FIAP ON

FUNDAMENTOS JAVA

PERSISTÊNCIA OO (GENERIC DAO)

JOSÉ YOSHIRIRO

9

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

9 PERSISTÊNCIA OO (GENERIC DAO)

9.1 Padrão de projetos DAO

No capítulo anterior, vimos que as operações com o banco de dados devem ser invocadas a partir de um objeto do tipo **EntityManager**. Porém, imagine um sistema com dezenas ou centenas de tabelas, todas mapeadas para entidades ORM. Onde você colocaria as invocações para esses métodos? Para ajudar a organizar projetos assim, existe um padrão de projeto chamado **DAO** (**Data Access Object**, ou "objeto de acesso a dados", em tradução livre). A autoria desse padrão é da Sun Microsystems (comprada em 2009 pela Oracle), que publicou um catálogo de padrões para aplicações corporativas chamado **Core J2EE Patterns**.

9.2 O que propõe o DAO

O **DAO** (**D**ata **O**bject **A**ccess) fornece uma camada de abstração entre a camada de lógica de negócios e a camada de acesso ao banco de dados. Os objetos de negócios acessam o banco de dados por meio de objetos de acesso a dados.

A orientação é criar uma classe DAO para cada tabela que o sistema acessa. Por exemplo, se o sistema acessa uma tabela chamada **tipo_estabelecimento**, devemos ter uma classe chamada **TipoEstabelecimentoDAO**. Então, as operações de criação, alteração, exclusão e consultas de registros a essa tabela ficariam centralizadas nessa classe. Um exemplo de como seria esse DAO pode ser observado no Código-fonte *Exemplo de classe "DAO" ("TipoEstabelecimentoDAO")*.

```
public class TipoEstabelecimentoDAO {

   public void inserir(TipoEstabelecimento novo) {
      // instruções para inserir no banco
   }

   public void atualizar(TipoEstabelecimento novo) {
      // instruções para atualizar no banco
   }
   public TipoEstabelecimento recuperar(Integer id) {
      // instruções para recuperar do banco
```

```
        }

        public List<TipoEstabelecimento> listar() {
           // instruções para recuperar todos do banco
        }

        public void excluir(Integer id) {
           // instruções para excluir do banco
        }

    }
```

Código-fonte 9.1 – Exemplo de classe "DAO" ("TipoEstabelecimentoDAO")
Fonte: Elaborado pelo autor (2017)

Os métodos de uma classe DAO recebem e devolvem objetos que possuam atributos compatíveis com os campos da tabela do DAO. Por esse motivo, usar DAO em conjunto com o JPA pode ser muito eficiente, pois já temos classes que "espelham" as tabelas, que são as entidades de ORM.

9.3 DAO x Repository

É bem provável que você já tenha lido ou venha a ler sobre o padrão de projetos **Repository**, que tem a mesma finalidade do **DAO** e com implementação muito parecida. A diferença é que a proposta original do DAO era que ele recebesse e retornasse objetos que não fossem de entidades, e, sim, de classes "intermediárias" (normalmente, chamadas de **DTOs – D**ata **T**ransfer **O**bjects). No Repository, as entidades são usadas como parâmetros e/ou retornos dos métodos. Vamos criar nossos DAOs usando diretamente entidades, para evitar que o capítulo se estenda muito e para facilitar o entendimento do padrão.

9.4 O que é um Generic DAO?

Observe novamente o Código-fonte *Exemplo de classe "DAO" ("TipoEstabelecimentoDAO")*. Um DAO para outra entidade seria muito diferente daquele? Os métodos teriam assinaturas e corpos muito parecidos, trocando-se as referências **TipoEstabelecimento** por referências da entidade desejada.

Assim, como diferentes DAOs para diferentes entidades teriam muita coisa em comum, cria-se uma superclasse que contém essas semelhanças, normalmente

chamada de **GenericDAO**. Os DAOs para as entidades do projeto apenas as estenderiam e sobrescreveriam os métodos que tivessem um comportamento diferente do padrão e/ou teriam novos métodos para operações específicas.

9.5 Exemplo de Generic DAO

No Código-fonte *Exemplo de classe "GenericDAO",* temos um exemplo de Generic DAO e, na sequência, a explicação completa de seu funcionamento.

```
package br.com.fiap.smartcities.dao;

import java.util.List;
import java.lang.reflect.ParameterizedType;

import javax.persistence.EntityManager;
import javax.persistence.criteria.CriteriaQuery;

public abstract class GenericDAO<E,C> {

     protected Class<E> classeEntidade;

     protected EntityManager em;

     public GenericDAO(EntityManager em) {
          this.em = em;
          this.classeEntidade = (Class<E>)
((ParameterizedType)
this.getClass().getGenericSuperclass()).getActualTypeArg
uments()[0];
     }

     public void salvar(E entidade) {
          this.em.persist(entidade);
     }

     public E recuperar(C chave) {
          return this.em.find(classeEntidade, chave);
     }

     public List<E> listar() {
          CriteriaQuery<E> query =
this.em.getCriteriaBuilder().createQuery(this.classeEnti
dade);
          CriteriaQuery<E> select =
query.select(query.from(this.classeEntidade));
```

```
            return
      this.em.createQuery(query.select(query.from(this.classeE
      ntidade))).getResultList();
           }

           public void excluir(C chave) {
                E entidadeExcluir = this.recuperar(chave);

                if (entidadeExcluir == null) {
                     throw new IllegalArgumentException(
                               "Nenhum registro de " +
      this.classeEntidade.getSimpleName() + "encontrado para a
      chave " + chave);
                }

                this.em.remove(entidadeExcluir);
           }

      }
```

Código-fonte 9.2 – Exemplo de classe “GenericDAO”
Fonte: Elaborado pelo autor (2017)

Note que a GenericDAO apresentada é **abstrata**, o que significa que não poderá ser instanciada. Assim, precisamos de classes **concretas** (ou seja, não abstratas) que a estendam para poder usar um DAO. Logo mais veremos exemplos de DAOs concretos.

Os valores de ***Generics*** ao lado do nome da classe **<E, C>** servirão para indicar os tipos da entidade e da chave primária que o DAO manipulará. Evitamos muita repetição de código apenas usando esse recurso.

Note que **E** e **C** são utilizados em todos os métodos. Sempre onde houver o **E**, significa que estamos lidando com o tipo da Entidade (tabela), e onde houver **C**, significa que estamos lidando com o tipo da chave primária. Lembre-se de que nem sempre as chaves primárias são números, pois as chaves primárias podem ser compostas. **Mas tem que ser E e C? Não.** Poderia ser o par de letras (ou até palavras) que preferir.

Os atributos **classeEntidade** e **em** são protegidos (**protected**), o que significa que serão visíveis para as subclasses de GenericDAO.

O construtor da classe recebe um argumento chamado **em** do tipo **javax.persistence.EntityManager**, já visto no capítulo anterior. A primeira coisa que faz é mudar o valor do atributo de instância **em**, fazendo-o receber o **em** do

argumento. Depois, há um código que parece um tanto complexo, mas, resumidamente, ele serve para indicar o valor do atributo **classeEntidade** com a classe da entidade do DAO. Esse tipo de código, muito provavelmente, só será usado no construtor do GenericDAO, por isso, não se preocupe em memorizá-lo.

O método **salvar()** recebe um argumento do tipo da Entidade do DAO e persiste ele no banco. Lembrando que o método **persist()** do JPA insere ou atualiza um registro no banco, dependendo de encontrar ou não o registro cuja chave primária seja a mesma da instância da Entidade recebida.

O método **recuperar()** recebe um argumento do tipo da Chave Primária da Entidade e recupera, caso exista, o registro do banco numa instância do tipo da Entidade. Para tentar recuperar o registro do banco, foi usado o método **find()** do JPA.

O método **listar()** solicita ao JPA que recupere todos os registros da tabela mapeada para a Entidade, retornando uma lista de Entidades. Caso nenhum registro seja encontrado, uma **lista vazia** (e não **null**) será retornada. Nesse método, fizemos uma consulta usando um recurso do JPA chamado **CriteriaQuery**. Com ele, podemos realizar consultas sem escrever nenhuma linha de **SQL**. Na primeira linha do método, solicitamos uma **CriteriaQuery** do tipo Entidade do DAO. Na segunda, apenas indicamos que queremos todos os registros da tabela mapeados para a classe da Entidade e que queremos o resultado na forma de **java.util.List** do Java.

O método **excluir()** recebe um argumento do tipo da Chave Primária da Entidade para excluir o registro do banco. Primeiro, ele tenta recuperar a Entidade a partir da Chave do argumento, usando o método recuperar() do próprio DAO. Caso o registro não exista, o método é interrompido pelo lançamento de uma **IllegalArgumentException** com a mensagem indicando que não foi encontrado registro para a Chave informada. Caso o registro seja encontrado, é excluído com a invocação do método remove() do JPA.

Com os métodos que possui, o nosso GenericDAO implementa o chamado **CRUD** (Create, Read, Update, Delete – “Criar, Ler, Atualizar, Excluir”, em tradução livre), ou seja, o conjunto de operações básicas realizadas em um banco de dados.

9.6 Extensões do Generic DAO

No tópico anterior, vimos um exemplo de GenericDAO abstrato. Mas ele por si só não nos possibilita trabalhar com nenhuma Entidade. Precisamos criar classes concretas, informado o tipo de Entidade e da Chave Primária.

9.6.1 DAO para TipoEstabelecimento

No Código-fonte *Classe "TipoEstabelecimentoDAO",* temos um exemplo de como seria um DAO para a entidade **TipoEstabelecimento**, a mesma vista nos capítulos anteriores. Vamos chamá-la de **TipoEstabelecimentoDAO**.

```
package br.com.fiap.smartcities.dao;

import javax.persistence.EntityManager;

import
br.com.fiap.smartcities.domain.TipoEstabelecimento;

public class TipoEstabelecimentoDAO extends
GenericDAO<TipoEstabelecimento, Integer> {

     public TipoEstabelecimentoDAO(EntityManager em) {
          super(em);
     }

}
```

Código-fonte 9.3 – Classe "TipoEstabelecimentoDAO"
Fonte: Elaborado pelo autor (2017)

Do jeito que está no Código-fonte *Exemplo de classe "GenericDAO",* anterior, uma instância de **TipoEstabelecimentoDAO** já consegue realizar o **CRUD** para a tabela mapeada na Entidade **TipoEstabelecimento**.

Note que, ao lado do nome da classe, indicamos que **TipoEstabelecimento** seria a classe da Entidade que ficará sob a responsabilidade da **TipoEstabelecimentoDAO** e que a classe da Chave Primária é do tipo **Integer** (o atributo **id**, anotado com **@Id** na **TipoEstabelecimento,** é do tipo **Integer**).

Por fim, o Java nos obriga a criar um construtor que recebe um **EntityManager**, porque existe um assim na superclasse abstrata. Basicamente, criamos e indicamos

que ele simplesmente imita o comportamento do construtor definido na superclasse (a GenericDAO).

9.6.2 DAO para avaliação

Para deixar mais claro como criar DAOs que estendem a GenericDAO, vamos criar outro para a entidade **Avaliacao** (a mesma do capítulo anterior) e chamá-lo de **AvaliacaoDAO** (vide Código-fonte *Classe "AvaliacaoDAO"*):

```
import br.com.fiap.smartcities.domain.Avaliacao;
import br.com.fiap.smartcities.domain.AvaliacaoId;

public class AvaliacaoDAO extends GenericDAO<Avaliacao,
AvaliacaoId>{

     public AvaliacaoDAO(EntityManager em) {
          super(em);
     }

}
```

Código-fonte 9.4 – Classe "AvaliacaoDAO"
Fonte: Elaborado pelo autor (2017)

Repare que, ao lado do nome da classe, indicamos que **Avaliacao** seria a classe da Entidade que ficará sob a responsabilidade da **AvaliacaoDAO** e que a classe da Chave Primária é do tipo **AvaliacaoId** (o atributo **id**, anotado com **@Id** na **Avaliacao**, é do tipo **AvaliacaoId**, sendo uma chave composta).

O construtor é semelhante ao **TipoEstabelecimentoDAO**. E assim será em qualquer outro DAO que criarmos como subclasse do GenericDAO.

9.7 Testando os DAOs criados a partir do GenericDAO

Para entender como usar DAOs criados a partir do GenericDAO, vamos criar uma classe base chamada **DAOTeste** no pacote **br.com.fiap.smartcities.testes**. Ela terá um método **main()** para que possa ser executada. Seu "esqueleto" pode ser visto no Código-fonte *Classe "DAOTeste" usada para testar nossos DAOs*.

```
public class DAOTeste {

     public static void main(String[] args) {
```

```
            EntityManager em = null;
            try {
                 em =
    Persistence.createEntityManagerFactory("smartcities").cr
    eateEntityManager();
                  TipoEstabelecimentoDAO dao = new
    TipoEstabelecimentoDAO(em);
                 em.getTransaction().begin();

                // os próximos códigos fonte ficarão aqui

                 em.getTransaction().commit();

            } catch (Exception e) {
                 e.printStackTrace();
                 em.getTransaction().rollback();
            } finally {
                 if (em != null) {
                     em.close();
                 }
                 System.exit(0);
            }
```

Código-fonte 9.5 – Classe “DAOTeste” usada para testar nossos DAOs
Fonte: Elaborado pelo autor (2017)

Observe como foi instanciado o objeto **dao** do tipo **TipoEstabelecimentoDAO** na segunda instrução dentro do bloco **try**.

No bloco **catch**, apenas solicitamos o cancelamento (*rollback*) da transação caso algum problema ocorra no banco de dados.

No bloco **finally**, solicitamos que o **em** seja fechado, caso tenha sido inicializado, e o encerramento da aplicação. O bloco **finally** sempre é executado, independentemente se o bloco **catch** foi executado ou não.

Os quatro Códigos-fonte *Invocando o método “salvar()” de um D*AO até *Invocando o método “excluir()” de um DA*, a seguir, devem ser colocados entre as instruções em.getTransaction().begin() e em.getTransaction().commit().

O **Código-fonte *Invocando o método “salvar()” de um DAO*** exemplifica como solicitar a criação de um registro de **TipoEstabelecimento** usando seu DAO.

```
    TipoEstabelecimento novoTipoEstabelecimento = new
    TipoEstabelecimento();

    novoTipoEstabelecimento.setNome("Parque");
```

```
dao.salvar(novoTipoEstabelecimento);
```

Código-fonte 9.6 – Invocando o método “salvar()” de um DAO
Fonte: Elaborado pelo autor (2017)

O método **salvar()** do **dao** só aceita objetos como o **TipoEstabelecimento**. Se tentar usar outra entidade, a IDE acusará erro de compilação. Após a execução do código com esse trecho de código, será criado um registro na tabela **tipo_estabelecimento**.

O Código-fonte *Invocando o método “recuperar()” de um DAO* exemplifica cvomo solicitar a recuperação de um registro de **TipoEstabelecimento** usando seu DAO.

```
TipoEstabelecimento entidade  = dao.recuperar(1);

if (entidade == null) {
     System.out.println("Não existe tipo de
estabelecimento para a chave 1");
}
else {
     System.out.println(" > " + entidade.getId() + " - "
+ entidade.getNome());
}
```

Código-fonte 9.7 – Invocando o método “recuperar()” de um DAO
Fonte: Elaborado pelo autor (2017)

O método **recuperar()** do **dao** só aceita objetos do tipo **Integer**. Se tentar usar objetos de outro tipo, a IDE acusará erro de compilação.

Usamos o valor **1** como argumento do **recuperar()** do **dao** apenas como exemplo. Quando for testar, use algum valor de Chave Primária que sabe existir no banco e verá na saída os valores **id** e **nome** de **TipoEstebelecimento**. Caso use uma Chave que não exista, será exibida a mensagem “Não existe tipo de estabelecimento para a chave X”.

O Código-fonte *Invocando o método “listar()” de um DAO* exemplifica somo solicitar todos os registros de **TipoEstabelecimento** usando seu DAO.

```
System.out.println("\nTipos de Estabelecimento:");

for (TipoEstabelecimento entidade : dao.listar()) {
     System.out.println(" > " + entidade.getId() + " - "
+ entidade.getNome());
}
```

Código-fonte 9.8 – Invocando o método “listar()” de um DAO
Fonte: Elaborado pelo autor (2017)

O método **listar()** do **dao** sempre devolverá uma lista (**java.util.List**), seja ela vazia, seja preenchida. Caso contenha itens, todos serão como os **TipoEstabelecimento**. Para cada registro encontrado, o programa exibirá na saída os valores **id** e **nome** de **TipoEstebelecimento**.

O Código-fonte *Invocando o método "excluir()" de um DAO* exemplifica como solicitar todos os registros de **TipoEstabelecimento** usando seu DAO.

```
dao.excluir(3);
```

Código-fonte 9.9 – Invocando o método "excluir()" de um DAO
Fonte: Elaborado pelo autor (2017)

O método **excluir()** do **dao** só aceita objetos do tipo **Integer**. Se tentar usar objetos de outro tipo, a IDE acusará erro de compilação.

Ao invocar o método **excluir()**, caso a Chave Primária indicada no argumento seja um valor de uma chave que realmente exista na tabela de **TipoEstebelecimento**, o registro será excluído. Caso contrário, tomaremos uma **IllegalArgumentException**, conforme vimos do Código-fonte *Exemplo de classe "DAO" ("TipoEstabelecimentoDAO")* deste capítulo.

Para lidar com um **dao** do tipo **AvaliacaoDAO**, a abordagem seria a mesma!

9.8 E se precisarmos de operações que não estão no GenericDAO?

Um GenericDAO nunca resolverá todos os problemas de uma aplicação que acessa várias tabelas no banco. Principalmente no que diz respeito a consultas, pois cada tabela em um banco de dados costuma ter consultas específicas para ela. Nesse caso, precisamos criar métodos nos DAOs concretos. Porém, a forma mais prática de criar consultas específicas é usando um conceito chamado JPQL, que será abordado no próximo capítulo.

REFERÊNCIAS

JBOSS.ORG. **Hibernate ORM 5.2.12. Final User Guide**. Disponível em: <https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html>. Acesso em: 20 out. 2017.

JENDROCK, Eric. **Persistence – The Java EE5 Tutorial**. Disponível em: <https://docs.oracle.com/javaee/5/tutorial/doc/bnbpy.html>. Acesso em: 20 out. 2017.

NETO, Oziel Moreira. **Entendendo e dominando o Java**. 3. ed. São Paulo: Universo dos Livros, 2012.

ORACLE. **Core J2EE Patterns – Data Access Object**. Disponível em: <http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>. Acesso em: 26 nov. 2017.

PANDA, Debu; RAHMAN, Reza; CUPRAK, Ryan; REMIJAN, Michael. **EJB 3 in Action**. 2. ed. Shelter Island: Manning Publications, 2014.