*Unisanta Science and Technology*, **2023, November**
Published Online 2023 Vol. 12 nº2 http://periodicos.unisanta.br/index.php/sat

**UNISANTA**
**Science and Technology**
**ISSN 2317-1316**

# Construção de um ORM como Estratégia Didático-Pedagógica no Ensino de Técnicas de Programação com Processamento de Metadados e Reflection

**Maurício Neves Asenjo, Alexandre Sobrino Ganança, Cláudio Souza Nunes, Fernando Augusto dos Santos Ribeiro, Helio Augusto de Lima Rangel, José Avelino dos Santos Moura, Paulo Roberto Schroeder de Souza.**

**UNISANTA** – Universidade Santa Cecília – Faculdade de Ciências Exatas, Engenharia e Arquitetura – Bacharelado em Sistemas de Informação e Tecnólogo em Análise e Desenvolvimento de Sistemas
Rua Oswaldo Cruz, 277 - Santos-SP, Brasil - CEP: 11045-907

**E-mail***:* asenjo@unisanta.br
*Received September*, 2023

**Resumo:** Este artigo descreve o desenvolvimento de uma interface BDOO para um banco de dados relacional, representando uma camada adicional a ser integrada em aplicações Java com o intuito de permitir a interação entre um aplicativo orientado a objetos e um banco de dados relacional para o ensino de técnicas de programação. São apresentados neste trabalho exemplos de código para cada método desenvolvido, incluindo o uso de reflexão, de modo a permitir a interação dinâmica com os tipos e propriedades dos objetos em tempo de execução, proporcionando aos estudantes a visão de um projeto real e útil para a exploração de conceitos abstratos, como metadados e a própria reflexão. A implementação dessa estratégia didático-pedagógica, do ponto de vista docente, resultou em um maior entusiasmo por parte dos alunos e em uma melhor assimilação dos conteúdos apresentados, algo passível de verificação através de pesquisa futura e específica.

**Palavras-chave:** programação orientada a objetos, banco de dados relacional, banco de dados orientado a objetos, mapeamento objeto-relacional, metadados, reflection.

# Building an ORM as a Didactic-Pedagogical Strategy in Teaching Programming Techniques with Metadata Processing and Reflection

**Abstract:** This article describes the development of a BDOO interface for a relational database, representing an additional layer to be integrated into Java applications in order to allow interaction between an object-oriented application and a relational database for teaching techniques of programming. Code examples are presented in this work for each method developed, including the use of reflection, allowing dynamic interaction with the types and properties of objects at runtime, providing students with the vision of a real and useful project for exploring abstract concepts, such as metadata and reflection itself. The implementation of this didactic-pedagogical strategy, from a teaching point of view, resulted in greater enthusiasm among students and a better assimilation of the content presented, something that can be verified through future and specific research.

**Keywords:** object-oriented programming, relational database, object-oriented database, object-relational mapping, metadata, reflection.

## 1. Introdução

Segundo Tetila (2021), um banco de dados relacional (BDR) é um tipo de sistema de gerenciamento de banco de dados (SGBD) que organiza dados em tabelas relacionadas. Cada tabela consiste em linhas e colunas, onde as linhas representam registros e as colunas representam atributos/campos. Normalmente, cada uma dessas tabelas possuí uma chave, também chamada de chave primária, que nada mais é do que um campo (ou um conjunto de campos) que identifica exclusivamente cada registro e garante que cada linha tenha uma identificação única.

As relações entre as tabelas são estabelecidas por meio de chaves estrangeiras, sendo um campo em uma tabela que faz referência à chave primária em outra tabela, definindo assim uma relação entre ambas. Umas das principais características de um BDR é a integridade referencial, que garante a consistência e a precisão dos dados ao estabelecer e manter relações e a consistência nos dados relacionados - assim, por exemplo, não é possível ter uma chave estrangeira que faz referência a uma chave primária inexistente (Guimarães, 2014). Ainda de acordo com o mesmo autor, SGBDs que suportam integridade referencial geralmente oferecem mecanismos automáticos para manter a consistência dos dados, de modo que se um registro relacionado for modificado, excluído ou adicionado, será realizado um ajuste automático, garantindo assim a integridade referencial. Outra característica importante de um BDR é que seu acesso é feito através de uma linguagem única e padronizada chamada de SQL (Structured Query Language), de modo que, independentemente da linguagem de programação utilizada para o acesso, isso será feito através de uma sentença SQL (Tetila, 2021).

De acordo com Nassu (2009), um banco de dados orientado a objetos (BDOO) é projetado para armazenar e manipular dados no formato de objetos, atuando como um grande repositório/container desses. Sua principal característica é uma maior aderência à teoria da programação orientada à objetos, tornando-se mais natural aos desenvolvedores familiarizados com a prática.

A comparação da performance entre bancos de dados relacionais e orientados a objetos varia conforme os requisitos do sistema, o modelo de dados e as consultas realizadas. Ainda assim, em cenários onde as consultas são predominantes e baseadas em relacionamentos simples entre tabelas, os BDRs geralmente apresentam melhor desempenho nas consultas pela possibilidade de criação de índices (Michaelides, 2001).

Um BDOR, ou Banco de Dados Objeto-Relacional, por sua vez, é um tipo de SGBD que combina características de bancos de dados relacionais tradicionais com conceitos da programação orientada a objetos – aproveitando, segundo Bhargavi (2013), as vantagens de ambas as tecnologias e fornecendo uma solução mais flexível e eficiente para a persistência de dados.

Este artigo descreve o desenvolvimento de uma interface BDOO para um banco de dados relacional, representando uma camada adicional a ser integrada em aplicações Java com o intuito de permitir a interação entre um aplicativo orientado a objetos e um banco de dados relacional para o ensino de técnicas de programação. Funcionando como ponte entre a lógica orientada a objetos no código do programa e a estrutura relacional do banco de dados, este tipo de ferramenta, conhecida como mapeamento objeto-relacional (ORM), visa automatizar o processo de mapeamento entre as entidades no código-fonte (objetos) e as tabelas no banco de dados relacional, eliminando a necessidade de escrita manual de consultas SQL para a inserção, atualização ou recuperação de dados, permitindo assim a persistência de dados provenientes de sistemas orientados a objetos no formato relacional.

## 2. Desenvolvimento

Para a construção dessa interface, uma das estratégias didático-pedagógicas para o ensino de técnicas de programação em uma das disciplinas do curso de Sistemas de Informação da Universidade Santa Cecília, optou-se pela utilização da linguagem de programação Java com o driver UCanAccess para o estabelecimento de conexão com o banco de dados. O exemplo que se segue foi, por exemplo, implementado através do software Microsoft Access, escolha feita única e exclusivamente em função da disponibilidade e simplicidade, destacando-se que qualquer outra plataforma pode ser utilizada.

Fortemente baseada no conceito de metadados, que nada mais são do que dados que fornecem informações sobre outros dados, a camada em questão lida com informações que descrevem características dos dados principais, como seu formato, estrutura, contexto e outros atributos relevantes (Campos, 2007). Como exemplo, considera-se a seguinte classe principal codificada em linguagem Java.

```
public class Livro {
private String titulo;
private String autor;
private String editora;

public Livro() {}
public void setTitulo(String _titulo) {titulo=_ti-
tulo;}
public void setAutor(String _autor) {autor=_autor;}
public void setEditora(String _editora) {editora=_edi-
tora;}

public String getTitulo() {return titulo;}
public String getAutor() {return autor;}
public String getEditora() {return editora;}
}
```

Denominada Livro, a classe conta com apenas três propriedades do tipo String (titulo, autor e editora), mas contempla diversos metadados, como o nome da classe, a quantidade de propriedades, seus nomes e tipos, quantidade de métodos, e também seus nomes e tipos, e se estes recebem ou não argumentos.

Considerando-se que a interface em questão visa automatizar a geração e execução de sentenças SQL, e que uma sentença SQL básica e válida do tipo Insert pode ser representada pelo código a seguir, serão necessários alguns metadados, como o nome da classe e suas propriedades para a composição do nome da tabela e suas colunas, que serão capturados através de uma técnica de programação denominada Reflection.

```
Insert Into <nome da tabela> (<coluna 1>, <coluna 2> ...
<coluna n>) Values (<dado 1>, <dado 2> ... <dado n>)
```

Reflection, ou reflexão em português, é um recurso avançado de programação que se refere à capacidade de um programa inspecionar, analisar e modificar sua própria estrutura durante o tempo de execução, examinando e interagindo dinamicamente com seus tipos, propriedades, métodos (metadados) e outros elementos em tempo de execução. Implementável em praticamente todas as linguagens modernas, orientadas a objetos e dinâmicas, permite, entre outras possibilidades e em tempo de execução, criar instâncias de classes, chamar métodos e acessar e modificar campos e propriedades. Por outro lado, o uso excessivo da reflexão pode tornar o código mais complexo, inclusive afetando o desempenho do programa em determinadas situações em comparação com operações diretas (Devmedia, 2022; Tudose et al, 2013), devendo ser usada com cuidado, de modo a garantir a manutenibilidade e a eficiência do software. Observa-se, a seguir, um trecho de código utilizado para refletir a classe Ex01:

```
public class Ex01 {
    private static int a;
    private static int b;

    public static void setA(int _a) {a=_a;}
    public static void setB(int _b) {b=_b;}
    public static int getA() {return a;}
    public static int getB() {return b;}
    public static int getSoma() {return a+b;}
}

import java.lang.reflect.*;

public class Reflection01 {

    public static void main(String[] args) {

        Ex01 obj = new Ex01();
        int qtdcampos = 0, qtdmetodos=0, i=0, j=0;

        // Descobrindo a classe do objeto obj
        System.out.println("Classe do objeto...........:
" + obj.getClass());


// Descobrindo quantas propriedades, campos,
// o objeto obj possuí

        for (Field f : obj.getClass().getDeclaredFi-
elds()) ++qtdcampos;
        System.out.println("Quantidade de propriedades.:
" + qtdcampos);

// Descobrindo quais são as propriedades do objeto obj

        for (Field f : obj.getClass().getDeclaredFi-
elds())
            System.out.println("Propriedade " + ++i +
"..............: " + f.getName() + " (" + f.getType()
+ ")");

// Descobrindo quantos métodos o objeto obj possuí

        for (Method m : obj.getClass().getDeclared-
Methods()) ++qtdmetodos;
        System.out.println("Quantidade de métodos......:
" + qtdmetodos);

// Descobrindo quais são os métodos do objeto obj

        for (Method m : obj.getClass().getDeclared-
Methods())
        {
            System.out.println("Método " + ++j +
"..............: " + m.getName() + " (" + m.getRe-
turnType() + ")");
            for (Type p : m.getParameterTypes())
                System.out.println("   Argumento: " + p);
        }
    }
```

É ainda importante observar que, na linguagem Java, as classes e métodos para a implementação dos conceitos de reflexão estão depositados no package java.lang.reflect.*, e que as classes Field, Method e Type são as responsáveis pela extração dos metadados do objeto "obj". Vê-se, abaixo, a tela resultante da execução do código anteriormente listado.

```
Classe do objeto...........: class reflection01.Ex01
Quantidade de propriedades.: 2
Propriedade 1..............: a (int)
Propriedade 2..............: b (int)
Quantidade de métodos......: 5
Método 1...............: setA (void)
   Argumento: int
Método 2...............: getA (int)
Método 3...............: getB (int)
Método 4...............: setB (void)
   Argumento: int
Método 5...............: getSoma (int)
```

## 3. Resultados e Discussão

Embora inicialmente tenha sido referenciada como uma camada de software a ser adicionada às aplicações, definiu-se desde o início que tal arquitetura seria composta por duas camadas distintas: a primeira, destinada ao processamento lógico e cujo objetivo seria o de automatizar a geração das sentenças SQL, e a segunda, de processamento físico, contemplando o acesso ao banco de dados e a respectiva persistência.

Como, em uma arquitetura de múltiplas camadas, a camada de acesso aos dados, conhecida como DAL (Data Access Layer) é onde se concentrarão os métodos para a

manipulação do banco de dados, as duas camadas foram batizadas de ALDAL e AFDAL, representando, respectivamente, as camadas de acesso lógico e físico, cujas funcionalidades serão a seguir explicadas.

```
public class ALDAL {

public static void geraTabela(Object obj)

public static void set(Object obj)

public static void delete(Object obj)

public static void update(Object dados, Object chaves)

public static void get(Object obj)
}

public class AFDAL {

public static void conecta(String _bd)

public static void desconecta()

public static void executeSQL(String _sql)

public static void executeSelect(String _sql, Object
obj)
}
```

Apresenta-se, a seguir, os métodos da ALDAL. O primeiro e mais simples de todos, denominado geraTabela, reflete o objeto recebido como argumento, extraindo o nome da classe e suas propriedades para a composição da sentença SQL.

```
public static void geraTabela(Object obj)
{
Field[] f  = obj.getClass().getDeclaredFields();
String sql = "Create Table Tab" + obj.getClass().get-
SimpleName() + " (";

for(int i=0; i<f.length; ++i)
{
   sql += f[i].getName() + " " + (f[i].getType().get-
SimpleName().equals("String")?"varchar(60)":f[i].ge-
tType());
   if (i!=(f.length-1)) sql = sql + ", ";
}
sql += ")";
System.out.println(sql);
AFDAL.conecta("Livros.mdb");
AFDAL.executeSQL(sql);
AFDAL.desconecta();
}
```

Destaca-se, no código anteriormente apresentado, a existência de um laço, implementado com a estrutura de controle de repetição for, responsável pela iteração envolvendo as propriedades do objeto e verificando se são do tipo string, uma vez que, em SQL, tal tipo é definido como varchar. Observa-se ainda que, uma vez que se trata de uma aplicação acadêmica para o ensino de técnicas de programação, antes que a sentença seja submetida ao banco de dados, a sentença gerada será exibida na tela, visando a conferência do resultado do método e que, a partir da transformação da camada aqui proposta em uma biblioteca para consumo por outras aplicações, será necessário excluir a linha que exibe a sentença gerada, além de considerar a possibilidade de erro no acesso físico ao banco.

No que diz respeito ao segundo método (set), é possível vislumbrar que parte do código é parcialmente semelhante ao anteriormente descrito geraTabela, sendo que, a partir de “values”, introduz-se uma nova funcionalidade - a lista de valores, que pode ser acessada pelo método get de cada propriedade. Tais valores podem ser refletidos ou gerados através da concatenação do prefixo "get" ao nome da propriedade. Cabe ainda destacar que optou-se pela segunda alternativa devido à existência prévia da propriedade.

```
public static void set(Object obj)
{
Field[] f  = obj.getClass().getDeclaredFields();
String sql = "Insert Into Tab" + obj.getClass().getSim-
pleName() + " (";
Method mtd;

for(int i=0; i<f.length; ++i)
{
   sql += f[i].getName();
   if (i!=(f.length-1)) sql = sql + ", ";
}
sql += ") values (";
for(int i=0; i<f.length; ++i)
{
   try
   {
   String  aux  =  "get"  +  f[i].getName().subs-
tring(0,1).toUpperCase() + f[i].getName().substring(1);
   mtd = obj.getClass().getMethod(aux);

   if                        (f[i].getType().getSimple-
Name().equals("String"))
      sql += "'" + mtd.invoke(obj) + "'";
   else
      sql += mtd.invoke(obj);
   }
   catch(Exception e) {}
   if (i!=(f.length-1)) sql = sql + ", ";
}
sql += ")";
System.out.println(sql);
AFDAL.conecta("Livros.mdb");
AFDAL.executeSQL(sql);
AFDAL.desconecta();
}
```

A parte anteriormente destacada refere-se à modificação no método geraTabela para, através de uma estrutura de controle de repetição for, navegar por cada propriedade do objeto, anexando o prefixo “get” a elas. Assim, dentro da variável auxiliar (aux), tem-se o nome do método a ser executado para a obtenção do dado correspondente.

Ocorre aqui, ainda, a introdução do conceito de “invoke”, através do qual o método desejado poderá ser chamado a partir de seu nome, referenciado por uma string. Tal recurso proporciona uma grande flexibilidade ao código e, dependendo do contexto, pode reduzir consideravelmente a quantidade de linhas de programação necessárias. É também importante ressaltar a necessidade de se incluir os valores de strings entre aspas simples, conforme requerido pela sintaxe SQL esperada. O método pode ser

testado a partir do seguinte script, cujo resultado é a seguir apresentado.

```
umlivro.setTitulo("2");
umlivro.setAutor("Asenjo");
umlivro.setEditora("UNISANTA");
umlivro.setAno("2022");
umlivro.setLocalizacao("Santos/SP");
ALDAL.set(umlivro);

Resultado em tela:

Insert Into TabLivro (titulo, autor, editora, ano, lo-
calizacao) values ('2', 'Asenjo', 'UNISANTA', '2022',
'Santos/SP')
```

São a seguir apresentados dois scripts de teste diferentes que, referentes ao método delete, ressaltam o processo de deleção em um banco de dados orientado a objetos.

```
Script de teste 01:

umlivro.setTitulo("Cem anos de solidão");
umlivro.setAutor(null);
umlivro.setEditora(null);
umlivro.setAno(null);
umlivro.setLocalizacao(null);
ALDAL.delete(umlivro);

Resultado em tela:

Delete from TabLivro where titulo = 'Cem anos de solidão'

Script de teste 02:

umlivro.setTitulo("Cem anos de solidão");
umlivro.setAutor(null);
umlivro.setEditora(null);
umlivro.setAno("1996");
umlivro.setLocalizacao(null);
ALDAL.delete(umlivro);

Resultado em tela:

Delete from TabLivro where titulo = 'Cem anos de solidão'
and ano = '1996'
```

Diferentemente dos dois métodos anteriores, tem-se aqui a construção da cláusula SQL “where”, composta pelos campos cujos conteúdos sejam diferentes de “null” e devidamente concatenados por “and”, compondo o código do método delete.

```
public static void delete(Object obj)
{
    Field[] f  = obj.getClass().getDeclaredFields();
    String sql = "Delete from Tab" + obj.getClass().ge-
tSimpleName() + " where ";
    Method mtd;
    String aux1, aux2;
    boolean flag = false;

    for (int i =0; i < f.length; ++i)
    {
        try
        {
        aux1 = "get" + f[i].getName().substring(0,1).toU-
pperCase() + f[i].getName().substring(1);
        mtd = obj.getClass().getMethod(aux1);
        aux2 = mtd.invoke(obj).toString();
        if (!aux2.equals(""))
        {
            if (flag) sql += " and "; else flag = true;
            sql += f[i].getName() + " = ";
            if                (f[i].getType().getSimple-
Name().equals("String"))
                sql += "'" + aux2 + "'";
            else
                sql += aux2;
        }
        }
        catch(Exception e) {}
    }

    System.out.println(sql);
    AFDAL.conecta("Livros.mdb");
    AFDAL.executeSQL(sql);
    AFDAL.desconecta();
}
```

Na prática, o método descrito também incorpora muitos dos recursos utilizados nos procedimentos anteriores. Na área em destaque, encontramos mais uma vez um laço no qual cada uma das propriedades é testada para determinar se possui conteúdo ou está nula. Vale observar a presença de um flag que sinalizará se, durante a geração do "where", deverá ser adicionado o "and" ou não.

A ordem na qual os métodos foram construídos foi didaticamente definida, e cada método apresentado até então possui uma particularidade em relação aos anteriores.

Seguindo essa estratégia, chega-se ao método de atualização, que, diferentemente dos anteriores, deve fazer a distinção entre as informações a serem utilizadas no "where" e no "set". Para garanti-la, optou-se por transmitir ao método dois objetos: um contendo as chaves (valores a serem utilizados no "where") e o outro contendo as informações a serem alteradas (valores a serem utilizados no "set"). A seguir, apresenta-se um procedimento de teste para o update.

```
Livro chaves = new Livro();
Livro dados = new Livro();

chaves.setTitulo("Cem anos de solidão");
chaves.setAno("1996");

dados.setAutor("Maurício Asenjo");
dados.setEditora("UNISANTA");
dados.setLocalizacao("SAntos/SP");
ALDAL.update(dados,chaves);

Resultado em tela:

Update TabLivro set autor = 'Maurício Asenjo', editora
= 'UNISANTA', localizacao = 'SAntos/SP' where titulo =
'Cem anos de solidão' and ano = '1996'
```

Observa-se a seguir o código do referido método, que praticamente duplica a ideia de verificação das propriedades em “null”, compondo o “where” e o “set” conforme destacado.

```
public static void update(Object dados, Object chaves)
{
    Field[] f  = dados.getClass().getDeclaredFields();
    String sql = "Update Tab" + dados.getClass().getSim-
pleName() + " set ";
```

```
    Method mtd;
    String aux1, aux2;
    boolean flag = false;

    for (int i =0; i < f.length; ++i)
    {
       try
       {
       aux1 = "get" + f[i].getName().substring(0,1).toU-
pperCase() + f[i].getName().substring(1);
       mtd = dados.getClass().getMethod(aux1);
       aux2 = mtd.invoke(dados).toString();
       if (!aux2.equals(""))
       {
          if (flag) sql += ", "; else flag = true;
          sql += f[i].getName() + " = ";
          if                  (f[i].getType().getSimple-
Name().equals("String"))
             sql += "'" + aux2 + "'";
          else
             sql += aux2;
       }
       }
       catch(Exception e) {}
    }

    sql += " where ";
    f  = chaves.getClass().getDeclaredFields();
    flag = false;

    for (int i =0; i < f.length; ++i)
    {
       try
       {
       aux1 = "get" + f[i].getName().substring(0,1).toU-
pperCase() + f[i].getName().substring(1);
       mtd = chaves.getClass().getMethod(aux1);
       aux2 = mtd.invoke(chaves).toString();
       if (!aux2.equals(""))
       {
          if (flag) sql += " and "; else flag = true;
          sql += f[i].getName() + " = ";
          if                  (f[i].getType().getSimple-
Name().equals("String"))
             sql += "'" + aux2 + "'";
          else
             sql += aux2;
       }
       }
       catch(Exception e) {}
    }

    System.out.println(sql);
    AFDAL.conecta("Livros.mdb");
    AFDAL.executeSQL(sql);
    AFDAL.desconecta();
}
```

Para concluir, apresenta-se o método "get", responsável pela geração das sentenças SQL do tipo "select", deliberadamente reservado para o final devido à sua singularidade em relação aos demais, destacando-se por ser o único que possui retorno de dados. Assim, enquanto as demais sentenças anteriormente automatizadas são do tipo "void", o método "get", por sua vez, proporcionará o retorno de um objeto.

```
public static void get(Object obj)
{
    Field[] f  = obj.getClass().getDeclaredFields();
    String sql = "Select * from Tab" + obj.getClass().ge-
tSimpleName() + " where ";
    Method mtd;
    String aux1, aux2;
    boolean flag = false;

    for (int i =0; i < f.length; ++i) {
       try
       {
       aux1 = "get" + f[i].getName().substring(0,1).toU-
pperCase() + f[i].getName().substring(1);
       mtd = obj.getClass().getMethod(aux1);
       aux2 = mtd.invoke(obj).toString();
       if (!aux2.equals("")) {
       {
          if (flag) sql += " and "; else flag = true;
          sql += f[i].getName() + " = ";
          if                  (f[i].getType().getSimple-
Name().equals("String"))
             sql += "'" + aux2 + "'";
          else
             sql += aux2;
       }
       }
       catch(Exception e) {}
    }

    System.out.println(sql);
    AFDAL.conecta("Livros.mdb");
    AFDAL.executeSelect(sql,obj);
    AFDAL.desconecta();
}
```

Uma análise inicial do método descrito pode aparentemente contradizer a afirmação anterior, uma vez que o método está declarado como void. No entanto, é importante considerar que, em Java, objetos são sempre passados por referência. Portanto, será empregado esse recurso, no qual o objeto transmitido ao método terá apenas as propriedades utilizadas para compor o "where" definidas, com as demais sendo completadas durante a consulta ao banco de dados. A seguir, é fornecido o script para teste.

```
umlivro.setTitulo("Cem anos de solidão");
umlivro.setAutor(null);
umlivro.setEditora(null);
umlivro.setAno(null);
umlivro.setLocalizacao(null);
ALDAL.get(umlivro);

Resultado em tela:

Select * from TabLivro where titulo = 'Cem anos de so-
lidão'
```

É possível que o funcionamento desse último método pareça confuso e acredita-se que a compreensão será facilitada ao se analisar a segunda camada do projeto, que será apresentada em artigo futuro. Vale ressaltar que o código até aqui discutido se refere exclusivamente à camada lógica, responsável pela geração de sentenças SQL. Ao término de cada método, enfim, ocorrerá uma chamada à camada física, responsável pelo acesso efetivo ao banco de dados.

```
import java.sql.*;
import java.io.*;
import java.lang.reflect.*;

public class AFDAL {
private static Connection con;

    public static void conecta(String _bd)
    {
```



 

```
      Erro.setErro(false);
      try
      {
         Class.forName("net.ucanaccess.jdbc.Ucanac-
cessDriver");
         con = DriverManager.getConnection("jdbc:uca-
naccess://" + _bd);
      }
      catch (Exception e)
      {
         Erro.setErro(e.getMessage());
      }
   }

   public static void desconecta()
   {
      Erro.setErro(false);
      try
      {
         con.close();
      }
      catch (Exception e)
      {
         Erro.setErro(e.getMessage());
      }
   }

   public static void executeSQL(String _sql)
   {
      Erro.setErro(false);
      try
      {
         Statement st = con.createStatement();
         st.executeUpdate(_sql);
         st.close();
      }
      catch(Exception e)
      {
         Erro.setErro(e.getMessage());
      }
   }

   public static void executeSelect(String _sql, Object
obj)
   {
      ResultSet rs;
      Erro.setErro(false);
      try
      {
         PreparedStatement st = con.prepareState-
ment(_sql);
         rs = st.executeQuery();
         if (rs.next())
         {
            Field[] f  = obj.getClass().getDeclaredFi-
elds();
            Method mtd;
            String aux;
            for(int i=0; i<f.length; ++i)
            {
               aux="set"+f[i].getName().substring( 0,
1 ).toUpperCase() + f[i].getName().substring( 1 );
               try
               {
                  mtd = obj.getClass().getMethod(aux,
new Class[] {f[i].getType()});
                  mtd.invoke(obj,    new    Object[]
{rs.getString(f[i].getName())});
               }
               catch(Exception e){}
            }
         }
         else
         {
            Erro.setErro(obj.getClass().getSimple-
Name() + " não localizado."); return;
         }
         st.close();
      }
      catch (Exception e)
      {
         Erro.setErro(e.getMessage());
      }
   }
}
```

Acredita-se que os métodos responsáveis pela conexão (denominado conecta) e desconexão (desconecta) dispensem maiores comentários, assim como o método executeSQL, utilizado para a execução de sentenças do tipo "create table", "insert", "delete" e "update", as quais são passadas como parâmetro ao método. Já o método executeSelect, destinado especificamente à execução de sentenças do tipo "select", requer dois argumentos: a sentença SQL “select” a ser executada, e o objeto no qual os dados da consulta serão retornados. A verificação em destaque é realizada para determinar se o objeto consultado foi localizado ou não. Se sim, os métodos "set" são compostos e invocados para atribuir os dados provenientes da consulta ao objeto, os quais são originalmente armazenados no ResultSet.

## 4. Conclusão

Este artigo descreveu o desenvolvimento de uma interface BDOO para um banco de dados relacional, adicionando uma camada extra em aplicações Java com o objetivo de facilitar a interação entre sistemas orientados a objetos e bancos de dados relacionais, especialmente no contexto do ensino de programação. Funcionando como uma ponte entre a lógica orientada a objetos e a estrutura relacional do banco de dados, esse tipo de ferramenta, conhecida como mapeamento objeto-relacional (ORM), visa automatizar o processo de mapeamento entre entidades no código-fonte (objetos) e tabelas no banco de dados relacional, eliminando a necessidade de se escrever consultas SQL manualmente para operações de inserção, atualização ou recuperação de dados, viabilizando a persistência de dados provenientes de sistemas orientados a objetos no formato relacional.

Proposta com o intuito de proporcionar aos alunos um projeto real e útil para a exploração de conceitos abstratos como metadados e reflexão, a aplicação dessa estratégia resultou em maior entusiasmo e uma melhor assimilação dos conteúdos apresentados por parte dos estudantes. Cabe destacar que muito embora a avaliação do sucesso da estratégia não deva se basear apenas na percepção docente, os resultados inicialmente obtidos parecem sugerir sua eficácia.

Como todo projeto de software, entende-se que este também esteja sujeito a melhorias contínuas. Para o futuro, por exemplo, considera-se o aprimoramento da cláusula "where" para lidar com operadores além da igualdade, a implementação de um esquema de consulta mais abrangente, permitindo assim a geração de consultas que

retornem múltiplos objetos, a capacidade de gerar automaticamente uma tabela a partir de uma classe (e vice-versa) utilizando os conceitos de metadados e reflexão, e a criação de métodos para serialização e desserialização de objetos em padrões como XML ou JSON. Destaca-se, por fim, que tais propostas foram levantadas e discutidas, durante momentos de reflexão e pesquisa sobre ferramentas similares disponíveis no mercado, com os próprios estudantes da disciplina.

**Referências**

BHARGAVI S. & Murthy B. R. **An overview of object-relational database management Systems.** International Journal of Engineering Research and Technology, 2013.

CAMPOS, L. F. de B. **Metadados digitais: revisão bibliográfica da evolução e tendências por meio de categorias funcionais.** Portal de Periódicos UFSC, 2007.

DEVMEDIA. 2022. **Conhecendo Java Reflection.** Disponível na Internet via WWW. URL: https://periodicos.utfpr.edu.br/rbect/article/view/8583. Acesso em 03 de março de 2022.

GUIMARÃES, C. C. **Fundamentos de bancos de dados.** UNICAMP, 2014.

MICHAELIDES, M. G. & Nicolaou, N. A. **A Comparative Study of Object-Oriented, Relational, and Object-Relational Databases.** Information Systems, 2001.

NASSU, E. A. & Setzer V. W. **Bancos de dados orientados a Objetos.** Blucher, 2009.

TETILA, E. C. **Banco de dados relacional: arquitetura, modelo entidade-relacionamento, linguagem SQL e normalização de dados.** Appris Editora, 2021.

TUDOSE, C. et al. **Java Reflection Performance Analysis Using Different Java Development**. Advances in Intelligent Control Systems and Computer Science. 18th International Conference on Control Systems and Computer Science (CSCS 18). Disponível na Internet via WWW. URL: https://link.springer.com/chapter/10.1007/978-3-642-32548-9_31. Acesso em 03 de março de 2022.



 