01

Melhorando a coesão de nossas classes

Transcrição

A ideia de passar responsabilidades para a classe `Divida` deixa nosso código mais encapsulado. Agora, tudo que é relacionado aos atributos dessa classe está nela mesma. Além disso, se precisarmos mudar alguma lógica relacionada a dívidas, sabemos onde mexer.

Vamos incrementar um pouco mais nosso sistema. Colocamos uma validação e alguns métodos auxiliares no CNPJ do credor e permitimos algumas filtragens nos pagamentos.

```
public class Divida {
  private double total;
  private double valorPago;
  private String credor;
  private String cnpjCredor;
  private ArrayList<Pagamento> pagamentos = new ArrayList<Pagamento>();

  public boolean cnpjValido() {
    return primeiroDigitoVerificadorDoCnpj() == primeiroDigitoCorretoParaCnpj()
        && segundoDigitoVerificadorDoCnpj() == segundoDigitoCorretoParaCnpj();
  }
  public String getCnpjCredor() {
    return this.cnpjCredor;
  }
  public String getCredor() {
    return this.credor;
  }
  public double getTotal() {
    return this.total;
  }
  public double getValorPago() {
    return this.valorPago;
  }
  private void paga(double valor) {
    if (valor < 0) {
      throw new IllegalArgumentException("Valor invalido para pagamento");
    }
    if (valor > 100) {
      valor = valor - 8;
    }
    this.valorPago += valor;
  }
  public ArrayList<Pagamento> pagamentosAntesDe(Calendar data) {
    ArrayList<Pagamento> pagamentosFiltrados = new ArrayList<Pagamento>();
    for (Pagamento pagamento : this.pagamentos) {
      if (pagamento.getData().before(data)) {
        pagamentosFiltrados.add(pagamento);
      }
    }
    return pagamentosFiltrados;
  }
  public ArrayList<Pagamento> pagamentosComValorMaiorQue(double valorMinimo) {
```

```
    ArrayList<Pagamento> pagamentosFiltrados = new ArrayList<Pagamento>();
    for (Pagamento pagamento : this.pagamentos) {
      if (pagamento.getValor() > valorMinimo) {
        pagamentosFiltrados.add(pagamento);
      }
    }
    return pagamentosFiltrados;
  }
  public ArrayList<Pagamento> pagamentosDo(String cnpjPagador) {
    ArrayList<Pagamento> pagamentosFiltrados = new ArrayList<Pagamento>();
    for (Pagamento pagamento : this.pagamentos) {
      if (pagamento.getCnpjPagador().equals(cnpjPagador)) {
        pagamentosFiltrados.add(pagamento);
      }
    }
    return pagamentosFiltrados;
  }
  private int primeiroDigitoCorretoParaCnpj() {
    // Extrai o primeiro dígito verificador do CNPJ armazenado
    // no atributo cnpj
  }
  private int primeiroDigitoVerificadorDoCnpj() {
    // Extrai o segundo dígito verificador do CNPJ armazenado
    // no atributo cnpj
  }
  public void registra(Pagamento pagamento) {
    this.pagamentos.add(pagamento);
    paga(pagamento.getValor());
  }
  private int segundoDigitoCorretoParaCnpj() {
    // Calcula o primeiro dígito verificador correto para
    // o CNPJ armazenado no atributo cnpj
  }
  private int segundoDigitoVerificadorDoCnpj() {
    // Calcula o primeiro dígito verificador correto para
    // o CNPJ armazenado no atributo cnpj
  }
  public void setCnpjCredor(String cnpjCredor) {
    this.cnpjCredor = cnpjCredor;
  }
  public void setCredor(String credor) {
    this.credor = credor;
  }
  public void setTotal(double total) {
    this.total = total;
  }
  public double valorAPagar() {
    return this.total - this.valorPago;
  }
}
```

Mas será que agora nossa classe `Divida` não tem responsabilidades demais? Veja quantos métodos ela possui! Vemos, inclusive, muitos métodos privados com lógicas possivelmente difíceis de usar e, consequentemente, de testar e dizer se estão corretas.

Repare também na quantidade de responsabilidades distintas da classe `Divida` : registrar pagamentos, aplicar descontos em pagamentos quando aplicáveis, filtrar os pagamentos já realizados, validar CNPJ, dentre outras. Quando uma classe tem muitas responsabilidades com pouca ou nenhuma relação entre si, dizemos que ela não é coesa.

Veja, por exemplo, que se quisermos utilizar somente a validação de CNPJ, precisaremos ainda assim criar uma instância de `Divida` para poder utilizar o código já existente. E se quisermos implementar a validação de CNPJ na classe `Pagamento` , também? Faz sentido criar uma instância de `Divida` só para fazer essa validação, nesse caso? Note como a baixa coesão da classe dificulta o reaproveitamento do código dentro dela.

No entanto, se olharmos com atenção, veremos que podemos agrupar os métodos de acordo com os atributos com os quais eles trabalham.

```
public class Divida {
  // atributos

  // métodos que trabalham com CNPJ
  public boolean cnpjValido() {...}
  private int primeiroDigitoCorretoParaCnpj() {...}
  private int primeiroDigitoVerificadorDoCnpj() {...}
  private int segundoDigitoCorretoParaCnpj() {...}
  private int segundoDigitoVerificadorDoCnpj() {...}
  public String getCnpjCredor() {...}
  public void setCnpjCredor(String cnpjCredor) {...}

  // métodos que trabalham com a lista de pagamentos
  public ArrayList<Pagamento> pagamentosAntesDe(Calendar data) {...}
  public ArrayList<Pagamento> pagamentosComValorMaiorQue(double valorMinimo) {...}
  public ArrayList<Pagamento> pagamentosDo(String cnpjPagador) {...}
  public void registra(Pagamento pagamento) {...}

  // outros métodos
}
```

Note como os grupos de métodos são independentes. Conseguimos separá-los rapidamente em classes diferentes. Podemos, por exemplo, criar uma classe `Cnpj` para lidar somente com o tratamento do CNPJ da nota fiscal:

```
public class Cnpj {
  private String cnpj;

  public boolean cnpjValido() {...}
  private int primeiroDigitoCorretoParaCnpj() {...}
  private int primeiroDigitoVerificadorDoCnpj() {...}
  private int segundoDigitoCorretoParaCnpj() {...}
  private int segundoDigitoVerificadorDoCnpj() {...}
  public String getCnpjCredor() {...}
  public void setCnpjCredor(String cnpjCredor) {...}
}
```

Os nomes do atributo e dos métodos ficaram um pouco estranhos, não? Podemos renomeá-los, inclusive refletindo que a nossa classe representa qualquer CNPJ agora, e não apenas o CNPJ de um credor:

```
public class Cnpj {
  private String valor;

  public boolean ehValido() {...}
  private int primeiroDigitoCorreto() {...}
  private int primeiroDigitoVerificador() {...}
  private int segundoDigitoCorreto() {...}
  private int segundoDigitoVerificador() {...}
  public String getValor() {...}
  public void setValor(String valor) {...}
}
```

E agora, na classe `Divida`, basta termos uma instância de `Cnpj` e gerar um getter para ela:

```
public class Divida {
  private double total;
  private double valorPago;
  private String credor;
  private Cnpj cnpjCredor = new Cnpj();
  private ArrayList<Pagamento> pagamentos = new ArrayList<Pagamento>();

  public Cnpj getCnpjCredor() {
    return this.cnpjCredor;
  }

  // outros métodos
}
```

Precisamos mudar o código que cria uma `Divida` na classe `BalancoEmpresa`:

```
public class BalancoEmpresa {
  public void registraDivida(String credor, String cnpjCredor, double valor) {
    Divida divida = new Divida();
    divida.setTotal(valor);
    divida.setCredor(credor);
    divida.getCnpjCredor().setValor(cnpjCredor); // agora usamos o getter e depois o setter
    dividas.put(cnpjCredor, divida);
  }
  public void pagaDivida(String cnpjCredor, double valor, String nomePagador, String cnpjPa
}
```

Fazendo isso, tiramos a responsabilidade de representar o CNPJ da classe `Divida`, deixando-a mais fácil de manter. Se precisarmos modificar algo associado ao CNPJ não precisamos procurar o código em várias classes, somente na classe `Cnpj`. Além disso, essa classe possui uma responsabilidade bem específica. Note como ela é pequena e reutilizável.

Utilizar tipos pequenos, como a classe `Cnpj`, também favorece a legibilidade do código (https://blog.caelum.com.br/pequenos-objetos-imutaveis-e-tiny-types/), principalmente para quem vai usar nosso código posteriormente.

Podemos fazer o mesmo com os pagamentos da dívida. Criamos uma nova classe que represente uma lista de pagamentos, especificamente. Mas, melhor ainda, podemos aproveitar código já existente no Java para deixar nosso

código mais expressivo: fazemos nossa nova classe estender `ArrayList`.

```
public class Pagamentos extends ArrayList<Pagamento> {
  public ArrayList<Pagamento> pagamentosAntesDe(Calendar data) {
    ArrayList<Pagamento> pagamentosFiltrados = new ArrayList<Pagamento>();
    for (Pagamento pagamento : this) {
      if (pagamento.getData().before(data)) {
        pagamentosFiltrados.add(pagamento);
      }
    }
    return pagamentosFiltrados;
  }
  public ArrayList<Pagamento> pagamentosDo(String cnpjPagador) {
    ArrayList<Pagamento> pagamentosFiltrados = new ArrayList<Pagamento>();
    for (Pagamento pagamento : this) {
      if (pagamento.getCnpjPagador().equals(cnpjPagador)) {
        pagamentosFiltrados.add(pagamento);
      }
    }
    return pagamentosFiltrados;
  }
  public ArrayList<Pagamento> pagamentosComValorMaiorQue(double valorMinimo) {
    ArrayList<Pagamento> pagamentosFiltrados = new ArrayList<Pagamento>();
    for (Pagamento pagamento : this) {
      if (pagamento.getValor() > valorMinimo) {
        pagamentosFiltrados.add(pagamento);
      }
    }
    return pagamentosFiltrados;
  }
}
```

Repare que é necessário alterarmos os métodos para percorrer os elementos de `this`.

Estendendo de `ArrayList`, ganhamos métodos para adicionar, remover e acessar diretamente itens da nota fiscal. Além disso, conseguimos usar uma instância da nossa classe no foreach do Java:

```
Pagamentos pagamentos = new Pagamentos();
// adiciona alguns pagamentos na colecao
for (Pagamento pagamento : pagamento) {
  System.out.println(pagamento.getValor());
}
```

Note que, na classe `Divida`, o método `registra` mexe com o atributo `valorPago` além da lista de itens:

```
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.pagamentos.add(pagamento);
  }
```

Mas veja também que esse atributo está diretamente relacionado com os pagamentos da dívida. O valor pago da dívida está diretamente relacionado com os pagamentos já realizados. Faz sentido, então, passarmos esse atributo para a nossa nova classe. Colocamos também um getter para podermos utilizá-lo no resto do sistema.

```
public class Pagamentos extends ArrayList<Pagamento> {
  private double valorPago;

  // outros métodos

  public double getValorPago() {
    return this.valorPago;
  }
}
```

Fazendo isso, podemos passar agora o método `registra` para a classe `Pagamentos`, deixando nela tudo o que é relacionado a uma lista de pagamentos:

```
public class Pagamentos extends ArrayList<Pagamento> {
  private double valorPago;

  // outros métodos

  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);
  }
}
```

Por fim, adaptamos a classe `Divida` para utilizar a nossa nova classe. Apagamos a lista e o atributo `valorPago` que passamos para a classe `Pagamentos` e, no lugar, colocamos um atributo do tipo dessa classe. Apagamos, também, os métodos copiados e, no lugar, colocamos um getter para o novo atributo.

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

  public Pagamentos getPagamentos() {...}
  public Cnpj getCnpj() {...}
```

```
    // outros métodos
  }
```

Além disso, precisamos mudar o método `pagaDivida` na classe `BalancoEmpresa` para usar nossa nova classe:

```
public class BalancoEmpresa {
  public void registraDivida(String credor, String cnpjCredor, double valor) {...}
  public void pagaDivida(String cnpjCredor, double valor, String nomePagador, String cnpjPa
    Divida divida = dividas.get(cnpjCredor);
    if (divida != null) {
      Pagamento pagamento = new Pagamento();
      pagamento.setCnpjPagador(cnpjPagador);
      pagamento.setPagador(nomePagador);
      pagamento.setValor(valor);
      divida.getPagamentos().registra(pagamento);
    }
  }
}
```

Compare, agora, a classe `Divida` antes e depois das nossas refatorações. Veja como ela ficou mais simples. Repare também como as responsabilidades, que antes estavam todas na classe `Divida`, agora estão separadas em classes com papéis bem definidos, ou seja, mais coesas. Além disso, a classe `Divida` só precisa saber agora o que as classes `Cnpj` e `Pagamentos` fazem, não como fazem: ganhamos em encapsulamento!

A separação de responsabilidades em classes é um conceito tão importante em orientação a objetos que é considerado um princípio do paradigma, conhecido como Single Responsibility Principle. Esse princípio diz que uma classe deve ter somente uma responsabilidade e que uma responsabilidade deve estar encapsulada inteiramente em uma classe. Isso tende a deixar nosso código mais simples, pois, com as responsabilidades bem isoladas em classes distintas, fica mais fácil saber qual parte do código executa cada lógica e onde precisamos mexer para alterar um determinado comportamento.

Note como antes tínhamos uma classe sem responsabilidade nenhuma, anêmica. Depois passamos para o oposto: uma classe com muitas responsabilidades distintas. Por fim, chegamos numa distribuição de responsabilidades mais aceitável, com classes pequenas e coesas. É importante observar que o que é aceitável depende do domínio que estamos representando e do quanto queremos quebrar as responsabilidades. Sempre é possível deixar as responsabilidades mais distribuídas; cabe ao desenvolvedor decidir até onde ele quer dividí-las.