PERSISTÊNCIA DE DADOS EM JAVA UTILIZANDO HIBERNATE: Mapeamento Objeto / Relacional

José Osvano da SILVA[*]
Luís Augusto Mattos MENDES[**]
Marcelo Caramuru Pimentel FRAGA[***]
Kátia GALATTI[****]

RESUMO

As variadas tecnologias disponíveis atualmente no mercado aliada a evolução das linguagens de programação possibilitam um cenário de interação entre bancos de dados relacionais e linguagens de programação orientadas a objetos. Juntamente a esse cenário soma-se o tempo cada vez menor para o desenvolvimento de sistemas tornando secundária a preocupação com a consistência de dados. O presente artigo visa mostrar a evolução da linguagem de programação Java a partir do auxílio do framework Hibernate, abordando as suas principais vantagens, desvantagens, o mapeamento das tabelas para objetos e como obter esses objetos em consultas.

PALAVRAS-CHAVE: Persistência. Frameworks. Hibernate. Banco de Dados.

ABSTRACT

The several technologies available today and the evolution of programming languages make it possible a scenario of interaction between relational databases and oriented programming languages. To this scenario it is added the ever-decreasing time for the development of systems making secondary the concern about data consistency. This article aims to show the evolution of the Java programming language with the help of the Hibernate framework, addressing the main advantages, disadvantages, the mapping of tables to objects and how to get these objects in queries.

KEYWORDS: *Persistence. Frameworks. Hibernate. Database.*

INTRODUÇÃO

A evolução das linguagens de programação e a necessidade cada vez maior de aplicações robustas e de qualidade que possam ser produzidas em um tempo cada vez menor fazem com que muitas tecno-

[*] Bacharel em Ciência da Computação pela Universidade Presidente Antônio Carlos – UNIPAC/Barbacena. Pós-Graduando em Melhoria de Processo de Software (MPS) pela Universidade Federal de Lavras – UFLA, e-mail: osvano@gmail.com

[**] Professor Assistente do CEFET-MG – Campus Divinópolis – Curso Técnico de Informática, e-mail: luisaugusto@div.cefetmg.br

[***] Professor Assistente do CEFET-MG – Campus Divinópolis – Curso Técnico de Informática, e-mail: caramuru@div.cefetmg.br

[****] Professor Assistente da FATEC -TQ – Campus Taquaritinga – Curso Superior em Tecnologia de Agronegócio, e-mail: kagalatti@hotmail.com

logias novas acabem surgindo como promessa de resolver todos os problemas. Entretanto, nem sempre é tão simples assim, afinal, o problema que todos os desenvolvedores sempre enfrentam é o tempo gasto para tratar a ligação de uma aplicação orientada a objetos com um banco de dados relacional. O tempo demandado para tal ligação começou a incomodar a comunidade Java visto que a mesma era prejudicada com essa demora. Em meio a várias idéias, uma se destacou bastante que foi a criação de uma solução chamada de mapeamento objeto/relacional (ORM), solução essa que foi implementada através de *frameworks*. A finalidade de um *framework* é capturar as funcionalidades comuns entre várias aplicações. Dentre esses *frameworks*, temos o Hibernate, que é uma implementação de código aberto.

O artigo está dividido da seguinte forma: a seção 2 trata da persistência de dados; a seção 3 apresenta a arquitetura do Hibernate; a seção 4 discute o mapeamento objeto/relacional; a seção 5 trata de consultas e a seção 6 apresenta informações sobre relacionamentos. Por fim, a seção 7 discorre sobre as considerações finais do presente artigo.

Persistência de Dados

Uma das grandes decisões em todos os projetos de software que necessitam trabalhar com algum SGBD (Sistema Gerenciador de Banco de Dados) é saber como tratar a persistência de dados, e essa situação acaba se agravando quando a linguagem de programação escolhida para o aplicativo é o Java. Durante vários anos, criaram-se grandes debates dentro da comunidade Java de como tratar de maneira eficiente essa persistência. Essa discussão saudável acabou iniciando alguns projetos promissores e que hoje estão sendo utilizados para ajudar os desenvolvedores nessa tarefa que até então consumia muito tempo no desenvolvimento de uma aplicação.

Para um melhor entendimento do que vem a ser a necessidade dessa persistência, primeiramente, deve-se observar a evolução das linguagens de programação em comparação com a evolução dos Bancos de Dados. A maioria delas hoje trabalha com orientação a objetos e essas mesmas linguagens orientadas necessitam manipular dados tabulares de SGBDs relacionais, devido ao fato da maioria dos Bancos de Dados ainda ser relacional. Isso muitas vezes é um problema, que faz com que os desenvolvedores tenham muito trabalho para manipular os dados de maneira eficiente. Juntando esse problema à necessidade de portabilidade entre Bancos de Dados para uma mesma aplicação, fez com que a comunidade encontrasse uma solução chamada mapeamento objeto/relacional (ORM). Dentre algumas das implementações dessa solução, podemos citar o Hibernate que é uma implementação de fonte aberta do ORM e objeto de nosso artigo.

O mapeamento de objetos relacionais ou ORM nada mais é que a persistência de objetos de maneira automatizada, mapeamento esse realizado dentro de um aplicativo Java. O ORM tem como objetivo fazer a transformação dos dados entre uma representação e outra (BAUER, 2005). A Figura 1 mostra o funcionamento da persistência de dados.

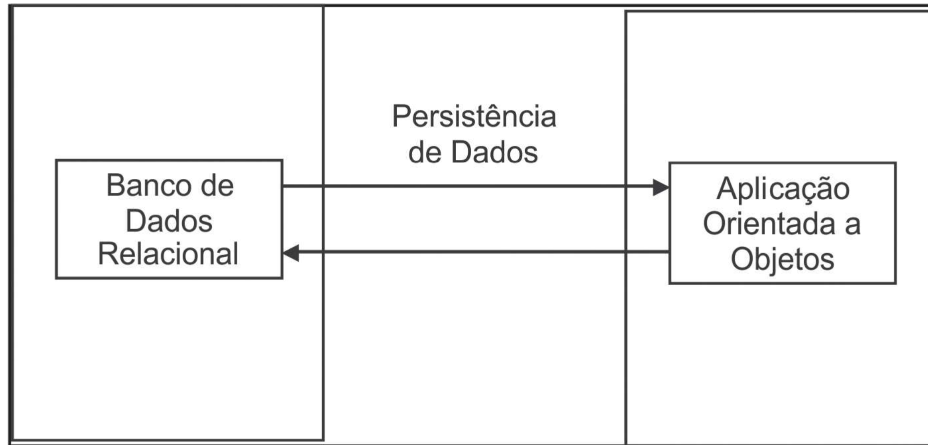

Figura 1 – Funcionamento da persistência de dados. (BAUER, 2005)

O Hibernate é um *framework* que implementou por completo a solução ORM e permite fazer o mapeamento de classes Java em tabelas de banco de dados e vice-versa. Apresenta como vantagens, recursos poderosos e provê suporte ao mapeamento de associações entre objetos, herança, polimorfismo, composição e coleções. Segundo Bauer (2005), o Hibernate possui alguns benefícios que merecem ser mencionados. São eles:

- **Produtividade:** O que todos os desenvolvedores buscam é poder se concentrar mais no problema do negócio em si e não perder muito tempo com a cansativa digitação de código para tratar a persistência. Com o Hibernate, boa parte desse trabalho é eliminado, aumentando assim a produtividade;
- **Manutenção:** A facilidade de compreensão de um sistema aumenta devido ao fato de possibilitar que o desenvolvedor digite menos linhas de código e por ser mais voltado para a solução do problema em questão, abstraindo a persistência;
- **Desempenho:** Existem várias técnicas para melhorar o desempenho de persistência que podem ser específicas de um SGBD e essas técnicas estão implementadas no Hibernate. As mesmas não são fáceis de serem descobertas e implementadas para uma solução de persistência não automatizada;
- **Independência de Fornecedor:** O Hibernate permite que o aplicativo fique portável entre a maioria dos Bancos de Dados SQL, tornado assim a aplicação, além de portável entre plataformas, por utilizar o Java, portável entre SGBDs.

Em contrapartida, muitas aplicações utilizam de muitos recursos proporcionados pelo próprio Banco de Dados para gerenciar suas informações, como: *Procedures*, *Triggers*, etc. Imaginando um Banco de Dados já pronto com essas funcionalidades já implementadas, a solução ORM não seria uma boa prática, devido ao fato da mesma implementar os mesmos tratamentos dentro da aplicação. Isso é considerado por muitos desenvolvedores como uma das grandes desvantagens do Hibernate.

Em se tratando da solução Hibernate, outra desvantagem que a comunidade sempre questiona é o fato da quantidade de arquivos de mapeamento que são criados: geralmente um arquivo para cada classe.

Arquitetura do Hibernate

A arquitetura do Hibernate nada mais é que um conjunto de interfaces as quais ficam na camada de negócios e de persistência da aplicação, assim, interagindo diretamente com as APIs JNDI *(Java Naming and Directory Interface),* JDBC *(Java Database Connectivity)* e JTA *(Java Transaction API)*.

De acordo com a Figura 2, classificada por Galhardo (2006), podemos distribuir as interfaces da seguinte forma:

- Interfaces responsáveis por executar operações de CRUD[1]: Sessão, Transação e Consulta *(Session, Transaction e Query)*;
- Interface que permite que a aplicação faça a configuração do Hibernate: Configuração *(Configuration)*;
- Interfaces responsáveis por fazer com que os eventos do Hibernate e a aplicação possam interagir: *Interceptor, Lifecycle e Validatable*;
- Interfaces que permitem que o Hibernate possa ter suas funcionalidades de mapeamento estendidas: *UserType, CompositeUserType e IdentifierGenerator*.

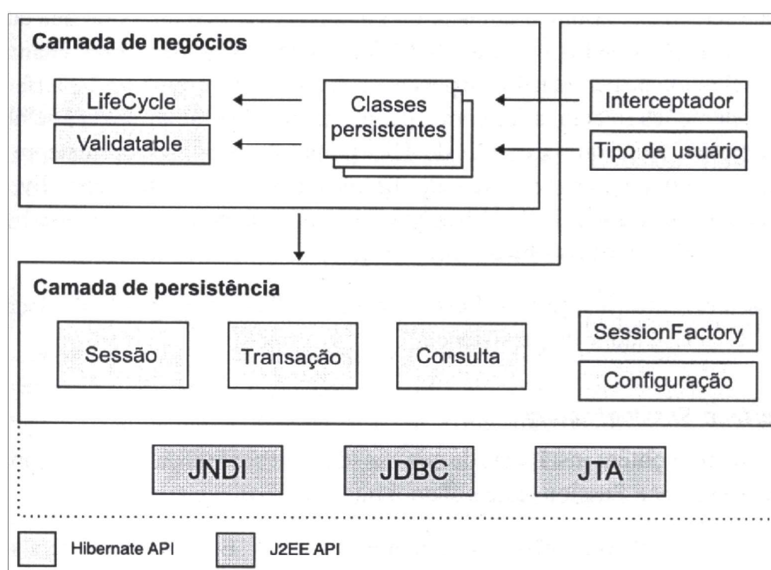

Figura 2 - Visão geral de alto nível da API do Hibernate (Bauer, 2005).

São cinco interfaces que são usadas em quase todos os aplicativos do Hibernate, classificadas por Bauer (2005) como:

- Interface *Session*: é a interface principal, uma instância de *Session* é considerada de peso leve e pode ser criada e destruída sempre que for necessário. Localiza-se entre a conexão e a transação;
- Interface *SessionFactory*: A instanciação de *SessionFactory* já não é tão leve como *Session*; na verdade o aplicativo provavelmente terá apenas um *SessionFactory* ou um para cada SGBD que o mesmo necessite acessar. Ela tem a finalidade de armazenar em cache instruções SQL e outros metadados de mapeamentos para serem usados em tempo de execução;

1 CRUD - acrônimo em inglês para criar, ler, atualizar e apagar

- Interface *Configuration*: A instanciação dessa interface é necessária para que o Hibernate comece a funcionar;
- Interface *Transaction*: Essa interface tem como objetivo gerenciar as transações dentro de um *Session* do Hibernate;
- Interfaces *Query* e *Criteria*: A Interface *Query* é usada para fazer consultas ao banco de dados. A Interface *Criteria,* por sua vez, permite que as consultas tenham critérios orientados a objetos. Ambas são consideradas de peso leve e podem ser usadas sempre que necessárias.

As interfaces de *Callback* têm como finalidade manter o Hibernate informado de tudo de interessante que acontecer com os objetos, como quando o mesmo é criado, excluído ou salvo. São três essas interfaces:

- Interfaces *Lifecycle* e *Validatable*: permitem que um objeto persistente reaja a eventos relativos ao seu próprio ciclo de vida;
- Interfaces *Interceptor*: Essa interface permite ao aplicativo processar *Callbacks* sem forçar as classes persistentes a implementar APIs específicas do Hibernate. (BAUER, 2005).

As interfaces de tipos é um recurso poderoso do Hibernate, através desse recurso que ele consegue fazer o mapeamento entre os tipos do Java e os tipos das colunas do Banco de Dados, existe ainda a possibilidade do usuário incluir seus próprios tipos através das interfaces *UserType* e *CompositeUserType.* (BAUER, 2005).

O Hibernate tem a maioria de suas funcionalidades configuráveis, mas quando isso não for suficiente é possível instanciar várias de suas interfaces para que as mesmas possam ser estendidas e com isso atendam à necessidade específica da aplicação.

Mapeamento Objeto/Relacional

Antes de criar qualquer aplicação no Hibernate, é necessário observar alguns detalhes importantes com relação às chaves primárias das tabelas. Em Java podemos definir a identidade de um objeto sobrescrevendo o método "Object.equals(Object object)", não podemos adotar o mesmo sistema para o Banco de Dados. O SGBD só entende de tabelas, chaves estrangeiras e primárias. A solução para poder contornar essa diferença é termos em nossos objetos um identificador não natural, geralmente chamado de ID, porque assim o Banco de Dados e o Hibernate poderão diferenciar os registros e os objetos, respectivamente. E, além disso, cuidar de seus relacionamentos, que no Hibernate são feitos através de um arquivo de mapeamento (LINHARES, 2006).

O Hibernate possui vários mecanismos internos para a geração de chaves primárias de maneira automática, conforme podemos observar na Tabela 1.

Tabela 1 - Mecanismos para a geração de chave primária (Galhardo, 2006)

Mecanismo	Descrição
Identity	Mapeado para colunas identity no DB2, MySQL, MSSQL, Sybase, HSQLDM, Infomix.
Sequence	Mapeado em seqüências no DB2, PostgreSQL, Oracle, SAP DB, Firebird (ou generator no Interbase).
Increment	Lê o valor máximo da chave primária e incrementa um. Deve ser usado quando a aplicação é a única a acessar o banco e de forma não concorrente.
Hilo	Usa um algoritmo chamado *high/low* para geração de chaves únicas.
uudi.hex	Usa uma combinação do IP com um timestamp para gerar um identificador único na rede. Esse mecanismo não é muito comum devido ao fato de gerar uma string de 32 posições como chave e com isso ocupar mais espaço que um identificador inteiro.

Mapear uma classe para uma tabela exige algumas convenções adotadas na classe. Uma delas é que a mesma siga o padrão de JavaBeans (a classe deverá possuir um construtor padrão sem argumentos, e métodos *getters* e *setters* para cada atributo). Esse padrão por ser muito antigo começou a ser conhecido como POJOs (Objetos Java Puros Antigos - *Plain Old Java Objects)*.

Para poder facilitar o entendimento dos exemplos a seguir vamos trabalhar com um modelo de exemplos de objetos de uma faculdade composto pelas classes: Pessoa, Aluno, Professor, Endereço, Turma, Disciplina e Curso que se relacionam como mostrado na Figura 3.

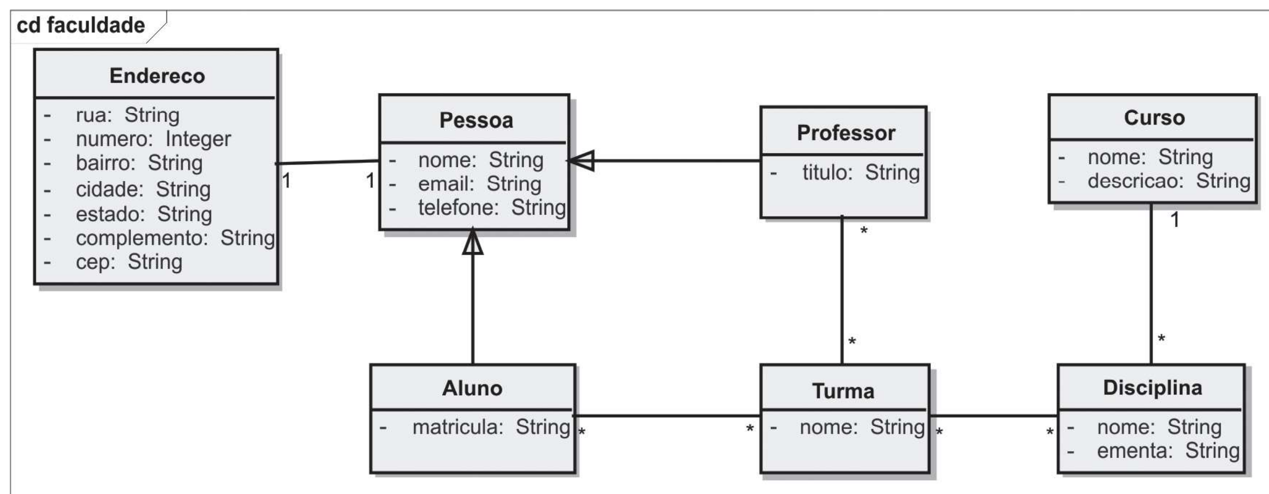

Figura 3 - Modelo de objetos de uma faculdade (Linhares, 2006)

Uma **Pessoa** possui um **Endereço** e, tanto um **Aluno** quanto um **Professor**, é um tipo de **Pessoa**. Um **Aluno** pode estar em várias **Turmas** e, da mesma forma, um **Professor** pode dar aulas em várias **Turmas**. Um **Curso** pode ter várias **Disciplinas**. Por sua vez, uma **Disciplina** está ligada a apenas um **Curso**. Uma **Turma** pode ter vários **Professores** e um **Professor** pode dar aulas em várias **Turmas.** Uma **Disciplina** pode pertencer a várias **Turmas** e uma **Turma** pode ter várias **Disciplinas**.

Em nossas classes devemos adotar as convenções de POJOs, conforme podemos observar na Figura 4 que mostra a implementação de um POJO para a classe **Pessoa**. E esses POJOs são mapeados em Hibernate através de um arquivo .XML *(Extensible Markup Language).* (BAUER, 2005).

```
package com.faculdade;

import java.io.Serializable;

public class Pessoa implements Serializable {
	private String nome;
	private String email;
	private String telefone;
	private Endereco endereco;
	private Integer id; //Identificador

	public Pessoa() {

	} //Construtor default sem argumentos
	... //Métodos Getters e Setters para cada atributo
```

Figura 4 - Exemplo de código fonte de parte de uma classe POJO

Outra convenção importante é utilizar os nomes e os tipos dos atributos da classe iguais aos nomes que foram definidos para os mesmos nas tabelas. Sendo assim, não há a necessidade de referenciar os nomes dos campos no arquivo de mapeamento, pois o Hibernate já subentende que os nomes são os mesmos.

O XML é uma especificação recomendada pelo órgão W3C *(World Wide Web Consortium)* que tem por finalidade implementar linguagens de marcação, para descrever dados de qualquer natureza. Isso possibilita às aplicações trocarem informações utilizando qualquer protocolo de comunicação. (DUMOULIN, 2004).

O Hibernate utiliza o XML para poder fazer o mapeamento entre classes e tabelas e, por default, utiliza a nomenclatura “Nome da Classe”.hbm.xml. Seguindo nosso exemplo acima, o mapeamento da classe **Pessoa** ficaria como mostrado na Figura 5.

```
<!-- Definições de DTD -->
<?xml version=”1.0” encoding=”UTF-8”?>
<!DOCTYPE hibernate-mapping
  PUBLIC “-//Hibernate/Hibernate Mapping DTD 3.0//EN”
  “http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd”>

<!-- Declaração de Mapeamento -->
<hibernate-mapping>

  <!—Classe pessoa mapeada para a tabela Pessoa -->
  <class name=”com.faculdade.Pessoa”>

    <!-- Identificador da classe -->
    <id name=”id”>
      <generator class=”increment”/>
    </id>

    <!-- Propriedades da classe -->
    <property name=”nome”/>
    <property name=”telefone”/>
    <property name=”email”/>

    <!-- Relacionamento da classe -->
    <one-to-one
      name=”endereco”
         class=”Endereco”
cascade=”save-update”/>

  </class>

</hibernate-mapping>
```

Figura 5 - Mapeamento XML do Hibernate para a Classe Pessoa

Um arquivo XML de mapeamento do Hibernate começa com as definições da DTD *(Document Type Definition)* e do nó raiz <hibernate-mapping>. Depois vem o nó onde definimos a classe que está sendo mapeada <class>. Dentro do nó <class>, fazemos as definições de <id>, <property> (onde serão definidos os nomes de cada atributo), lembrando que os atributos só necessitam ser definidos quando houver diferenças entre os nomes dos mesmos e os campos da tabela. Logo depois, dentro ainda do nó <class>, são feitas as definições dos relacionamentos, caso os mesmos existam. Os relacionamentos são tratados na seção 6.

Consultas

Umas das maiores necessidades de uma aplicação que armazene dados em um Banco de Dados, é

permitir consultas a esses dados, consultas essas que deverão ser realizadas de maneira eficiente e de fácil implementação. O Hibernate deu uma atenção toda especial a esse assunto e criou uma linguagem chamada HQL *(Hibernate Query Language)*, que é um dialeto orientado a objetos da linguagem de consulta SQL. (BAUER, 2005).

Observe o exemplo da Figura 6 que mostra como obter todas as Pessoas com o nome José.

```
Query q = session.createQuery("from Pessoa p where p.nome = :fnome");
q.setString("fnome", "José");
List result = q.list();
```

Figura 6 - Consulta utilizando o HQL

É possível também fazer a consulta através de Critérios, que permitam que se construam consultas, passando para elas objetos em tempo de execução. Essa técnica permite a atribuição de restrições dinamicamente (BAUER, 2005), conforme a Figura 7 mostra na nossa mesma consulta.

```
Criteria criteria = session.createCriteria(Pessoa.class);
criteria.add(Expression.like("nome", "José");
List result = criteria.list();
```

Figura 7 - Consulta utilizando Criteria

Outra forma de consulta implementada no Hibernate é a consulta através de exemplos, chamado de QBE *"Query by example"*. Esse recurso permite que a consulta receba uma instância da própria classe. Nessa instância é atribuído valores para alguns ou todos os atributos do objeto e, dependendo da necessidade de filtro dos dados, esse objeto é passado para a pesquisa que se utiliza de todos os atributos não nulos para filtrar os dados no Banco de Dados (BAUER, 2005). Observe a Figura 8 e veja como ficaria nossa mesma pesquisa utilizando esse recurso.

```
Pessoa exemploPessoa = new Pessoa();
exemploPessoa.setNome("José");
Criteria criteria = session.createCriteria(Pessoa.class);
criteria.add(Example.create(exemploPessoa);
List result = criteria.list();
```

Figura 8 - Consulta utilizando QBE

Relacionamentos

Não podemos imaginar uma estrutura de classes gerenciando tabelas de um Banco de Dados que se relacionam sem pensarmos em como essas classes consistirão esses relacionamentos. Um dos recursos que deve ser atendido ao se utilizar uma ferramenta de ORM é que a mesma permita que os objetos sejam consistidos também em seus relacionamentos. O Hibernate permite, de maneira relativamente simples, gerenciar esses relacionamentos. A seguir, serão mostrados pequenos exemplos de como seriam feitos os mapeamentos dos principais relacionamentos.

A herança possibilita herdar as características de uma classe e permite atribuir novas características a essa mesma classe. Isso é um dos principais recursos da orientação a objetos, só que quando juntamos isso ao gerenciamento de informações tabulares enxergamos a incompatibilidade de paradigmas mais facilmente. O Hibernate permite fazer o mapeamento de herança de três formas diferentes:

- Tabela por classe concreta: Cada classe é mapeada para uma tabela no banco;
- Tabela por Hierarquia: Todas as classes de uma mesma hierarquia são mapeadas em uma única tabela;
- Tabela por Sub-Classe: Mapeia-se cada classe, inclusive a classe pai, para tabelas diferentes. (BAUER, 2005).

A Figura 9 apresenta um exemplo de herança entre as classes professor e pessoa.

```
<hibernate-mapping>

   <!--  Mapeamento de Herança usando uma tabela para cada classe -->
   <joined-subclass name="Professor" extends="Pessoa">
      <key column="Pessoa_id"/>
      <property name="titulo"/>

      <!-- Especificação do tipo de relacionamento -->
      <set name="turmas"
         inverse="true">
         <key column="Pessoa_Professor_id"/>
         <one-to-many class="Turma"/>
      </set>

   </joined-subclass>
</hibernate-mapping>
```

Figura 9 - Herança entre as classes Professor e Pessoa. (Linhares, 2006)

Por sua vez, a associação é um termo definido para relacionamentos entre as entidades. Os tipos mais comuns desses relacionamentos são: n - para - n, 1 - para - n e n - para - 1. Existem também relacionamentos 1- para – 1, conforme podemos observar no exemplo de mapeamento de nossa classe **Pessoa**. No nosso exemplo anterior, quando definimos:

```
...
      <set name="turmas"
         inverse="true">
         <key column="Pessoa_Professor_id"/>
         <one-to-many class="Turma"/>
      </set>
...
```

Figura 10 - Associação entre Professor e Turma (Linhares, 2006)

onde temos <one-to-many> , como apresentado na Figura 10, estamos definindo que existe uma associação de um para muitos entre Professor e Turma.

Podemos ter ainda associações definidas como <many-to-one>, <one-to-one> ou <many-to-many>.

CONSIDERAÇÕES FINAIS

As linguagens de programação tendem a evoluir de forma muito mais rápida que os Bancos de Dados. Essa evolução provoca incompatibilidades entre os paradigmas, no caso mencionado nesse artigo, entre os paradigmas relacionais e orientado a objetos. O Hibernate foi construído para contornar e gerenciar essa incompatibilidade. Conforme observado, passamos a nos concentrar em nossa aplicação com o gerenciamento de objetos e não mais com o de registros.

A persistência de dados proporcionada pelo Hibernate nos permite concluir que os desenvolvedores Java podem contar com mais uma funcionalidade que fará com que aplicativos possam ser criados de forma mais rápida, mantendo sua qualidade, reduzindo assim o custo de aplicações, e possibilitando que essa linguagem possa adquirir cada vez mais espaço no mercado.

A solução ORM não possui apenas a implementação do Hibernate. Existem várias outras já prontas e em desenvolvimento. Cabe a cada desenvolvedor escolher a que melhor atenda às suas necessidades, analisar os recursos, vantagens e desvantagens de cada *framework*. Assim, independente da implementação escolhida, temos nas mãos uma poderosa ferramenta para cuidar dos trabalhos repetitivos e para nos proporcionar recursos extremamente satisfatórios.

Essa solução abre as portas para facilitar a independência de Banco de Dados entre as aplicações, para permitir que as aplicações possam acessar mais de um Banco de Dados ao mesmo tempo e para proporcionar a mesma integração entre sistemas legados (sistemas desenvolvidos por outras empresas, com funcionalidades específicas, mas com dados necessários para a aplicação em questão) de maneira extremamente eficiente.

REFERÊNCIAS

BAUER, Christian; KING, Gavin. *Hibernate em Ação*. Rio de Janeiro: Ciência Moderna, 2005.
DUMOULIN, Cedric; FRANCISCUS, George; WINTERFELD, David. *Struts em Ação*. Rio de Janeiro: Ciência Moderna, 2004.
GALHARDO, Raphaela; LIMA, Gleydson. *Introdução ao Hibernate*. Disponível em: < http://www.jspbrasil.com.br/mostrar/4 >. Acesso em 15 de agosto de 2006.
LINHARES, Maurício. *Introdução ao Hibernate 3*. Disponível em: < http://www.guj.com.br/java.tutorial.artigo.174.1.guj >. Acesso em 10 de Setembro de 2006.