01

Revisitando a Orientação a Objetos

Transcrição

Quantas vezes nos deparamos no dia a dia com aquele pedaço de código, aquele arquivo gigante, difícil de entender e, portanto, quase impossível de modificar?

```
public class Cadastro {

  private String nome;
  private String cpf;
  private String cnpj;
  private boolean pessoaFisica;

  public boolean processa(String acao, String tipo, String nome,
                          String cpf, String cnpj) {
    if(cpf != null) {
      String erro = "";
      if (cpf.length() < 11)
        erro += "Sao necessarios 11 digitos para verificacao do CPF! \n\n";
      if(cpf.matches(".*\\D.*"))
        erro += "A verificacao de CPF suporta apenas numeros! \n\n";
      if(cpf == "00000000000" || cpf == "11111111111" ||
         cpf == "22222222222" || cpf == "33333333333" ||
         cpf == "44444444444" || cpf == "55555555555" ||
         cpf == "66666666666" || cpf == "77777777777" ||
         cpf == "88888888888" || cpf == "99999999999") {
        erro += "Numero de CPF invalido!";
      }
      int[] a = new int[11];
      int b = 0;
      int c = 11;
      for(int i = 0; i < 11; i++) {
        a[i] = cpf.charAt(i) - '0';
        if(i < 9) b += (a[i] * --c);
      }
      int x = b % 11;
      if(x < 2) {
        a[9] = 0;
      }
      else {
        a[9] = 11-x;
      }
      b = 0;
      c = 11;
      for(int y = 0; y < 10; y++) b += (a[y] * c--);
      x = b % 11;
      if(x < 2) {
        a[10] = 0;
      }
      else {
        a[10] = 11-x;
      }
```

```
      if((cpf.charAt(9) - '0' != a[9]) || (cpf.charAt(10) - '0' != a[10])) {
        erro +="Digito verificador com problema!";
      }
      if(erro.length() > 0) {
        return false;
      }
    }
    if("cadastra".equals(acao)) {
      if("pessoa_fisica".equals(tipo)) {
        if(cnpj != null) {
          return false;
        }
        else {
          this.nome = nome;
          this.cpf = cpf;
          this.pessoaFisica = true;
        }
      }
      else if ("pessoa_juridica".equals(tipo)) {
        if(cpf != null) {
          return false;
        }
        else {
          this.nome = nome;
          this.cnpj = cnpj;
          this.pessoaFisica = false;
        }
      }
    }
    else if("atualiza".equals(acao)) {
      if("pessoa_fisica".equals(tipo)) {
        if(cnpj != null) {
          return false;
        }
        else {
          this.nome = nome;
          this.cpf = cpf;
          this.pessoaFisica = true;
        }
      }
      else if ("pessoa_juridica".equals(tipo)) {
        if(cpf != null) {
          return false;
        }
        else {
          this.nome = nome;
          this.cnpj = cnpj;
          this.pessoaFisica = false;
        }
      }
    }

    return false;
  }

  public String getNome() {
    return nome;
  }
```

```
  public void setNome(String nome) {
    this.nome = nome;
  }

  public String getCpf() {
    return cpf;
  }

  public void setCpf(String cpf) {
    this.cpf = cpf;
  }

  public String getCnpj() {
    return cnpj;
  }

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

  public boolean isPessoaFisica() {
    return pessoaFisica;
  }

  public void setPessoaFisica(boolean pessoaFisica) {
    this.pessoaFisica = pessoaFisica;
  }
}
```

Não precisamos ir muito longe para encontrarmos mais de um desses arquivos em um mesmo projeto. Ele é o motivo de piada nas nossas conversas durante os almoços e eventos, mas por que eles continuam aparecendo no nosso dia a dia?

Boas práticas de OO servem para diminuir os efeitos negativos de um código mal estruturado. Facilitam sua manutenção e adaptação à medida que precisamos adicionar funcionalidades ou alterar o comportamento do sistema.

```
Tomemos como exemplo um sistema de controle financeiro de uma empresa. Nesse sistema, teremos ur
```

Como todo tutorial de Orientação a Objetos, a prática comum é criar a classe com seus getters e setters:

```
public class Divida {
  private double total;
  private double valorPago;
  private String credor;
  private String cnpjCredor;

  public double getTotal() {
    return this.total;
  }
  public void setTotal(double total) {
    this.total = total;
  }
```

```
  public double getValorPago() {
    return this.valorPago;
  }
  public void setValorPago(double valorPago) {
    this.valorPago = valorPago;
  }
  public String getCredor() {
    return this.credor;
  }
  public void setCredor(String credor) {
    this.credor = credor;
  }
  public String getCnpjCredor() {
    return this.cnpjCredor;
  }
  public void setCnpjCredor(String cnpjCredor) {
    this.cnpjCredor = cnpjCredor;
  }
}
```

Essa classe é utilizada pelo `BalancoEmpresa` para registrar e atualizar dívidas. O método que registra dívidas cria uma instância de `Divida`, preenche o valor, os dados do credor e guarda essa dívida num mapa em que a chave é o CNPJ do credor.

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

  public void registraDivida(String credor, String cnpjCredor, double valor) {
    Divida divida = new Divida();
    divida.setTotal(valor);
    divida.setCredor(credor);
    divida.setCnpjCredor(cnpjCredor);
    dividas.put(cnpjCredor, divida);
  }
}
```

Temos também um método que paga parte de uma dívida. Ele recebe o CNPJ de um credor e o valor que foi pago, busca a `Divida` correspondente no banco usando o CNPJ do credor e a atualiza.

```
public void pagaDivida(String cnpjCredor, double valor) {
  Divida divida = dividas.get(cnpjCredor);
  if (divida != null) {
    divida.setValorPago(divida.getValorPago() + valor);
  }
}
```

Imagine que a empresa tem um outro sistema só para gerenciar as dívidas. Ele possui a classe `GerenciadorDeDividas`, que tem um método que também permite pagar uma dívida:

```
public class GerenciadorDeDividas {
  public void efetuaPagamento(Divida divida, double valor) {
    divida.setValorPago(divida.getValorPago() + valor);
```

```
  }

  // outros métodos
}
```

Agora alteramos o valor pago de uma dívida em dois lugares. Será que isso é ruim? Suponha que precisamos descontar uma taxa de R$ 8 dos pagamentos de dívida maiores que R$ 100. Ou seja, se o pagamento é de R$ 200 o pagamento real é de R$ 192. Se o pagamento é de R$ 40 o pagamento real também é de R$ 40. Em quantos lugares precisaremos mexer? Dois:

```
public class BalancoEmpresa {
  public void pagaDivida(String cnpjCredor, double valor) {
    Divida divida = dividas.get(cnpjCredor);
    if (divida != null) {
      if (valor > 100) {
        valor = valor - 8;
      }
      divida.setValorPago(divida.getValorPago() + valor);
    }
  }
}
```

```
public class GerenciadorDeDividas {
  public void efetuaPagamento(Divida divida, double valor) {
    if (valor > 100) {
      valor = valor - 8;
    }
    divida.setValorPago(divida.getValorPago() + valor);
  }
}
```

No nosso caso já são dois pontos de utilização, e poderiam ser muito mais! Como ter certeza que não há outros pontos do código que também usavam a variável diretamente? Ou em projetos de terceiros que usam o nosso como biblioteca?

O problema é que temos dados (valor pago da dívida) separados de comportamento (pagamento da dívida). Vemos que a classe `Divida` é burra: ela não faz nada, somente armazena valores; ela é uma estrutura de dados.

Para juntarmos os dados e o comportamento, vamos criar o método `paga(double valor)` na classe `Divida`.

```
public class Divida {
  public void paga(double valor) {
    if (valor > 100) {
      valor = valor - 8;
    }
    this.valorPago += valor;
  }
  // getters e setters
}
```

Podemos agora simplesmente chamar o novo método quando quisermos pagar uma parte de uma dívida. O método `pagaDivida` da nossa classe `BalancoEmpresa` fica assim:

```
  public void pagaDivida(String cnpjCredor, double valor) {
    Divida divida = dividas.get(cnpjCredor);
    if (divida != null) {
      divida.paga(valor);
    }
  }
```

Enquanto isso, na classe `GerenciadorDeDividas`, mudamos o código do método `efetuaPagamento` para:

```
  public void efetuaPagamento(Divida divida, double valor) {
    divida.paga(valor);
  }
```

Agora, se precisarmos acrescentar alguma verificação no valor a ser pago, só precisamos alterar o método `paga` da classe `Divida`. Por exemplo, podemos querer verificar que o valor a ser pago é positivo. Afinal, não faz sentido pagar um valor negativo!

```
public class Divida {
  public void paga(double valor) {
    if (valor < 0) {
      throw new IllegalArgumentException("Valor invalido para pagamento");
    }
    if (valor > 100) {
      valor = valor - 8;
    }
    this.valorPago += valor;
  }
  // getters e setters
}
```

Nosso problema foi resolvido, mas ainda é possível adicionar um valor da maneira antiga (usando o getter e setter). Nada impede que alguém utilize esse código dessa forma. Pior ainda, uma vez que essas variáveis estão expostas, é mais fácil para um desenvolvedor quebrar o padrão de utilização do `paga` e utilizar o setter, cometendo um erro, uma vez que o getter e setter são nomes padronizados: sem pensar o desenvolvedor acaba introduzindo um novo bug no sistema.

Queremos forçar o programador a utilizar o método `paga` quando for pagar uma parte da dívida.

Consideremos então o método `setValorPago`. Ele permite que mudemos o valor já pago da dívida a qualquer momento. Não queremos isso!

Qual a única forma de mudar o valor pago de uma dívida? Acrescentando um pagamento. E só! O método `setValorPago` não faz sentido nesse caso! Podemos então removê-lo.

```
public class Divida {
  private double total;
  private double valorPago;
  private String credor;
  private String cnpjCredor;
```

```
  public double getTotal() {
    return this.total;
  }
  public void setTotal(double total) {
    this.total = total;
  }
  public double getValorPago() {
    return this.valorPago;
  }
  public String getCredor() {
    return this.credor;
  }
  public void setCredor(String credor) {
    this.credor = credor;
  }
  public String getCnpjCredor() {
    return this.cnpjCredor;
  }
  public void setCnpjCredor(String cnpjCredor) {
    this.cnpjCredor = cnpjCredor;
  }
  public void paga(double valor) {
    if (valor < 0) {
      throw new IllegalArgumentException("Valor invalido para pagamento");
    }
    if (valor > 100) {
      valor = valor - 8;
    }
    this.valorPago += valor;
  }
}
```

Quando estamos desenvolvendo um sistema orientado a objetos é comum criarmos classes para isolar os dados em conjuntos que façam mais sentido estarem juntos, o que em muitos casos é uma boa ideia.

Um problema clássico acontece quando a lógica associada a esses dados continua espalhada pelo código, quebrando o controle que aquele objeto tinha sobre seus dados. A manutenção era difícil, custosa e frágil, uma vez que diversos pontos tinham que ser alterados a cada nova modificação. Cuidando com atenção de seus membros, o controle não é perdido e a mudança é em um ponto único.

Observe que, no exemplo utilizado, o simples fato de unir dados e comportamento facilitou a adição de novas funcionalidades e facilitou a mudança das funcionalidades existentes.

No nosso exemplo, o `BalancoEmpresa` sabe o que a classe `Divida` faz, ou seja, controla seu próprio valor pago. Mas a classe `BalancoEmpresa` não sabe como a `Divida` faz esse controle. Ou seja, o modo como a classe `Divida` altera seu próprio valor pago está escondido das outras classes, está encapsulado. É graças a esse encapsulamento que conseguimos alterar esse comportamento facilmente.

Em um bom design orientado a objetos, dado uma mudança, você sabe onde alterar, já que esses pontos são razoavelmente explícitos. Encapsulamento é um dos caminhos.

Vamos incrementar um pouco nosso modelo. Agora, além de guardar o valor já pago, a classe `Divida` vai guardar informações de cada pagamento realizado. Para isso, vamos criar uma nova classe para representar um pagamento. Essa classe vai guardar o nome e o CNPJ de quem o fez, bem como o valor pago.

```
public class Pagamento {
  private String pagador;
  private String cnpjPagador;
  private double valor;

  public String getPagador() {
    return this.pagador;
  }
  public void setPagador(String pagador) {
    this.pagador = pagador;
  }
  public String getCnpjPagador() {
    return this.cnpjPagador;
  }
  public void setCnpjPagador(String cnpjPagador) {
    this.cnpjPagador = cnpjPagador;
  }
  public double getValor() {
    return this.valor;
  }
  public void setValor(double valor) {
    this.valor = valor;
  }
}
```

A `Divida` foi incrementada e agora guarda uma lista de itens do tipo `Pagamento` e, como esperado, criamos também seu getter e setter:

```
public class Divida {
  private ArrayList<Pagamento> pagamentos = new ArrayList<Pagamento>();
  // outros atributos

  public ArrayList<Pagamento> getPagamentos() {
    return this.pagamentos;
  }
  public void setPagamentos(ArrayList<Pagamento> pagamentos) {
    this.pagamentos = pagamentos;
  }

  // outros métodos
}
```

Mas será que queremos o setter? Assim como o valor pago não pode ser mudado para um número arbitrário, não queremos que qualquer um possa simplesmente trocar a nossa lista de pagamentos a qualquer momento. É possível, inclusive, fazer o seguinte uso indesejado:

```
Divida divida = new Divida();
divida.setPagamentos(null);
```

Estamos novamente abrindo a implementação da nossa classe. A `Divida` perde o controle desse atributo. Mais uma vez permitimos que outras classes alterem nossos atributos da forma que quiserem, quebramos o encapsulamento. Não queremos o método `setPagamentos` ! Podemos então removê-lo.

Vejamos como está a classe `BalancoEmpresa`.

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

  public void registraDivida(String credor, String cnpjCredor, double valor) {
    Divida divida = new Divida();
    divida.setTotal(valor);
    divida.setCredor(credor);
    divida.setCnpjCredor(cnpjCredor);
    dividas.put(cnpjCredor, divida);
  }

  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.paga(valor);
      divida.getPagamentos().add(pagamento);
    }
  }
}
```

Note que o método `pagaDivida` é responsável por adicionar um novo pagamento e atualizar o valor pago da dívida. Para manter a consistência, a classe `GerenciadorDeDividas` também deve adicionar o novo pagamento na dívida, além de continuar atualizando o valor pago.

O código é exatamente o mesmo da classe `BalancoEmpresa`, logo podemos resolver o problema facilmente copiando e colando o código de uma classe para a outra:

```
public class GerenciadorDeDividas {
  public void efetuaPagamento(Divida divida, String nomePagador, String cnpjPagador, double
    Pagamento pagamento = new Pagamento();
    pagamento.setCnpjPagador(cnpjPagador);
    pagamento.setPagador(nomePagador);
    pagamento.setValor(valor);
    divida.paga(valor);
    divida.getPagamentos().add(pagamento);
  }
}
```

O que aconteceria se precisássemos adicionar um pagamento na dívida em outro ponto no nosso código? Não seria improvável termos um código assim:

```
  public void adicionaPagamentoNaDivida(String pagador, String cnpj, double valor) {
    Pagamento pagamento = new Pagamento();
    pagamento.setCnpjPagador(cnpj);
    pagamento.setPagador(pagador);
    pagamento.setValor(valor);
```

```
    divida.getPagamentos().add(pagamento);
  }
```

Qual é o problema nesse código? O código em si está correto, mas esquecemos de adicionar o valor do pagamento na dívida (método `paga`).

A ação de registrar um novo pagamento em uma dívida consiste em inserir o pagamento na lista e adicionar o valor do pagamento no total pago. Note que essa ação está relacionada à dívida. Por que não criar um método `registra` na `Divida` que recebe um `Pagamento` e realiza as atividades necessárias?

```
public class Divida {
  private ArrayList<Pagamento> pagamentos = new ArrayList<Pagamento>();
  private double valorPago;
  // outros atributos

  public void paga(double valor) {
    if (valor > 100) {
      valor = valor - 8;
    }
    this.valorPago += valor;
  }
  public void registra(Pagamento pagamento) {
    this.pagamentos.add(pagamento);
    this.paga(pagamento.getValor());
  }
  // outros métodos
}
```

Agora, basta chamarmos esse novo método onde é necessário registrar um pagamento de uma dívida:

```
public class BalancoEmpresa {
  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.registra(pagamento);
    }
  }
}
```

Dessa forma, a responsabilidade de registrar um pagamento de uma dívida, que antes estava espalhada em várias classes, agora está centralizada em um único ponto. Se alguma regra relacionada a essa ação mudar, sabemos exatamente onde devemos mexer.

Note que não faz mais sentido deixarmos qualquer um chamar o método `paga` na classe `Divida`. Podemos deixá-lo privado.

As mudanças que fizemos vão de acordo com um princípio muito importante de orientação a objetos. Encapsulando o comportamento da classe `Divida`, paramos de pedir os seus dados para fazer o que queríamos e passamos a dizer a ela o que queríamos que fosse feito. O princípio, conhecido como Tell, Don't Ask, diz exatamente isso: quando você está interagindo com outro objeto (`BalancoEmpresa` registrando pagamentos na `Divida` por exemplo), você deve dizer o que quer que esse outro objeto faça (chamada do método `registra()`) e não perguntar sobre o seu estado para tomar decisões (implementação anterior que usava o `getPagamentos()`).

Veja como partimos de uma classe `Divida` que não fazia nada, apenas guardava valores. Ela era o que é conhecido como modelo anêmico: uma classe de modelo sem comportamento algum. Seguindo o princípio de unir comportamento e dados, escondemos os dados e a forma como interagimos com eles, encapsulamos esse comportamento. A classe `Divida` deixou de ser anêmica.