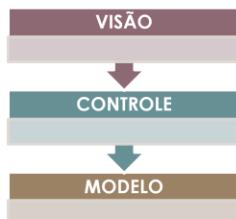
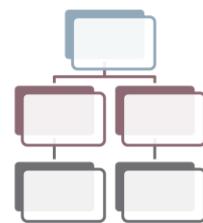


# Arquitetura e Desenho de Software

## AULA 15



Profa. Milene Serrano



# Agenda



Considerações Iniciais

Padrões GoFs:  
\* Categoria: Comportamentais

Considerações Finais

## Considerações Iniciais



## Padrões GoF

Uma solução consolidada para um problema recorrente no desenvolvimento e manutenção de software orientado a objetos.

- **Referência Relevante:** Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Design Patterns: Elements os Reusable Object-Oriented Software. Addison-Wesley, 1995.
- **Gang of Four - GoF**

GoF  
Comportamental



## Padrões de Projeto – GoFs Comportamentais



Padrões voltados para alterações no nível do comportamento dos objetos...



Auxiliam quando é necessário, por exemplo, usar vários algoritmos diferentes, cada qual mais apropriado para um determinado contexto...



Permitem, nesse caso, usar mecanismos/recursos para facilitar tanto a incorporação de novos algoritmos para novos contextos quanto a seleção de qual algoritmo usar dado um contexto.

## Padrões de Projeto – GoFs Comportamentais

### Principais Gofs Comportamentais

#### Command

Controlar as chamadas a um determinado componente, modelando cada requisição como um objeto. Permitir que as operações possam ser desfeitas ou registradas.

#### Iterator

Fornecer um modo eficiente para percorrer sequencialmente os elementos de uma coleção, sem expor a estrutura interna da coleção.

#### Mediator

Diminuir a quantidade de "ligações" entre objetos introduzindo um mediador, o qual realizará toda a comunicação.

Ao final dessa aula, em **Complementar**, têm detalhes sobre vários desses padrões...

## Padrões de Projeto – GoFs Comportamentais

### Principais Gofs Comportamentais

Observer

State

Strategy

Definir um mecanismo eficiente para reagir às alterações realizadas em determinados objetos.

Alterar o comportamento de um determinado objeto de acordo com o estado no qual ele se encontra.

Permitir, de maneira simples, a variação dos algoritmos utilizados na resolução de um determinado problema.

Ao final dessa aula, em **Complementar**, têm detalhes sobre vários desses padrões...

## Padrões de Projeto – GoFs Comportamentais

### Principais Gofs Comportamentais

#### Template Method

Definir a ordem na qual determinados passos devem ser realizados na resolução de um problema e permitir que esses passos possam ser realizados de formas diferentes de acordo com a situação.

#### Visitor

Permitir atualizações específicas em uma coleção de objetos de acordo com o tipo particular de cada objeto atualizado.

#### Memento

Permite armazenar o estado interno de um objeto em um determinado momento, para que seja possível retorná-lo a este estado, sem que isso cause problemas com o encapsulamento. Uma classe é responsável por salvar o estado do objeto desejado; enquanto que outra classe fica responsável por armazenar todas essas cópias (mementos).

#### Chain Of Responsibility

Evitar a dependência entre um objeto receptor e um objeto solicitante. A base mantém um ponteiro como "próximo". Cada classe derivada implementa sua própria contribuição para manusear o pedido (request).

Ao final dessa aula, em **Complementar**, têm detalhes sobre vários desses padrões... O Memento, por exemplo, é fundamental em nossas vidas! : )

## GoF Comportamental Strategy

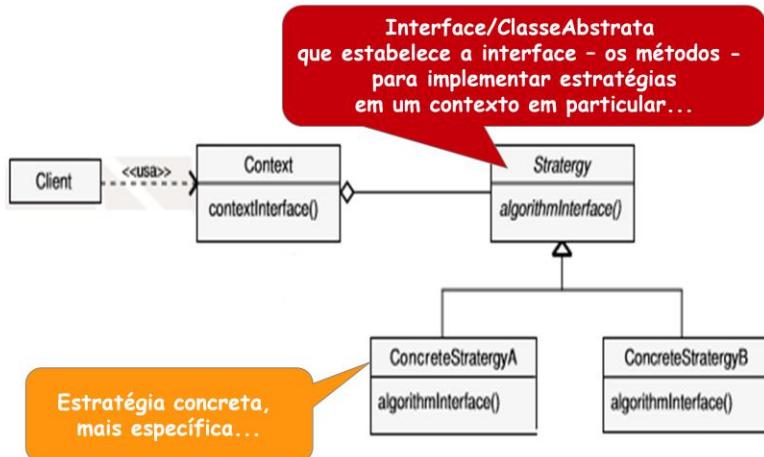


## GoF Comportamental – Strategy

### Objetivo

- Permitir de maneira simples a variação dos algoritmos utilizados na resolução de um determinado problema.

## GoF Comportamental – Strategy



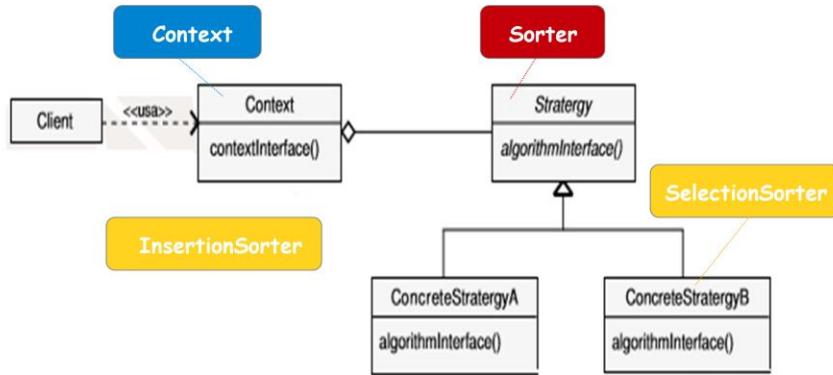
Modelagem Genérica

## GoF Comportamental – Strategy

### Participantes

- **Strategy** (ex. *Sorter*) - interface para padronizar as diferentes estratégias de um algoritmo.
- **ConcreteStrategy** (ex. *InsertionSorter*, *SelectionSorter*)  
- implementação particular de um Strategy.
- **Context** - mantém uma referência para um objeto Strategy e pode permitir que esse acesse os seus dados.

## GoF Comportamental – Strategy



Modelagem Aplicada/Instaciada

## GoF Comportamental – Strategy

MOTIVAÇÃO

Quando desejamos realizar um pagamento, precisamos decidir qual forma de pagamento será adequada (ex. via cartão de crédito ou via boleto bancário). Portanto, podem haver diversas formas de pagamento.

Existem alguns recursos que podem ajudar nessa tarefa, como é o caso dos carrinhos de compra, por exemplo.

Normalmente, esses recursos permitem que o usuário escolha a forma de pagamento (ex cartão de crédito, boleto bancário, PayPal, dentre outros) que deseja realizar. Essa escolha afeta os passos para se concretizar o pagamento.

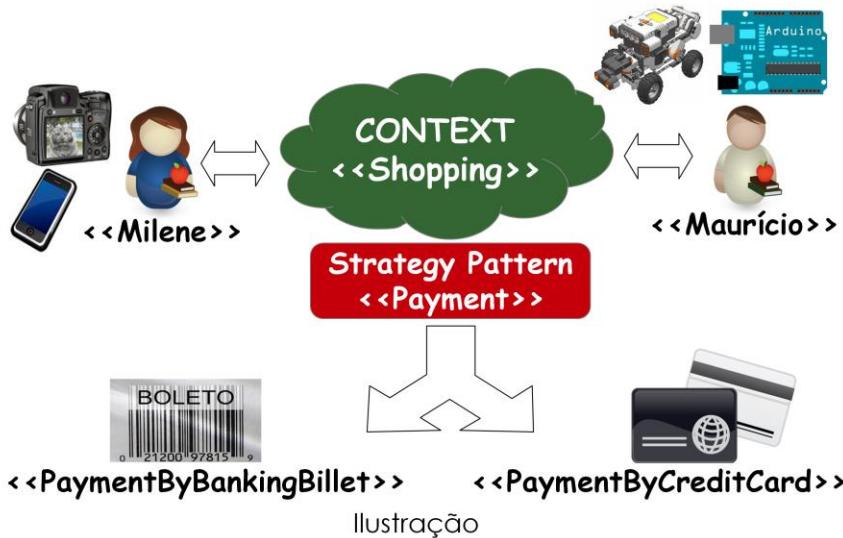
# GoF Comportamental – Strategy

MOTIVAÇÃO

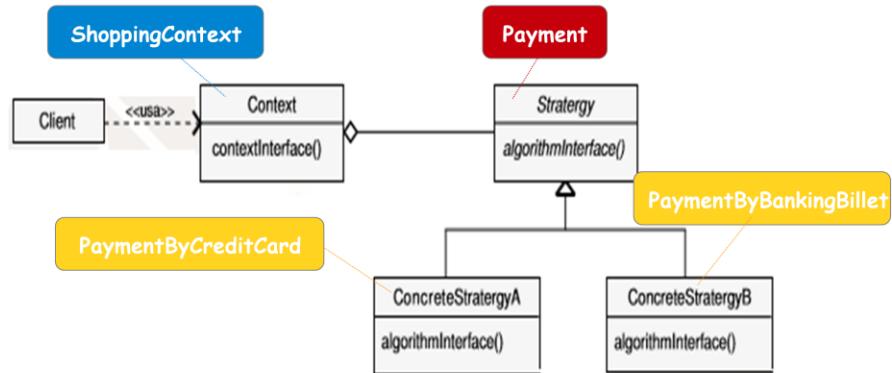
O padrão *Strategy* propõe uma solução que pode ser adotada nesse cenário. Fundamentalmente, deve-se possibilitar a variação facilitada do algoritmo a ser utilizado na resolução do problema. No contexto, diferentes algoritmos são usados para se proceder o pagamento de uma compra, dependendo do modo como o usuário deseja pagar essa compra.

**Strategy Pattern**  
**(Padrão Estratégia)**

## GoF Comportamental – Strategy



## GoF Comportamental – Strategy



Modelagem Aplicada/Instaciada

# GoF Comportamental – Strategy

SOLUÇÃO

Podemos empregar o padrão de projeto GoF, comportamental, *Strategy*.

Como seria a implementação em Java?



## GoF Comportamental – Strategy

Payment.java

```
package strategy;

public interface Payment {

    public void pay(double amount);

}
```

Se optarem por programar  
algum comportamento *default*  
para algum método, podem  
implementar como classe abstrata, ok?

Possível Implementação

## GoF Comportamental – Strategy

```
① *PaymentByCreditCard.java ✘
package strategy;

import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Locale;

public class PaymentByCreditCard implements Payment {

    private String creditCardOwnerName;
    private String creditCardNumber;
    private int creditCardCheckerCode;
    private String creditCardExpiryDate;

    SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy");

    public PaymentByCreditCard(String creditCardOwnerName, String creditCardNumber,
                               int creditCardCheckerCode, Calendar creditCardExpiryDate) {
        this.creditCardOwnerName = creditCardOwnerName;
        this.creditCardNumber = creditCardNumber;
        this.creditCardCheckerCode = creditCardCheckerCode;
        this.creditCardExpiryDate = simpleDateFormat.format(creditCardExpiryDate.getTime());
    }
    ...
}
```

Tudo específico  
para pagamentos  
com cartões de crédito!

Possível Implementação

# GoF Comportamental – Strategy

```
*PaymentByCreditCard.java ✘
...
@Overrider
public void pay(double amount) {
    //Payment Strategy by using a Credit Card
    //Here is the place to implements the details to deal with a payment by using a Credit Card
    double formattedAmount = Double.valueOf(String.format(Locale.US, "%.2f", amount));
    System.out.println(formattedAmount + " paid with credit/debit card");
    System.out.println("The credit card's details are:");
    System.out.println("Credit Card's Owner Name: " + this.creditCardOwnerName);
    System.out.println("Credit Card's Number: " + this.creditCardNumber);
    System.out.println("Credit Card's Checker Code: " + this.creditCardCheckerCode);
    System.out.println("Credit Card's Expiry Date: " + this.creditCardExpiryDate);
}
```

Possível Implementação

# GoF Comportamental – Strategy

```
*PaymentByBankingBillet.java
package strategy;

import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Locale;

public class PaymentByBankingBillet implements Payment {

    private String bankingBilletPayerName;
    private String bankingBilletBeneficiaryName;
    private String bankingBilletNumber;
    private String bankingBilletExpiryDate;

    SimpleDateFormat simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy");

    public PaymentByBankingBillet(String bankingBilletPayerName, String bankingBilletBeneficiaryName,
        String bankingBilletNumber, Calendar bankingBilletExpiryDate) {
        this.bankingBilletPayerName = bankingBilletPayerName;
        this.bankingBilletBeneficiaryName = bankingBilletBeneficiaryName;
        this.bankingBilletNumber = bankingBilletNumber;
        this.bankingBilletExpiryDate = simpleDateFormat.format(bankingBilletExpiryDate.getTime());
    }
    ...
}
```

Tudo específico  
para pagamentos  
com boleto bancário!

Possível Implementação

# GoF Comportamental – Strategy

```
*PaymentByBankingBillet.java ✘
...
@Override
public void pay(double amount) {
    //Payment Strategy by using a Banking Billet
    //Here is the place to implements the details to deal with a payment by using a Banking Billet
    double formattedAmount = Double.valueOf(String.format(Locale.US, "%.2f", amount));
    System.out.println(formattedAmount + " paid with banking billet");
    System.out.println("The banking billet's details are:");
    System.out.println("Banking Billet's Payer Name: " + this.bankingBilletPayerName);
    System.out.println("Banking Billet's Beneficiary Name: " + this.bankingBilletBeneficiaryName);
    System.out.println("Banking Billet's Number: " + this.bankingBilletNumber);
    System.out.println("Banking Billet's Expiry Date: " + this.bankingBilletExpiryDate);
}
```

Possível Implementação

## GoF Comportamental – Strategy

Uma referência ao pagamento!

Uma lista de itens e suas operações básicas de adição e remoção. Apenas para embasar o contexto de forma mais adequada, ok?

```
*ShoppingContext.java
package context;

import java.util.ArrayList;
import java.util.List;
import strategy.Payment;

public class ShoppingContext{
    List<Item> items;
    Payment payment;

    public ShoppingContext(){
        this.items = new ArrayList<Item>();
    }

    public void addItem(Item item){
        this.items.add(item);
    }

    public void removeItem(Item item){
        this.items.remove(item);
    }
    ...
}
```

Possível Implementação

## GoF Comportamental – Strategy

```
ShoppingContext.java
```

```
...
    public double calculateTotalPrice(){
        double sum = 0;
        for(Item item : items){
            sum += item.getItemPrice();
        }
        return sum;
    }

    //Payment processing
    public void pay(Payment payment){
        double amount = this.calculateTotalPrice();
        this.payment = payment;
        this.payment.pay(amount);
    }

    //Other important typical processing for shopping context...
}
```

Mais um método para embasar o contexto, permitindo o cálculo total da compra com base nos preços dos itens...

Chamando o método pay implementado nas diferentes estratégias...

Possível Implementação

## GoF Comportamental – Strategy

```
StrategyTest.java ✘
package test;

import java.util.Calendar;

import context.Item;
import context.ShoppingContext;
import strategy.Payment;
import strategy.PaymentByCreditCard;
import strategy.PaymentByBankingBillet;

public class StrategyTest {

    public static void main(String[] args) {
        ShoppingContext shoppingContext;

        Item lego = new Item("1234", 100.56);
        Item arduino = new Item("5678", 34.00);

        Item camera = new Item("1234", 200.00);
        Item cellphone = new Item("5678", 180.55);

        Payment payment;
        Calendar date = Calendar.getInstance();
        ...
    }
}
```

Apenas testando  
o uso do *Strategy*...

Existem várias possibilidades  
de implementação  
nesse ponto...

Possível Implementação

## GoF Comportamental – Strategy

```
StrategyTest.java ...
...
System.out.println("\n Paying by Credit Card:");
shoppingContext = new ShoppingContext();
shoppingContext.addItem(lego);
shoppingContext.addItem(arduino);
payment = new PaymentByCreditCard("Mauricio Serrano", "4073.0200.0000.0002", 111, date);
shoppingContext.pay(payment);

System.out.println("\n Paying by Banking Billet:");
shoppingContext = new ShoppingContext();
shoppingContext.addItem(camera);
shoppingContext.addItem(cellphone);
payment = new PaymentByBankingBillet("Milene Serrano", "Americanas.com",
        "40730.02000 00000.000234 12345.678900 3 33390000038055", date);
shoppingContext.pay(payment);
}
```

Via Cartão de Crédito

Via Boleto Bancário

Possível Implementação

# GoF Comportamental – Strategy

```
Problems @ Javadoc Declaration Console 
<terminated> StrategyTest (2) [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe

Paying by Credit Card:
134.56 paid with credit/debit card
The credit card's details are:
Credit Card's Owner Name: Mauricio Serrano
Credit Card's Number: 4073.0200.0000.0002
Credit Card's Checker Code: 111
Credit Card's Expiry Date: 07/05/2014

Via Cartão de Crédito

Paying by Banking Billet:
380.55 paid with banking billet
The banking billet's details are:
Banking Billet's Payer Name: Milene Serrano
Banking Billet's Beneficiary Name: Americanas.com
Banking Billet's Number: 40730.02000 00000.000234 12345.678900 3 33390000038055
Banking Billet's Expiry Date: 07/05/2014

Via Boleto Bancário
```

Possível Implementação

## GoF Comportamental *Template Method*

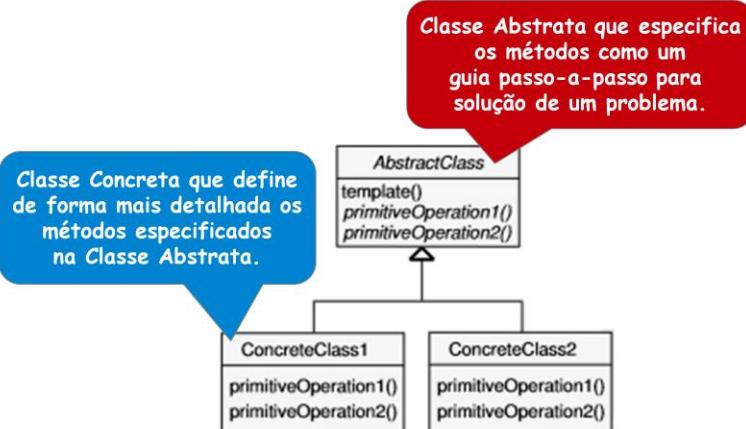


## GoF Comportamental – Template Method

### Objetivo

- Definir a ordem na qual determinados passos devem ser realizados na resolução de um problema e permitir que esses passos possam ser realizados de formas diferentes de acordo com a situação.

## GoF Comportamental – Template Method



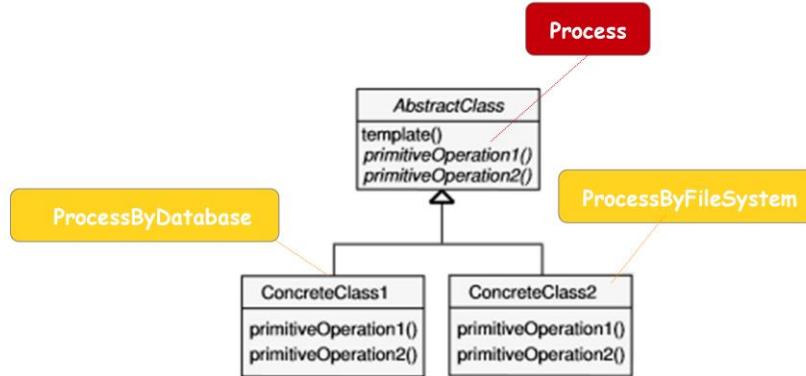
Modelagem Genérica

## GoF Comportamental – Template Method

### Participantes

- **AbstractClass** (ex. *Process*) - Classe abstrata que define o *template method*, com a ordem de execução das operações. As operações primárias permanecem como métodos abstratos, sendo concretizadas nas *ConcreteClasses*.
- **ConcreteClass** (ex. *ProcessByDatabase*, *ProcessByFileSystem*) - Classes concretas que implementam os métodos abstratos – i.e. operações primárias que possuem comportamentos dependentes do objeto específico - especificados na *AbstractClass*.

## GoF Comportamental – Template Method



Modelagem Aplicada/Instaciada

## GoF Comportamental – Template Method

MOTIVAÇÃO

Suponha que existam operações bem definidas para processamento de dados.

Apesar dessas operações serem sempre as mesmas para qualquer processamento (ex. `consultarDados`, `processarDados`, `exibirResultado`), seus comportamentos variam dependendo da abordagem utilizada. Por exemplo, processar dados a partir de uma base de dados difere de processar dados a partir de um sistema de arquivos.

Como melhorar a manutenção e a reusabilidade nesse cenário?

**Template Method Pattern**  
**(Padrão Método Template)**

## GoF Comportamental – Template Method

SOLUÇÃO

Podemos empregar o padrão de projeto GoF, comportamental, *Template Method*.

Como seria a implementação em Java?



## GoF Comportamental – Template Method

```
AbstractClass.java ✘
package abstractPart;

public abstract class AbstractClass {
    public final void templateMethod() {
        operation1();
        operation2();
        operation3();
    }

    public abstract void operation1();
    public abstract void operation2();
    public abstract void operation3();
}
```

Process

Possível Implementação

## GoF Comportamental – Template Method

```
ConcreteClass.java ✘
package concretePart;

import abstractPart.AbstractClass;

public class ConcreteClass extends AbstractClass {
    public ConcreteClass() {
        // TODO Auto-generated constructor stub
    }

    @Override
    public void operation1() {
        // TODO Auto-generated method stub
        System.out.println("Implementing the body of the operation1, specific for this ConcreteClass");
    }

    @Override
    public void operation2() {
        // TODO Auto-generated method stub
        System.out.println("Implementing the body of the operation2, specific for this ConcreteClass");
    }

    @Override
    public void operation3() {
        // TODO Auto-generated method stub
        System.out.println("Implementing the body of the operation3, specific for this ConcreteClass");
    }
}
```

ProcessByDatabase

Possível Implementação

## GoF Comportamental – Template Method

```
TemplateMethodTest.java
```

```
package test;

import abstractPart.AbstractClass;
import concretePart.ConcreteClass;

public class TemplateMethodTest {

    /**
     * Testing the Template Method Pattern...
     */
    public static void main(String[] args) {
        AbstractClass generic = new ConcreteClass();
        generic.templateMethod();

    }
}
```

Possível Implementação

## GoF Comportamental – Template Method

```
Problems @ Javadoc Declaration Console 
<terminated> TemplateMethodTest () [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Oct 19, 2014 3:55:49 PM)
Implementing the body of the operation1, specific for this ConcreteClass
Implementing the body of the operation2, specific for this ConcreteClass
Implementing the body of the operation3, specific for this ConcreteClass
```

Possível Implementação

# Extra Classe



## Extra Classe\_01

Complementem o repositório de catálogo de padrões de projeto, agora, implementando os padrões GoFs comportamentais!

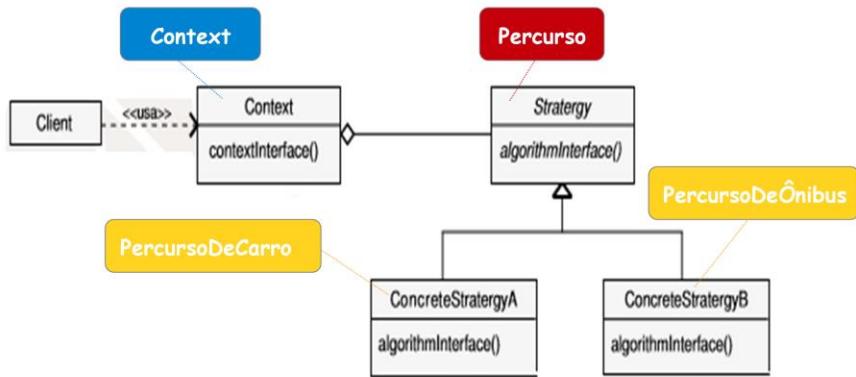
Boa implementação! : )

## Extra Classe\_02

Considerem a modelagem e o contexto apresentado no próximo slide.

Procurem implementar em Java o padrão Strategy nesse cenário! :)

## Extra Classe\_02



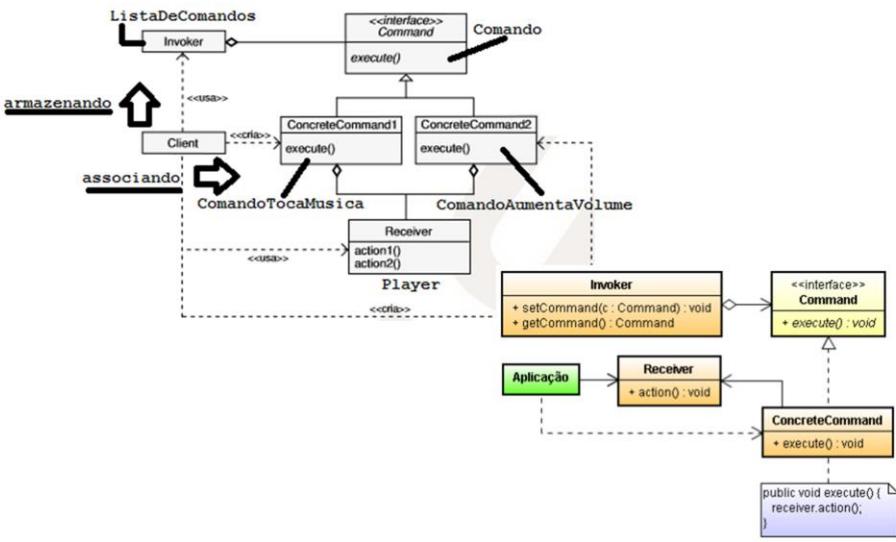
# Complementar



## Complementar

- Seguem *slides* com outros padrões GoFs comportamentais.
- Procurem implementá-los.
- Qualquer dúvida, entrem em contato comigo ou com os nossos monitores.

## Command - Modelagem



Possíveis modelagens...

Pequenas variações são comuns na literatura, em especial na relação da entidade *Client* e as entidades *Receiver* e *ConcreteCommands*.

Procurei separar duas que são comumente encontradas.

A primeira é mais purista.

A segunda é uma adaptação.

Ilustrado com *Client*, *ListaDeComandos*, *Command*, *ComandoTocaMusica*, *ComandoAumentaVolume* e *Player*.

Poderia ser, conforme colocado na implementação disponível no próximo slide.

Ou outro uso, em outro domínio, aplicado de forma similar. Ok?

## Command - Implementação

- Por gentileza, consultem:

<https://pt.wikipedia.org/wiki/Command>

- Reparem que têm, ao final do artigo, outros *links* para implementações desse padrão em outras linguagens.

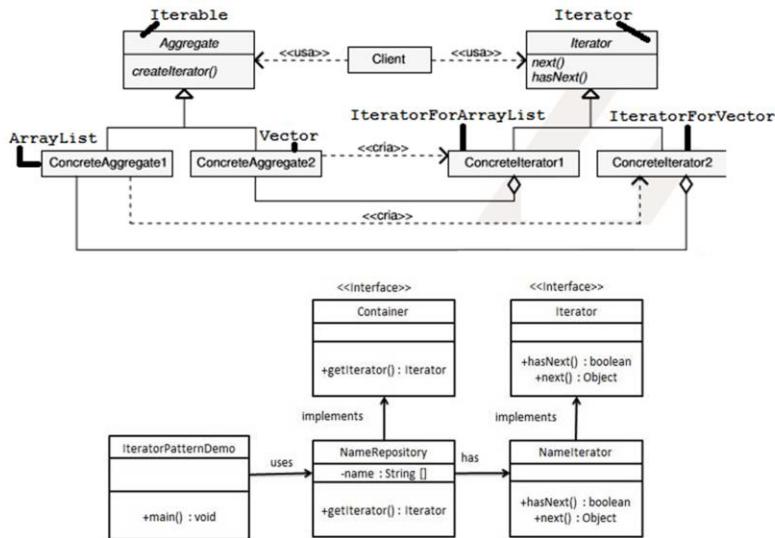
Possível implementação...

Achei o exemplo da *Wikipedia* bem didático, e em um contexto simples de entender.  
Sempre ajustar detalhes é necessário.

Procure implementar.

Qualquer dúvida, entre em contato comigo ou com os monitores. Ok?

## Iterator - Modelagem



Possíveis modelagens...

Pequenas variações são comuns na literatura, em especial na relação da entidade *Client* e as entidades concretas bem como na relação entre a Concreta que implementa a interface *Aggregate* e a Concreta que implementa a interface *Iterator*. Procurei separar duas que são comumente encontradas.

A primeira é mais purista.

A segunda é uma adaptação. Aqui, uma das principais adaptações visíveis na modelagem está no fato da relação entre *NameRepository* e *NameIterator* ser navegável na direção da primeira para a segunda, via associação simples.

No padrão purista, repare que essa relação é de todo-partes, mais forte. Ok?

Ilustrado com *Client*, *Iterable*, *ArrayList*, *Vector*, *Iterator*, *IteratorForArrayList* e *IteratorForVector*.

Poderia ser, conforme colocado nas implementações disponíveis no próximo *slide*.

Ou outro uso, em outro domínio, aplicado de forma similar. Ok?

## Iterator - Implementação

- Por gentileza, consultem:

[https://www.tutorialspoint.com/design\\_pattern/iterator\\_pattern.htm](https://www.tutorialspoint.com/design_pattern/iterator_pattern.htm)  
[https://sourcemaking.com/design\\_patterns/iterator](https://sourcemaking.com/design_patterns/iterator)

- Reparem que têm, ao final do artigo, outros *links* para implementações desse padrão em outras linguagens.

Possíveis implementações...

Existem muitos outros *sites* com propostas de implementação desse padrão.

Trata-se de um padrão mais complexo, apesar de muito utilizado.

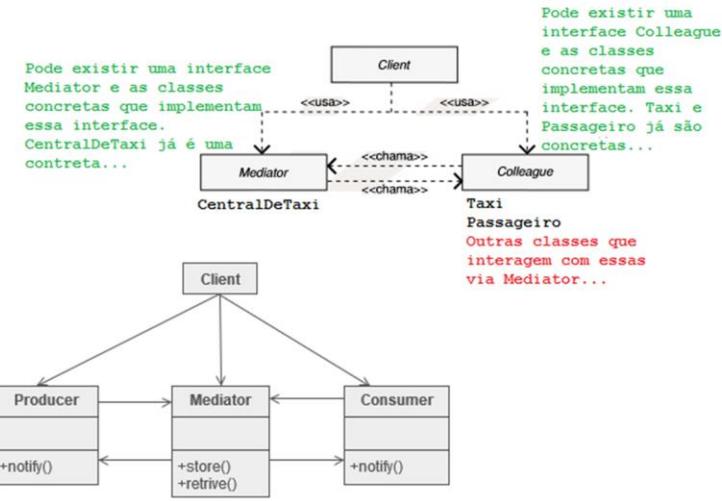
Procure implementar um exemplo que seja o mais purista possível, mas que seja em um contexto que o ajude a entender a aplicabilidade desse padrão.

As implementações aqui indicadas são adaptações. Ok?

Sempre ajustar detalhes é necessário.

Qualquer dúvida, entre em contato comigo ou com os monitores. Ok?

## Mediator - Modelagem



Possíveis modelagens...

Pequenas variações são comuns na literatura.

Aqui, ambas são bem puristas.

Ilustrado com *Client*, *CentralDeTaxi*, *Taxi* e *Passageiro*.

Poderia ser, conforme colocado nas implementações disponíveis no próximo slide.

Ou outro uso, em outro domínio, aplicado de forma similar. Ok?

## Mediator - Implementação

- Por gentileza, consultem:

[https://sourcemaking.com/design\\_patterns/mediator/java/2](https://sourcemaking.com/design_patterns/mediator/java/2)

<https://brizeno.wordpress.com/category/padroes-de-projeto/mediator/>

- Reparem que têm, ao final do artigo, outros *links* para implementações desse padrão em outras linguagens.

Possíveis implementações...

Existem muitos outros *sites* com propostas de implementação desse padrão.  
Trata-se de um padrão simples, fácil de ser implementado.

No primeiro exemplo, são utilizados conceitos de concorrência.

Portanto, trata-se de um exemplo bom, mas precisa entender o problema de concorrência Produtor-Consumidor para compreender bem o exemplo.

No segundo, o contexto é mais amigável.

Trata-se de simular um aplicativo capaz de trocar mensagens entre diversas plataformas móveis:

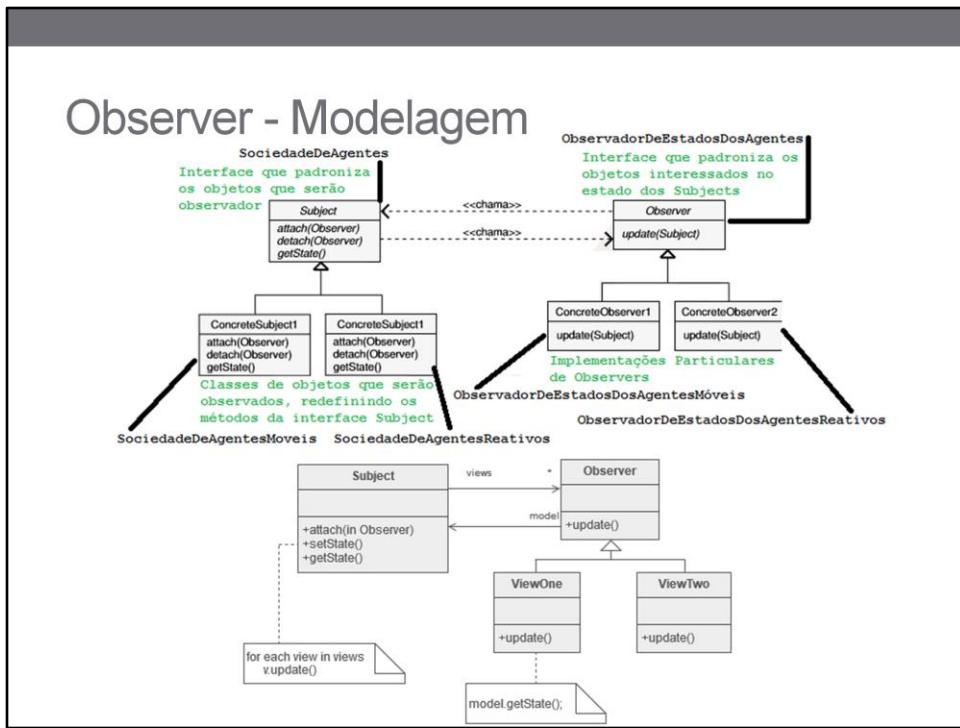
- um Android enviando mensagem para um iOS;
- um Symbian trocando mensagens com um Android.

O problema é que cada uma destas plataformas implementa maneiras diferentes de receber mensagens.

Assim, tem-se o MEDIADOR facilitando essa conversa.

Sempre ajustar detalhes é necessário.

Qualquer dúvida, entre em contato comigo ou com os monitores. Ok?



Possíveis modelagens...

Pequenas variações são comuns na literatura.

Aqui, ambas são bem puristas.

Ilustrado com *SociedadeDeAgentes*, *SociedadeDeAgentesMóveis*,  
*SociedadeDeAgentesReativos*, *ObservadorDeEstadosDosAgentes*,  
*ObservadorDeEstadosDosAgentesMóveis* e  
*ObservadorDeEstadosDosAgentesReativos*.

Poderia ser, conforme colocado nas implementações disponíveis no próximo slide.

Ou outro uso, em outro domínio, aplicado de forma similar. Ok?

## Observer - Implementação

- Por gentileza, consultem:

[https://sourcemaking.com/design\\_patterns/observer/java/1](https://sourcemaking.com/design_patterns/observer/java/1)

[https://www.tutorialspoint.com/design\\_pattern/observer\\_pattern.htm](https://www.tutorialspoint.com/design_pattern/observer_pattern.htm)

- Reparem que têm, ao final do artigo, outros *links* para implementações desse padrão em outras linguagens.

Possíveis implementações...

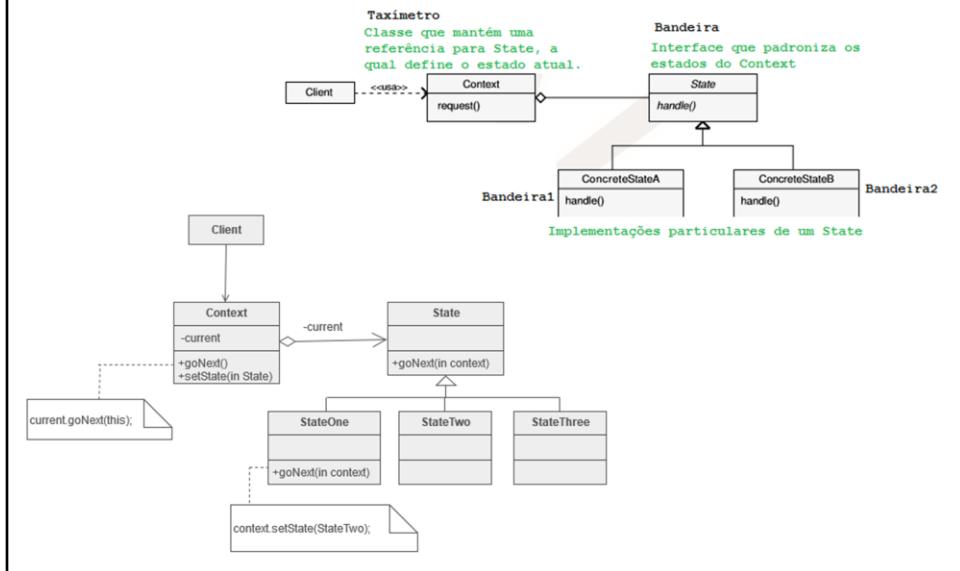
Existem muitos outros *sites* com propostas de implementação desse padrão. Trata-se de um padrão bem utilizado, sendo comum observar o mesmo sendo implementado em códigos gerados via plataformas automatizadas.

Procure reparar, caso esteja utilizando uma em seus trabalhos.

Sempre ajustar detalhes é necessário.

Qualquer dúvida, entre em contato comigo ou com os monitores. Ok?

## State - Modelagem



Possível modelagem...

Pequenas variações são comuns na literatura.

Essa é purista.

Qualquer variação, por gentileza, considere como adaptações desse padrão.

Repare como é semelhante à modelagem do padrão de projeto Strategy, visto na aula desse material.

A diferença entre eles é semântica:

- o State foca em estados diferentes, e
- o Strategy foca em estratégias diferentes.

Ilustrado com *Client*, *Taximetro*, *Bandeira*, *Bandeira1* e *Bandeira2*.

Poderia ser, conforme colocado na implementação disponível no próximo slide.

Ou outro uso, em outro domínio, aplicado de forma similar. Ok?

## State - Implementação

- Por gentileza, consultem:

[https://sourcemaking.com/design\\_patterns/state/java/5](https://sourcemaking.com/design_patterns/state/java/5)

- Reparem que têm, ao final do artigo, outros *links* para implementações desse padrão em outras linguagens.

Possível implementação...

Existem muitos outros *sites* com propostas de implementação desse padrão.  
Trata-se de um padrão bem utilizado.

Sempre ajustar detalhes é necessário.

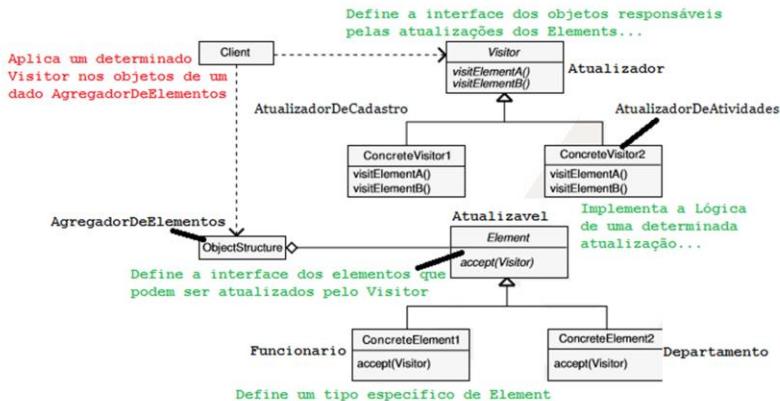
Repare que várias implementações disponíveis na internet consideram associações simples entre *Context* e *State*.

No purismo, essa relação é de **todo-parte**.

Portanto, nos casos que implementar usando uma associação simples, considerar que está sendo implementada uma adaptação do padrão *State*.

Qualquer dúvida, entre em contato comigo ou com os monitores. Ok?

# Visitor - Modelagem



Possível modelagem...

Pequenas variações são comuns na literatura.

Essa é purista.

Qualquer variação, por gentileza, considere como adaptações desse padrão.

Ilustrado com *Client*, *Atualizável*, *Funcionario*, *Departamento*, *Atualizador*, *AtualizadorDeCadastro* e *AtualizadorDeAtividade*.

Poderia ser: *Client*, *Atualizável*, *Cadastro*, *Atividade*, *Atualizador*, *AtualizadorDeFuncionario* e *AtualizadorDeDepartamento*.

Tudo dependerá do contexto.

Poderia ser ainda, conforme colocado na implementação disponível no próximo slide.

Ou outro uso, em outro domínio, aplicado de forma similar. Ok?

## Visitor - Implementação

- Por gentileza, consultem:

[https://sourcemaking.com/design\\_patterns/visitor/java/1](https://sourcemaking.com/design_patterns/visitor/java/1)

- Reparem que têm, ao final do artigo, outros *links* para implementações desse padrão em outras linguagens.

Possível implementação...

Existem muitos outros *sites* com propostas de implementação desse padrão.

Sempre ajustar detalhes é necessário.

Repare que várias implementações disponíveis na internet desconsideram a entidade que representa *ObjectStructure* como agregador de elementos.

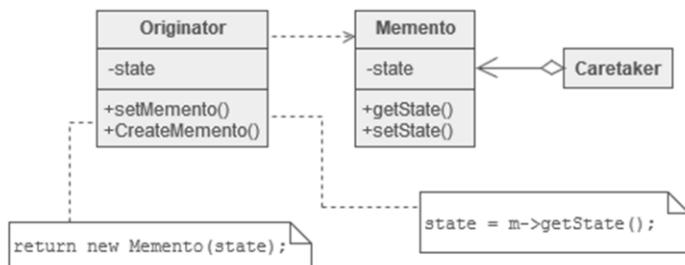
Muitas vezes, usam a própria *main* para trabalhar a questão de agregador de elementos. Ok?

No purismo, essa entidade é bem evidente.

Portanto, nos casos que implementar usando outra lógica, considerar que está sendo implementada uma adaptação do padrão *Visitor*.

Qualquer dúvida, entre em contato comigo ou com os monitores. Ok?

## Memento - Modelagem



Possível modelagem...

Pequenas variações são comuns na literatura.

Essa é purista, com relação de **todo-parte** entre **CareTaker** e **Memento**, pois podemos ter vários **MEMENTOS** envolvidos na lógica de solução indicada por esse padrão.

Qualquer variação, por gentileza, considere como adaptações desse padrão.

Ilustrado de forma mais genérica, conforme colocado na implementação disponível no próximo slide.

Ou outro uso, em outro domínio, aplicado de forma similar. Ok?

## Memento - Implementação

- Por gentileza, consultem:

[https://sourcemaking.com/design\\_patterns/memento/java/1](https://sourcemaking.com/design_patterns/memento/java/1)

[https://www.tutorialspoint.com/design\\_pattern/memento\\_pattern.htm](https://www.tutorialspoint.com/design_pattern/memento_pattern.htm)

- Reparem que têm, ao final do artigo, outros *links* para implementações desse padrão em outras linguagens.

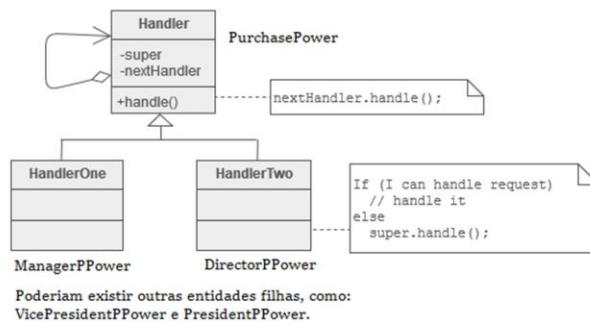
Possíveis implementações...

Existem muitos outros *sites* com propostas de implementação desse padrão.

Sempre ajustar detalhes é necessário.

Qualquer dúvida, entre em contato comigo ou com os monitores. Ok?

## Chain of Responsibility - Modelagem



Possível modelagem...

Pequenas variações são comuns na literatura.

Essa é purista.

Qualquer variação, por gentileza, considere como adaptações desse padrão.

Ilustrado com *PurchasePower*, *ManagerPPower*, *DirectorPPower*, *VicePresidentPPower* e *PresidentPPower*, conforme colocado em uma das implementações disponíveis no próximo slide.

Ou outro uso, em outro domínio, aplicado de forma similar. Ok?

## Chain of Responsibility - Implementação

- Por gentileza, consultem:

[https://pt.wikipedia.org/wiki/Chain\\_of\\_Responsibility](https://pt.wikipedia.org/wiki/Chain_of_Responsibility)

[https://www.tutorialspoint.com/design\\_pattern/chain\\_of\\_responsibility\\_pattern.htm](https://www.tutorialspoint.com/design_pattern/chain_of_responsibility_pattern.htm)

- Reparem que têm, ao final do artigo, outros *links* para implementações desse padrão em outras linguagens.

Possíveis implementações...

Existem muitos outros *sites* com propostas de implementação desse padrão.

Sempre ajustar detalhes é necessário.

Existem variações em termos de implementação.

Esse exemplo está bem dentro do que se espera na implementação do padrão.

Mas, mesmo esses usam associações e não relações de todo-parte em alguns pontos.

Portanto, sempre que a lógica variar em algum ponto, interferindo na modelagem

desse padrão, considerar que está sendo implementada uma adaptação do padrão

*Chain of Responsibility*.

Qualquer dúvida, entre em contato comigo ou com os monitores. Ok?

## Considerações Finais



## Considerações Finais

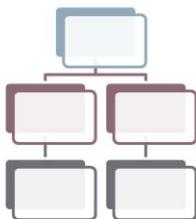
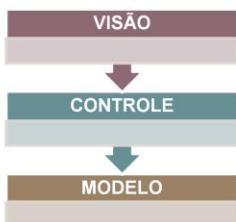
- Nessa aula, conhecemos os padrões GoFs comportamentais.
- Continuem os estudos! Só se aprende praticando!
- Nas referências, têm vários materiais complementares! : )

# Referências



## Referências

- LARMAN, Craig. Utilizando UML e Padrões: Uma Introdução a Análise e ao Projeto
- Orientado a Objetos. 3a. edição. Bookman, 2007.
- COCKBURN, Alistair. Escrevendo Casos de Uso Eficazes. Bookman, 2005.
- SILVA, Ricardo Pereira. UML 2 em Modelagem Orientada a Objetos. Visual Books, 2007.
- PRESSMAN, Roger S. Engenharia de Software. 6a. edição . McGraw-Hill, 2006.
- IEEE. SWEBOK-Guide to the Software Engineering Body of Knowledge, 2004.
- SOMMERVILLE, Ian. Engenharia de Software. 8a. edição. Pearson, 2007.



FIM

Dúvidas?

CONTATO:  
[mileneserrano@unb.br](mailto:mileneserrano@unb.br)  
ou  
[mileneserrano@gmail.com](mailto:mileneserrano@gmail.com)

Sugestões?

