01

Aumentando a flexibilidade com injeção de dependências

Transcrição

Atualmente, nossa classe `BalancoEmpresa` usa um mapa para armazenar as dívidas de acordo com o credor:

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

  public void registraDivida(Divida divida) {
    dividas.put(divida.getDocumentCredor(), divida);
  }

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

Imagine que queremos armazenar essas dívidas num banco de dados, agora. Então, precisamos escrever todo o código para conectar num banco de dados e gravar os dados da dívida nele, bem como o código para recuperar dados dele. Seguindo a ideia de divisão de responsabilidades, criamos uma classe para isolar essa responsabilidade. Por simplicidade, vamos apenas simular o funcionamento do banco de dados com um mapa:

```
public class BancoDeDados {
  private Map<Documento, Divida> dividasNoBanco = new HashMap<Documento, Divida>();

  public BancoDeDados(String endereco, String usuario, String senha) {
    System.out.println("Conectado!");
  }

  public void salva(Divida divida) {
    dividasNoBanco.put(divida.getDocumentoCredor(), divida);
  }

  public Divida carrega(Documento documentoCredor) {
    return dividasNoBanco.get(documentoCredor);
  }

  public void desconecta() {
    System.out.println("Desconectado!");
  }
}
```

Agora, na nossa classe `BalancoEmpresa`, usamos nossa nova classe para armazenar as dívidas num banco de dados:

```
public class BalancoEmpresa {
  private BancoDeDados bd = new BancoDeDados("servidor", "usuario", "senha");
```

```
  public void registraDivida(Divida divida) {
    bd.salva(divida);
  }

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

Mas será que é uma boa nossa classe `BalancoEmpresa` criar esse `BancoDeDados` ? Note que precisamos passar algumas configurações para criá-lo. Lembre, também, que essa criação, que envolve conectar com o banco, pode ser demorada e, no momento, é feita todas as vezes que criamos uma instância de `BalancoEmpresa` . Além disso, sempre conectamos, mas nunca desconectamos. Quando conseguimos saber que nossa instância não vai ser mais usada?

Note, ainda, que nossa classe depende de `BancoDeDados` , e vai atrás de um a cada vez que criamos uma nova instância. E se, em vez de nossa classe ir atrás das dependências dela, ela pudesse "pedir" essas dependências? Assim, poderíamos deixar a lógica de criação de um `BancoDeDados` num lugar mais apropriado, como no método `main` da nossa aplicação. Assim, conectamos só uma vez no banco de dados e, sempre que precisarmos de uma instância de `BalancoEmpresa` , passamos a nossa instância de `BancoDeDados` para ela.

Mas como podemos fazer para nossa classe `BalancoEmpresa` "pedir" uma dependência? Note que uma instância da classe `BancoDeDados` só pode ser criada se passarmos os argumentos necessários no construtor, ou seja, não conseguimos criá-la sem passar esses argumentos. Podemos aproveitar essa ideia para impedir que a classe `BalancoEmpresa` seja instanciada sem um `BancoDeDados` . Pedimos uma instância no construtor e guardamos essa instância para usá-la nos métodos:

```
public class BalancoEmpresa {
  private BancoDeDados bd;

  public BalancoEmpresa(BancoDeDados banco) {
    this.bd = banco;
  }

  public void registraDivida(Divida divida) {
    bd.salva(divida);
  }

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

Assim, na classe principal da nossa aplicação, basta criar uma vez um `BancoDeDados` e passá-lo para cada instância que criarmos da classe `BalancoEmpresa` :

```
public class MinhaAplicacao {
  public static void main(String[] args) {
    BancoDeDados bd = new BancoDeDados("servidor", "usuario", "senha");
    BalancoEmpresa balanco = new BalancoEmpresa(bd);
    registraDividas(balanco);
    realizaPagamentos(balanco);
    bd.desconecta();
  }
  private static void registraDividas(BalancoEmpresa balanco) {
    Divida d1 = new Divida();
    Divida d2 = new Divida();
    // preenche dados das dividas
    balanco.registraDivida(d1);
    balanco.registraDivida(d2);
  }
  private static void realizaPagamentos(BalancoEmpresa balanco) {
    Pagamento p1 = new Pagamento();
    Pagamento p2 = new Pagamento();
    Cnpj credor = new Cnpj("00.000.000/0001-01");
    // preenche dados dos pagamentos
    balanco.pagaDivida(credor, p1);
    balanco.pagaDivida(credor, p2);
  }
}
```

Note como tiramos a responsabilidade da classe `BalancoEmpresa` de criar uma dependência dela. Em vez disso, injetamos essa dependência pelo construtor. A construção da classe `BancoDeDados` pode ficar muito mais complicada que isso, mas as classes que dependem dela não são mais afetadas. Desacoplamos ainda mais nosso código!

A refatoração que acabamos de fazer no nosso código aplica um conceito conhecido como injeção de dependências. Basicamente, a ideia é que sua classe deixe explícito quais objetos ela precisa para funcionar. Quem for instanciar essa classe, precisa fornecer essas dependências. Assim, deixamos nossas classes mais coesas e desacoplamos um pouco mais uma classe de suas dependências.

Podemos aplicar uma ideia que vimos agora há pouco para deixar nosso código ainda mais desacoplado. Repare que, até agora, pedimos uma instância de uma classe específica no construtor: a classe `BancoDeDados` . Acoplamos nossa classe `BalancoEmpresa` a uma classe específica para guardar seus dados. E se quiséssemos trocar a forma como eles são armazenados? Num teste automatizado, por exemplo, podemos querer guardá-los num arquivo temporário em vez de um banco de dados real.

Para desacoplar nossa classe de uma implementação específica de armazenamento de dados, podemos criar uma interface chamada, por exemplo, `ArmazenadorDeDividas` . E quais métodos ela deve ter? Será que todas as formas de armazenar dados precisam desconectar? Provavelmente não, mas gravar e carregar dívidas, sim. Nossa interface deve ficar com essa cara, então:

```
public interface ArmazenadorDeDividas {
  public void salva(Divida divida);
  public Divida carrega(Documento documentoCredor);
}
```

Não precisamos alterar nada na classe `BancoDeDados` para que ela implemente nossa nova interface. Só precisamos dizer ao Java que ela implementa essa interface:

```
public class BancoDeDados implements ArmazenadorDeDividas {
  // código não muda
}
```

Agora, nossa classe `BalancoEmpresa` não precisa mais depender de um `BancoDeDados` especificamente, mas qualquer `ArmazenadorDeDividas`.

```
public class BalancoEmpresa {
  private ArmazenadorDeDividas dividas;

  public BalancoEmpresa(ArmazenadorDeDividas dividas) {
    this.dividas = dividas;
  }
  // o resto do código continua igual
}
```

Veja que, agora, se quisermos usar um arquivo para armazenar as dívidas na classe `BalancoEmpresa`, basta criar uma classe que implementa a nossa interface `ArmazenadorDeDividas`:

```
public class ArquivoDeDividas implements ArmazenadorDeDividas {
  private File arquivo;

  public ArquivoDeDividas(String nomeDoArquivo) {
    this.arquivo = new File(nomeDoArquivo);
  }

  @Override
  public Divida carrega(Documento documentoCredor) {
    // procura a divida no arquivo e a devolve
  }

  @Override
  public void salva(Divida divida) {
    // grava no arquivo
  }
}
```

E passá-la para a classe `BalancoEmpresa`:

```
public class MinhaAplicacao {
  public static void main(String[] args) {
    //ArmazenadorDeDividas bd = new BancoDeDados("servidor", "usuario", "senha");
    ArmazenadorDeDividas arquivo = new ArquivoDeDividas("dividas.txt");
    BalancoEmpresa balanco = new BalancoEmpresa(arquivo);
    registraDividas(balanco);
    realizaPagamentos(balanco);
    //bd.desconecta();
  }

  private static void registraDividas(BalancoEmpresa balanco) {
    Divida d1 = new Divida();
    Divida d2 = new Divida();
```

```
    // preenche dados das dividas
    balanco.registraDivida(d1);
    balanco.registraDivida(d2);
  }

  private static void realizaPagamentos(BalancoEmpresa balanco) {
    Pagamento p1 = new Pagamento();
    Pagamento p2 = new Pagamento();
    Cnpj credor = new Cnpj("00.000.000/0001-01");
    // preenche dados dos pagamentos
    balanco.pagaDivida(credor, p1);
    balanco.pagaDivida(credor, p2);
  }
}
```

Veja como nosso código ficou mais fácil de manter com o desacoplamento: só precisamos mudar nosso código na classe `MinhaAplicacao` para mudar a forma como as dívidas são armazenadas. A mudança não afeta em nada a classe `BalancoEmpresa`.

Usar interfaces nos traz polimorfismo: conseguimos trocar a implementação de uma determinada lógica sem precisar mexer no resto do código. E injeção de dependências nos permite trocar a implementação de um determinado componente do sistema de forma mais simples, sem precisar mexer em todos os lugares que dependem dele. Unindo os dois conceitos, conseguimos desenvolver um sistema muito mais fácil de manter e estender.