FRAMEWORKS JAVA, .NET & WEBSERVICES

GENERALIZANDO O DAO

2A

LISTA DE CÓDIGOS-FONTE

LISTA DE FIGURAS

SUMÁRIO

1 GENERALIZANDO O DAO

No ano passado, utilizamos o *Design Pattern* DAO para acessar o banco de dados e tivemos que criar um DAO para cada entidade presente no sistema. Sabia que existe uma forma de simplificar isso? Veremos agora um pouco sobre Generics e como podemos implementar em DAO, reaproveitando um mesmo DAO para várias entidades ao mesmo tempo!

1.1 PADRÃO DE PROJETOS DAO

No capítulo anterior, vimos que as operações no 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 é que existe um padrão de projeto chamado **DAO** (**Data Access Object**, ou "objeto de acesso a dados", em tradução livre). O autor desse padrão foi a Sun Microsystems (comprada em 2009 pela Oracle), que publicou um catálogo de padrões para aplicações corporativas chamado **Core J2EE Patterns**.

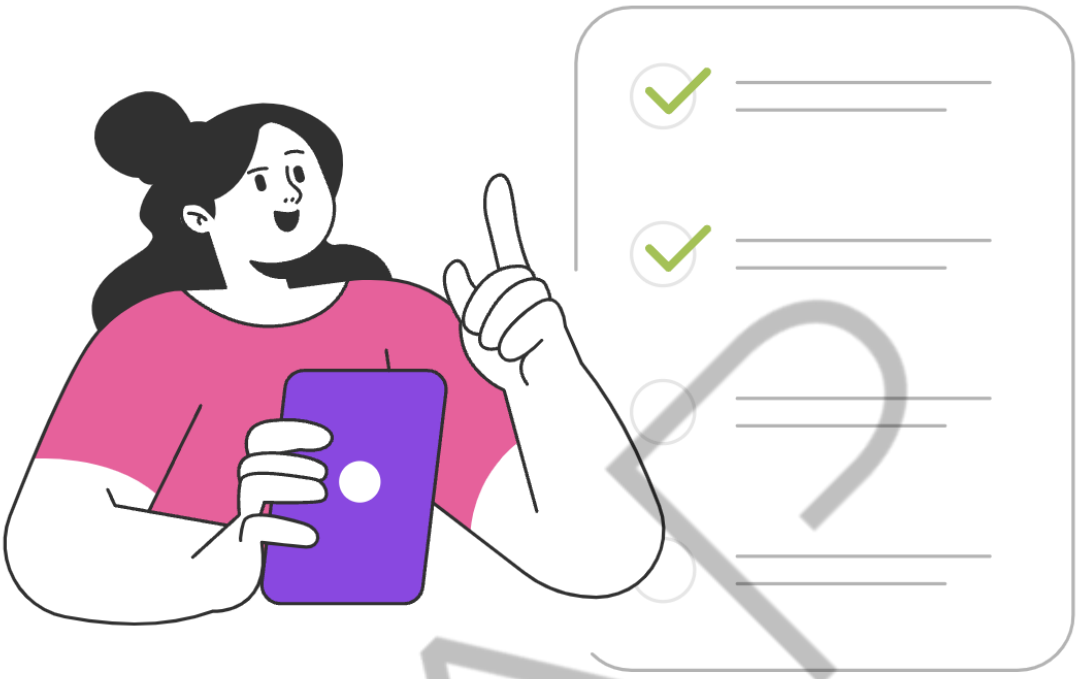

Figura 1 – Estudos do capítulo anterior
Fonte: Shutterstock (2018)

1.2 O que propõe o DAO?

O padrão DAO (Data Access Object) fornece uma camada de abstração que separa a lógica de negócios da camada de acesso ao banco de dados. Essa separação permite que os objetos de negócios interajam com o banco de dados por meio de objetos de acesso a dados, sendo muito usado por programadores.

Além disso, ele é amplamente utilizado em projetos de desenvolvimento de software com a finalidade de separar a lógica de negócios da camada de acesso ao banco de dados. Isso torna o código mais modular e fácil de manter, além de melhorar a escalabilidade e a eficiência do sistema como um todo. Alguns dos principais exemplos de uso do DAO incluem:

- Acesso a banco de dados: o DAO é usado para realizar operações de leitura, gravação e exclusão de dados em um banco de dados. Ele é especialmente útil para lidar com a complexidade da conexão ao banco de

dados, abstraindo as camadas de acesso e fornecendo uma API simples e coerente para a lógica de negócios.

- Gerenciamento de transações: o DAO é frequentemente usado para gerenciar transações de banco de dados. Por exemplo, ao realizar uma operação que envolve várias atualizações no banco de dados, o DAO pode garantir que a transação seja tratada de forma atômica, ou seja, que todas as atualizações sejam bem-sucedidas ou que nenhuma atualização seja realizada.
- Integração com outros sistemas: o DAO também pode ser usado para integrar diferentes sistemas ou serviços em um aplicativo. Por exemplo, um aplicativo pode usar um DAO para fazer uma solicitação a um serviço da web ou a um sistema de arquivos, abstraindo as complexidades do acesso aos diferentes sistemas.
- Testes de unidade: o DAO também é útil para testes de unidade, permitindo que os desenvolvedores criem testes automatizados para garantir que a lógica de negócios esteja funcionando corretamente sem precisar acessar o banco de dados real. Ao usar um DAO simulado ou em memória, os testes podem ser executados rapidamente e com maior controle e precisão.

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 dessa tabela ficariam centralizadas nessa classe. Um exemplo de como seria esse DAO está no código-fonte abaixo:

```
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

public class TipoEstabelecimentoDAO {

    public void inserir(TipoEstabelecimento novo) throws SQLException {
        try {
            // instruções para inserir no banco
        } catch (SQLException ex) {
            System.out.println("Erro ao inserir no banco de dados: " + ex.getMessage());
            throw ex;
        }
    }

    public void atualizar(TipoEstabelecimento novo) throws SQLException {
        try {
            // instruções para atualizar no banco
        } catch (SQLException ex) {
            System.out.println("Erro ao atualizar no banco de dados: " + ex.getMessage());
            throw ex;
        }
    }
```

```
    public TipoEstabelecimento recuperar(Integer id) throws SQLException {
        try {
            // instruções para recuperar do banco
            return new TipoEstabelecimento(); // retorno de exemplo
        } catch (SQLException ex) {
            System.out.println("Erro ao recuperar do banco de dados: " +
ex.getMessage());
            throw ex;
        }
    }

    public List<TipoEstabelecimento> listar() throws SQLException {
        try {
            // instruções para recuperar todos do banco
            return new ArrayList<TipoEstabelecimento>(); // retorno de exemplo
        } catch (SQLException ex) {
            System.out.println("Erro ao listar no banco de dados: " +
ex.getMessage());
            throw ex;
        }
    }

    public void excluir(Integer id) throws SQLException {
        try {
            // instruções para excluir do banco
        } catch (SQLException ex) {
```

```
            System.out.println("Erro ao excluir do banco de dados: " +
        ex.getMessage());
            throw ex;
        }
    }
}
```

Código-fonte 1 – Exemplo de classe “DAO” (“TipoEstabelecimentoDAO”)
Fonte: Elaborado pelo autor (2017)

No código acima adicionamos a palavra-chave "throws SQLException" na assinatura de cada método, indicando que esse método pode lançar uma exceção do tipo SQLException. Isso é necessário porque, em um ambiente de banco de dados, pode haver erros de execução, como falhas de conexão, problemas de sintaxe ou restrições de integridade de dados.

Em cada método adicionamos um bloco "try-catch", que envolve as instruções de banco de dados. Se ocorrer uma exceção durante a execução, o bloco "catch" será ativado e a mensagem de erro será impressa no console. Também usamos a instrução "throw ex" para relançar a exceção, para que o código que chamou o método possa lidar com ela de forma adequada. Isso também é importante porque permite que a

aplicação capture a exceção e realize uma ação apropriada, como informar o usuário sobre o erro.

Os métodos de uma classe DAO recebem e devolvem objetos que possuam atributos compatíveis com os campos da tabela do DAO. Daí o motivo de usar DAO em conjunto com o JPA, pois já temos classes que “espelham” as tabelas, as quais são as entidades de ORM.

1.3 O que é um Generic DAO?

Um Generic DAO é uma classe que pode ser utilizada como uma espécie de "template" para criar DAOs (Data Access Object) para diferentes entidades em um sistema. Com ele, é possível criar as operações básicas de acesso a banco de dados, como inserção, atualização, exclusão e consulta, sem a necessidade de reescrever esses métodos para cada entidade.

Usar um Generic DAO pode ser muito vantajoso, pois economiza tempo e esforço de desenvolvimento, além de facilitar a manutenção do código, já que as operações básicas estão centralizadas em um único lugar. Além disso, ao criar um DAO para uma nova entidade, basta herdar a classe genérica e personalizar os métodos, o que torna o processo de criação de DAOs muito mais ágil.

Porém, é importante lembrar que o Generic DAO não é uma solução para todos os problemas de acesso a banco de dados, especialmente quando se trata de consultas mais complexas ou específicas para uma determinada entidade. Nesses casos, é necessário criar métodos personalizados no DAO da entidade em questão ou utilizar ferramentas como JPQL para criar consultas mais avançadas.

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

Agora pense na implementação desses métodos, relembre do capítulo anterior, quais são as diferenças para realizar o CRUD da entidade TipoEstabelecimento para a entidade Estabelecimento? Também será só o tipo da referência que será passada para o Entity Manager, por exemplo, para cadastrar, basta utilizar o método

persist(objeto), informando o objeto que será cadastrado, que pode ser o TipoEstabelecimento, Estabelecimento ou qualquer outra entidade.

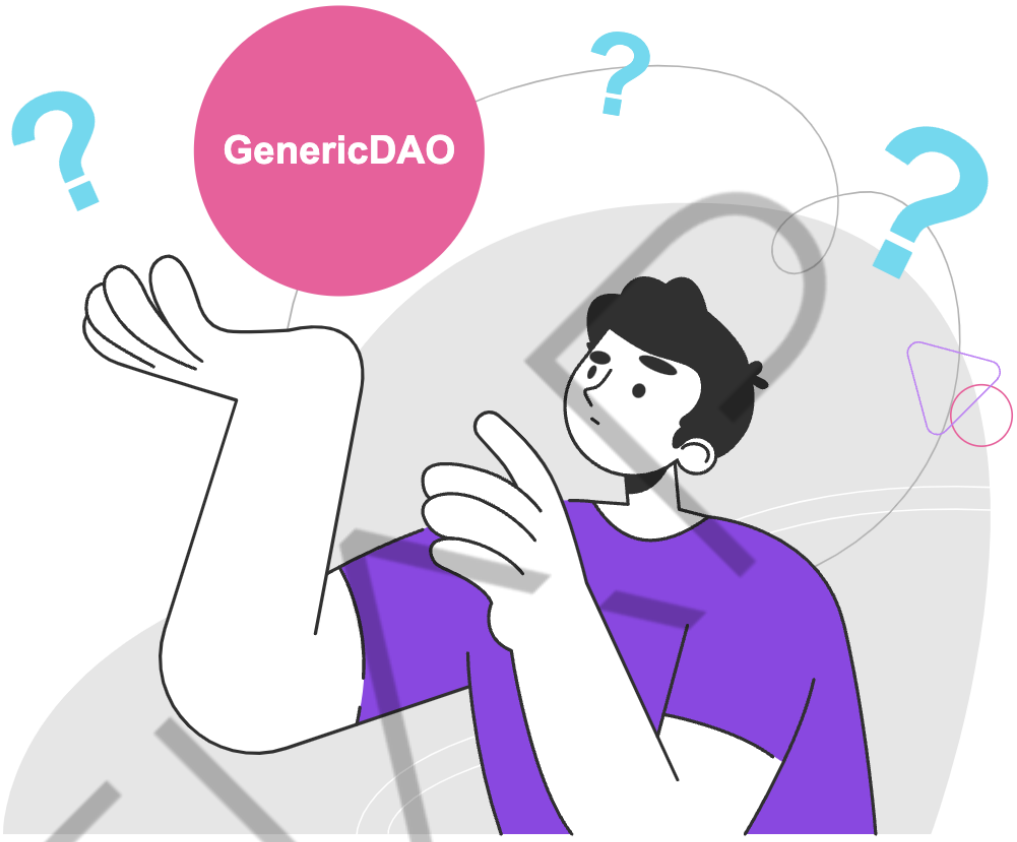

Figura 2 – O que é GenericDAO?
Fonte IconScout (2023)

Assim, como diferentes DAOs para diferentes entidades teriam muita coisa em comum, é melhor criar uma superclasse que contenha essas semelhanças, normalmente, chamada de **GenericDAO**. Os DAOs para as entidades do projeto apenas a estenderiam e sobrescreveriam os métodos que tivessem um comportamento diferente do padrão e/ou teriam novos métodos para operações específicas.

1.3.1 Exemplo de Generic DAO

No código-fonte a seguir, temos um exemplo de GenericDAO e, na sequência, a explicação completa de seu funcionamento:

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


import java.lang.reflect.ParameterizedType;
import java.util.List;
import javax.persistence.EntityManager;

public abstract class GenericDAO<T,K> {

    protected EntityManager em;

    private Class<T> clazz;

    @SuppressWarnings("all")
    public GenericDAO(EntityManager em) {
        this.em = em;
        clazz    =    (Class<T>)    ((ParameterizedType)
getClass()

    .getGenericSuperclass()).getActualTypeArguments()[0];
    }

    public void cadastrar(T entidade) {
        em.persist(entidade);
    }

    public void atualizar(T entidade) {
        em.merge(entidade);
    }

    public void excluir(K codigo) throws Exception {
        T entidade = buscar(codigo);
        if (entidade == null){
            throw new Exception("Codigo invalido");
        }
        em.remove(entidade);
    }

    public List<T> listar(){
        return      em.createQuery("from      "      +
clazz.getName(), clazz).getResultList();
    }

    public T buscar(K codigo) {
        return em.find(clazz,codigo);
    }

    public void commit() throws Exception {
        try{
```

```
                    em.getTransaction().begin();
                    em.getTransaction().commit();
                }catch(Exception e){
                    if (em.getTransaction().isActive())
                        em.getTransaction().rollback();
                    e.printStackTrace();
                    throw new Exception("Erro no commit");
                }
            }


        }
```

Código-fonte 2 – Exemplo de classe “GenericDAO”
Fonte: Elaborado pelo autor (2017)

O código acima implementa um DAO genérico para acesso a um banco de dados através do Hibernate. A classe abstrata GenericDAO usa o tipo genérico T para representar a entidade que será persistida no banco de dados e o tipo genérico K para representar a chave primária da entidade.

O construtor da classe recebe um objeto EntityManager, que é usado para realizar operações no banco de dados. Ele também usa a reflexão para obter o tipo da entidade que está sendo manipulada e armazená-lo na variável clazz.

Os métodos cadastrar, atualizar e excluir permitem realizar as operações básicas de CRUD em uma entidade, enquanto o método listar retorna uma lista de todas as entidades do tipo T armazenadas no banco de dados. O método buscar permite recuperar uma única entidade do tipo T por sua chave primária.

O método commit() realiza a confirmação de uma transação pendente. Ele verifica se uma transação está ativa, executa o commit e, em caso de exceção, executa um rollback e lança uma exceção indicando que ocorreu um erro no commit.

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.

Presentes na linguagem Java a partir da versão 1.5, Generics possibilitam que um tipo de dado seja passado como parâmetro para uma classe ou método, tendo um resultado diferente para aquele trecho de código, dependendo do valor passado.

Generics são verificados em tempo de compilação e não tempo de execução, como, por exemplo, um type cast ou operador instanceof.

No exemplo do nosso Generic Dao, o compilador irá verificar o tipo que estamos passando para T e K, interrompendo o processo caso seja um valor inválido.

Os valores de ***Generics*** ao lado do nome da classe, **<T, K>**, servirão para indicar os tipos da Entidade (Table) e da Chave Primária (Key) que o DAO irá manipular. Só com o uso desse recurso evitamos muita repetição de código.

Note que **T** e **K** são usados em todos os métodos. Sempre onde houver o **T**, significa que estamos lidando com o tipo da Entidade. E onde houver **K**, significa que estamos lidando com o tipo da chave primária. Lembre-se de que nem sempre as chaves primárias são números, porque podem ser compostas. Assim, a letra **K** poderia ser do tipo de uma classe, com a AvaliacaoId, por exemplo. **Mas tinham que ser T e K? Não.** Poderia ser o par de letras (ou até palavras) que você preferir, talvez, possamos utilizar E e C, para Entidade e Chave.

Os atributos **clazz** 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**, que 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 **clazz** com a classe da entidade do DAO, ou seja, ele recupera o tipo da Entidade (Estabelecimento, TipoEstabelecimento etc.) que o GenericDao irá trabalhar. Esse tipo de código, muito provavelmente, só será usado no construtor do GenericDAO. Por isso, não se preocupe se em memorizá-lo.

O método **cadastrar()** recebe um argumento do tipo da Entidade do DAO e persiste no banco de dados. Lembrando que o método **persist()** do JPA insere um novo registro no banco de dados. A implementação é bem simples! Somente uma linha de código! Será que não está faltando nada?

Sim! Para realizar alguma alteração no banco de dados, precisamos trabalhar com transações (commit), porém, não é recomendado colocá-las dentro da implementação do método, pois não teríamos como controlar várias operações no

banco com a mesma transação. É por isso que criamos um método chamado **commit()**, para abrir e finalizar uma transação com o banco de dados. Observe que esse método lança uma exceção, caso algo de errado aconteça no commit.

O método **atualizar()** é muito parecido com o método cadastrar(), só muda o método utilizado do JPA, o merge(). Um detalhe interessante é que esse método também é capaz de cadastrar uma entidade caso ela não exista. Assim, outra forma de implementação seria só um método **salvar(),** que cadastra ou atualiza a entidade, dependendo se a entidade existe ou não no banco de dados. Podendo, assim, remover os métodos cadastrar() e atualizar().

O método **buscar()** 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 **JPQL**. Com ele, podemos realizar consultas customizadas no banco de dados.

Não se preocupe com a sintaxe neste momento, logo mais teremos um capítulo dedicado a esse assunto. Porém, é simples de entender a implementação, criamos uma *query* com JPQL para buscar todos os registros da entidade na base de dados. E com o método getReturnList(), executamos a *query*, recebendo 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 buscar() do próprio DAO. Caso o registro não exista, o método é interrompido pelo lançamento de uma **Exception** com a mensagem indicando que não foi encontrado registro para a Chave informada. Aqui podemos melhorar o código com a criação de uma *exception* específica, que mostra que não foi encontrada a entidade para a remoção. Caso o registro seja encontrado, é excluído com a invocação do método remove() do JPA.

Figura 3 – Programador
Fonte: IconScout (2023)

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

2 EXTENSÕES DO GENERIC DAO

De forma geral, o GenericDAO abstrato é um template que fornece a base para (quando falo sobre um "template", estou me referindo a um padrão ou modelo que pode ser utilizado como ponto de partida para a criação de outras coisas.) a implementação de classes DAO concretas que manipulam entidades do banco de dados. Ele é capaz de fornecer os métodos básicos de CRUD (Create, Read, Update, Delete), além de métodos genéricos para consultas e transações.

No entanto, o GenericDAO sozinho não é capaz de trabalhar com nenhuma entidade do banco de dados, pois ele não sabe qual classe de entidade ou chave primária ele deve manipular. Por isso, é necessário criar classes concretas, que herdam do GenericDAO abstrato e informam qual é a entidade e a chave primária a serem manipuladas.

Dessa forma, as classes concretas estendem o GenericDAO abstrato e implementam os métodos específicos de cada entidade, como consultas e atualizações. Com isso, o GenericDAO abstrato se torna um molde reutilizável para a criação de classes DAO específicas, evitando a necessidade de reescrever código comum para cada entidade a ser manipulada no banco de dados.

2.1 DAO para TipoEstabelecimento

No código-fonte abaixo, 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> {

    // Construtor que recebe um EntityManager e chama o construtor da classe pai com o mesmo EntityManager
```

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

}
```

Código-fonte 3 – Classe “TipoEstabelecimentoDAO”
Fonte: Elaborado pelo autor (2017)

A classe TipoEstabelecimentoDAO estende a classe GenericDAO, que é uma classe genérica que fornece as operações básicas de persistência para entidades JPA.

O tipo genérico TipoEstabelecimento representa a entidade que será persistida no banco de dados e o tipo genérico Integer representa o tipo da chave primária dessa entidade.

A classe TipoEstabelecimentoDAO não contém nenhum método adicional, mas herda todos os métodos da classe pai. Isso permite que sejam realizadas operações de persistência no banco de dados para a entidade TipoEstabelecimento, incluindo a inserção, atualização, exclusão, busca por chave primária e listagem de todos os registros dessa entidade no banco de dados.

Ao adicionar os comentários, fica mais fácil entender o propósito da classe e o seu papel dentro do sistema. Isso é importante para que outros desenvolvedores possam entender o código e fazer modificações ou correções no futuro.

Do jeito que está no Código-fonte “Exemplo de classe “GenericDAO”, 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 **int**, porém, precisamos informar uma classe, por isso, usamos o Integer).

Por fim, o Java nos obriga a criar um construtor que recebe um **EntityManager**, porque existe um único construtor na superclasse abstrata que recebe um parâmetro do tipo EntityManager. Basicamente, criamos e indicamos que ele simplesmente imita o comportamento do construtor definido na superclasse (a GenericDAO).

2.2 DAO para Avaliação

Para deixar mais claro como criar DAOs que estendam a GenericDAO, vamos criar outro para a entidade **Avaliacao** (a mesma do capítulo anterior) e chamá-lo de **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 4 – Classe “AvaliacaoDAO”
Fonte: Elaborado pelo autor (2017).

Assumindo que a classe GenericDAO fornece as operações básicas de persistência para entidades JPA e que a classe Avaliacao representa a entidade que será persistida no banco de dados, enquanto AvaliacaoId representa o tipo composto da chave primária da entidade.

A classe AvaliacaoDAO estende a classe GenericDAO, herdando todos os métodos de persistência de entidades da classe pai, incluindo inserção, atualização, exclusão, busca por chave primária e listagem de todos os registros dessa entidade no banco de dados.

O construtor da classe recebe um EntityManager e chama o construtor da classe pai passando o mesmo EntityManager.

Note 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** (essa é a classe que mapeia os atributos que formam a chave composta da entidade).

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

2.3 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" está no código-fonte a seguir:

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

	import javax.persistence.EntityManager;
	import javax.persistence.Persistence;

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

	public class DAOTeste {

	    public static void main(String[] args) {
	        EntityManager em = null;
	        try {
	            em                                          =
Persistence.createEntityManagerFactory("smartcities-
orm").createEntityManager();
	            TipoEstabelecimentoDAO    dao    =    new
TipoEstabelecimentoDAO(em);


	            //Implementações que utilizam os métodos
do DAO


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

	    }
```

```
    }
```

Código-fonte 5 – Classe “DAOTeste” usada para testar nossos DAOs
Fonte: Elaborado pelo autor (2017).

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

No bloco **catch** simplesmente lançamos uma exceção, caso aconteça algo de errado. Em uma aplicação “real” com interface gráfica, podemos tratar a *exception* para exibir uma mensagem mais amigável ao usuário.

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 do bloco **catch** ter sido executado ou não.

Os códigos-fonte a seguir devem ser colocados dentro do bloco **try**, depois de instanciar a classe TipoEstabelecimenntoDAO.

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

```
TipoEstabelecimento tipo = new TipoEstabelecimento();
                tipo.setNome("Bar e Restaurante");

                dao.cadastrar(tipo);
                dao.commit();
```

Código-fonte 6 – Invocando o método “salvar()” de um DAO
Fonte: Elaborado pelo autor (2017)

O método **cadastrar()** do **DAO** só aceita objetos do tipo **TipoEstabelecimento**. Se tentar usar outra entidade, a IDE já acusará erro de compilação. Precisamos também chamar o método *commit* para efetivar o cadastro no banco de dados. Assim, após a execução desse trecho de código, um registro novo será criado na tabela **tipo_estabelecimento**.

O **Código-Fonte “Invocando o método “recuperar()” de um DAO”** exemplifica como solicitar a recuperação de um registro de **TipoEstabelecimento** usando seu DAO:

```
TipoEstabelecimento entidade  = dao.buscar(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 7 – Invocando o método “recuperar()” de um DAO
Fonte: Elaborado pelo autor (2017)

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

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

Observe que **não** é necessário chamar o método commit() nesse tipo de operação.

O **Código-Fonte “Invocando o método “listar()” de um DAO”** exemplifica como 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 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 vazia, seja preenchida. Caso contenha itens, todos serão do tipo **TipoEstabelecimento**. Para cada registro encontrado, o programa exibirá na saída os valores **id** e **nome** de **TipoEstabelecimento.**

O **Código-Fonte “Invocando o método “excluir()” de um DAO”** exemplifica como remover um registro do **TipoEstabelecimento** usando seu DAO:

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

Código-fonte 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 já 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, vamos receber uma **Exception**, com a mensagem “Código Inválido”.

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

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

Embora um GenericDAO possa ser uma solução genérica para muitos casos de acesso a banco de dados, ele pode não atender a todas as necessidades de uma aplicação que acessa várias tabelas. Especialmente quando se trata de consultas, cada tabela em um banco de dados geralmente tem necessidades específicas que não podem ser resolvidas por um único método genérico.

Assim, é importante criar métodos específicos nos DAOs concretos para atender a essas necessidades. Uma forma prática de criar essas consultas específicas é através do uso de JPQL (Java Persistence Query Language), que é um conceito que será abordado no próximo capítulo. Com JPQL, podemos escrever consultas mais elaboradas que permitem uma maior flexibilidade e poder na recuperação de dados.

Figura 4 – Banco de dados
Fonte: IconScout (2023)

A solução completa para a implementação do GenericDAO pode ser baixada em:

Git: https://github.com/FIAP/smartcities-orm/tree/05-generic-dao

Zip: https://github.com/FIAP/smartcities-orm/archive/refs/heads/05-generic-dao.zip

REFERÊNCIAS

JENDROCK, Eric. **Persistence – The Java EE 5 Tutorial.** [s.d.]. Disponível em: <https://docs.oracle.com/javaee/5/tutorial/doc/bnbpy.html>. Acesso em: 12 jan. 2021.

MIHALCEA, Vlad et al. **Hibernate ORM 5.2.12. Final User Guide**. [s.d.]. Disponível em: <https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html>. Acesso em: 12 jan. 2021.

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

ORACLE. **Core J2EE Patterns – Data Access Object**. [s.d.]. Disponível em: <http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>. Acesso em: 12 jan. 2021.

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