01

Reduzindo acoplamento com polimorfismo

Transcrição

Nosso código já está bastante desacoplado: uma classe não precisa saber como outra funciona, apenas o que ela faz. Mas será que é possível deixar nossas classes ainda mais independentes umas das outras? Veja que nossa classe `Pagamentos` usa especificamente uma instância de `ArrayList` para armazenar os pagamentos registrados:

```
public class Pagamentos {
  private ArrayList<Pagamento> pagamentos = new ArrayList<Pagamento>();
  private double valor;

  // métodos
}
```

Será que precisamos especificamente de uma `ArrayList`? Há outras classes na linguagem Java que nos permitem armazenar objetos e percorrê-los posteriormente. Uma delas, que já foi citada anteriormente, é a classe `HashSet`. Há algumas diferenças entre essas duas classes: o `HashSet`, por exemplo, não permite adicionar o mesmo objeto duas vezes. Mas, no nosso caso, poderíamos usar tanto um `ArrayList` como um `HashSet`. Então, vamos trocar o `ArrayList` da classe `Pagamentos` por um `HashSet`:

```
public class Pagamentos {
  private HashSet<Pagamento> pagamentos = new HashSet<Pagamento>();
  private double valor;

  // métodos
}
```

Repare que os lugares onde percorríamos os elementos da coleção não quebraram. As linhas abaixo continuam funcionando, mesmo com `this.pagamentos` sendo um `HashSet` em vez de um `ArrayList`:

```
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;
}
```

Como será que isso funciona? Como o Java consegue percorrer tipos diferentes de coleções? Será que temos, dentro da linguagem, um código como o apresentado a seguir?

```
public void percorreColecao(Object colecao) {
  if (tipo da colecao == "HashSet") {
    // percorre de um jeito
  } else if (tipo da colecao == "ArrayList") {
```

```
      // percorre de outro jeito
    } else if ...
    }
  }
```

O que aconteceria se eu criasse um novo tipo de coleção? Seria necessário acrescentar um `if` para a minha coleção nesse código. Leve isso para o seu código: toda vez que aparecer uma nova "entidade" no seu código, você precisa lembrar e procurar todos os lugares que fazem "ifs", e adicionar um a mais. A manutenção de algo desse tipo seria difícil, não?

Imagine se a implementação de `ArrayList` e a maneira de percorrê-lo mudassem: eles teriam que lembrar de mudar esse código também! Seria uma implementação muito acoplada e pouco coesa.

Agora, imagine se cada implementação de uma coleção tivesse um método que vai devolvendo um a um os elementos dela:

```
public class HashSet {
  public Object proximo() {...}
}

public class ArrayList {
  public Object proximo() {...}
}
```

Se todas as coleções tivessem esse método, a implementação do lado do Java ficaria bem mais fácil:

```
public void percorreColecao(Object colecaoQualquer) {
  Object elemento = colecaoQualquer.proximo();
  // faz alguma coisa
  elemento = colecaoQualquer.proximo();
  // faz alguma coisa
}
```

Mas o código que escrevemos acima não compila. O Java não tem como saber que ele pode chamar o método `proximo` na variável `colecao`, porque esse método não existe na classe `Object`. A variável `colecao` precisaria ser do tipo `HashSet` ou `ArrayList`. Mas o problema é que ela pode ser de qualquer um desses dois tipos.

A linguagem Java possui um mecanismo que permite especificar que uma ou mais classes possuem determinados métodos, que é a interface. Na interface, declaramos um conjunto de métodos mas não especificamos a implementação. Com a interface em mãos, indicamos que uma classe do nosso sistema implementa aquela interface.

Para o nosso caso das coleções, queremos dizer para o Java que tanto `ArrayList` como `HashSet` têm o método `proximo`. Então podemos criar a interface `Colecao`:

```
public interface Colecao {
  public Object proximo();
}
```

Em seguida, indicamos que nossas classes implementam essa interface, ou seja, fornecem uma implementação para os métodos que só declaramos na interface:

```
public class HashSet implements Colecao {
  public Object proximo() {...}
}

public class ArrayList implements Colecao {
  public Object proximo() {...}
}
```

Com isso, o seguinte código compila e funciona como esperado:

```
public void percorreColecao(Colecao colecao) {
  Object elemento = colecao.proximo();
  // faz alguma coisa
  elemento = colecao.proximo();
  // faz alguma coisa
}
```

Note que, agora, não importa o verdadeiro tipo da variável `colecao`. Só importa que ela tenha os métodos da interface `Colecao`. O código do método `percorreColecao` não está mais acoplado a uma classe específica: ele funciona para qualquer classe que siga a interface `Colecao`.

Podemos desacoplar ainda mais o modo de percorrer uma coleção de sua implementação. Imagine se tivéssemos um objeto cuja única função fosse nos fornecer uma sequência de elementos. Não importaria onde ou como esses elementos estivessem armazenados: esse objeto esconderia de nós essa informação. Ele poderia ser construído a partir de uma coleção e ir avançando nela conforme fôssemos pedindo elementos:

```
public class SequenciaDeElementos {
  private Colecao colecao;

  public SequenciaDeElementos(Colecao colecao) {
    this.colecao = colecao;
  }

  public Object proximo() {
    return this.colecao.proximo();
  }
}
```

Veja que, dessa forma, podemos passar apenas um objeto `SequenciaDeElementos` em vez de uma coleção inteira para alguém que queira percorrer seus elementos. Impedimos que esse alguém adicione ou altere algum elemento na nossa coleção.

Agora, será que a classe que armazena elementos também deve ser responsável por saber percorrê-los? Veja que são duas responsabilidades distintas. Podemos trazer essa responsabilidade para a nossa nova classe. Mas como implementaremos o método `proximo()`? Cada tipo de coleção guarda seus elementos de um jeito.

```
public class SequenciaDeElementos {
  private Colecao colecao;

  public SequenciaDeElementos(Colecao colecao) {
    this.colecao = colecao;
```

```
  }

  public Object proximo() {
    // ???
  }
}
```

Precisaremos, então, criar uma classe como essa para cada tipo de coleção que tivermos:

```
public class SequenciaDeElementosDeArrayList {
  private ArrayList lista;

  public SequenciaDeElementosDeArrayList(ArrayList lista) {
    this.lista = lista;
  }

  public Object proximo() {
    // aqui implementamos o algoritmo para percorrer uma ArrayList
  }
}

public class SequenciaDeElementosDeHashSet {
  private HashSet hashSet;

  public SequenciaDeElementosDeHashSet(HashSet hashSet) {
    this.hashSet = hashSet;
  }

  public Object proximo() {
    // aqui implementamos o algoritmo para percorrer um HashSet
  }
}
```

Mas assim caímos no mesmo problema de antes: como percorremos qualquer coleção de modo genérico, se agora temos uma classe para cada implementação de coleção? Novamente, podemos recorrer a interfaces para isso. Veja que só precisamos garantir que essas classes tenham o método `Object proximo()`. Vamos criar uma interface para representar isso:

```
public interface SequenciaDeElementos {
  public Object proximo();
}
```

Assim, podemos fazer as implementações específicas de sequências de elementos implementarem a interface `SequenciaDeElementos` e, novamente, conseguimos usar o mesmo código para percorrer qualquer tipo de coleção, não importando qual seja ela.

```
public void processaElementos(SequenciaDeElementos elementos) {
  Object elemento1 = elementos.proximo();
  Object elemento2 = elementos.proximo();
  // e assim por diante
}
```

Mas veja que não conseguimos criar um objeto para percorrer uma coleção de um modo genérico. Precisaríamos sempre descobrir qual o tipo da coleção para criar uma `SequenciaDeElementos`! Por isso, precisamos que uma coleção qualquer nos dê uma `SequenciaDeElementos` já pronta. Podemos exigir isso na interface `Colecao`:

```
public interface Colecao {
  public SequenciaDeElementos percorrerColecao();
}
```

Assim, cada implementação de `Colecao` fica responsável por criar o objeto que sabe percorrê-la:

```
public class ArrayList implements Colecao {
  public SequenciaDeElementos percorrerColecao() {
    return new SequenciaDeElementosDeArrayList(this);
  }
  // outros métodos
}

public class HashSet implements Colecao {
  public SequenciaDeElementos percorrerColecao() {
    return new SequenciaDeElementosDeHashSet(this);
  }
  // outros métodos
}
```

O exemplo acima ilustra a ideia que as implementações de coleções da linguagem Java realmente seguem. No Java, temos a interface `Iterator`, que corresponde à nossa interface `SequenciaDeElementos`:

```
public inteface Iterator {
  public boolean hasNext();
  public Object next();
  public void remove(); // remove o elemento recem-recuperado
}
```

Temos, também, a interface `Collection`, que toda coleção de elementos implementa. Assim como nossa interface `Colecao`, ela tem um método que devolve um objeto que sabe percorrê-la:

```
public interface Collection {
  public Iterator iterator(); // deve saber criar um objeto que sabe percorrê-la

  boolean add(Object o);
  boolean contains(Object o);
  // e vários outros métodos
}
```

Mas os criadores da linguagem decidiram abstrair um pouco mais. Uma coleção de elementos pode ser percorrida, mas podemos ter também uma classe que não armazena elementos, mas permite percorrer uma sequência deles; uma classe que gere números em sequência, por exemplo, não é uma coleção, mas pode ser percorrida:

```
GeradorDeNumeros gerador = new GeradorDeNumeros();
Iterator iterador = gerador.iterator();
```

```
iterador.next(); // 1
iterador.next(); // 2
iterador.next(); // 3
iterador.next(); // 4
```

Assim, a linguagem Java possui uma interface que representa esse conceito de um objeto que pode ser percorrido: a interface `Iterable`. Essa interface especifica apenas o método `iterator`, que deve devolver um elemento que saiba percorrer o objeto que a implementa.

```
public interface Iterable {
  public Iterator iterator();
}
```

Uma coleção, em particular, é um objeto que pode ser percorrido. Então, na verdade, a interface `Collection` estende a interface `Iterable`: toda `Collection` é, antes de tudo, um `Iterable`:

```
public interface Collection extends Iterable {
  // já somos obrigados a implementar o método iterator()
  // não precisamos declará-lo de novo aqui

  boolean add(Object o);
  boolean contains(Object o);
  // e vários outros métodos
}
```

A linguagem Java, então, implementa um mecanismo que permite percorrer de um modo padrão qualquer conjunto de elementos. Basta que a classe que representa esse conjunto de elementos implemente a interface `Iterable`. Note o quão poderoso é isso: o modo de percorrer uma coleção é totalmente desacoplado do modo de armazenar os elementos dessa coleção. Qualquer classe que implementar essa interface pode ser percorrida da mesma forma:

```
Iterable<Pagamento> colecao = new ArrayList<Pagamento>(); // ou HashSet<Pagamento>
for (Pagamento pagamento : colecao) {
  System.out.println(pagamento.getValor());
}
```

Repare que usamos um `ArrayList` no exemplo. Mas será que isso é possível?

```
Iterable<Pagamento> colecao = new Pagamentos();
for (Pagamento pagamento : colecao) {
  System.out.println(pagamento.getValor());
}
```

Sim, é possível! Basta fazer nossa classe `Pagamentos` implementar a interface `Iterable`. Mas como implementar o método `iterator` dessa interface?

```
public class Pagamentos implements Iterable<Pagamento> {
  private ArrayList<Pagamento> pagamentos = new ArrayList<Pagamento>();
  private double valorPago;
```

```
  @Override
  public Iterator<Pagamento> iterator() {
    // como implementar esse método?
  }
}
```

Não queremos iterator nos elementos armazenados da variável `pagamentos` ? Essa variável é do tipo `ArrayList` , que implementa `Iterable` . Então ela também tem o método `iterator` , que devolve justamente o que precisamos: um objeto para percorrer os pagamentos armazenados na nossa classe. Assim, nossa implementação do método `iterator` deve só chamar o `iterator` da nossa `ArrayList` :

```
public class Pagamentos implements Iterable<Pagamento> {
  private ArrayList<Pagamento> pagamentos = new ArrayList<Pagamento>();
  private double valorPago;

  @Override
  public Iterator<Pagamento> iterator() {
    return this.pagamentos.iterator();
  }

  // outros métodos
}
```

Pronto! Fazendo isso, conseguimos fazer com que o Java saiba percorrer nossa coleção. Esse código funciona, agora:

```
Pagamentos colecao = new Pagamentos();
for (Pagamento pagamento : colecao) {
  System.out.println(pagamento.getValor());
}
```

Veja só o poder de usar interfaces e do polimorfismo: o Java foi criado muito antes de nossa classe existir. Mesmo assim, conseguimos fazer nossa classe integrar perfeitamente com a linguagem. Isso porque o Java não acoplou a implementação da iteração em coleções a uma classe específica, mas sim a uma interface, o que é um acoplamento muito menor!

Será que podemos aplicar essa ideia na hora de desenvolver nosso próprio código? Vamos a um exemplo: até agora, apenas empresas podiam realizar pagamentos e cobrar dívidas, já que estamos usando o CNPJ para identificar o pagador e o credor. Mas seria melhor se uma pessoa física também pudesse ter e cobrar dívidas. Só que, para tanto, precisamos armazenar um CPF. Vamos seguir a mesma ideia do CNPJ e criar uma classe para representar um CPF, lembrando de implementar validação, o `equals()` e o `hashCode()` , já que vamos usá-la para agrupar dívidas num `HashMap` também.

```
public class Cpf {
  private String valor;

  public Cpf(String valor) {
    this.valor = valor;
  }

  public String getValor() {
    return this.valor;
```

```
  }

  public boolean equals(Object o) {
    if (!(o instanceof Cpf)) {
      return false;
    }
    Cpf outro = (Cpf) o;
    return this.valor.equals(outro.valor);
  }

  public int hashCode() {
    return this.valor.hashCode();
  }

  public boolean ehValido() {
    return primeiroDigitoVerificadoEstaCorreto()
            && segundoDigitoVerificadorEstaCorreto();
  }

  private boolean primeiroDigitoVerificadoEstaCorreto() {
    // Calcula o primeiro digito verificador do CPF se
    // ele estiver correto e compara com o valor preenchido
  }

  private boolean segundoDigitoVerificadorEstaCorreto() {
    // Calcula o segundo digito verificador do CPF se
    // ele estiver correto e compara com o valor preenchido
  }
}
```

Agora, na classe `Divida`, precisamos colocar um CPF, também:

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

  public Cpf getCpfCredor() {
    return this.cpfCredor;
  }
  public void setCpfCredor(Cpf cpfCredor) {
    this.cpfCredor = cpfCredor;
  }

  // outros métodos
}
```

Mas veja que, do jeito que está, podemos ter tanto um CPF como um CNPJ para a dívida. Precisamos adaptar o código da classe `BalancoEmpresa` para funcionar com o CPF também.

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

```
  private HashMap<Cpf, Divida> dividasPF = new HashMap<Cpf, Divida>();

  public void registraDivida(Divida divida) {
    if (divida.getCnpjCredor() != null) {
      dividasPJ.put(divida.getCnpjCredor(), divida);
    } else {
      dividasPF.put(divida.getCpfCredor(), divida);
    }
  }

  public void pagaDivida(Cnpj cnpjCredor, Pagamento pagamento) {
    Divida divida = dividasPJ.get(cnpjCredor);
    if (divida != null) {
      divida.registra(pagamento);
    }
  }

  public void pagaDivida(Cpf cpfCredor, Pagamento pagamento) {
    Divida divida = dividasPF.get(cpfCredor);
    if (divida != null) {
      divida.registra(pagamento);
    }
  }
}
```

O código não ficou muito legal. Duplicamos o método `pagaDivida`, criamos dois mapas e colocamos um `if` no método `registraDivida`. Imagine se tivéssemos mais tipos de documento, quantos `if`s não teríamos!

Mas CPF e CNPJ têm o mesmo papel, que é servir de identificação para um credor ou um pagador. Podemos criar uma interface para representar esse papel. Assim, nossas duas classes que têm esse papel podem implementá-la para que possamos tratá-las da mesma forma, de acordo com esse papel.

Vamos criar a interface `Documento` e fazer as classes `Cpf` e `Cnpj`. Um documento precisa ser válido, então podemos colocar o método `ehValido` nessa interface. Também precisa ter um número, então podemos adicionar o método `getValor`. Perceba que, intencionalmente, já demos o mesmo nome para o método de validação e para o método que devolve o número do documento nas duas classes.

```
public interface Documento {
  public boolean ehValido();
  public String getValor();
}

public class Cpf implements Documento {
  // o código continua igual
}

public class Cnpj implements Documento {
  // o código continua igual
}
```

Com isso, podemos mudar a classe `Divida`. Tiramos os atributos `cpfCredor` e `cnpjCredor` e, no lugar, colocamos um atributo `documentoCredor`. Esse atributo vai poder ser tanto um CPF como um CNPJ como qualquer outra classe que implemente essa interface.

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

  public Documento getDocumentoCredor() {
    return this.documentoCredor;
  }
  public void setDocumentoCredor(Documento documentoCredor) {
    this.documentoCredor = documentoCredor;
  }
  // outros métodos
}
```

Perceba que antes era possível existir uma `Divida` com um CPF e um CNPJ de um credor, o que não faz sentido. Agora, tendo somente um `Documento` isso não é mais possível: podemos ter somente um CPF ou um CNPJ ou qualquer outro tipo de documento que implemente a nossa interface!

Podemos mudar a classe `BalancoEmpresa` , também. Conseguimos remover toda a duplicação que inserimos anteriormente:

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

  public void registraDivida(Divida divida) {
    if (divida.getDocumentoCredor() != null) {
      dividas.put(divida.getDocumentoCredor(), divida);
    }
  }

  public void pagaDivida(Documento documentoCredor, Pagamento pagamento) {
    Divida divida = dividas.get(documentoCredor);
    if (divida != null) {
      divida.registra(pagamento);
    }
  }
}
```

Note que conseguimos, inclusive, validar o documento do credor antes de registrar a dívida, não importando qual documento seja:

```
public void registraDivida(Divida divida) {
  if (divida.getDocumentoCredor() != null && divida.getDocumentoCredor().ehValido()) {
    dividas.put(divida.getDocumentoCredor(), divida);
  }
}
```

Veja que, agora, se quisermos criar outro tipo de documento, tudo o que precisamos fazer é criar uma classe que implementa a interface `Documento` ; nada mais!

```
public class CNH implements Documento {
  public boolean ehValido() {
    // valida o número da carta de habilitação
  }
  public String getValor() {
    // devolve o número da CNH
  }
}
```

Nossa classe `Divida`, agora, não depende mais de uma classe específica para representar o documento do credor: diminuímos ainda mais o acoplamento! A classe `BalancoEmpresa` também ficou mais desacoplada: ela não precisa saber qual tipo de documento ela está validando.

Atacamos um outro problema ainda: a quantidade de `if`s no código, também conhecida como complexidade ciclomática. Basicamente, quanto mais `if`s no código, mais valores diferentes possíveis para nossas variáveis. Fica mais difícil entender o que acontece num determinado trecho de código e, portanto, é mais difícil mantê-lo. Graças ao polimorfismo, não precisamos mais decidir qual trecho de código precisamos executar para validar um documento: deixamos a tarefa para o Java.

Com o polimorfismo, deixamos nosso código mais desacoplado e, portanto, mais flexível e fácil de manter.