapters*
- *JSDK API: Wrappers de tipos em Java*
  - *Double, Integer, Character, etc. Adaptam tipos primitivos à interface de java.lang.Object.*
- *Uso de JTable, JTree, JList (javax.swing)*
  - *A interface TableModel e as classes AbstractTableModel e DefaultTableModel oferecem uma interface para o acesso aos campos de uma JTable*
  - *Um adapter é útil para traduzir operações específicas do domínio dos dados (planilha, banco de dados, etc.) às operações da tabela.*

## Quando usar?

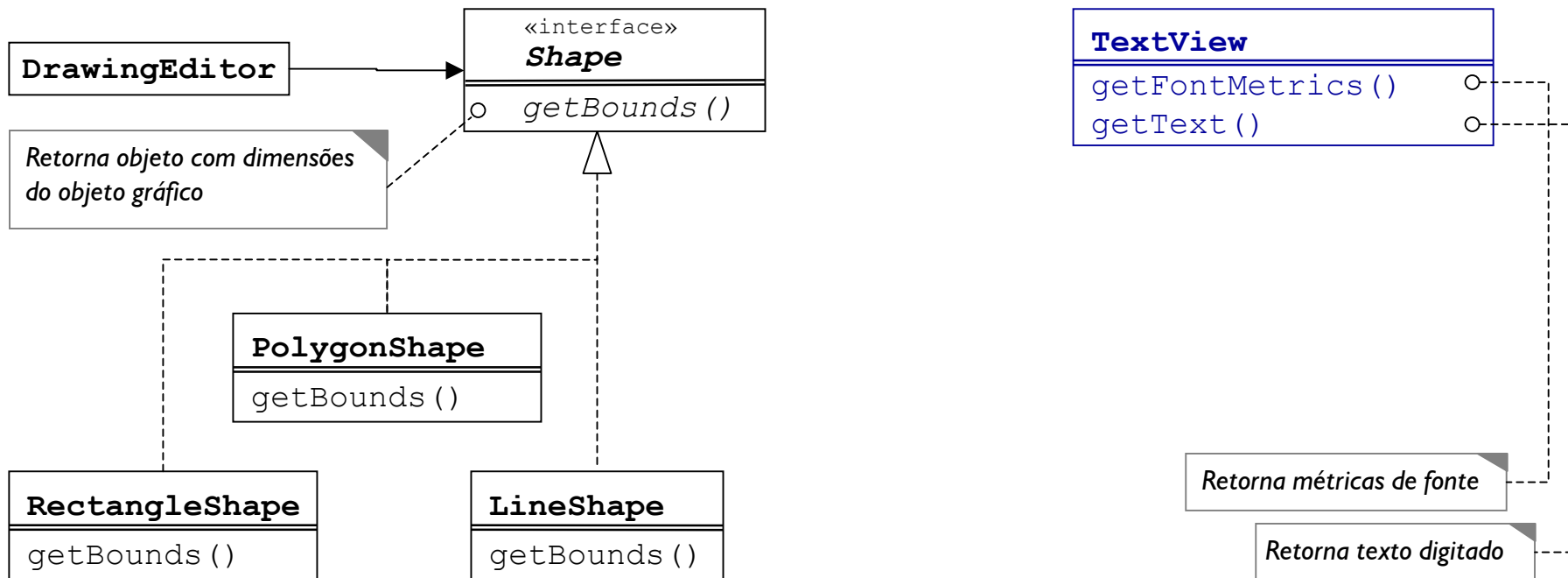
- *Sempre que for necessário adaptar uma interface para um cliente*
- *Class Adapter*
  - *Quando houver uma interface que permita a implementação estática*
- *Object Adapter*
  - *Quando menor acoplamento for desejado*
  - *Quando o cliente não usa uma interface Java ou classe abstrata que possa ser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

# *Padrões semelhantes ou relacionados*

- *Bridge*
  - *Possui estrutura similar mas tem outra finalidade: separar uma interface de sua implementação para que possam ser alteradas independentemente*
  - *Adapter serve para alterar a interface de um objeto existente*
- *Decorator*
  - *Acrescenta funcionalidade a um objeto sem alterar sua interface (mais transparente)*
- *Proxy*
  - *Representa outro objeto sem mudar sua interface*

# Exercícios

1.1 Complete o diagrama de classes abaixo para que um objeto **TextView** possa participar da aplicação **DrawingEditor**



1.2 Justifique sua escolha pelo tipo de Adapter usado (de classe ou de objeto)

# Exercícios

*1.3 Escreva uma classe que permita que o cliente VectorDraw, que já usa a classe Shape, use as operações de RasterBox (e Coords) para obter os mesmos dados*

```
public class VectorDraw {  
    ...  
  
    Shape s;  
    // Obtém instancia de Shape  
    int x = s.getX();  
    int height = s.getHeight();  
  
    ....  
  
}
```

```
public class Shape {  
    protected int x, y, height, width;  
  
    public int getX() { return x; }  
    public int getY() { return y; }  
    public int getHeight() { return height; }  
    public int getWidth() { return width; }  
  
}
```

```
public class RasterBox {  
    private Coords topLeft, bottomRight;  
    public Coords getTopLeft() {  
        return topLeft;  
    }  
    public Coords getBottomRight() {  
        return bottomRight;  
    }  
  
}
```

```
public class Coords {  
    public int x, y;  
  
}
```

*1.4 Challenge 3.2  
(livro-texto)*

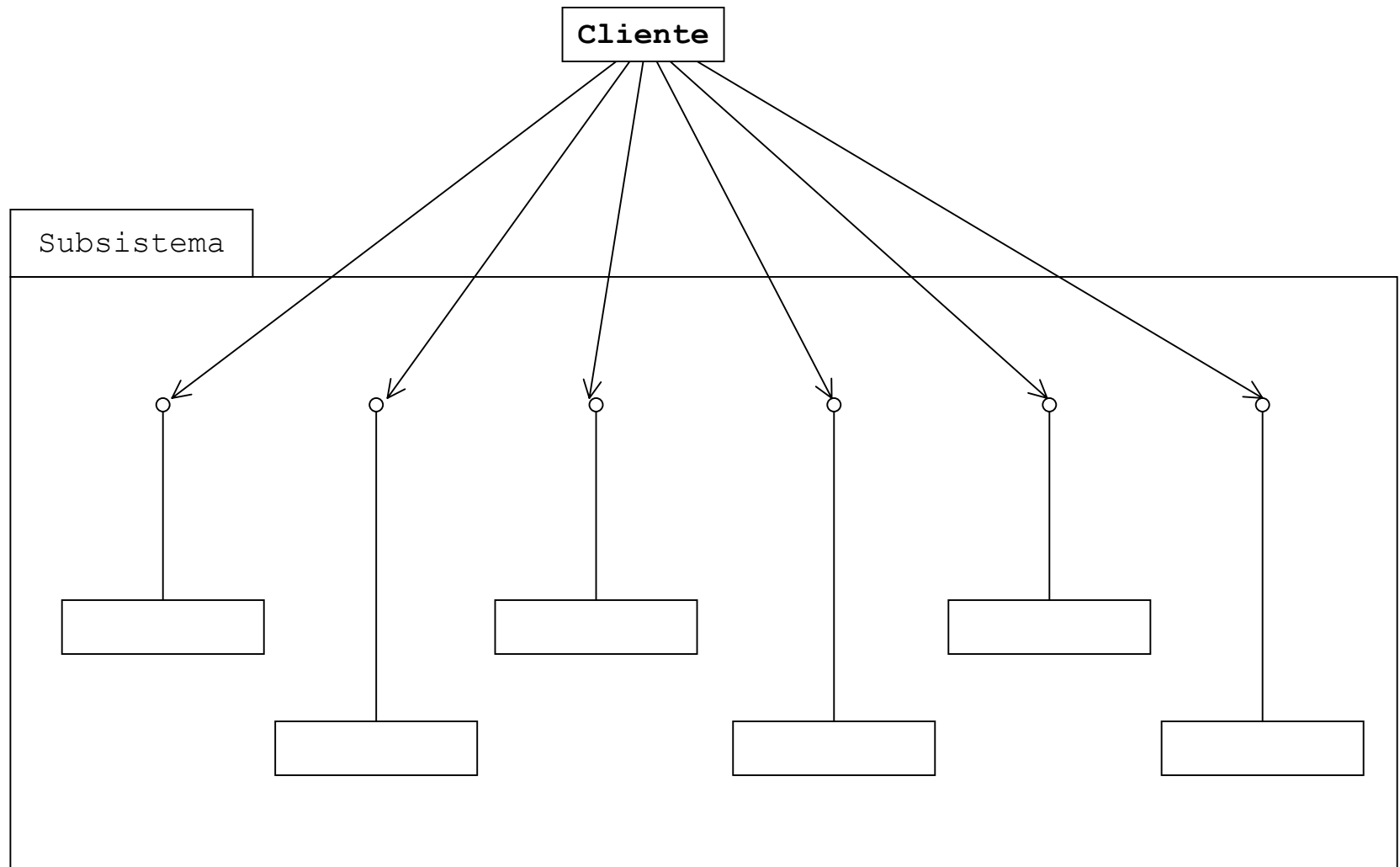
# 2

## Façade

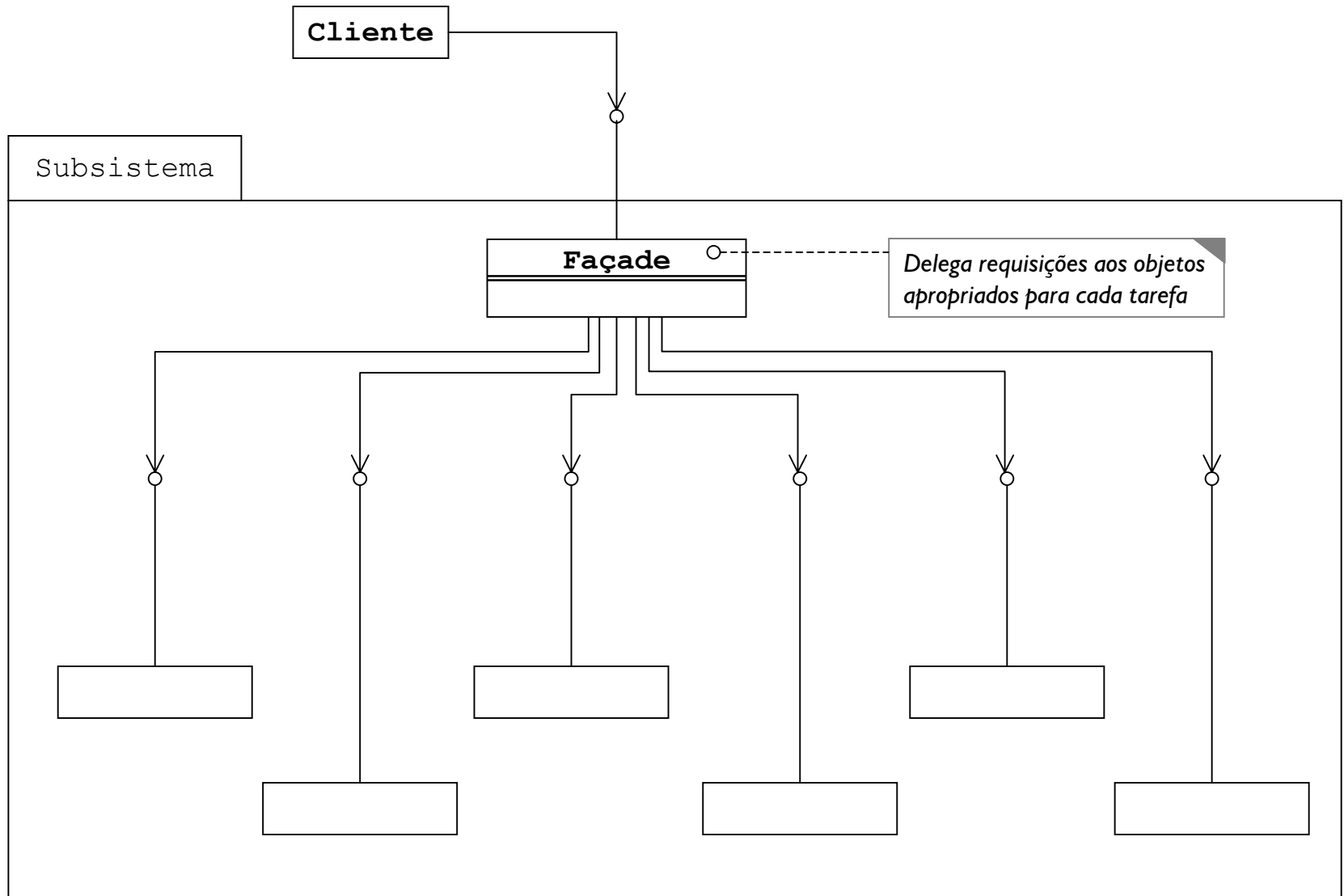
*"Oferecer uma interface única para um conjunto de interfaces de um subsistema. Façade define uma interface de nível mais elevado que torna o subsistema mais fácil de usar." [GoF]*



# Problema



# Estrutura de Façade



# Faça 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(clienteID);  
        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);  
    }  
}
```

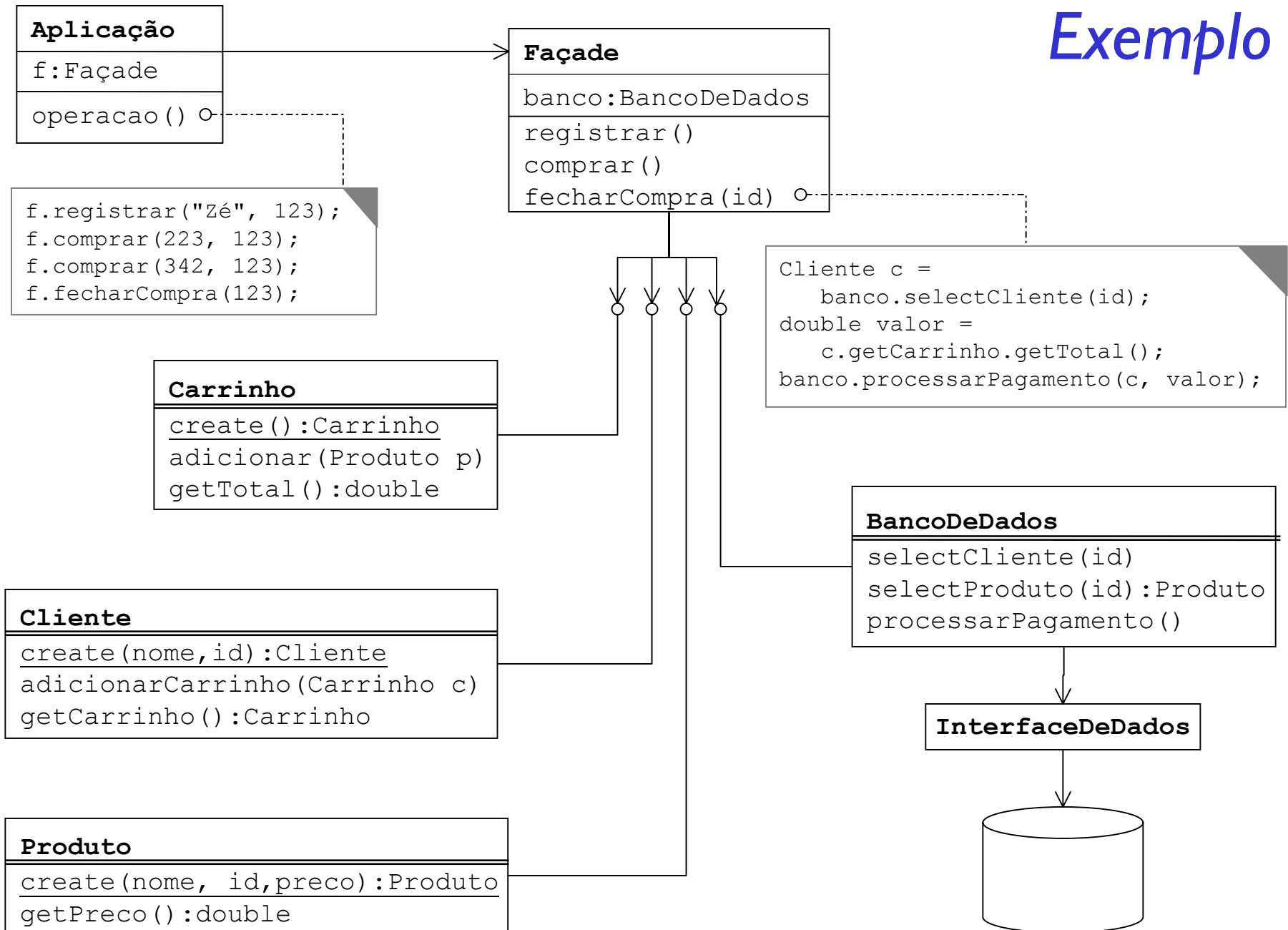
```
public class Carrinho {  
    static Carrinho create() {...}  
    void adicionar(Produto p) {...}  
    double getTotal() {...}  
}
```

```
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() {...}  
}
```

# Exemplo



## Quando usar?

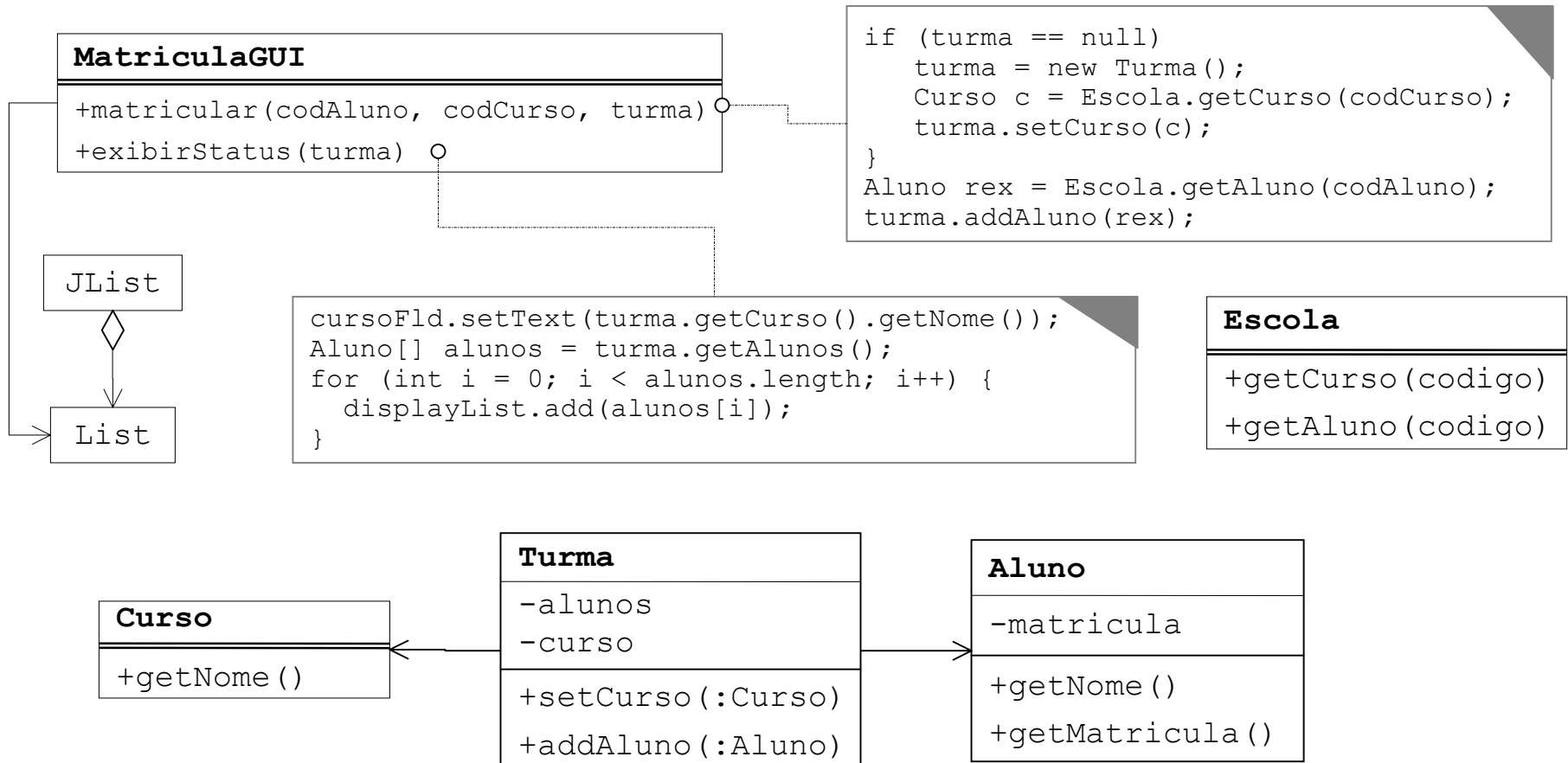
- *Sempre que for desejável criar uma interface para um conjunto de objetos com o objetivo de facilitar o uso da aplicação*
  - *Permite que objetos individuais cuidem de uma única tarefa, deixando que a fachada se encarregue de divulgar as suas operações*
- *Façades e Singletons*
  - *Fachadas freqüentemente são implementadas como singletons*

# Façades, Utilities e Demos

- *Uma fachada que possui apenas métodos estáticos é chamada de **Utility** [3]*
- *Uma **Demo** [1] (demonstração) é um exemplo que mostra como usar uma classe ou subsistema*
  - *Demos tem o mesmo valor que fachadas*
- *Demos x fachadas*
  - *Demo: geralmente uma aplicação standalone, não-reutilizável que mostra uma forma de usar o subsistema*
  - *Fachada: classe configurável e reutilizável com uma interface de alto nível que torna um subsistema mais fácil de usar.*

# Exercício

2.1 A aplicação abaixo tem uma GUI. Será necessário implementar uma UI orientada a caracter e uma interface Web. O que poderia ser usado para reduzir a duplicação de código e tornar a utilização das classes envolvidas mais simples? Implemente!



## Exercícios (2)

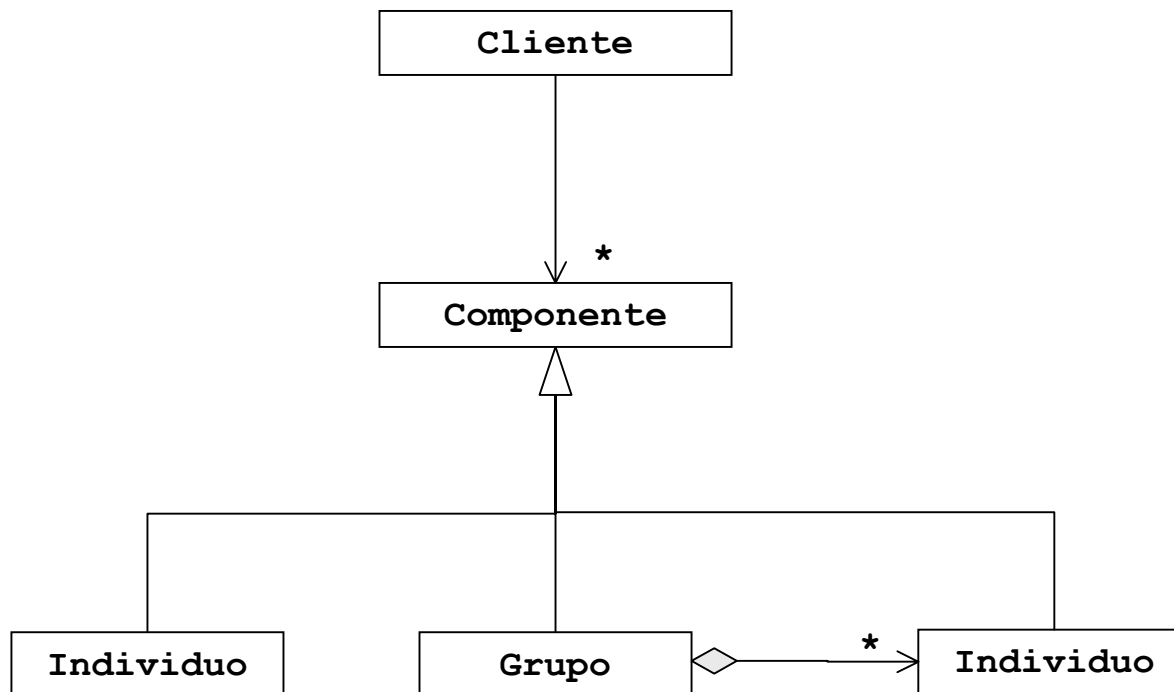
- 2.2 *Implemente e teste o exemplo mostrado de Façade em Java (simule o banco com Strings)*
- 2.3 *Qual a diferença entre Façade e Adapter (se você tiver um Façade para um único objeto?)*



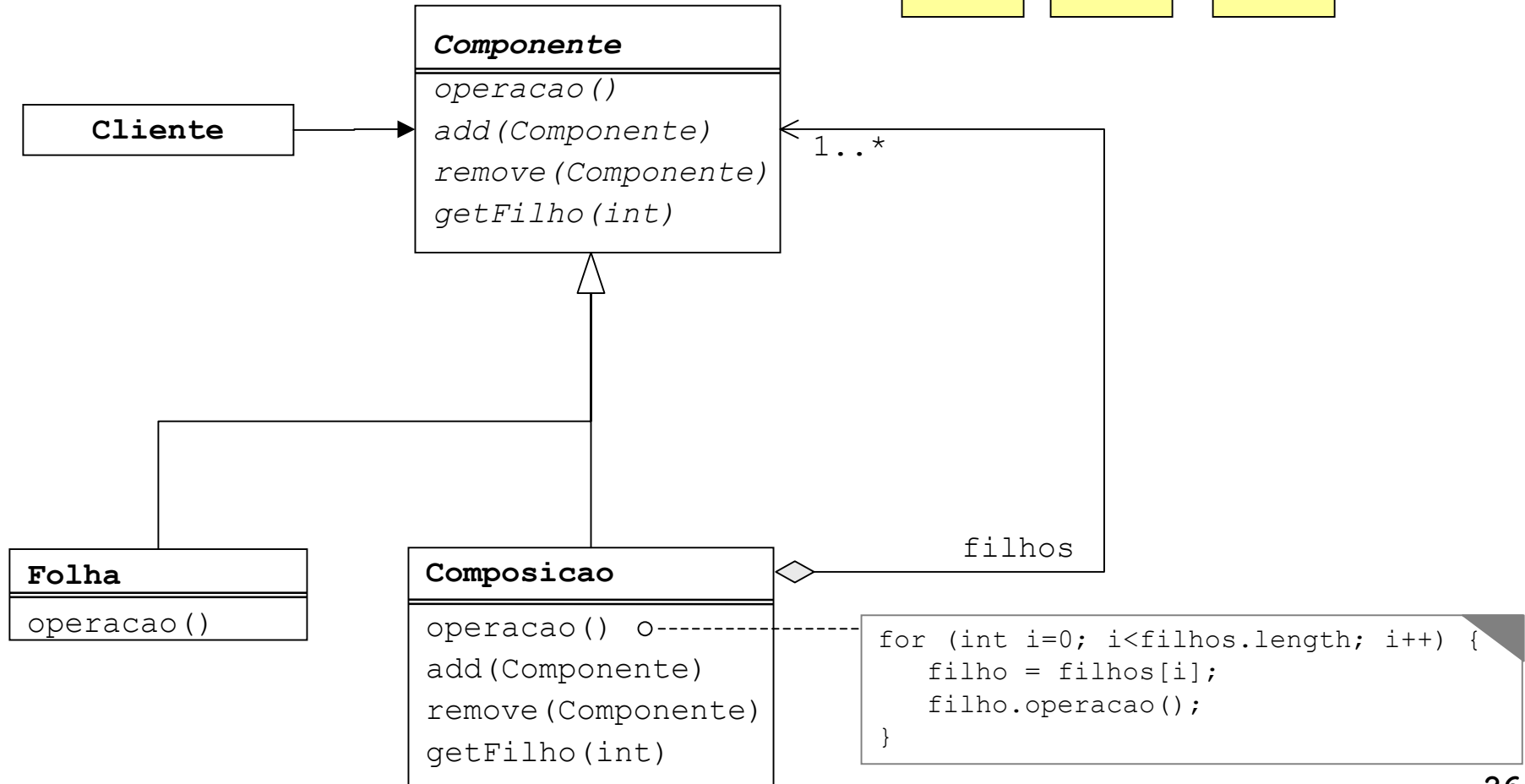
# Composite

*"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



# 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;
    }
}
```

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

## 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*

# Questões importantes de implementação

- *Referências explícitas ao elemento pai*
  - *getParent()*
- *Compartilhamento*
  - *Risco de guardar pais múltiplos (ciclos)*
- *Operações add() e remove() nos filhos*
  - *getComponent() para saber se é folha ou composite*
- *Quem deve remover componentes?*
  - *Composite destroi seus filhos quando é destruído*

# Exercícios

3.1. Que padrão (ões) você usaria para resolver o problema abaixo?

- O congresso inscreve participantes, que podem ser um indivíduo ou instituição. Cada indivíduo tem um assento

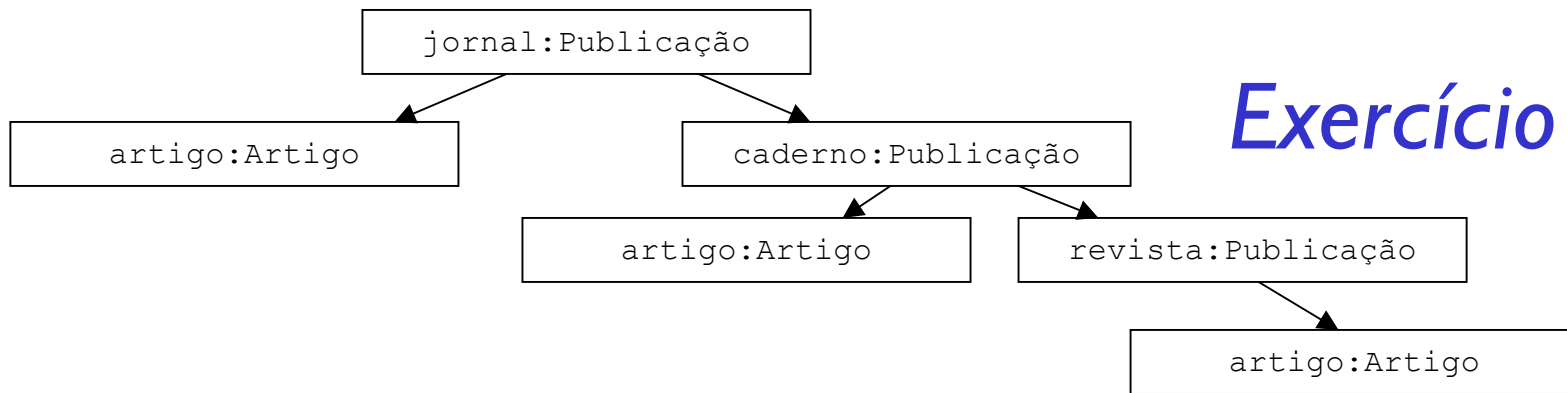
Congresso
totalParticipantes()
totalAssentos()

Indivíduo
getAssento()

Instituição
getMembros()

- Mostre como implementar uma solução

## Exercício 3.2



- Escreva uma interface `Publicação` que trate de forma equivalente coleções (compostas de outras publicações, como revistas, jornais, cadernos) e artigos individuais indivisíveis.
- Escreva uma aplicação de testes que construa o diagrama de objetos acima e
  - Imprima o número de publicações e de artigos
  - Imprima o conteúdo de `toString()` que deve imprimir o `toString` de cada publicação (deve conter o nome e, se for artigo, o autor).

<b>Coleção</b>
<code>publicacoes:List</code>
<code>Coleção(nome:String)</code>
<code>getPublicações():List</code>
<code>addPublicação(Satelite)</code>
<code>toString():String</code>

<b>Artigo</b>
<code>Artigo(nome:String, autores:String[])</code>
<code>toString():String</code>



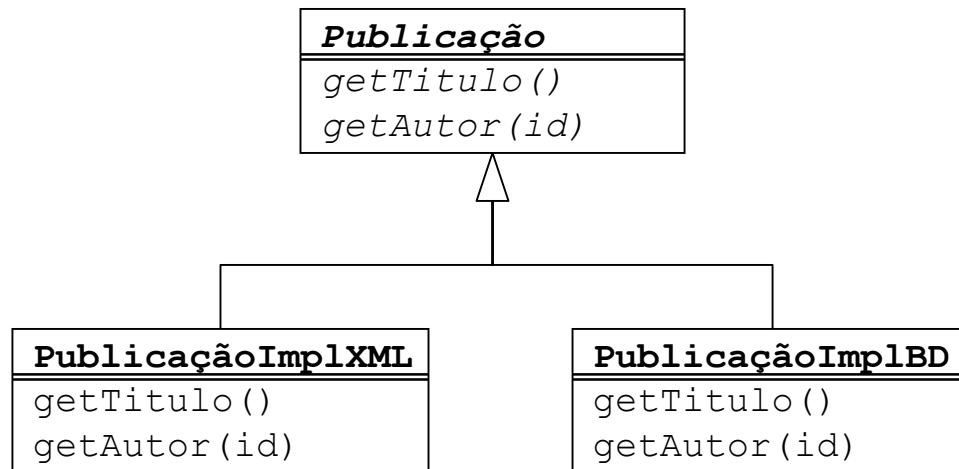
# 4

## Bridge

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

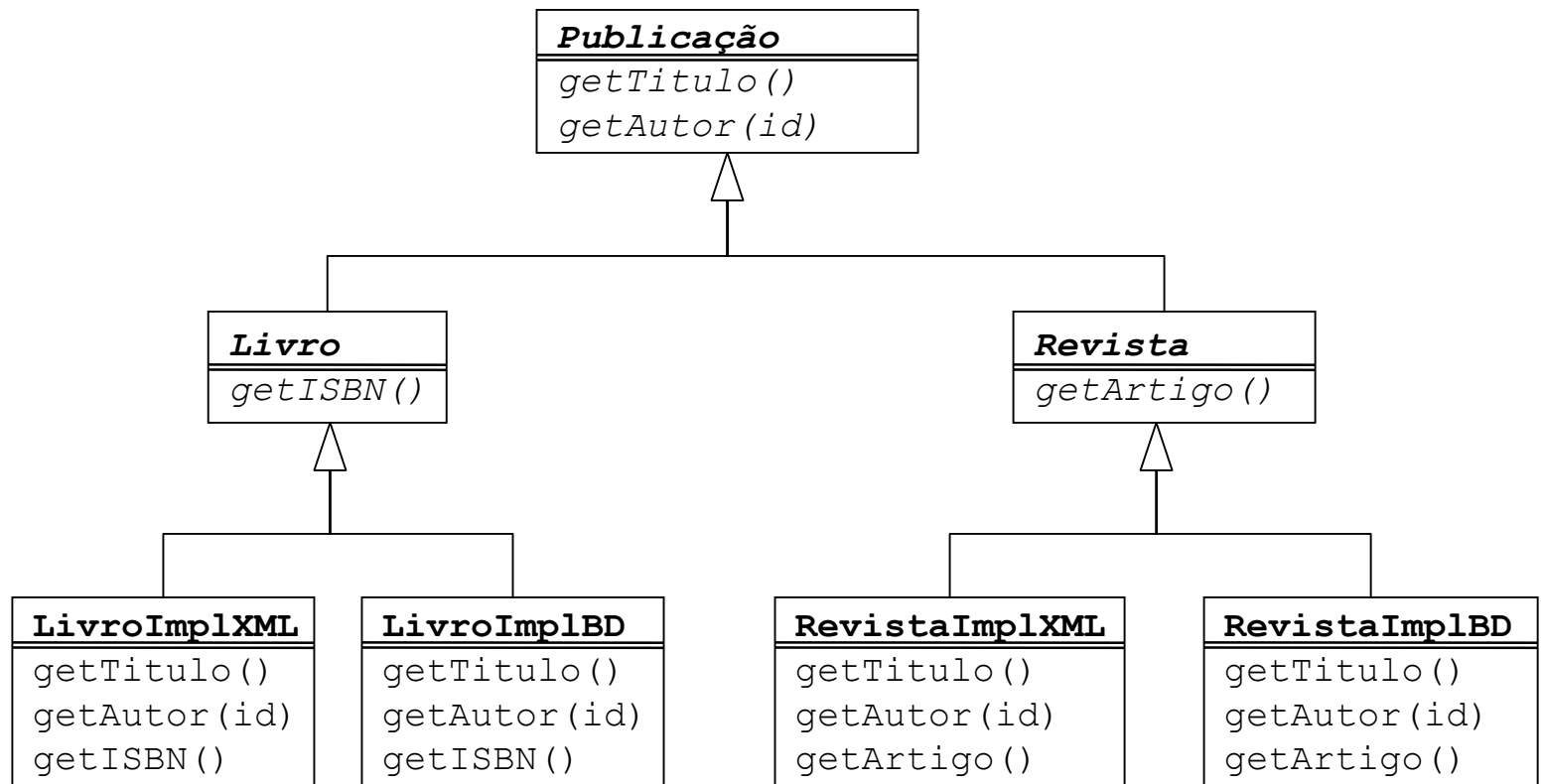
# Problema (I)

- *Necessidade de um driver*

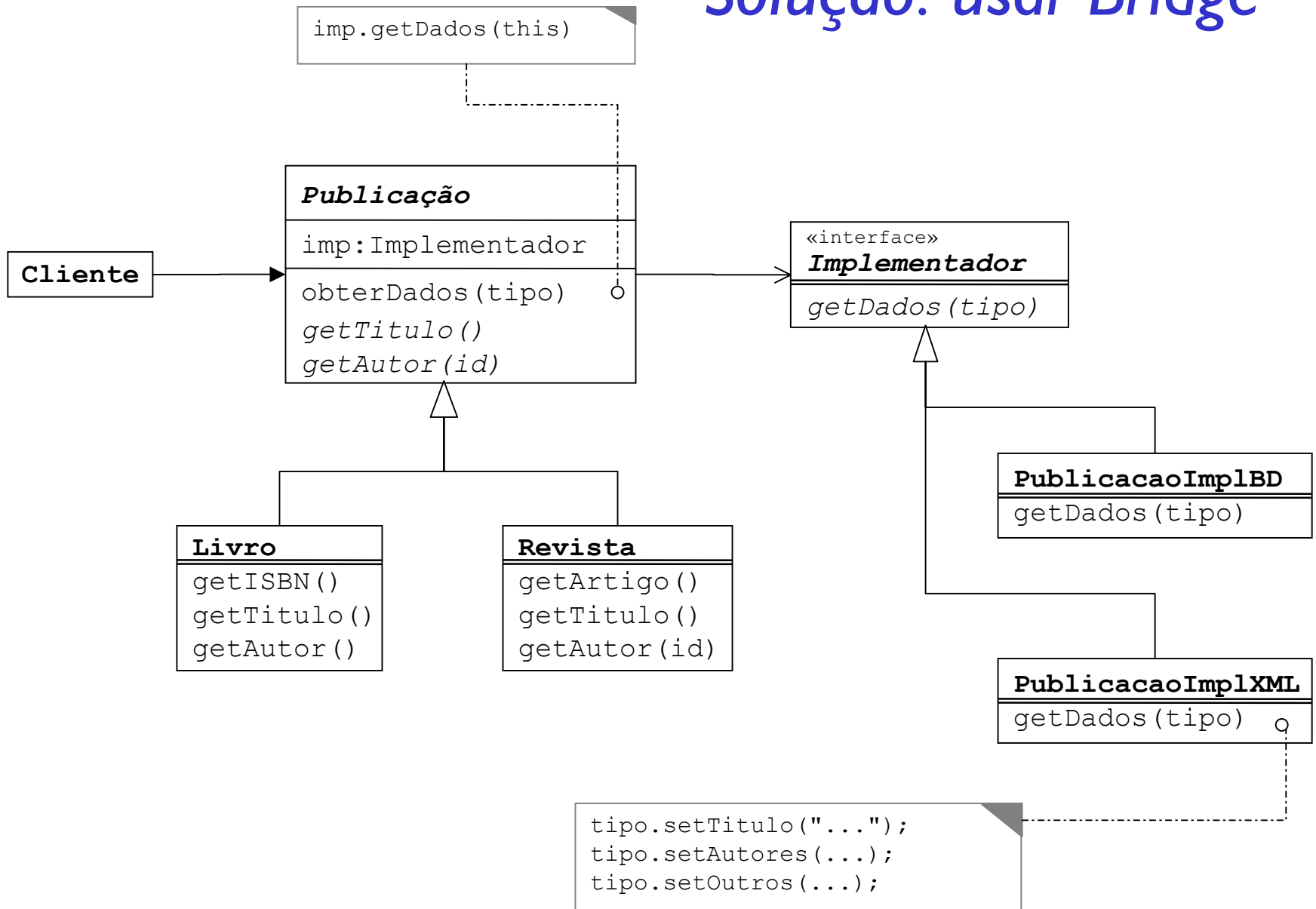


## Problema (II)

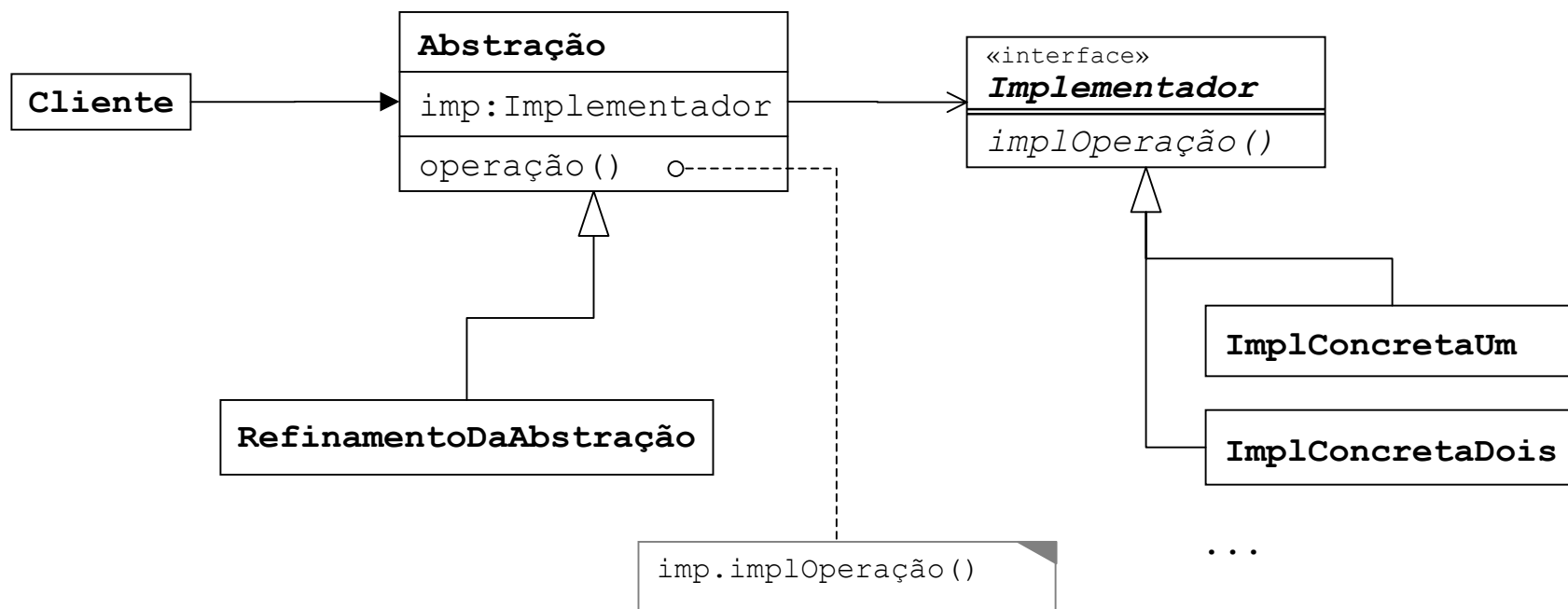
- *Mas herança complica a implementação*



# Solução: usar Bridge



# Estrutura de Bridge



*A Abstração define operações de nível 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*

# Conseqüências de uso de Bridge

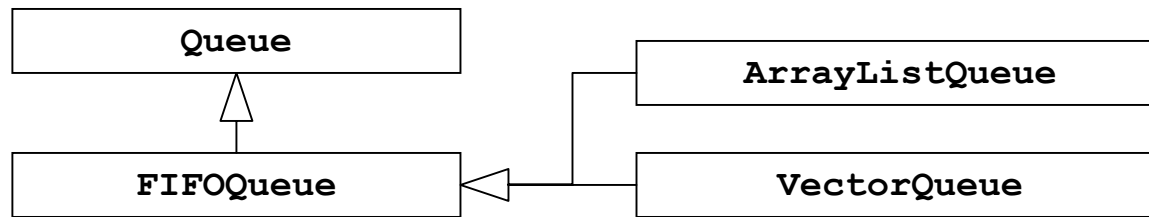
- *Detalhes de implementação totalmente inacessíveis aos clientes*
- *Eliminação de dependências em tempo de compilação das implementações*
- *Implementação da abstração pode ser configurada em tempo de execução*

# *Padrões relacionados*

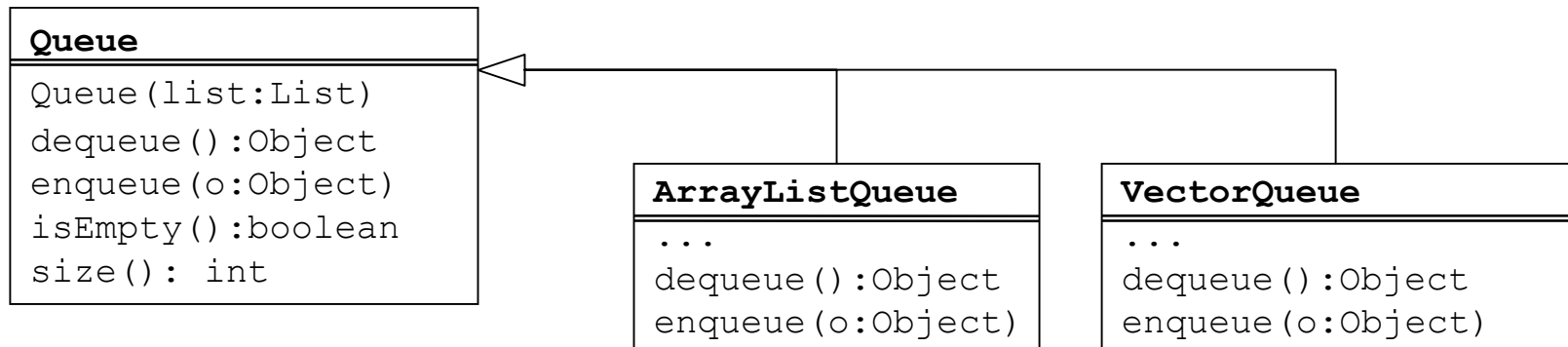
- *Adapter*
  - *Usado para fazer classes não relacionadas trabalharem juntas. Geralmente aplicado a sistemas depois que são projetados*
  - *Bridge é projetada previamente para permitir que interfaces e implementações variem independentemente*
- *Abstract Factory*
  - *Pode ser usado para escolher uma implementação em tempo de execução transparentemente.*



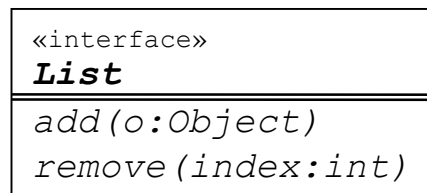
# Exercícios



- 4.1 Considere a hierarquia abaixo. Será necessário criar uma subclasse *FIFOQueue* (acima). O que pode ser feito? (Escolha a melhor alternativa e use, se possível, classes da API Java)



- 4.2 Faça um Bridge usando a classe `java.util.List` como solução de implementação. Se Implemente os métodos `queue()`, `dequeue()`, `size()` e `isEmpty()`.



## 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*

# Fontes

- [1] Steven John Metsker, *Design Patterns Java Workbook*. Addison-Wesley, 2002, Caps. 2 a 6. *Exemplos em Java, diagramas em UML e exercícios sobre Adapter, Façade, Composite e Bridge.*
- [2] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995. Adapter, Facade, Bridge & Composite. *Referência com exemplos em C++ e Smalltalk.*
- [3] James W. Cooper. *The Design Patterns Java Companion*. <http://www.patterndepot.com/put/8/JavaPatterns.htm>