02

Criação de objetos e o Builder

O mundo real é complexo. E essa complexidade é geralmente refletida nas classes do nosso sistema. Imagine uma Nota Fiscal: ela é composta de razão social, cnpj, data de emissão, valor bruto, imposto, desconto, valor liquido, ítens, observações e assim por diante.

Imagine essa classe, tendo atributos que representam cada uma das informações acima. Imagine também que essa classe receba todos esses atributos no construtor:

```
public class NotaFiscal {

    private String razaoSocial;
    private String cnpj;
    private Calendar dataDeEmissao;
    private double valorBruto;
    private double impostos;
    public List<ItemDaNota> itens;
    public String observacoes;

    public 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;
    }

    // getters

}
```

A classe que cria uma `NotaFiscal` utiliza esse construtor:

```
class Teste {
  public static void main(String[] args) {
    List<ItemDaNota> itens = // recupera os itens da nota
    double valorTotal = 0;
    for(ItemDaNota item : itens) {
      valorTotal += item.getValor();
    }
    double impostos = valorTotal * 0.05;

    NotaFiscal nf = new NotaFiscal("razao social qualquer", "um cnpj", Calendar.getInstance
  }
}
```

Veja que esse construtor é extenso e difícil de entender. Além do mais, a regra de criação do objeto acabou ficando espalhado pelo código do método `main()`.

Pior, as vezes ainda temos o problema de parâmetros opcionais. Nesse caso, o programador começa a criar diferentes construtores, cada um com uma possível combinação de parâmetros de entrada, tornando a sua classe mais difícil ainda de ser lida.

Para resolver o problema, separaremos a lógica da criação desse objeto complexo. Essa nova classe será responsável por pedir todos os parâmetros necessários, montar o que precisa, e enfim produzir uma Nota Fiscal, de forma com que o cliente dessa classe consiga entender o que está acontecendo.

Veja o código abaixo:

```
class Teste {
  public static void main(String[] args) {
    CriadorDeNotaFiscal criador = new CriadorDeNotaFiscal();
    criador.paraEmpresa("Caelum Ensino e Inovação");
    criador.comCnpj("23.456.789/0001-12");
    // e assim por diante...
  }
```

Veja que a classe `CriadorDeNotaFiscal` provê métodos mais amigáveis, e vai guardando esses dados. Vamos começar a implementar essa classe. Veja o código abaixo. Passar os dados de razão social e cnpj agora ficaram mais fáceis:

```
class CriadorDeNotaFiscal {
  private String razaoSocial;
  private String cnpj;

  public void paraEmpresa(String razaoSocial) {
    this.razaoSocial = razaoSocial;
  }

  public void comCnpj(String cnpj) {
    this.cnpj = cnpj;
  }
}
```

Vamos, por exemplo, resolver o problema do valor bruto da nota, a partir de cada item. Vamos criar um método `comItem()`, que receberá um ítem da nota. Esse método, ao receber o ítem, já vai acumulando o valor total, e calculando o imposto de 5%:

```
class CriadorDeNotaFiscal {
  private String razaoSocial;
  private String cnpj;
  private double valorTotal;
  private double impostos;
  private List<ItemDaNota> todosItens = new ArrayList<ItemDaNota>();

  public void paraEmpresa(String razaoSocial) {
    this.razaoSocial = razaoSocial;
  }
```

```
  public void comCnpj(String cnpj) {
    this.cnpj = cnpj;
  }

  public void comItem(ItemDaNota item) {
    todosItens.add(item);
    valorBruto += item.getValor();
    impostos += item.getValor() * 0.05;
  }
}

class Teste {

  public static void main(String[] args) {
    CriadorDeNotaFiscal criador = new CriadorDeNotaFiscal();
    criador.paraEmpresa("Caelum");
    criador.comCnpj("123.456.789/0001-10");
    criador.comItem(new ItemDaNota("item 1", 100.0));
    criador.comItem(new ItemDaNota("item 2", 200.0));
    criador.comItem(new ItemDaNota("item 3", 300.0));
  }
}
```

Para completar, basta resolver o problema da data atual, e das observações. Podemos fazer da seguinte forma, continuando a implementação dessa `CriadorDeNotaFiscal`:

```
class CriadorDeNotaFiscal {
  private String razaoSocial;
  private String cnpj;
  private double valorTotal;
  private double impostos;
  private Calendar data;
  private String observacoes;

  private List<ItemDaNota> todosItens = new ArrayList<ItemDaNota>();

  public void comObservacoes(String observacoes) {
    this.observacoes = observacoes;
  }

  public void naDataAtual() {
    this.data = Calendar.getInstance();
  }

  // código continua aqui
}

class Teste {

  public static void main(String[] args) {
    CriadorDeNotaFiscal criador = new CriadorDeNotaFiscal();
    criador.paraEmpresa("Caelum");
    criador.comCnpj("123.456.789/0001-10");
    criador.comItem(new ItemDaNota("item 1", 100.0));
    criador.comItem(new ItemDaNota("item 2", 200.0));
    criador.comItem(new ItemDaNota("item 3", 300.0));
```

```
    criador.comObservacoes("entregar nf pessoalmente");
    criador.naDataAtual();
  }
}
```

Temos agora maneiras elegantes de passar os dados da nota fiscal. Falta só, após passarmos todas as informações, gerar a nota! Para isso, adicionamos um último método em nosso builder, que juntará todas essas informações e criará a nota fiscal.

```
class CriadorDeNotaFiscal {
  private String razaoSocial;
  private String cnpj;
  private double valorBruto;
  private double impostos;
  private Calendar data;
  private String observacoes;

  private List<ItemDaNota> todosItens = new ArrayList<ItemDaNota>();

  public NotaFiscal constroi() {
    return new NotaFiscal(razaoSocial, cnpj, data, valorBruto, impostos, todosItens, observa
  }

  // código continua aqui
}

class Teste {

  public static void main(String[] args) {
    CriadorDeNotaFiscal criador = new CriadorDeNotaFiscal();
    criador.paraEmpresa("Caelum");
    criador.comCnpj("123.456.789/0001-10");
    criador.comItem(new Item("item 1", 100.0));
    criador.comItem(new Item("item 2", 200.0));
    criador.comItem(new Item("item 3", 300.0));
    criador.comObservacoes("entregar nf pessoalmente");
    criador.naDataAtual();

    NotaFiscal nf = criador.constroi();
  }
}
```

Veja que agora a regra de criação de objeto NotaFiscal, que antes estava espalhada pelo código (estava jogado na `main()`), agora está centralizado em uma classe que só tem isso como responsabilidade. Além do código estar em um lugar só, essa classe ainda provê uma interface bem clara e simples de ser usada, facilitando a vida das classes que irão utilizá-la.

Para classes cuja responsabilidade é a de criar outros objetos, geralmente complexos como o do exemplo acima, damos o nome de Builder. Ela esconde toda a lógica da criação de objetos complexos. Vamos renomear a classe:

```
class NotaFiscalBuilder {
  // código aqui da antiga CriadorDeNotaFiscal
}
```

Esse código ainda pode ser melhorado. Repare em nossa `main()`, que repetimos várias vezes o código `builder.`. Isso faz com o que o código fique repetitivo e difícil de ser lido. Podemos alterar isso, e fazer com que esse trecho de código seja escrito apenas uma vez. Por exemplo:

```
class Teste {

  public static void main(String[] args) {
    NotaFiscal nf = new NotaFiscalBuilder().paraEmpresa("Caelum")
                      .comCnpj("123.456.789/0001-10")
                      .comItem(new Item("item 1", 100.0))
                      .comItem(new Item("item 2", 200.0))
                      .comItem(new Item("item 3", 300.0))
                      .comObservacoes("entregar nf pessoalmente")
                      .naDataAtual()
                      .constroi();
  }
}
```

Veja que fomos invocando um método atrás do outro. Mas como isso é possível? Basta trocarmos o retorno de todos os métodos do builder: de `void` para `NotaFiscalBuilder`. Veja que, se todo método devolver `NotaFiscalBuilder`, poderemos chamar o próximo método logo em seguida:

```
class NotaFiscalBuilder {
  private String razaoSocial;
  private String cnpj;
  private double valorTotal;
  private double impostos;
  private Calendar data;
  private String observacoes;

  private List<ItemDaNota> todosItens = new ArrayList<ItemDaNota>();

  public NotaFiscalBuilder paraEmpresa(String razaoSocial) {
    this.razaoSocial = razaoSocial;
    return this; // retorno eu mesmo, o próprio builder, para que o cliente continue utiliz
  }

  public NotaFiscalBuilder comCnpj(String cnpj) {
    this.cnpj = cnpj;
    return this;
  }

  public NotaFiscalBuilder comItem(ItemDaNota item) {
    todosItens.add(item);
    valorBruto += item.getValor();
    impostos += item.getValor() * 0.05;
    return this;
  }

  // código continua aqui com a mesma ideia
  // substituindo void por NotaFiscalBuilder e retornando this em todos eles...
}
```

Veja que retornamos `this`. Ou seja, os métodos do Builder agora guardam a informação, e retorna ele mesmo. Nosso builder tem agora uma interface fluente. A vantagem de usar method chaining, nome da técnica onde conseguimos invocar um método atrás do outro, é justamente a clareza do código e a eliminação do uso da variável `criador` (`builder`) repetidas vezes.

Objetos complexos existem e vão continuar existindo. Caso ele seja complexo, o desenvolvedor deve pensar em prover um Builder para o objeto, e facilitar a vida das classes que precisam instanciar essas classes complexas.