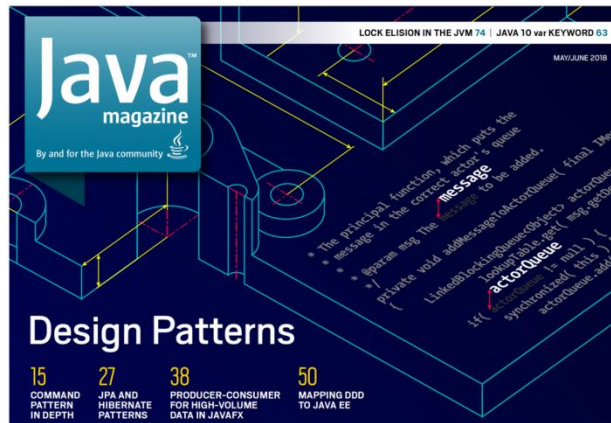




# Unidade 18

## Design Patterns

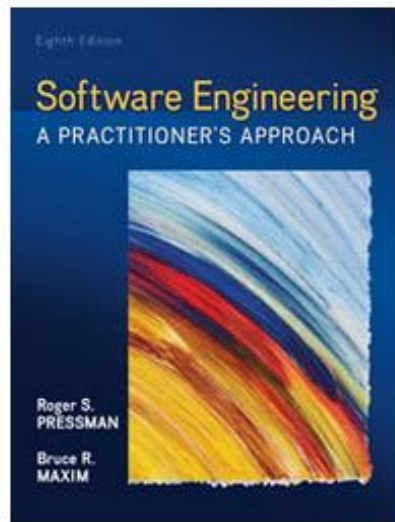


Prof. Aparecido V. de Freitas  
Doutor em Engenharia  
da Computação pela EPUSP  
aparecidovfreitas@gmail.com

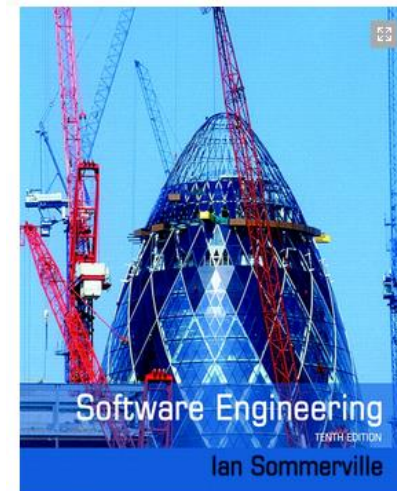


# Bibliografia

- **Software Engineering – A Practitioner's Approach – Roger S. Pressman – Eight Edition – 2014**
- **Software Engineering – Ian Sommerville – 10<sup>th</sup> edition - 2015**
- Engenharia de Software – Uma abordagem profissional – Roger Pressman - McGraw Hill, Sétima Edição - 2011
- Engenharia de Software – Ian Sommerville – Nona Edição – Addison Wesley, 2007



[Software Engineering: A Practitioner's Approach, 8/e](#)





# Padrões de Projeto de Software

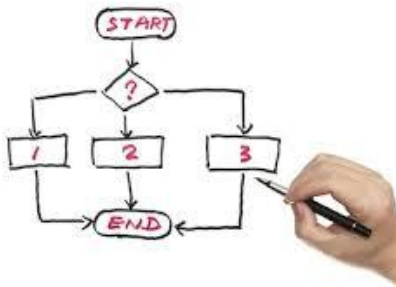
## Design Patterns





# Introdução

- ⊕ Como já visto, software se modifica tanto durante o desenvolvimento quanto em produção;
- ⊕ Mudanças de software, em geral, são difíceis de serem mantidas;
- ⊕ Projetos de software, em geral podem ter módulos com alto acoplamento, dificultando manutenção;





Como facilitar a manutenção de um software?





# Boas práticas para Manutenção de Software



- ⊕ Aplicar bons princípios de **Programação Orientada a Objetos**;
- ⊕ Escrever classes com **alta coesão**;
- ⊕ Escrever classes com **baixo acoplamento**;
- ⊕ Empregar **padrões de projeto de software** (boas práticas).





# Design Patterns



- ✦ Em Engenharia de Software, um padrão de desenho ou padrão de projeto é uma solução geral para um problema que ocorre com frequência dentro de um determinado contexto no projeto de software.







# Design Patters

- ⊕ Classes e métodos muito grandes;
- ⊕ Ninhos de if's e diversos for's no mesmo método;
- ⊕ **Side Effects** – Alteração em uma classe acarreta alteração em diversas outras classes;
- ⊕ Algumas boas práticas foram catalogadas em um conjunto de alternativas para solucionar problemas de Design de Código;
- ⊕ Esse conjunto de boas práticas é conhecido por “**Design Patterns**”.







## Pattern Strategy





# Código complexo e muitas regras

- ⊕ Como exemplo, suponha uma aplicação com uma classe chamada **Orcamento** que tem um atributo chamado **valorOrcamento**;

```
package provaP4;
```

```
public class Orcamento {
```

```
    private double valorOrcamento;
```

```
    public Orcamento(double valorReceita) {
```

```
        this.valorOrcamento = valorReceita;
```

```
    }
```

```
    public double getValorOrcamento() {
```

```
        return valorOrcamento;
```

```
    }
```

```
    public void setValorReceita(double valorOrcamento) {
```

```
        this.valorOrcamento = valorOrcamento;
```

```
    }
```

```
}
```

Orcamento
double valorOrcamento
double getvalorOrcamento() void setvalorOrcamento(double)



## Código complexo e muitas regras

- ✦ Com a classe **Orcamento**, pode-se instanciar objetos que representam orçamentos;
- ✦ Mas, para todo orçamento é necessário, considerar-se impostos. Por exemplo, suponha que seja necessário calcular o ICMS;
- ✦ Para o cálculo dos impostos, pode-se projetar uma outra classe chamada **CalculadoraImpostos** com um método **Calc\_Imposto()** que irá receber um orçamento como parâmetro e retornar o valor do imposto correspondente ao ICMS;
- ✦ Supondo-se que o valor do imposto seja 10% do orçamento, o cálculo corresponde ao valor do imposto multiplicado por 0.1.

Orcamento
double valorOrcamento
double getvalorOrcamento() void setvalorOrcamento(double)

CalculadoraImpostos
double Calc_Imposto(orçamento)



## Código complexo e muitas regras

- Com, nossa aplicação agora já pode calcular o imposto de 10% sobre o Orçamento, correspondente ao INSS.

CalculadoraImpostos
double Calc_Imposto(orcamento)

```
package provaP4;

public class CalculadoraImpostos {

    public double CalcImposto(Orcamento valor) {

        double valorImposto = valor.getValorOrcamento()* 0.10;

        System.out.println("Valor do Imposto: " + valorImposto);

        return valorImposto;
    }

}
```

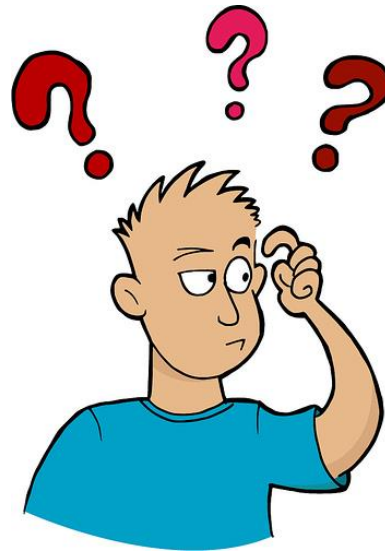


## Código complexo e muitas regras

⊕ Mas, o usuário nos informou **posteriormente** que ainda é preciso calcular outro imposto;

⊕ O ISS corresponde a 5% do valor do orçamento;

⊕ **Como alterar a aplicação corrente para considerar esse novo imposto?**



Orcamento
double valorOrcamento
double getvalorOrcamento() void setvalorOrcamento(double)

CalculadoraImpostos
double Calc_Imposto(orçamento)



## Considerando novo imposto...

- ✦ Pode-se alterar o método **Calc\_Imposto()** da Classe **CalculadoraImpostos** e acrescentar-se uma informação de controle que irá informar ao método qual a regra de negócio correspondente ao imposto a ser calculado;
- ✦ Ou seja, para o **ICMS** o imposto deve ser 10% do orçamento, enquanto que para o **ISS** o valor do imposto correspondente deve ser 5%.

CalculadoraImpostos
<code>double CalcImposto(orçamento, <b>tipoImposto</b>)</code>



# Reescrevendo-se a classe CalculadoraImpostos

⊕ O método Calc\_Imposto agora consegue calcular INSS e ISS.

CalculadoraImpostos
double Calc_Imposto(orcamento, tipoImposto)

```
package provaP4;
```

```
public class CalculadoraImpostos {
```

```
    public Double CalcImposto(Orcamento valor, String tipoImposto) {
```

→ 

```
        if (tipoImposto.equals("INSS")) {
```

```
            double valorImposto = valor.getValorOrcamento()* 0.10;
```

```
            System.out.println("Valor do INSS: " + valorImposto);
```

```
            return valorImposto;
```

```
        }
```

→ 

```
        else if (tipoImposto.equals("ISS")) {
```

```
            double valorImposto = valor.getValorOrcamento()* 0.05;
```

```
            System.out.println("Valor do ISS: " + valorImposto);
```

```
            return valorImposto;
```

```
        }
```

```
        return null;
```

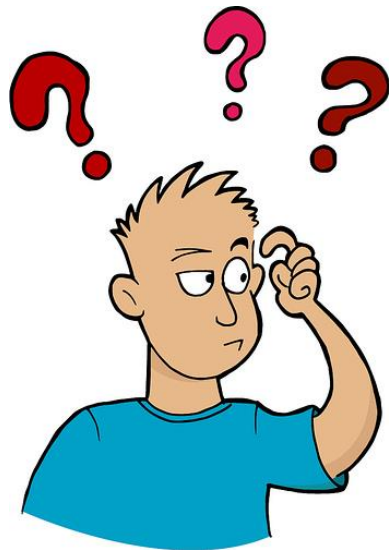
```
    }
```

```
}
```





O que se pode afirmar sobre o projeto das classes ?



Orcamento
double valorOrcamento
double getvalorOrcamento() void setvalorOrcamento(double)

CalculadoraImpostos
double Calc_Imposto(orçamento, <b>tipoImposto</b> )



## Reescrevendo-se a classe CalculadoraImpostos

- ⊕ Pode-se observar que uma das consequências da alteração do código criado é que os cálculos correspondentes aos diversos impostos estão distribuídos na classe **CalculadoraImpostos**;
- ⊕ Assim, a classe CalculadoraImpostos com o método Calc\_Imposto não apresenta coesão, uma vez que ela calcula diversos impostos;
- ⊕ Ou seja, as regras de negócio não estão devidamente encapsuladas e se tornam muito suscetíveis a serem replicadas em outros pontos do código da aplicação;
- ⊕ Para melhorar a coesão do módulo, pode-se implementar o encapsulamento das regras de negócio em uma classe especializada para cada imposto. Assim, cada classe seria responsável pelo cálculo de seu correspondente imposto.

ICMS
double CalculaICMS(orçamento)

ISS
double CalculaISS(orçamento)



## Redesenhando-se as classes...

```
package designpattern;
```

```
public class ICMS {
```

```
    public Double CalculaISS(Orcamento valor) {
```

```
        double valorImposto = valor.getValorOrcamento()* 0.10;
```

```
        System.out.println("Valor do ICMS: " + valorImposto);
```

```
        return valorImposto;
```

```
    }
```

```
}
```

ICMS
double CalculaICMS(orçamento)



## Redesenhando-se as classes...

```
package designpattern;
```

```
public class ISS {
```

```
    public Double CalculaISS(Orcamento valor) {
```

```
        double valorImposto = valor.getValorOrcamento()* 0.05;
```

```
        System.out.println("Valor do ISS: " + valorImposto);
```

```
        return valorImposto;
```

```
    }
```

```
}
```

ISS
double CalculaISS(orçamento)




# Reescrevendo-se a classe CalculadoraImpostos

```
package designpattern;
```

```
public class CalculadoraImpostos {
```

```
    public Double CalcImposto(Orcamento valor, String tipoImposto) {
```



```
        if (tipoImposto.equals("INSS") ) {
```

```
            ICMS icms = new ICMS();
```

```
            double valorICMS = icms.CalculaICMS(valor);
```

```
            return valorICMS;
```

```
        }
```



```
        else if (tipoImposto.equals("ISS")) {
```

```
            ISS iss = new ISS();
```

```
            double valorISS = iss.CalculaISS(valor);
```

```
            return valorISS;
```

```
        }
```

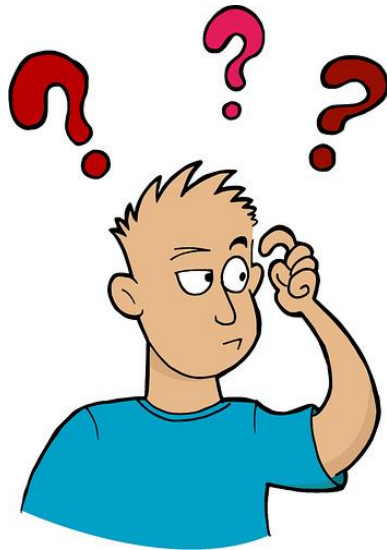
```
        return null;
```

```
    }
```

```
}
```



# Houve alguma melhoria no projeto?



Orcamento
double valorOrcamento
double getvalorOrcamento() void setvalorOrcamento(double)

CalculadoraImpostos
double Calc_Imposto(orçamento, <b>tipoImposto</b> )

ICMS
double CalculaICMS(orçamento)

ISS
double CalculaISS(orçamento)



# Avaliando-se o novo projeto

- ⊕ Houve uma melhora no código;
- ⊕ Porém, como ponto crítico pode-se afirmar que quando houver mais algum novo imposto a ser calculado, será necessário alterar-se o código do cálculo do imposto, adicionando-se mais um bloco de if, além da criação de uma nova classe que encapsulará o novo imposto.

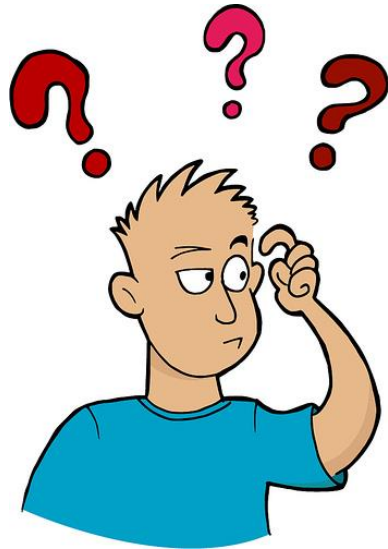
```
public class CalculadoraImpostos {  
    public Double CalcImposto(Orçamento valor, String tipoImposto) {  
        if (tipoImposto.equals("INSS") ) {  
  
            ICMS icms = new ICMS();  
  
            double valorICMS = icms.CalculaICMS(valor);  
  
            return valorICMS;  
        }  
        else if (tipoImposto.equals("ISS")) {  
  
            ISS iss = new ISS();  
  
            double valorISS = iss.CalculaISS(valor);  
  
            return valorISS;  
        }  
        return null;  
    }  
}
```





Orcamento
double valorOrcamento
double getvalorOrcamento() void setvalorOrcamento(double)

Será que ainda podemos melhorar o projeto?



CalculadoraImpostos
double Calc_Imposto(orçamento, tipoImposto)

ICMS
double CalculaICMS(orçamento)

ISS
double CalculaISS(orçamento)



# Eliminando os condicionais com Polimorfismo

- ✚ O que se deseja no código é eliminar-se os condicionais, ou seja, eliminar-se o ninho de if's presentes na classe CalculadoraImpostos;
- ✚ Com essa abordagem, poder-se-ia eliminar o parâmetro **tipoImposto** que referencia o Tipo de Imposto e presente no método **CalcImposto()**.

```
public class CalculadoraImpostos {  
    public Double CalcImposto(Orcamento valor, String tipoImposto) {  
        if (tipoImposto.equals("INSS") ) {
```

```
        .  
        .  
        .
```



# Primeira Solução

- ✚ Poder-se-ia criar dois métodos separados na classe **CalculadoraImpostos**. Um para o **ICMS** e outro para o **ISS**;

```
package designpattern;
```

```
public class CalculadoraImpostos {
```

```
    public Double CalcISS(Orcamento valor) {
```

```
        ICMS icms = new ICMS();
```

```
        double valorICMS = icms.CalculaICMS(valor);
```

```
        return valorICMS;
```

```
    }
```

```
    public Double CalcICMS(Orcamento valor) {
```

```
        ICMS icms = new ICMS();
```

```
        double valorICMS = icms.CalculaICMS(valor);
```

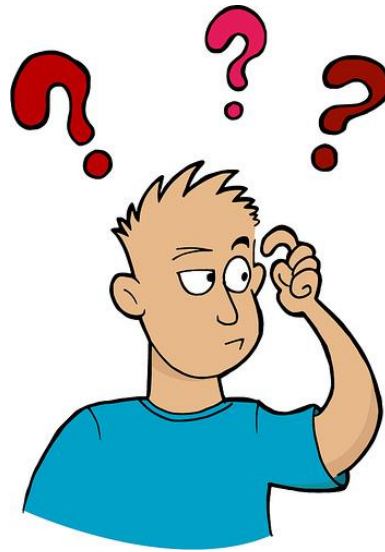
```
        return valorICMS;
```

```
    }
```

```
}
```



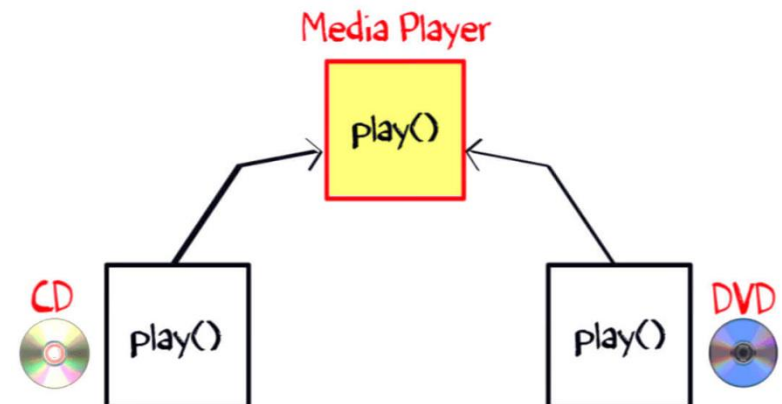
Houve alguma melhoria no projeto?





## Houve melhoria no projeto?

- ✦ Sim, no entanto ao invés de termos vários if's temos agora várias classes;
- ✦ O ideal seria termos um único método, **genérico**, que conseguiria realizar o cálculo para qualquer imposto, sem nenhum if dentro dele;
- ✦ Na verdade, precisamos que o código fique flexível o bastante para que se possa utilizar diversos impostos na execução do cálculo;
- ✦ Podemos para isso, podemos criar uma interface chamada **Imposto** e devemos fazer com que as classes **ISS** e **ICMS** implementem esta interface.





## Definindo a Interface

```
package designpattern;  
  
public interface Imposto {  
  
    Double CalculaImposto (Orcamento orcamento);  
  
}
```



# ICMS implementando a interface

```
package designpattern;  
  
public class ICMS implements imposto {  
  
    public Double CalculaImposto(Orcamento valor) {  
  
        double valorImposto = valor.getValorOrcamento()* 0.10;  
  
        System.out.println("Valor do ICMS: " + valorImposto);  
  
        return valorImposto;  
  
    }  
}
```





# ISS implementando a interface

```
package designpattern;

public class ISS implements Imposto {

    public Double CalculaImposto(Orcamento valor) {

        double valorImposto = valor.getValorOrcamento()* 0.05;

        System.out.println("Valor do ISS: " + valorImposto);

        return valorImposto;

    }

}
```



```
package designpattern;

public class CalculadoraImpostos {

    public Double realizaCalculo(Orcamento valor, Imposto imposto) {

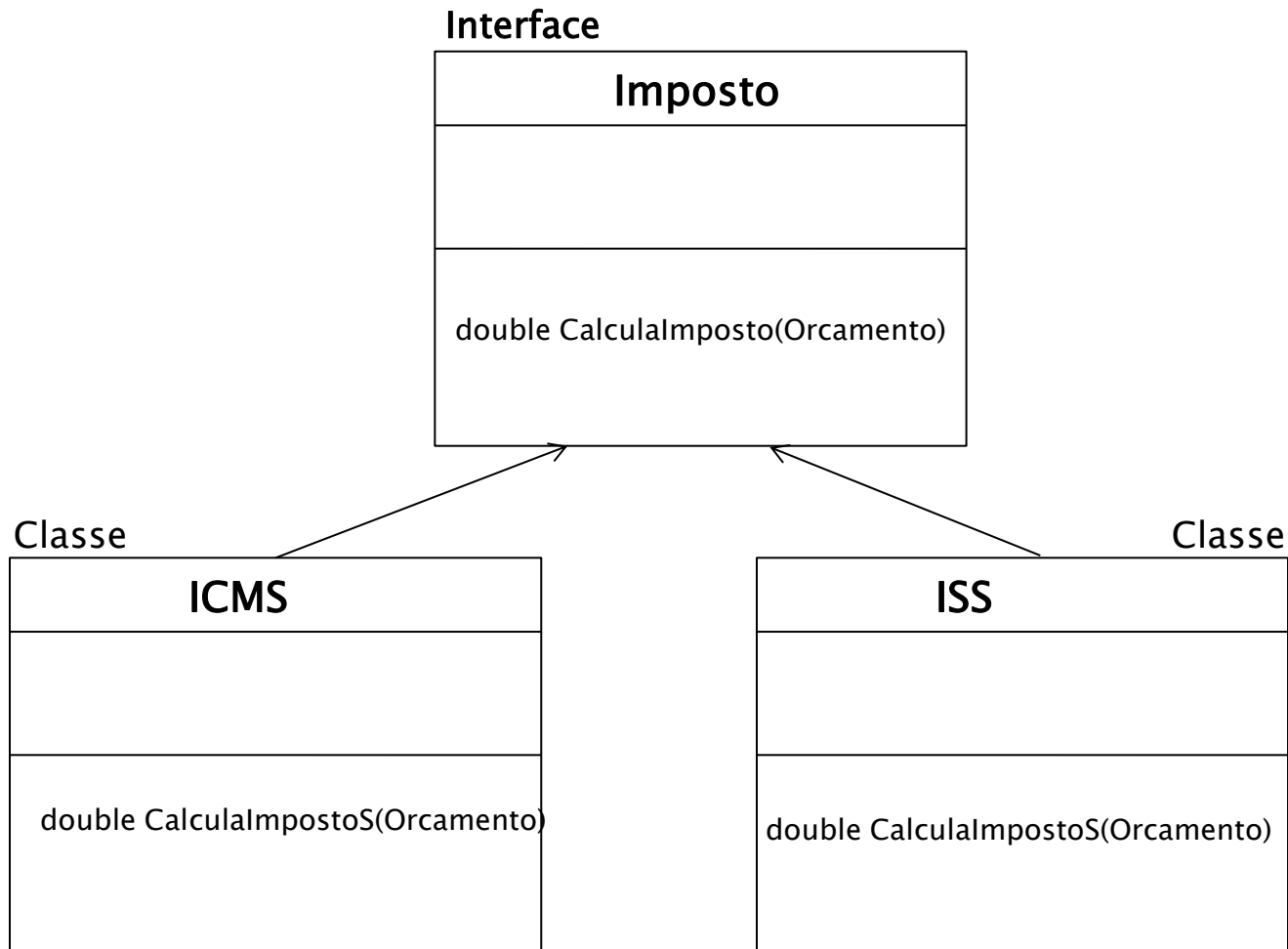
        Double valorImposto = imposto.CalculaImposto(valor);

        System.out.println("Valor do Imposto: " + valorImposto);

        return valorImposto;
    }
}
```



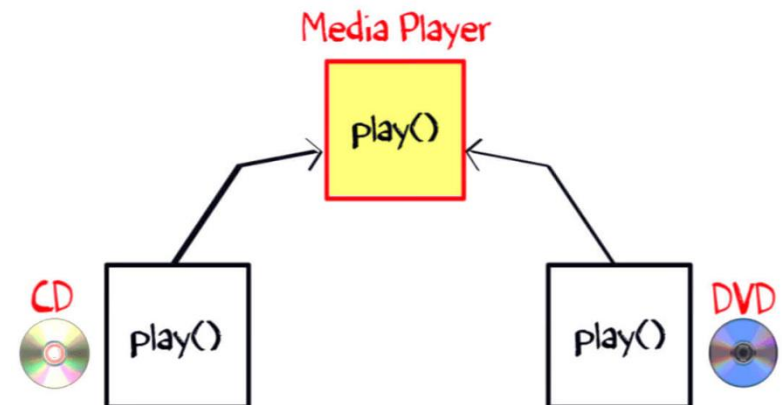
# Desenho final do Projeto





## Desenho final do Projeto

- ⊕ Sim, no entanto ao invés de termos vários if's temos agora várias classes;
- ⊕ O ideal seria termos um único método, **genérico**, que conseguiria realizar o cálculo para qualquer imposto, sem nenhum if dentro dele;
- ⊕ Na verdade, precisamos que o código fique flexível o bastante para que se possa utilizar diversos impostos na execução do cálculo;
- ⊕ Podemos para isso, podemos criar uma interface chamada **Imposto** e devemos fazer com que as classes **ISS** e **ICMS** implementem esta interface.





## Validando o projeto

```
package designpattern;  
  
public class TesteImpostos {  
  
    public static void main(String[] args) {  
  
        Imposto iss = new ISS();  
  
        Imposto icms = new ICMS();  
  
        Orcamento orcamento = new Orcamento(1000.0);  
  
        CalculadoraImpostos calculo = new CalculadoraImpostos();  
  
        calculo.realizaCalculo(orcamento, iss);  
  
        calculo.realizaCalculo(orcamento, icms);  
  
    }  
}
```



## Pattern Chain of Responsibility

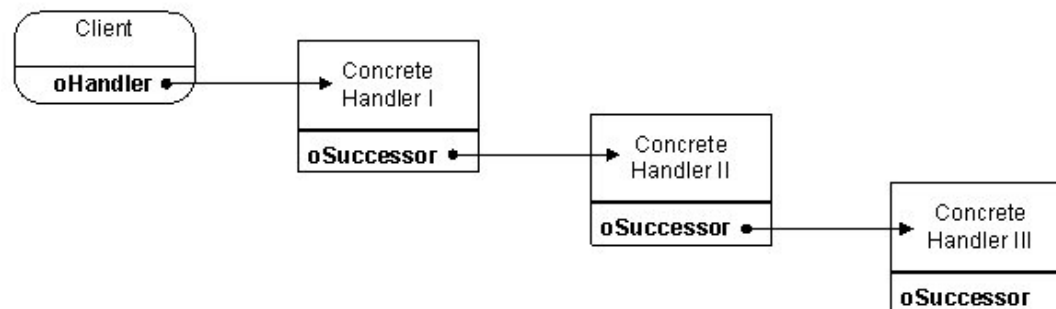




# Chain of Responsibility

- ⊕ Na Programação Orientada a Objetos, deve-se tentar ao máximo manter os objetos com baixo acoplamento de informações;
- ⊕ Este princípio de projeto, acarreta melhoria na qualidade do software;
- ⊕ O padrão “**Chain of Responsibility**” tem como objetivo representar um encadeamento de objetos para realizar o processamento de uma série de requisições diferentes.

Chain of Responsibility Pattern







## Pattern Chain of Responsibility

- ⊕ Considere que um orçamento pode receber desconto de acordo com o tipo de venda que será efetuada;
- ⊕ Por exemplo, se o cliente comprar mais de **10 itens**, ele receberá **10%** de desconto;
- ⊕ Se o orçamento for superior ou igual a **R\$10.000,00**, ele receberá **20%** de desconto;





- ⊕ Um orçamento é composto por diversos itens;

```
package chainOfResponsability;

public class Item {
    private String nome;
    private double valor;

    public Item(String nome, double valor) {
        this.nome = nome;
        this.valor = valor;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public double getValor() {
        return valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }
}
```



⊕ Um orçamento é composto por diversos itens;

```
package chainOfResponsability;
import java.util.List;

public class Orcamento {
    private double valor;
    private List<Item> itens;

    public Orcamento(double valor, List<Item> itens) {
        this.valor = valor;
        this.itens = itens;
    }

    public double getValor() {
        return valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }

    public List<Item> getItens() {
        return itens;
    }

    public void setItens(List<Item> itens) {
        this.itens = itens;
    }

    public void adicionaItem(Item item) {
        itens.add(item);
    }
}
```



# Implementando as regras de Negócio

```
package chainOfResponsability;

public class CalculadoraDescontos {

    public double calculaDesconto (Orcamento orcamento) {

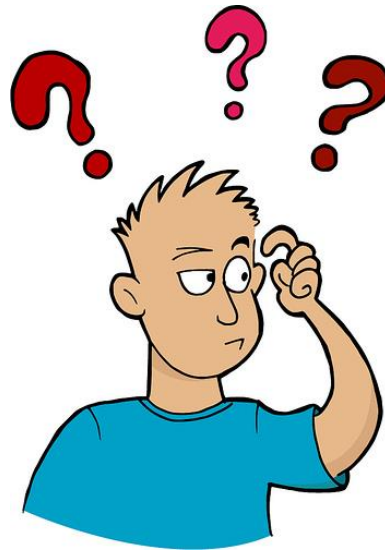
        if (orcamento.getItems().size() > 10)
            return orcamento.getValor() * 0.10;

        else if (orcamento.getValor() > 10000.0 )
            return orcamento.getValor() * 0.20;

        return 0.0;
    }
}
```



O que se pode dizer do projeto?





# Pattern Chain of Responsibility

- ⊕ A implementação da classe **CalculadoraDescontos** corresponde a um ninho de if's;
- ⊕ O método `calculaDesconto` não apresenta coesão, uma vez que calcula diversos impostos oriundos de diferentes regras de negócio;
- ⊕ Ou seja, as regras de negócio não estão devidamente encapsuladas e se tornam muito suscetíveis a serem replicadas em outros pontos do código da aplicação;
- ⊕ Para melhorar a coesão do módulo, pode-se implementar o encapsulamento das regras de negócio em uma **classe especializada para cada desconto**. Assim, cada classe seria responsável pelo cálculo de seu correspondente desconto.





## DescontoPorMaisDeDezItens

### Primeira Regra de Negócio

```
package chainOfResponsability;  
  
public class DescontoPorMaisDeDezItens {  
  
    public double calculaDesconto(Orcamento orcamento) {  
  
        if (orcamento.getItens().size() > 10)  
            return orcamento.getValor() * 0.10;  
  
        return 0.0;  
    }  
}
```



## DescontoPorMaisDeMilReais Segunda Regra de Negócio

```
package chainOfResponsability;  
  
public class DescontoPorMaisDeMilReais {  
  
    public double calculaDesconto(Orcamento orcamento) {  
  
        if (orcamento.getValor() > 1000.0)  
            return orcamento.getValor() * 0.20;  
  
        return 0.0;  
    }  
}
```





# Ajuste da Classe CalculadoraDesconto

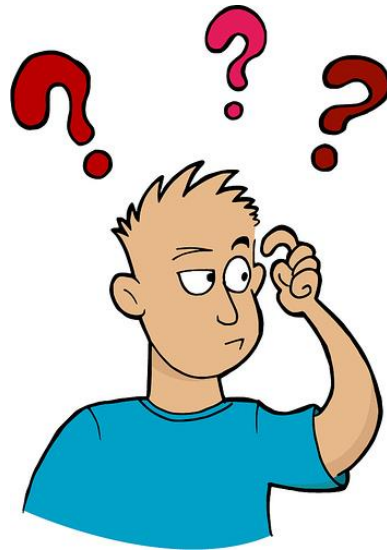
```
package chainOfResponsability;  
  
public class CalculadoraDescontos {  
  
    public double calculaDesconto (Orcamento orcamento) {  
  
        DescontoPorMaisDeDezItens objDescontoRegra1 = new DescontoPorMaisDeDezItens();  
  
        double desconto = objDescontoRegra1.desconta(orcamento);  
  
        if (desconto == 0 ) {  
            DescontoPorMaisDeMilReais objDescontoRegra2 = new DescontoPorMaisDeMilReais();  
            desconto = objDescontoRegra2.desconta(orcamento);  
        }  
        return desconto;  
    }  
}
```

Se o desconto da primeira regra de negócio for zero, deve-se tentar o desconto da próxima regra de negócio!





# Houve melhoria no projeto?





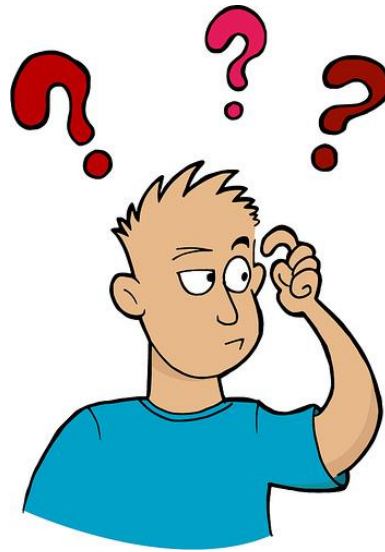
## Observações

- ⊕ Houve melhoria no código;
- ⊕ Cada regra de negócio está implementada em sua respectiva classe;
- ⊕ Porém, o problema está na forma como se deve fazer a **sequência** de descontos ser aplicada na ordem, pois é necessário incluir-se mais um **if** sempre que houver um novo desconto.

```
public double calculaDesconto (Orcamento orcamento) {  
  
    DescontoPorMaisDeDezItens objDescontoRegra1 = new DescontoPorMaisDeDezItens();  
  
    double desconto = objDescontoRegra1.desconta(orcamento);  
  
    → if (desconto == 0 ) {  
        DescontoPorMaisDeMilReais objDescontoRegra2 = new DescontoPorMaisDeMilReais();  
        desconto = objDescontoRegra2.desconta(orcamento);  
    }  
    return desconto;  
}
```



## Como melhorar o código?





## Melhorando o código

- ⊕ Todos os descontos têm algo em comum;
- ⊕ Todos eles calculam o desconto dado um orçamento;
- ⊕ Pode-se criar uma abstração para representar um desconto genérico.

```
package chainOfResponsability;  
  
public interface Desconto {  
  
    double calculaDesconto(Orcamento orcamento);  
  
    void setProximo(Desconto proximo);  
  
}
```



```
package chainOfResponsability;

public class DescontoPorMaisDeDezItens implements Desconto {

    private Desconto proximo;

    public Desconto getProximo() {
        return proximo;
    }

    public void setProximo(Desconto proximo) {
        this.proximo = proximo;
    }

    public double calculaDesconto(Orcamento orcamento) {

        if (orcamento.getItens().size() > 10)
            return orcamento.getValor() * 0.10;

        return proximo.calculaDesconto(orcamento);
    }
}
```



```
package chainOfResponsability;

public class DescontoPorMaisDeMilReais implements Desconto {

    private Desconto proximo;

    public Desconto getProximo() {
        return proximo;
    }

    public void setProximo(Desconto proximo) {
        this.proximo = proximo;
    }

    public double calculaDesconto(Orcamento orcamento) {

        if (orcamento.getValor() > 1000.0)
            return orcamento.getValor() * 0.20;

        return proximo.calculaDesconto(orcamento);
    }
}
```



```
package chainOfResponsability;

public class SemDesconto implements Desconto {

    public double calculaDesconto(Orcamento orcamento) {

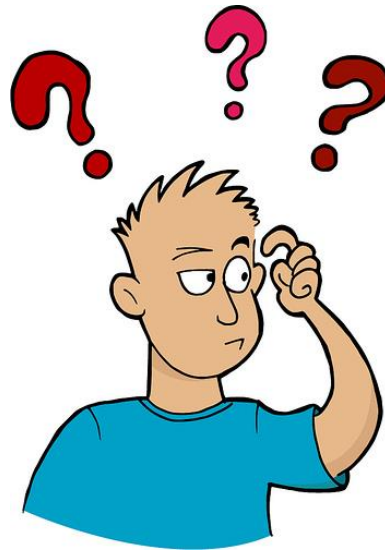
        return 0.0;
    }

    public void setProximo(Desconto desconto) {
        // nao tem
    }
}
```





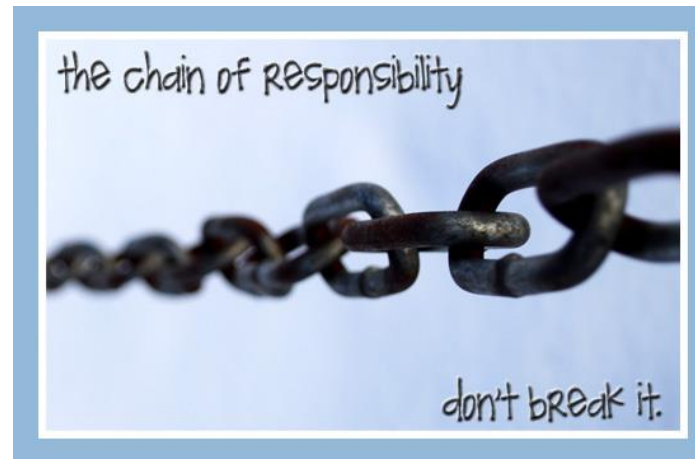
## O que mudou no código





# Observações

- ⊕ Se um orçamento atende a uma regra de negócio de desconto, ele então é calculado;
- ⊕ Caso contrário, passa-se para a próxima regra de negócio, qualquer que seja ela;
- ⊕ Cria-se assim, um encadeamento das regras de negócio;
- ⊕ Assim, para um desconto pouco importa qual é o próximo, uma vez que eles estão totalmente desacoplados;
- ⊕ Estes descontos formam como se fosse uma “corrente”, ou uma lista ligada. Daí o nome de padrão de Projeto: **Chain of Responsibility**.





# Encadeando as Regras de Negócio

```
package chainOfResponsability;  
  
public class CalculadoraDescontos {  
  
    public double calculaDesconto (Orcamento orcamento) {  
  
        Desconto d1 = new DescontoPorMaisDeDezItens();  
  
        Desconto d2 = new DescontoPorMaisDeMilReais();  
  
        d1.setProximo(d2);  
  
        d2.setProximo(d3);  
  
        return d1.calculaDesconto(orcamento);  
  
    }  
}
```



# Executando o código

```
package chainOfResponsability;

import java.util.ArrayList;
import java.util.List;

public class TesteDescontos {

    public static void main(String[] args) {

        CalculadoraDescontos calc = new CalculadoraDescontos();

        Item i1 = new Item("Notebook", 3200);
        Item i2 = new Item("HD", 800);

        List<Item> listaItens = new ArrayList<Item>();
        listaItens.add(i1);
        listaItens.add(i2);

        Orcamento orcamento = new Orcamento(4000.0, listaItens);
        double desconto = calc.calculaDesconto(orcamento);

        System.out.println(desconto);
    }
}
```



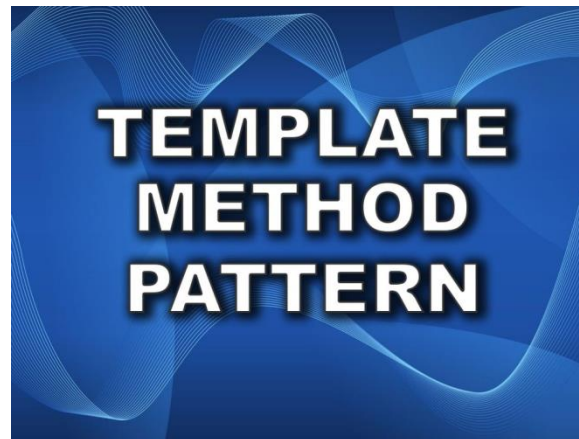
# Pattern Template





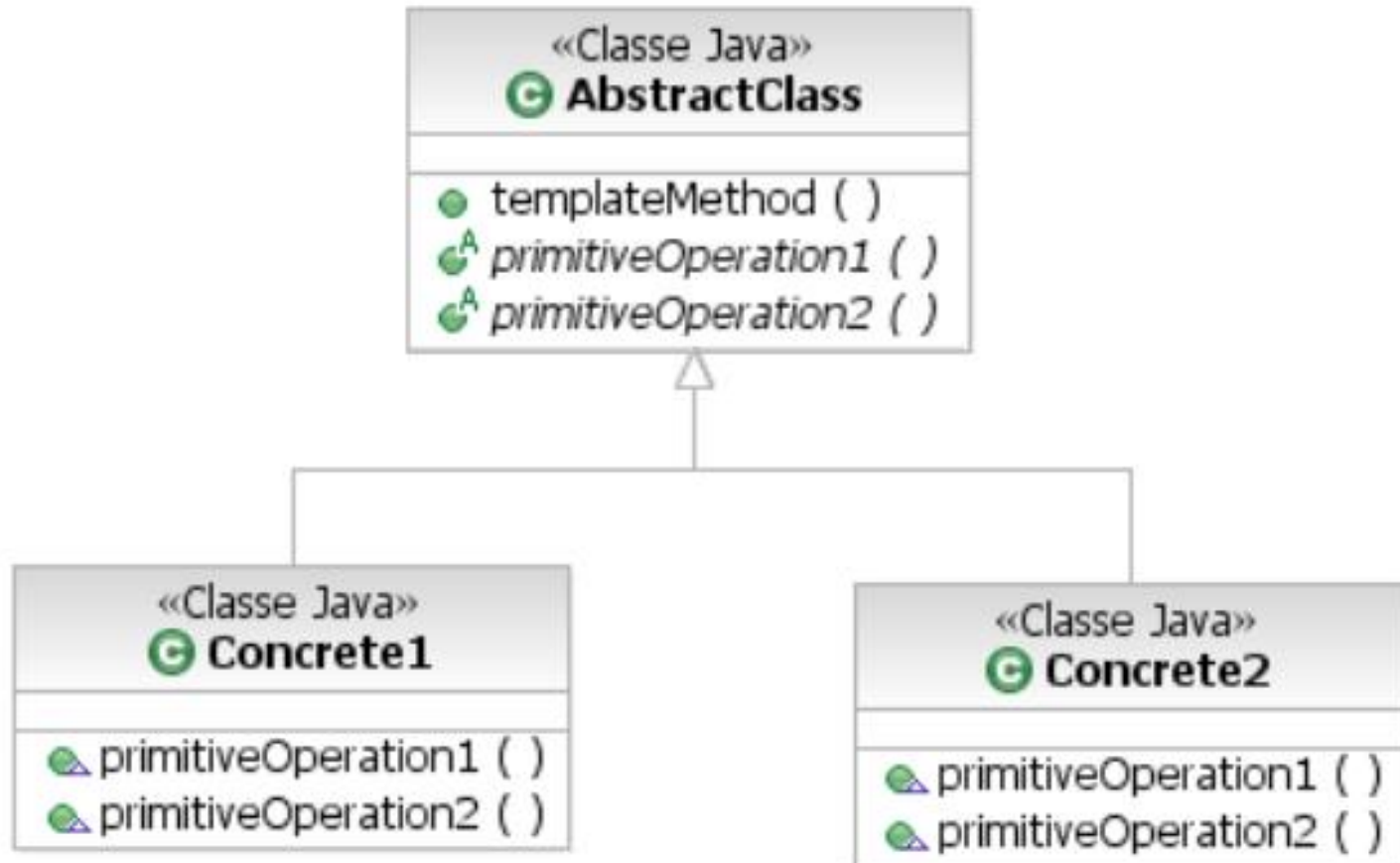
# Padrão Template

- ⊕ Este padrão define os passos de um Algoritmo e permite que a implementação de um ou mais desses passos seja feita por subclasses;
- ⊕ Com isso, o padrão **Template** protege o algoritmo e fornece métodos abstratos para que as subclasses possam implementá-los;
- ⊕ O padrão **Template** define o esqueleto de um algoritmo dentro de um método, transferindo alguns de seus passos para as subclasses. Este padrão permite que as subclasses redefinam certos passos de um algoritmo sem alterar a estrutura do próprio algoritmo.





# Padrão Template





## Exemplo – Padrão Template

- ⊕ Problema: Considere um **player** de música que oferece diversas maneiras de se reproduzir as músicas de um playlist;
- ⊕ Suponhamos que se possa reproduzir a lista de músicas da seguinte forma:
  - ✓ Em ordem de nome de música
  - ✓ Em ordem por nome de autor
  - ✓ Em ordem de ano de lançamento







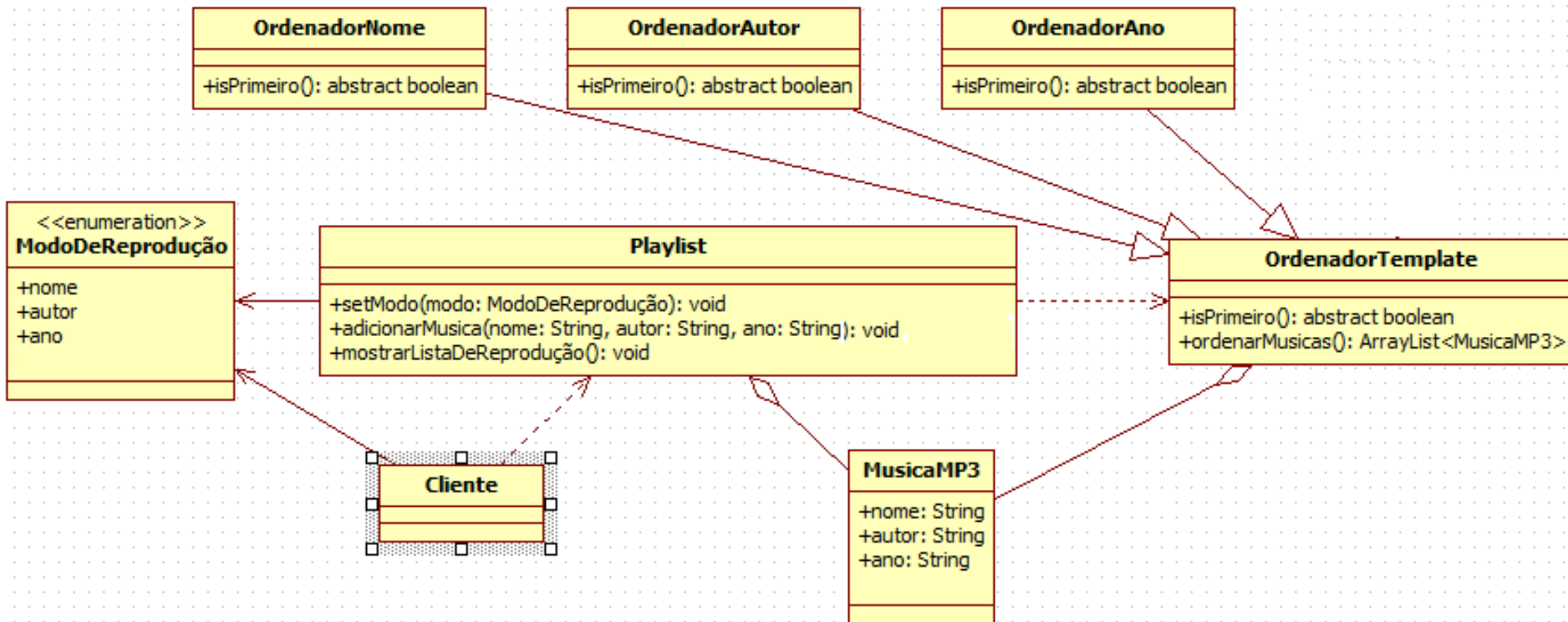
## Exemplo – Padrão Template

- ✦ Pode-se observar que o algoritmo em questão é o mesmo, independentemente de qual modo será feita a reprodução da playlist.





## Exemplo – Padrão Template





# Classe MusicaMP3

```
package template;
```

```
public class MusicaMP3 {
```

```
    String nome;  
    String autor;  
    String ano;
```

```
public MusicaMP3(String nome, String autor, String ano) {
```

```
    this.nome = nome;  
    this.autor = autor;  
    this.ano = ano;
```

```
    }  
}
```



## enum ModoReproducao

```
package template;  
  
public enum ModoReproducao {  
  
    porNome, porAutor, porAno  
  
}
```



# Ordenador Template

```
package template;
import java.util.ArrayList;

public abstract class OrdenadorTemplate {
    public abstract boolean isPrimeiro(MusicaMP3 musica1, MusicaMP3 musica2);

    protected ArrayList<MusicaMP3> ordenarMusica(ArrayList<MusicaMP3> lista) {
        ArrayList<MusicaMP3> novaLista = new ArrayList<MusicaMP3>();
        for (MusicaMP3 musicaMP3 : lista) {
            novaLista.add(musicaMP3);
        }

        for (int i = 0; i < novaLista.size(); i++) {
            for (int j = i; j < novaLista.size(); j++) {
                if (!isPrimeiro(novaLista.get(i), novaLista.get(j))) {
                    MusicaMP3 temp = novaLista.get(j);
                    novaLista.set(j, novaLista.get(i));
                    novaLista.set(i, temp);
                }
            }
        }

        return novaLista;
    }
}
```



## Ordenador por Nome

```
package template;
```

```
public class OrdenadorPorNome extends OrdenadorTemplate {
```

```
    @Override
```

```
    public boolean isPrimeiro(MusicaMP3 musica1, MusicaMP3 musica2) {  
        if (musica1.nome.compareToIgnoreCase(musica2.nome) <= 0)  
            return true;
```

```
        return false;
```

```
    }
```

```
}
```



# Ordenador por Autor

```
package template;
```

```
public class OrdenadorPorAutor extends OrdenadorTemplate {
```

```
    @Override
```

```
    public boolean isPrimeiro(MusicaMP3 musica1, MusicaMP3 musica2) {  
        if (musica1.autor.compareToIgnoreCase(musica2.autor) <= 0)  
            return true;  
        return false;  
    }
```

```
}
```



## Ordenador por Ano

```
package template;
```

```
public class OrdenadorPorAno extends OrdenadorTemplate {
```

```
    @Override
```

```
    public boolean isPrimeiro(MusicaMP3 musica1, MusicaMP3 musica2) {
```

```
        if (musica1.ano.compareToIgnoreCase(musica2.ano) <= 0)
```

```
            return true;
```

```
        return false;
```

```
    }
```

```
}
```





# Classe Playlist

```
package template;
```

```
import java.util.ArrayList;
```

```
public class Playlist {  
    protected ArrayList<MusicaMP3> musicas;  
    protected OrdenadorTemplate ordenador;  
  
    public Playlist(ModoReproducao modo) {  
        musicas = new ArrayList<MusicaMP3>();  
        switch (modo) {  
            case porAno:  
                ordenador = new OrdenadorPorAno();  
                break;  
            case porAutor:  
                ordenador = new OrdenadorPorAutor();  
                break;  
            case porNome:  
                ordenador = new OrdenadorPorNome();  
                break;  
            default:  
                break;  
        }  
    }  
}
```



# Classe Playlist

```
public void setModoReproducao(ModoReproducao modo) {  
  
    this.ordenador = null;  
    switch (modo) {  
        case porAno:  
            ordenador = new OrdenadorPorAno();  
            break;  
        case porAutor:  
            ordenador = new OrdenadorPorAutor();  
            break;  
        case porNome:  
            ordenador = new OrdenadorPorNome();  
            break;  
        default:  
            break;  
    }  
}
```



## Classe Playlist

```
public void adicionarMusica(String nome, String autor, String ano) {  
    musicas.add(new MusicaMP3(nome, autor, ano));  
}  
  
public void mostrarListaDeReproducao() {  
    ArrayList<MusicaMP3> novaLista = new ArrayList<MusicaMP3>();  
    novaLista = ordenador.ordenarMusica(musicas);  
  
    for (MusicaMP3 musica : novaLista) {  
        System.out.println(musica.nome + " - " + musica.autor +  
            "\n Ano: " + musica.ano);  
    }  
}  
}
```



# Classe de execução

```
package template;
```

```
public class TestePlayList {
```

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

```
    Playlist minhaPlayList = new Playlist(ModoReproducao.porNome);  
    minhaPlayList.adicionarMusica("Everlong", "Foo Fighters", "1997");  
    minhaPlayList.adicionarMusica("Song 2", "Blur", "1997");  
    minhaPlayList.adicionarMusica("American Jesus", "Religion", "1993");  
    minhaPlayList.adicionarMusica("No Cigar", "Milencollin", "2001");  
    minhaPlayList.adicionarMusica("Ten", "Pearl Jam", "1991");
```



## Classe de execução

```
System.out.println("=== Lista por Nome de Musica ===");  
minhaPlayList.mostrarListaDeReproducao();
```

```
System.out.println("\n=== Lista por Autor ===");  
minhaPlayList.setModoReproducao(ModoReproducao.porAutor);  
minhaPlayList.mostrarListaDeReproducao();
```

```
System.out.println("\n=== Lista por Ano ===");  
minhaPlayList.setModoReproducao(ModoReproducao.porAno);  
minhaPlayList.mostrarListaDeReproducao();
```

```
}
```

```
}
```



## Pattern Decorator





# Padrão Decorator

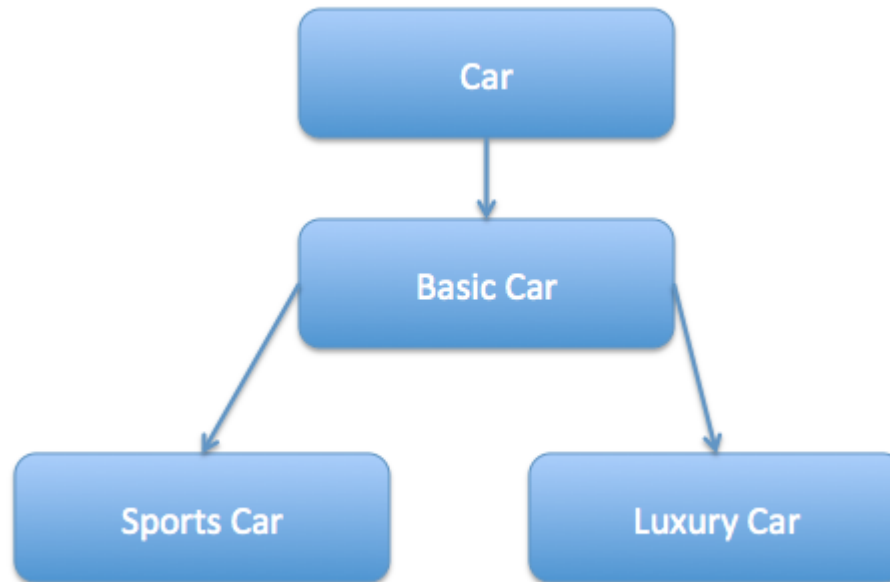
- ✦ É um padrão de projeto que anexa responsabilidades adicionais a um objeto, de forma dinâmica;
- ✦ Com esse pattern pode-se modificar a funcionalidade de um objeto em tempo de execução;
- ✦ Ao mesmo tempo, outras instâncias da mesma classe não serão afetadas por isso, de modo que o objeto em particular, obtém o comportamento modificado.





# Padrão Decorator – Exemplo

- ✦ Suponha-se que se queira implementar diferentes tipos de carros a partir de uma interface;
- ✦ Pode-se criar a interface **Car** para definir o método de montagem e obter um carro básico que pode ainda ser estendido para Carro Esportivo e Carro de Luxo.







## Padrão Decorator – Exemplo

- ✦ Mas, e se quisermos obter um carro que, **em tempo de execução**, tenha características do carro de luxo e também do carro esportivo, a implementação ficará complexa;
- ✦ Essa complexidade certamente irá aumentar, se tivermos dez tipos diferentes de carros;
- ✦ Para essa situação, o padrão **Decorator** irá auxiliar na implementação.

