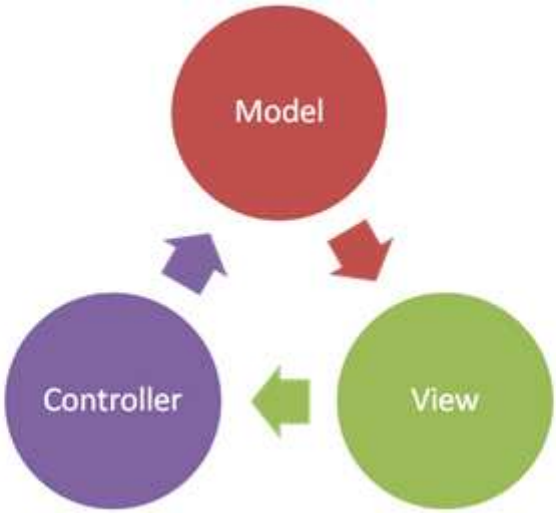
Design Pattern: criando uma classe DAO genérica

Há algum tempo atrás era muito comum encontrar a lógica de persistência de dados junto das regras de negócio. Isto aumentava consideravelmente a complexidade de entendimento e manutenção do projeto. Com o passar do tempo as técnicas de desenvolvimento amadureceram e deram origem a alguns padrões de projeto fortemente utilizados nos dias de hoje, como por exemplo o padrão de projeto MVC (Model – View – Controller). Este design pattern divide a aplicação em camadas com responsabilidades específicas, sendo:

- **Model**: responsável por abrigar as lógicas de persistência e conversão de dados do SGDB em objetos da aplicação, de forma genérica ou não.
- **View**: objetivo de abrigar todas as informações visuais contendo entradas de dados, regras de visualização, entre outros.
- **Controller**: responsável por aplicar as regras de negócio da aplicação.



No contexto acima, é possível comparar o design pattern **DAO (Data Access Object)** com a camada Model do padrão MVC, uma vez que o mesmo surgiu da necessidade de separar as lógicas de persistência das lógicas de negócio da aplicação. Este padrão tem como objetivo trocar informações de forma independente com o SGBD e fornecer os conceitos básicos para a realização de CRUDs ou recursos mais complexas. Quando aplicado de forma correta, toda a necessidade de busca e cadastro de informações é abstraída, tornando o processo de manipulação de dados das entidades transparente às demais camadas do sistema, ou seja, uma classe Pessoa pode ter um DAO dedicado a realizar operações específicas no SGBD, como por exemplo, consultar pessoas por CPF, idade, etc.

O problema

Levando em consideração que uma classe X pode ter uma representação XDAO responsável por executar as suas lógicas de persistência, imagine se houver 100, 150, 1000 entidades que necessitam de integração com o banco de dados. Seria necessário criar a mesma quantidade de classes DAO para executar as regras de persistência destas entidades?

A solução

Graças ao conceito de Generics presente no Java, é possível criar uma classe DAO que será capaz de abstrair um tipo qualquer de entidade da aplicação e executar comandos específicos, eliminando assim os chamados boilerplates (código clichê). Desta forma, se um número N de entidades do sistema tem características semelhantes, ao invés de se criar N classes DAO, cada uma representando um modelo, a aplicação passaria a ter apenas **uma classe genérica** abstraindo as funcionalidades em comum entre um grupo específico de objetos que necessitam de comunicação com o banco de dados.

Obs: Para minimizar o esforço de criação das queries de banco de dados e elevar a produtividade deste artigo, será utilizada a biblioteca JPA (Java Persistence API) a qual abstrai os métodos de CRUD de forma simples e legível.

Recursos necessários

1. IDE de sua preferência
2. JARs da biblioteca JPA, implementação Hibernate
3. Servidor MYSQL 5
4. JAR do conector MYSQL

Estrutura do artigo

1. Criação do projeto de exemplo e configuração da biblioteca JPA
2. Criando a classe de conexão com o banco de dados
3. Criação das classes de modelo
4. Criação da classe DAO genérica
5. Testando a classe DAO genérica

1. Criação do projeto de exemplo e configuração da biblioteca JPA

Crie um projeto Java e em seguida crie o diretório META-INF na pasta raiz. Esta pasta será responsável por hospedar o arquivo de configuração da JPA, o *persistence.xml*.

Antes de criar este arquivo, faça o download dos JARs do Hibernate com JPA e o conector MYSQL. Feito o download, acrescente-os no build path do projeto.

O *persistence.xml* é responsável por definir os parâmetros de funcionamento da JPA e deve conter informações como: nome da unidade de persistência, string de conexão com o bando de dados, usuário, senha, nome do banco, driver de conexão, entre outros. Como o objetivo do artigo não é esmiuçar este arquivo, segue abaixo o modelo básico para completar este tutorial.

persistence.xml XHTML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
 xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persi

 <persistence-unit name="dao-generico" transaction-type="RESOURCE_LOCAL">
 <provider>org.hibernate.ejb.HibernatePersistence</provider>

 <properties>
 <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />
 <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/dao" ></prop
 <property name="javax.persistence.jdbc.user" value="root" />
 <property name="javax.persistence.jdbc.password" value="root" />
 <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
 <property name="hibernate.connection.shutdown" value="true" />
 <property name="hibernate.hbm2ddl.auto" value="update" />
 <property name="hibernate.show_sql" value="false" />
 <property name="hibernate.format_sql" value="false"/>
 </properties>

 </persistence-unit>
</persistence>
```

2. Criando a classe de conexão com o banco de dados

Agora crie a classe que será responsável por estabelecer a conexão com o banco de dados utilizado pelo sistema.

ConnectionFactory

```java
package br.com.raphaelneves.connection;

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

public class ConnectionFactory {

 private static EntityManagerFactory factory = Persistence.createEntityManagerFactory("dao-g

 public static EntityManager getEntityManager(){
 return factory.createEntityManager();
 }

}
```

Por ser um recurso caro para a aplicação, o atributo factory será utilizado com o modificador non-access static, ou seja, este atributo será único para toda instância produzida da classe ConnectionFactory. Desta forma, a factory será criada apenas na primeira vez em que a classe ConnectionFactory for utilizada.

3. Criação das classes de modelo

As classes modelo indicam a representação de entidades que necessitam de integração com o banco de dados. Para isto, crie uma interface que estipula o método getId(). Este método será responsável por retornar a chave primária das entidades "clientes", que por sua ver é do tipo Long. A interface EntidadeBase será o contrato das entidades da aplicação que precisam utilizar recursos do SGBD, ou seja, toda e qualquer entidade que trabalhe com persistência de dados deverá "assinar" este contrato.

EntidadeBase.java

```
1 public interface EntidadeBase{
2  public Long getId();
3 }
```

Sabendo que as entidades modelo terão o contrato de EntidadeBase assinado, é possível amarrar a especificação da classe DAO genérica para que ela aceite todo tipo de entidade que seja implemente esta interface e atenda aos requisito de IS-A EntidadeBase. Crie duas classes, Pessoa e Carro, implementando a interface EntidadeBase e especifique os atributos e comportamenos conforme abaixo:

Pessoa.java

```
1  package br.com.raphaelneves.pojo;
2
3  import javax.persistence.Column;
4  import javax.persistence.Entity;
5  import javax.persistence.GeneratedValue;
6  import javax.persistence.GenerationType;
7  import javax.persistence.Id;
8  import javax.persistence.Table;
9
10 @Entity
11 @Table(name="tb_pessoa")
12 public class Pessoa implements EntidadeBase {
13
14  private Long id;
15  private String nome;
16  private int idade;
17
18  @Id
19  @GeneratedValue(strategy=GenerationType.AUTO)
20  @Column(name="id")
21  public Long getId() {
22  return id;
23  }
24
25  public void setId(Long id) {
26  this.id = id;
27  }
```

```
28
29  @Column(name="nome")
30  public String getNome() {
31  return nome;
32  }
33
34  public void setNome(String nome) {
35  this.nome = nome;
36  }
37
38  @Column(name="idade")
39  public int getIdade() {
40  return idade;
41  }
42
43  public void setIdade(int idade) {
44  this.idade = idade;
45  }
46
47 }
```

Carro.java

```
1  package br.com.raphaelneves.pojo;
2
3  import javax.persistence.Column;
4  import javax.persistence.Entity;
5  import javax.persistence.GeneratedValue;
6  import javax.persistence.GenerationType;
7  import javax.persistence.Id;
8  import javax.persistence.Table;
9
10 @Entity
11 @Table(name="tb_carro")
12 public class Carro implements EntidadeBase {
13
14  private Long id;
15  private String modelo;
16  private int anoFabricacao;
17
18  @Id
19  @GeneratedValue(strategy=GenerationType.AUTO)
20  @Column(name="id")
21  public Long getId() {
22  return id;
23  }
24
25  public void setId(Long id) {
26  this.id = id;
27  }
28
29  @Column(name="modelo")
30  public String getModelo() {
31  return modelo;
32  }
33
34  public void setModelo(String modelo) {
35  this.modelo = modelo;
36  }
37
38  @Column(name="ano_fabricacao")
39  public int getAnoFabricacao() {
```

```
40  return anoFabricacao;
41  }
42
43  public void setAnoFabricacao(int anoFabricacao) {
44  this.anoFabricacao = anoFabricacao;
45  }
46
47 }
```

As anotações @Entity, @Table, @Column, @Id, @GeneratedValue são recursos fornecidos pela JPA a fim de especificar que uma determinada classe seja identificada como uma classe modelo para manipulação de dados junto ao SGBD. Em palavras mais grosseiras, estas anotações fazem com que a classe represente uma entidade, tabela ou coluna no banco de dados.

4. Criação da classe DAO genérica

Agora que a aplicação é capaz de gerar uma conexão com o banco de dados e mapear as entidades de modelo, crie uma classe com o nome de DaoGenerico. Esta classe será responsável por centralizar as lógicas de persistência em comum entre o grupo de entidades do contrato EntidadeBase. Este artigo abordará os métodos para adicionar, modificar, pesquisar por chave primária e excluir um registro no banco de dados, o famoso CRUD.

```
1 public class DaoGenerico<T extends EntidadeBase> {}
```

Esta declaração de classe permite gerar uma instância de DaoGenerico representando qualquer entidade que assine o contrato EntidadeBase (*T extends EntidadeBase*). Em seguida, defina um atributo manager, do tipo EntityManager e com modificador non-access static. Este atributo permitirá a utilização dos métodos da JPA.

Para facilitar e simplificar o entendimento, o ciclo de vida do EntityManager ***não será gerenciado.***

DaoGenerico.java

```
1 public class DaoGenerico<T extends EntidadeBase> {
2
3  private static EntityManager manager = ConnectionFactory.getEntityManager();
4 }
```

Um breve exemplo de como instanciar um objeto de DaoGenerico representando a entidade Pessoa.

```
1 DaoGenerico<Pessoa> daoPessoa = new DaoGenerico<Pessoa>();
```

A JPA possui um método capaz de realizar a busca se baseando pelo tipo de uma entidade modelo e sua chave primária, no caso o atributo ID das classes de modelo. A sintaxe desse método é:

```
1 manager.find(Entidade.class, chavePrimaria);
```

Crie o método de busca, findById, recebendo como parâmetro o tipo da entidade modelo desejada e o ID da mesma. Além disto, será definido como retorno uma entidade do tipo T. Observe que T é a entidade modelo que contratou os serviços de EntidadeBase.

```
1 public class DaoGenerico<T extends EntidadeBase> {
2
3  private static EntityManager manager = ConnectionFactory.getEntityManager();
4
5  public T findById(Class<T> clazz, Long id){
6  return manager.find(clazz, id);
7  }
8 }
```

Os métodos de cadastrar e editar possuem um comportamento muito parecido, salvar algo no banco de dados. Sendo assim, é possível verificar se o argumento do método possui um ID nulo ou não. Caso a verificação seja verdadeira, a entidade será salva, caso contrário atualizada.

Para isto, o método saveOrUpdate será criado sem retorno, pois o objetivo no momento é apenas inserir/atualizar o registro no banco, e receberá como parâmetro uma entidade do tipo T. Imagine que a entidade modelo tenha várias outras entidades que são persistidas em cascata. Para garantir a integridade dos dados no banco, será aplicado o conceito de transação. Desta forma, se houver algum erro no processo de persistência em cascata, será feito o rollback de todo o processo envolvido neste contexto.

```
1  public class DaoGenerico<T extends EntidadeBase> {
2
3   private static EntityManager manager = ConnectionFactory.getEntityManager();
4
5   public T findById(Class<T> clazz, Long id){
6   return manager.find(clazz, id);
7   }
8
9   public void saveOrUpdate(T obj){
10  try{
11  manager.getTransaction().begin();
12  if(obj.getId() == null){
13  manager.persist(obj);
14  }else{
15  manager.merge(obj);
16  }
17  manager.getTransaction().commit();
18  }catch(Exception e){
19  manager.getTransaction().rollback();
20  }
21  }
22 }
```

O método de exclusão funcionará um pouco diferente dos demais, pois se faz necessário realizar uma busca da entidade que será removida para somente então removê-la. Infelizmente este é um ciclo implementado pelo próprio método remove() da JPA. O método remove da classe genérica não terá

retorno e receberá como parâmetro o tipo da classe modelo e a chave primária da mesma, no caso o ID. Em seguida, o método de busca que já está criado será acionado e retornará a entidade que será excluída.

DaoGenerico

```
1  public class DaoGenerico<T extends EntidadeBase> {
2
3   private static EntityManager manager = ConnectionFactory.getEntityManager();
4
5   public T findById(Class<T> clazz, Long id){
6   return manager.find(clazz, id);
7   }
8
9   public void saveOrUpdate(T obj){
10  try{
11  manager.getTransaction().begin();
12  if(obj.getId() == null){
13  manager.persist(obj);
14  }else{
15  manager.merge(obj);
16  }
17  manager.getTransaction().commit();
18  }catch(Exception e){
19  manager.getTransaction().rollback();
20  }
21  }
22
23  public void remove(Class<T> clazz, Long id){
24  T t = findById(clazz, id);
25  try{
26  manager.getTransaction().begin();
27  manager.remove(t);
28  manager.getTransaction().commit();
29  }catch (Exception e) {
30  manager.getTransaction().rollback();
31  }
32  }
33
34 }
```

5. Testando a classe DAO genérica

Para testar os recursos genéricos de persistência crie as classes InsertApplication, UpdateApplication, FindByIdApplication e RemoveApplication, cada uma representando uma funcionalidade de CRUD na implementação do método main().

```
1  public class InsertApplication {
2
3   public static void main(String[] args) {
4
5   Pessoa pessoa = new Pessoa();
6   Carro carro = new Carro();
7
8   DaoGenerico<Pessoa> daoPessoa = new DaoGenerico<Pessoa>();
9   DaoGenerico<Carro> daoCarro = new DaoGenerico<Carro>();
```

```
10
11
12   pessoa.setNome("Raphael Neves");
13   pessoa.setIdade(28);
14
15   carro.setModelo("Mustang");
16   carro.setAnoFabricacao(1989);
17
18   daoPessoa.saveOrUpdate(pessoa);
19   daoCarro.saveOrUpdate(carro);
20
21   System.out.println("Entidades salvas com sucesso!");
22
23   }
24
25 }
```

```
INFO: HHH000232: Schema update complete
Entidades salvas com sucesso!
```

tb_carro | Enter a SQL expression to filter results (use Ctrl+

	id	ano_fabricacao	modelo
1	3	1.989	Mustang

tb_pessoa | Enter a SQL expression to filter res

	id	idade	nome
1	3	28	Raphael Neves

```
1  public class FindByIdApplication {
2
3   public static void main(String[] args) {
4
5   DaoGenerico<Pessoa> daoPessoa = new DaoGenerico<Pessoa>();
6   DaoGenerico<Carro> daoCarro = new DaoGenerico<Carro>();
7
8   Pessoa pessoa = daoPessoa.findById(Pessoa.class, 3L);
9   Carro carro = daoCarro.findById(Carro.class, 3L);
10
11  System.out.println("### Entidade Pessoa encontrada ###");
12  System.out.println("ID: " + pessoa.getId());
13  System.out.println("NOME: " + pessoa.getNome());
14
15  System.out.println("");
16
17  System.out.println("### Entidade Carro encontrada ###");
18  System.out.println("ID: " + carro.getId());
19  System.out.println("MODELO: " + carro.getModelo());
20
21
22  }
23
24 }
```

```
Nov 25, 2016 11:44:50 AM org.hibernate.tool.hbm2ddl.SchemaUpdate exec
INFO: HHH000232: Schema update complete
### Entidade Pessoa encontrada ###
ID: 3
NOME: Raphael Neves

### Entidade Carro encontrada ###
ID: 3
MODELO: Mustang
```

```
1  public class UpdateApplication {
2
3   public static void main(String[] args) {
4
5   DaoGenerico<Pessoa> daoPessoa = new DaoGenerico<Pessoa>();
6
7   Pessoa pessoa = daoPessoa.findById(Pessoa.class, 3L);
8   pessoa.setNome("Raphael Oliveira Neves");
9   daoPessoa.saveOrUpdate(pessoa);
10  System.out.println("Entidade atualizada com sucesso.");
11
12  }
13
14 }
```


```
1  public class RemoveApplication {
2   public static void main(String[] args) {
3
4   DaoGenerico<Pessoa> daoPessoa = new DaoGenerico<Pessoa>();
5   DaoGenerico<Carro> daoCarro = new DaoGenerico<Carro>();
6
7   daoPessoa.remove(Pessoa.class, 3L);
8   daoCarro.remove(Carro.class, 3L);
9
10  System.out.println("Entidades removidas com sucesso!");
11
12  }
13 }
```

Você pode baixar o código fonte deste artigo no meu GitHub.

Até a próxima!

Artigo publicado originalmente em Blog Raphael Neves