Inversão de Controle: Cadê a minha chave de fenda?

Quando o assunto é organização de classes e objetos, as dependências entre as diversas partes do sistema são um assunto delicado, e devemos nos esforçar ao máximo para diminuir o acomplamento.

Para ilustrar a discussão, podemos aproveitar um caso recorrente em diversos projetos de Software. Imagine que estamos tentando encapsular todo o acesso a dados de uma aplicação, centralizando todo este acesso em objetos específicos para este fim. Para tal, aplicaremos o design pattern *Data Access Object* (DAO):

```
public class ProjetoDAO {
    public void salva(Projeto projeto) {
        // ...
    }

    public void remove(Projeto projeto) {
        // ...
    }

    public Projeto carrega(Integer id) {
        // ...
    }
}
```

Partindo do princípio que estamos usando diretamente JDBC para o acesso ao banco, precisamos de uma conexão (`java.sql.Connection`) em todos os métodos do DAO. O uso de algum framework para acesso a dados ou de mapeamento objeto relacional não resolveria o problema já que precisaríamos de algo análogo a uma `Session` ou `EntityManager`.

O problema é que o DAO foi criado para encapsular os detalhes de acesso aos dados, porém precisa de alguns recursos (conexões, neste caso) para fazer o seu trabalho. Seria o mesmo caso com um `EnviadorDeEmails` que precisa de uma conexão com o servidor de emails, por exemplo.

Como podemos obter uma referência para `Connection`? Como fazer para conseguir estas dependências? A primeira alternativa seria abrir conexões diretamente nos métodos que precisam delas, deixando os detalhes do trabalho sujo por conta do próprio DAO:

```
public class ProjetoDAO {
    public void salva(Projeto projeto) throws ... {
        Class.forName("com.mysql.jdbc.Driver");
        String url = "jdbc:mysql://localhost/test";
        String usuario = "root";
        String senha = "123";

        Connection connection = DriverManager.getConnection(url, usuario, senha);

        // uso da conexão ...
```

```
        connection.close();
    }

    // outros métodos
}
```

O código acima serve para conectar em uma base do MySQL. É inviável repetir todo este código em todos os outros lugares que precisam de conexão. Além disso, ainda existem diversos fatos importantes a serem considerados, como o uso de um **pool de conexão** para gerenciamento mais inteligente dos recursos e externalização das configurações.

O primeiro passo para amenizar o problema é centralizar a responsabilidade de criar conexões em algum local. Desta forma, podemos depender apenas de uma interface e ignorar completamente os detalhes de criação de conexões (se a conexão é nova, se veio de um pool, qual driver estamos usando, etc). O uso de uma *factory* nos permite esconder e centralizar os detalhes de criação de uma Conexão e trocar essa estratégia mais adiante, caso necessário. Baixo acoplamento!

Mas não o suficiente. Discutiremos o assunto adiante, na seção *"Fábricas e o mito do baixo acoplamento"*.

```
public class ProjetoDAO {
    public void salva(Projeto projeto) throws SQLException {
        Connection connection = new ConnectionFactory().getConnection();

        // uso da conexão ...

        connection.close();
    }

    public void remove(Projeto projeto) throws SQLException {
        Connection connection = new ConnectionFactory().getConnection();

        // uso da conexão ...

        connection.close();
    }

    // outros métodos
}
```

O gerenciamento das conexões está bem mais simples, mas ainda não resolve o problema. Será que com o código deste jeito, conseguiríamos salvar dois `Projetos` usando a mesma conexão? E executar dois métodos do DAO dentro de uma mesma transação? Isto infelizmente não é possível pois, a cada nova invocação de método, estamos adquirindo uma nova conexão. Essa é uma má prática conhecida como *Sessão por Operação*, *Transação por Operação*, ou ainda *Conexão por Operação* (*Session/Connection per Operation*).

Para tentar resolver o problema, alguns projetos tentam agrupar as diversas operações que precisam da mesma conexão (ou transação) dentro de métodos do DAO:

```
public class ProjetoDAO {
    public void salvaDois(Projeto p1, Projeto p2) {
        Connection connection = new ConnectionFactory().getConnection();
        // ....
        connection.close();
    }

    public void salvaVarios(Projeto ... projetos) {
        Connection connection = new ConnectionFactory().getConnection();
        // ...
        connection.close();
    }

    public void salvaAtualizaERemove(Projeto aSalvar, Projeto aAtualizar, Projeto aRemover) {
        Connection connection = new ConnectionFactory().getConnection();
        // ...
        connection.close();
    }
}
```

Com quantos métodos terminaremos neste DAO? Este é um claro exemplo de péssima divisão de responsabilidades, já que o DAO faz coisas demais: abre conexão, cuida de transação, fecha conexão, além da tarefa que realmente lhe cabe: acessar dados. Cada novo tipo de transação da aplicação exige um novo método no DAO. Na Caelum, apelidamos esse padrão de "classe **Sistema**", que geralmente tem muitas linhas.

De alguma forma precisamos permitir que a mesma conexão seja usada em diversos métodos do DAO, portanto guardá-la como atributo. Mas onde abrir a conexão?

```
public class ProjetoDAO {
    private Connection connection;

    public void inicia() {
        this.connection = new ConnectionFactory().getConnection();
    }
}
```

Criar um método para iniciar o DAO e abrir a conexão resolve o problema, mas cria outro. Qualquer um pode esquecer de chamar o método `inicia()`, fazendo com que `Connection` seja `null`. O nosso objeto DAO não é um **Bom Cidadão** (*Good Citizen Pattern*), já que não veio ao mundo com todas suas dependências preparadas e **pronto para servir**.

O princípio do **Bom Cidadão** [10] diz que devemos sempre deixar nossos objetos em um estado consistente. Para isso, o uso do construtor é essencial, onde são preenchidas todas as dependências e configurações necessárias para que o objeto possa

trabalhar adequadamente, sem a subsequente necessidade de invocar *setters* ou outros métodos de inicialização e configuração. O princípio ainda vai além, definindo algumas boas práticas na manipulação de exceções, logging e programação defensiva, com o objetivo de que esses objetos sejam "seguros" em relação a código de terceiros.

Usando o construtor para resolver o problema e forçando a aquisição da conexão no momento da instanciação do DAO, chegamos no seguinte código:

```
public class ProjetoDAO {
    private Connection connection;

    public ProjetoDAO() {
        this.connection = new ConnectionFactory().getConnection();
        // pode abrir uma transação
    }

    public void fecha() {
        // poderiamos consolidar a transação
        this.connection.close();
    }
}
```

A conexão é aberta no construtor, porém Java não tem destrutores. Mesmo que tivesse, ou usássemos o método `finalize()`, não seria a solução já que nunca temos a garantia de quando um objeto será coletado. É fundamental termos controle total sobre onde conexões são abertas e fechadas, tendo em vista que esse é um objeto "caro" como arquivos, threads, sockets e outros que usam de I/O ou chamadas ao sistema operacional.

O método `fecha` parece ser a solução, já que agora podemos instanciar o DAO e invocar várias operações dentro da mesma conexão/transação. Ainda assim essa solução possui alguns poblemas:

- O método `fecha()` quebra o encapsulamento, já que expõe um detalhe específico desta implementação de DAO que estamos usando: existe uma conexão que deve ser fechada. Se trocarmos a implementação do DAO para persistir em arquivos XML, por exemplo, não haveria mais necessidade de ter o método `fecha()`, pois não existiria mais nenhuma conexão a ser fechada;

- A boa prática diz: *"se eu abri, eu fecho"*! Desta forma, evitamos espalhar a responsabilidade de gerenciar o recurso que estamos manipulando. O problema é que o DAO abre a conexão, mas não sabe quando fechá-la e delega esta tarefa a quem usa o DAO;

  Os usuários da classe DAO ficam responsáveis por invocar o método `fecha()` e saber onde fechar algo que nem foram eles que abriram. A responsabilidade de gerenciar conexões fica assim espalhada pelo sistema. Todos são responsáveis por saber que uma conexão existe, quando é aberta e quando deve ser fechada;

- Não há erros de compilação quando alguém esquece de chamar o método `fecha()`, portanto ninguém é obrigado a chamá-lo. Como a responsabilidade se espalha por

todo o sistema, a chance de alguém esquecer de fechar a conexão aumenta muito (para não dizer que é inevitável). Isso gera os clássico problemas de vazamento de conexões, que não são fechadas e podem resultar no desastroso efeito do banco rejeitando novas conexões quando existirem muitas abertas.

Todos estes problemas são graves, mas se ainda não são convincentes o suficiente, aqui vai mais um: com esta abordagem ainda não é possível salvar um `Projeto`, atualizar um `Usuario`, e deletar uma `Historia` usando a mesma conexão, ou dentro da mesma transação. Como cada DAO tem a sua própria conexão, não há como compartilhar a mesma entre vários deles.

E como resolver o problema? Conheça a história do Alberto e o princípio da inversão de controle.

Alberto, o apertador de parafusos

Alberto é um apertador de parafusos. Esta é a sua responsabilidade e ele veio ao mundo apenas para fazer isto (*Single Responsibility Principle* [33]). Gelson é o seu gerente e costuma pedir:

- Alberto, aperte este parafuso aqui!

Como um funcionário dedicado, Alberto sempre vai com muita boa vontade atender. Ao chegar perto do parafuso, Alberto percebe que está sem chave de fenda. Ou seja, para fazer o seu trabalho, Alberto precisa antes de tudo de uma chave de fenda (assim como o DAO precisa de uma conexão).

A primeira alternativa do Alberto é ir pegar a chave de fenda na sala 143. Dentro desta sala, a chave fica no armário do canto esquerdo, de número 75. Como está sempre trancado, Alberto sabe que a chave fica na portaria da empresa, com o Joaquim.

Após pegar a chave com o Joaquim, ir até a sala 143 e abrir o armário 75, Alberto pega o estojo verde da direita, que contém a chave de fenda, retira a chave de lá e pode, enfim, apertar o parafuso para seu gerente!

O grande problema aqui é que o Alberto precisa saber muito mais do que veio ao mundo para fazer. Precisa saber o número da sala e do armário, precisa saber que a chave está na portaria e conhecer o Joaquim, caso contrário ele não conseguiria a chave. Além de tudo, precisa saber que a chave está no estojo verde da direita. Um trabalhão e só para poder apertar um parafuso. Tudo isto só porque ele precisava de uma chave de fenda.

Pior ainda, Alberto não é o único que precisa da chave de fenda. O pedreiro também usa, o encanador, o eletricista, etc. Todos eles precisam saber como obter uma chave de fenda. O que acontece se alguém mudar a chave de fenda de lugar? É preciso avisar todo mundo! Alto acoplamento!

E se alguém esquece de devolver a chave? E se o gerente comprar mais chaves de fenda para poder paralelizar o trabalho e as coloca em outros armários? O grande problema é que todos devem saber como pegar uma chave de fenda. Os objetos vão atrás daquilo que precisam para fazer o seu trabalho.

A outra alternativa de Alberto, quando descobre que está sem chave de fenda, é cruzar os braços:

- Não trabalho sem uma chave de fenda.

Desta forma, parece que o gerente Gelson é quem terá de ir atrás da chave de fenda para entregar ao Alberto. Mas Gelson não precisa fazer isso sozinho: ele pode pedir ao Manoel, que é o *pegador de chaves de fenda*! Assim temos a chance de dividir responsabilidade no sistema. Chance esta, que não havia antes.

Antes todos eram responsáveis por saber como pegar uma chave. Agora podemos centralizar esta responsabilidade em apenas um local. É importante que o **ciclo de vida** dos nossos recursos sejam tratados em locais centralizados, em especial quando esses recursos são caros.

Ao dizer que os objetos não vão mais atrás de suas depedências, mas que agora devem apenas recebê-las de alguém, estamos invertendo o controle. **Inversão de Controle** é apenas uma questão de postura com relação ao desenvolvimento. Ao invés de fazer os nossos objetos irem atrás daquilo que precisam, fazemos com que recebam tudo já pronto e mastigado.

Antes a classe `ProjetoDAO` era responsável por **controlar** a abertura e o fechamento da conexão, assim como, possivelmente, o destino das transações envolvidas. **Invertendo o controle** [19], em vez do DAO ser responsável pelo ciclo de vida e detalhes sobre a conexão e transação, o seu invocador será o novo responsável por isso, já entregando para ele todas as dependências (a chave de fenda) mastigadas.

O código do DAO fica mais simples e enxuto, pois agora não é preciso se preocupar mais com os detalhes da conexão e transação:

```
public class ProjetoDAO {
    private Connection connection;

    public ProjetoDAO(Connection connection) {
        this.connection = connection;
    }

    // métodos do DAO

}
```

IoC, Injeção de Dependências e Testes

Inversão de controle é uma técnica há muito tempo discutida [34], e que hoje aparece bastante dado os **containers de inversão de controle** [24].

Existem diversas maneiras de inverter o controle de uma aplicação. Até aqui, vimos a inversão através da **injeção de dependências por construtor**, considerada uma das formas mais elegantes por, além de tudo, respeitar o princípio do Bom Cidadão. Mas há outros tipos, como a injeção por setter ou diretamente em atributos.

Até mesmo os EJBs apresentam alto grau de inversão de controle desde suas primeiras versões (o que não necessariamente indica facilidade durante a codificação).

Mas quem fará a **ligação** (*wiring*) entre a conexão e nosso DAO? Existe a possibilidade de se trabalhar com vários frameworks que possam fazer essa **injeção** (outro termo para ligação; *ligar*, *amarrar* e *injetar* são termos que aparecem com frequência). **Injeção de dependências é uma das formas de implementar inversão de**

controle, mas não a única.

Na injeção de dependências, temos um dependente (o DAO) e uma dependência (a conexão). Quem amarra tudo isso é um provedor (*provider*). Esse provedor pode ser desde um service locator, uma factory, ou ainda um framework. Repare que não há a necessidade sempre de um framework, tudo pode ser amarrado por código escrito direto pelo programador; **o importante é isolar o gerenciamento do ciclo de vida dos objetos, de quem depende deles**.

Os primeiros frameworks a trabalharem com inversão de controle usavam essas diferentes táticas para trabalhar com as dependências: Apache Avalon através de *service locators*, Spring Framework e XWork através de injeção por *setters* e Pico Container através de injeção por *construtor*.

As novas especificações de JSF e EJB possuem mais recursos para injeção de dependências, e há ainda a discussão entre a JSR 299, *Contexts and Dependency Injection* - antiga WebBeans e inspirada no JBoss Seam, e a JSR 330, *Dependency Injection for Java* - apelidada de `@Inject`. A JSR 299 tem Gavin King, que também é o criador do JBoss Seam, como líder e especifica a injeção de dependências dentro de contêiners, tendo como alvo o Java EE 6.0.

Já a JSR 330 especifica a injeção de dependências fora de contêiners, tendo como alvo o Java SE 7.0. Nasceu de um acordo entre os líderes das bibliotecas de injeção de dependências mais famosas do mercado: Paul Hammant (PicoContainer), Bob Lee (Guice) e Rod Johnson (Spring framework). Apesar das divergências entre os criadores das duas especificações, o JCP é enfático ao dizer que as duas JSRs precisam convergir para algo comum e não disputarem entre si.

Como trabalhar com alguns desses frameworks, suas vantagens e desvantagens, será visto em uma seção posterior. Inversão de controle, em especial por injeção de dependências, esta não só intimamente ligado com o desacoplamento de classes, como também ao uso de testes de unidade. Como veremos, trabalhar constantemente com testes de unidade costuma implicar no uso de injeção de dependências, caso contrário a dificuldade de testar as classes isoladamente torna-se uma enorme barreira: o código fica muito acoplado.