01

Lidando com o acoplamento

Transcrição

No capítulo 2, migramos a funcionalidade de registrar um pagamento, da classe `Divida` para a classe `Pagamentos`.

```
public class Pagamentos {
  private double valorPago;
    // ...
  public void registra(Pagamento pagamento) {
        double valor = pagamento.getValor();
        if (valor < 0) {
            throw new IllegalArgumentException("Valor invalido para pagamento");
        }
        if (valor > 100) {
            valor = valor - 8;
        }
        this.valorPago += valor;
        this.add(pagamento);
    }
}

public class Divida {
    private double total;
    private String credor;
    private Cnpj cnpjCredor;
    private Pagamentos pagamentos = new Pagamentos();
    // ...
}
```

De modo que um objeto `Divida` "sabe" o quanto já foi pago consultando seu atributo `Pagamentos pagamentos`, como, por exemplo, no método `double valorAPagar()`.

```
public class Divida {
    private double total;
    private Pagamentos pagamentos = new Pagamentos();
    // ...

  public double valorAPagar() {
     return this.total - this.pagamentos.getValorPago();
    }
}
```

Considerando que o registro de pagamentos foi delegado para a classe `Pagamentos`, nosso `BalancoEmpresa`, quando precisa registrar o pagamento refrente à dívida de um credor, utiliza a funcionalidade presente no método `void pagaDivida(Cnpj cnpjCredor, Pagamento pagamento)`.

```
public class BalancoEmpresa {
    private HashMap<Cnpj, Divida> dividas = new HashMap<Cnpj, Divida>();
    // ...
```

```
    public void pagaDivida(Cnpj cnpjCredor, Pagamento pagamento) {
            Divida divida = dividas.get(cnpjCredor);
            if (divida != null) {
                    // precisamos saber que o objeto divida tem um atributo pagamentos
                    // que por sua vez oferece o método que registra um pagamento
                    divida.getPagamentos().registra(pagamento);
            }
    }
}
```

Perceba, no entanto, que, para registrar o pagamento de uma dívida, precisamos entender que a classe `Divida` possui um atributo do tipo `Pagamentos` e que esse oferece o método `void registra(Pagamento pagamento)`. Isso fere o princípio "Tell, Don't Ask" apresentado no primeiro capítulo pois deveríamos simplesmente dizer para o objeto `Divida` registrar o pagamento, utilizando seu comportamento mas sem acoplarmos este código a detalhes de sua implementação. Essa situação, em que chamamos a funcionalidade que queremos em um membro do objeto com o qual estamos "conversando", neste caso a `divida`, deu origem à Lei de Demeter (http://en.wikipedia.org/wiki/Law_of_Demeter), uma regra que, justamente, nos convida a evitar este tipo de situação. Segundo a lei, você não deve fazer invocações do tipo `a().b().c()...`; você só deve "avançar um nível" nesse objeto, ou seja, no máximo `a().b();`.

Podemos melhorar nosso design, como segue:

```
public class BalancoEmpresa {
    private HashMap<Cnpj, Divida> dividas = new HashMap<Cnpj, Divida>();
        // ...

    public void pagaDivida(Cnpj cnpjCredor, Pagamento pagamento) {
        Divida divida = dividas.get(cnpjCredor);
        if (divida != null) {
            divida.registra(pagamento); // O objeto Divida registra o pagamento
        }
    }
}
```

Note que a implementação do método `void pagaDivida(...)` agora não precisa saber que, internamente, um objeto `Divida` gerencia os pagamentos por meio de um atributo do tipo `Pagamentos`. Veja como fica o método `void registra(Pagamento pagamento)`, agora oferecido por nossa classe `Divida`.

```
public class Divida {
    ...
    private Pagamentos pagamentos = new Pagamentos();

  ...
  public void registra(Pagamento pagamento) {
            // a classe agora delega o registro de um pagamento para seu atributo pagamentos
            pagamentos.registra(pagamento);
    }
}
```

Veja que, agora, a classe `Divida` delega o registro de um pagamento para o seu atributo privado `Pagamentos pagamentos`, chamando nele o seu método `void registra(...)`. Agora, a dinâmica de registro de pagamentos de uma

`Divida` está definida internamente em seu método `void registra(...)` e, num momento futuro, se for necessário alterarmos o registro de pagamento de uma dívida, poderemos fazê-lo precisamente na implementação deste método.

Note que a classe `BalancoEmpresa` registra o pagamento de uma dívida da maneira como esperamos que seja feita, ou seja, utilizando o método `void registra(Pagamento pagamento)` oferecido pelo objeto `Divida`. Entretanto, observando a classe `GerenciadorDeDivida`, percebemos que é utilizado outro mecanismo.

```
public class GerenciadorDeDividas {
    public void efetuaPagamento(Divida divida, Pagamento pagamento) {
        divida.getPagamentos().registra(pagamento);
    }
}
```

O método `void efetuaPagamento(...)` acessa pelo método `Pagamentos getPagamentos()` o atributo `pagamentos` do objeto `Divida` e, nele, chama o método `registra()`. Esta implementação de registro de pagamento permanece acoplada ao fato de que internamente uma dívida possui um atributo `Pagamentos pagamentos` que sabe registrar um pagamento.

Para obrigarmos o código que utiliza nossa classe `Divida` a registrar um pagamento da maneira que desejamos, ou seja, a utilizar o método `void registra(Pagamento pagamento)`, podemos melhorar o encapsulamento do atributo `pagamentos`. Para esta finalidade, podemos neste momento simplesmente remover o método `Pagamentos getPagamentos()`.

```
public class Divida {
  // ...
  // sem getter ou setter, o atributo pagamentos está inteiramente encapsulado
    private Pagamentos pagamentos = new Pagamentos();

  // para registrar um pagamento, deve ser usado o método registra desta classe
  public void registra(Pagamento pagamento) {
        pagamentos.registra(pagamento);
    }
}
```

Devemos sempre refletir sobre como uma classe se apresenta para o mundo exterior, ou seja, para as outras classes que a utilizarão. Este trabalho é feito ao definirmos a interface pública de nossa classe, isto é, os métodos e (muito raramente) atributos que deixaremos visíveis para que o restante da aplicação acesse a funcionalidade de nossa classe. É importante notar que a interface pública de nossas classes deve, desde o primeiro momento, ser bem pensada, já que mudanças na maneira como uma classe se apresenta ao mundo exterior tendem a trazer grandes problemas. Por exemplo, a retirada do método `Pagamentos getPagamentos()` da classe `Divida` vai gerar problemas na classe `GerenciadorDeDividas`, cujo código deixará de compilar e precisará ser adequado.

```
public class GerenciadorDeDividas {
    public void efetuaPagamento(Divida divida, Pagamento pagamento) {
    // classe Divida nao oferece mais o método getPagamentos()
    // o código abaixo não compila
        divida.getPagamentos().registra(pagamento);
    }
}
```

Precisaremos corrigir o método `void efetuaPagamento(...)` para utilizar corretamente a interface pública, ou seja, os métodos públicos da classe `Divida`, como segue.

```
public class GerenciadorDeDividas {
    public void efetuaPagamento(Divida divida, Pagamento pagamento) {
    // agora adequamos o código para acessarmos o método publico registra(...) oferecido pe
    // o código compila normalmente
        divida.registra(pagamento);
    }
}
```

Nosso exemplo é composto por poucas classes. Perceba, no entanto, que um sistema feito em Java frequentemente é composto de milhares de classes. Uma pequena alteração na interface pública de nossa classe `Divida`, nesse caso a retirada de um método público, repercutiu em outras 2 classes no nosso pequeno exemplo: `BalancoEmpresa` e `GerenciadorDeDividas`. Imagine, numa aplicação do dia-a-dia, qual seria a repercussão de uma alteração deste gênero. A quantidade de refatorações necessárias poderá ser muito grande ou, até mesmo, inviável, caso a classe cujo método foi removido fizer parte de uma biblioteca usada por outras empresas. Por isso, devemos sempre refletir muito antes de oferecer métodos públicos em nossas classes. Note que nem falamos em refletir sobre atributos públicos, pois seu uso é completamente desencorajado. Deixar atributos públicos dificulta muito o encapsulamento dos comportamentos da classe.

Um outro exemplo: relatório de dívidas

Suponha que agora é necessário gerarmos um relatório que exiba as informações de uma dívida da empresa onde as informações devem estar propriamente formatadas. Para formatarmos os valores `double` utilizaremos a classe `NumberFormat` do Java, de modo que nosso `RelatorioDeDivida` deverá contar com um método `void geraRelatorio()` como segue:

```
public class RelatorioDeDivida {

    private Divida divida;
  ...

    public void geraRelatorio() {
        System.out.println("Credor: " + divida.getCredor());
        System.out.println("Cnpj credor: " + divida.getCnpjCredor());

    NumberFormat formatadorDeNumero = NumberFormat.getCurrencyInstance(new Locale("pt", "BR'

    // agora utilizamos uma instância da classe NumberFormat para fazer a exibição dos valo
        System.out.println("Valor a pagar: " + formatadorDeNumero.format(divida.getValorAPa
        System.out.println("Valor total: " + formatadorDeNumero.format(divida.getTotal()));
    }
}
```

A primeira coisa a observarmos nesta implementação é que o objeto `RelatorioDeDivida` já recebe em seu construtor a `Divida` para a qual deverá gerar um relatório. No método `void geraRelatorio()`, quando queremos imprimir o Cnpj do credor, fazemos diretamente a concatenação `"Cnpj credor: " + divida.getCnpjCredor()` de uma `String` com um

objeto `Cnpj`. Isso gera, na impressão dessa linha, algo como br.com.caelum.Cnpj@a8426bc2. Para termos uma exibição melhor do `Cnpj` da Divida podemos chamar seu método `getValor()`, como segue.

```
public class RelatorioDeDivida {

    private Divida divida;
  ...

    public void geraRelatorio() {
    ...
    System.out.println("Cnpj credor: " + divida.getCnpjCredor().getValor());
    ...
    }
}
```

No entanto, perceba que dessa forma estamos acoplando a implementação de nosso `RelatorioDeDivida` à implementação da classe `Cnpj`, pois o código do método `void geraRelatorio()` deve saber que a classe `Cnpj` oferece o método público `String getValor()`. Isso fere novamente a Lei de Demeter.

Levando em conta que todo objeto em Java já possui o método `String toString()` que tem a funcionalidade de convertê-lo para uma `String`, podemos sobrescrevê-lo. Isso livrará o método `geraRelatorio` de se acoplar a implementação específica da classe `Cnpj`. Dessa forma, devemos fazer na classe `Cnpj` a sobrescrita do método `String toString()`. Assumindo que o atributo privado `String valor` da classe `Cnpj` já estará devidamente formatado, podemos dar a seguinte implementação.

```
public class Cnpj {
    private String valor;
  ...

    @Override
    public String toString() {
        return this.valor;
    }
}
```

Agora, nosso método `geraRelatorio` já é capaz de dar uma exibição melhor para cada uma das características que julgamos importantes de uma `Divida`.

Para formatarmos o "valor a pagar" e o "valor total", que são valores do tipo double, utilizamos um formatador do tipo `NumberFormat`. Perceba que o objeto `NumberFormat formatadorDeNumero` foi obtido por meio de um método estático `getCurrencyInstance`, que aceita como argumento um `Locale` que representa as características textuais de uma determinada língua em uma região geográfica.

Da maneira como fizemos o método `void geraRelatorio()`, a impressão de nosso `RelatorioDeDivida` ficará sempre restrita a formatação brasileira para valores monetários. Note que, no método `void geraRelatorio()`, estamos nos acoplando a uma determinada implementação de formatador. Se quiséssemos gerar um relatório com a formatação monetária vigente nos Estados Unidos, por exemplo, teríamos que recompilar a classe com a seguinte alteração.

```
public class RelatorioDeDivida {

    private Divida divida;
```

```
...

  public void geraRelatorio() {
      System.out.println("Credor: " + divida.getCredor());
      System.out.println("Cnpj credor: " + divida.getCnpjCredor());

  // agora nosso formatador está dirigido para a formatação monetária dos Estados Unidos
  NumberFormat formatadorDeNumero = NumberFormat.getCurrencyInstance(new Locale("en", "US"

      System.out.println("Valor a pagar: " + formatadorDeNumero.format(divida.getValorAPag
      System.out.println("Valor total: " + formatadorDeNumero.format(divida.getTotal()));
  }
}
```

Se, a cada vez que precisarmos de uma formatação diferente para os valores do relatório, precisarmos recompilar a classe `RelatorioDeDivida`, isso significa que ela está muito acoplada com a implementação de formatação. Dessa forma, a manutenabilidade do nosso sistema estará comprometida. Como poderíamos alterar esta classe de maneira que seu uso fosse mais flexível? Pense que o formatador não precisa ser obtido dentro do método `geraRelatorio`. Este formatador pode ser decidido no momento em que quisermos gerar o relatório. Podemos enviá-lo na chamada do método `geraRelatorio`, como neste trecho de código que serve de exemplo.

```
...

Divida divida = new Divida();
    divida.setCredor("Credor 1");
    divida.setCnpjCredor(new Cnpj("00.000.000/0001-01"));
    divida.setTotal(3000);

NumberFormat formatadorDeNumero = NumberFormat.getCurrencyInstance(new Locale("en", "US")

// passamos o formatadorDeNumero como argumento na chamada do método que gera o relatório
    new RelatorioDeDivida(divida).geraRelatorio(formatadorDeNumero);

...
```

Dessa maneira, no momento em que decidirmos gerar o relatório, decidimos a implementação de formatador que será utilizada. Este é um uso bem mais flexível da classe `RelatorioDeDivida`. Vamos fazer a adequação no método `geraRelatorio()` para podermos utilizá-lo desta forma.

```
public class RelatorioDeDivida {

  private Divida divida;
...

  public void geraRelatorio(NumberFormat formatadorDeNumero) {
      System.out.println("Credor: " + divida.getCredor());
      System.out.println("Cnpj credor: " + divida.getCnpjCredor());

  // utilizamos o formatador recebido como parâmetro do método
      System.out.println("Valor a pagar: " + formatadorDeNumero.format(divida.getValorAPag
      System.out.println("Valor total: " + formatadorDeNumero.format(divida.getTotal()));
```

```
        }
    }
```

Pronto! Agora o `RelatorioDeDivida` independe do algoritmo de formatação de número! Muito mais flexibilidade!