

Padrões de Projeto de Software Orientado a Objetos

# Mediator, Iterator, Strategy

Profa. Thienne Johnson

# Conteúdo

- ▶ E. Gamma and R. Helm and R. Johnson and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

## Design Patterns

Elements of Reusable  
Object-Oriented Software

Erich Gamma  
Richard Helm  
Ralph Johnson  
John Vlissides



Foreword by Grady Booch



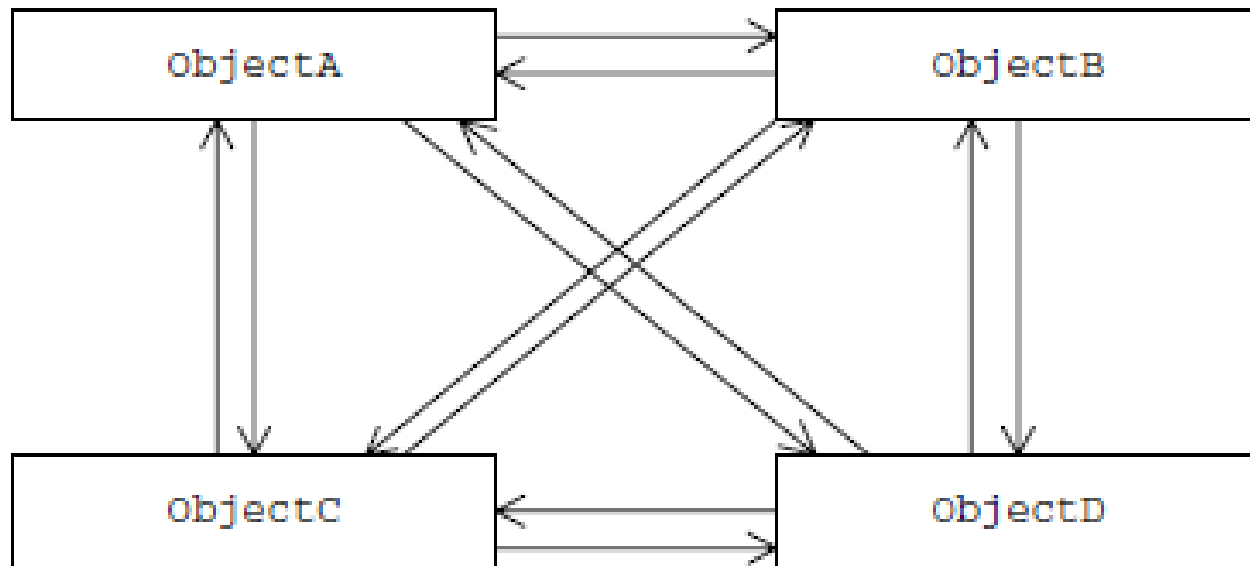
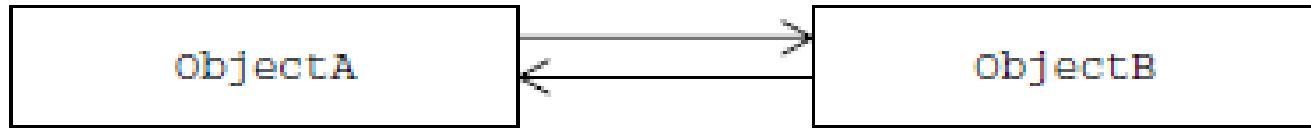
# Padrões comportamentais

- ▶ Os padrões comportamentais se preocupam com algoritmos e a atribuição de responsabilidade entre os objetos.
- ▶ Os padrões comportamentais não descrevem apenas padrões de objetos ou classes, mas também padrões de comunicação entre eles.

# Mediator

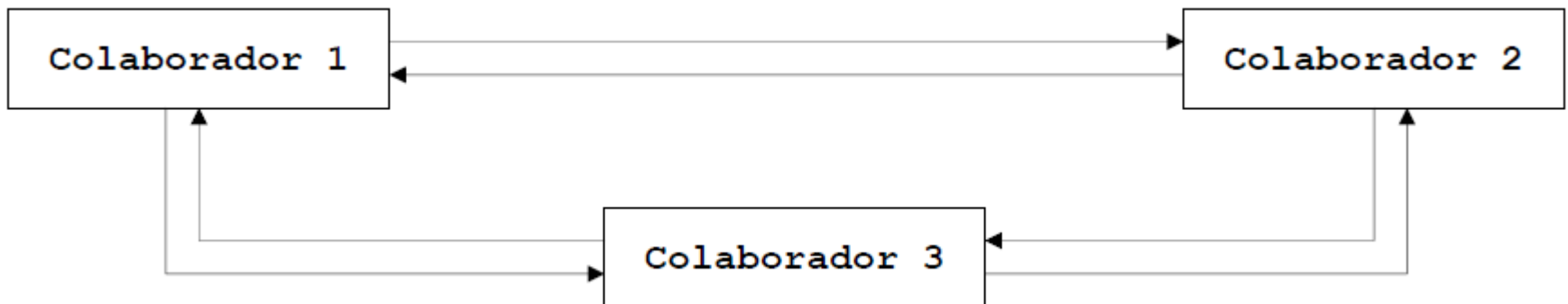
Padrão Comportamental

# Problema



# 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?

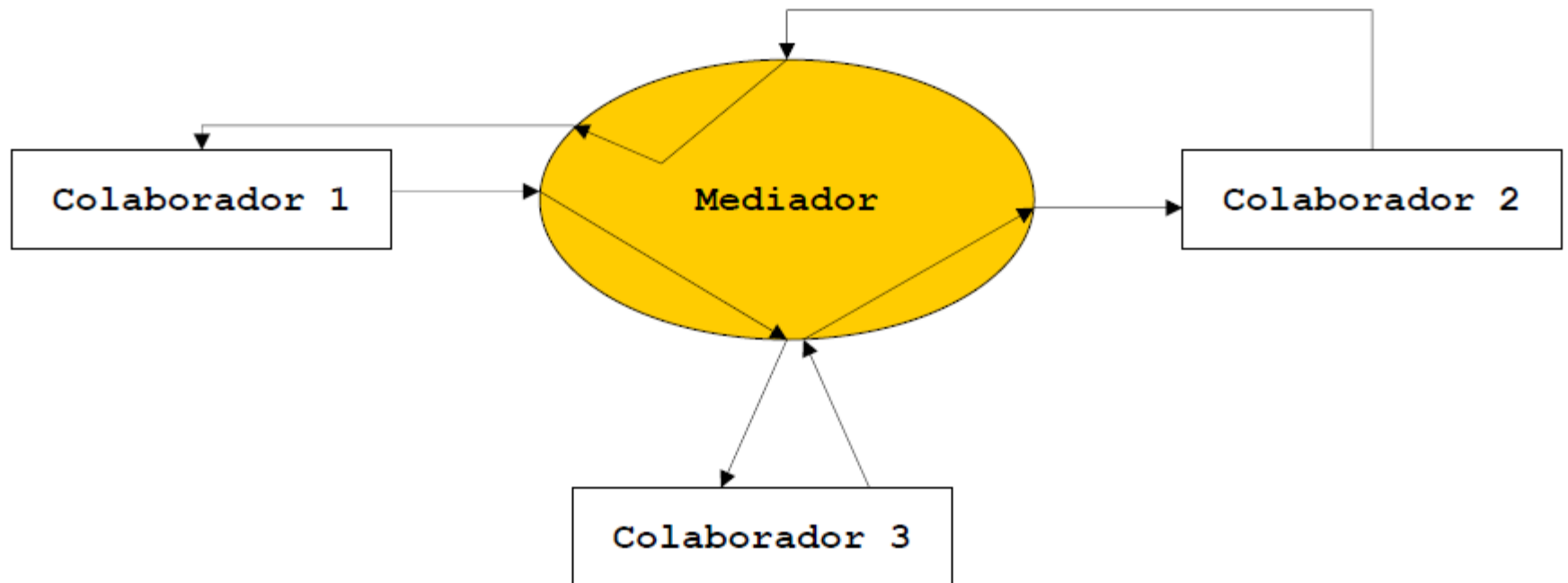


# Intenção

- ▶ Define um objeto que encapsula a comunicação entre um grupo de objetos
- ▶ Promove acoplamento fraco ao não permitir que objetos se refiram uns aos outros explicitamente,
  - Isso permite que sua interação seja variada independentemente.

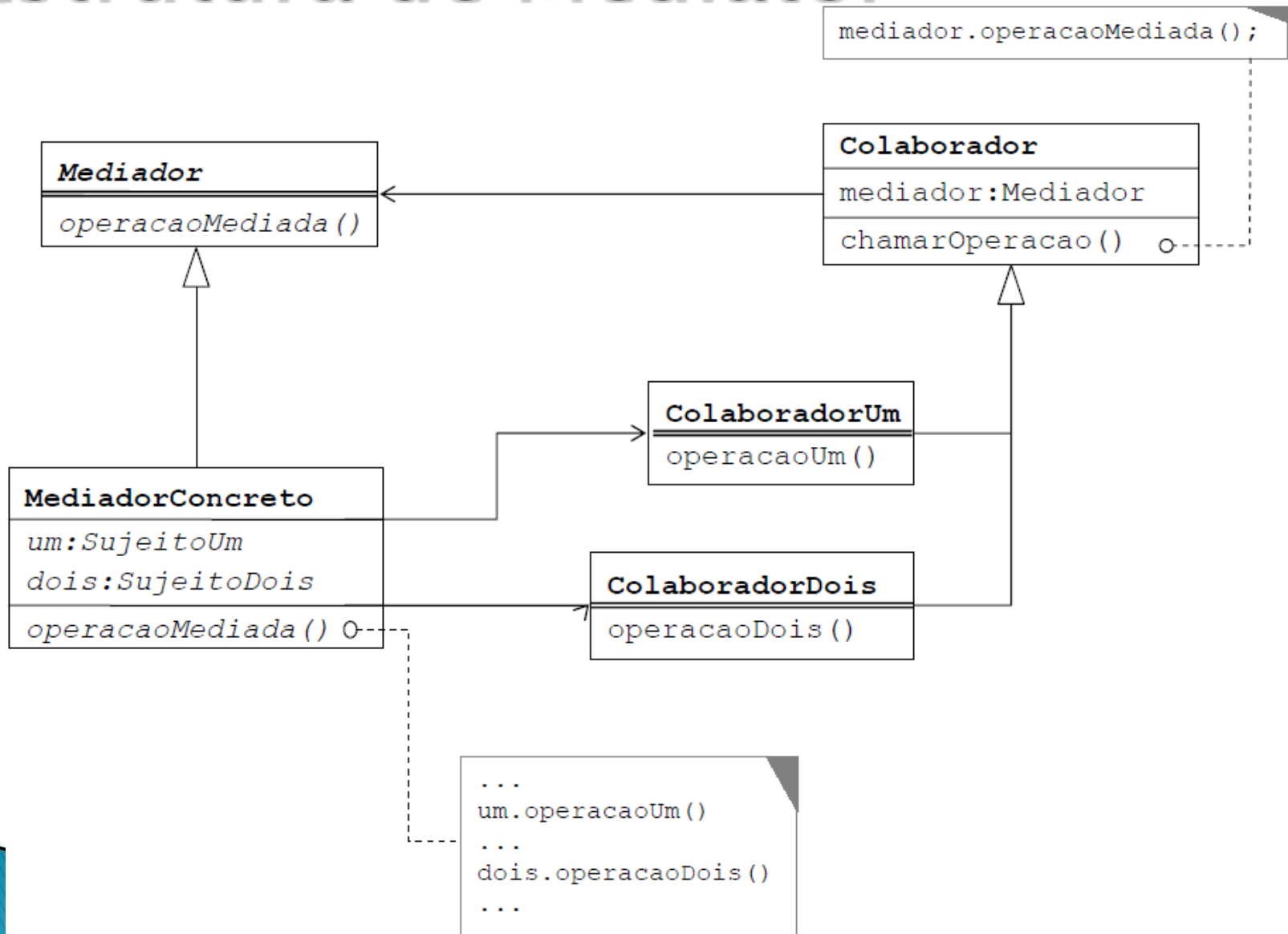
# Solução

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





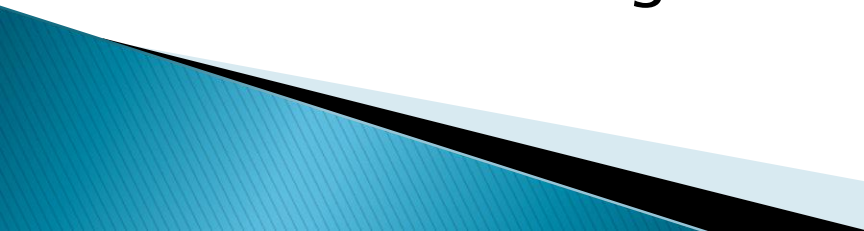
# Estrutura de Mediator



# *Descrição da solução*

- ▶ Um objeto Mediator 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 Mediator é usada pelos colaboradores para iniciar a comunicação e receber notificações
- ▶ O mediador recebe requisições dos remetentes
- ▶ O mediador repassa as requisições aos destinatários

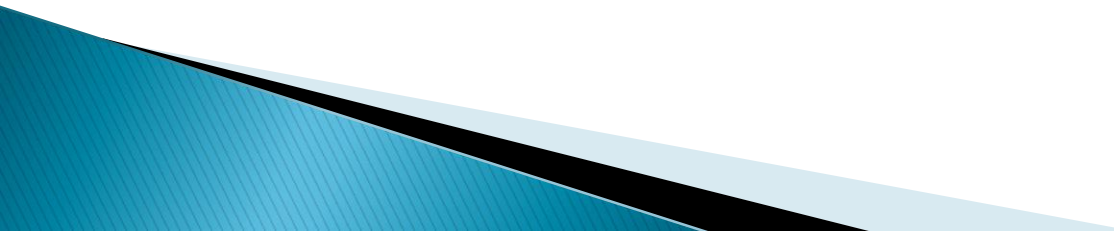
# Aplicabilidade

- ▶ Um conjunto de objetos se comunica de formas bem definidas mas complexas.
    - A interdependência resultante é não-estruturada e difícil de entender.
  - ▶ Reusar um objeto é difícil porque ele se refere e se comunica com muitos outros objetos
  - ▶ Um comportamento que é distribuído entre diversas classes deve ser customizado sem uso de subclassing.
- 

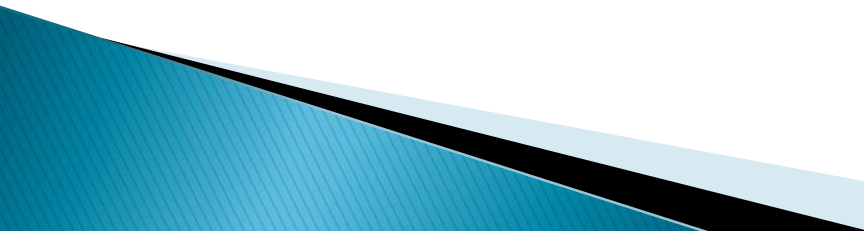
# Participantes

- ▶ **Mediator (DialogDirector)**
  - Define uma interface para comunicar com objetos *Colleague*.
- ▶ **ConcreteMediator (FontDialogDirector)**
  - Implementa comportamento cooperativo ao coordenar os objetos *Colleague*.
  - Conhece e mantém os *Colleagues*.
- ▶ **Classes Colleague (ListBox, EntryField)**
  - Cada classe *Colleague* conhece seu objeto *Mediator*.
  - Cada *Colleague* comunica com o seu *Mediator* quando necessário

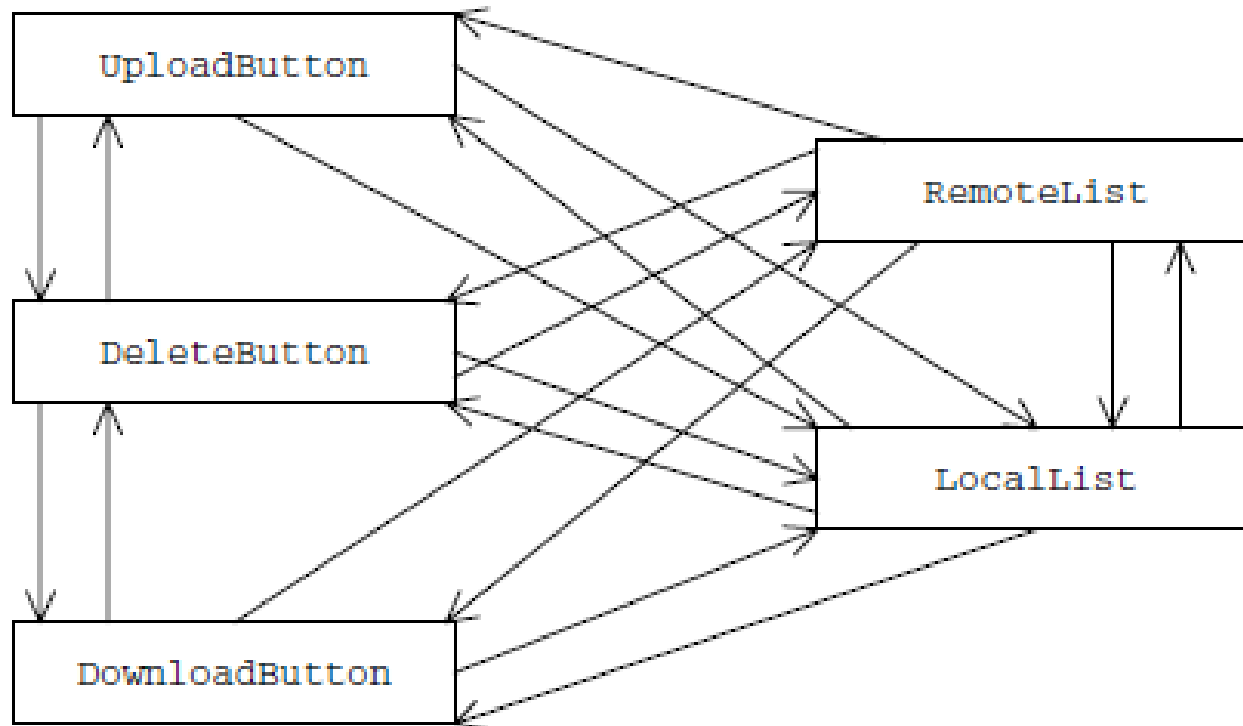
# Colaborações

- ▶ *Colleagues* enviam e recebem requisições do objeto mediador.
  - ▶ O mediador implementa o comportamento cooperativo ao rotear requisições entre os colleague(s) apropriados.
- 

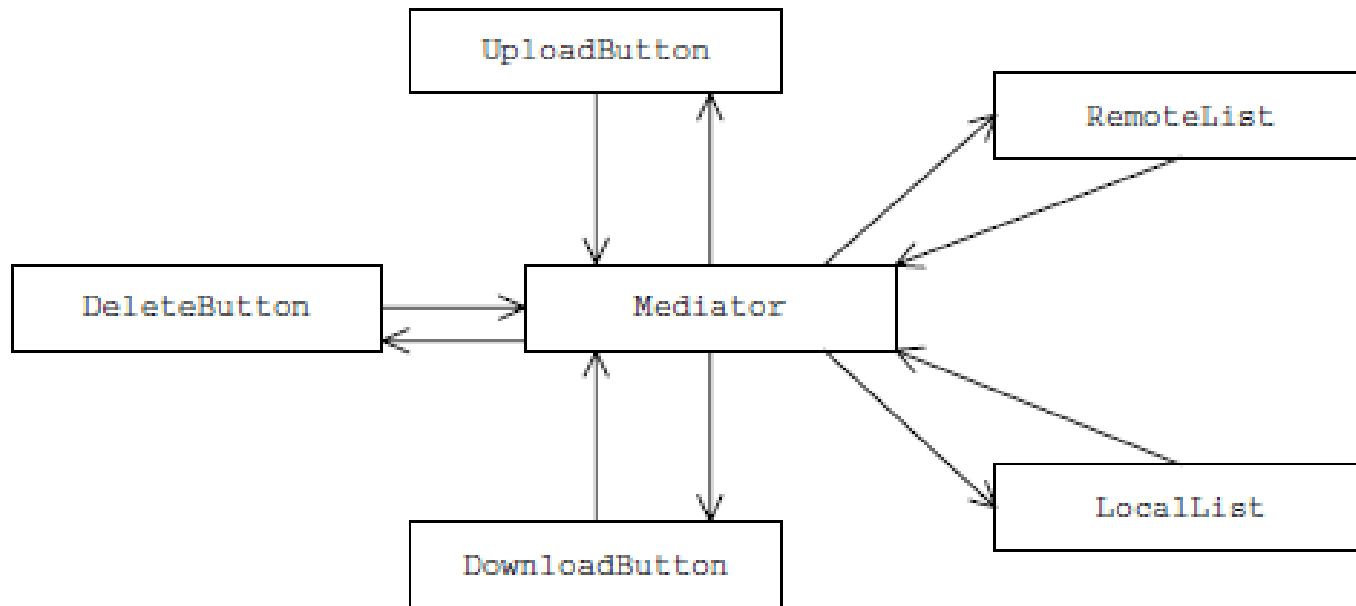
# Conseqüências

- ▶ Limita subclassing
  - ▶ Separa colleagues
  - ▶ Simplifica protocolos de objetos.
    - Substitui interações muitos-para-muitos com interações um-para-muitos
    - Um-para-muitos mais fácil de entender, manter e estender.
  - ▶ Abstrai como objetos cooperam
  - ▶ Centraliza controle
- 

# Exemplo 1 - Sistema de FTP



# Exemplo 1 (cont.)





# Exemplo 1

- ▶ Caixa de diálogo com 1 display e 3 botões: view, book e search.
  - Quando clicar em view, os outros 2 botões devem ser habilitados. O display deve refletir qual botão foi pressionado.
- ▶ Criamos 4 classes: ButtonView, ButtonBook, ButtonSearch e LabelDisplay.
  - As classes implementam a interface Command que permite executar as ações dos botões por uma interface comum.
  - As classes não se conhecem. Só se referem ao mediador.
- ▶ Criamos o mediador que faz referência aos objetos GUI para controlar e coordenar a interação desses objetos.
  - A classe MediatorExample mostra os componentes na tela.
  - A classe implementa a interface ActionListener na qual os botões devem se registrar

## Exemplo 2

```
import java.awt.event.ActionListener;
import javax.swing.JButton;

public class ButtonView extends JButton implements Command {
    Mediator mediator;

    public ButtonView(ActionListener listener, Mediator mediator){
        super("View");
        addActionListener(listener);
        this.mediator = mediator;
        mediator.registerView(this);
    }
    public void execute() {
        mediator.view();
    }
}
```

## Exemplo 2

```
import java.awt.event.ActionListener;
import javax.swing.JButton;

public class ButtonSearch extends JButton implements Command {
    Mediator mediator;

    ButtonSearch(ActionListener listener, Mediator mediator) {
        super("Search");
        addActionListener(listener);
        this.mediator = mediator;
        mediator.registerSearch(this);
    }

    public void execute() {
        mediator.search();
    }
}

public interface Command {
    public void execute();
}
```

## Exemplo 2

```
public class Mediator {
    ButtonView buttonView;
    ButtonBook buttonBook;
    ButtonSearch buttonSearch;
    LabelDisplay labelDisplay;
    public void registerView(ButtonView buttonView) {
        this.buttonView = buttonView;
    }
    public void registerBook(ButtonBook buttonBook) {
        this.buttonBook = buttonBook;
    }
    public void registerSearch(ButtonSearch
buttonSearch) {
        this.buttonSearch = buttonSearch;
    }
    public void registerDisplay(LabelDisplay labelDisplay)
    {
        this.labelDisplay = labelDisplay;
    }
    public void view() {
        buttonView.setEnabled(false);
        buttonBook.setEnabled(true);
        buttonSearch.setEnabled(true);
        labelDisplay.setText("Viewing...");
    }
}
```

```
    public void book() {
        buttonBook.setEnabled(false);
        buttonView.setEnabled(true);
        buttonSearch.setEnabled(true);
        labelDisplay.setText("Booking...");
    }
    public void search() {
        buttonSearch.setEnabled(false);
        buttonBook.setEnabled(true);
        buttonView.setEnabled(true);
        labelDisplay.setText("Searching...");
    }
}
```

## Exemplo 2

class MediatorExample extends JFrame implements ActionListener {

```
    Mediator mediator = new Mediator();
```

```
    public MediatorExample() {
```

```
        Panel p = new JPanel();
```

```
        p.add(new ButtonView(this, mediator));
```

```
        p.add(new ButtonBook(this, mediator));
```

```
        p.add(new ButtonSearch(this, mediator));
```

```
        getContentPane().add(new LabelDisplay(mediator), BorderLayout.NORTH);
```

```
        getContentPane().add(p, BorderLayout.SOUTH);
```

```
        setTitle("Mediator Example");
```

```
        setSize(300, 200);
```

```
        addWindowListener(new WindowAdapter() {
```

```
            public void windowClosing(WindowEvent e) {
```

```
                System.exit(0);
```

```
            }
```

```
        });
```

```
        setVisible(true);    }
```

```
    public void actionPerformed(ActionEvent e) {
```

```
        if (e.getSource() instanceof Command) {
```

```
            Command c = (Command)e.getSource();
```

```
            c.execute();
```

```
        }
```

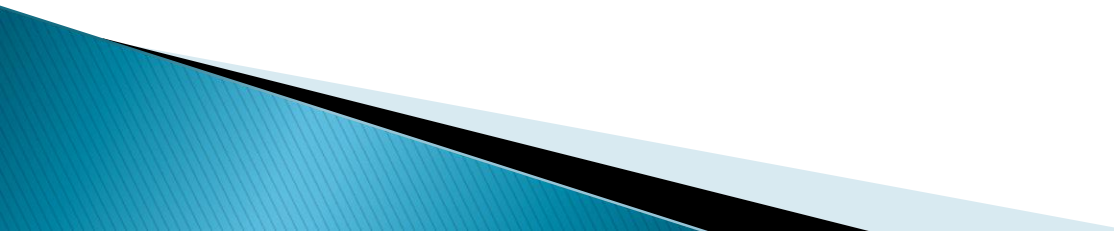
```
    }
```

```
    public static void main(String[] args) {
```

```
        new MediatorExample();
```

```
    }
```

# Related Patterns

- ▶ Facade (208)
  - ▶ Observer (326)
- 

# Mediator vs Façade

**Table 31.1 Mediator versus Façade**

<i>Mediator</i>	<i>Façade</i>
A Mediator is used to abstract the necessary functionality of a group of objects with the aim of simplifying the object interaction.	A Façade is used to abstract the required functionality of a subsystem of components, with the aim of providing a simplified, higher level interface.
All objects interact with each other through the Mediator. The group of objects knows the existence of the Mediator.	Clients use the Façade to interact with subsystem components. The existence of the Façade is not known to the subsystem components.
Because the Mediator and all the objects that are registered with it can communicate with each other, the communication is bidirectional.	Clients can send messages (through the Façade) to the subsystem but not vice versa, making the communication unidirectional.
A Mediator can be assumed to stay in the middle of a group of interacting objects. Using a Mediator allows the implementation of any of the interacting objects to be changed without any impact on the other objects that interact with it only through the Mediator.	A Façade lies in between a client object and the subsystem. Using a Façade allows the implementation of the subsystem to be changed completely without any impact on its clients, provided the clients are not given direct access to the subsystem's classes.
By subclassing the Mediator, the behavior of the object interrelationships can be extended.	By subclassing the Façade, the implementation of the higher level interface can be changed.

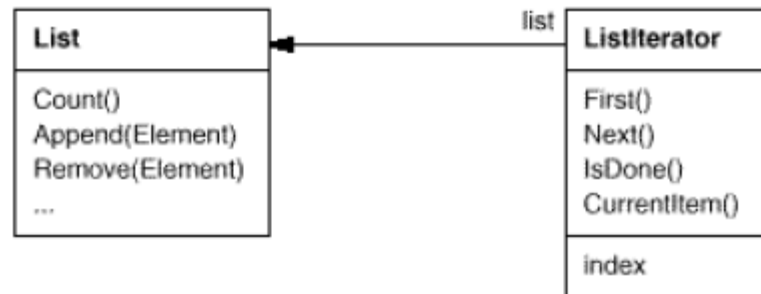
# Iterator

Padrão Comportamental



# Intenção

- ▶ Iterators servem para acessar o conteúdo de um objeto agregado sem expor sua representação interna
  - Collections
- ▶ Oferece uma interface uniforme para atravessar diferentes estruturas agregadas



- ▶ Também conhecido como
  - Cursor

# Problema

Tipo de referência é genérico

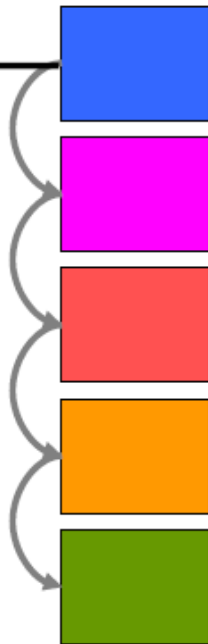


Object o =  
iterator.next()

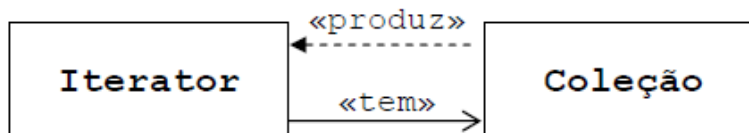
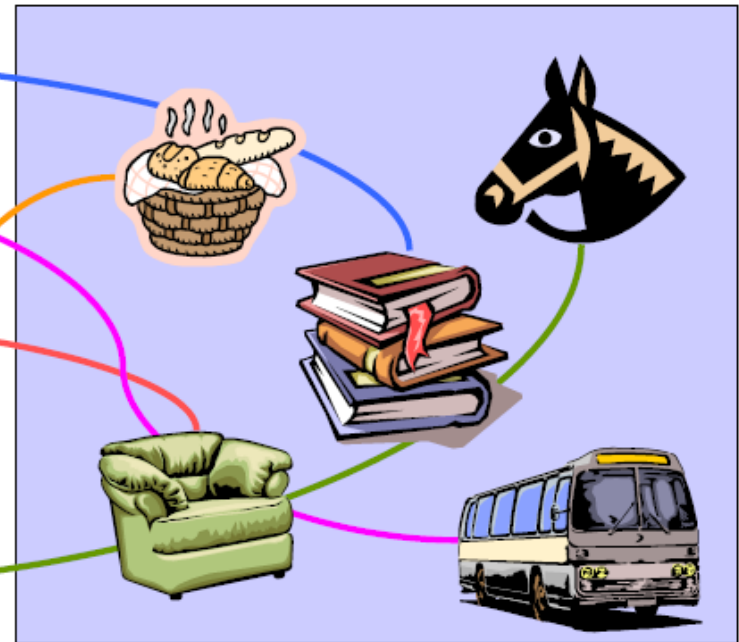


iterator.hasNext() ?

Iterator



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



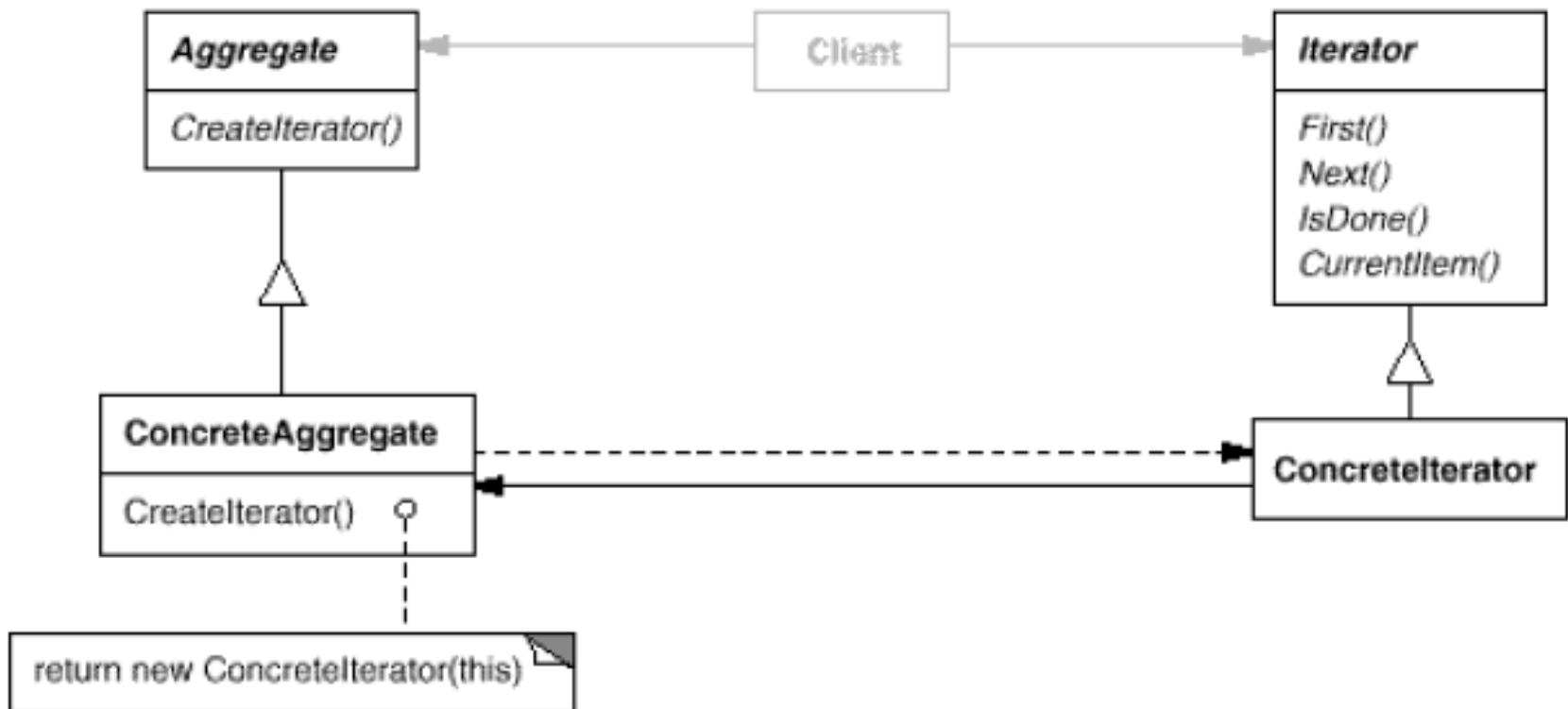
# *Para que serve?*

- ▶ 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`:

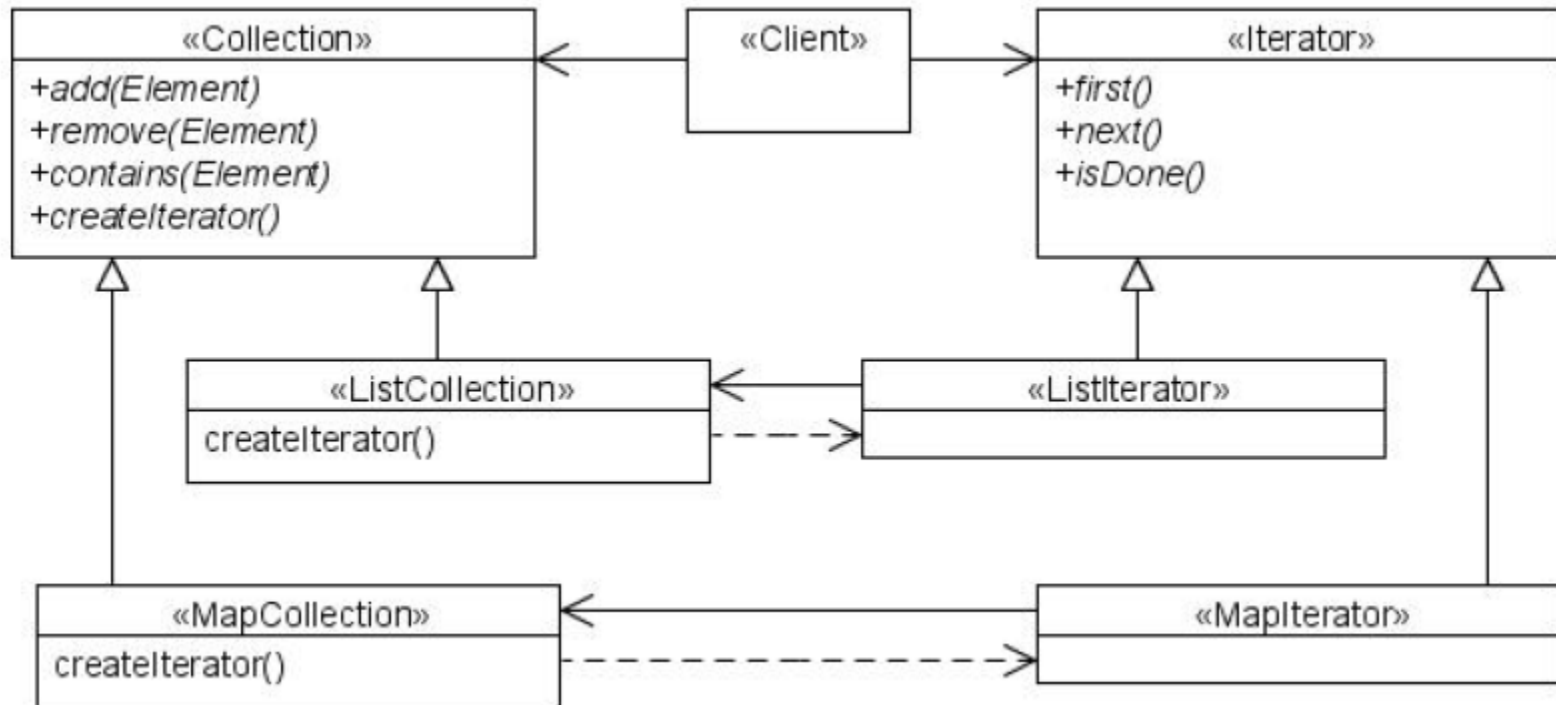
```
package java.util;  
public interface Iterator<E> {  
    boolean hasNext();  
    Object next();  
    void remove();  
}
```

- ▶ `iterator()` é um exemplo do padrão `FactoryMethod`

# Estrutura



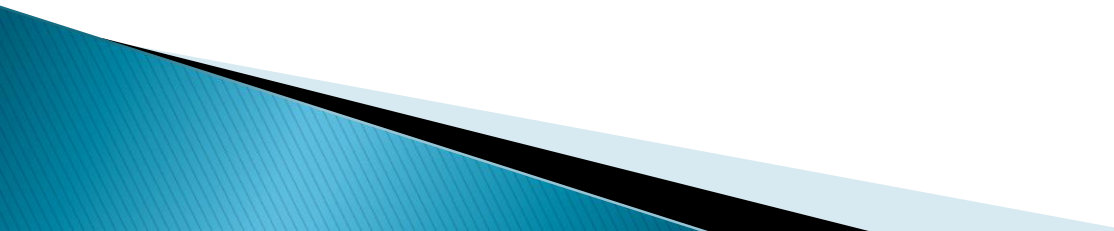
# Estrutura – Exemplo



# Aplicabilidade

- ▶ Usado para acessar os elementos de um *agregado* seqüencialmente
- ▶ Coleções Java como ArrayList e HashMap tem sua implementação do padrão Iterator

# Benefícios

- ▶ O mesmo Iterator pode ser usado por diferentes agregados.
  - ▶ Permite atravessar o agregado de formas diferentes dependendo da necessidade.
  - ▶ Encapsula a estrutura interna de como a interação ocorre.
  - ▶ Não precisamos ‘inchar’ a classe com operações para diferentes formas de atravessar o agregado.
- 

# Participantes

## ▶ Iterator

- define uma interface para acessar e transversar elementos.

## ▶ Concreteliterator

- Implementa a interface Iterator.
- Mantém a posição atual de navegação no agregado.

## ▶ Aggregate

- Define uma interface para criar um objeto Iterator.

## ▶ ConcreteAggregate

- implementa a interface de criação do Iterator para retornar uma instância do Concreteliterator.



# Exemplo 1

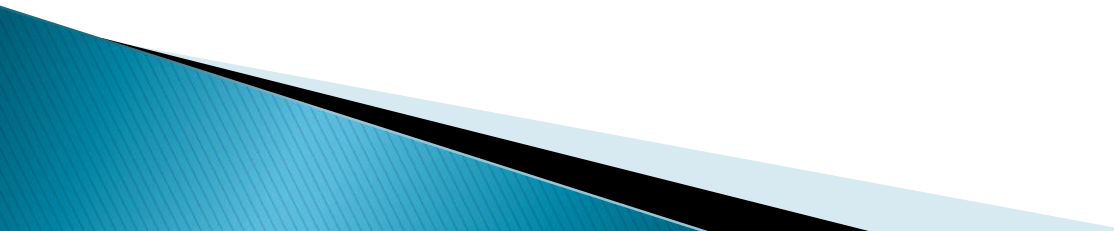
```
import java.util.*;
public class BitSetIterator implements Iterator<Boolean> {
    private final BitSet bitset;
    private int index;
    public BitSetIterator(BitSet bitset) {
        this.bitset = bitset;
    }
    public boolean hasNext() {
        return index < bitset.length();
    }
    public Boolean next() {
        if (index >= bitset.length()) {
            throw new NoSuchElementException();
        }
        boolean b = bitset.get(index++);
        return new Boolean(b);
    }
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

## Exemplo 1

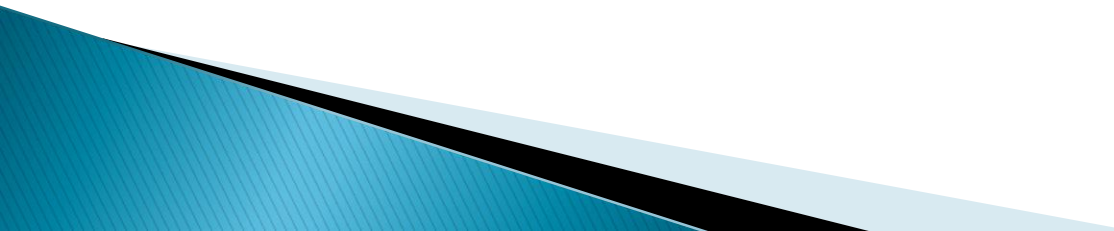
```
public class TestClientBitSet {
    public static void main(String[] args) {
        // create BitSet and set some bits
        BitSet bitset = new BitSet();
        bitset.set(1);
        bitset.set(19);
        bitset.set(20);
        bitset.set(47);
        BitSetIterator iter = new BitSetIterator(bitset);
        while (iter.hasNext()) {
            Boolean b = iter.next();
            String tf = (b.booleanValue() ? "T" : "F");
            System.out.print(tf);
        }
        System.out.println();
    }
}
```

# Exemplo 2

```
import java.util.*;
public class TestClientIterator {
    public static void main(String[] args) {
        ArrayList<Object> al = new ArrayList<Object>();
        al.add(new Integer(42));
        al.add(new String("test"));
        al.add(new Double("-12.34"));
        for(Iterator<Object> iter=al.iterator();
            iter.hasNext();)
            System.out.println( iter.next() );
    }
}
```



# Padrões relacionados

- ▶ Composite (183)
  - ▶ Factory Method (121)
  - ▶ Memento (316)
- 

# Strategy

Padrão Comportamental

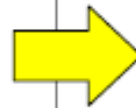
# Intenção

- ▶ Definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis.
- ▶ Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam.

# Intenção

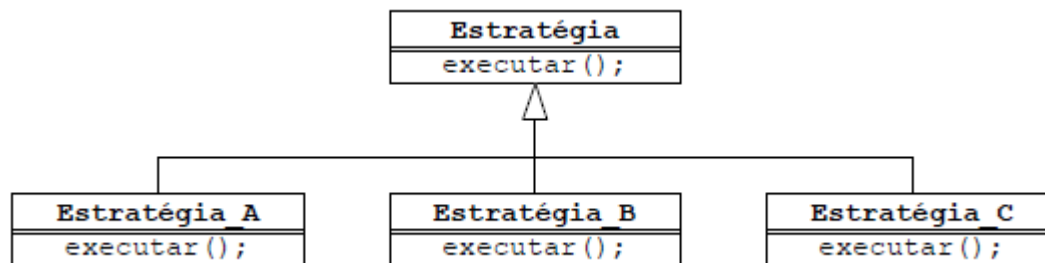
*Várias estratégias, escolhidas de acordo com opções ou condições*

```
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();
```

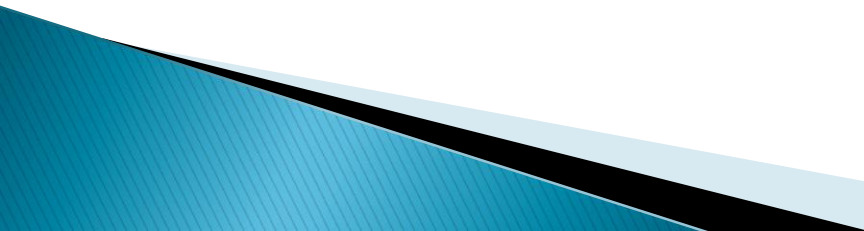


# Conhecido como

- ▶ Também conhecido como:
  - Policy



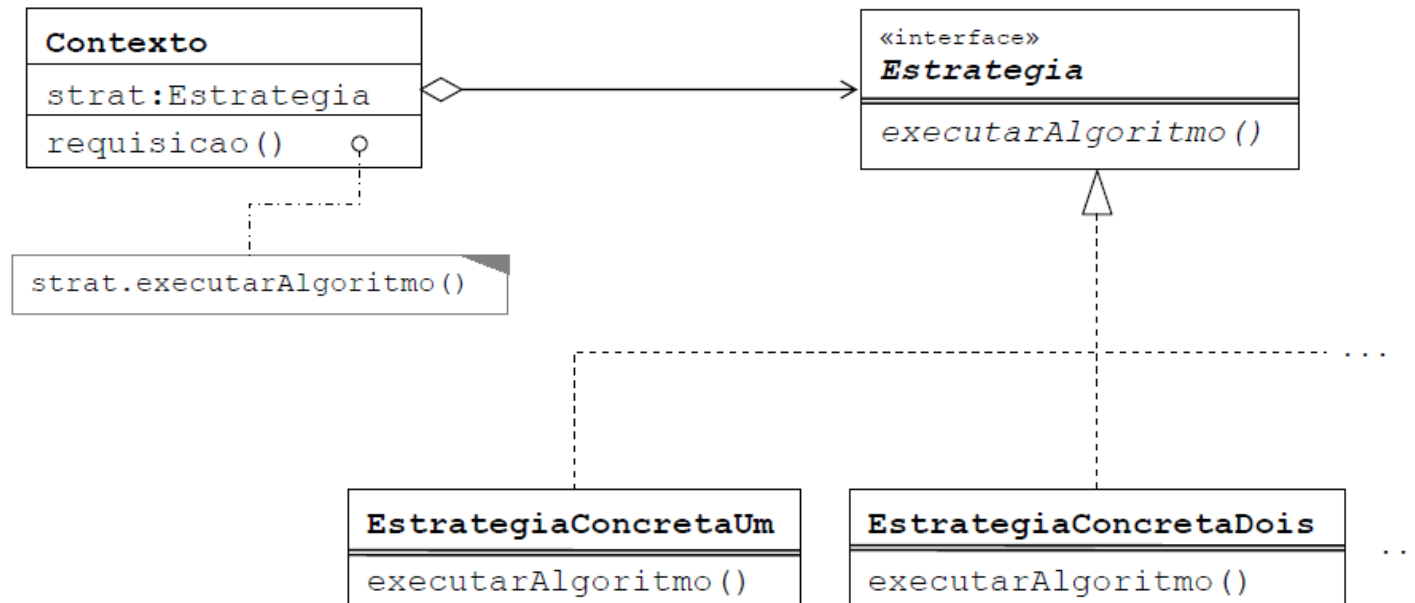
# *Aplicabilidade*

- ▶ 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
- 

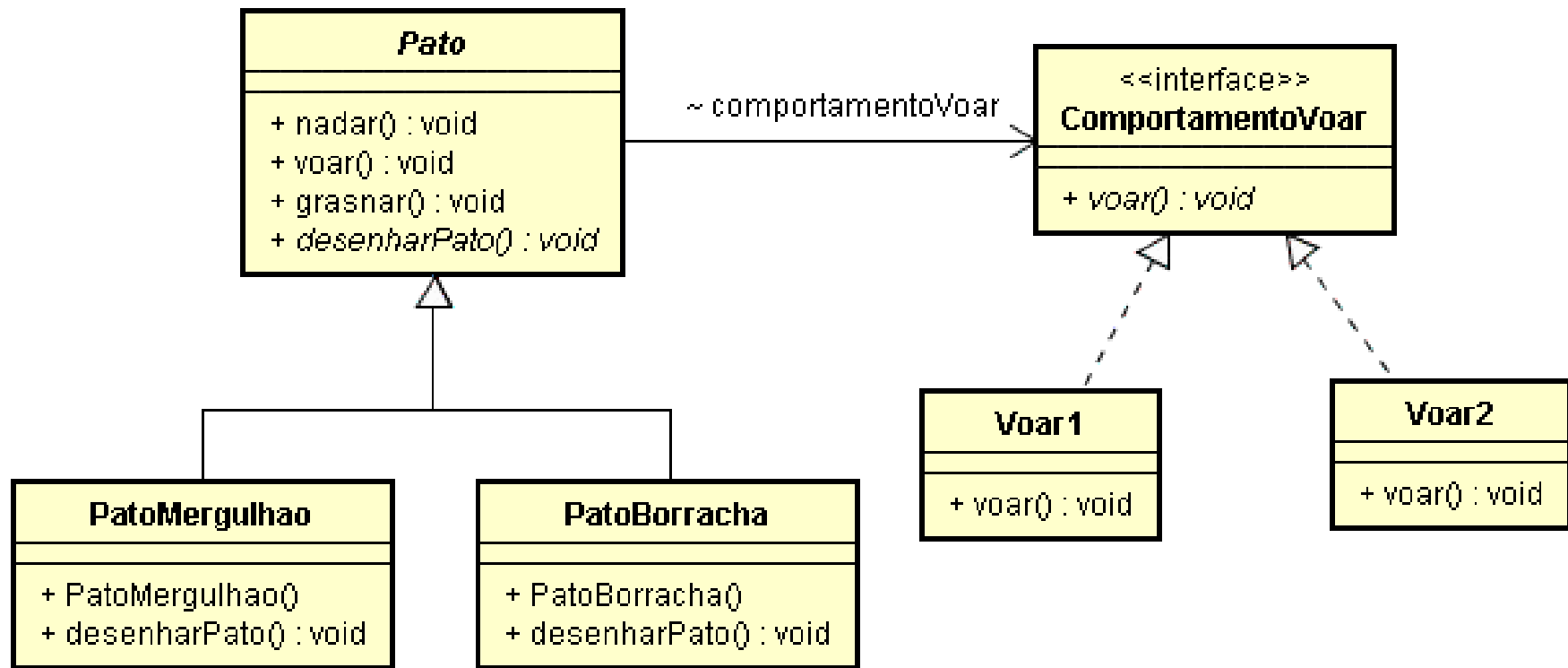
# Como funciona?

- ▶ Manter a implementação de cada algoritmo em uma classe separada
- ▶ Cada algoritmo é chamado de *strategy*.
- ▶ O objeto que usa o *strategy* é chamado de context.
- ▶ Mudar o comportamento de um objeto *context*:
  - Mudar o objeto *strategy* para o qual implementa o algoritmo desejado
- ▶ Todos os objetos *strategy* deve oferecer a mesma interface.
  - Implementar uma interface em comum ou uma subclasse da classe abstrata comum que declare a interface necessária

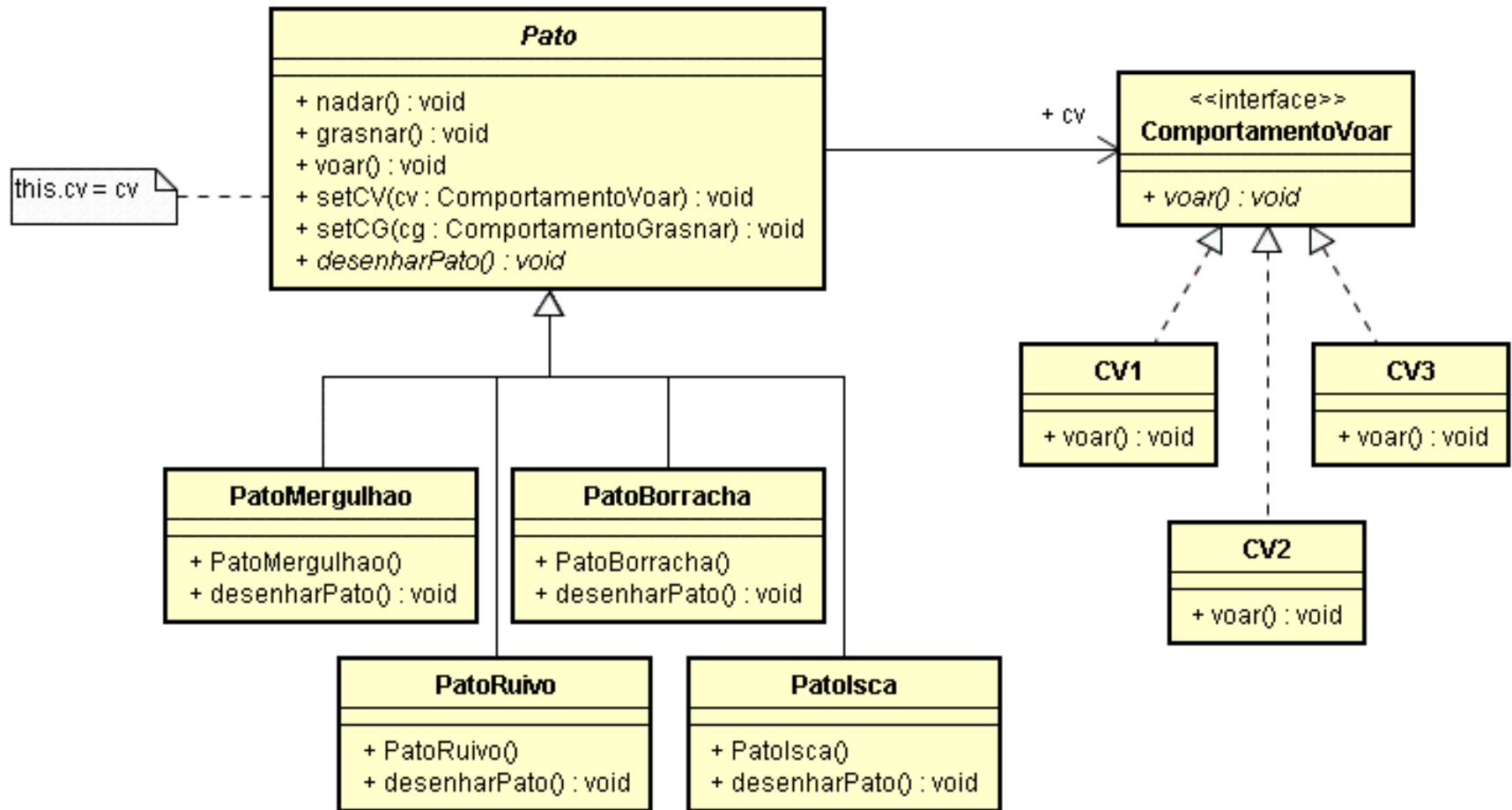
# Estrutura



# Exemplo de estrutura



# Exemplo, Trocando o comportamento em tempo de execução



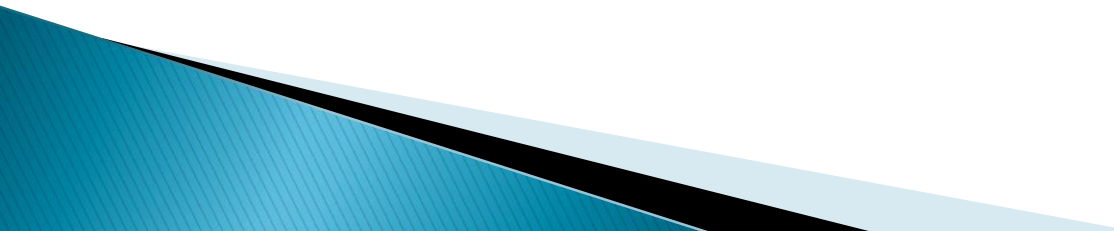
# Participantes

- ▶ Strategy
  - Define uma interface comum para todos os algoritmos suportados.
- ▶ ConcreteStrategy
  - Implementa o algoritmo usando a interface de Strategy.
- ▶ Context
  - É configurado com um objeto ConcreteStrategy

# Colaborações

- ▶ Strategy e Context interagem para implementar o algoritmo escolhido.
- ▶ Um Context repassa solicitações dos seus clientes para sua estratégia.

# Benefícios

- ▶ Reduz o uso de declarações condicionais em um cliente.
  - ▶ Esconde dados específicos do algoritmo do cliente.
  - ▶ Fornece uma alternativa a subclassing.
- 



# Conseqüências

- ▶ Clientes devem saber das diferentes estratégias
- ▶ Aumenta o número de objetos em uma aplicação.

# Exemplo 1

- ▶ Sistema de cobrança precisa imprimir datas.
  - Mas as datas variam dependendo do país: EUA, Brasil.
  - Dependendo da extensão: só número, mês escrito, etc.
  - Exemplos: October 2, 2005 2 Oct 2005 10/2/05 02/10/05  
20051002 2005275
- ▶ Dá para utilizar Strategy?

## Exemplo 2

```
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();
    }
}
```

# Exemplo 3

```
public interface SortInterface {
    public void sort(double[] list); }
public class QuickSort implements SortInterface {
    public void sort(double[] u) {
        sort(u, 0, u.length - 1);    }
    private void sort(double[] a, int left, int right) {
        if (right <= left) return;
        int i = part(a, left, right);
        sort(a, left, i-1);
        sort(a, i+1, right);
    }
    private int part(double[] a, int left, int right) {
        int i = left;
        int j = right;
        while (true) {
            while (a[i]< a[right])
                i++;
            while (smaller(a[right], a[--j]))
                if (j == left) break;
                if (i >= j) break;
                swap(a, i, j);
        }
        swap(a, i, right);
        return i;
    }
}
```

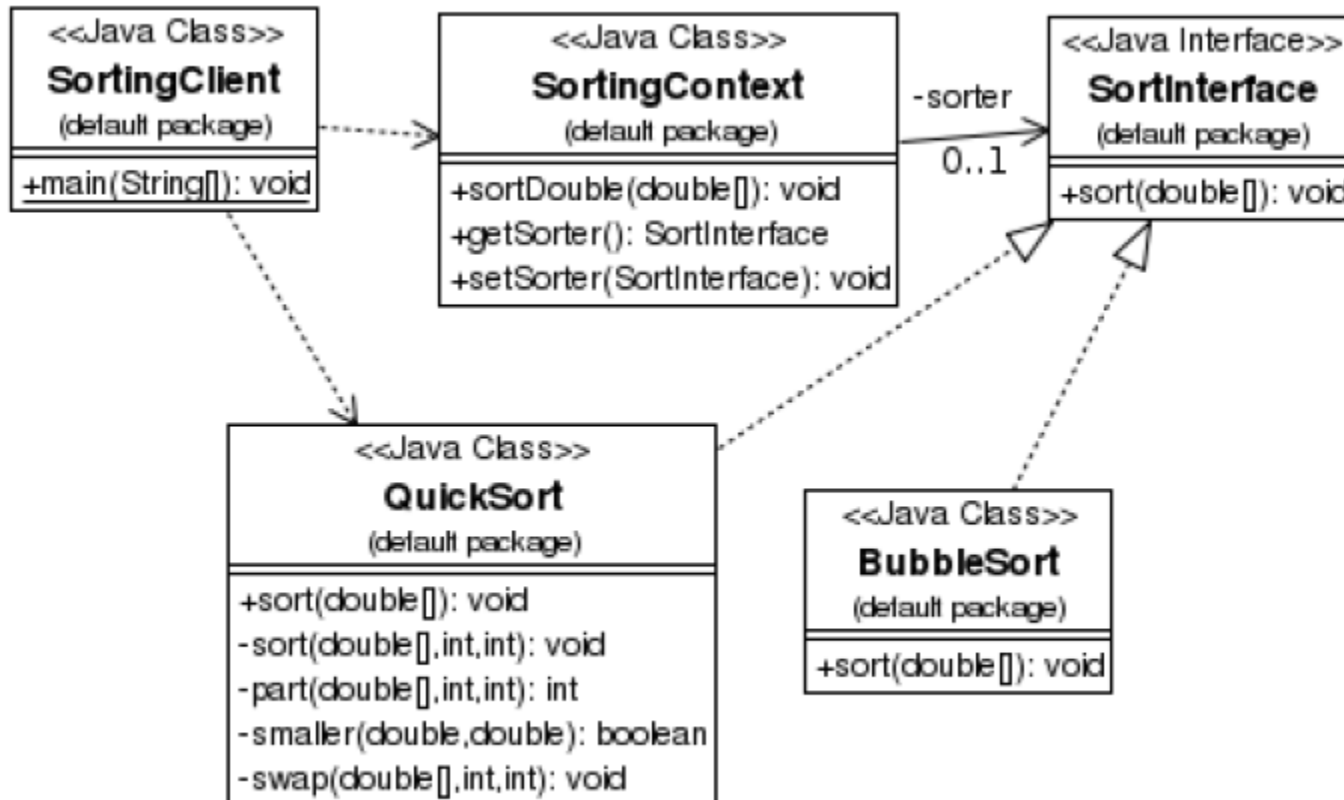
```
        private boolean smaller(double x, double y) {
            return (x < y);
        }
        private void swap(double[] a, int i, int j) {
            double swap = a[i]; a[i] = a[j]; a[j] = swap;
        }
    }
    public class BubbleSort implements SortInterface {
        public void sort(double[] list) {
            //Bubblesort algorithm here
        }
    }
}
```

# Exemplo 3

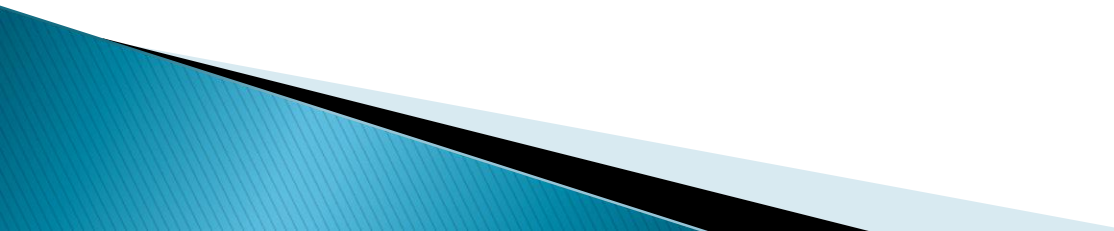
```
public class SortingContext {
    private SortInterface sorter = null;
    public void sortDouble(double[] list) {
        sorter.sort(list);
    }
    public SortInterface getSorter() {
        return sorter;
    }
    public void setSorter(SortInterface sorter) {
        this.sorter = sorter;
    }
}
```

```
public class SortingClient {
    public static void main(String[] args) {
        double[] list = {1,2.4,7.9,3.2,1.2,0.2,10.2,22.5,19.6,14,12,16,17};
        SortingContext context = new SortingContext();
        context.setSorter(new QuickSort());
        context.sortDouble(list);
        for(int i =0; i< list.length; i++) {
            System.out.println(list[i]);
        }
    }
}
```

# Exemplo 3



# Exercício 1

- ▶ Um sistema de atendimento em um banco manipula consultas dos clientes existentes e potenciais usando uma aplicação de chat online.
  - ▶ Nos horário de pico, cada atendente deve poder trabalhar com mais de um cliente simultaneamente.
  - ▶ Qual padrão de projeto é adequado neste caso?
  - ▶ Desenhe um exemplo de estrutura do mecanismo para este caso.
- 

# Exercício 2

- ▶ Identifique que padrão pode ser usado para construir um applet que aparece na tela de formas diferentes de acordo com o tamanho da tela do browser.
  - ▶ Desenhe um exemplo de estrutura do mecanismo para este caso.
- 