



Design Patterns

João Vitor Freitas

Sumário

- Introdução
- *Design Patterns:*
 - Builder
 - Factory
 - Strategy
 - Template Method
 - Chain of Responsibility
- Prova

Introdução

Introdução - Problemas comuns

- Classes e métodos gigantescos (5 mil linhas?);
- if if if if if;
- Redundância de código;
- Mudanças em vários pontos do código;
- Díficil de manter, entender e muito acoplado.

Introdução - O que é *Design Pattern*?

- Técnicas catalogadas para resolver problemas;
- Essas técnicas ganharam o nome de *Design Patterns* ou Padrões de projetos;
- Um padrão de projeto nada mais é que uma solução elegante para um problema recorrente;
- Mais importante do que conhecer é saber quando usar!

Design Patterns

Design Patterns

Criação	Estrutural	Comportamental	
Factory Method	Adapter	Chain of Responsibility	Observer
Abstract Factory	Facade	Command	State
Builder	Bridge	Interpreter	Strategy
Prototype	Decorator	Iterator	Template Method
Singleton	Flyweight	Mediator	Visitor
	Composite	Memento	
	Proxy		

Design Patterns

Criação	Estrutural	Comportamental	
Factory Method		Chain of Responsibility	
Builder			Strategy
			Template Method

Builder

```
public class NotaFiscal {  
  
    private String razaoSocial;  
    private String cnpj;  
    private Calendar dataDeEmissao;  
    private double valorBruto;  
    private double impostos;  
    private List<ItemDaNota> itens;  
    private String observacoes;  
  
    NotaFiscal(String razaoSocial, String cnpj, Calendar dataDeEmissao,  
               double valorBruto, double impostos, List<ItemDaNota> itens,  
               String observacoes) {  
        this.razaoSocial = razaoSocial;  
        this.cnpj = cnpj;  
        this.dataDeEmissao = dataDeEmissao;  
        this.valorBruto = valorBruto;  
        this.impostos = impostos;  
        this.itens = itens;  
        this.observacoes = observacoes;  
    }  
}
```



```
@Test
public void constroiNotaFiscal() {
    ArrayList<ItemDaNota> itemDaNotas = new ArrayList<>();
    itemDaNotas.add(new ItemDaNota("nota1", 50d));
    itemDaNotas.add(new ItemDaNota("nota2", 50d));
    double valorTotal = 0d;
    for (ItemDaNota item : itemDaNotas) {
        valorTotal += item.getValor();
    }
    double imposto = valorTotal * 0.05d;

    NotaFiscal notaFiscal = new NotaFiscal(
        "razaoSocial",
        "999.999.999-99",
        Calendar.getInstance(),
        valorTotal,
        imposto,
        itemDaNotas,
        "observação da nota"
    );

    assertEquals(notaFiscal.getItems().size(), 2);
}
```



```
class NotaFiscalBuilder {
    private String razaoSocial;
    private String cnpj;
    private double valorBruto;
    private double impostos;
    private List<ItemDaNota> todosItens = new ArrayList<>();
    private String observacoes;
    private Calendar data;

    NotaFiscalBuilder() {
        this.data = Calendar.getInstance();
    }

    NotaFiscalBuilder paraEmpresa(String razaoSocial) {
        this.razaoSocial = razaoSocial;

        return this;
    }

    NotaFiscalBuilder comCnpj(String cnpj) {
        this.cnpj = cnpj;

        return this;
    }

    NotaFiscalBuilder com(ItemDaNota item) {
        todosItens.add(item);
        valorBruto += item.getValor();
        impostos += item.getValor() * 0.05;

        return this;
    }
}
```

```
NotaFiscal build() {  
    return new NotaFiscal(razaoSocial, cnpj, data, valorBruto, impostos,  
                           todosItens, observacoes  
    );  
}
```



```
@Test
public void constroiNotaFiscal() {
    NotaFiscal notaFiscal = new NotaFiscalBuilder()
        .paraEmpresa("Razao Social")
        .comCnpj("999.999.999-99")
        .naData(Calendar.getInstance())
        .com(new ItemDaNota("nota1", 50d))
        .com(new ItemDaNota("nota2", 50d))
        .comObservacoes("observação da nota")
        .build();

    assertEquals(notaFiscal.getItems().size(), 2);
}
```



```
@Test
public void constroiNotaFiscal() {
    ArrayList<ItemDaNota> itemDaNotas = new ArrayList<>();
    itemDaNotas.add(new ItemDaNota("nota1", 50d));
    itemDaNotas.add(new ItemDaNota("nota2", 50d));
    double valorTotal = 0d;
    for (ItemDaNota item : itemDaNotas) {
        valorTotal += item.getValor();
    }
    double imposto = valorTotal * 0.05d;

    NotaFiscal notaFiscal = new NotaFiscal(
        "razaoSocial",
        "999.999.999-99",
        Calendar.getInstance(),
        valorTotal,
        imposto,
        itemDaNotas,
        "observação da nota"
    );

    assertEquals(notaFiscal.getItens().size(), 2);
}
```

```
@Test
public void constroiNotaFiscal() {
    NotaFiscal notaFiscal = new NotaFiscalBuilder()
        .paraEmpresa("Razao Social")
        .comCnpj("999.999.999-99")
        .naData(Calendar.getInstance())
        .com(new ItemDaNota("nota1", 50d))
        .com(new ItemDaNota("nota2", 50d))
        .comObservacoes("observação da nota")
        .build();

    assertEquals(notaFiscal.getItens().size(), 2);
}
```


Builder - Quando usar?

- Objeto complexo de ser criado;
- Que possui diversos atributos;
- Que possui uma lógica de criação complicada.

Factory Method

```
class Programa {  
    public static void main(String[] args) {  
        try {  
            Connection c = DriverManager.getConnection("jdbc:mysql://localhost/banco", "root", "1234");  
            PreparedStatement ps = c.prepareStatement("select * from tabela");  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
}
```



```
class ConnectionFactory {  
    static Connection getConnection() {  
        try {  
            return DriverManager.getConnection("jdbc:mysql://localhost/banco", "root", "1234");  
        } catch (SQLException e) {  
            e.printStackTrace();  
            return null;  
        }  
    }  
}
```

```
class Programa {  
    public static void main(String[] args) throws SQLException {  
        Connection c = ConnectionFactory.getConnection();  
        if (c != null) {  
            PreparedStatement ps = c.prepareStatement("select * from tabela");  
        }  
    }  
}
```

Factory - Quando usar?

- Quando temos que isolar o processo de criação de um objeto em um único lugar;
- Geralmente ela não precisa de muitas informações para criar o objeto como o *Builder*;
- Mas Pode retornar diferentes instâncias.

Strategy


```
import static org.junit.Assert.assertEquals;  
  
public class CalculadorDeImpostoTest {  
  
    @Test  
    public void calculaIcms() {  
        final Orcamento orcamento = new Orcamento(20d);  
        Double resultado = new CalculadorDeImposto().realizaCalculo(orcamento, "ICMS");  
        Double resultadoEsperado = 2d;  
        assertEquals(resultado, resultadoEsperado);  
    }  
  
    @Test  
    public void calculaIss() {  
        final Orcamento orcamento = new Orcamento(50d);  
        Double resultado = new CalculadorDeImposto().realizaCalculo(orcamento, "ISS");  
        Double resultadoEsperado = 3d;  
        assertEquals(resultado, resultadoEsperado);  
    }  
  
}
```



```
class CalculadorDeImposto {  
    Double realizaCalculo(Orcamento orcamento, String imposto) {  
        if (imposto.equals("ICMS")) {  
            final Double icms = orcamento.getValor() * 0.1;  
            System.out.println(icms);  
            return icms;  
        } else if (imposto.equals("ISS")) {  
            final Double iss = orcamento.getValor() * 0.06;  
            System.out.println(iss);  
            return iss;  
        } else {  
            return null;  
        }  
    }  
}
```



```
class CalculadorDeImposto {  
  
    Double realizaCalculo(Orcamento orcamento, String imposto) {  
        if (imposto.equals("ICMS")) {  
            final Double icms = orcamento.getValor() * 0.1; ←  
            System.out.println(icms);  
            return icms;  
        } else if (imposto.equals("ISS")) {  
            final Double iss = orcamento.getValor() * 0.06; ←  
            System.out.println(iss);  
            return iss;  
        } else {  
            return null;  
        }  
    }  
}
```



```
interface Imposto {  
  
    Double calcula(final Orcamento orcamento);  
  
}
```

```
class ICMS implements Imposto {  
  
    @Override  
    public Double calcula(Orcamento orcamento) {  
        return orcamento.getValor() * 0.1;  
    }  
  
}
```

```
class ISS implements Imposto {  
  
    @Override  
    public Double calcula(Orcamento orcamento) {  
        return orcamento.getValor() * 0.06;  
    }  
  
}
```



```
class CalculadorDeImposto {  
  
    Double realizaCalculo(Orcamento orcamento, Imposto imposto) {  
        return imposto.calcula(orcamento);  
    }  
  
}
```

```
@Test  
public void calculaIcms() {  
    ICMS icms = new ICMS();  
    final Orcamento orcamento = new Orcamento(20d);  
    Double resultado = new CalculadorDeImposto().realizaCalculo(orcamento, icms);  
    Double resultadoEsperado = 2d;  
    assertEquals(resultado, resultadoEsperado);  
}
```

Strategy - Quando usar?

- Quando temos um conjunto de algoritmos similares, e precisamos alternar entre eles;

Template Method


```
class ICPP implements Imposto {  
    @Override  
    public Double calcula(Orcamento orcamento) {  
        if(orcamento.getValor() > 500) {  
            return orcamento.getValor() * 0.07;  
        } else {  
            return orcamento.getValor() * 0.05;  
        }  
    }  
}
```

```
class IKCV implements Imposto {  
    @Override  
    public Double calcula(Orcamento orcamento) {  
        if(orcamento.getValor() > 500 && temItemMaiorQuem100Reais(orcamento)) {  
            return orcamento.getValor() * 0.10;  
        } else {  
            return orcamento.getValor() * 0.06;  
        }  
    }  
}
```



```
class ICPP implements Imposto {  
    @Override  
    public Double calcula(Orcamento orcamento) {  
        if(orcamento.getValor() > 500) { ←  
            return orcamento.getValor() * 0.07;  
        } else {  
            return orcamento.getValor() * 0.05;  
        }  
    }  
}
```

```
class IKCV implements Imposto {  
    @Override  
    public Double calcula(Orcamento orcamento) { ←  
        if(orcamento.getValor() > 500 && temItemMaiorQuem100Reais(orcamento)) {  
            return orcamento.getValor() * 0.10;  
        } else {  
            return orcamento.getValor() * 0.06;  
        }  
    }  
}
```



```
abstract class TemplateDeImpostoCondicional implements Imposto {  
  
    @Override  
    public Double calcula(Orcamento orcamento) {  
        if(isUtilizaMaiorTaxacao(orcamento)) {  
            return maximaTaxacao(orcamento);  
        } else {  
            return minimaTaxacao(orcamento);  
        }  
    }  
}  
  
    abstract Boolean isUtilizaMaiorTaxacao(Orcamento orcamento);  
  
    abstract Double maximaTaxacao(Orcamento orcamento);  
  
    abstract Double minimaTaxacao(Orcamento orcamento);  
  
}
```



```
class ICPP extends TemplateDeImpostoCondicional {  
    @Override  
    Boolean isUtilizaMaiorTaxacao(Orcamento orcamento) {  
        return orcamento.getValor() > 500;  
    }  
  
    @Override  
    Double maximaTaxacao(Orcamento orcamento) {  
        return orcamento.getValor() * 0.07;  
    }  
  
    @Override  
    Double minimaTaxacao(Orcamento orcamento) {  
        return orcamento.getValor() * 0.05;  
    }  
}
```


Template Method- Quando usar?

- Quando temos diferentes algoritmos que possuem estruturas parecidas.



Chain of Responsibility

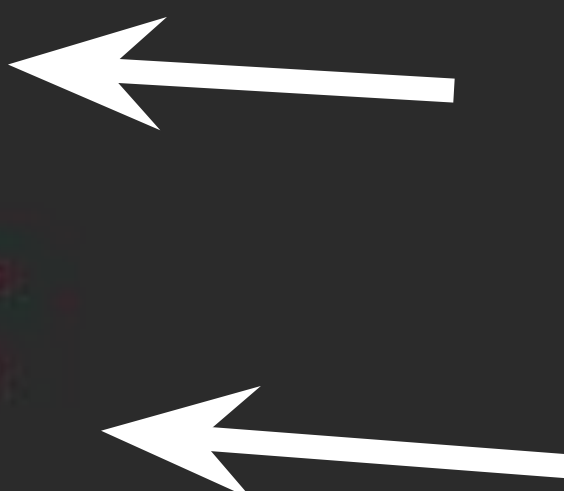
```
public class CalculadorDeDescontosTest {  
  
    @Test  
    public void calculaDesconto() {  
        Orcamento orcamento = new Orcamento(50);  
        CalculadorDeDescontos calculadorDeDescontos = new CalculadorDeDescontos();  
        Double resultado = calculadorDeDescontos.calcular(orcamento);  
        Double resultadoEsperado = 0d;  
        assertEquals(resultado, resultadoEsperado);  
    }  
  
}
```



```
class CalculadorDeDescontos {  
    Double calcular(Orcamento orcamento) {  
        if (orcamento.getItems().size() > 5) {  
            return orcamento.getValor() * 0.1;  
        } else if (orcamento.getValor() > 500) {  
            return orcamento.getValor() * 0.07;  
        }  
  
        return 0d;  
    }  
}
```



```
class CalculadorDeDescontos {  
    Double calcular(Orcamento orcamento) {  
        if (orcamento.getItems().size() > 5) {  
            return orcamento.getValor() * 0.1;  
        } else if (orcamento.getValor() > 500) {  
            return orcamento.getValor() * 0.07;  
        }  
  
        return 0d;  
    }  
}
```




```
interface Desconto {  
    Double desconta(Orcamento orcamento);  
    void setProximo(Desconto desconto);  
}
```

```
public class DescontoPorMaisDeCincoItens implements Desconto {  
    private Desconto proximo;  
  
    @Override  
    public Double desconta(Orcamento orcamento) {  
        if (orcamento.getItens().size() > 5) {  
            return orcamento.getValor() * 0.1;  
        } else {  
            return proximo.desconta(orcamento);  
        }  
    }  
  
    @Override  
    public void setProximo(Desconto proximo) { this.proximo = proximo; }  
}
```



```
class CalculadorDeDescontos {  
    Double calcular(Orcamento orcamento) {  
        Desconto d1 = new DescontoPorMaisDeCincoItens();  
        Desconto d2 = new DescontoPorMaisDeQuinhentosReais();  
        Desconto d3 = new SemDesconto();  
  
        d1.setProximo(d2);  
        d2.setProximo(d3);  
  
        return d1.desconta(orcamento);  
    }  
}
```

Chain - Quando usar?

- Quando temos uma lista de comandos a serem executados de acordo com algum cenário em específico;
- E quando sabemos qual o próximo cenário que deve ser validado, caso o anterior não satisfaça a condição.

Prova!

Prova de conceito

- Site <https://kahoot.com>
- Enter pin

Obrigado!

MAXIMA^{TECH}