



Programação Orientada a Objetos II

Padrões de Projeto - f

Prof. Dr. Fábio Fagundes Silveira

`fsilveira@unifesp.br`

`http://fabiosilveira.net`

UNIFESP – Universidade Federal de São Paulo



Créditos



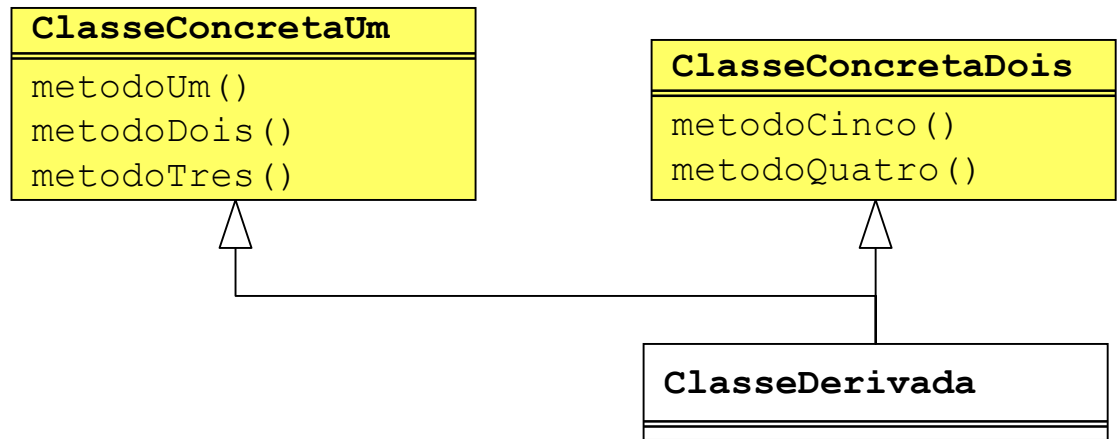
- Grande parte destes slides foram baseados:
 - no curso de Padrões de Projeto, preparado e ministrado por Helder Rocha, da empresa Argonavis; e
 - no livro: Design Patterns: Elements of Reusable Object-oriented Software - Erich Gamma; Richard Helm; Ralph Johnson; John Vlissides

Introdução: Extensão

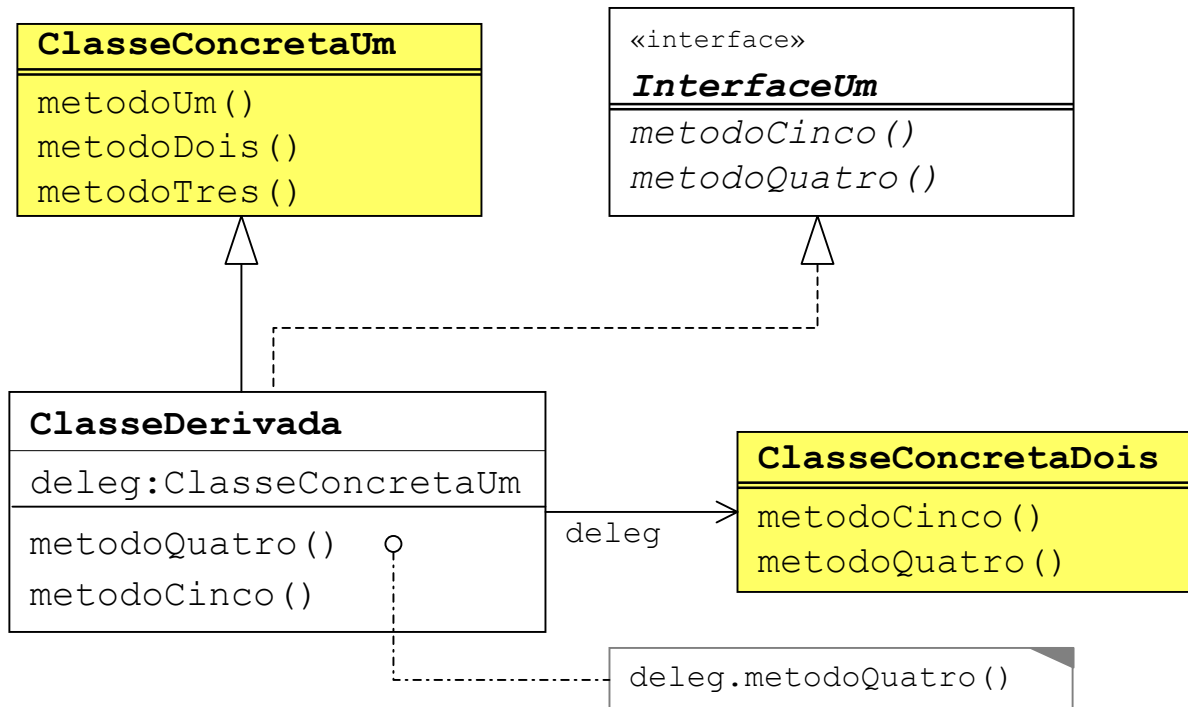
- *Extensão é a adição de uma classe, interface ou método a uma base de código existente [2]*
- *Formas de extensão*
 - *Herança (criação de novas classes)*
 - *Delegação (para herdar de duas classes, pode-se estender uma classe e usar delegação para "herdar" o comportamento da outra classe)*
- *Desenvolvimento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



- Classes existentes
- Classes novas

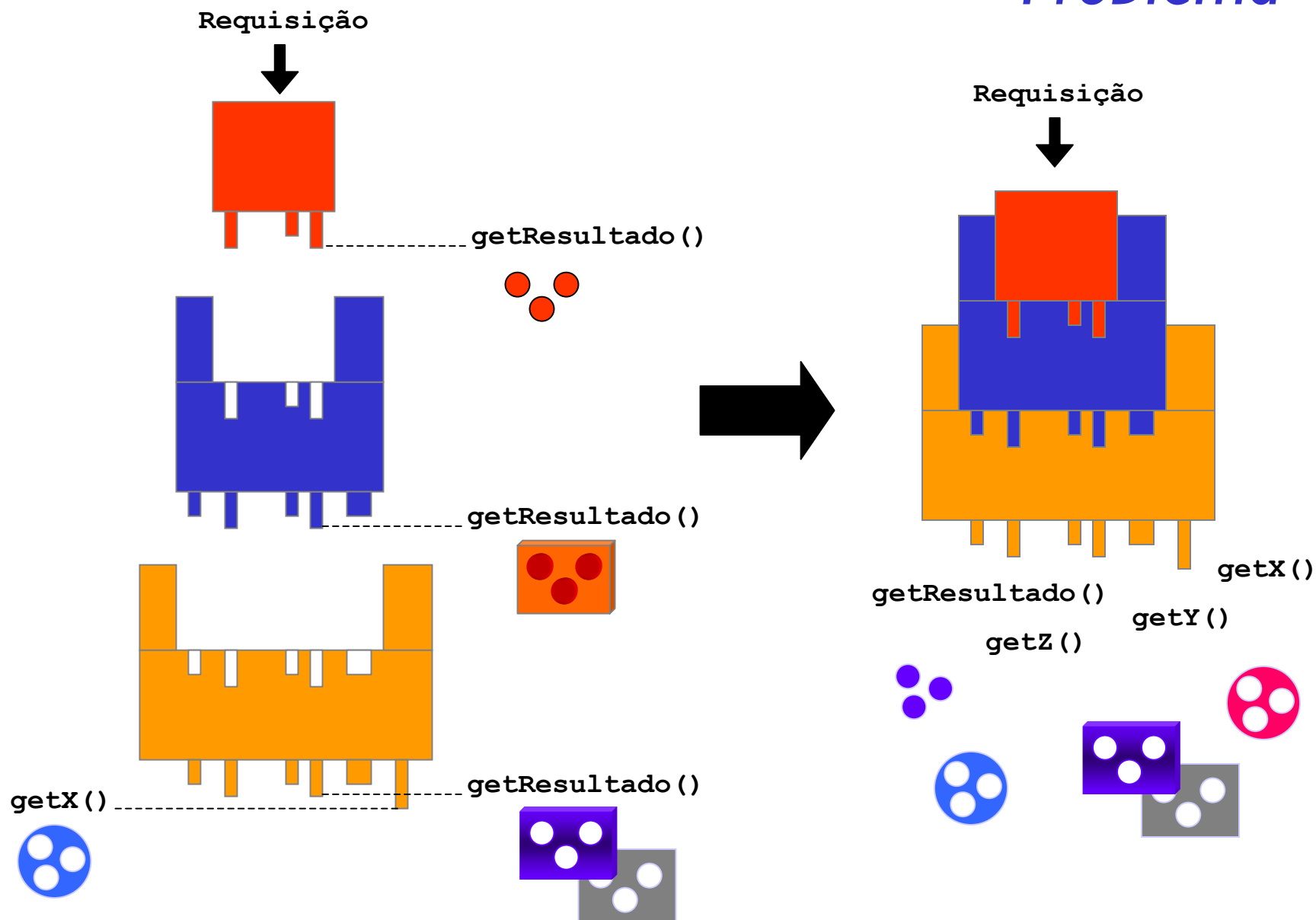
Além da extensão

- 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.

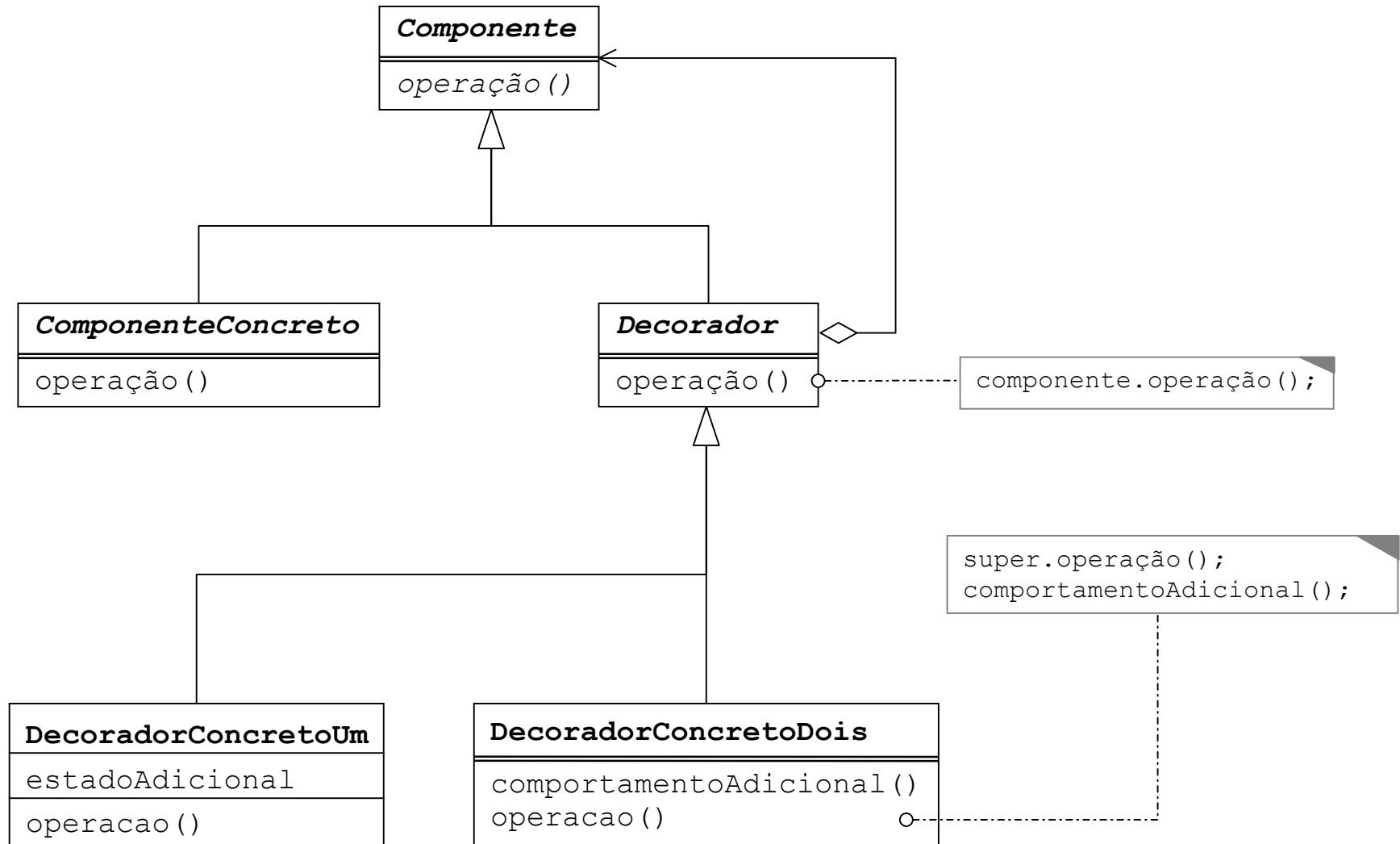
Decorator

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

Problema



Estrutura de Decorator



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) {
        ...
    }
}
```

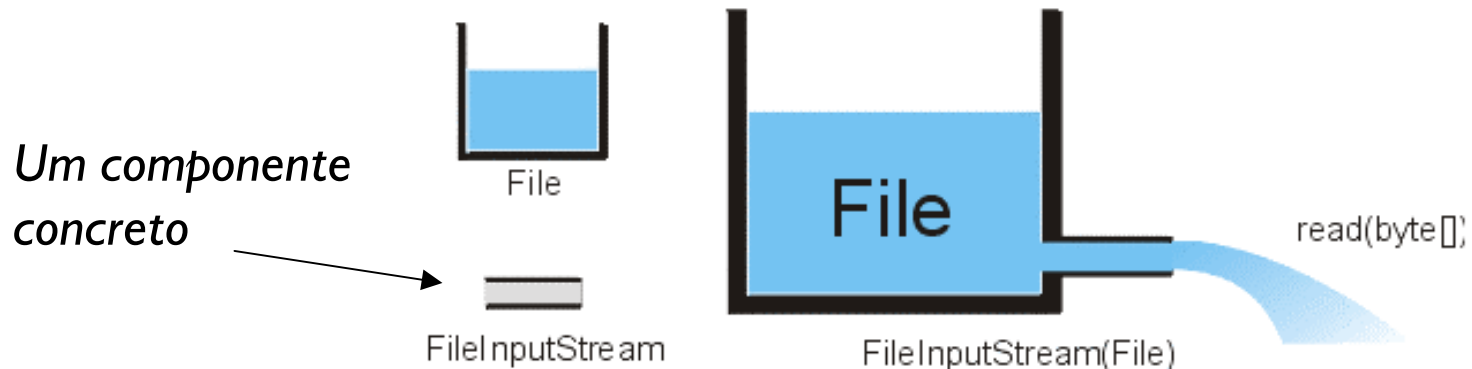
```
public interface Componente {
    Object getDados();
    void operacao(Object arg);
}
```

```
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);
    }
}
```

Decorator no J2SDK

- *Embora na literatura sobre design patterns (GoF) a maior parte das aplicações apresentadas para uso de decoradores seja em aplicações gráficas, em Java o Swing usa outras abordagens*
 - *Ex: ScrollPane "decora" um TextArea, mas as chamadas não são feitas através do ScrollPane*
- *Em Java, o uso mais comum de decoradores é nos objetos que representam fluxos de entrada e saída (I/O streams)*
 - *java.io: InputStream, OutputStream, Reader, Writer, etc.*

I/O Streams

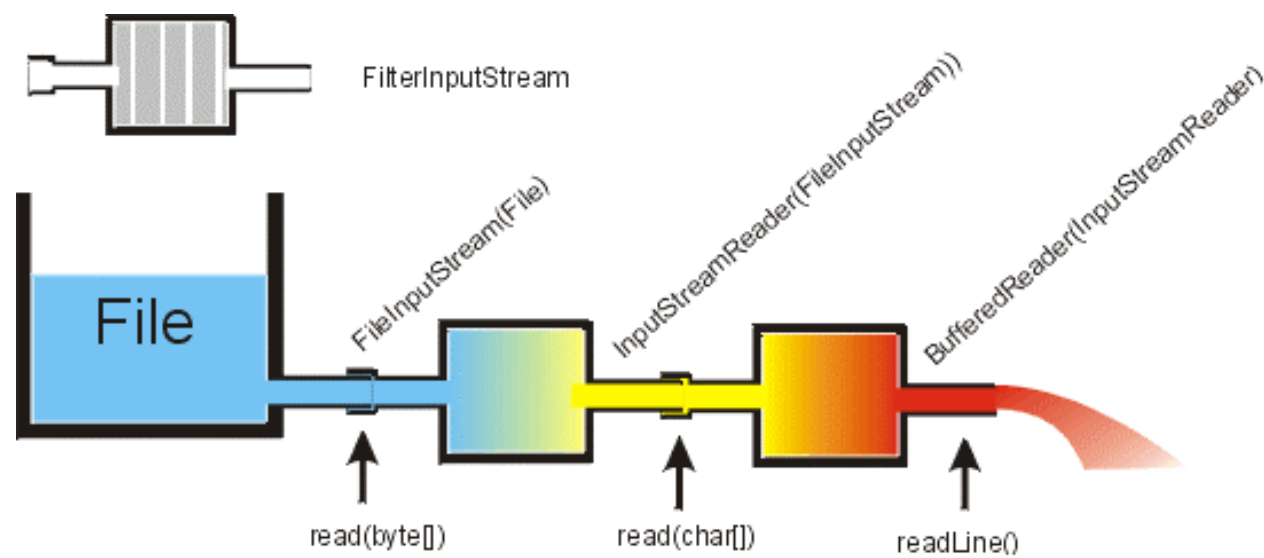


```
// objeto do tipo File
File tanque = new File("agua.txt");

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

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

Concatenação de I/O streams



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

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

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

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

Concatenação do decorador

Uso de método com comportamento alterado

Comportamento adicional

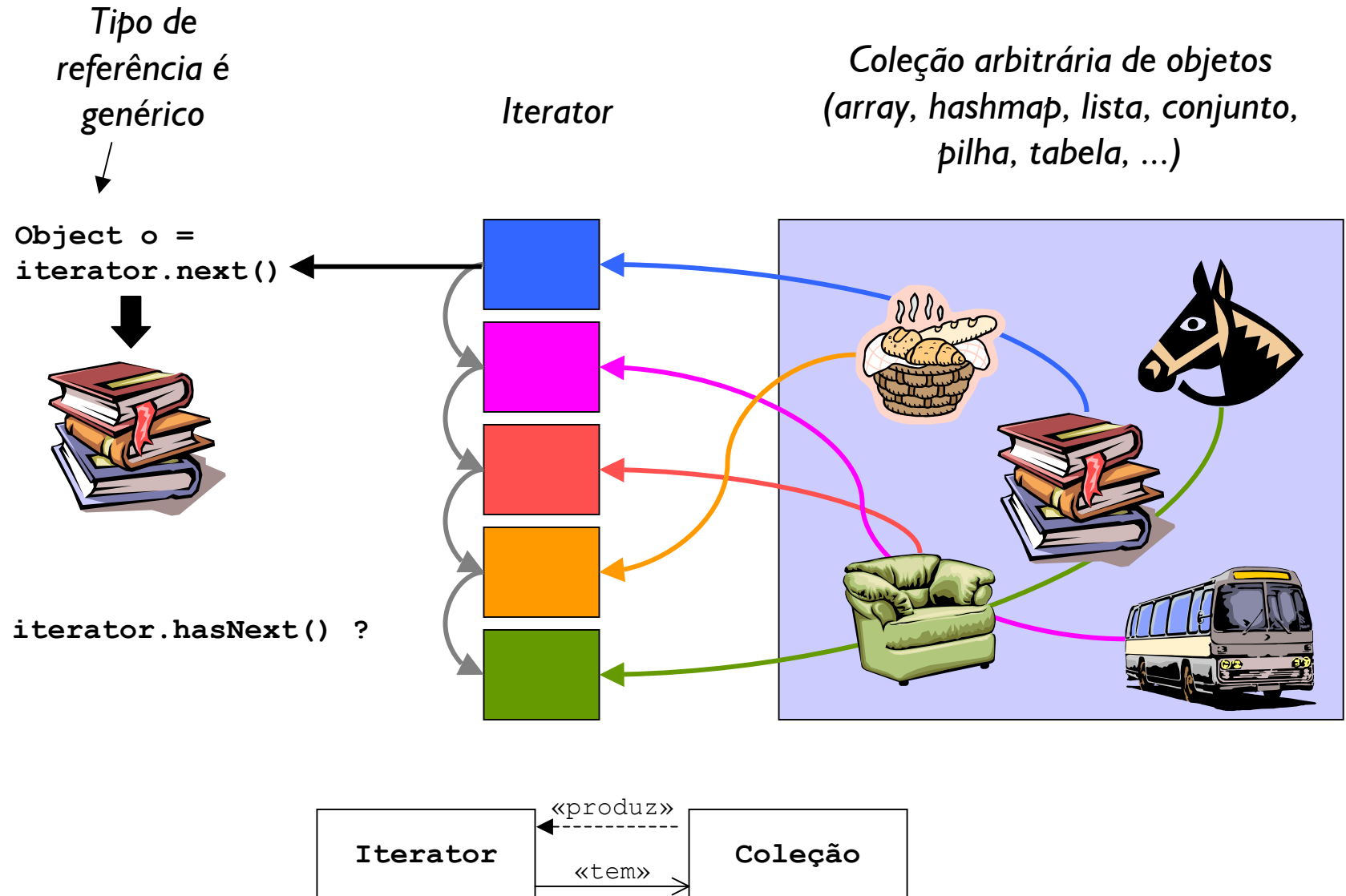
Exercícios

- 21.1 Crie um objeto simples que armazene um texto que possa ser recuperado com um método `getTexto()`. Crie decoradores que retornem o texto: a) em caixa-alta, b) invertido e c) cercado por tags `` e ``. Teste os decoradores individualmente e em cascata.
- 21.2 Crie um decorator `ComandoReader` que possa decorar um `Reader`. O objeto não deve alterar o comportamento dos métodos `read()` originais mas deve oferecer um método `readComando()` que retorna um objeto `Command`
 - O objeto `Command` deve ser construído a partir do stream recebido. Podem ser cinco tipos: `NullCommand`, `NewCommand`, `DeleteCommand`, `GetCommand` e `GetAllCommand`
 - Os strings de entrada devem vir no formato `<comando> <um ou mais espaços em branco> <argumentos>`. O número de argumentos esperados depende do comando: 1) `new id nome`, 2) `delete id`, 3) `get id`, 4) `all`. Comandos incorretos ou desconhecidos retornam `NullCommand`.
- 21.3 Teste o `ComandoReader` passando-lhe um stream de caracteres (leia um string como um stream).

Iterator

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

Problema



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 {  
    boolean hasNext();  
    Object next();  
    void remove();  
}
```
- *`iterator()` é um exemplo de *Factory Method**

Iterator em Java

```
HashMap map = new HashMap();  
map.put("um", new Coisa("um"));  
map.put("dois", new Coisa("dois"));  
(...)  
  
Iterator it = map.values().iterator();  
while(it.hasNext()) {  
    Coisa c = (Coisa)it.next();  
    System.out.println(c);  
}
```

É preciso fazer cast
de todos os objetos
retornados



- *Para implementar um iterator para uma coleção, use delegação:*
 - *Inclua um iterator na classe que gerencia a coleção e um método getIterator() ou similar*
 - *Implemente métodos next(), hasNext(), etc. extraíndo os dados na coleção, fazendo o cast e retornando o objeto no tipo correto.*

Exercícios

- 22.1 *Escreva um type-safe iterator para objetos da hierarquia de Figuras (Circulos, Retangulos, etc.): objetos retornados pelo iterator devem ser do tipo Figura.*
 - *Implemente o Iterator na ListaDeFiguras (use internamente o List.iterator)*
 - *Use um método iterator() e esconda a implementação em uma classe interna*
 - *Use next() e hasNext() para navegar*

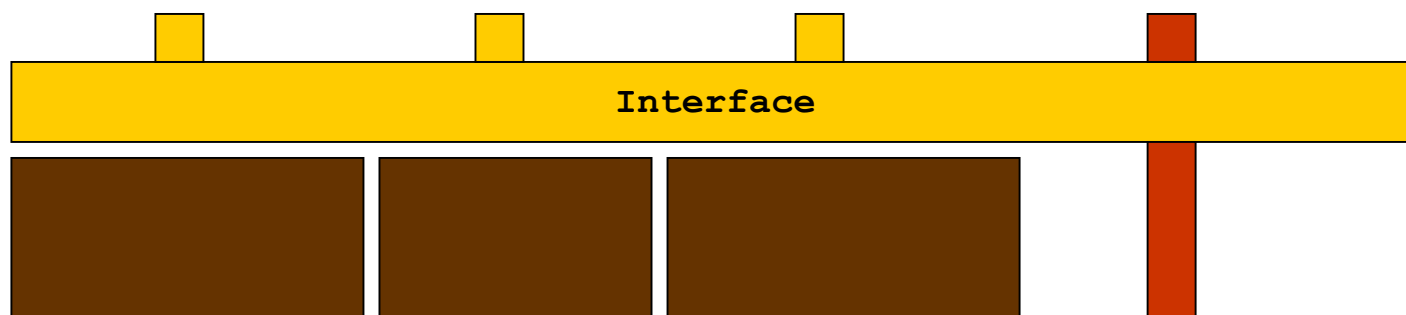
Visitor

"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

Operação de suporte
à extensão



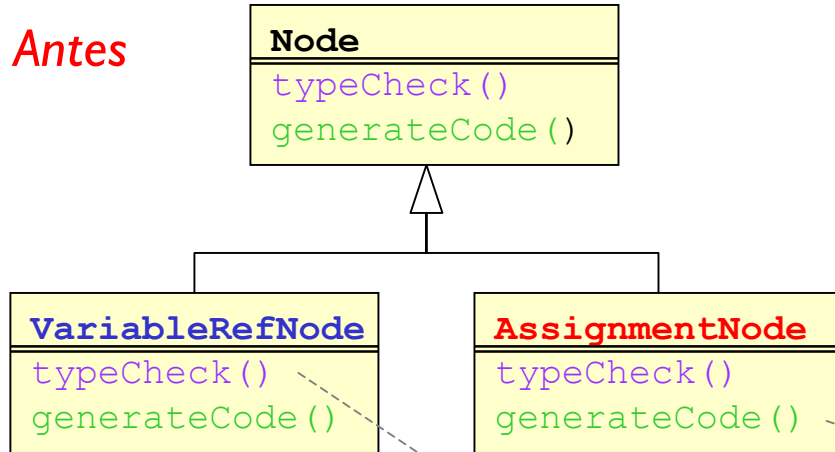
Operações
novas
plugáveis



Para que serve?

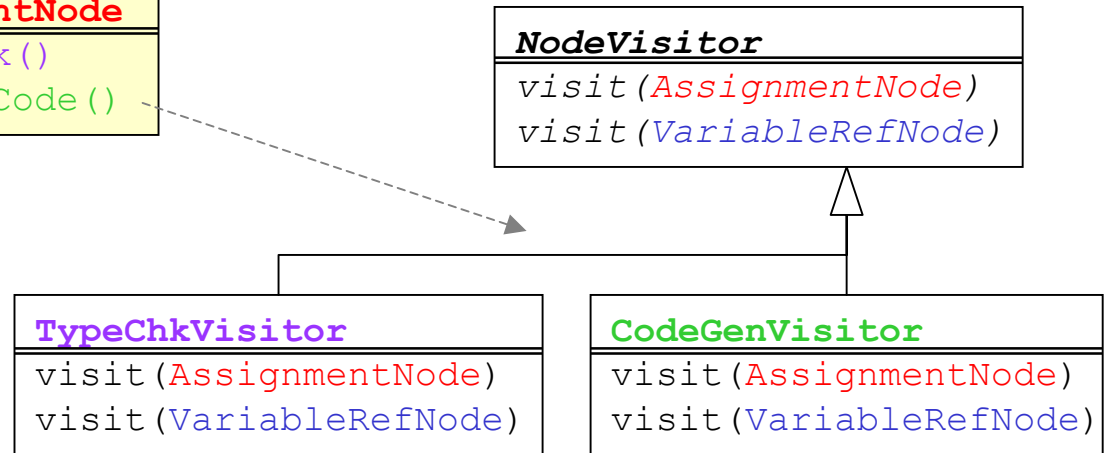
- *Visitor permite*
 - *Plugar nova funcionalidade em objetos sem precisar mexer na estrutura de herança*
 - *Agrupar e manter operações relacionadas em uma classe e aplicá-las, quando conveniente, a outras classes (evitar espalhamento e fragmentação de interesses)*
 - *Implementar um Iterator para objetos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

Antes

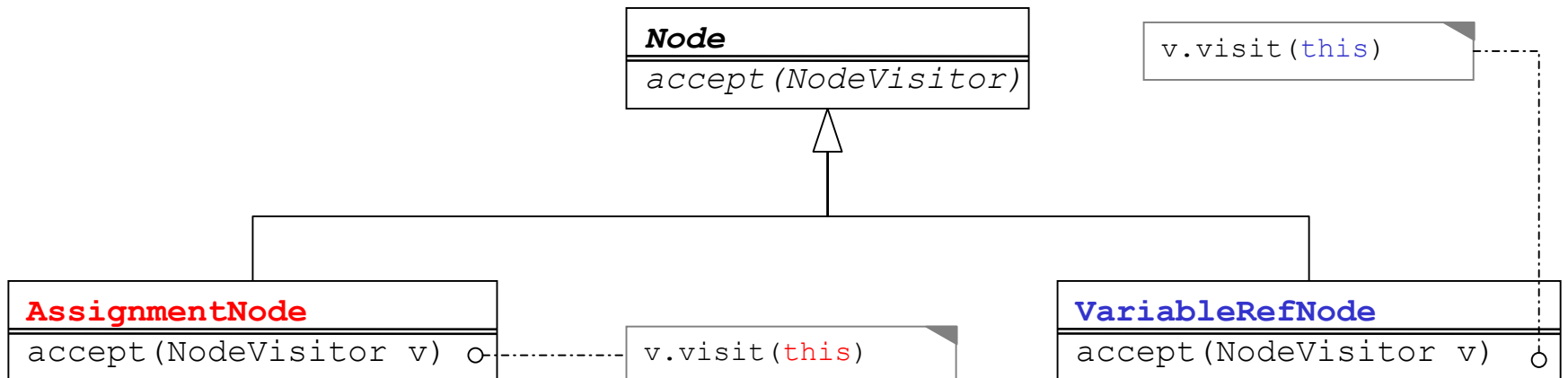


Visitor: exemplo GoF

Depois



Depois



Estrutura de Visitor

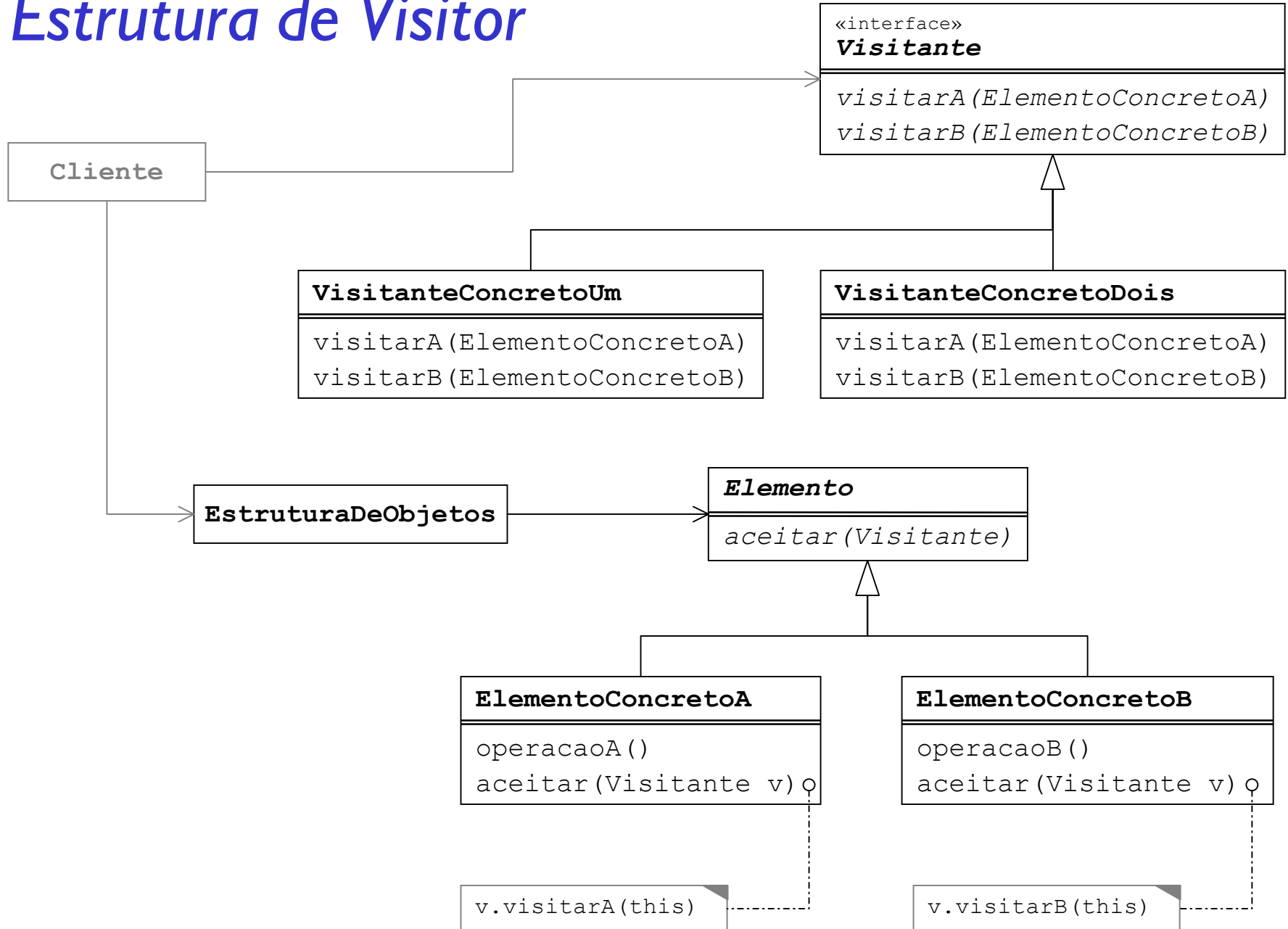
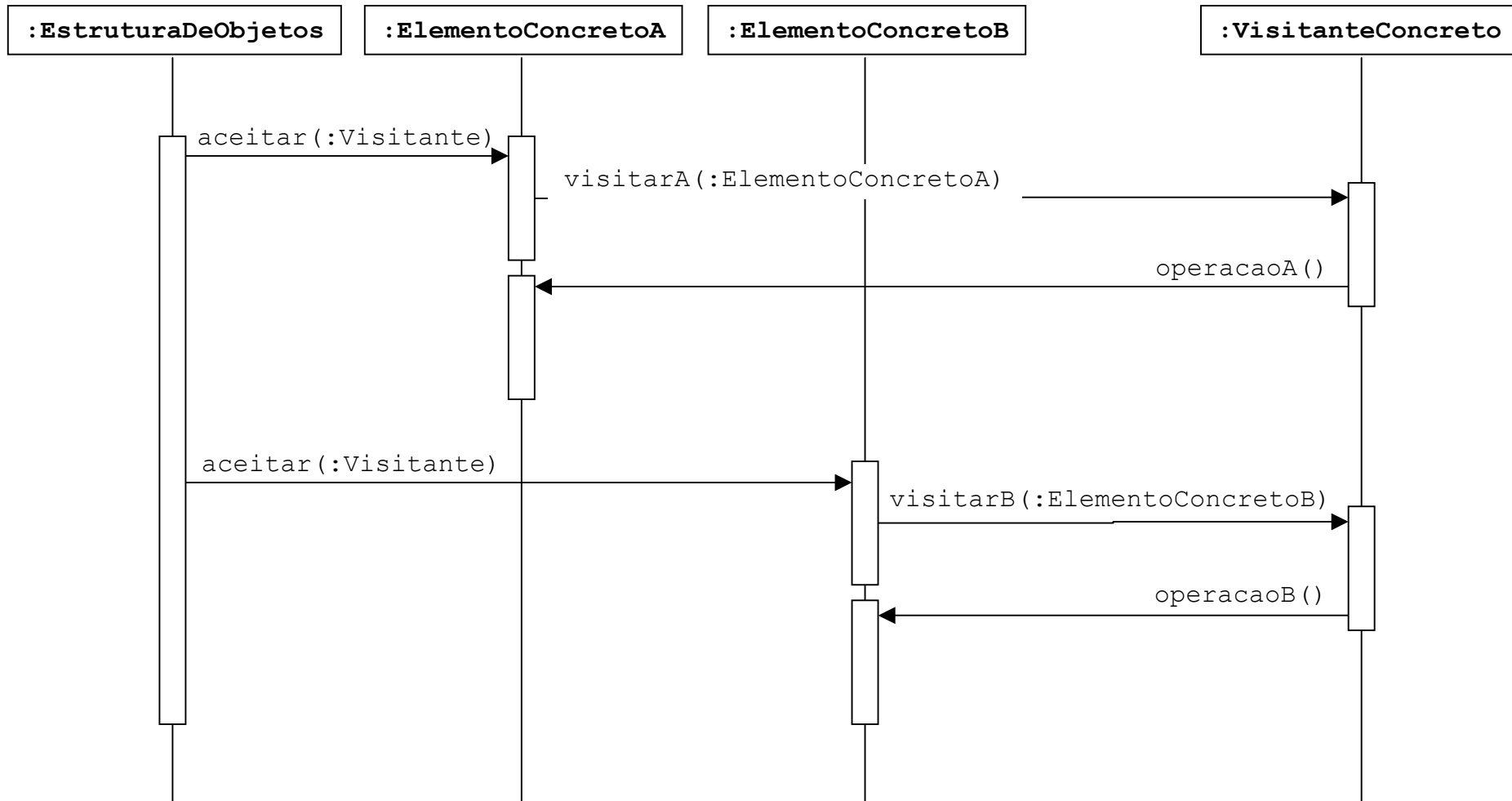


Diagrama de seqüência



Refatoramento para Visitor em Java: Antes

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

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

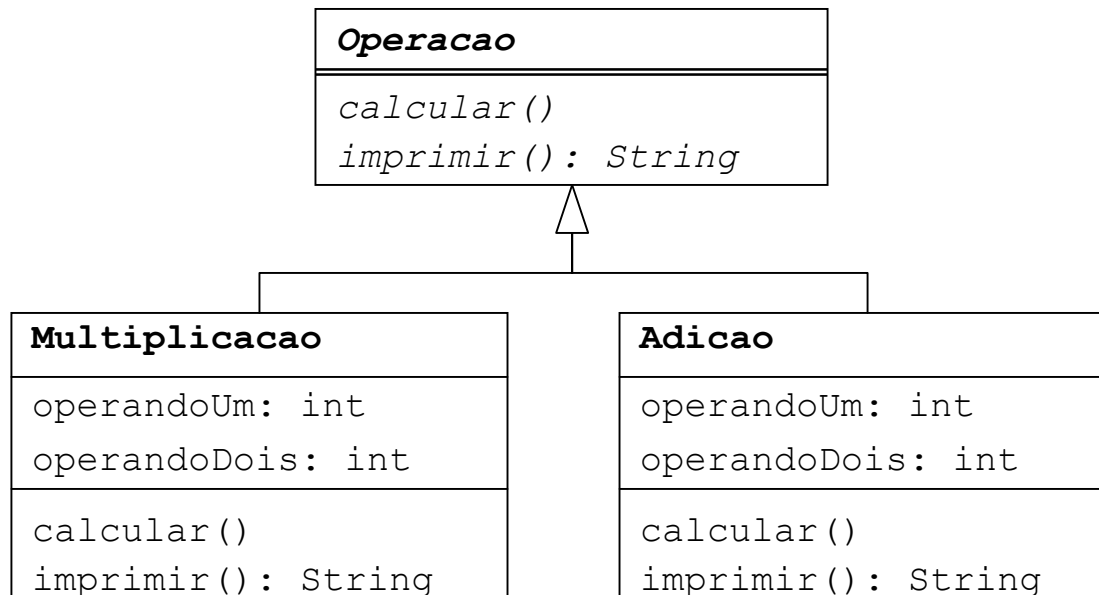
```
public class Grafico implements Documento {  
    public Object aceitar(Visitante v) {  
        return v.visitar(this);  
    }  
    public void gerarPNG() {...}  
    public String toString() {...}  
}
```

Prós e contras

- **Vantagens**
 - *Facilita a adição de novas operações*
 - *Agrupar 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*

Exercícios

- 23.1 Acrescente uma nova operação no exemplo mostrado (Documento) que permita gravar documentos em XML (GerarXML), sem mudar a interface de Documento
 - Use Visitor para implementar a operação
 - Acrescente um método exclusivo da operação GerarXML que simule a geração de XML (apenas imprima "gerando XML" e garanta que este método seja chamado durante a operação.
- 23.2 Refatore a hierarquia abaixo para usar Visitor



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.*

Fontes

- [1] Steven John Metsker, *Design Patterns Java Workbook*. Addison-Wesley, 2002, Caps. 26 a 29. *Exemplos em Java, diagramas em UML e exercícios sobre Decorator, Iterator, Visitor.*
- [2] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995. Decorator, Iterator & Visitor. *Referência com exemplos em C++ e Smalltalk.*