Java
Introdução à Programação por Padrões

Programação III

José Luis Oliveira; Carlos Costa

O que é um padrão de software?

- Metodologia testada e documentada para alcançar um objectivo
  - Padrões são comuns em várias áreas da engenharia
- Design Patterns, ou Padrões de Software
  - Padrões para alcançar objectivos na engenharia de software usando classes e métodos em linguagens orientadas por objectos
  - "Design Patterns" de Erich Gamma, John Vlissides, Ralph Jonhson e Richard Helm, conhecidos como "The Gang of Four", ou GoF, descreve 23 padrões.

Porquê 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
  - Conhecendo os padrões torna-se mais fácil a compreensão de sistemas existentes
- Desenvolver software de melhor qualidade
  - Os padrões utilizam eficientemente polimorfismo, herança, modularidade, composição, abstracção para construir código reutilizável, eficiente, de alta coesão e baixo acoplamento
- Usar metáforas comuns a diferentes linguagens e frameworks de desenvolvimento
  - Faz o sistema ficar menos complexo ao permitir que se fale num nível mais alto de abstracção

Formas de classificação

- Há várias formas de classificar os padrões. GoF classifica-os de duas formas:
  - Por propósito: (1) criação de classes e objectos, (2) alteração da estrutura de um programa, (3) controlo do seu comportamento
  - Por alcance: classe ou objecto
- Metsker classifica-os em 5 grupos, por intenção (problema a ser solucionado):
  - (1) oferecer uma interface,
  - (2) atribuir uma responsabilidade,
  - (3) realizar a construção de classes ou objectos
  - (4) controlar formas de operação
  - (5) implementar uma extensão para a aplicação

Classificação dos 23 padrões segundo GoF

	Propósito		
	1. Criação	2. Estrutura	3. Comportamento
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

Classificação dos padrões segundo Metsker

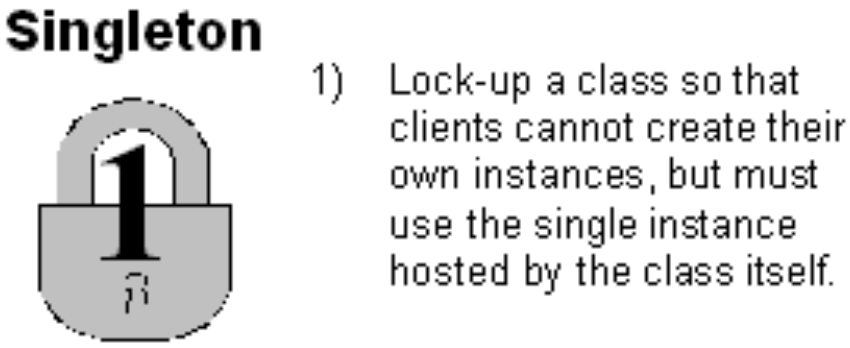
Intenção	Padrões
1. Interfaces	Adapter, Facade, Composite, Bridge
2. Responsabilidade	Singleton, Observer, Mediator, Proxy, Chain of Responsibility, Flyweight
3. Construção	Builder, Factory Method, Abstract Factory, Prototype, Memento
4. Operações	Template Method, State, Strategy, Command, Interpreter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

Alguns padrões …

- Singleton
  - Garantir que uma classe só tenha uma única instância fornecendo um ponto de acesso global
- Decorator
  - Anexar responsabilidades adicionais a um objecto dinamicamente
- Iterator
  - Fornecer uma maneira de aceder sequencialmente a elementos de uma colecção sem expor a sua representação interna
- Factory
  - Definir uma interface para criar um objecto mas deixar que subclasses decidam que classe instanciar

Singleton

- Problema
  - "Garantir que uma classe só tenha uma única instância, acedida a partir de um ponto de acesso global." [GoF]
- Aplicações
  - Driver de acesso a uma base de dados
  - Sistema de Ficheiros
  - Ficheiro de Log



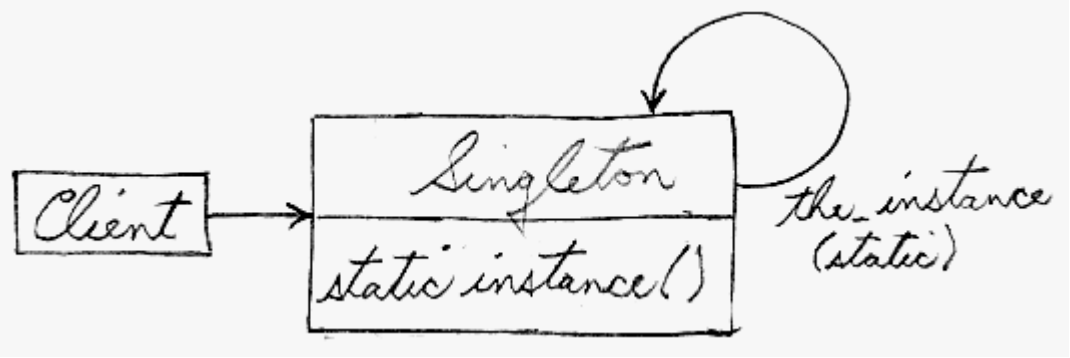
http://www.vincehuston.org/dp/singleton.html

Questões?

- Como garantir que só existe um objecto da classe?
- Como controlar ou impedir a construção normal?
- Como definir o acesso a uma única instância?
- Como garantir que o sistema funciona se a classe participar numa hierarquia de classes?

Singleton Pattern

Singleton
-instance : Singleton
-Singleton() +getInstance() : Singleton



Singleton em Java

```
class Singleton {
    private String name;

    static private Singleton instance =
      new Singleton("Ermita");
    private Singleton(String name) {
      this.name = name;
    }
    static public Singleton getInstance() {
      return instance;
    }
}
```

```
Singleton errado = new Singleton("Coelho");
```

```
Singleton um = Singleton.getInstance();
Singleton dois = Singleton.getInstance();
Singleton tres = Singleton.getInstance();
System.out.printf("%s\n%s\n%s\n", um, dois, tres);
```

```
Singleton@1e0be38
Singleton@1e0be38
Singleton@1e0be38
```

Singleton: Princípios

- Definir o construtor como privado (ou protected)

```
private Singleton(String name)
```

- Definir uma referência estática privada para apontar para o único objecto da classe

```
static private Singleton instance
```

- Definir um método de acesso a essa instância

```
static public Singleton getInstance()
```

  - Os clientes poderão aceder ao objecto através deste (ou de outros) métodos

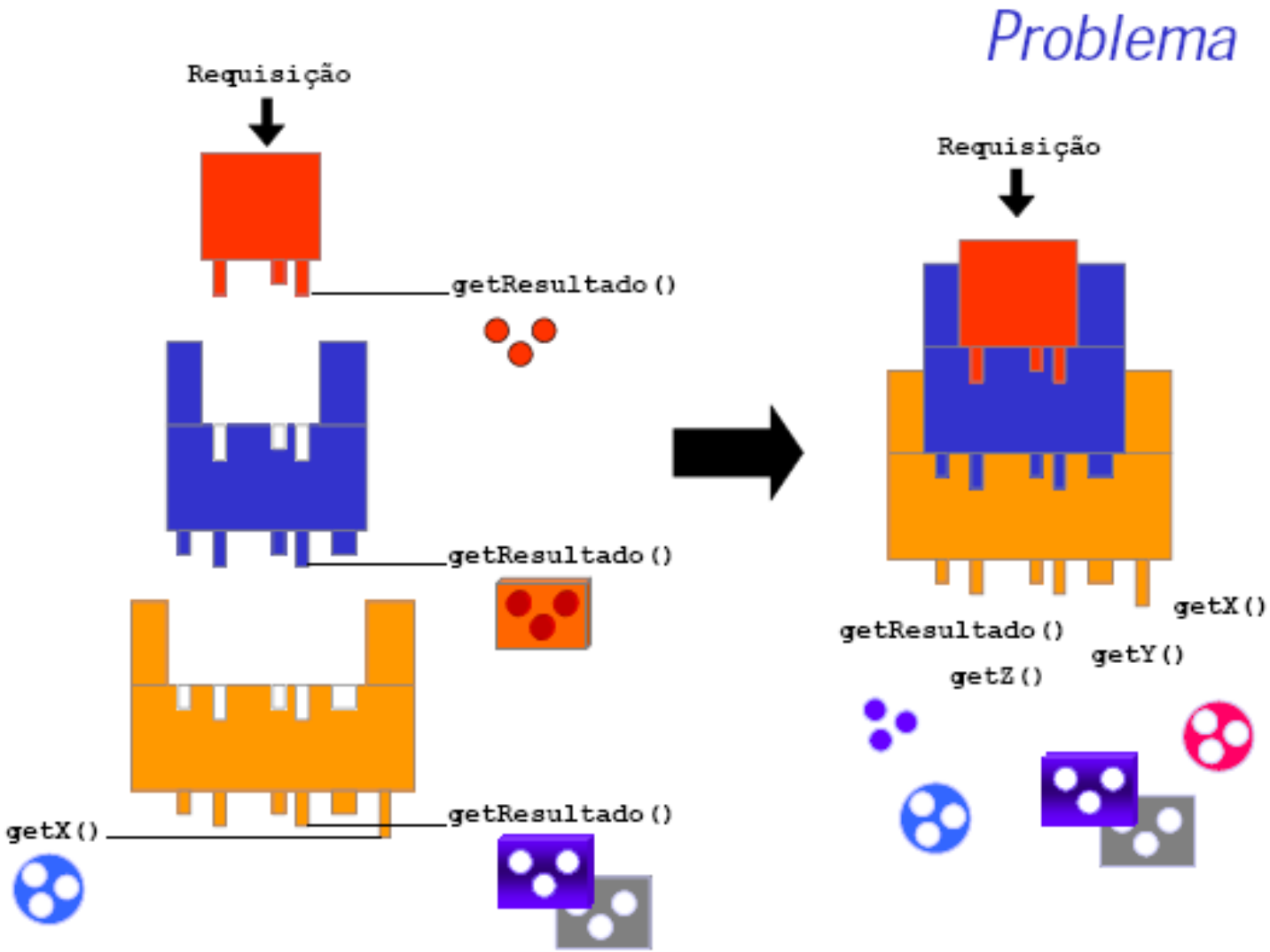
Decorador (*Decorator*)

- Problema
  - "Anexar responsabilidades adicionais a um objecto dinamicamente. Os Decoradores oferecem uma alternativa flexível ao uso de herança para estender uma funcionalidade." [GoF]
- Aplicações
  - Sistema de I/O em Java
  - Susbtituição de herança (múltipla e complexa)

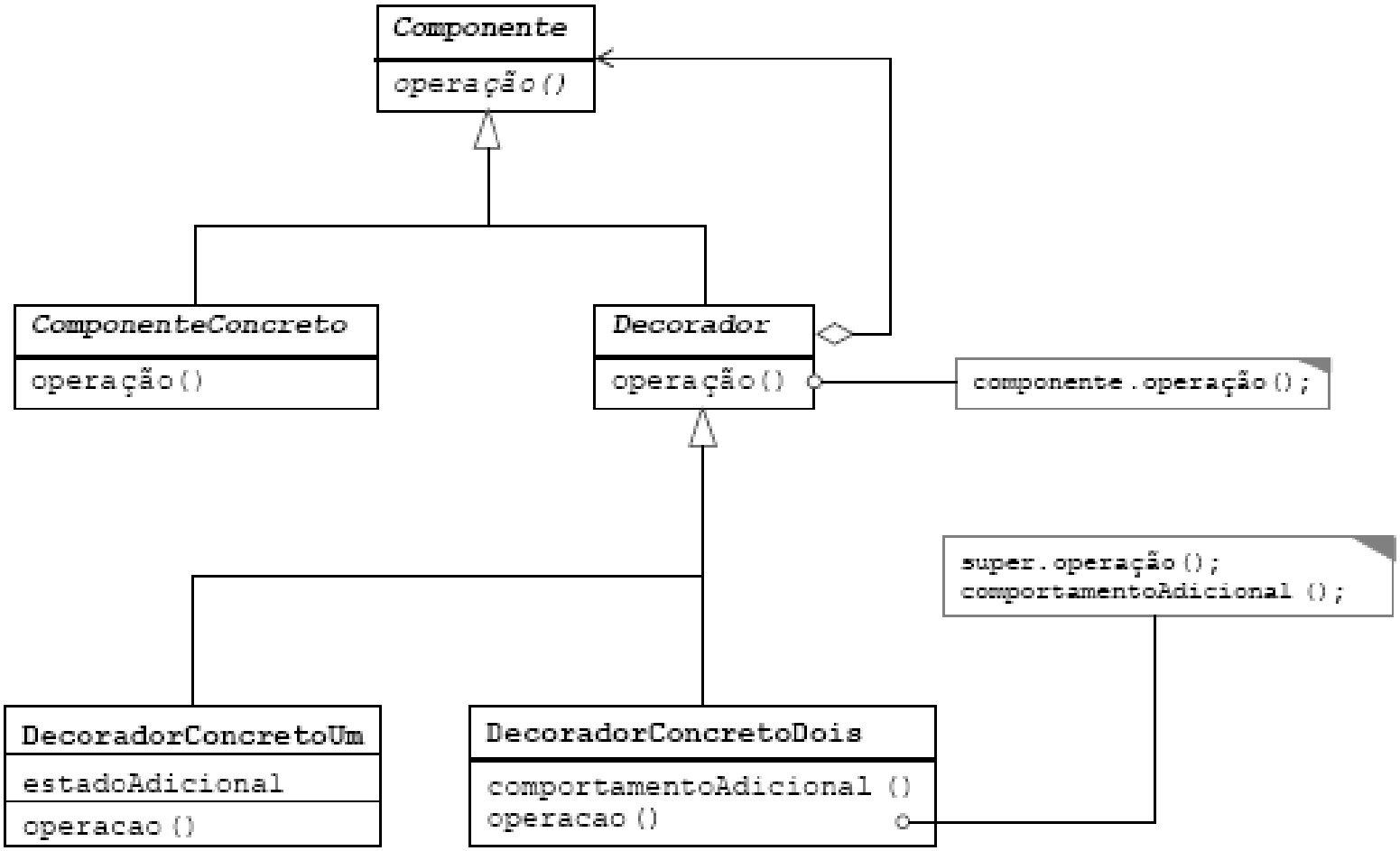
Questões?

- Considere as seguintes entidades:
  - Futebolista (joga, passa, remata),
  - Tenista (joga, serve),
  - GuardaRedes (joga, passa, remata, defendeBaliza)
  - Jogador (joga)
  - ...Quais as relações entre estas classes?
- Vamos complicar:
  - O Rui joga Basquete e Futebol
  - A Ana joga Badminton e Basquete
  - O Paulo joga Xadrez, Futebol e Basquete
  - .... Solução?

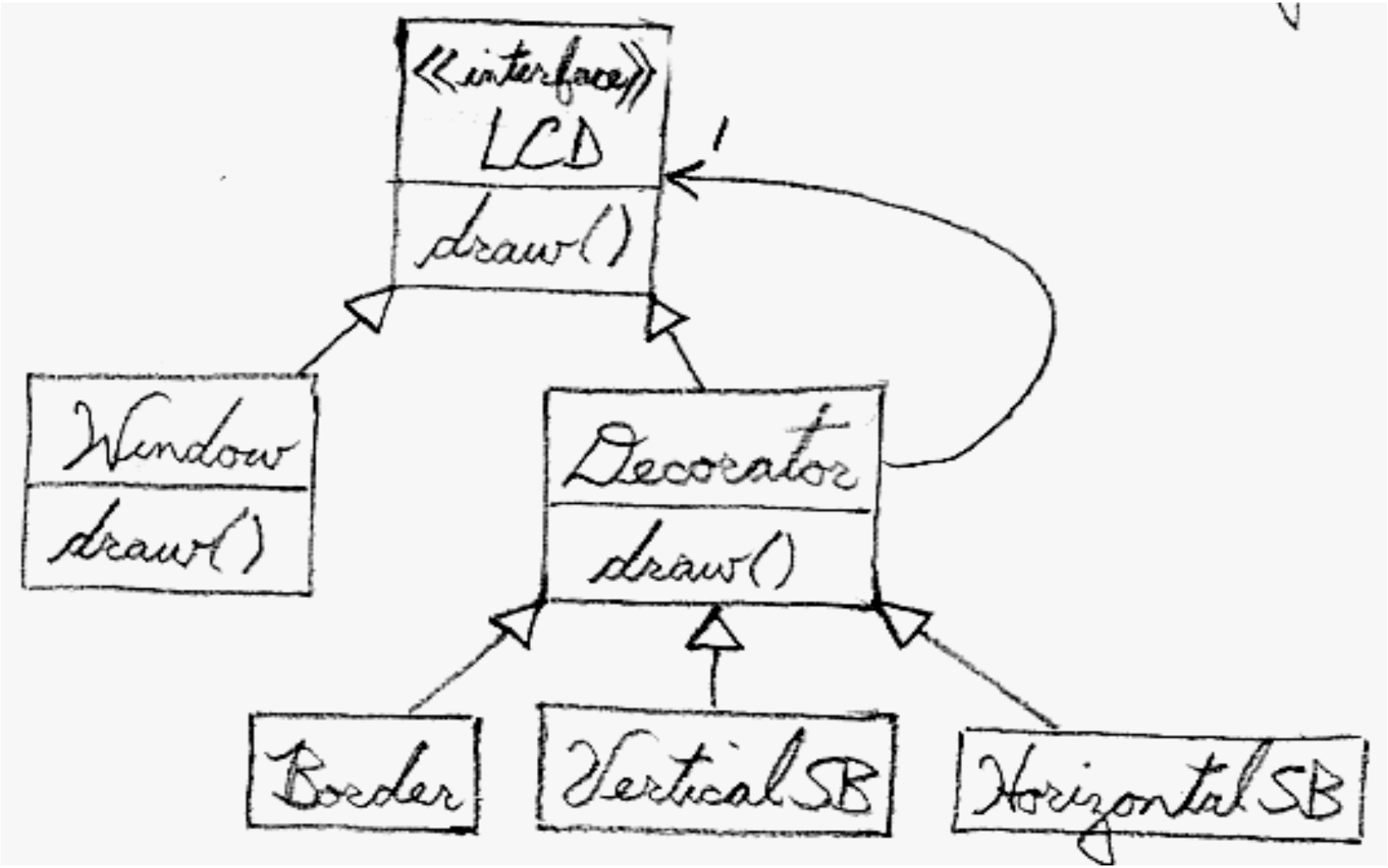
Decorador


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

Modelo de um Decorador



Modelo de um Decorador - Exemplo



Exemplo de Decorador

```
interface JogadorInterface {
    void joga();
}

class Jogador implements JogadorInterface {
    private String name;
    Jogador(String n) { name = n; }
    @Override public void joga()
        { System.out.print("\n"+name+" joga "); }
}

abstract class JogDecorator implements JogadorInterface {
    protected JogadorInterface j;
    JogDecorator(JogadorInterface j) { this.j = j; }
    public void joga()       { j.joga(); }
}
```

Exemplo de Decorador

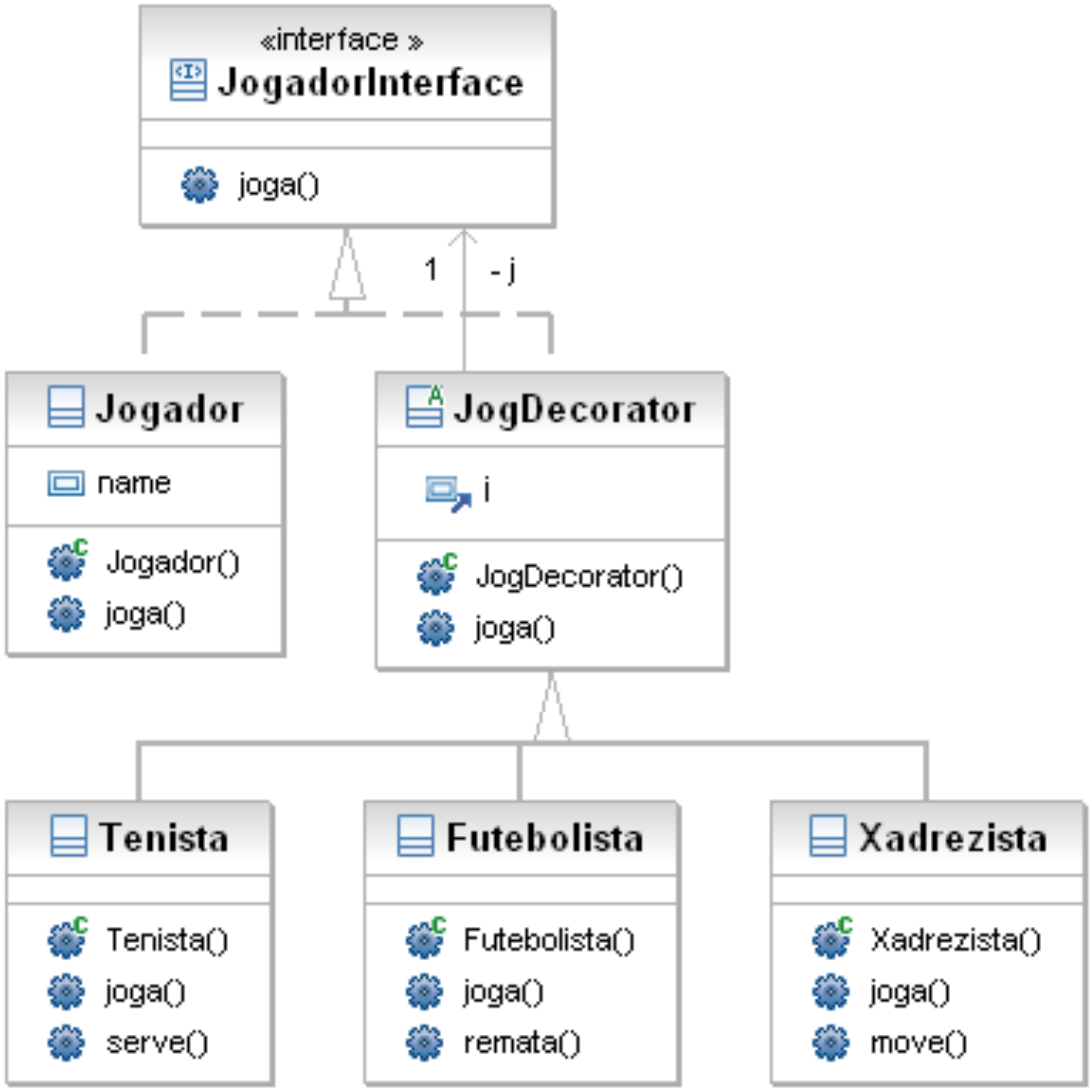
```
class Futebolista extends JogDecorator {
    Futebolista(JogadorInterface j) { super(j); }
    @Override public void joga()
       { j.joga(); System.out.print("futebol ");  }
    public void remata()  { System.out.println("-- Remata!"); }
}

class Xadrezista extends JogDecorator {
    Xadrezista(JogadorInterface j) { super(j); }
    @Override public void joga()  { j.joga();
       System.out.print("xadrez ");  }
}

class Tenista extends JogDecorator {
    Tenista(JogadorInterface j) { super(j); }
    @Override public void joga()
       { j.joga(); System.out.print("tenis ");  }
    public void serve()  { System.out.println("-- Serve!"); }
}
```

Exemplo de Decorador

```
public class Composition {
    public static void main(String args[]) {
        JogadorInterface j1 = new Jogador("Rui");
        Futebolista f1 = new Futebolista(new Jogador("Luis"));
        Xadrezista x1 = new Xadrezista (new Jogador("Ana"));
        Xadrezista x2 = new Xadrezista (j1);
        Xadrezista x3 = new Xadrezista (f1);
        Tenista t1 = new Tenista(j1);
        Tenista t2 =
          new Tenista(
            new Xadrezista (
              new Futebolista(
                new Jogador("Bruna"))));

        JogadorInterface lista[] = { j1, f1, x1, x2, x3, t1, t2 };
        for (JogadorInterface ji: lista)
          ji.joga();
    }
}
```

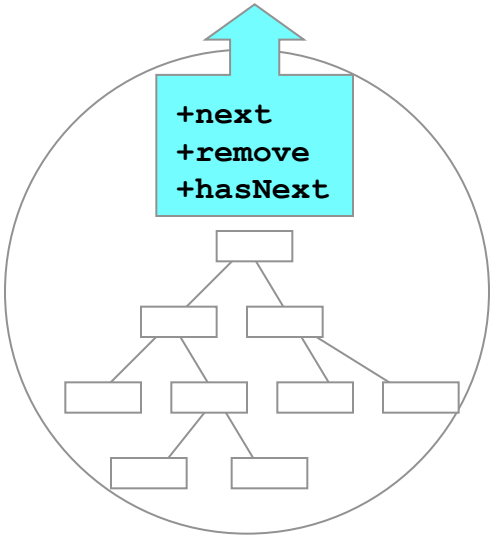
```
Rui joga
Luis joga futebol
Ana joga xadrez
Rui joga xadrez
Luis joga futebol xadrez
Rui joga tenis
Bruna joga futebol xadrez tenis
```

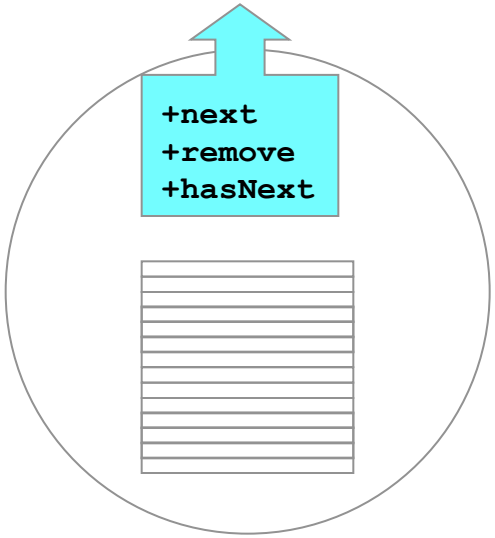
Exemplo de Decorador

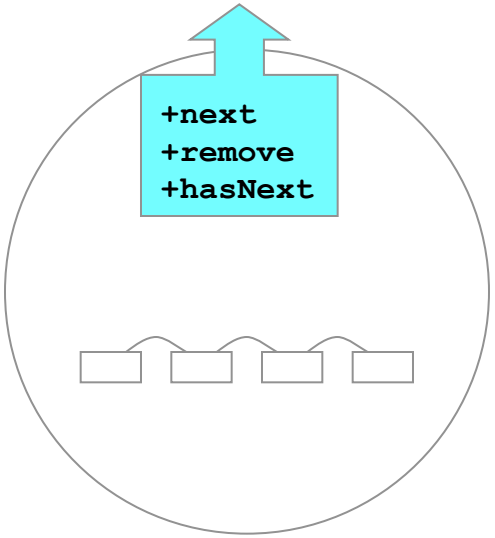


Iterador

- Problema
  - “Fornecer uma forma comum de aceder aos elementos de um objecto agregado sequencialmente sem expôr a sua representação interna." [GoF]
- Aplicações







Exemplo - Listar colecções

```
    // For a set or list
for (Iterator it=collection.iterator(); it.hasNext(); ) {
    Object element = it.next();
}

    // For keys of a map
for (Iterator it=map.keySet().iterator(); it.hasNext(); ) {
    Object key = it.next();
}

    // For values of a map
for (Iterator it=map.values().iterator(); it.hasNext(); ) {
    Object value = it.next();
}

    // For both the keys and values of a map
for (Iterator it=map.entrySet().iterator(); it.hasNext(); ) {
    Map.Entry entry = (Map.Entry)it.next();
    Object key = entry.getKey();
    Object value = entry.getValue();
}
```

Operações

Colecção

- Construir um iterador - uma Fábrica devolve um iterador que aponta para o primeiro elemento do conjunto
  - *Collection*.iterator()

Iterador

- Devolver a referência para o próximo elemento
  - next();
- Determinar se ainda existem mais elementos na sequência
  - boolean hasNext();
- Remover o elemento da colecção apontado pelo iterador
  - remove();

Iteradores em Java

- Os iteradores são implementados nas colecções de Java
- Um iterador pode ser obtido através do método *Collection.iterator()* que devolve uma instância de java.util.Iterator.
  - Interface java.util.Iterator:

```
package java.util;

public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

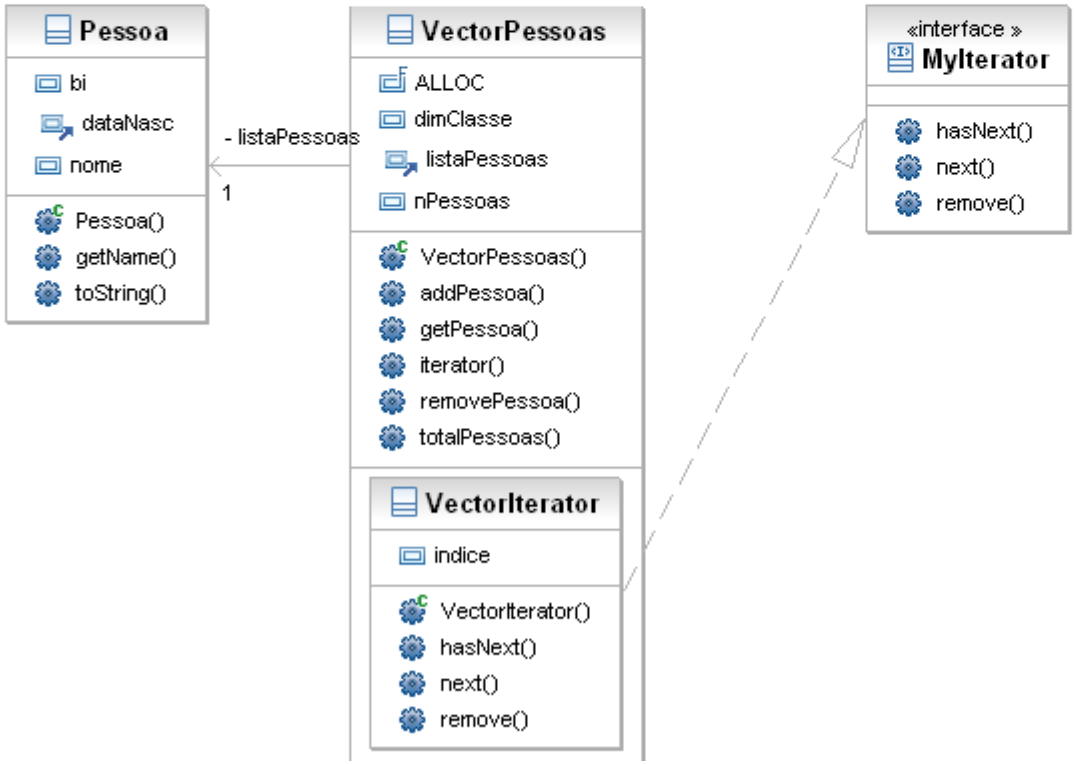
Iteradores em Java

- Para implementar um iterador numa colecção, deve usar-se delegação:
  - Incluir um iterador na classe que gere o conjunto e um método iterator() ou similar
  - Implemente métodos next(), hasNext(), etc. extraíndo os dados na colecção.

```
VectorPessoas vp = new VectorPessoas();
for (int i=0; i<10; i++)
    vp.addPessoa(new Pessoa("Bebé_"+i, 1000+i, Data.today()));
MyIterator vec = vp.iterator();
while ( vec.hasNext() )
    System.out.println( vec.next() );
```

Exemplo: Conjunto de Pessoas

```
public interface MyIterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

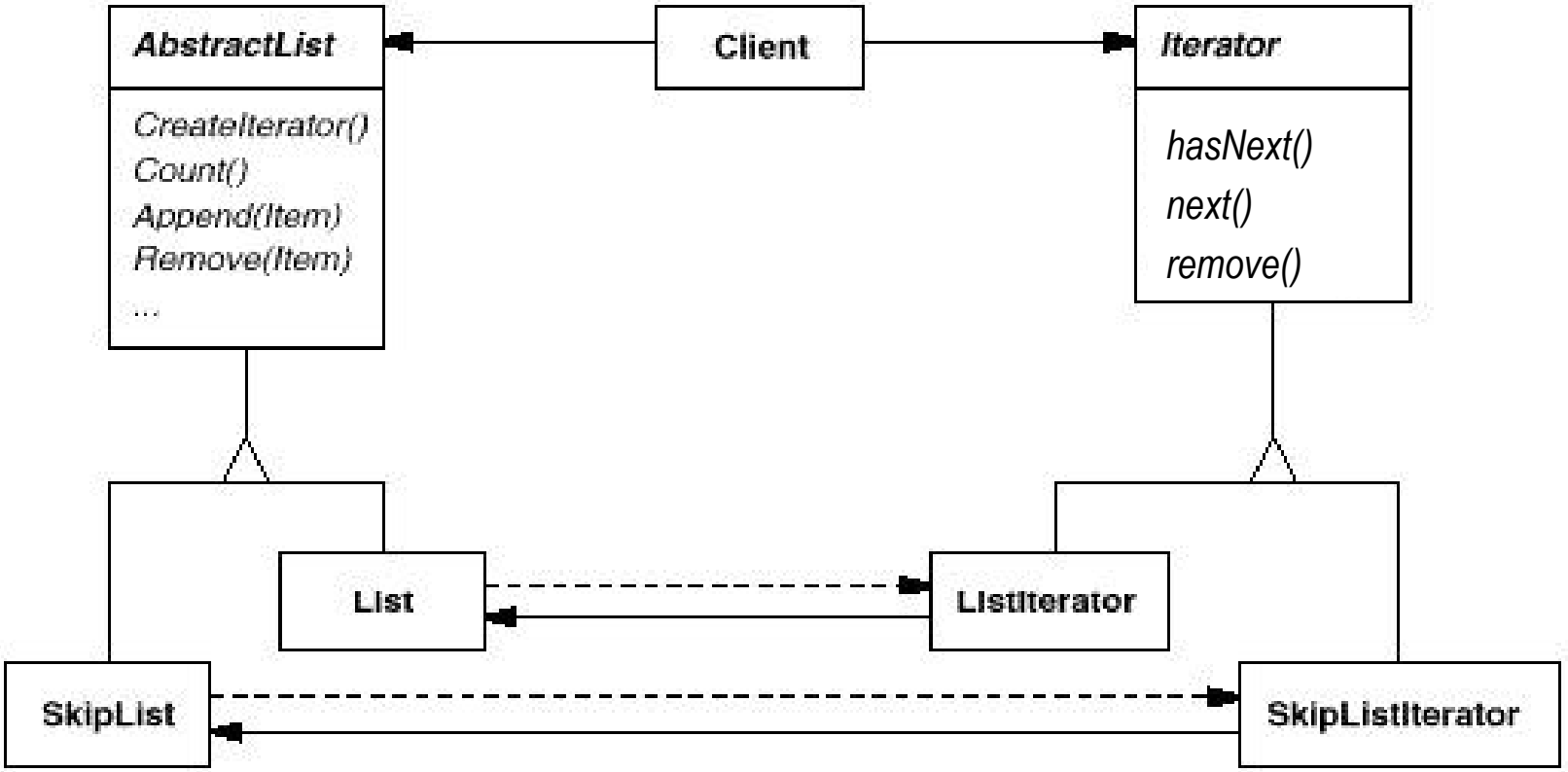


```
public class VectorPessoas {
   private Pessoa[] listaPessoas;
   private int nPessoas;
   private final int ALLOC = 50;
   private int dimClasse = ALLOC;

   public VectorPessoas() {
        listaPessoas = new Pessoa[ALLOC];
        nPessoas = 0;
   }

   public boolean addPessoa(Pessoa est) {
        if (est == null)
           return false;
        if (nPessoas>=dimClasse) {
           dimClasse += ALLOC;
           Pessoa[] newArray = new Pessoa[dimClasse];
           System.arraycopy(listaPessoas, 0, newArray, 0, nPessoas );
           listaPessoas = newArray;
        }
        listaPessoas[nPessoas++] = est;
        return true;
   }
```

```
public boolean removePessoa(Pessoa p) {
     for (int i = 0; i < nPessoas; i++) {
             if (listaPessoas[i] == p) {
                     nPessoas--;
                     for (int j = i; j < nPessoas; j++)
                             listaPessoas[j] = listaPessoas[j + 1];
                     return true;
             }
     }
     return false;
}

public int totalPessoas() {
     return nPessoas;
}

public Pessoa getPessoa(int i) {
     return listaPessoas[i];
}

MyIterator iterator() {
     return (this).new VectorIterator();
}
```

```
    private class VectorIterator implements MyIterator{
        private int indice;
        VectorIterator() {
                indice = 0;
        }
        public boolean hasNext() {
                return (indice < nPessoas);
        }
        public Object next() {
            if (hasNext()) {
                Object r = listaPessoas[indice];
                indice++;
                return r;
            }
            throw new IndexOutOfBoundsException("only "
                + nPessoas + " elements");
        }
        public void remove() {
            throw new UnsupportedOperationException(
              "Operação não suportada!");
        }
    }
}
```

Iterador genérico



Fábrica (*Factory Method*)

- Problema
  - "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]
- Aplicações
  - Iteradores - Um iterador é um exemplo de Factory Method

Como implementar?

- É possível criar um objecto sem ter conhecimento algum de sua classe concreta?
  - Esse conhecimento deve estar em alguma parte do sistema, mas não precisa estar no cliente
  - Factory Method define uma interface comum para criar objectos
  - O objecto específico é determinado nas diferentes implementações dessa interface
- Cria-se uma classe Factory com um método estático que, com base num parâmetro de entrada, decide qual o construtor (privado) a usar
- O objecto criador pode ser seleccionado com base noutros critérios que não requeiram parâmetros

Exemplo

```
interface Arvore {
    void regar();
    void colherFruta();
}

class Figueira implements Arvore {
    protected Figueira() {System.out.println("Figueira plantada."); }
    public void regar() { System.out.println("Figueira: Regar muito pouco"); }
    public void colherFruta() { System.out.println("Hum.. figos!"); }
}

class Pessegueiro implements Arvore {
    protected Pessegueiro() {System.out.println("Pessegueiro plantado."); }
    public void regar() { System.out.println("Pessegueiro: Regar normal"); }
    public void colherFruta() { System.out.println("Boa.. pêssegos!"); }
}

class Nespereira implements Arvore {
    protected Nespereira() {System.out.println("Nespereira plantada."); }
    public void regar() { System.out.println("Nespereira: Regar pouco"); }
    public void colherFruta() { System.out.println("Ahh.. nêsperas!"); }
}
```

Exemplo

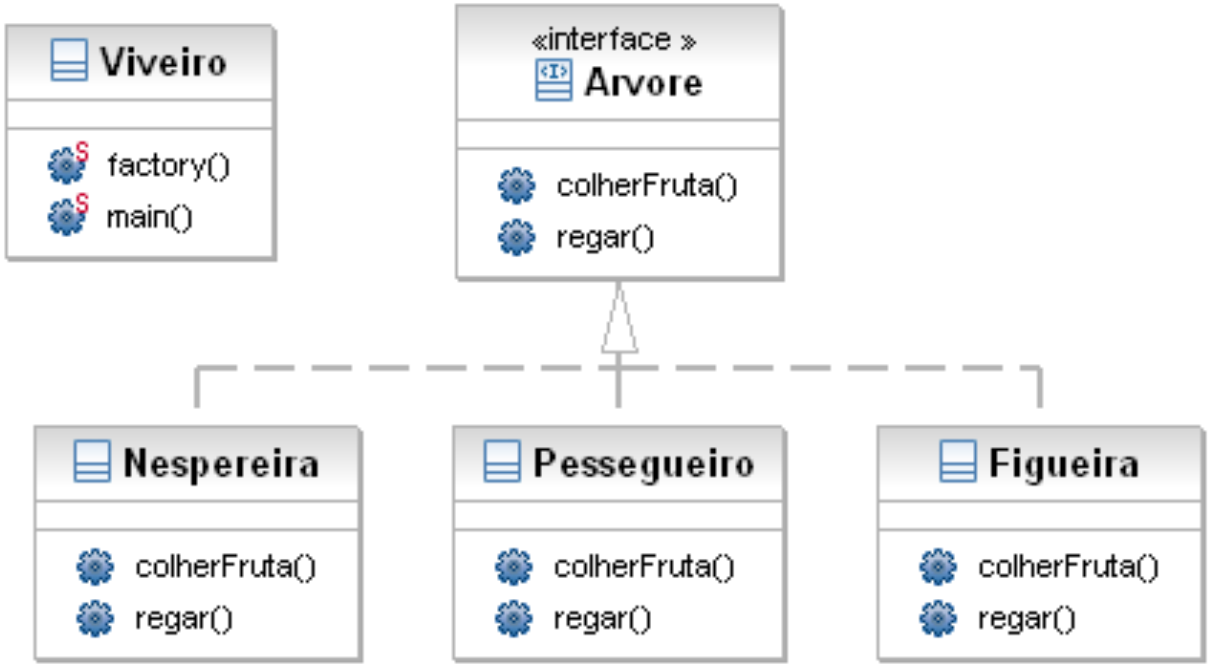
```
class Viveiro {
    public static Arvore factory(String pedido) {
       if (pedido.equalsIgnoreCase("Figueira"))
         { return new Figueira(); }
       if (pedido.equalsIgnoreCase("Pessegueiro"))
         { return new Pessegueiro(); }
       if (pedido.equalsIgnoreCase("Nespereira"))
         { return new Nespereira(); }
       else
         throw new IllegalArgumentException(”Árvore não existente!");
}
```

Exemplo

```
public static void main(String[] args) {
    Arvore pomar[] = {
        Viveiro.factory("Figueira"),
        Viveiro.factory("Pessegueiro"),
        Viveiro.factory("Figueira"),
        Viveiro.factory("Nespereira")
    };
    for (Arvore a: pomar)
      a.regar();
    for (Arvore a: pomar)
      a.colherFruta();
    }
}
```

```
Figueira plantada.
Pessegueiro plantado.
Figueira plantada.
Nespereira plantada.
Figueira: Regar muito pouco
Pessegueiro: Regar normal
Figueira: Regar muito pouco
Nespereira: Regar pouco
Hum.. figos!
Boa.. pêssegos!
Hum.. figos!
Ahh.. nêsperas!
```

Exemplo - Estrutura



Sumário

- Singleton, Iterador, Decorador, Fábrica .. mas há muitos outros padrões de programação
- Tal como os algoritmos ajudam a resolver problemas computacionais os padrões contribuem para desenvolver software mais bem estruturado