02

Executando diferentes ações e o Observer

Imagine que, após todo o processo de geração de notas fiscais de um sistema, ainda é necessário enviar a nota fiscal por e-mail para o cliente, salvar no banco de dados, enviar por SMS e ainda imprimi-la.

Em uma implementação mais tradicional teríamos, logo após o processo de geração da nota fiscal, uma sequência de métodos que fariam cada uma das atividades. Podemos implementar cada uma dessas atividades em métodos privados, dentro da classe `NotaFiscalBuilder`, que é a responsável por gerar a NotaFiscal.

```
class NotaFiscalBuilder {
  // classe aqui

  private void enviaPorEmail(NotaFiscal notaFiscal) {
    System.out.println("enviando por e-mail");
  }

  private void salvaNoBanco(NotaFiscal notaFiscal) {
    System.out.println("salvando no banco");
  }

  private void enviaPorSms(NotaFiscal notaFiscal) {
    System.out.println("enviando por sms");
  }

  private void imprime(NotaFiscal notaFiscal) {
    System.out.println("imprimindo notaFiscal");
  }
}
```

(Por questões de simplicidade, substituímos os códigos que deveriam mandar e-mail, persistir no banco etc, por simples `System.out.println()`. Isso não vai nos atrapalhar, mas imagine o tempo todo que o código ali é complexo.)

Tendo esses métodos implementados basta invocá-los logo após a criação da NotaFiscal no método `constroi()`:

```
class NotaFiscalBuilder {

  public NotaFiscal constroi() {
    NotaFiscal notaFiscal = new NotaFiscal(razaoSocial, cnpj, valorTotal, impostos, data,

    // invocando as ações posteriores
    enviaPorEmail(notaFiscal);
    salvaNoBanco(notaFiscal);
    enviaPorSms(notaFiscal);
    imprime(notaFiscal);

              return notaFiscal;
  }

  // resto da classe aqui
}
```

Esse código apresenta muitos problemas. O primeiro deles é que essa classe já é complexa, e sua complexidade só tende a piorar. Ela faz muitas coisas diferentes: manda e-mail, persiste no banco de dados, e assim por diante.

Uma primeira melhoria nele seria extrair as responsabilidades para diferentes classes. Ou seja, uma classe responsável somente por persistir no banco de dados, uma somente responsável por enviar e-mails e assim por diante. E, por fim, o `NotaFiscalBuilder`, ao invés de conter todo o código, reutilizar o código escrito nessas classes especialistas. Veja o código abaixo:

```
class NotaFiscalBuilder {
  // código aqui...

  public NotaFiscal constroi() {
    NotaFiscal notaFiscal = new NotaFiscal(razaoSocial, cnpj, valorTotal, impostos, data,

    new EnviadorDeEmail().enviaPorEmail(notaFiscal);
    new NotaFiscalDao().salvaNoBanco(notaFiscal);
    new EnviadorDeSms().enviaPorSms(notaFiscal);
    new Impressora().imprime(notaFiscal);
  }
}

public class EnviadorDeEmail {

  public void enviaPorEmail(NotaFiscal notaFiscal) {
    System.out.println("enviando por e-mail");
  }
}

public class NotaFiscalDao {
  public void salvaNoBanco(NotaFiscal notaFiscal) {
    System.out.println("salvando no banco");
  }
}

public class EnviadorDeSms {
  public void enviaPorSms(NotaFiscal notaFiscal) {
    System.out.println("enviando por sms");
  }
}

public class Impressora {
  public void imprime(NotaFiscal notaFiscal) {
    System.out.println("imprimindo notaFiscal");
  }
}
```

Melhoramos o código. As classes `EnviadorDeEmail`, `NotaFiscalDao`, `EnviadorDeSms`, e `Impressora` agora tem somente uma única responsabilidade. Precisamos nos concentrar na classe NotaFiscalBuilder, pois ela ainda é problemática. Repare no método `constroi()`. Ele utiliza muitas outras classes para realizar sua tarefa. Ele é altamente acoplado a essas classes!

A grande questão é: como diminuir o acoplamento entre o método `constroi()`, que precisa invocar essa lista de ações, e as classes que realizam essas ações?

Para começar, vamos encontrar algo em comum entre todas as ações que acontecem após a nota ser gerada: todas elas fazem algo com a Nota Fiscal logo após ela ser gerada. Podemos então criar uma interface para representar esse comportamento em comum:

```
interface AcaoAposGerarNota {
  void executa(NotaFiscal notaFiscal);
}

public class EnviadorDeEmail implements AcaoAposGerarNota {

  public void executa(NotaFiscal notaFiscal) {
    System.out.println("enviando por e-mail");
  }
}

public class NotaFiscalDao implements AcaoAposGerarNota {
  public void executa(NotaFiscal notaFiscal) {
    System.out.println("salvando no banco");
  }
}

public class EnviadorDeSms implements AcaoAposGerarNota {
  public void executa(NotaFiscal notaFiscal) {
    System.out.println("enviando por sms");
  }
}

public class Impressora implements AcaoAposGerarNota {
  public void executa(NotaFiscal notaFiscal) {
    System.out.println("imprimindo notaFiscal");
  }
}
```

Todas essas ações agora são `AcaoAposGerarNota`. Repare que, para o método `constroi()`, pouco importa qual ação está sendo executada. Ele simplesmente dispara uma ou mais ações.

Com essa informação, podemos reescrever nosso método, fazendo com que ele dispare uma lista de ações, independente de quais ações:

```
public class NotaFiscalBuilder {
  List<AcaoAposGerarNota> todasAcoesASeremExecutadas;

public NotaFiscal constroi() {

  NotaFiscal notaFiscal = new NotaFiscal(razaoSocial, cnpj, valorTotal, impostos, data, ol

  for(AcaoAposGerarNota acao : todasAcoesASeremExecutadas) {
    acao.executa(notaFiscal);
  }
}
}
```

Repare que agora o método `constroi()` não se importa com quais ações serão executadas. Ele simplesmente notifica essas ações. Precisamos só preencher essa lista com as ações que devem ser notificadas. Em nosso `NotaFiscalBuilder`, podemos criar um método para receber esses notificadores e guardar nessa lista:

```
public class NotaFiscalBuilder {
  private List<AcaoAposGerarNota> todasAcoesASeremExecutadas;

  public NotaFiscalBuilder() {
    this.todasAcoesASeremExecutadas = new ArrayList<AcaoAposGerarNota>();
  }

  public void adicionaAcao(AcaoAposGerarNota novaAcao) {
    this.todasAcoesASeremExecutadas.add(novaAcao);
  }

  // codigo continua aqui...

}
```

Agora todas as ações devem ser adicionadas nessa lista para que, assim que a nota fiscal for gerada, essas ações sejam executadas.

Quando temos classes que serão notificadas sobre alguma coisa (no nosso caso, notificadas sobre a geração de uma nota fiscal) e um notificador que, assim que executa uma ação, notifica todos que estão na lista sobre o evento ocorrido, implementamos o padrão de projeto conhecido por Observer.

Veja um código de exemplo, que utiliza esse Observer:

```
public class TesteAcao {
  public static void main(String[] args) {
    NotaFiscalBuilder builder = new NotaFiscalBuilder();
    builder.adicionaAcao(new EnviadorDeEmail());
    builder.adicionaAcao(new NotaFiscalDao());
    builder.adicionaAcao(new EnviadorDeSms());
    builder.adicionaAcao(new Impressora());

    NotaFiscal notaFiscal = builder.paraEmpresa("Caelum")
                    .comCnpj("123.456.789/0001-10")
                    .comItem(new ItemDaNota("item 1", 100.0))
                    .comItem(new ItemDaNota("item 2", 200.0))
                    .comItem(new ItemDaNota("item 3", 300.0))
                    .comObservacoes("entregar notaFiscal pessoalmente")
                    .naDataAtual()
                    .constroi();
  }
}
```

Veja como é fácil agora adicionar novas ações após a geração da nota. Basta adicionar um novo "observador" ou, no nosso caso, uma nova Ação Após Gerar Nota.

O Observer desacopla seu código e possibilita que seu código execute diferentes ações após algum evento. Além disso, como o código acima demonstra, criar e executar novas ações é uma tarefa fácil agora, facilitando a manutenção e evolução desse trecho de código.