UNIVERSIDADE FEDERAL DE PERNAMBUCO

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
CENTRO DE INFORMÁTICA

PERSISTÊNCIA EM BANCO DE DADOS: UM ESTUDO PRÁTICO SOBRE AS API JPA E JDO

TRABALHO DE GRADUAÇÃO

POR

NATÁLIA DE LIMA DO NASCIMENTO

Recife, Novembro de 2009

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

NATÁLIA DE LIMA DO NASCIMENTO

"PERSISTÊNCIA EM BANCO DE DADOS: UM ESTUDO PRÁTICO SOBRE AS API JPA E JDO"

ESTE TRABALHO FOI APRESENTADO Á GRADUAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA A CONCLUSÃO DO CURSO DE CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR (A): PROF. ROBSON DO NASCIMENTO FIDALGO

RECIFE, NOVEMBRO/2009

Agradecimentos

Nesses 4 anos de caminhada dentro da universidade, eu conheci muitas pessoas. Algumas passaram, outras ficaram, mas todas, de alguma forma, auxiliaram no meu desenvolvimento.

Primeiramente eu gostaria de agradecer a Deus, pois sem Ele nada disso teria acontecido.

Gostaria de agradecer a minha família, por sempre ter me apoiado, desde o início da minha vida e em especial meus pais, pelos sacrifícios feitos para que eu pudesse chegar até aqui.

Gostaria também de agradecer aos meus amigos e namorado que me acompanharam em todos esses anos, nas alegrias e tristezas da vida acadêmica, compartilhando sorrisos e desesperos.

Também não posso deixar de lembrar dos meus professores, que tanto me ensinaram todos esses anos, e que foram cruciais para o meu crescimento como pessoa e como profissional, e em especial a Robson, por ter me oferecido tantas oportunidades e por ter aceitado ser meu orientador, nessa tarefa árdua.

Enfim, agradeço a todos que passaram pela minha vida e me ajudaram a ser quem eu sou hoje.

Resumo

Aplicações de software geralmente têm que lidar com armazenamento de dados persistentes, onde ao final da execução de um programa, os dados armazenados em memória RAM ficam persistidos em disco. Para a plataforma Java, há uma grande variedade de padrões e soluções proprietárias para realizar essa função, tornando difícil a escolha da melhor solução. Nesta plataforma de programação, a interface de conectividade, persistência e programação com BD (Banco de Dados) mais difundida é a JDBC (Java Database Conectivity). Além de JDBC, atualmente existem outras propostas, como os frameworks Hibernate e TopLink, que diferente de JDBC, permitem persistir e manipular diretamente objetos Java ao invés de ter que mapear estes objetos em linhas e colunas de uma tabela relacional. Dessa forma, todo o trabalho de mapear, traduzir pesquisas, manipular objetos em SQL e conectividade com o BD passou a ser função do framework de persistência.

Dado que existem várias formas de mapear objetos para BD relacionais, foram desenvolvidas duas API (Application Programming Interface) com o objetivo de padronizar este mapeamento, a saber; JPA (Java Persistence API) e JDO (Java Data Objects). Assim, este trabalho foi desenvolvido com o objetivo de explicar, de forma prática, cada uma dessas API e, ao final, fazer uma análise comparativa entre elas.

Palavras-chave: JPA, JDO, persistência, mapeamento

Sumário

Lista de Figuras

Lista de Quadros

1. Introdução

Este trabalho faz um estudo de duas API (Application Programming Interface) de persistência de dados: JPA (Java Persistence API) [Sun09b] e JDO (Java Data Objects) [Sun09c]. Persistência de dados é o armazenamento não-volátil de dados [WIKI09a][BLJ09]. Ou seja, ao final da execução de uma aplicação de software, os dados armazenados em memória RAM ficam persistidos em disco, até que estes sejam excluídos pelo usuário.

Em uma arquitetura de software divididas em camadas, a camada de persistência ou de acesso aos dados é a parte do software responsável por se comunicar com o BD (banco de dados) [Uni09] ou com o framework de persistência [Sil09] (ver Figura 1).

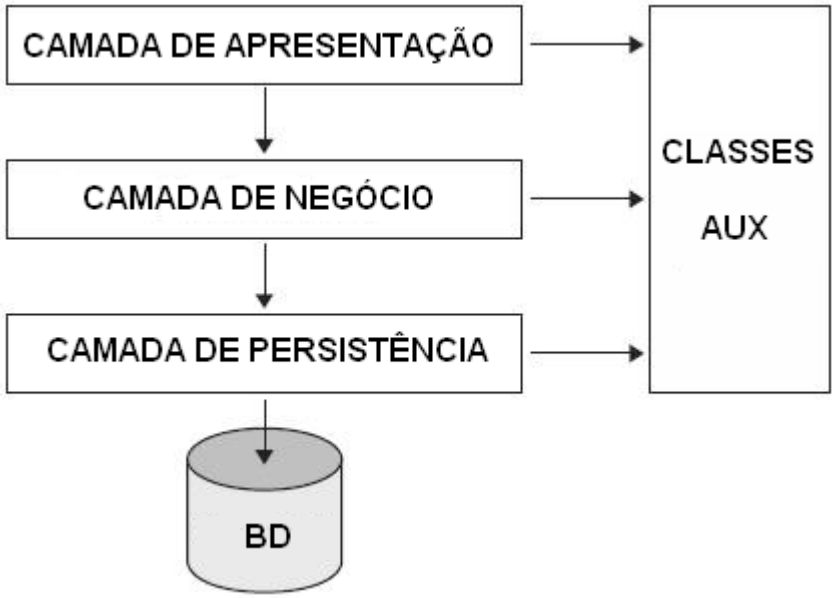

Figura 1 - Camada de Persistência

A camada de persistência permite a conexão com o BD e a manipulação de seus dados. Além disso, é responsável por transformar esquemas de objetos em esquemas relacionais, já que em muitos casos o BD é relacional. Assim, o BD pode ser acessado usando um framework de persistência ou escrevendo comandos em SQL [Sil09].

Software desenvolvidos em Java [Sun09f] e que acessam um BD Relacional (BDR) [Ric02] requerem um mapeamento entre os objetos da aplicação e a estrutura tabular do BDR (o contrário

também é verdade). Este mapeamento chama-se Object-relational mapping (ORM) [Amb97] e é resumidamente apresentado da seguinte forma: as tabelas do BD são representadas através de classes e os registros de cada tabela são representados como instâncias das classes correspondentes. Com essa técnica, a necessidade de comandos SQL se torna mínima, pois o desenvolvedor poderá usar uma interface de programação simples onde o framework pode fazer todo o trabalho de persistência.

A forma como o ORM é configurado depende da ferramenta que será utilizada. Por exemplo, usando Hibernate [Hib09a] com Java, o ORM pode ser feito através de arquivos XML [XML09] ou do uso do padrão annotations [Sun09a] da própria linguagem. Então, visando a padronização do ORM, que era feito de inúmeras formas, dependendo da solução escolhida, foram propostas duas API, JPA e JDO. As duas trazem além de padronização, transparência e abstração no desenvolvimento de aplicações, tornando o desenvolvimento mais leve, limpo e simples. Por isso, são consideradas fortes candidatas a se manterem como preferidas no desenvolvimento desses tipos de aplicações [WIKI09c]. Ressalta-se que annotation é uma forma especial de metadados que pode ser adicionado ao código Java. Com o uso de annotation, consegue-se adicionar informações sobre um programa, as quais não fazem parte do programa em si, não tem efeito direto na operação do código [Sun09a]. JPA e algumas versões mais atualizadas de JDO utilizam o padrão de annotations adotado na versão 5.0 de Java.

1.1. Motivação

O desenvolvimento deste trabalho foi motivado pela existência de várias formas de realizar o ORM, há busca por soluções que padronizassem essa técnica de mapeamento. Dessa forma, foram encontradas duas API que fazem esse trabalho, JPA e JDO, e com isso, foi gerado o interesse de disponibilizar um material de estudo prático e comparativo dessas duas API.

1.2. Objetivo

O objetivo deste trabalho é oferecer um documento que, de forma prática, exemplifique a utilização das API JPA e JDO, comparando-as com base nas suas linguagens de acesso a dados,

JPQL [Sun09g] e JDOQL [Sun09h], respectivamente. Além disso, estas API terão seu poder de expressividade comparado com o poder de expressividade de SQL.

1.3. Exemplo Guia e Sistemática de Estudo

Esta seção apresenta o esquema conceitual de um BD que será usado como exemplo guia, bem como a sistemática que será empregada para o estudo e comparação das API JPA e JDO.

1.3.1. Esquema conceitual

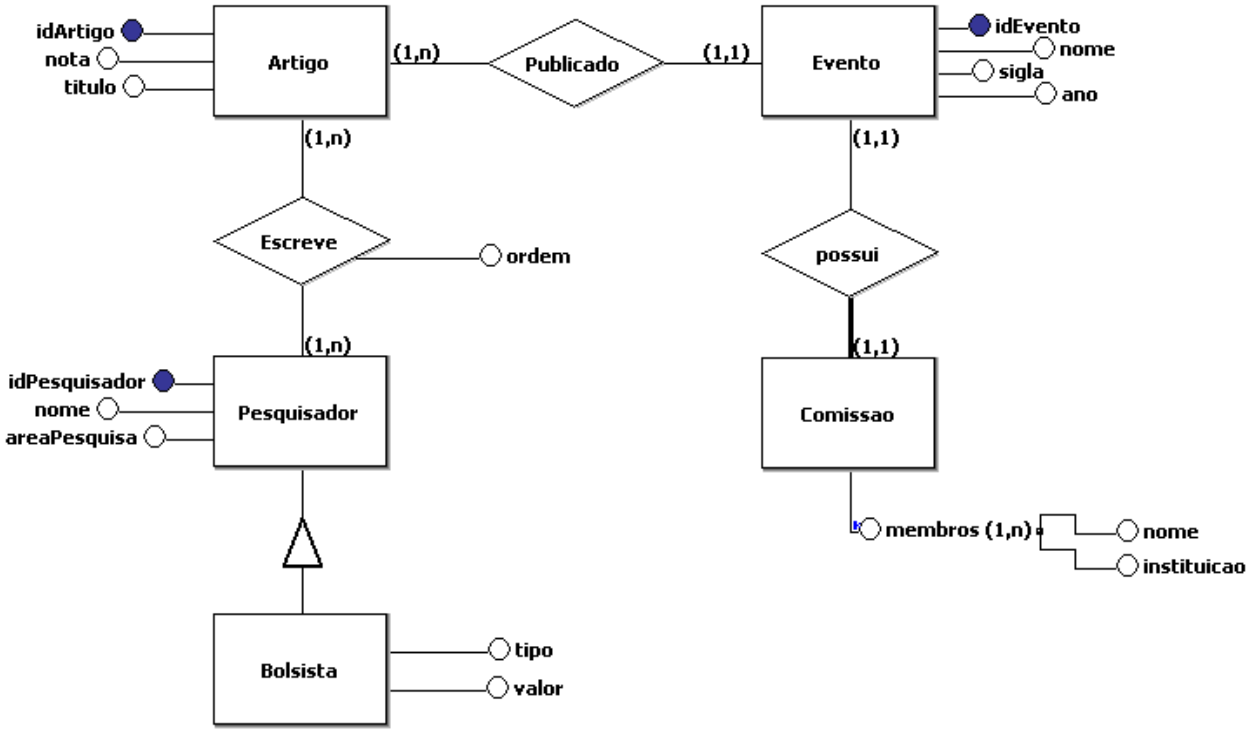

Figura 2 - Esquema conceitual para o desenvolvimento do exemplo prático

A Figura 2 corresponde ao esquema conceitual, feito na ferramenta CASE BRModelo [BRM09], que modela um BD fictício para gestão de eventos científicos. O mini-mundo foi modelado com o objetivo de conter as várias formas de relacionamento, M:N, 1:N e 1:1, e alguns dos conceitos mais encontrados em esquemas ER [Mel09], a saber: herança, atributos multivalorados e compostos, entidade fraca e atributo na relação.

No esquema acima, um Artigo só pode ser publicado em um Evento, porém um Evento pode ter vários Artigos publicados. Um Artigo pode ser escrito por vários Pesquisadores assim como vários Pesquisadores podem escrever vários Artigos, com uma ordem de aparição nos Artigos. Um Pesquisador pode ou não ser Bolsista. Um Evento possui apenas uma Comissão, assim como uma Comissão se relaciona apenas com um Evento, e só existe se o Evento existir. Uma Comissão pode possuir um ou vários membros.

1.3.2. Sistemática de Estudo

A sistemática de estudo deste trabalho foi construída visando facilitar o entendimento das duas API e é definida da seguinte forma:

- *Conceitos Básicos* – nesta seção são apresentados alguns conceitos básicos das API, características e implementações;
- *Exemplo Prático* – nesta seção é apresentado como o exemplo guia da seção 1.3.1 é desenvolvido segundo a especificação de cada API. Esta seção é subdividida da seguinte forma:
  - *Configuração do ambiente*: explica quais ferramentas foram utilizadas para o desenvolvimento da aplicação bem como as bibliotecas necessárias;
  - *Criação do Projeto*: explica como criar o projeto;
  - *Criação de Entidades*: explica como criar as classes do projeto;
  - *Definindo o ORM*: nesta seção começa a explicação de como fazer o ORM, e é dividido em várias seções, onde cada uma explicará o mapeamento de uma parte da aplicação. São elas:
    - *Definir Tabelas e Colunas;*
    - *Definir Herança;*
    - *Definir Atributo Composto e/ou Multivalorado;*
    - *Definir Relacionamentos;*
      - *Um-Para-Um;*
      - *Muitos-Para-Um;*
      - *Muitos-Para-Muitos;*

- *Definir Propriedades da Persistência*: nesta seção será explicado como criar os arquivos de configuração necessários para cada API;
- *Trabalhando a Persistência*: nesta seção serão mostradas as classes responsáveis pelo gerenciamento dos objetos e como elas são criadas no projeto. Em seguida serão mostrados exemplos das operações básicas com objetos:
  - *Inserção;*
  - *Alteração;*
  - *Remoção;*
  - *Busca.*

1.4. Estrutura do Trabalho

Este documento divide-se em 5 capítulos. Neste capítulo é feita uma apresentação do trabalho, com uma introdução aos conceitos necessários para seu entendimento, a motivação, o objetivo, o exemplo guia, a sistemática de estudo e a estrutura deste documento. O capítulo 2 discorre sobre JPA, apresentando uma breve introdução sobre a API e a implementação do exemplo guia segundo a especificação de JPA. O capítulo 3 versa sobre JDO, com a mesma abordagem do capítulo 2. O capítulo 4 apresenta um estudo sobre as diferenças entre JPA e JDO, tomando como base principal as suas linguagens de acesso a dados, JPQL e JDOQL respectivamente. E por fim, o capítulo 5 apresenta a conclusão obtida com o trabalho, contribuições e trabalhos futuros.

2. Java Persistence API

Este capítulo apresenta os principais conceitos de JPA e, a partir da sistemática apresentada na seção 1.3.2, como esta API é utilizada para implementar o exemplo guia da seção 1.3.1.

2.1. Conceitos básicos de JPA

JPA foi desenvolvida pela equipe de software do EJB (Enterprise JavaBeans) 3.0[Sun09e] como parte do JSR 220 (Java Specification Request) [JPA09]. Esta API permite o desenvolvimento rápido e simplificado de aplicações distribuídas, transacionais, seguras e portáveis baseadas na tecnologia Java. Mas seu uso não é limitado a componentes de software da EJB. Ela pode ser utilizada diretamente por aplicações web e desktop, sendo esta última a plataforma escolhida para o desenvolvimento do exemplo guia deste trabalho.

A API JPA disponibiliza um conjunto de interfaces de programação que permitem o armazenamento de objetos Java chamados de entidades do BD. De forma simplificada, pode-se entender uma entidade como um objeto Java regular (ou POJO - Plain Old Java Object) [WIKI09e]. A API é utilizada na camada de persistência para o desenvolvedor obter uma maior produtividade, pois controla a persistência de objetos dentro de Java. Ou seja, é a implementação da API que irá se preocupar em transformar os objetos em linhas e colunas nas tabelas do BD.

A entidade é um objeto Java mapeado para um BDR. Para isso, é necessário que seja declarada explicitamente uma entidade a fim de distingui-la de outros objetos Java regulares que podem ser usados na aplicação, e é preciso definir uma forma de mapear a entidade em uma das tabelas do BD. Uma vez que estas informações são consideradas metadados, JPA baseia-se no padrão Java SE 5 [Sun09d] para prover estes metadados no formato de *annotations*. JPA também permite o uso de XML para definir estes metadados [HGR09]. No entanto, *annotations* são mais fáceis de entender e implementar. Por isso, escolheu-se este formato para o desenvolvimento do exemplo guia. Ressalta-se que uma vez feito o ORM, se houver uma mudança da solução que implemente JPA, não será necessário realizar alterações significantes no ORM, pois qualquer solução que implemente JPA será capaz de entender a linguagem de mapeamento. Apenas bibliotecas e arquivos de configuração deverão sofrer alterações relacionadas à nova implementação [Sil08]. Além disto, outras vantagens tornam o uso de JPA atraente [Oco07]:

- Não é necessária a criação de objetos de acesso a dados complexos;

- A API auxilia no gerenciamento de transações;
- Possibilidade de escrever código padrão que interage com qualquer base de dados relacional, livre de código específico de fabricantes;
- É possível descartar SQL, dando preferência a uma linguagem de consulta que usa os nomes das classes e suas propriedades; e
- É possível usar e gerenciar POJO.

Atualmente, há várias implementações de JPA no mercado. Entre elas, pode-se citar: EclipseLink (eclipse) [Ecl09a], TopLink (Oracle) [Ora09]a, Hibernate (RedHat) [Hib09b], TopLink Essentials (GlassFish) [Gla09], Kodo (Oracle) [Ora09b], OpenJPA (Apache) [Apa09a] e Ebean (SourceForge) [Sou09]. Cada implementação apresenta seus pontos positivos e negativos, levando em consideração licenças, velocidade de processamento de consultas, erros, bibliotecas adicionais, entre outros fatores. Para a implementação do exemplo guia deste trabalho foram escolhidas as ferramentas Hibernate e Eclipse, pois estas são bem consolidadas e aceitas pela comunidade de desenvolvedores.

2.2. Exemplo Prático

Esta seção segue a sistemática apresentada na seção 1.3.2 e apresenta como o exemplo guia da seção 1.3.1 foi implementado usando JPA (Hibernate com Eclipse).

2.2.1. Configuração do ambiente

Para iniciar o desenvolvimento de uma aplicação utilizando JPA, é necessário que sejam feitas algumas configurações no ambiente de desenvolvimento. Este exemplo requer Java 5 (JDK 1.5) [Sun09i] ou superior e Eclipse 3.2 [Ecl09b] ou superior. É necessária também a biblioteca que contenha javax.persistence e os vários *annotations* associados ao JPA. Neste caso, como a implementação da API utilizada é a do Hibernate, as seguintes bibliotecas serão necessárias:

- Hibernate 3.3.1 GA
  - antlr-2.7.6.jar
  - commons-collections-3.1.jar
  - dom4j-1.6.1.jar

    - hibernate3.jar
    - javassist-3.4.GA.jar
- Hibernate *Annotations* 3.4.0 GA
    - hibernate-*annotations* .jar
    - hibernate-commons-*annotations* .jar
- Hibernate Entity Manager 3.4.0 GA
    - hibernate-entitymanager.jar
- SLF4J 1.5.8
    - slf4j-api-1.5.8.jar
    - slf4j-simple-1.5.8.jar
- Open EJB 3.1.1
    - javaee-api-5.0-2.jar

A especificação de javax.persistence está incluída na biblioteca javaee-api-5.0-2.jar. Além das bibliotecas para configurar JPA e o Hibernate, também é necessária a biblioteca do BD escolhido. Neste caso, será usado o SGBD MySQL [Mys09], e a biblioteca utilizada será mysql-connector-java-5.1.10-bin.jar.

2.2.2. Criação do Projeto

Para seguir com a implementação, é necessário criar o projeto e adicionar as bibliotecas citadas na seção anterior, para que o sistema possa operar sem erros de configuração.

O próximo passo é a criação das classes que serão utilizadas. O projeto pode conter tanto classes que serão mapeadas para a base de dados como classes de auxílio que não farão parte do banco. Para diferenciar essas classes, é necessário indicar explicitamente quais delas serão mapeadas em entidades [ASF09a].

Os passos descritos a seguir são baseados na API especificada por JPA e não correspondem às características de implementação do Hibernate, podendo ser seguidos para desenvolvimento de aplicações que utilizem qualquer solução citada na seção 2, como TopLink ou EclipseLink, por exemplo, ou que implementem JPA.

2.2.3. Criação de Entidades

Para exemplificar a criação de entidades, será utilizada a classe Pesquisador. O primeiro passo para a criação das entidades, é a criação da classe simples, com seus atributos, métodos get e set, e o construtor. A única restrição imposta por JPA é que seja inserido um construtor sem argumentos, mas não impede que outros construtores também sejam adicionados. A Figura 3 mostra como foi definida a classe Pesquisador.

```
public class Pesquisador {

    private int id;
    private String nome;
    private String areaPesquisa;

    private Set<Artigo> artigos = new HashSet<Artigo>();

    public Pesquisador(){

    }

    public Pesquisador(String nome, String areaPesquisa) {
        setNome(nome);
        setAreaPesquisa(areaPesquisa);
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }
    {...}
```

Figura 3 – Classe básica Pesquisador

2.2.4. Definindo o ORM

O ORM é o mapeamento dos objetos do sistema em uma estrutura tabular do BDR. Nesta seção, serão explicados os passos necessários para mapear os objetos e seus relacionamentos, herança, atributos multivalorados, compostos, entidades fracas e atributos de relacionamento.

2.2.4.1. Definir Tabelas e Colunas

Para converter a classe Pesquisador em uma entidade válida, basta adicionar dois *annotations* do pacote javax.persistence:

- @Entity – para declarar que a classe é uma entidade; e
- @Id – para declarar o identificador único, correspondendo à chave primária na base de dados.

A *annotation* @Entity é inserida da definição da classe. É essa informação que irá dizer que essa entidade corresponderá a uma tabela no BD. A *annotation* @Id pode ser inserida tanto na definição do campo que será o identificador da chave primária como no método get correspondente. Escolhendo a posição da *annotation* @Id, também é escolhido o mecanismo pelo qual a implementação da JPA acessará a entidade. Se a *annotation* for inserida em um campo, a implementação de JPA irá ler e escrever dados diretamente dos campos da instância da entidade (field bases access). Se a posição escolhida for no método get, a implementação de JPA irá preferir trabalhar com os métodos gets e sets, ignorando qualquer variável da instância que pode estar disponível (property based access) [HGR09].

Apesar de neste exemplo ser usada uma chave primária única, também é possível o mapeamento de chaves compostas. Para isso, será utilizado o exemplo da Figura 4. Como pode ser observado na figura, a chave composta será formada pelo atributo nome e sobrenome. Então, deve-se criar uma classe com esses atributos, e declará-la @Embeddable. Dessa forma, todas as classes que tiverem um atributo do tipo ContatoPK, terá na sua tabela relativa os atributos nome e sobrenome, e não uma referência a ContatoPK. Sendo assim, declara-se um atributo contatoPK em Contato do tipo @EmbeddedId. Dessa forma, os atributos nome e sobrenome vão ser mapeados para a tabela Contato como chave composta [BL06].

```java
@Embeddable
public class ContatoPK {

    @Column(nullable=false)
    private String nome;
    @Column(nullable=false)
    private String sobrenome;
    {...}
```

```java
@Entity
public class Contato {

    @EmbeddedId
    private ContatoPK contatoPK;
    {...}
```

Figura 4 – Exemplo de chave primária composta

Voltando para o exemplo a ser estudado, a declaração de entidade e de chave primária ficaria como apresentado na Figura 5:

```java
@Entity
public class Pesquisador {

    @Id
    @GeneratedValue
    private int id;
    private String nome;
    private String areaPesquisa;

    private Set<Artigo> artigos = new HashSet<Artigo>();

    public Pesquisador(){

    }
    {...}
```

Figura 5 – Exemplo de chave primária única e declaração de entidade

Além da *annotation* @Id, pode-se inserir a *annotation* @GeneratedValue (ver Figura 5), que especifica que o valor da chave primária será gerado automaticamente pelo sistema seguindo uma sequência, partindo de 1.

Além dessas *annotation*, também pode-se definir os nomes das tabelas e das colunas do banco inserindo @Table na definição da classe e @Column na definição dos atributos. No entanto, JPA faz suposições quanto ao nome da tabela e das colunas, se estes dados não forem explicitados. Então, caso se deseje alterar o nome das tabelas e das colunas para que seja diferente do nome das classes e atributos, deve-se deixar claro inserindo essas informações.

O Quadro 1 mostra as principais regras padrão quanto a essas definições.

Área	Regra Padrão	Ignorada por
Nome da tabela	Uma entidade é mapeada geralmente em uma tabela tendo o mesmo nome que a classe. Nesse caso, o mapeamento respeita letras maiúsculas e minúsculas, sendo considerado tudo maiúscula.	@Table
Campos persistentes	Todos os campos ou propriedades são persistentes	@Transient
Nome de colunas	Um campo persistente ou propriedade é mapeado em uma coluna com o mesmo nome. Segue a mesma regra de maiúsculas e minúsculas do nome de tabelas.	@Column

Quadro 1 - Regras para mapeamento de entidades [HGR09]

A partir do Quadro 1, a classe Pesquisador pode ficar com o nome da tabela "TB_PESQUISADOR", o atributo id com o nome "idPesquisador" e os demais atributos com o nome especificado na classe, caso o mapeamento seja realizado da forma mostrada na Figura 6.

```
@Entity
@Table (name="TB_PESQUISADOR")
public class Pesquisador {

    @Id
    @GeneratedValue
    @Column (name="idPesquisador")
    private int id;
    private String nome;
    private String areaPesquisa;

    private Set<Artigo> artigos = new HashSet<Artigo>();
    {...}
```

Figura 6 – Mapeamento da Entidade Pesquisador

2.2.4.2. Definir Herança

Em Java é uma situação normal a presença de herança em aplicações. JPA disponibiliza 3 formas de mapeamento de herança. Para tornar o estudo de JPA mais completo, será utilizado o esquema da Figura 7, que possibilita a exemplificação de todas as formas de

mapeamento de herança suportadas por JPA e ao final será descrita a escolha da estratégia de mapeamento escolhida para este trabalho.

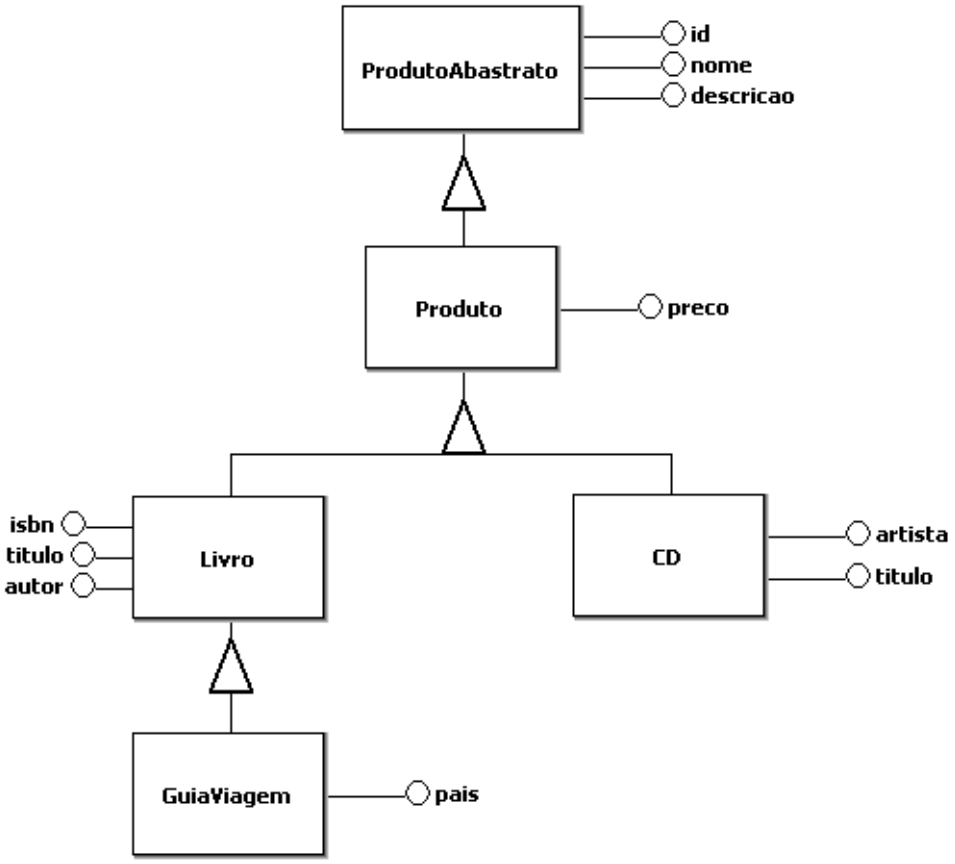

Figura 7 – Exemplo de herança [Dat09b]

- A estratégia padrão é chamada de SINGLE TABLE. Escolhendo esta estratégia, a classe base, ou seja, o "topo" da hierarquia de herança, será mapeada para uma tabela e todas as subclasses serão persistidas nessa tabela. Dessa forma, existirá apenas uma tabela com todos os atributos das classes pertencentes a essa hierarquia de herança. O esquema ficaria da forma mostrada na Figura 8.

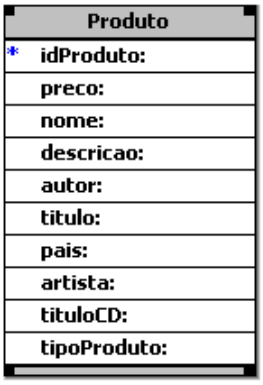

Figura 8 – Herança SINGLE_TABLE [Jpo09]

- Outra forma é obter uma tabela para cada classe da hierarquia de herança, e cada tabela terá apenas colunas para os atributos definidos na sua classe. Os campos da superclasse são persistidos na tabela da superclasse. Consequentemente, para recuperar todos os valores dos campos de uma subclasse, precisa ser feito um *join* de todas as tabelas das superclasses da hierarquia. Em JPA, esta forma é chamada de JOINED. A Figura 9 mostra como as tabelas seriam persistidas no BD.

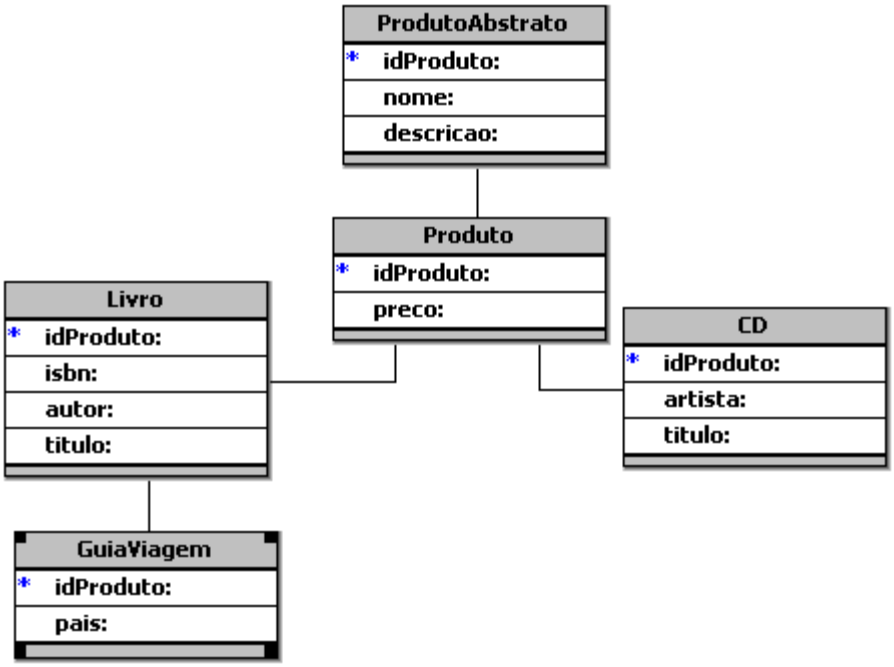

Figura 9 – Herança JOINED [Jpo09]

- A terceira forma é como a segunda, exceto pelo fato de que cada tabela terá as colunas para todos os campos herdados. Esta forma é chamada de TABLE_PER_CLASS. A Figura 10 mostra como as classes seriam persistidas no BD.

Figura 10 – Mapeamento TABLE_PER_CLASS [Jpo09]

No exemplo guia da seção 1.3.1 há uma herança entre Pesquisador e Bolsista, onde um Pesquisador pode ser um bolsista. A forma escolhida para implementar a herança foi a segunda, a JOINED, de forma que ao criar o esquema no BD, haverá uma tabela Pesquisador e uma Bolsista, onde a chave primária de Bolsista será a chave primária de Pesquisador e os campos em cada tabela serão os campos definidos em cada classe.

Na Figura 11 e na Figura 12 é possível observar como foi feito o mapeamento para esse tipo de herança.

```
@Entity
public class Bolsista extends Pesquisador {

    private String tipo;
    private float valor;

    public Bolsista(){

    }

    public Bolsista(String nome, String instituicao, String tipo, float valor){
        super(nome, instituicao);
        this.setTipo(tipo);
        this.setValor(valor);
    }
    {...}
```

Figura 11 – Mapeamento de herança em Bolsista

```java
@Entity
@Inheritance (strategy = InheritanceType.JOINED)
public class Pesquisador {

    @Id
    @GeneratedValue
    @Column(name="idPesquisador")
    private int id;
    private String nome;
    private String areaPesquisa;
    {...}
```

Figura 12 – Mapeamento de herança em Pesquisador

Como se pode observar, a superclasse Pesquisador recebe a *annotation* @Inheritance, com o valor de estratégia JOINED, que é responsável por informar que será esse o tipo de herança dessa hierarquia. Na classe Bolsista, apenas é implementada a herança em Java, usando o *extends*.

2.2.4.3. Definir Atributo Composto e/ou Multivalorado

Para definir um atributo multivalorado, não foi encontrada outra forma a não ser considerar uma relação 1:N, onde a entidade será o lado 1 e o atributo multivalorado será uma nova entidade (o lado N) onde a chave primária será a chave estrangeira da entidade maior e seus atributos serão o próprio valor do atributo. Como exemplo, pode-se considerar no exemplo guia, a entidade Comissão que possui o atributo multivalorado *membros*. Dessa forma, Membro passou a ser uma entidade e seus atributos passaram a ser nome, instituição e idComissao. Nesse caso, o atributo membros também é composto, mas a regra também vale para atributos simples.

Já no caso de um atributo composto sem ser multivalorado, há outra forma de persistência. É declarando o atributo composto como uma classe @Embeddable e na classe que possui esse atributo, na declaração do atributo, coloca-se @Embedded. Segue um exemplo na Figura 13 para auxiliar na compreensão.

```java
@Embeddable
public class Endereco {

    private String logradouro;
    private String cidade;
    private String cep;
    {...}
```

```java
@Entity
public class Pessoa {

    private String nome;

    @Embedded
    private Endereco endereco;

    {...}
```

Figura 13 – Mapeamento de atributos compostos

Nesta forma de mapeamento, os atributos de Endereço serão inseridos na tabela Pessoa e ao remover um objeto Pessoa, seu endereço também será automaticamente removido.

2.2.4.4. Definir Relacionamentos

Assim como entidades no mundo real, os objetos persistentes geralmente não trabalham sozinhos. Classes de entidades podem interagir com outras classes, tanto usando quanto provendo serviços.

Nas próximas subseções serão exemplificados os tipos de relacionamento que podem ocorrer entre classes: um-para-um (one-to-one), muitos-para-um (many-to-one) e muitos-para-muitos (many-to-many).

2.2.4.4.1. Um-Para-Um

Considerando o exemplo das entidades Evento e Comissao, onde cada evento possui uma Comissao e uma Comissão só pertence a um Evento, obtém-se um relacionamento do tipo um-para-um.

Para o caso onde não existe uma entidade fraca, uma das entidades terá uma chave estrangeira que apontará para a outra entidade. Sendo assim, foi considerado que a classe Evento teria um atributo idComissao, cujo valor corresponderá à sua referência na classe Comissao. A Figura 14 mostra como ficaria o mapeamento nas classes Evento e Comissao.

```java
@Entity
public class Evento {

    {..}

    @OneToOne (targetEntity=entidades.Comissao.class,
               cascade={CascadeType.ALL},
               fetch = FetchType.LAZY,
               optional=true)
    @JoinColumn (name="idComissao",
                 unique=true,
                 nullable=true,
                 insertable=true,
                 updatable=true)
    private Comissao comissao;

    {..}
```

```java
@Entity
public class Comissao {

    {..}

    @OneToOne(mappedBy="comissao")
    public Evento evento;

    {..}
```

Figura 14 – Mapeamento Um-Para-Um

No mapeamento acima foi estabelecida uma associação bidirecional, ou seja, cada uma das entidades possui uma *annotation* do tipo @OneToOne. Mas ao transpor o mapeamento para o BD, apenas a tabela Evento possuirá o atributo endereço devido ao *annotation* de JPA @JoinColumn, onde é especificado que esse atributo será persistido como uma coluna da tabela Evento e seu nome será idComissao. O atributo evento de Comissao é usado para, ao fazer uma busca por endereço, esse campo vem preenchido com o evento correspondente, que é mapeado pelo atributo endereço, localizado na entidade Evento.

Para o caso do esquema apresentado na seção 1.3.1, a entidade Comissão é fraca, ou seja, sua chave primária e estrangeira serão as mesmas, e serão iguais a chave primária de Evento, ou seja, não há inserção de uma nova coluna para a chave estrangeira. Sendo assim, a *annotation* @PrimaryKeyJoinColumn é utilizada no lugar da @JoinColumn, e invertida a entidade que recebe essa *annotation*, já que a entidade forte é Evento. Neste caso, o mapeamento deverá ser feito conforme mostrado na Figura 15.

```java
@Entity
public class Evento {

    @Id
    private int id;

    @OneToOne(mappedBy="evento")
    private Comissao comissao;

    {...}
```

```java
@Entity
public class Comissao {

    @Id
    public int id;

    @OneToOne
    @PrimaryKeyJoinColumn
    public Evento evento;

    {...}
```

Figura 15 - Mapeamento Um-Para-Um com entidade fraca

Segundo [BK07], a especificação de JPA não inclui um método padronizado para lidar com o problema de compartilhamento de geração de chave, o que significa que a aplicação deve ficar responsável por fornecer o valor do identificador da entidade fraca, no caso Comissao antes de salvá-la no banco.

O Quadro 2 e o Quadro 3 contém mais algumas características dos *annotations* utilizados para o mapeamento um-para-um.

OneToOne	Define uma relação de um para um
targetEntity	Define o nome completo (package + className) da
cascade	Define se as ações serão executadas em cascata, podendo ser: inserção (PERSIST), alteração (MERGE), remoção (REMOVE),atualização (REFRESH) ou todas (ALL)
fetch	Define se o relacionamento será carregado por demanda (LAZY) ou por imediato (EAGER)
optional	Define se o relacionamento é obrigatório ou não
mappedBy	Define o nome do atributo no relacionamento bidirecional

Quadro 2 - Atributos do mapeamento um-para-um [Sil08]

JoinColumn	Define a coluna do relacionamento
name	Define o nome da coluna do relacionamento
referencedColumnName	Define o nome da coluna de referencia caso não seja o identificador da outra Entidade
unique	Define se é uma coluna única
nullable	Define se é a coluna pode ter valor nulo
insertable	Define se é usada na inserção
updatable	Define se é usada na atualização

Quadro 3 - Atributos da coluna do relacionamento [Sil08]

2.2.4.4.2. Muitos-Para-Um

Para o mapeamento muitos-para-um, foi utilizado o exemplo do relacionamento entre Artigo e Evento, onde um Artigo pode participar de apenas um Evento, mas um Evento pode conter mais de um Artigo. Neste caso, a relação será caracterizada pela chave estrangeira da entidade de lado N no lado 1, ou seja, Artigo deverá conter um atributo idEvento, que será a referência para o Evento a que se relaciona.

Na Figura 16 pode-se observar como ficará a implementação.

```java
@Entity
public class Evento {
    {...}
    @OneToMany(targetEntity=entidades.Artigo.class,
               cascade = { CascadeType.MERGE},
               fetch=FetchType.LAZY,
               mappedBy="evento")
    @JoinColumn(name="idEvento")
    private Set<Artigo> artigos;
    {...}
```

```java
@Entity
public class Artigo {
    {...}
    @ManyToOne(targetEntity=entidades.Evento.class,
               cascade = {CascadeType.MERGE},
               fetch = FetchType.LAZY,
               optional=true)
    @JoinColumn(name = "idEvento")
    private Evento evento;
    {...}
```

Figura 16 – Mapeamento muitos-para-um

Observa-se que as características do *annotation* @ManyToOne e @OneToMany também estão presentes no mapeamento @OneToOne, e possuem o mesmo significado (ver Quadro 2 e Quadro 3). Também vale ressaltar que a declaração de algumas dessas características não são obrigatórias e também é possível declarar de várias formas, desde que não vá de encontro com as regras de mapeamento. Caso isso ocorra, um erro será gerado pelo sistema.

2.2.4.4.3. Muitos-Para-Muitos

Já para o mapeamento muitos-para-muitos, foi utilizado o exemplo do relacionamento entre Artigo e Pesquisador, onde um Artigo pode ser escrito por um ou mais Pesquisadores, assim como um Pesquisador pode escrever um ou mais Artigos. Um relacionamento sem atributo deve gerar uma tabela extra, contendo suas chaves primárias, enquanto um relacionamento com atributos deve gerar uma tabela contendo as chaves primárias das entidades relacionadas e os atributos da relação. O exemplo mostrado na Figura 17 apresenta um mapeamento M:N sem atributo.

```java
@Entity
@Inheritance (strategy = InheritanceType.JOINED)
public class Pesquisador {
      {...}

      @ManyToMany(targetEntity = entidades.Artigo.class,
                  mappedBy="pesquisadores",
                  cascade={CascadeType.MERGE},
                  fetch=FetchType.EAGER)
      private Set<Artigo> artigos = new HashSet<Artigo>();

      {...}
```

```java
@Entity
public class Artigo {
    {...}

    @ManyToMany(cascade={CascadeType.MERGE},
                fetch=FetchType.EAGER)
    @JoinTable(name = "ESCREVE",
            joinColumns = {
                @JoinColumn(name = "idArtigo")},
            inverseJoinColumns = {
                @JoinColumn(name = "idPesquisador")
            })
    private Set<Pesquisador> pesquisadores =
            new HashSet<Pesquisador>();
    {...}
```

Figura 17 – Mapeamento muitos-para-muitos sem atributo na relação

Nesse mapeamento, pode-se notar que aparece um novo *annotation*, JoinTable, que é usado para mapear esse relacionamento em uma nova tabela, que conterá joinColumns e inverseJoinColumns. Nesse caso, idArtigo e idPesquisador, e a nova tabela se chamará "ESCREVE". Já para as características de @ManyToMany não houve mudanças quanto ao significado de cada uma. Além de estabelecer uma nova tabela, o JoinTable permite que, ao fazer uma consulta à tabela de Artigo, o campo pesquisadores venha carregado com os pesquisadores que escreveram o artigo, assim como acontece se a busca for por Pesquisador.

Para um relacionamento M:N com atributo, deve-se criar uma entidade extra, e serão consideradas relações 1:N entre a entidade extra e as entidades da relação. A Figura 18 apresenta como ficou o mapeamento nas classes Artigo e Pesquisador para relacionamento com atributo.

```
@Entity
public class Artigo {
    {...}
    @OneToMany(mappedBy = "artigo")
    private Set<ArtigoPesquisador>
                pesquisadorRelacao = new HashSet<ArtigoPesquisador>();
    {...}
```

```
@Entity
@Inheritance (strategy = InheritanceType.JOINED)
public class Pesquisador {
    {...}
    @OneToMany(mappedBy="pesquisador")
    private Set<ArtigoPesquisador>
            artigoRelacao = new HashSet<ArtigoPesquisador>();
    {...}
```

Figura 18 - Mapeamento em Artigo e Pesquisador de relacionamento M:N com atributo

O relacionamento M:N então se torna 2 relacionamentos 1:N. Na Figura 18 é possível ver os lados 1 da relação. Na Figura 19, é possível ver o lado N da relação, a classe extra ArtigoPesquisador. Nesta classe serão especificados atributos que se relacionam com Artigo e Pesquisador. Ao definir @JoinColumn, especifica-se que a entidade terá uma chave estrangeira para a entidade em questão. E ao definir os identificadores da classe com o mesmo nome das chaves estrangeiras, a entidade terá as chaves primárias e estrangeiras persistidas na mesma coluna.

```
@Entity
@Table (name="ArtigoPesquisador")
@IdClass(entidades.ArtigoPesquisadorId.class)
public class ArtigoPesquisador {

    @Id
    @Column(name="idArtigo")
    private int idArtigo;
    @Id
    @Column(name="idPesquisador")
    private int idPesquisador;

    @ManyToOne
    @JoinColumn(name="idArtigo")
    private Artigo artigo;
    @ManyToOne
    @JoinColumn(name="idPesquisador")
    private Pesquisador pesquisador;

    private int ordem;
    {...}
```

```
public class ArtigoPesquisadorId
                implements Serializable{


    private int idArtigo;
    private int idPesquisador;
    {...}
```

Figura 19 - Mapeamento na classe ArtigoPesquisador

2.2.5. Definir Propriedades da Persistência

O esquema de dados de uma aplicação JPA consiste geralmente de diversas entidades de classes, as quais precisam ser mapeadas juntas na mesma base de dados. As classes de entidade formam uma unidade lógica que é chamada de unidade de persistência (persistence unit). Em uma aplicação JPA, é necessária a definição da unidade de persistência a partir da configuração de um arquivo chamado "persistence.xml". É possível, mas não obrigatório, que sejam listadas explicitamente as entidades que formam a unidade de persistência. Se não forem listadas, a implementação JPA varre a aplicação e detecta as entidades automaticamente. Neste

arquivo também podem ser incluídas características da implementação JPA, como o driver do banco utilizado, a base de dados, a senha, entre outras informações [ASF09a].

Quanto a base de dados, é indicado que seja sempre especificada nesse arquivo. Tecnicamente, especificar o nome da fonte de dados relacionada à unidade de persistência é opcional no arquivo persistence.xml. Se o nome da fonte de dados é omitida, alguns provedores automaticamente utilizam um construtor padrão de fonte de dados, outros assumem um nome padrão, mas não provém a fonte de dados propriamente dita, e outros podem rejeitar a unidade de persistência por estar incompleta. Por essa razão, é extremamente recomendado que o nome da fonte de dados a ser usada seja sempre especificado.

De acordo com a aplicação utilizada nesse trabalho, o arquivo de configuração ficou como mostrado na Figura 20.

```
    <persistence-unit name="gerenciadorEventos"
                      transaction-type="RESOURCE_LOCAL">
            <class>entidades.Artigo</class>
            <class>entidades.Evento</class>
            <class>entidades.Pesquisador</class>
            <class>entidades.Bolsista</class>
            <class>entidades.Comissao</class>
            <class>entidades.Membro</class>

        <properties>
            <property name="hibernate.show_sql" value="false" />
            <property name="hibernate.format_sql" value="false" />

            <property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Driver" />
            <property name="hibernate.connection.url" value="jdbc:mysql://localhost/monografiajpa" />

            <property name="hibernate.connection.username" value="root" />
            <property name="hibernate.connection.password" value="123456" />

            <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
            <property name="hibernate.hbm2ddl.auto" value="update" />
        </properties>
    </persistence-unit>
</persistence>
```

Figura 20 – Arquivo persistence.xml

Este arquivo de configuração se repetirá para outras implementações de JPA, havendo mudanças apenas no que está dentro da tag <properties>, onde serão especificadas as informações da base de dados a ser utilizada

2.2.6. Trabalhar a Persistência

Em uma aplicação com JPA, trabalha-se normalmente com as entidades, já que estas são objetos simples de Java. Mas para que as mudanças sejam refletidas no banco, é preciso utilizar o *entity manager*, que irá gerenciar as instâncias de entidades. Para que isso ocorra, é preciso que se coloque explicitamente a entidade sob o controle do referido *entity manager*. Com ele controlando a entidade, suas mudanças serão sincronizadas com a base de dados. Uma vez que esse controle acabe, a entidade volta a ser apenas um objeto Java.

A interface javax.persistence.EntityManager é utilizada para interagir com o entity manager. Serão mostradas as 4 formas básicas de interação: inserção, alteração, exclusão e busca.

Para realizar as operações, é necessário primeiramente inicializar o EntityManager, como pode-se ver na Figura 21.

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

public class EMUtil {
    private static EntityManagerFactory emf;

    public static EntityManager getEntityManager() {
        if (emf == null){
            emf = Persistence.createEntityManagerFactory("gerenciadorEventos");
        }
        EntityManager em = emf.createEntityManager();
        return em;
    }

    public static void fechar(){
        emf.close();
    }

}
```

Figura 21 – Inicializando o EntityManager

O nome passado no método createEntityManagerFactory deverá ser o mesmo nome especificado no arquivo persistence.xml, no campo nome da tag <persistence-unit> visto na seção 2.2.5.

2.2.6.1. Inserção

Para realizar uma inserção é simples. Basta passar o objeto criado com os valores dos atributos como parâmetro para o método *persist( )*, como mostrado na Figura 22

```
public void cadastrar(Artigo artigo) {

    EntityManager entityManager = EMUtil.getEntityManager();
    EntityTransaction transaction = entityManager.getTransaction();

    if (artigo == null) {
        return;
    }
    try{
        transaction.begin();
        entityManager.persist(artigo);
        transaction.commit();
    } catch (Exception e){
        System.out.println("Erro ao cadastrar artigo");
    }

}
```

Figura 22 – Inserindo um objeto

2.2.6.2. Alteração

A operação de alteração também segue o mesmo raciocínio, apenas mudando o método utilizado, que neste caso será o *merge( )*, como mostrado na Figura 23.

```
public void atualizar(Artigo artigo) {

    EntityManager entityManager = EMUtil.getEntityManager();
    EntityTransaction transaction = entityManager.getTransaction();
    try {
        transaction.begin();
        entityManager.merge(artigo);
        transaction.commit();
    } catch (Exception e){
        System.out.println("Erro ao atualizar");
    }

}
```

Figura 23 – Alterando um objeto

2.2.6.3. Exclusão

Para excluir um objeto, é preciso que ele já esteja em poder do *entity manager*. Para isso, pode-se dentro do escopo do mesmo *entity manager* dar um *merge* ou um *find*, e aí sim chamar o método *remove( )*. Segue o exemplo na Figura 24.

```
public void excluir(Artigo artigo) {

    EntityManager entityManager = EMUtil.getEntityManager();
    EntityTransaction transaction = entityManager.getTransaction();

    if (artigo == null) {
        return;
    }
    try{
        transaction.begin();
        Artigo a = entityManager.find(Artigo.class, artigo.getId());
        entityManager.remove(a);
        transaction.commit();
    } catch (Exception e){
        System.out.println("Erro ao deletar");
    }
}
```

Figura 24 – Removendo um objeto

2.2.6.4. Busca

Para buscar um objeto, é preciso informar o tipo e o id da entidade, que serão passados como parâmetro no método *find( )*, como mostrado na Figura 25:

```
public Artigo buscar(Artigo artigo) {
    Artigo retorno=null;

    EntityManager entityManager = EMUtil.getEntityManager();
    EntityTransaction transaction = entityManager.getTransaction();
    try {
        transaction.begin();
        retorno = entityManager.find(Artigo.class, artigo.getId());
        transaction.commit();
    } catch (Exception e){
        System.out.println("Erro ao buscar");
    }

    return retorno;

}
```

Figura 25 – Buscando um objeto

2.3. Conclusão

Este capítulo introduziu os principais conceitos sobre JPA, e em seguida mostrou como realizar ORM utilizando esta API. O exemplo prático foi divido em várias partes, sendo usado o esquema apresentado na seção 1.3.1 para exemplificar o mapeamento. Foi visto que JPA consegue abordar grande parte dos conceitos relacionais, apenas apresentando problemas em atributos multivalorados. Ao final, foram apresentadas as 4 formas básicas de interação com o BD: inserção, atualização, remoção e busca, onde foi visto que não foi necessário o uso de linguagens de consulta.

O próximo capítulo apresentará uma introdução sobre JDO e em seguida um exemplo prático, seguindo o mesmo esquema do apresentado neste capítulo, utilizando a API JDO.

3. Java Data Objects

Este capítulo apresenta os principais conceitos de JDO e como esta API foi utilizada para implementar o exemplo guia da seção 1.3.1, seguindo a sistemática de estudo apresentada na seção 1.3.2.

3.1. Conceitos Básicos de JDO

A especificação de JDO [Sun09c], JSR12 [Web09b], define uma API de padronização de ORM e persistência transparente de POJO e acesso a BD. Usando JDO, desenvolvedores de aplicações em Java podem escrever código para acessar os dados armazenados no BD sem usar qualquer tipo de linguagem específica para este fim.

Os dois principais objetivos da arquitetura de JDO são prover uma visão transparente de persistência de dados aos desenvolvedores e permitir implementações de armazenamento de dados em servidores de aplicação. É importante notar que JDO não define o tipo do BD. Pode ser utilizada a mesma interface para persistir objetos Java em bases de dados relacional, objeto-relacional, XML ou qualquer outra base [Sun09c]. Alguns dos benefícios do uso de JDO são [Sun09c]:

- *Portabilidade* - aplicações escritas usando a API de JDO podem operar com várias implementações disponíveis de diferentes distribuidores sem precisar mudar uma linha de código ou recompilar;
- *Acesso transparente à base de dados* - desenvolvedores de aplicação escrevem código para acessar dados armazenados sem precisar utilizar código de acesso à base de dados específico;
- *Facilidade de uso* - a API JDO permite que desenvolvedores de aplicação foquem em seus esquemas de objetos e deixem os detalhes de persistência à implementação de JDO;
- *Alto desempenho* - esta tarefa é delegada às implementações Java que podem melhorar os padrões de acesso a dados para obter um melhor desempenho; e
- *Integração com EJB* - aplicações podem tirar vantagem dos recursos de EJB como processamento remoto de mensagens, coordenação de transações automáticas distribuídas e segurança.

São encontradas diversas implementações da API JDO. Algumas mais desenvolvidas e otimizadas que outras. O Quadro 4 mostra algumas das implementações:

Nome	Licença	JDO Espec	Armazenamento de Dados
DataNucleusAccessPlatform	Não-Comercial	1.0, 2.0, 2.1, 2.2, 2.3	RDBMS, db4o, LDAP, Excel, XML, NeoDatis, JSON, OpenDocumentFormat (ODF), Google BigTable, HADOOP HBase
JDOInstruments	Não-Comercial	1.0	JDOInstruments
JPOX	Não-Comercial	1.0, 2.0, 2.1	RDBMS, db4o
Kodo	Comercial	1.0, 2.0	RDBMS, XML
ObjectDB para Java/JDO	Comercial	1.0, 2.0	ObjectDB
Objectivity	Comercial	1.0	ObjectivityDB
Orient	Comercial	1.0	Orient
hywy's PE: J	Comercial	1.0	RDBMS
SignSoft intelliBO	Comercial	1.0	intelliBO
Speedo	Não-Comercial	1.0	RDBMS
TJDO	Não-Comercial	1.0	RDBMS
Versant	Comercial	1.0, 2.0	Versant Object Database
Xcalia	Comercial	1.0, 2.0	RDBMS, XML, Versant ODBMS, Jalisto, Web services, transações de mainframe e telas (CICS, IMS...), pacotes de aplicativos (ERP, CRM,SFA...), componentes (EJB...).

Quadro 4 - Implementações de JDO [ASF09a]

Para a implementação do exemplo guia da seção 1.3.1, será usada a implementação DataNucleusAccess, por ser considerada uma das mais desenvolvidas e já utilizar a nova versão JDO 2.3 e a IDE Eclipse 3.2.

3.2. Exemplo Prático

Esta seção segue a sistemática apresentada na seção 1.3.2 e apresenta como o exemplo guia da seção 1.3.1 foi implementado usando JDO (DataNucleusAccess com Eclipse) [Dat09a]. Ressalta-se que, assim como a implementação do exemplo guia com JPA não está limitada ao uso de Hibernate, a implementação do exemplo guia com JDO também não está restrita à implementação do DataNucleusAccess.

3.2.1. Configuração do ambiente

Para a implementação do exemplo guia da seção 1.3.1 foi usado um plugin da implementação DataNucleusAccess para o Eclipse e o SGBD escolhido foi o MySQL 5.1. Como o objetivo é analisar a API de JDO, serão omitidos detalhes a respeito desse plugin. Mas, para maiores informações basta acessar o site do distribuidor: http://www.datanucleus.com/. Para dar início a implementação, são necessárias as seguintes bibliotecas:

- datanucleus-core-1.1.3.jar
- datanucleus-enhancer-1.1.3.jar
- datanucleus-jpa-1.1.3.jar
- datanucleus-rdbms-1.1.3.jar
- asm-3.1.jar
- jdo2-api-2.3-ea.jar
- log4j-1.2.8.jar
- mysql-connector-java-5.1.10-bin.jar

3.2.2. Criação do Projeto

Para seguir com a implementação, é necessário criar o projeto e adicionar as bibliotecas citadas na seção anterior, para que o sistema possa operar sem erros de configuração.

O próximo passo é a criação das classes que serão utilizadas. JDO separa as classes de uma aplicação, através dos metadados, em três categorias [Mah05]:

- Persistence-capable - esta categoria representa classes em que suas instâncias podem ser persistidas em uma base de dados. Essas classes precisam ser "aprimoradas" (*enhanced*) de acordo com a especificação do metadado JDO antes de serem usadas em um ambiente JDO;
- Persistence-aware - essas classes manipulam classes persistence-capable. A classe JDOHelper [Apa09b] provê métodos que permitem interrogar o estado de persistência de uma instância de classe persistence-capable. Essas classes são "aprimoradas" com o mínimo de metadados; e
- Normal - essas classes não são persistentes e não possuem conhecimento de persistência. Não necessitam de nenhum metadado.

3.2.3. Criação de Entidades

A criação de entidades em JDO não difere muito da criação de classes normais de projetos Java, ou seja, não requerem muito código diferente. É necessária apenas a definição do tipo de classe definido na seção anterior, para determinar como deve ser persistida. Com JDO 2.3 é possível fazer isso de várias formas:

- XML Metadado;
- *Annotations*;
- *Annotations* + XML; e
- API de Metadados em tempo de execução.

Para esse trabalho, a informação de persistência básica foi feita com *annotations* e depois foi adicionado informação em XML. A definição é feita conforme a Figura 26.

```
@PersistenceCapable
public class Artigo {

    private String titulo;
    private float nota;

    private Set<Pesquisador> pesquisadores = new HashSet<Pesquisador>();

    private Evento evento;

    public Artigo(){

    }
    {...}
```

Figura 26- Definição do tipo de classe

Da mesma forma como JPA, JDO requer um construtor padrão, como pode ser visto na Figura 26. Depois de definir o tipo de classe, é definida a forma de persistência dos atributos da classe.

3.2.4. Definindo o ORM

JDO utiliza arquivos de metadados baseados em XML para especificar informações relativas à persistência incluindo quais classes da aplicação deverão ser persistidas. O arquivo de metadados pode conter informações de uma única classe persistente ou um ou mais pacotes onde haverá a definição de várias classes persistentes.

Este arquivo pode definir propriedades de persistência – para uma classe ou um pacote inteiro – em um ou mais arquivos XML. O nome do arquivo para uma classe deverá ser o mesmo nome da classe, seguido da extensão .jdo. Portanto, o arquivo de metadados para a classe Artigo deverá se chamar Artigo.jdo, e deverá estar localizado no mesmo diretório do arquivo Artigo.class. Os metadados para um pacote inteiro deverão estar contidos em um arquivo chamado package.jdo, que poderá conter informações de várias classes e vários sub-pacotes [Han05]. A ordem de busca pelo arquivo de metadados é a seguinte:

- META-INF/package.jdo
- WEB-INF/package.jdo
- package.jdo
- entidades/package.jdo
- entidades/package.jdo
- entidades/Artigo.jdo

Para definir que o arquivo é um metadado de JDO, o mesmo deve começar com o cabeçalho definindo o XML e a DTD utilizada. Após essa definição, abre-se uma tag <jdo> e <package>, onde será especificado o nome do pacote a que esse arquivo XML se refere. A Figura 27 mostra como deve ser o início do arquivo de mapeamento.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo PUBLIC
    "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
    "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
    <package name="entidades">

    (...)
```

Figura 27 – Cabeçalho do arquivo de mapeamento

Nesta seção, serão explicados os passos necessários para mapear os objetos e seus relacionamentos, herança, atributos multivalorados, compostos, entidades fracas e atributos de relação.

3.2.4.1. Definir Tabelas e Colunas

O primeiro passo para definir o mapeamento é a declaração das tabelas. Para isso, utiliza-se a tag <class>, que possui atributos como *nome*, onde será especificada qual classe esse mapeamento se refere, *table*, que define o nome da tabela no BD referente a essa classe e identity-type, onde será decidido se a implementação do JDO vai gerar as identidades dos objetos ou se será a aplicação. Nesse tutorial foi escolhida a opção de deixar a definição das identidades para a implementação. Para especificar o tipo de identidade, tem-se a propriedade "identity-type", que pode assumir dois valores: datastore ou application. O primeiro especifica que a identidade dos objetos será definida pela implementação de JDO, já o segundo especifica que a aplicação será responsável pela definição da identidade. Essa propriedade é vista na Figura 28.

```
<class
      name="Artigo" identity-type="datastore">

    <datastore-identity>
        <column name="idArtigo"/>
    </datastore-identity>

    (...)
```

Figura 28 – Definição de identity-type

Quando não se define o atributo *table*, a tabela será criada com o nome da classe. Na tag <column>, é definido o nome que o atributo id terá na tabela Artigo, nesse caso "idArtigo". Além de nome, esta tag também contém definições de tamanho e tipo do atributo, podendo ser definida da seguinte maneira:

```
<column name="IDENTIFIERNAME" length="100" jdbc-type="VARCHAR"/>
```

Para definir o mapeamento dos atributos da classe, é especificada a tag <field>, onde serão definidos todos os atributos persistentes da classe em questão. Na tag <field>, podem-se definir características quanto a forma de persistência, propriedades quanto a relacionamentos e

definir que um determinado atributo é chave primária. Para definir uma chave primária, utiliza-se "primary-key=*true*". No exemplo da Figura 29, esse campo está como *false*, pelo atributo não se tratar de uma chave primária. Para uma chave composta, basta inserir "primary-key=true" nos campos que compõem a chave.

```
<class
      name="Artigo">

    {...}
    <field
          name="titulo"
          primary-key="false"
          persistence-modifier="persistent"/>

    {...}
```

Figura 29 – Mapeamento de atributos das classes

3.2.4.2. Definir Herança

Para mapear heranças em JDO também existem três formas, assim como JPA. E para um melhor entendimento serão exemplificados os três tipos. A Figura 7 (que mostra um diagrama de classes UML para representar a herança de produtos) também será utilizada nesta seção, pela mesma razão explicada na seção sobre herança de JPA.

- A primeira opção, mais simples de entender, é a *new-table*. Nesta forma, cada classe possui sua própria tabela na base de dados. Tem a vantagem de ser a definição mais normalizada de dados, porém tem como desvantagem o fato de ser lenta em desempenho, já que serão necessários acessos a várias tabelas para encontrar um objeto de uma subclasse. Para fazer o mapeamento, deve-se inserir apenas o seguinte código no arquivo de metadados em cada classe:

  "<inheritance strategy="new-table"/>"

  Neste mapeamento, cada tabela conterá seus próprios atributos e a chave primária de cada uma será a chave primária da classe do topo da hierarquia (ProdutoAbstrato, no caso), como pode-se observar na Figura 30.

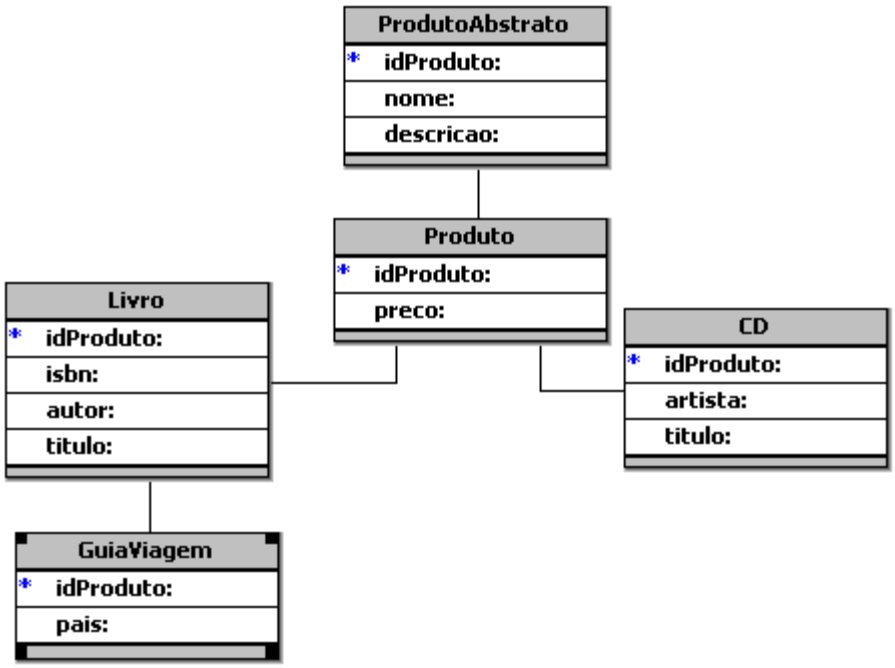

Figura 30 – Herança new-table [Dat09b]

- A segunda forma é a chamada subclass-table. Nessa opção, é preciso escolher uma classe para ter seus campos persistidos na tabela de sua subclasse. É geralmente escolhido quando se tem uma classe abstrata e não faz sentido ter uma tabela separada para essa classe. No exemplo utilizado, pode-se considerar desnecessário ter-se uma tabela separada para a classe abstrata ProdutoAbstrato. Para fazer o mapeamento deste tipo de herança, deve-se inserir o mesmo código do exemplo acima em cada uma das classes, exceto na que não se deseja reproduzir no banco, onde o código a ser inserido é:

```
"<inheritance strategy="subclass-table"/>"
```

Aplicando esse tipo de mapeamento para a herança entre Produto e ProdutoAbstrato, tem-se o formato no BD especificado na Figura 31.

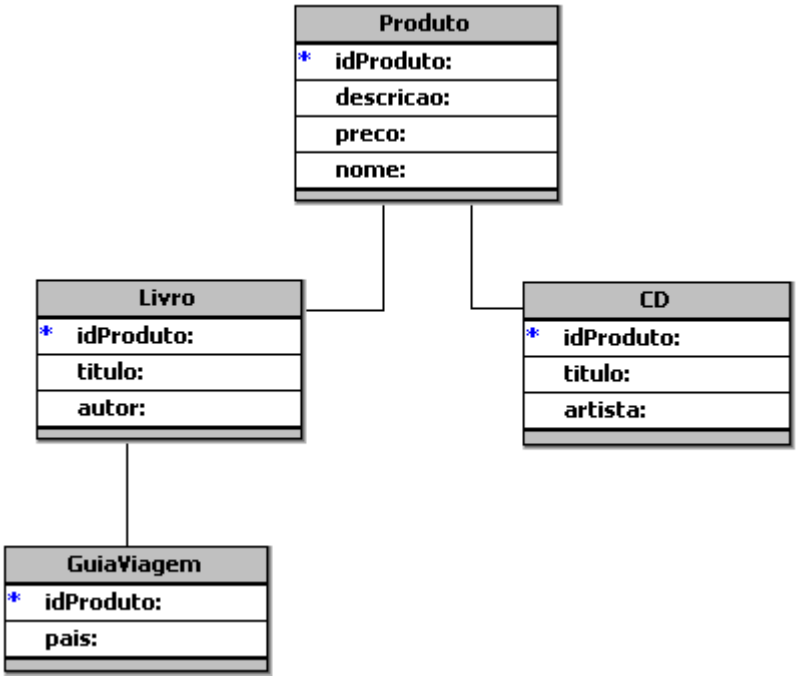

Figura 31 – Herança subclass-table [Dat09b]

- A terceira e última opção é a superclass-table. Nessa forma, é escolhida uma classe para ter seus campos persistidos pela tabela de sua superclasse. Sua vantagem é que a recuperação de um objeto é feita através de uma única chamada SQL a uma tabela. A desvantagem é que uma única tabela pode conter um número muito grande de colunas, e a leitura e o desempenho da base de dados podem sofrer, e é necessária uma coluna a mais, o *discriminator*. Para este exemplo, a tabela ProdutoAbstrato será ignorada e Produto será considerado a classe base. Se não há interesse em ter uma tabela Livro e uma tabela CD, então se terá uma tabela Produto onde todos os atributos serão armazenados. Então o código de descrição de produto ficará como na Figura 32, e o no mapeamento das tabelas que não irão ser criadas no BD deverá ser inserido o seguinte código:

```
<inheritance strategy=”superclass-table”/>
```

```
<class name="Produto">
    <inheritance strategy="new-table">
        <discriminator strategy="class-name">
            <column name="TIPO_PRODUTO"/>
        </discriminator>
    </inheritance>
    <field name="id" primary-key="true">
        <column name="IDPRODUTO"/>
    </field>
    <field name="preco">
        <column name="PRECO"/>
    </field>
</class>
```

Figura 32 – Mapeamento de Produto

Dessa forma, o resultado no BD ficará conforme a Figura 33.

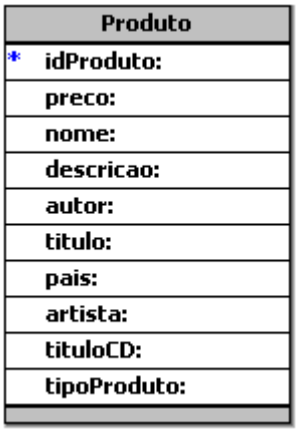

Figura 33 – Herança superclass-table [Dat09b]

O tipo de herança escolhido para o exemplo prático deste trabalho foi a estratégia *new-table*, onde haverá uma tabela para cada classe da hierarquia de herança. A Figura 34 mostra como ficou o mapeamento no arquivo de metadados.

```
<class
      name="Bolsista">

      <inheritance strategy="new-table"/>

      <field
          name="tipo"
          persistence-modifier="persistent"/>
      <field
          name="valor"
          persistence-modifier="persistent"/>
</class>
```

```
<class
      name="Pesquisador">

    <inheritance strategy="new-table"/>

    <datastore-identity>
        <column name="idPesquisador"/>
    </datastore-identity>

    <field
          name="nome"
          persistence-modifier="persistent"/>
    <field
          name="areaPesquisa"
          persistence-modifier="persistent"/>

    <field name="artigos">
        <collection
            element-type="entidades.Artigo"/>
        <join/>
    </field>

</class>
```

Figura 34 – Mapeamento de herança

3.2.4.3. Definir Atributo Composto e/ou Multivalorado

Em JDO é possível fazer o mapeamento de atributos multivalorados de duas formas: uma quando o atributo é simples, e outra quando é composto. Quando um atributo multivalorado se refere a uma coleção de valores primitivos, uma tabela adicional é necessária no BD. Essa tabela conterá o id da entidade e os valores do atributo. Um exemplo é dado na Figura 35.

```
class A {
        String ida;
        Collection strings;
}
```

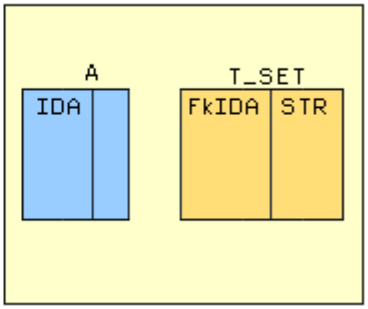

Figura 35 – Atributo Multivalorado [Spe09]

```
<class name="A" >
    <field name="ida" primary-key="true" column="IDA"/>
    <field name="strings" table="T_SET">
        <collection element-type="String"/>
        <element column="STR"/>
        <join column="FKIDA"/>
    </field>
</class>
```

Figura 36 – Mapeamento de atributo multivalorado [Spe09]

Como se pode ver na Figura 36, o conjunto de Strings é armazenado na tabela T_SET. A coluna correspondente ao elemento é SRT. A união desse elemento a essa tabela é baseada no identificador da classe. A coluna da chave estrangeira é a 'FKIDA'. Caso a classe seja formada por uma chave primária composta, então se deve declarar uma coluna para cada chave primária dentro do escopo do <join/>.

Caso o atributo multivalorado seja composto, o mapeamento ocorre de maneira semelhante, exceto pela definição do <element>, como se pode observar na Figura 37.

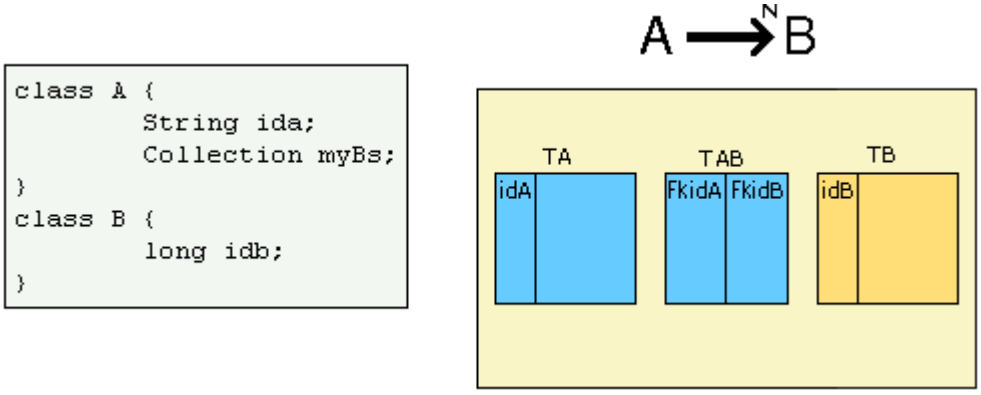

Figura 37 – Atributo multivalorado composto [Spe09]

Como se pode observar na Figura 38, o mapeamento do atributo composto multivalorado, os conjuntos de B são armazenados na tabela TAB. A coluna correspondente às referências das instâncias de B é FKIDB. Essa coluna referencia implicitamente a coluna de identificação da classe referida (B). A união com a tabela TAB é baseada no identificador da classe A. A coluna da chave estrangeira é FKIDA. Da mesma forma que a anterior, se A for formada por chave primária composta, então se incrementa <join> com as colunas referentes às chaves primárias.

```
<class name="A" >
    <field name="ida" primary-key="true" column="IDA"/>
    <field name="myBs" table="TAB">
        <collection element-type="B"/>
        <element column="FKIDB"/>
        <join column="FKIDA"/>
    </field>
</class>
<class name="B" >
    <field name="idb" primary-key="true" column="IDB"/>
</class>
```

Figura 38 – Mapeamento de atributo multivalorado composto [Spe09]

Pela implementação utilizada nesse exemplo prático não dar suporte a esse tipo de mapeamento, a relação entre Membro e Comissão, onde Membro é um atributo composto e multivalorado de Comissão, a relação foi construída como uma relação 1:N.

3.2.4.4. Definir Relacionamentos

Da mesma forma que em JPA, é possível obter relacionamentos de 3 tipos: um-para-um (one-to-one), muitos-para-um (many-to-one), muitos-para-muitos (many-to-many). Nas próximas 3 sub-seções serão apresentadas essas 3 formas de relacionamento.

3.2.4.4.1. Um-Para-Um

Para trabalhar com o mapeamento um-para-um, foi utilizado como exemplo o relacionamento entre Evento e Comissao. Ao final do mapeamento, obtêm-se a referência de Comissao em Evento, no esquema da base de dados.

A Figura 39 mostra o mapeamento das duas classes:

```
<class
      name="Evento">

  (...)

    <field
          name="comissao"
          persistence-modifier="persistent"/>

</class>
```

```
<class
      name="Comissao">

    (...)

    <field
          name="evento"
          persistence-modifier="persistent"
          mapped-by="comissao"/>
</class>
```

Figura 39 – Mapeamento de relacionamento um-para-um

Declara-se que há um campo comissao persistente em Evento. Na classe Comissao, é declarado que existe um campo evento, persistente, que é mapeado por um campo comissao, ou seja, o valor do campo comissao de um Evento está relacionado a uma comissao que contenha como identificador o valor deste campo, e o Evento atrelado a uma comissao é o que possui idComissao igual ao id do objeto comissao em questão.

Ao considerar Comissão uma entidade fraca, a forma de mapeamento sofre algumas alterações, como pode ser visto na Figura 40.

```
<class
      name="Evento">

      {...}

    <field
          name="comissao"
          persistence-modifier="persistent"
          mapped-by="evento"/>
</class>
```

```
<class
      name="Comissao"
      objectid-class="Comissao$PK"
      table="Comissao">

    <field name="evento"
          primary-key="true"
          column="IDEVENTO"/>

    {...}
</class>
```

Figura 40 - Mapeamento de entidades fracas em relacionamentos 1:1

Neste caso, além de definir um atributo do tipo de Comissao em Evento e um atributo do tipo Evento em Comissao, é preciso definir a chave primária de Comissao (object-id) na classe, que será igual a chave primária de Evento. A Figura 41 mostra a declaração da chave primária PK em Comissão. Declara-se uma classe estática que implementa a interface Serializable de Java. Nessa classe, define-se um atributo do tipo da classe de onde se obterá a chave primária de Comissao. Dessa forma, a chave primária de Comissão terá o mesmo valor da chave primária de Evento e Comissão só terá uma coluna, que será tanto a chave primária quanto a chave estrangeira.

```
public class Comissao {
    {...}
    public static class PK implements Serializable
    {
        public LongIdentity evento;

        public PK()
        {
        }

        public PK(String s)
        {
            StringTokenizer st = new StringTokenizer(s, "::");
            this.evento = new LongIdentity(Evento.class, st.nextToken());

        }

        public String toString()
        {
            return (evento.toString());
        }

        public int hashCode()
        {
            return evento.hashCode();
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return this.evento.equals(otherPK.evento);
            }
            return false;
        }
    }
    {...}
```

Figura 41 - Declaração da chave primária na entidade fraca

3.2.4.4.2. Muitos-Para-Um

Para este outro mapeamento, muitos-para-um, foi utilizado como exemplo o relacionamento entre Artigo e Evento. Neste caso, assim como com relação a JPA, a tabela Artigo conterá o id do Evento que está atrelado.

Pode ser observado na Figura 42 como fica o mapeamento em questão.

```
<class
      name="Artigo">

    (...)

    <fetch-group name="includingEvento">
    <field
          name="evento"
          persistence-modifier="persistent"/>
     </fetch-group>
</class>
```

```
<class
      name="Evento">

    (...)

    <field name="artigos"
          persistence-modifier="persistent"
          mapped-by="evento">
       <collection
            element-type="entidades.Artigo"/>
    </field>

</class>
```

Figura 42 – Mapeamento muitos-para-um

No mapeamento da classe Evento, especifica-se uma coleção de artigos persistente, que será mapeada pelo atributo evento. Também deve ser especificado o tipo da coleção, que no caso do exemplo, é do tipo *entidades.Artigo*. Já no mapeamento da classe Artigo, declara-se o atributo evento, que será a chave estrangeira da relação.

A *tag* <fecth-group> é utilizada para especificar que um atributo é instância de outro objeto que não o que está sendo mapeado. Ao definir essa *tag*, ela poderá ser utilizada ao carregar objetos nas consultas. Dessa forma, chamando este *fetch-group* "includingEvento", ao carregar objetos do tipo Artigo, o atributo evento irá ser carregado com o objeto Evento relacionado. Um exemplo de utilização poderá ser encontrado na seção 4 deste trabalho.

3.2.4.4.3. Muitos-Para-Muitos

No mapeamento muitos-para-muitos, foi usado como exemplo a relação entre Artigo e Pesquisador, onde um artigo pode ser escrito por vários pesquisadores e vários

pesquisadores podem escrever um mesmo artigo. No caso onde a relação não possui atributo, é definido um relacionamento M:N entre as duas entidades, e ao criar o esquema no BD, serão criada uma tabela contendo as chaves primárias das entidades da relação. Já no caso onde há atributos na relação, deverá ser criada uma classe extra, e serão definidos relacionamentos 1:N entre as entidades da relação e a entidade extra criada.Ao montar o esquema no BD, será criada a tabela da mesma forma que relacionamentos sem atributo, no entanto serão acrescentados os atributos da relação.

Pode-se analisar o primeiro caso de mapeamento de relacionamento M:N (sem atributo) na Figura 43.

```
<class
      name="Artigo">

    (...)

   <field name="pesquisadores"
          persistence-modifier="persistent"
          mapped-by="artigos"
          table="ESCREVE">
        <collection
              element-type="entidades.Pesquisador"/>
        <join>
            <column name="idArtigo"/>
        </join>
        <element>
            <column name="idPesquisador"/>
        </element>
    </field>

</class>
```

```
<class
      name="Pesquisador">

    <field name="artigos">
        <collection
              element-type="entidades.Artigo"/>
        <join/>
    </field>

</class>
```

Figura 43 – Mapeamento muitos-para-muitos sem atributos no relacionamento

Neste tipo de relacionamento, só é necessário declarar em apenas um dos mapeamentos de classe as informações da relação. Como mostrado na Figura 43, foi declarado na classe Arquivo uma coleção de pesquisadores, persistente, que é mapeada pelo atributo artigos, que também é uma coleção da classe Pesquisadores. O item *table* define o nome da tabela da relação, no caso "ESCREVE". O item *join* especifica o nome da coluna referente à classe onde estão sendo definidas as características do mapeamento, e no item *element*, especifica-se o nome da coluna referente ao outro lado do relacionamento.

A Figura 44 mostra como deve ficar o mapeamento de relacionamentos com atributos.

```
<class
      name="Pesquisador">
      {..}
    <field name="artigoRelacao"
           persistence-modifier="persistent"
           mapped-by="pesquisador">
        <collection
           element-type="entidades.ArtigoPesquisador"/>
    </field>

</class>
```

```
<class
      name="Artigo">
      {..}
     <field name="pesquisadorRelacao"
            persistence-modifier="persistent"
            mapped-by="artigo">
         <collection
            element-type="entidades.ArtigoPesquisador"/>
     </field>
</class>
```

Figura 44 – Mapeamento de Pesquisador e Artigo em relacionamento M:N com atributos

Assim como em mapeamento um-para-um com entidade fraca, também é preciso definir uma classe estática onde serão declaradas as chaves primárias da classe extra, ArtigoPesquisador. A Figura 45 mostra como ficará o mapeamento da nova classe ArtigoPesquisador e a Figura 46 mostra como deverá ser composta a nova classe. Como se pode observar, define-se os atributos dos tipos das entidades da relação, os atributos da relação e a chave primária que será composta pelas chaves primárias de artigo e de pesquisador.

```
<class name="ArtigoPesquisador"
       objectid-class="ArtigoPesquisador$PK"
       table="ArtigoPesquisador">
    <field name="artigo"
           primary-key="true"
           column="idArtigo"/>
    <field name="pesquisador"
           primary-key="true"
           column="idPesquisador"/>
    <field name="ordem"
           column="ordem"/>
</class>
```

Figura 45 - Mapeamento de ArtigoPesquisador

```
public class ArtigoPesquisador {

    private Artigo artigo;
    private Pesquisador pesquisador;

    private String ordem;

    public static class PK implements Serializable
    {
        public LongIdentity artigo;
        public LongIdentity pesquisador;

        public PK()
        {
        }

        public PK(String s)
        {
            StringTokenizer st = new StringTokenizer(s, "::");
            this.artigo = new LongIdentity(Artigo.class, st.nextToken());
            this.pesquisador = new LongIdentity(Pesquisador.class, st.nextToken());
        }

        public String toString()
        {
            return (artigo.toString() + "::" + pesquisador.toString());
        }

        public int hashCode()
        {
            return artigo.hashCode() ^ pesquisador.hashCode();
        }

        public boolean equals(Object other)
        {
            if (other != null && (other instanceof PK))
            {
                PK otherPK = (PK)other;
                return this.artigo.equals(otherPK.artigo)
                    && this.pesquisador.equals(otherPK.pesquisador);
            }
            return false;
        }
       {...}
```

Figura 46 - Classe ArtigoPesquisador

3.2.5. Definir Propriedades da Persistência

Para que a aplicação reconheça a base de dados a que irá se conectar, é preciso definir algumas propriedades. Para isso, pode-se declarar diretamente no código ou então fazer uso de um arquivo auxiliar, passando apenas como parâmetro o nome do arquivo, para que a aplicação carregue os dados. Dessa segunda forma, a aplicação se torna mais independente, quanto a não precisar alterar o código, apenas o arquivo de configuração.

A Figura 47 mostra o arquivo de configuração criado para a aplicação de exemplo.

```
javax.jdo.option.Mapping=mysql
javax.jdo.option.ConnectionDriverName=com.mysql.jdbc.Driver
javax.jdo.option.ConnectionURL=jdbc:mysql://localhost/monografiajdo
javax.jdo.option.ConnectionUserName=root
javax.jdo.option.ConnectionPassword=123456
```

Figura 47 – Arquivo de configuração da base de dados

3.2.6. Trabalhar a Persistência

Há 3 (três) interfaces essenciais na utilização do JDO:

- javax.jdo.PersistenceManagerFactory - constrói os objetos de acesso à estrutura de dados subjacente;
- javax.jdo.PersistenceManager - fornece o acesso à implementação JDO; e
- javax.jdo.Transaction - fornece o serviço de demarcação de transições.

A partir dessas classes é que se dá a persistência de fato dos objetos criados na aplicação. A classe PersistenceManagerFactory é obtida a partir do arquivo de configuração da base de dados definido na seção 3.2.5. Com sua criação, obtém-se a classe PersistenceManager, de onde se obtém a transação. A Figura 48 mostra como ficou a criação das classes.

```
public class EMUtil {
    private static PersistenceManagerFactory pmf;

    public static PersistenceManager getPersistenceManager() {
        if (pmf == null){
            pmf = JDOHelper.getPersistenceManagerFactory("datanucleus.properties");
        }
        PersistenceManager em = pmf.getPersistenceManager();
        return em;
    }

    public static void fechar(){
        pmf.close();
    }

}
```

Figura 48 – Criando classes para persistência

3.2.6.1. Inserção

O processo de inserção é simples. Cria-se um objeto e chama-se o método makePersistent, da classe PersistenceManager, dentro do escopo da transação, como pode ser observado na Figura 49.

```
public void cadastrar(Artigo artigo) {

    PersistenceManager persistenceManager = EMUtil.getPersistenceManager();
    Transaction tx = persistenceManager.currentTransaction();

    if (artigo == null) {
        return;
    }
    try{
        tx.begin();
        persistenceManager.makePersistent(artigo);
        tx.commit();
    } catch (Exception e){
        System.out.println("Erro ao cadastrar artigo");
    }

}
```

Figura 49 – Exemplo de inserção

3.2.6.2. Alteração

Para fazer atualização de objetos, uma forma de se realizar é buscar o objeto e fazer a atualização dentro da mesma transação, como mostrado na Figura 50.

```
public void atualizar(Object id) {

    PersistenceManager persistenceManager = EMUtil.getPersistenceManager();
    Transaction tx = persistenceManager.currentTransaction();
    try {
        tx.begin();
        Artigo artigo = (Artigo) persistenceManager.getObjectById(id);
        artigo.setTitulo("Novo Título");
        tx.commit();
    } catch (Exception e){
        System.out.println("Erro ao atualizar");
    }

}
```

Figura 50 – Exemplo de Atualização

3.2.6.3. Remoção

Para realizar a remoção de um objeto, deve-se buscar o objeto e depois invocar o método deletePersistent da classe PersistenceManager, passando o objeto como parâmetro, sendo toda a operação dentro do mesmo escopo da transação, como mostrado na Figura 51.

```
public void excluir(Object id) {

    PersistenceManager persistenceManager = EMUtil.getPersistenceManager();
    Transaction tx = persistenceManager.currentTransaction();

    try{
        tx.begin();
        Artigo artigo = (Artigo) persistenceManager.getObjectById(id);
        persistenceManager.deletePersistent(artigo);
        tx.commit();
    } catch (Exception e){
        System.out.println("Erro ao deletar");
    }
}
```

Figura 51 – Exemplo de remoção

3.2.6.4. Busca

Para realizar a busca de um objeto específico, basta chamar o método getObjectById, da classe PersistenceManager, passando como parâmetro o id do objeto, como exemplificado na Figura 52.

```
public Artigo buscar(Object id) {
    Artigo retorno=null;

    PersistenceManager persistenceManager = EMUtil.getPersistenceManager();
    Transaction tx = persistenceManager.currentTransaction();
    try {
        tx.begin();
        retorno = (Artigo)persistenceManager.getObjectById(id);
        tx.commit();
    } catch (Exception e){
        System.out.println("Erro ao buscar");
    }

    return retorno;

}
```

Figura 52 – Exemplo de busca

3.3. Conclusão

Este capítulo introduziu os principais conceitos sobre JDO, e em seguida mostrou como realizar ORM utilizando esta API. O exemplo prático foi divido em várias partes, sendo usado o esquema apresentado na seção 1.3.1 para exemplificar o mapeamento. Foi visto que JDO consegue abordar grande parte dos conceitos relacionais, e não apresentou problemas em atributos mutivalorados, diferentemente de JPA, como foi visto no capítulo 2. Ao final, foram apresentadas as 4 formas básicas de interação com o BD: inserção, atualização, remoção e busca, onde foi visto que não foi necessário o uso de linguagens de consulta.

O próximo capítulo apresentará uma análise comparativa entre as API JPA e JDO, mostrando suas semelhanças e diferenças e serão apresentadas algumas características de suas linguagens, JPQL [Sun09g] e JDOQL [Sun09h], respectivamente.

4. Análise Comparativa

Como pôde ser visto nas seções anteriores, JPA e JDO oferecem uma solução de padronização de ORM, e apresentam algumas diferenças e semelhanças, como será visto no decorrer desta seção.

- Tipo de Banco de Dados – JPA dá suporte apenas a BD relacionais, enquanto JDO dá suporte a qualquer tipo de BD;
- Restrições na construção da aplicação – tanto JPA quanto JDO requerem um construtor sem parâmetros nas classes a serem persistidas, porém JPA também não dá suporte a classes e métodos do tipo *final*;
- Herança – em JPA só é possível definir um tipo de estratégia de herança para uma hierarquia, sendo a classe raiz, o topo da hierarquia, que define a estratégia a ser utilizada. Já em JDO para cada entidade da hierarquia é possível definir uma estratégia diferente, já que a estratégia da herança é definida no mapeamento de cada classe da hierarquia;
- Tipos suportados – a Figura 53 mostra os tipos de Java que JDO e JPA dão suporte;

JDO	JPA
Java primitive types, wrappers of primitive types, java.lang.String, **java.lang.Number**, java.math.BigInteger, java.math.BigDecimal, **java.util.Currency, java.util.Locale**, java.util.Date, java.sql.Time, java.sql.Date, java.sql.Timestamp, java.io.Serializable, **boolean[]**, byte[], char[], **double[], float[], int[], long[], short[]**, **java.lang.Object**, **interface**, **Boolean[]**, Byte[], Character[], **Double[], Float[], Integer[], Long[], Short[], BigDecimal[], BigInteger[], String[], PersistenceCapable[], interface[], Object[]**, Enums, java.util.Collection, java.util.Set, java.util.List, java.util.Map, **Collection/List/Map of simple types, Collection/List/Map of reference (interface/Object) types**, Collection/List/Map of persistable types	Java primitive types, wrappers of the primitive types, java.lang.String, java.math.BigInteger, java.math.BigDecimal, java.util.Date, **java.util.Calendar**, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.Serializable, byte[], Byte[], char[], Character[], Enums, java.util.Collection, java.util.Set, java.util.List, java.util.Map Collection/List/Map of persistable types

Figura 53 - Tipos suportados por JDO e JPA [ASF09b]

- Linguagem – JPA possui a linguagem JPQL (Java Persistence Query Laguage) e JDO possui a linguagem JDOQL (Java Data Objects Query Language). Apesar de possuírem suas próprias linguagens de consulta, ambas API dão suporte a SQL também;
  - Case-sensitive – JPQL é case-sensitive. Já JDOQL só faz diferença entre letras todas minúsculas ou todas maiúsculas (lowercase/UPPERCASE);

- Operadores – os operadores de JPQL são:
  - Unario (+) (–);
  - Multiplicação (*) ;
  - Division (/);
  - Adição (+);
  - Subtração (–);
  - Relacionais (=) (>) (>=) (<) (<=) (<>), [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]; e
  - Operadores lógicos (AND, OR, NOT).

  Já os operadores de JDOQL são:
  - Unario (~) (!) (+) (-);
  - Multiplicação (*);
  - Divisão (/) (%);
  - Adição (+);
  - Subtração (-);
  - Relacionais (>=) (>) (<=) (<) (instanceof);
  - Igualdade (==) (=!);
  - Booleanos AND (&) OR (|); e
  - Condicionais AND ("&&") OR (||);

  Como se pode observar, os operadores de JDOQL se aproximam muito mais dos conceitos de objetos e de programação do que JPQL, pois possui operadores semelhantes aos operadores utilizados em Java, como '&&', '||', 'instanceof' e '=!', por exemplo. Já JPQL possui operadores semelhantes aos de SQL, sendo bem parecida com essa linguagem de consulta;

- Suporte a literais – JPQL não dá suporte à literais do tipo *Date*. Já JDOQL apesar da linguagem não dá suporte a alguns tipos de literais, ela é capaz de importar bibliotecas de Java, de forma que consegue trabalhar com quantidades maiores de tipos;

- Funções de agregação – Tanto JPQL quanto JDOQL dão suporte às funções MIN, MAX, COUNT, AVG e SUM. Todas essas funções são utilizadas por SQL;

- Subqueries – JPQL dá suporte aos operadores utilizados em subqueries ANY, ALL, SOME, EXISTS. Esses operadores são derivados da linguagem SQL. Já JDOQL utiliza funções específicas da linguagem, *contains* e *isEmpty*;

- Aliases – JPQL dá suporte a alias e é obrigatório seu uso. Já JDOQL não dá suporte a alias, mas utiliza variáveis;

- Procedures – nenhuma das linguagens suporta procedures; e

- Estrutura das consultas – as estruturas das consultas de JPQL e JDOQL podem ser vistas na Figura 54.

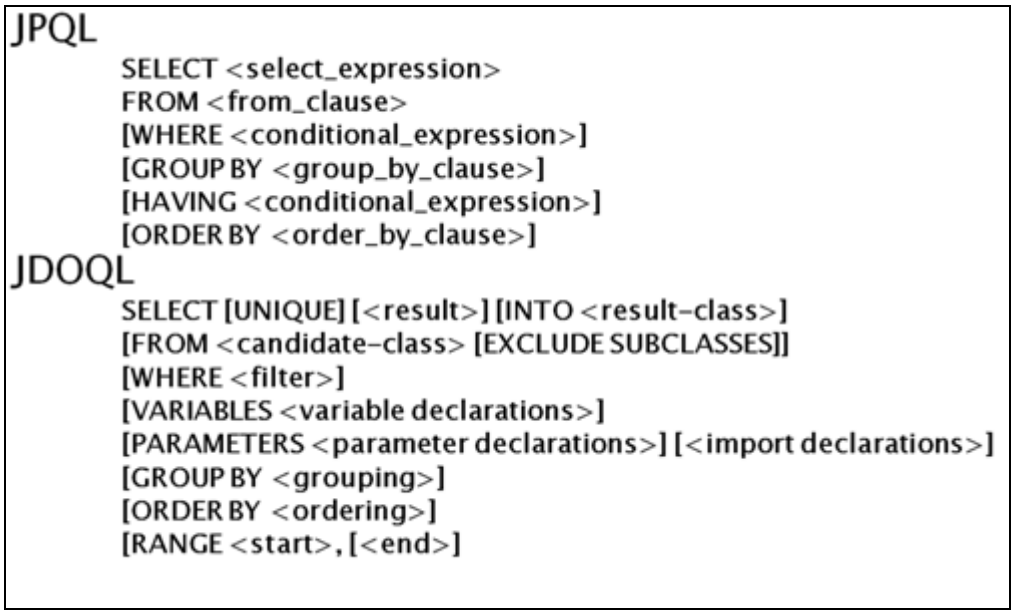

Figura 54 - Estrutura das consultas de JPQL e JDOQL [Dat09c] [KS06]

4.1. Consultas

- **SELECT * FROM PESQUISADOR**

Em JDO não é necessário trabalhar com nenhuma linguagem de acesso para essa consulta, e o retorno são objetos do tipo Pesquisador, sem a necessidade de converter manualmente o resultado em objeto, como pode ser visto na Figura 55.

```
Query q = persistenceManager.newQuery(entidades.Pesquisador.class);
retorno = (List<Pesquisador>)q.execute();
```

Figura 55 – Consulta 1 com JDO

Em JPA já é necessário o uso de linguagem de acesso, mas também já retorna um objeto completo, sem a necessidade da transformação manual, que no caso da Figura 56 seria uma lista de Pesquisadores.

```
Query q = entityManager.createQuery ("SELECT x FROM Pesquisador x");
retorno = q.getResultList ();
```

Figura 56 – Consulta 1 com JPA

- **SELECT * FROM Artigo WHERE nota BETWEEN 7 AND 9;**

JDOQL não dá suporte a cláusula BETWEEN. Assim, a forma encontrada de realizar a consulta foi utilizando a linguagem SQL. No entanto, o retorno deixa de ser um objeto específico, no caso Artigo, para ser um Object de Java. Sendo assim, é necessário fazer o mapeamento do resultado para o objeto desejado. Para utilizar SQL, o primeiro parâmetro da função newQuery é "*javax.jdo.query.SQL*", que especifica que a linguagem de consulta utilizada será SQL. A Figura 57 mostra como realizar a consulta em uma aplicação que utiliza JDO.

Outra forma de realizar a consulta utilizando JDOQL neste caso seria a utilização dos operadores "<=" e ">=", que retornaria objetos do tipo Artigo.

```
Query q = persistenceManager.newQuery("javax.jdo.query.SQL",
        "SELECT * FROM Artigo WHERE nota between ?1 AND ?2");
retorno = (List)q.execute(9.0f,10.0f);
```

Figura 57 – Consulta 2 com JDO

JPA dá suporte à cláusula BETWEEN, e mantém o mapeamento de dados implícito na aplicação, sendo o retorno objetos do tipo Artigo, como pode-se observar na Figura 58.

```
Query q = entityManager.createQuery ("SELECT x FROM Artigo x " +
                                     "WHERE x.nota between ?1 and ?2");
q.setParameter (1, 7.0f).setParameter (2, 9.0f);
retorno = q.getResultList ();
```

Figura 58 – Consulta 2 com JPA

- **SELECT a.titulo, e.sigla FROM Artigo a, Evento e WHERE a.idEvento = e.idEvento**

Para esse tipo de consulta, é possível configurar o JDO para que os objetos sejam carregados com seus atributos primitivos, e também seus atributos que forem instâncias de outras classes. Para isso, é preciso adicionar a segunda linha da Figura 59 onde o parâmetro será o nome definido no arquivo de metadados, como se pode ver na Figura 60. Dessa forma, o resultado da consulta será todos os Artigos onde idEvento não é *null*, ou seja, possui um Evento a que o Artigo se relaciona, e dentro de Artigo virá o objeto evento carregado.

Para obter como resultado objetos do tipo Evento carregado com os Artigos que se relacionam com ele, basta trocar Artigo por Evento e adicionar o fetch-group no mapeamento em Evento, no atributo artigos.

```
PersistenceManager persistenceManager = EMUtil.getPersistenceManager();
persistenceManager.getFetchPlan().addGroup("includingEvento");
Transaction tx = persistenceManager.currentTransaction();
tx.setNontransactionalRead(true);
try {
    tx.begin();

    Query q = persistenceManager.newQuery(entidades.Artigo.class, "evento != null");
    retorno = (List<Artigo>)persistenceManager.detachCopyAll((List<Artigo>)q.execute());
    {...}
```

Figura 59 – Consulta 3 com JDO

```
<class
      name="Artigo">

      {...}

      <fetch-group name="includingEvento">
          <field
              name="evento"
              persistence-modifier="persistent"/>
      </fetch-group>
</class>
```

Figura 60 – Mapeamento de atributos

Em JPA, pode-se seguir o mesmo raciocínio de JDO, mas também é possível trazer os objetos Evento e Artigo separados. No segundo caso usa-se a consulta na Figura 61, e o resultado são arrays de Objects de tamanho 2 cada, onde o primeiro elemento do array é um objeto do tipo Artigo e o segundo é um Evento.

```
Query q = entityManager.createQuery ("SELECT x, y FROM Artigo x, Evento y " +
                                     "WHERE x.evento=y.id");
retorno = q.getResultList ();
```

Figura 61 – Consulta 3 com JPA – Exemplo 1

Seguindo o raciocínio do exemplo de JDO, também se pode buscar todos os Artigos com os Eventos carregados nos objetos retornados. A diferença entre JPA e JDO é que, em JPA não é necessário, a configuração do "fetch-group". O que se tem em JPA é o "fetchtype", que apenas indica se o carregamento dos atributos que são instância de outros objetos será feito sempre ao carregar o "objeto mãe" (EAGER) ou apenas quando ele for acessado (LAZY), que ocorre de forma transparente ao desenvolvedor e ao usuário. Desta forma, a consulta ficaria conforme a Figura 62. Mas, além do operador "!=", JPQL também dá suporte ao operador "is/is not", que não tem suporte em JDOQL.

```
Query q = entityManager.createQuery ("SELECT x FROM Artigo x " +
                                     "WHERE x.evento != null");
retorno = q.getResultList ();
```

Figura 62 – Consulta 3 com JPA – Exemplo 2

- **SELECT p.nome, a.titulo FROM Pesquisador p INNER JOIN Escreve e ON p.idPesquisador = e.idPesquisador INNER JOIN Artigo a ON a.idArtigo = e.idArtigo WHERE a.nota>9.6;**

Realizando essa consulta em JDOQL, o resultado será um objeto do tipo Pesquisador, que terá um conjunto de Artigos que possuem nota superior a 9.6. Neste caso, deve ser criada uma query com a entidade Pesquisador, declarada uma variável someArtigo do tipo Artigo e utilizado o método contains de JDOQL, que busca dentro da coleção de artigos se existe algum Artigo e que esse artigo tenha nota superior a 9.6. Como se pode observar na Figura 63, a consulta se aproxima muito do pensamento de objetos.

```
Query q = persistenceManager.newQuery(entidades.Pesquisador.class);
q.declareVariables("Artigo someArtigo");
q.setFilter("artigos.contains(someArtigo) && someArtigo.nota > 9.6");
retorno = (List<Pesquisador>)q.execute();
```

Figura 63 - Consulta 4 com JDO

Há dois tipos de retorno para essa consulta: um objeto Pesquisador ou um objeto Pesquisador e um objeto Artigo. Basta acrescentar depois da cláusula SELECT um 'y', representando Artigo. Como se pode observar na Figura 64, apesar da consulta parecer com a de SQL, não é necessário adicionar a expressão de comparação entre os id de Pesquisador da tabela Pesquisador e da tabela Artigo. A expressão se torna mais simples.

```
Query q = entityManager.createQuery ("SELECT x FROM Pesquisador x " +
        "INNER JOIN x.artigos y where y.nota > 9.6");
retorno = q.getResultList ();
```

Figura 64 - Consulta 4 com JPA

4.2. Considerações

Como pôde ser visto na seção anterior, a estrutura de JPQL se assemelha bastante com a estrutura de SQL, porém a primeira trabalha com objetos enquanto a segunda trabalha com tabelas. Já JDOQL apresenta uma estrutura mais voltada à orientação a objetos, tanto por trabalhar com objetos quanto por possuir métodos como *isEmpty( )*, por exemplo, e conseguir importar algumas bibliotecas.

O uso das linguagens facilita o trabalho do desenvolvedor por trabalharem com objetos e retornar já objetos prontos, sem a necessidade de quebrar um objeto em atributos para montar consultas ou ter que reconstruí-lo ao obter o retorno de consultas.

JPQL possui muitos operadores semelhantes à SQL, enquanto JDOQL possui alguns operadores utilizados em programação, como (&&) ou (||), o que reforça a idéia de JDOQL se aproximar mais ao conceito de trabalhar com objetos.

JPA, talvez por ser mais recente, ainda se encontra em desvantagem, como no mapeamento de herança por exemplo, porém pode-se dizer que JPA e JDO se assemelham quanto a gama de conceitos relacionais que conseguem atender e dão suporte a muitas propriedades do ORM. Por fim, no próximo capítulo será apresentada a conclusão a respeito do trabalho desenvolvido.

5. Conclusão

Como pode ser visto no decorrer do trabalho, as duas API oferecem uma grande facilidade no desenvolvimento de aplicações de persistência de dados, devido a sua padronização e transparência nas operações. Grande parte do trabalho necessário antes do surgimento delas foi retirado do desenvolvedor e vem sendo realizado pela implementação das API. E, além disso, a portabilidade das aplicações ficou maior. Trouxe uma maior liberdade para mudanças.

Não é obrigatório que a quantidade de arquivos seja menor. O que a padronização do ORM estabelece é uma forma de trabalhar com objetos, o que beneficia o desenvolvedor, e também a portabilidade da aplicação, por não se limitar à implementação escolhida para realizar a persistência.

Outro fator positivo para a utilização dessas interfaces é que apesar de não ser necessário o uso de linguagens de acesso a dados como SQL, essas API dão suporte a esse tipo de linguagem de forma a não limitar o poder de manuseio dos dados. Ao mesmo tempo em que trás uma abstração, em alguns casos até excluindo totalmente o uso de linguagens de BD, ele também possibilita seu uso.

Essas API podem ser consideradas concorrentes, mas já possuem aplicações que as utilizam juntas, de forma que elas se complementam. E as duas juntas formam uma ferramenta poderosa que poderá dominar o mercado de desenvolvimento, e quem sabe definir o padrão definitivo da persistência de dados em Java.

5.1. Contribuições

- Apresentação de duas API para padronização de persistência em Java, com recomendações de utilizações, que poderão auxiliar futuras implementações de aplicações com seus usos; e
- Apresentação de possibilidades de união entre das duas API, de forma que não apenas concorrem, mas também podem se complementar.

5.2. Limitações

Algumas limitações são encontradas nas API estudadas.

- JPA dá suporte apenas a BD relacionais (RDBMS).
- Por se tratarem de API, algumas funcionalidades previstas no desenvolvimento de JPA e JDO, desenvolvedores dependem das implementações, que nem sempre provêm todas as funcionalidades previstas nas API;
- As linguagens das API não dão suporte a *procedures*;

5.3. Trabalhos Futuros

Alguns trabalhos futuros a partir deste documento podem ser:

- Desenvolvimento de uma nova implementação dessas API, utilizando-as de forma complementar;
- Estudo sobre as mudanças na nova versão de JPA, 2.0;
- Estudo comparativo entre os padrões estabelecidos por JPA e/ou JDO e outras ferramentas de mapeamento; e
- Desenvolvimento de uma ferramenta de mapeamento OR que gere a partir de um modelo lógico as classes Java já mapeadas com *annotations*.

Referências Bibliográficas

[Amb97] S. W. Ambler. Mapping Objects To Relational Databases. AmbSoftInc., 1997. Acessado em 11/12/2009.
Disponível em: {http://jeffsutherland.com/objwld98/mappingobjects.pdf}

[Apa09a] OpenJPA. Acessado em: 11/12/2009. Disponível em{http://openjpa.apache.org/}.

[Apa09b] JDOHelper. Acessado em: 11/12/2009. Disponível em {https://svn.apache.org/repos/asf/db/jdo/site/docs/api20/apidocs/javax/jdo/JDOHelper.html}.

[ASF09a] Apache Software Foundation. JDO Implementations. Acessado em: 9/12/2009. Disponível em {http://db.apache.org/jdo/impls.html}

[ASF09b]ApacheSoftware Foudantion. Wich Persistence Specification? Acessado em 07/12/2009. Disponível em {http://db.apache.org/jdo/jdo_v_jpa.html}.

[BK07] C. Bauer e G. King. "Java Persistence with Hibernate". Manning, 2007.

[BL06] B. Leonard. "Using Composite Keys with JPA". java.net, 2006. Acessado em: 07/12/2009. Disponível em {http://weblogs.java.net/blog/bleonard/archive/2006/11/using_composite.html}.

[BO06] R. Biswas e E. Ort. "The Java Persistence API – A Simpler Programming Model for Entity Persistence", Sun Developer Network, 2006. Acessado em 24/11/2009.
Disponível em: {http://java.sun.com/developer/technicalArticles/J2EE/jpa/}.

[BRM09] BRModelo. Acessado em: 07/12/2009. Disponível em: {http://sis4.com/brModelo/}

[Dat09a] DataNucleus and Eclipse. Acessado em 11/12/2009. Disponível em {http://www.datanucleus.org/products/accessplatform_1_1/guides/eclipse/index.html}.

[Dat09b] Data Nucleus - JDO Inheritance Strategies. Acessado em 11/12/2009. Disponível em {http://www.datanucleus.org/products/accessplatform_1_1/guides/eclipse/index.html}.

[Dat09c] Data Nucleus - JDO Queries. Acessado em 11/12/2009. Disponível em: {http://www.datanucleus.org/products/accessplatform_1_1/jdo/jdoql.html}

[Ecl09a] EclipseLink. Acessado em: 11/12/2009. Disponível em {http://www.eclipse.org/eclipselink/}.

[Ecl09b] EclipseLink. Acessado em: 11/12/2009. Disponível em {http://www.eclipse.org/}.

[Gla09] Glassfish - Toplink Essentials. Acessado em: 11/12/2009. Disponível em {https://glassfish.dev.java.net/javaee5/persistence/}.

[Han05] J. Hanson. “An Intro to Java Object Persistence with JDO”. Devx.com,2005. Acessado em: 29/11/2009. Disponível em {http://www.devx.com/Java/Article/26703/176/HTML/1}.

[HGR09] S. Heider, A. Görler e J. Reisbich. “Getting Started with Java Persistence API and SAP JPA 1.0”, **SAP AG**, 2009, versão atualizada.

[Hib09a] Hibernate. Acessado em: 11/12/2009. Disponível em{https://www.hibernate.org/}.

[Hib09b] Java Persistence with Hibernate. Acessado em: 11/12/2009. Disponível em {https://www.hibernate.org/397.html}.

[Jon09] B. L. Jones. Data Persistence and Java Data Objects – JDO. developer.com, 2001. Acessado em: 11/12/2009. Disponível em {http://www.developer.com/java/data/article.php/918111/Data-Persistence-and-Java-Data-Objects----JDO.htm}

[JPA09] JPA Specification. Acessado em: 9/12/2009.
Disponível em {http://jcp.org/aboutJava/communityprocess/final/jsr220/}

[Jpo09] JPox – JPA Inheritance Strategies. JPox.com. Acessado em: 29/11/2009.
Disponível em {http://www.jpox.org/docs/1_2/jpa_orm/inheritance.html}.

[KS06] M. Keith e M. Schincariol. “Pro EJB 3 – Java Persistence API”. Apress, 2006.

[Mah05] Q. H. Mahmoud. “Getting Started With Java Data Objects (JDO): A Standard Mechanism for Persisting Plain Java Technology Objects”. Sun Developer Network, 2005. Acessado em 29/11/2009. Disponível em:
{http://java.sun.com/developer/technicalArticles/J2SE/jdo/}.

[Mel09] R. S. Mello. Revisão e Dicas de Projeto Conceitual – Modelo ER. Acessado em: 11/12/2009. Disponível em
{http://www.inf.ufsc.br/~ronaldo/ine5623/2-RevisaoDicasModelagemConceitual.pdf}.

[MS09a] W. M. Sacramento. “Introdução à Java Persistence API – JPA”. Acessado em 24/11/2009.
Disponível em: {http://devmedia.com.br/articles/viewcomp.asp?comp=4590}.

[MS09b] D. M. Silva. “JPA – Hibernate”. PUC – RJ. Acessado em: 29/11/2009.
Disponível em: {http://wiki.les.inf.puc-rio.br/uploads/2/28/JPA_-_Hibernate.pdf}.

[Mys09] MySQL. Acessado em: 11/12/2009. Disponível em {http://www.mysql.com/}.

[Oco07] J. O’Conner. “Using Java Persistence API in Desktop Applications”. Sun Developer Network, 2007. Acessado em 24/11/2009. Disponível em:
{http://java.sun.com/developer/technicalArticles/J2SE/Desktop/persistenceapi/}.

[Ora09a] Toplink. Acessado em: 11/12/2009. Disponível em {http://www.oracle.com/technology/products/ias/toplink/index.html}.

[Ora09b] Kodo. Acessado em: 11/12/2009. Disponível em {http://download-llnw.oracle.com/docs/cd/E13189_01/kodo/docs41/}.

[Ric02] I.L. M. Ricarte. Bancos de Dados Relacionais. Unicamp, 2002. Acessado em 11/12/2009. Disponível em {http://www.dca.fee.unicamp.br/cursos/PooJava/javadb/bdrel.html}.

[Sil08] D. M. Silva. "JPA – Hibernate". PUC, 2008. Acessado em 24/11/2009. Disponível em: {http://wiki.les.inf.puc-rio.br/uploads/5/50/PRDS2008.1_Modulo_6.2.pdf }.

[Sil09] D.S. Silva. Camada de Persistência de Dados: DAO e ActiveRecord. Acessado em: 9/12/2009.
Disponível em {http://manifesto.blog.br/1.5/Blog/Programacao/dao-active-record.html}.

[Spe09] Speedo user manual: Edition of .jdo file(Mapping) . Acessado em: 09/11/2009.
Disponível em {http://speedo.ow2.org/doc/dev_jdo2file.html}

[Sou09] Ebeam. Acessado em: 11/12/2009.
Disponível em {http://sourceforge.net/projects/ebeanorm/}

[Sun09a] Sun Microsystems. "*Annotations* ". Acessado em :30/11/2009.
Disponível em {http://java.sun.com/j2se/1.5.0/docs/guide/language/*annotations* .html}.

[Sun09b] Sun Microsystems. Java Persistence API. Acessado em: 9/12/2009.
Disponível em {http://java.sun.com/javaee/technologies/persistence.jsp}.

[Sun09c] Sun Microsystems. Java Data Objects. Acessado em: 9/12/2009.
Disponível em {http://java.sun.com/jdo/}.

[Sun09d] Sun Microsystems. Java SE 1.5.0. Acessado em: 9/12/2009.
Disponível em {http://java.sun.com/j2se/1.5.0/}.

[Sun09e] Sun Microsystems. Enterprise JavaBeans Technology. Acessado em: 9/12/2009.
Disponível em {http://java.sun.com/products/ejb/}.

[Sun09f] Sun Microsystems. The Source of Java Developers. Acessado em: 11/12/2009.
Disponível em {http://java.sun.com/}.

[Sun09g] Sun Microsystems. The Java Persistence Query Language. Acessado em 11/12/2009.
Disponível em
{http://java.sun.com/javaee/5/docs/tutorial/backup/update3/doc/QueryLanguage.html}

[Sun09h] Sun Microsystems. About JDOQL Queries. Acessado em: 11/12/2009.

Disponível em {http://docs.sun.com/app/docs/doc/819-4721/beakd?a=view}

[Sun09i] Sun Microsystems. J2SE 5.0. Acessado em: 11/12/2009. Disponível em {http://java.sun.com/j2se/1.5.0/}.

[Uni09] Banco de Dados Básico. Unicamp, SP. Acessado em: 11/12/2009. Disponível em {ftp://ftp.unicamp.br/pub/apoio/treinamentos/bancodados/cursodb.pdf}.

[Web09a] Using and Understanding Java Data Objects. Acessado em: 9/12/2009. Disponível em {http://book.javanb.com/using-and-understanding-java-data-objects/LiB0025.html}.

[Web09b] Java Data Objects (JDO) Specification. Acessado em 9/12/2009. Disponível em{http://jcp.org/en/jsr/detail?id=12}.

[WIKI09a] "Persistência de Dados". Acessado em 30/11/2009. Disponível em {http://pt.wikipedia.org/wiki/Persist%C3%AAncia_de_dados}.

[WIKI09b]"API". Acessado em 30/11/2009. Disponível em {http://pt.wikipedia.org/wiki/API}.

[WIKI09c] Mapeamento Objeto-Relacional. Acessado em: 9/12/2009. Disponível em {http://pt.wikipedia.org/wiki/Mapeamento_objeto-relacional}.

[WIKI09d] Object-relational Mapping. Acessado em: 9/12/2009. Disponível em {http://en.wikipedia.org/wiki/Object-relational_mapping}.

[WIKI09e] POJO. Acessado em: 9/12/2009. Disponível em {http://en.wikipedia.org/wiki/Plain_Old_Java_Object}.

[Web09c] Java Community Process. Acessado em: 9/12/2009. Disponível em {http://jcp.org/en/home/index}.

[XML09] Extensible Markup Language (XML). Acessado em: 11/12/2009. Disponível em: {http://www.w3.org/XML/}.