Hibernate com XML

Por: Raphaela Galhardo Fernandes
Gleydson de A. Ferreira Lima

_____

raphaela@j2eebrasil.com.br,
gleydson@j2eebrasil.com.br

JavaRN - http://javarn.dev.java.net
J2EEBrasil - http://www.j2eebrasil.com.br

Natal, Fevereiro de 2006

Sumário

1. Conceitos Gerais de Persistência

Imagine um usuário fazendo uso de uma aplicação, por exemplo, um sistema para controlar suas finanças. Ele passa a fornecer como dados de entrada todos os seus gastos mensais para que a aplicação lhe gere, por exemplo, gráficos nos quais ele possa avaliar seus gastos. Finalizada, a sua análise financeira, o usuário resolve desligar o computador em que a aplicação se encontra. Imagine agora que o usuário teve novos gastos, voltou a ligar o computador, acessou novamente o sistema de finanças e que gostaria de realizar novas análises com todos os seus gastos acumulados. Se a aplicação não armazenar, de alguma forma, os primeiros gastos fornecidos, o usuário teria que informá-los novamente e em seguida, acrescentar os novos gastos para fazer a nova análise, causando grande trabalho e o sistema não sendo eficiente. Para resolver esse tipo de problema, uma solução seria armazenar (persistir) os dados lançados a cada vez pelo usuário em um banco de dados relacional, utilizando SQL (*Structured Query Language*).

2. Mapeamento Objeto Relacional

Seção Escrita Por: Juliano Rafael Sena de Araújo

Por vários anos os projetos de aplicações corporativas tiveram uma forte necessidade de se otimizar a comunicação da lógica de negócio com a base de dados. Essa necessidade ganhou mais intensidade com o crescimento dessas aplicações, crescimento esse, tanto em requisitos (funcionalidade) quanto em volume de dados armazenados em seu banco de dados.

Na década de 80, foram criados os bancos de dados relacionais (BDR) que substituíram as bases de dados de arquivos. Para esse tipo de base de dados foi criada uma linguagem, a SQL. Essa linguagem foi toda baseada na lógica relacional e por isso contava com diversas otimizações em suas tarefas se comparadas com as outras tecnologias existentes. A partir de então foi diminuído o tempo gasto para as operações de persistência, mesmo com um grande volume de dados. Entretanto, essa linguagem não propiciava aos desenvolvedores uma facilidade para que a produtividade fosse aumentada.

Uma solução que surgiu no início da década de 90 foi à criação de um modelo de banco de dados baseado no conceito de orientação a objetos. Este modelo visava facilitar, para os desenvolvedores, a implementação da camada de persistência da aplicação, pois eles já estavam familiarizados com o paradigma de orientação a objetos, consequentemente, a produtividade certamente aumentaria.

Na prática, esse modelo de dados não foi utilizado em grandes aplicações, visto que elas tinham um volume de dados muito grande e esse modelo era ineficiente em termos de tempo de resposta, pois ao contrário dos bancos de dados relacionais, eles não tinham um modelo matemático que facilitasse as suas operações de persistências.

Então a solução foi usar os BDR e desenvolver ferramentas para que o seu uso seja facilitado. Uma dessas ferramentas é o *framework Hibernate* que usa o conceito de mapeamento objeto relacional (MOR).

As subseções seguintes apresentam os conceitos relacionados ao mapeamento objeto relacional.

2.1 Mapeamento objeto relacional

Como foi descrito anteriormente, mapeamento objeto relacional funciona com a transformação dos dados de um objeto em uma linha de uma tabela de um banco de dados, ou de forma inversa, com a transformação de uma linha da tabela em um objeto da aplicação. Abordando essa idéia, alguns problemas poderão existir, como, se um objeto tiver uma coleção de outros objetos. Nas próximas subseções serão abordados os mapeamentos básicos e mais utilizados.

2.1.1 Mapeamento de um objeto com tipos primitivos

Esse é o mapeamento mais simples, onde um objeto tem apenas tipos de dados básicos. Vale salientar que entendesse por tipos básicos aqueles que possuem um correspondente em SQL, ou seja, o tipo String da linguagem Java é considerado um tipo básico, pois ele possui um correspondente em SQL.

Na Figura 1, observa-se o mapeamento de três objetos do tipo Veiculo na tabela de um banco de dados. Caso a aplicação deseje saber o veículo da cor vermelha, por exemplo, então o objeto que tem o Pálio como modelo é retornado.

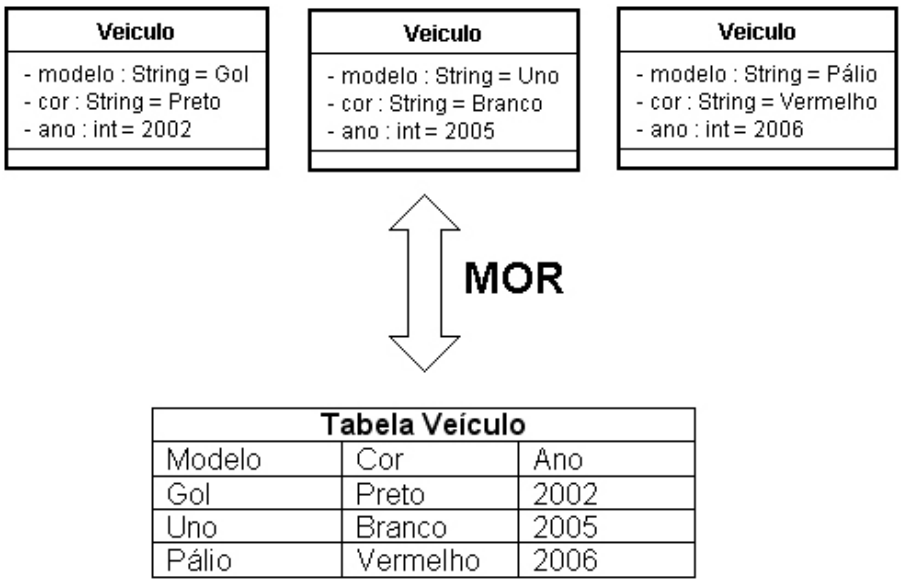

Figura 1 - Exemplo de mapeamento de tipos básicos

2.1.2 Mapeamento de objetos que contém uma coleção de objetos

Esse tipo de mapeamento é quando um objeto possui um conjunto de outros objetos. Para obter esse conceito é necessário adicionar, ao exemplo da Figura 1, uma nova classe chamada de `Fabricante` que conterá as informações: nome, que armazenará o nome desse fabricante e o `veiculos`, que conterá o conjunto de veículos do fabricante (`telefone` e `endereco` não são informações relevantes no exemplo). Faz-se necessário a adição de uma informação na classe `Veiculo` chamada de `fabricante`, como mostrado na Figura 2.

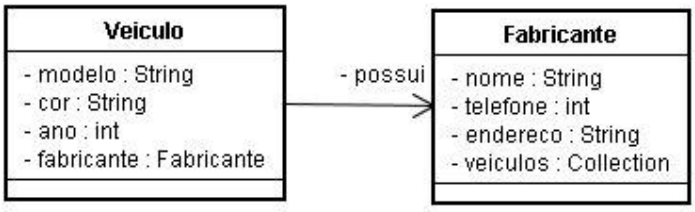

Figura 2 - Relacionamento entre as classes Fabricante e Veiculo

O atributo `fabricante` adicionado em `Veiculo` é simplesmente para relacionar um veículo ao seu fabricante, enquanto o `veiculos` de `Fabricante` referencia a classe `Veiculo`.

Como foram realizadas mudanças no domínio do exemplo orientado a objetos, faz-se necessária uma alteração no modelo do banco de dados. Primeiramente, o vínculo que foi realizado entre um fabricante e um veículo deverá

ser implementado através de uma chave estrangeira (referência a uma outra tabela, pois em BDR não existe o conceito de coleções) que se localizará na tabela `VEICULO`, caracterizando o mapeamento 1 para N, ou seja, um veículo possui apenas um fabricante e um fabricante possui vários (N) veículos. Essa chave estrangeira será realizada entre a informação nome da classe `Fabricante` e o atributo `fabricante` da classe `Veiculo`.

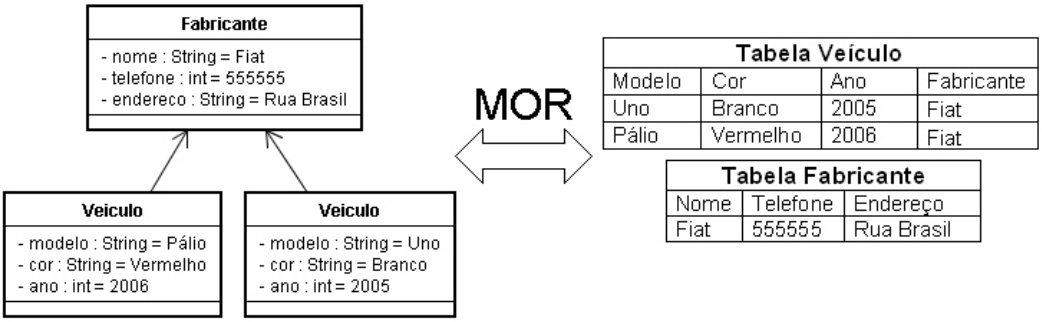

Figura 3 – Exemplo de mapeamento 1 para N

Para um melhor entendimento, a Figura 3 mostra o mapeamento dos objetos para as tabelas no BDR. Nele observa-se que cada veículo está associado com um fabricante, isso implica dizer que na tabela de `VEICULO` existe uma referência para a tabela `FABRICANTE`. Essa associação é feita através da coluna `fabricante` da tabela `VEICULO`. Pensando em termos práticos, quando um veículo for inserido no banco de dados, ele será ligado ao seu respectivo fabricante, no momento da inserção.

Para recuperar os veículos inseridos o desenvolvedor terá que implementar uma busca que retorne além das informações do veículo as informações do seu fabricante, isso poderá ser realizado fazendo a seguinte consulta SQL: `SELECT * FROM fabricante WHERE nome = <fabricante>`, onde `<fabricante>` é o nome do fabricante que está na tabela `VEICULO`, por exemplo, caso a aplicação deseje as informações do veículo do modelo Pálio, então para se buscar o fabricante a seguinte consulta será realizada: `SELECT * FROM fabricante WHERE nome = 'Fiat'`.

Utilizando a mesma idéia, os fabricantes são buscados; a única diferença é que agora a consulta que buscará os veículos associados com o fabricante retornará uma coleção de veículos (`SELECT * FROM veiculo WHERE fabricante = <nomeFabricante>`).

Esse tipo de mapeamento é muito utilizado em aplicações em que os objetos

fazem muita referência a outros objetos.

Com a idéia do MOR vários projetos de ferramenta começaram a ser desenvolvidas, visando facilitar a implementação da camada de persistência, dentre elas o *framework Hibernate*, que um software livre de código aberto e que está tendo uma forte adesão de novos projetos corporativos. Essa ferramenta será descrita na próxima seção.

3. Introdução ao Hibernate

O *Hibernate* é um *framework* de mapeamento objeto relacional para aplicações Java, ou seja, é uma ferramenta para mapear classes Java em tabelas do banco de dados e vice-versa. É bastante poderoso e dá suporte ao mapeamento de associações entre objetos, herança, polimorfismo, composição e coleções.

O *Hibernate* não apresenta apenas a função de realizar o mapeamento objeto relacional. Também disponibiliza um poderoso mecanismo de consulta de dados, permitindo uma redução considerável no tempo de desenvolvimento da aplicação.

4. Arquitetura

A arquitetura do *Hibernate* é formada basicamente por um conjunto de interfaces. A Figura 4 apresenta as interfaces mais importantes nas camadas de negócio e persistência. A camada de negócio aparece acima da camada de persistência por atuar como uma cliente da camada de persistência. Vale salientar que algumas aplicações podem não ter a separação clara entre as camadas de negócio e de persistência.

De acordo com a Figura 4, as interfaces são classificadas como:

- Interfaces responsáveis por executar operações de criação, deleção, consulta e atualização no banco de dados: Session, Transaction e Query;
- Interface utilizada pela aplicação para configurar o *Hibernate*: Configuration;
- Interfaces responsáveis por realizar a interação entre os eventos do *Hibernate* e a aplicação: `Interceptor`, `Lifecycle` e `Validatable`.
- Interfaces que permitem a extensão das funcionalidades de mapeamento do *Hibernate*: `UserType`, `CompositeUserType`, `IdentifierGenerator`.

O *Hibernate* também interage com APIs já existentes do Java: JTA, JNDI e JDBC.

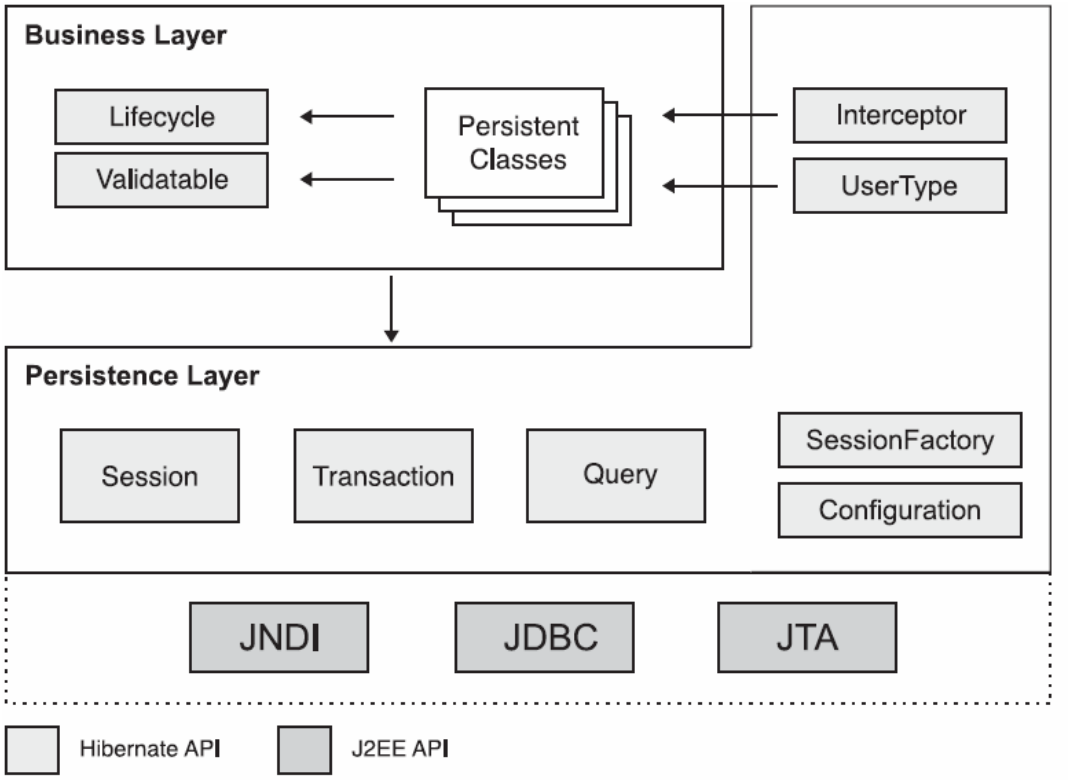

Figura 4 - Arquitetura do *Hibernate*

De todas as interfaces apresentadas na Figura 4, as principais são: `Session`, `SessionFactory`, `Transaction`, `Query`, `Configuration`. Os sub-tópicos seguintes apresentam uma descrição mais detalhada sobre elas.

4.1 Session (org.hibernate.Session)

O objeto `Session` é aquele que possibilita a comunicação entre a aplicação e a persistência, através de uma conexão JDBC. É um objeto leve de ser criado, não deve ter tempo de vida por toda a aplicação e não é *threadsafe*. Um objeto `Session` possui um *cache* local de objetos recuperados na sessão. Com ele é possível criar, remover, atualizar e recuperar objetos persistentes.

4.2 SessionFactory (org.hibernate.SessionFactory)

O objeto `SessionFactory` é aquele que mantém o mapeamento objeto relacional em memória. Permite a criação de objetos `Session`, a partir dos quais os dados são acessados, também denominado como fábrica de objetos `Sessions`.

Um objeto `SessionFactory` é *threadsafe*, porém deve existir apenas uma instância dele na aplicação, pois é um objeto muito pesado para ser criado várias vezes.

4.3 Configuration (org..hibernate.Configuration)

Um objeto `Configuration` é utilizado para realizar as configurações de inicialização do *Hibernate*. Com ele, define-se diversas configurações do *Hibernate*, como por exemplo: o *driver* do banco de dados a ser utilizado, o dialeto, o usuário e senha do banco, entre outras. É a partir de uma instância desse objeto que se indica como os mapeamentos entre classes e tabelas de banco de dados devem ser feitos.

4.4 Transaction (org.hibernate.Transaction)

A interface `Transaction` é utilizada para representar uma unidade indivisível de uma operação de manipulação de dados. O uso dessa interface em aplicações que usam *Hibernate* é opcional. Essa interface abstrai a aplicação dos detalhes das transações JDBC, JTA ou CORBA.

4.5 Interfaces Criteria e Query

As interfaces `Criteria` e `Query` são utilizadas para realizar consultas ao banco de dados.

5. Classes Persistentes

As classes persistentes de uma aplicação são aquelas que implementam as entidades domínio de negócio. O *Hibernate* trabalha associando cada tabela do banco de dados a um POJO (*Plain Old Java Object*). POJO´s são objetos Java que seguem a estrutura de *JavaBeans* (construtor padrão sem argumentos, e métodos *getters* e *setters* para seus atributos). A Tabela 1 apresenta a classe `Pessoa` representando uma classe POJO.

Tabela 1 - Exemplo de Classe POJO

```
package br.com.jeebrasil.dominio.Pessoa;

public class Pessoa {

    private int id;
    private long cpf;
    private String nome;
    private int idade;
    private Date dataNascimento;

    public Pessoa(){}

    public int getId() {
        return id;
    }

    private void setId(int Long id) {
        this.id = id;
    }

    public long getCpf(){
         return cpf;
    }

    public void setCpf(long cpf){
         this.cpf = cpf;
    }

    public Date getDataNascimento() {
        return dataNascimento;
    }

    public void setDataNascimento (Date dataNascimento) {
        this.dataNascimento = dataNascimento;
    }

    public String getNome () {
        return nome;
    }

    public void setNome (Stringnome) {
        this.nome = nome;
    }
    public int getIdade(){
        return idade;
    }
    public void setIdade(int idade){
        this.idade = idade;
    }
}
```

Considerações:

- O *Hibernate* requer que toda classe persistente possua um construtor

padrão sem argumentos, assim, o *Hibernate* pode instanciá-las simplesmente chamando `Construtor.newInstance();`

- Observe que a classe `Pessoa` apresenta métodos *setters* e *getters* para acessar ou retornar todos os seus atributos. O *Hibernate* persiste as propriedades no estilo *JavaBeans*, utilizando esses métodos;
- A classe `Pessoa` possui um atributo `id` que é o seu identificador único. É importante que, ao utilizar *Hibernate,* todos os objetos persistentes possuam um identificador e que eles sejam independentes da lógica de negócio da aplicação.

5.1 Identidade/Igualdade entre Objetos

Em aplicações Java, a identidade entre objetos pode ser obtida a partir do operador **==**. Por exemplo, para verificar se dois objetos **obj1** e **obj2** possuem a mesma identidade Java, basta verificar se **obj1 == obj2**. Dessa forma, dois objetos possuirão a mesma identidade Java se ocuparem a mesma posição de memória.

O conceito de igualdade entre objetos é diferente. Dois objetos, por exemplo, duas *Strings*, podem ter o mesmo conteúdo, mas verificar se as duas são iguais utilizando o operador **==**, pode retornar um resultado errado, pois como já citado, o operado **==** implicará em uma verificação da posição de memória e não do conteúdo. Assim, para verificar se dois objetos são iguais em Java, deve-se utilizar o método **equals**, ou seja, verificar se **obj1.equals(obj2)**.

Incluindo o conceito de persistência, passa a existir um novo conceito de identidade, a identidade de banco de dados. Dois objetos armazenados em um banco de dados são idênticos se forem mapeados em uma mesma linha da tabela.

5.2 Escolhendo Chaves Primárias

Um passo importante ao utilizar o *Hibernate* é informá-lo sobre a estratégia utilizada para a geração de chaves primárias.

Uma chave é candidata é uma coluna ou um conjunto de colunas que identifica unicamente uma linha de uma tabela do banco de dados. Ela deve satisfazer as seguintes propriedades:

- Única;
- Nunca ser nula;
- Constante.

Uma única tabela pode ter várias colunas ou combinações de colunas que satisfazem essas propriedades. Se a tabela possui um único atributo que a identifique, ele é por definição a sua chave primária. Se possuir várias chaves candidatas, uma deve ser escolhida para representar a chave primária e as demais serem definidas como chaves únicas.

Muitas aplicações utilizam como chaves primárias chaves naturais, ou seja, que têm significados de negócio. Por exemplo, o atributo `cpf` da tabela `Pessoa` (associada à classe `Pessoa`). Essa estratégia pode não ser muito boa em longo prazo, já que uma chave primária adequada deve ser constante, única e não nula. Dessa forma, se for desejado que a chave primária da tabela `Pessoa` seja uma outra ao invés do `cpf`, podem surgir problemas já que provavelmente o `cpf` deve ser referenciado em outras tabelas. Um problema que poderia acontecer seria a remoção do `cpf` da tabela.

O *Hibernate* apresenta vários mecanismos internos para a geração de chaves primárias. Veja a Tabela 2.

Tabela 2 - Mecanismo de Geração de Chaves Primárias

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

6. Objetos Persistentes, Transientes e *Detached*

Nas diversas aplicações existentes, sempre que for necessário propagar o estado de um objeto que está em memória para o banco de dados ou vice-versa, há a necessidade de que a aplicação interaja com uma camada de persistência. Isto é feito, invocando o gerenciador de persistência e as interfaces de consultas do *Hibernate*. Quando interagindo com o mecanismo de persistência, é necessário para a aplicação ter conhecimento sobre os estados do ciclo de vida da persistência.

Em aplicações orientadas a objetos, a persistência permite que um objeto continue a existir mesmo após a destruição do processo que o criou. Na verdade, o que continua a existir é seu estado, já que pode ser armazenado em disco e então, no futuro, ser recriado em um novo objeto.

Em uma aplicação não há somente objetos persistentes, pode haver também objetos transientes. Objetos transientes são aqueles que possuem um ciclo de vida limitado ao tempo de vida do processo que o instanciou. Em relação às classes persistentes, nem todas as suas instâncias possuem necessariamente um estado persistente. Elas também podem ter um estado transiente ou *detached*.

O *Hibernate* define estes três tipos de estados: persistentes, transientes e *detached.* Objetos com esses estados são definidos como a seguir:

- Objetos Transientes: são objetos que suas instâncias não estão nem estiveram associados a algum contexto persistente. Eles são instanciados, utilizados e após a sua destruição não podem ser reconstruídos automaticamente;
- Objetos Persistentes: são objetos que suas instâncias estão associadas a um contexto persistente, ou seja, tem uma identidade de banco de dados.
- Objetos *detached*: são objetos que tiveram suas instâncias associadas a um contexto persistente, mas que por algum motivo deixaram de ser associadas, por exemplo, por fechamento de sessão, finalização de sessão. São objetos em um estado intermediário, nem são transientes nem persistentes.

O ciclo de vida de um objeto persistente pode ser resumido a partir da Figura 5.

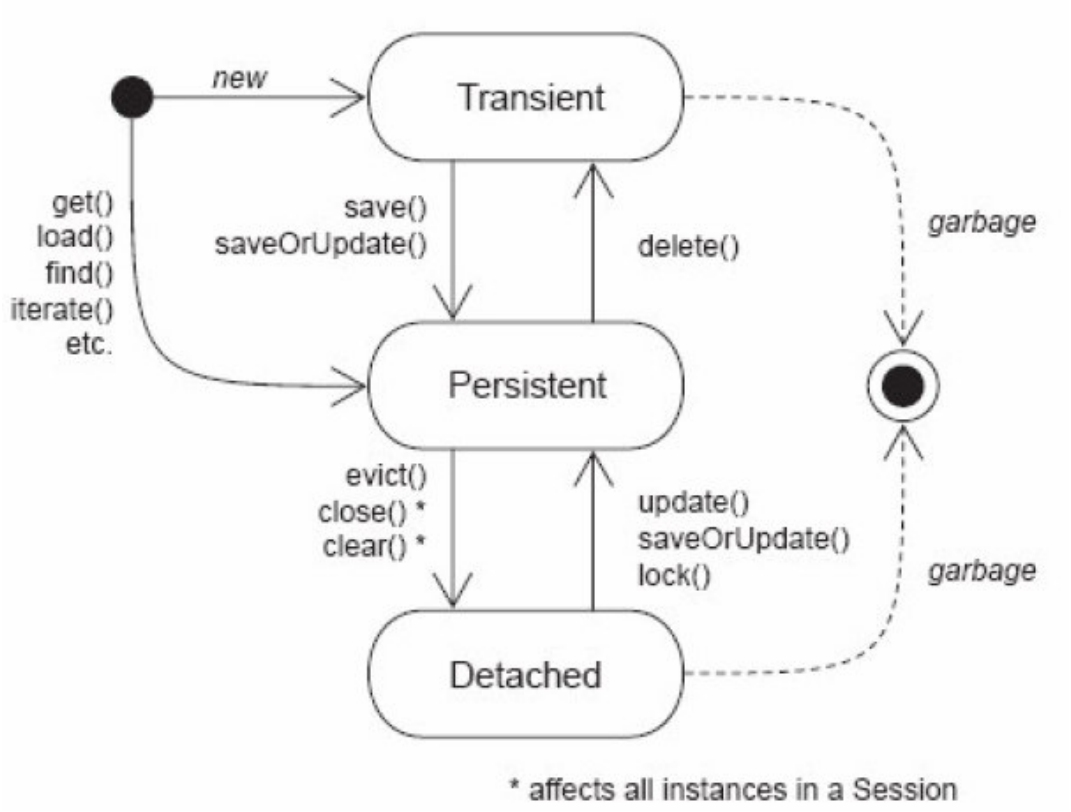

Figura 5: Ciclo de Vida - Persistência

De acordo com a figura acima, inicialmente, o objeto pode ser criado e ter o estado transiente ou persistente. Um objeto em estado transiente se torna persistente se for criado ou atualizado no banco de dados. Já um objeto em estado persistente, pode retornar ao estado transiente se for apagado do banco de dados. Também pode passar ao estado *detached*, se, por exemplo, a sessão com o banco de dados por fechada. Um objeto no estado *detached* pode voltar ao estado persistente se, por exemplo, for atualizado no banco de dados. Tanto do estado *detached* quanto do estado transiente o objeto pode ser coletado para destruição.

7. Instalando o Hibernate

Instalar o *Hibernate* é uma tarefa bastante simples. O primeiro passo é copiar sua versão do site http://hibernate.org, disponível em um arquivo compactado. Por fim, este arquivo deve ser descompactado e seu conteúdo consiste em um conjunto de arquivos JARs. Esses arquivos devem ser referenciados no *classpath* da aplicação, juntamente com a classe do *driver* do banco de dados utilizado.

8. Primeiro Exemplo de Mapeamento

Inicialmente, o *Hibernate* precisa saber como carregar e armazenar objetos de classes persistentes. Para isso, existe um arquivo de mapeamento XML que informa que tabela do banco de dados deve ser acessada para uma dada classe persistente e quais colunas na tabela são referentes a quais atributos da classe. Qualquer classe pode ser mapeada desde que seja um POJO, ou seja, possua um construtor sem argumento e seus atributos mapeados possuam métodos *getter* e *setter*. É recomendável que as tabelas possuam chaves primárias, de forma que se possa aproveitar ao máximo as funcionalidades do *Hibernate*.

Para alguns dos exemplos a serem ilustrados, considere o digrama UML mostrado na Figura 6. Nele, é apresentado o domínio simplificado de uma universidade. Como se pode observar, uma universidade é formada por um identificador (`id`), um nome, possui um endereço e um conjunto de centros. Cada centro também possui um identificador (`id`), um nome, uma universidade a que pertence e é formado por um conjunto de departamentos. Cada departamento associa-se a um conjunto de cursos, possui uma coleção de professores e possui um professor como sendo seu chefe, além do seu identificador `id` e de seus demais atributos nome e sigla. Um curso possui um identificador, um código, um nome, uma sigla, um professor como seu coordenador, associa-se a um ou mais departamentos e possui um conjunto de alunos. Um professor possui um identificador, uma matrícula, um nome, um cpf e um departamento onde é lotado. Por fim, um aluno possui um identificador, uma matrícula, um nome, um cpf e um curso a que pertence.

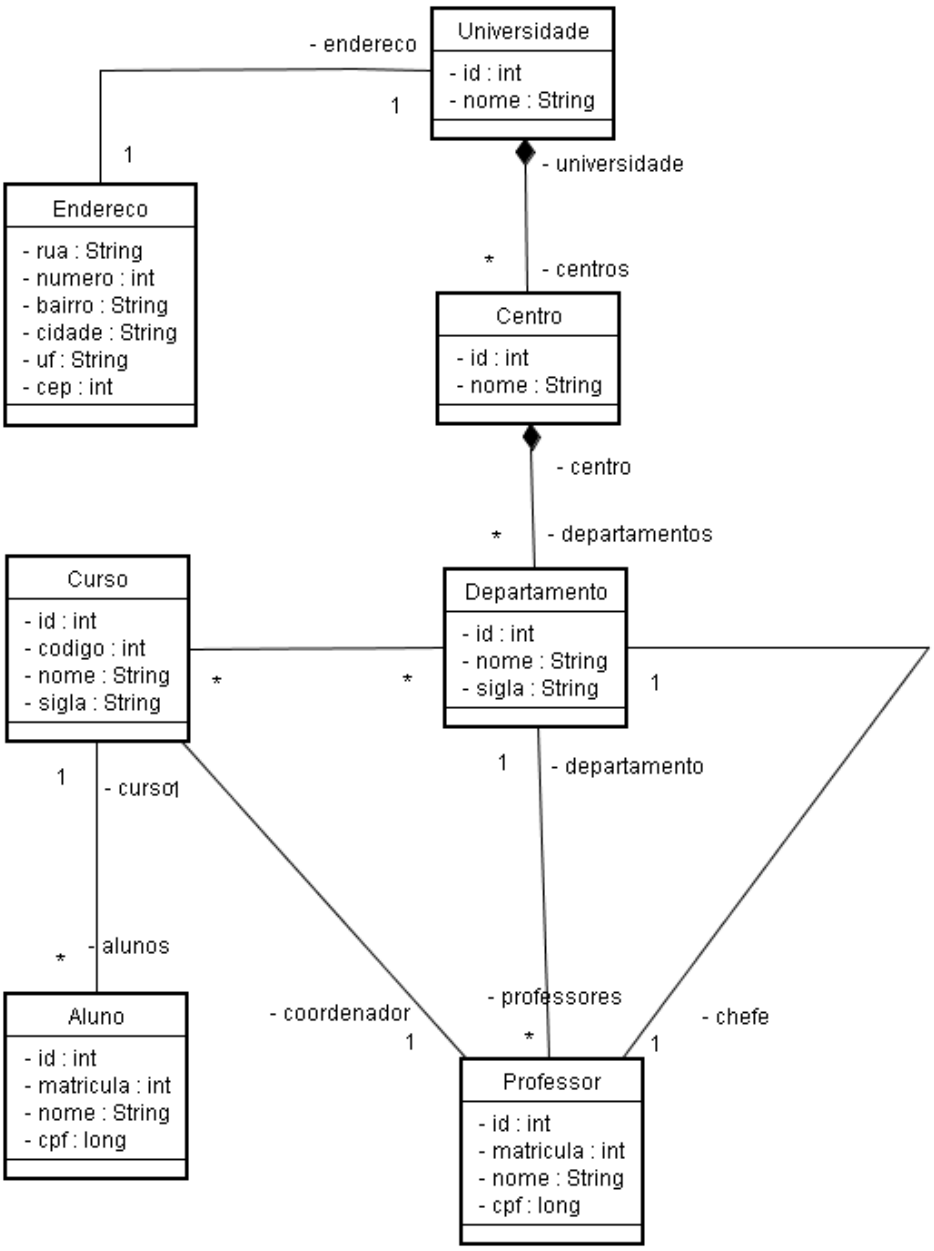

Figura 6 - Diagrama UML: Domínio Universidade

Neste primeiro exemplo, será apresentado o mapeamento da classe `Aluno` em um arquivo XML, que o *Hibernate* utiliza para saber que tabela no banco de dados a representa. Os arquivos de mapeamentos XML entre classes e tabelas do banco de dados devem ser nomeados como `*.hbm.xml` (convenção definida pela comunidade de desenvolvedores do *Hibernate*). É nos arquivos com a extensão `.hbm.xml` que o *Hibernate* realiza o mapeamento objeto relacional.

A classe que representa a entidade `Aluno` do diagrama da Figura 6 está ilustrada na Tabela 3.

Tabela 3 - Classe de Domínio: Aluno

```
package br.com.jeebrasil.dominio;

public class Aluno {

	private int id;
	private int matricula;
	private String nome;
	private long cpf;
	private Curso curso;

	public void Aluno(){}

	public long getCpf() {
		return cpf;
	}
	public void setCpf(long cpf) {
		this.cpf = cpf;
	}
	public Curso getCurso() {
		return curso;
	}
	public void setCurso(Curso curso) {
		this.curso = curso;
	}
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public int getMatricula() {
		return matricula;
	}
	public void setMatricula(int matricula) {
		this.matricula = matricula;
	}
	public String getNome() {
		return nome;
```

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

Todos os arquivos XML que mapeiam as classes para as tabelas de banco de dados no *Hibernate* possuem a estrutura básica mostrada na Tabela 4. O arquivo XML começa normalmente com as definições da DTD (Definição do Tipo do Documento) e da tag raiz, o `<hibernate-mapping>`, depois vem a tag que nos interessa neste caso, `<class>`.

.Os elementos para o mapeamento objeto relacional encontram-se entre as tags `hibernate-mapping`.

Tabela 4 - Estrutura Básica do Arquivo *.hbm.xml

```
<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
     ...
</hibernate-mapping>
```

Para a classe `Aluno` considere seu arquivo de mapeamento como sendo o `Aluno.hbm.xml`. A partir deste momento, o cabeçalho do arquivo XML será ocultado e o conteúdo entre as tags `hibernate-mapping` será apresentado.

O primeiro passo do mapeamento é informar que classe de domínio se refere a que tabela do banco de dados, através da tag `<class>`, como mostrado na Tabela 5. O único atributo obrigatório desta tag é o `name`, que contém o nome completo da classe. Se o nome da classe for diferente do nome da tabela a que se referencia, o nome da tabela é informado a partir do atributo `table`.

Tabela 5 - Aluno.hbm.xml 1

```
<hibernate-mapping>
    <class name="br.com.jeebrasil.dominio.Aluno"  table="ALUNO">
```

```
      ...
    </class>
</hibernate-mapping>
```

Em seguida, mapeia-se a chave primária da tabela, como mostrado na Tabela 6.

Tabela 6 - Aluno.hbm.xml 2

```
<hibernate-mapping>
    <class name="br.com.jeebrasil.dominio.Aluno"  table="ALUNO">
        <id name="id" column="ID_ALUNO" type="int">
            <generator class="sequence">
                  <param name="sequence">aluno_seq</param>
            </generator>
        </id>
    </class>
</hibernate-mapping>
```

A tag `id` identifica a chave primária. O atributo `name="id"` informa o nome do atributo da classe Java que se refere à chave primária da tabela. O *Hibernate* utiliza os métodos *getter* e *setter* para acessar este atributo. O atributo `column` informa ao *Hibernate* que coluna na tabela é a chave primária, no caso `ID_ALUNO`. O atributo `generator` informa qual a estratégia para a geração da chave primária, para esse exemplo, a estratégia utilizada foi *sequence*. Dessa forma, o nome da seqüência também deve ser informado (`aluno_seq`) através da tag `param`.

Agora as declarações das propriedades persistentes das classes serão incluídas no arquivo de mapeamento. Por padrão, os atributos das classes não são considerados como persistentes. Para se tornarem, é necessário incluí-los no arquivo de mapeamento.

Na Tabela 7, são incluídos os mapeamentos das propriedades persistentes através da tag `property`. Essas tags indicam propriedades simples dos objetos, como por exemplo, *String*, os tipos primitivos e seus *wrappers*, objetos *Date*, *Calendar*, entre outros.

Tabela 7 - Aluno.hbm.xml 3

```
<hibernate-mapping>
    <class name="br.com.jeebrasil.dominio.Aluno"  table="ALUNO">
        <id name="id" column="ID_ALUNO" type="int">
```

```
            <generator class="sequence">
                  <param name="sequence">aluno_seq</param>
            </generator>
        </id>
       <property name="matricula" type="int" column="MATRICULA"/>
       <property name="nome"/>
       <property name="cpf" type="long" column="CPF"
            not-null="false"/>
       <many-to-one name="curso"
            class="br.com.jeebrasil.dominio.Curso" column="ID_CURSO"/>
    </class>
</hibernate-mapping>
```

Os nomes das propriedades da classe são definidos pelo atributo XML `name`, o tipo da propriedade pelo atributo `type` e a coluna da tabela a que se refere, por `column`. Observe que a propriedade *nome* não possui seu tipo e coluna a que se referenciam definidos. Se o atributo `column` não aparece no mapeamento da propriedade, o *Hibernate* considera que a coluna na tabela do banco de dados a que se referencia possui o mesmo nome que o definido pelo atributo `name`. Em relação ao atributo `type` não ser definido, o *Hibernate* também tenta, analisando o tipo da coluna na tabela, converter para o tipo adequado Java.

O atributo `not-null` presente no mapeamento do atributo `cpf` serve para informar se a coluna pode ser ou não nula na tabela. Se for `true`, não pode ser nula. Se for `false`, pode assumir valor nulo.

A última tag do arquivo `<many-to-one>` define o relacionamento n-para-1 que a classe Aluno tem com a classe Curso. Uma descrição mais detalhada desta tag será feita posteriormente.

Depois de criar todas as classes persistentes com seus respectivos mapeamentos `*.hbm.xml`, deve-se realizar algumas configurações do *Hibernate*.

9. Configurando o Hibernate

Pode-se configurar o Hibernate de três maneiras distintas:

- Instanciar um objeto de configuração (**`org.hibernate.cfg.Configuration`**) e inserir suas propriedades programaticamente;
- Usar um arquivo **`.properties`** com as suas configurações e indicar os

arquivos de mapeamento programaticamente;

- Usar um arquivo XML (**`hibernate.cfg.xml`**) com as propriedades de inicialização e os caminhos dos arquivos de mapeamento.

Será apresentada a configuração a partir do arquivo `hibernate.cfg.xml`. Um exemplo deste arquivo de configuração pode ser visto na Tabela 8. Vários parâmetros podem ser configurados. Basicamente, deve-se configurar:

- A URL de conexão com o banco de dados;
- Usuário e senha do banco de dados;
- Números máximo e mínimo de conexões no pool;
- Dialeto.

Tabela 8 - hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <!-- a SessionFactory instance listed as /jndi/name -->
    <session-factory name="java:comp/env/hibernate/SessionFactory">

        <!-- properties -->
        <property name="connection.driver_class">
            org.postgresql.Driver
        </property>
        <property name="connection.url">
            jdbc:postgresql://localhost:5432/banco
        </property>
        <property name="dialect">
            org.hibernate.dialect.PostgreSQLDialect
        </property>
        <property name="show_sql">true</property>
        <property name="connection.username">postgres</property>
        <property name="connection.password">postgres</property>
        <property name="connection.pool_size">10</property>

        <!-- mapping files -->
        <mapping resource="br/con/jeebrasil/conf/Aluno.hbm.xml"/>
```

```
        <mapping resource="br/con/jeebrasil/conf/Centro.hbm.xml"/>
        <mapping resource="br/con/jeebrasil/conf/Curso.hbm.xml"/>
        <mapping resource="br/con/jeebrasil/conf/Departamento.hbm.xml"/>
        <mapping resource="br/con/jeebrasil/conf/Professor.hbm.xml"/>
        <mapping resource=" br/con/jeebrasil/conf/Universidade.hbm.xml"/>

    </session-factory>
</hibernate-configuration>
```

Resumindo as descrições das propriedades a serem configuradas:

- `hibernate.dialect`: implementação do dialeto SQL específico do banco de dados a ser utilizado. Usado para identificar as particularidades do banco de dados;
- `hibernate.connection.driver_class`: nome da classe do *driver* JDBC do banco de dados que está sendo utilizado;
- `hibernate.connection.url`: é a URL de conexão específica do banco que está sendo utilizado;
- `hibernate.connection.username`: é o nome de usuário com o qual o Hibernate deve se conectar ao banco;
- `hibernate.connection.password`: é a senha do usuário com o qual o Hibernate deve se conectar ao banco;
- `hibernate.connection.pool_size`: tamanho do pool de conexões;
- `hibernate.connection.isolation`: define o nível de isolamento. Parâmetro opcional;
- `hibernate.show_sql`: utilizado para definir se os SQL's gerados pelo Hibernate devem ou não ser exibidos (`true | false`).

O *Hibernate* trabalha com dialetos para um grande número de bancos de dados, tais como: *DB2, MySQL, Oracle, Sybase, Progress, PostgreSQL, Microsoft SQL Server, Ingres, Informix* entre outros. Possíveis valores para os dilates estão presentes na Tabela 9.

Tabela 9 - Possíveis valores de dialetos

```
DB2 – org.hibernate.dialect.DB2Dialect
HypersonicSQL – org.hibernate.dialect.HSQLDialect
Informix – org.hibernate.dialect.InformixDialect
Ingres – org.hibernate.dialect.IngresDialect
```

Interbase – org.hibernate.dialect.InterbaseDialect
Pointbase – org.hibernate.dialect.PointbaseDialect
PostgreSQL – org.hibernate.dialect.PostgreSQLDialect
Mckoi SQL – org.hibernate.dialect.MckoiDialect
Microsoft SQL Server – org.hibernate.dialect.SQLServerDialect
MySQL – org.hibernate.dialect.MySQLDialect
Oracle (any version) – org.hibernate.dialect.OracleDialect
Oracle 9 – org.hibernate.dialect.Oracle9Dialect
Progress – org.hibernate.dialect.ProgressDialect
FrontBase – org.hibernate.dialect.FrontbaseDialect
SAP DB – org.hibernate.dialect.SAPDBDialect
Sybase – org.hibernate.dialect.SybaseDialect
Sybase Anywhere – org.hibernate.dialect.SybaseAnywhereDialect

O final do arquivo `hibernate.cfg.xml` é onde devem ser informados os arquivos de mapeamentos das classes que o *Hibernate* deve processar. Se algum arquivo de mapeamento não for definido neste local, a classe a que se refere não poderá ser persistida utilizando o *Hibernate*.

10. Manipulando Objetos Persistentes

O *Hibernate* utiliza objetos `Session` para persistir e recuperar objetos. Um objeto `Session` pode ser considerado como uma sessão de comunicação com o banco de dados através de uma conexão JDBC.

O código fonte exibido na Tabela 10 mostra a criação e persistência de um objeto do tipo `Aluno`.

Tabela 10 - Exemplo de Persistência

```
...
1. try{
2.   //SessionFactory deve ser criado uma única vez durante a execução
3   //da aplicação
4   SessionFactory sf = new  Configuration()
5      .configure("/br/com/jeebrasil/conf/hibernate.cfg.xml")
6.       .buildSessionFactory();
7.
8.   Session session = sf.openSession(); //Abre sessão
9.   Transaction tx = session.beginTransaction(); //Cria transação
10.
11.   //Cria objeto Aluno
```

```
12.   Aluno aluno = new Aluno();
13.   aluno.setNome("Luis Eduardo Pereira Júnior");
14.   aluno.setMatricula(200027803);
15.   aluno.setCpf(1234567898);
16.   aluno.setCurso(curso); //Considera-se que o objeto "curso" já
17.                          //havia gravado na base de dados e
18.                          //recuperado para ser atribuído neste
19.                          // momento.
20.   session.save(aluno); //Realiza persistência
21.   tx.commit(); //Fecha transação
22.   session.close(); //Fecha sessão

23. }catch(HibernateException e1){
24.   e1.printStackTrace();
25. }catch(SQLException e2){
26.   e2.printStackTrace();
27. }
```

O código presente nas linhas 4-6 deve ser chamado uma única vez durante a execução da aplicação. O objeto `SessionFactory` armazena os mapeamentos e configurações do *Hibernate*. É um objeto pesado e lento de se criar.

A Tabela 11 apresenta alguns dos métodos que podem ser invocados a partir do objeto `Session`.

Tabela 11 - Métodos invocados a partir do objeto `Session`

save(Object)	Inclui um objeto em uma tabela do banco de dados.
saveOrUpdate(Object)	Inclui um objeto na tabela caso ele ainda não exista (seja transiente) ou atualiza o objeto caso ele já exista (seja persistente).
delete(Object)	Apaga um objeto da tabela no banco de dados.
get(Class, Serializable id)	Retorna um objeto a partir de sua chave primária. A classe do objeto é passada como primeiro argumento e o seu identificador como segundo argumento.

Em relação ao método `saveOrUpdate`, uma questão que se pode formular é "Como o *Hibernate* sabe se o objeto em questão já existe ou não no banco de dados, ou seja, se ele deve ser criado ou atualizado?". A resposta é simples: o desenvolvedor deve informar isso a ele. Essa informação é incluída no arquivo de

configuração `*.hbm.xml` da classe na tag que define a chave primária. No caso da classe Aluno, a tag da chave primária no arquivo `Aluno.hbm.xml` deve ter o atributo <`unsaved-value`> adicionado, como mostrado na Tabela 12. Neste caso com o atributo `unsaved-value="0"` significa que, no momento da chamada ao método `saveOrUpdate(Objetc obj)`, se o atributo identificador do objeto estiver com valor 0 (zero) significa que ele deve ser criado na tabela do banco de dados. Dessa forma, se o seu valor for diferente de zero, o objeto deve ter sua linha na tabela atualizada (deve-se garantir que o valor do identificador do objeto se refere a um valor da chave primária da tabela).

Tabela 12 - Trecho de Aluno.hbm.xml

```
…
<id name="id" column="ID_ALUNO" type="int" unsaved-value="0">
   <generator class="sequence">
      <param name="sequence">aluno_seq</param>
   </generator>
</id>
…
```

As Tabela 13 e Tabela 14 apresentam exemplos dos métodos invocados a partir do objeto `Session`.

Tabela 13 - Exemplo de Busca e Atualização de Objeto

```
...
   Session session = sf.openSession();
   Transaction tx = session.beginTransaction();

   //Busca objeto aluno da base de dados com chave primária = 1
   Aluno aluno = (Aluno) session.get(Aluno.class, 1);
   //Atualiza informação de matrícula.
   aluno.setMatricula(200027807);
   //Como o identificador do objeto aluno é diferente de 0,
   //a sua matrícula é atualizada já que foi alterada
   session.saveOrUpdate(aluno);
   tx.commit();
   session.close();
...
```

Tabela 14 - Exemplo de Remoção de Objeto

```
...
   Session session = sf.openSession();
   Transaction tx = session.beginTransaction();

   Aluno aluno = new Aluno();
   //Existe linha na tabela aluno com chave primária = 2
   aluno.setId(2);
   //Deleta aluno com id = 2 da tabela.
   //Somente necessária informação do seu identificador
   session.delete (aluno);

   tx.comiit();
   session.close();
...
```

11. Associações

O termo associação é utilizado para se referir aos relacionamentos entre as entidades. Os relacionamentos n-para-n, n-para-1 e 1-para-n são os mais comuns entre as entidades de um banco de dados.

Todos os exemplos apresentados nesta seção baseiam-se no diagrama de classes mostrado na Figura 6.

11.1 Associações 1-n (one-to-many)

Para exemplificar o relacionamento 1-n, considere o relacionamento entre a entidade `Centro` e a entidade `Universidade` da Figura 7. O relacionamento diz que uma universidade possui um conjunto de *n* centros e um centro está associado a apenas uma única universidade. Considere as classes de domínio Java de uma universidade e de um centro, respectivamente, mostradas na Tabela 15 e na Tabela 16.

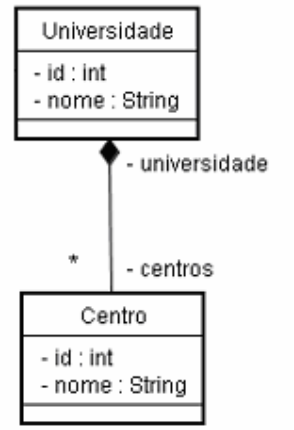

Figura 7 - Relacionamento entre Centro e Universidade

Tabela 15 - Classe de Domínio: Universidade

```
package br.com.jeebrasil.dominio;
import java.util.Collection;
public class Universidade implements Serializable{
        private int id;
        private String nome;
        private Endereco endereco;
        private Collection centros;

        //Implementação dos métodos setter e getter
        ...
}
```

Tabela 16 - Classe de Domínio: Centro

```
package br.com.jeebrasil.dominio;
import java.util.Collection;
public class Centro implements Serializable{
      private int id;
      private String nome;
      private Universidade universidade;
      //departamentos -> Atributo mapeado das mesma forma que a
      //coleção centros em Universidade
      private Collection departamentos;

      //Implementação dos métodos setter e getter
}
```

A classe de domínio `Universidade` é a que possui um mapeamento do tipo 1-n. O seu mapeamento pode ser visto na Tabela 17. Neste momento, as informações do endereço da universidade foram desconsideradas.

Tabela 17 - Mapeamento 1-n: tabela universidade

```
...
<hibernate-mapping>
    <class name="br.com.jeebrasil.dominio.Universidade"
           table="UNIVERSIDADE">
        <id name="id" column="ID_UNVERSIDADE" type="int">
            <generator class="increment"/>
        </id>
        <property name="nome"/>

        <!-- Mapeamento da Coleção de centros -->
        <set name="centros" inverse="true" lazy="true">
            <key column="ID_UNIVERSIDADE"/>
            <one-to-many class="br.com.jeebrasil.dominio.Centro"/>
        </set>
    </class>
</hibernate-mapping>
```

Observa-se que para realizar o mapeamento 1-n, ou seja, da coleção `centros` foi utilizada uma tag `set`. Um *set* ou um conjunto representa uma coleção de objetos não repetidos que podem ou não estar ordenados. O atributo `name` define a propriedade que está sendo tratada para realizar o relacionamento 1-n. O atributo `key` representa a coluna da tabela relacionada (`Centro`) que possui a chave estrangeira para a classe `Universidade`. O nome da coluna da chave estrangeira (ID_UNIVERSIDADE) é informado no atributo `column`. Na tag `<one-to-many>` informa-se a classe a qual pertence à coleção de objetos, no caso `br.com.jeebrasil.dominio.Centro`.

A tag `set` também apresenta um atributo denominado `inverse`. Esse atributo é utilizado para que o *Hibernate* saiba como tratar a associação entre duas tabelas. Quando um lado da associação define o atributo `inverse` como `true`, indica que a ligação do relacionamento entre a associação será de responsabilidade do "outro lado" da associação.

Considerando o relacionamento entre uma universidade e seus centros, no mapeamento do conjunto de centros em universidade, o atributo `inverse` é igual a

`true`. Dessa forma, a criação ou atualização do relacionamento entre um centro e uma universidade será feita durante a persistência ou atualização de um objeto `Centro`.

A partir do exemplo apresentado na Tabela 18, cria-se uma instância da classe `Universidade` e define-se um valor para o atributo nome. Em seguida, uma instância da classe `Centro` também é criada, definindo o atributo nome e o atributo universidade (como sendo a instância de `Universidade` anteriormente criada). Por fim, o objeto `Centro` criado é adicionado à coleção de centros do objeto `Universidade`, o qual é persistido.

Tabela 18 - Exemplificando uso do atributo inverse

```
...
Universidade univ = new Universidade()
univ.setNome("Universidade Federal do Rio Grande do Norte");

Centro centro = new Centro();
centro.setNome("Centro de Tecnologia");
centro.setUniversidade(univ);

univ.setCentros(new HashSet<Centro>());
univ.getCentros().add(centro);

session.save(univ);
...
```

A Tabela 19 apresenta o que acontece se o atributo `inverse` no mapeamento 1-n for definido como `false`. O que acontece é que o *Hibernate* insere uma linha na tabela `UNIVERSIDADE` e em seguida tenta atualizar o relacionamento entre uma universidade e um centro, no caso a partir de um `UPDATE` na tabela `CENTRO`, setando a chave estrangeira para a tabela `UNIVERSIDADE`.

Tabela 19 – SQL gerado pelo Hibernate com atributo inverse=false

```
Hibernate: insert into UNIVERSIDADE (nome, ID_ENDERECO,
ID_UNIVERSIDADE) values (?, ?, ?)
Hibernate: update CENTRO set ID_UNIVERSIDADE=? where ID_CENTRO=?
```

Verifica-se, então, que com o atributo `inverse` sendo igual a `false`, o *Hibernate* tenta atualizar o relacionamento entre uma universidade e um centro logo após a inserção da universidade. Neste caso, como o relacionamento com a

universidade está sendo feito com um objeto transiente de um `Centro`, o que vai ocorrer é que o *Hibernate* vai tentar atualizar a chave estrangeira para a tabela `UNIVERSIDADE` em uma linha da tabela `CENTRO` que não existe, acontecendo o erro exibido na Tabela 20.

Tabela 20 – Erro gerado após teste com o atributo inverse=false no relacionamento 1-n em Universidade

```
ERRO:
Exception in thread "main" org.hibernate.TransientObjectException:
object references an unsaved transient instance – save the transient
instance before flushing: br.com.jeebrasil.dominio.Centro ...
```

De acordo com o mapeamento da Tabela 17, o atributo `inverse` é definido como `true`. Então, para o código da Tabela 18 vai ser gerado o SQL exibido na Tabela 21. Neste caso, veja que apenas é dado um `INSERT` na tabela `UNIVERSIDADE`, ou seja, no momento da inserção de uma universidade o *Hibernate* não atualiza o relacionamento entre ela e seus centros, pois espera que ele seja feito no momento da inserção/atualização de um objeto `Centro`.

Tabela 21 - SQL gerado pelo Hibernate com atributo inverse=true

```
Hibernate: insert into UNIVERSIDADE (nome, ID_ENDERECO,
ID_UNIVERSIDADE) values (?, ?, ?)
```

Resumindo, se o atributo `inverse` não for definido como `true`, o *Hibernate* não tem como saber qual dos dois lados foi atualizado, ou seja, vai sempre atualizar os dois lados de uma vez, uma atualização para cada classe da relação, o que seria desnecessário. Caso contrário, o *Hibernate* passa, a saber, de qual lado fazer a atualização e fazendo uma única vez.

Na tag `set` também está presente o atributo `lazy`. Ele é utilizado para resolver o seguinte problema: quando se realiza um `select` em um objeto `Universidade` implica em serem feitos `n` (número de centros da universidade) outros `select's` para buscar os seus centros. Dessa forma, a resolução do problema é feita apenas definindo o atributo `lazy` como sendo `true`. A coleção de centros passa a ser *lazy-loading*, o que significa que somente será recuperada quando solicitada, ou seja, a coleção de centros de uma universidade só seria solicitada caso o programador a acesse através da chamada ao método `getCentros()`.

11.2 Associações n-1 (many-to-one)

O relacionamento n-1 será apresentado a partir do relacionamento `<many-to-one>` existente entre a tabela `Centro` e a tabela `Universidade`. Neste caso, o relacionamento está presente no mapeamento da classe `Centro`, como mostrado na Tabela 22.

Tabela 22 - Mapeamento n-1: tabela centro

```
...
<hibernate-mapping>
    <class name="br.com.jeebrasil.dominio.Centro"
           table="CENTRO">
        <id name="id" column="ID_CENTRO" type="int">
            <generator class="increment"/>
        </id>
        <property name="nome" type="java.lang.String"/>

        <!-- Mapeamento da Universidade -->
       <many-to-one name="universidade"
            class="br.com.jeebrasil.dominio.Universidade"
cascade="none"
            fech="join" update="true" insert="true" lazy="true"
            column="id_universidade"/>

       <set name="departamentos" lazy="true" inverse="true">
            <key column="ID_CENTRO"/>
            <one-to-many
class="br.com.jeebrasil.dominio.Departamento"/>
      </set>
    </class>
</hibernate-mapping>
```

Como mostrado no mapeamento da classe `Centro`, o relacionamento n-1 é mapeado a partir da tag `<many-to-one>`. Essa tag apresenta um conjunto de atributos, que podem assumir os valores apresentados na Tabela 23.

Tabela 23 – Mapeamento n-1: atributos

```
 <many-to-one name="propertyName"
     class="ClassName" column="column_name"
     fetch="join|select" update="true|false" lazy="true|false"
     insert="true|false" cascade="all|none|save-update|delete"
 />
```

- `name`: nome do atributo na classe Java;
- `column`: coluna do banco de dados. É uma chave estrangeira;
- `class`: nome da classe Java da entidade relacionada;
- `insert` e `update`: indica se o atributo será incluído e alterado ou somente lido;
- `cascade`: indica com que ação em cascata o relacionamento será tratado.
    - `none`: associação é ignorada;
    - `save-update`:os objetos associados vão ser inseridos ou atualizados automaticamente quando o objeto "pai" for inserido ou atualizado;
    - `delete`: os objetos associados ao objeto "pai" vão ser deletados;
    - `all`: junção de `delete` e `save-update`;
    - `all-delete-orphan`: o mesmo que `all`, mas o *Hibernate* deleta qualquer objeto que tiver sido retirado da associação;
    - `delete-orphan`: se o objeto não fizer mais parte da associação, ele removido.
- `fetch`: se definido como `join` é usado para realizar *joins* sem restrição de nulidade (`outer-join`). Se for `select`, um novo *select* é feito para recuperar a informação da associação.
- `lazy`: se igual a `true`, o objeto só será recuperado se solicitado; se igual a `false`, o objeto sempre será recuperado.

11.3 Associações n-n (many-to-many)

O relacionamento n-n será feito a partir do relacionamento entre as entidades `Departamento` e `Curso` mostrado na Figura 8.

Figura 8 - Relacionamento n-n entre Curso e Departamento

Um relacionamento n-n implica em existir uma nova tabela para mapear o relacionamento no banco de dados. Vamos denominar essa nova tabela como `DEPARTAMENTO_CURSO`, como mostrado na Figura 9. Dessa forma, um departamento possui uma coleção de cursos e um curso uma coleção de departamentos. A existência dessas coleções é opcional. Por exemplo, pode ser que em um sistema real não seja necessário saber todos os departamentos de determinado curso, mas se for realmente necessário, o *Hibernate* apresenta outros mecanismos para a obtenção desta informação.

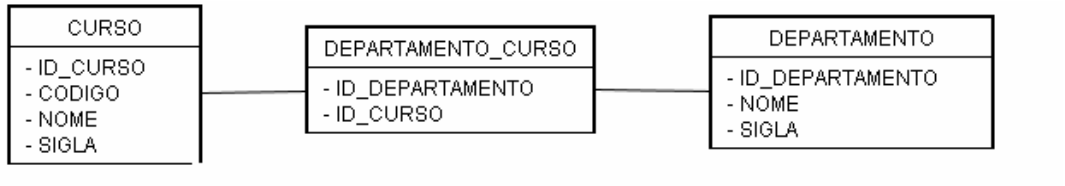

Figura 9 - Tabela de relacionamento DEPARTAMENTO_CURSO

As classes Java das entidades `Departamento` e `Curso` estão ilustradas nas Tabela 24 e Tabela 25, respectivamente.

Tabela 24 - Classe de Domínio: Departamento

```
package br.com.jeebrasil.dominio;
import java.util.Collection;
public class Departamento implements Serializable{
	private int id;
	private String nome;
	private String sigla;
	private Centro centro;
	private Collection professores;
	private Collection cursos;

	//	Implementação dos métodos setter e getter
}
```

Tabela 25 - Classe de Domínio: Curso

```
package br.com.jeebrasil.dominio;
import java.util.Collection;
public class Curso implements Serializable{
	private int id;
	private int codigo;
```

```
      private String nome;
      private String sigla;
      private Collection departamentos;
      private Collection alunos;

      //Implementação dos métodos setter e getter
}
```

A Tabela 26 apresenta o mapeamento da classe Departamento. Já a Tabela 27, o mapeamento da classe Curso.

Tabela 26 - Departamento.hbm.xml

```
...
<hibernate-mapping>
    <class name="br.com.jeebrasil.dominio.Departamento"
           table="DEPARTAMENTO">
        <id name="id" column="ID_DEPARTAMENTO" type="int">
            <generator class="increment"/>
        </id>
        <property name="nome" type="java.lang.String"/>
        <property name="sigla" type="java.lang.String
        <many-to-one name="centro"
class="br.com.jeebrasil.dominio.Centro"
                  column="ID_CENTRO"
                  cascade="save-update"/>
        <set name="professores">
            <key column="ID_DEPARTAMENTO"/>
            <one-to-many class="br.com.jeebrasil.dominio.Professor"/>
        </set>

       <!-- Mapeamento dos cursos -->
       <set name="cursos" table="DEPARTAMENTO_CURSO"
            inverse="true">
            <key column="ID_DEPARTAMENTO"/>
            <many-to-many column="ID_CURSO"
                   class="br.com.jeebrasil.dominio.Curso"/>
       </set>
    </class>
</hibernate-mapping>
```

Tabela 27 - Curso.hbm.xml

```
...
<hibernate-mapping>
    <class name="br.com.jeebrasil.dominio.Curso"  table="CURSO">
        <id name="id" column="ID_CURSO" type="int">
            <generator class="increment"/>
        </id>
        <property name="codigo"/>
        <property name="nome"/>
        <property name="sigla"/>
        <set name="alunos">
            <key column="ID_CURSO"/>
            <one-to-many class="br.com.jeebrasil.dominio.Aluno"/>
      </set>

      <!-- Mapeamento dos departamentos-->
      <set name="departamentos" table="DEPARTAMENTO_CURSO">
            <key column="ID_CURSO"/>
            <many-to-many column="ID_DEPARTAMENTO"
                        class="br.com.jeebrasil.dominio.Departamento"/>
      </set>
    </class>
</hibernate-mapping>
```

Observa-se que tanto no mapeamento da coleção `cursos` em `Departamento` quanto no da coleção `departamentos` em `Curso`, o relacionamento n-n é feito a partir de uma tag `set`. Os mapeamentos das duas coleções apresentam uma tag `key`, na qual o atributo `column` indica a chave estrangeira do pai na tabela de relacionamento `DEPARTAMENTO_CURSO`. Apresentam também a tag `many-to-many` utilizada para indicar a entidade filha e sua chave estrangeira no relacionamento. A única diferença entre o mapeamento das duas coleções é que na tag `set` de cursos no mapeamento da entidade `Departamento` o atributo `inverse` é igual a `true`, significando que na tabela `DEPARTAMENTO_CURSO` só será inserido o relacionamento entre as duas entidades, quando um curso for inserido no banco de dados associado a um departamento.

Neste caso, não seria necessário mapear a coleção de cursos em `Departamento`, pois não está sendo utilizada para popular a tabela de relacionamento. Como já citado, para recuperar a coleção de cursos de um departamento, o *Hibernate* apresenta outros mecanismos.

Em outros relacionamentos n-n, pode ser que seja necessário inserir/atualizar o relacionamento na tabela de relacionamento durante a inserção/atualização de qualquer um dos lados das entidades, portanto, basta que o atributo `inverse` nas duas coleções seja mapeado como `false`.

O código presente na Tabela 28 cria um instância de um objeto `Departamento`. Em seguida recupera um objeto persistente da classe `Curso` com identificador igual a **1**, adiciona esse objeto `Curso` na coleção de cursos do objeto `Departamento` criado e, por fim, persiste o departamento. O SQL gerado é apresentado na Tabela 29. Observa-se que apenas um `SELECT` na tabela `CURSO` é feito e um `INSERT` na tabela `DEPARTAMENTO`. Não há uma inclusão de linha na tabela de relacionamento `DEPARTAMENTO_CURSO`, pois o atributo `inverse` da coleção de cursos no mapeamento da classe `Departamento` foi definido como `true`.

Tabela 28 - Persistência de Departamento

```
Departamento depart = new Departamento();
depart.setNome("Departamento 1");
Curso curso = (Curso)session.get(Curso.class, 1);
depart.getCursos().add(curso);
session.save(depart);
```

Tabela 29 - SQL para comandos da Tabela 28

```
Hibernate: select curso0_.ID_CURSO as ID1_0_, curso0_.codigo as
   codigo1_0_, curso0_.nome as nome1_0_, curso0_.sigla as sigla1_0_,
   curso0_.ID_COORDENADOR as ID5_1_0_ from CURSO curso0_ where
   curso0_.ID_CURSO=?
Hibernate: insert into DEPARTAMENTO (nome, sigla, ID_CENTRO, ID_CHEFE,
   ID_DEPARTAMENTO) values (?, ?, ?, ?, ?)
```

Já o código presente na Tabela 30, cria uma instância da classe `Curso`, busca o objeto persistente `Departamento` de identificador **1** na base de dados, adiciona o departamento a coleção de `departamentos` do objeto `Curso` e realiza a a sua persistência. O SQL presente na Tabela 31 mostra que um `SELECT` foi feito na tabela `DEPARTAMENTO`, um `INSERT` na tabela `CURSO` e um outro `INSERT` na tabela DEPA`RTAMENTO_CURSO`. Nesse caso, o relacionamento foi persistido, pois no mapeamento da coleção `departamentos` da classe `Curso`, como o atributo `inverse` não foi definido como `true`, assume-se que ele é `false`.

Tabela 30 - Persistência de Curso

```
Curso curso = new Curso();
Departamento d = (Departamento)session.get(Departamento.class, 1);
curso.getDepartamentos().add(d);
session.save(curso);
```

Tabela 31 - SQL para comandos da Tabela 30

```
Hibernate: select departamen0_.ID_DEPARTAMENTO as ID1_0_,
   departamen0_.nome as nome4_0_, departamen0_.sigla as sigla4_0_,
   departamen0_.ID_CENTRO as ID4_4_0_, departamen0_.ID_CHEFE
   as ID5_4_0_
   from DEPARTAMENTO departamen0_ where departamen0_.ID_DEPARTAMENTO=?
Hibernate: insert into CURSO (codigo, nome, sigla, ID_COORDENADOR,
   ID_CURSO) values (?, ?, ?, ?, ?)
Hibernate: insert into DEPARTAMENTO_CURSO (ID_CURSO, ID_DEPARTAMENTO)
   values (?, ?)
```

11.4 Associações n-n com Atributos

Imagine que seria necessário guardar a data em que foi feita a associação entre um determinado curso e um determinado departamento, ou seja, necessário ter um novo atributo na tabela `DEPARTAMENTO_CURSO`. Dessa forma, a tabela `DEPARTAMENT_CURSO` não seria formada apenas pelos identificadores de curso e de departamento, mas sim também pela data. Para essa situação, os mapeamentos dos relacionamentos `<many-to-many>` nas tabelas `Curso` e `Departamento` não resolveriam o problema. Então, deve-se criar uma nova classe `DepartamentoCurso`, como mostrado no diagrama da Figura 10. Como pode ser visto nesta figura, existem agora relacionamentos `<many-to-one>` e uma chave primária dupla.

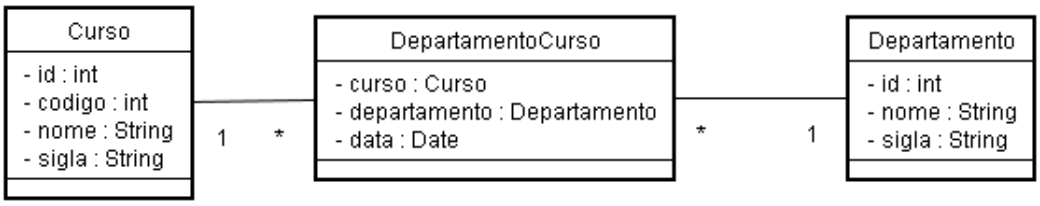

Figura 10 - Mapeamento n-n com Atributo

Dessa forma, o mapeamento da tabela DEPARTAMENTO_CURSO seria feito

como mostrado na Tabela 32.

Tabela 32 - DepartamentoCurso.hbm.xml

```
...
<hibernate-mapping>
    <class name="br.com.jeebrasil.dominio.DepartamentoCurso"
           table="DEPARTAMENTO_CURSO">
        <composite-id name="compositeID"

class="br.com.jeebrasil.dominio.DepartamentoCursoID">

            <key-many-to-one name="curso" column="ID_CURSO"
                class="br.com.jeebrasil.dominio.Curso"/>
            <key-many-to-one name="departamento "
                column="ID_DEPARTAMENTO"
                class="br.com.jeebrasil.dominio.Departamento "/>
        </composite-id>

        <property name="data" type="java.util.Date" column="data"/>

    </class>
</hibernate-mapping>
```

11.4.1 Composite-id

Como pode ser visto na Tabela 32, há o mapeamento de uma chave composta. Neste caso, a chave é um objeto da classe `Curso` e um objeto da classe `Departamento`. O primeiro passo é a criação de uma classe de domínio para a chave composta, `DepartamentoCursoID` (Tabela 33).

Tabela 33 - Classe de Domínio: DepartamentoCursoID

```
public class DepartamentoCursoID implements Serializable{

      private Departamento departamento;
      private Curso curso;
      //Métodos getter e setter
      ...
}
```

Para esse exemplo, a classe `DepartamentoCurso` possuirá um atributo `Date data` e um atributo `DepartamentoCursoID compositeID`. Veja Tabela 34.

Tabela 34 - Classe de Domínio: DepartamentoCurso

```
public class DepartamentoCurso implements Serializable{

      private DepartamentoCursoID compositeID;
      private Date data;
      //Métodos getter e setter
      ...
}
```

Observando a Tabela 32, vê-se que a tag `<composite-id>` mapeia a chave composta. O atributo `name` informa o atributo que mapeia a chave composta e o atributo `class` a classe de domínio, no caso `br.com.jeebrasil.dominio.DepartamentoCursoID`. O corpo da tag é formado por duas outras tags `<key-many-to-one>` que informam os atributos da chave composta dentro da classe `DepartamentoCursoID`.

Para exemplificar o relacionamento n-n com atributos, observe o exemplo da Tabela 35. Primeiro um departamento e um curso são buscados da base de dados, ambos com identificadores iguais a 1. Em seguida, cria-se uma instância de um objeto da classe `DepartamentoCursoID` que representa a chave composta. Os valores que compõem a chave, curso e departamento, são atribuídos. Finalmente, cria-se um objeto da classe `DepartamentoCurso` que representa a tabela de relacionamento entre as entidades, define-se a sua chave composta e a data de criação, persistindo-o na base de dados.

Tabela 35 - Exemplo: Relacionamento n-n com atributos

```
...
Departamento d = (Departamento)session.get(Departament1.class, 1);
Curso c = (Curso)session.get(Curso.class, 1);

DepartamentoCursoID dcID = new DepartamentoCursoID();
dcID.setDepartamento(d);
dcID.setCurso(c);
```

```
DepartamentoCurso dc = new DepartamentoCurso();
dc.setCompositeID(dcID);
dc.setData(new Date());

session.save(dc);
...
```

O resultado da execução do código presente na Tabela 35 pode ser visto na Tabela 36.

Tabela 36 - Resultado da execução do código presente na Tabela 35

```
Hibernate: select departamen0_.ID_DEPARTAMENTO as ID1_0_ from
   DEPARTAMENTO1 departamen0_ where departamen0_.ID_DEPARTAMENTO=?
Hibernate: select curso1x0_.ID_CURSO as ID1_0_ from CURSO1 curso1x0_
   where curso1x0_.ID_CURSO=?
Hibernate: insert into DEPARTAMENTO_CURSO (data, ID_CURSO,
   ID_DEPARTAMENTO) values (?, ?, ?)
```

11.5 Associações 1-1 (one-to-one)

Da Figura 6, considere o relacionamento 1-1 entre as entidades `Universidade` e `Endereco`, ou seja, uma universidade tem um único endereço e um endereço pertence apenas a uma única universidade.

As Tabela 37 e Tabela 15 apresentam as classes Java para as entidades `Universidade` e `Endereco`, respectivamente.

Tabela 37 - Classe de Domínio: Endereço

```
package br.com.jeebrasil.dominio;
public class Endereco implements Serializable{
        private int id;
        private String rua;
        private int numero;
        private String bairro;
        private String cidade;
        private String uf;
        private int cep;
```

```
        private Universidade universidade;

        //Implementação dos métodos setter e getter
        ...
}
```

Existem duas formas de se mapear este relacionamento 1-1. A primeira estratégia é no mapeamento da entidade `Universidade` adicionar um mapeamento <many-to-one> para a tabela `Endereco`. O mapeamento da entidade `Endereco` é visto na Tabela 38 e não apresenta nenhuma novidade.

Tabela 38 - Endereco.hbm.xml

```
...
<hibernate-mapping>
    <class name="br.com.jeebrasil.dominio.Endereco"  table="ENDERECO">
        <id name="id" column="ID_ENDERECO" type="int">
            <generator class="native"/>
        </id>
        <property name="rua"/>
        <property name="numero"/>
        <property name="bairro"/>
        <property name="cidade"/>
        <property name="uf"/>
        <property name="cep"/>
    </class>
</hibernate-mapping>
```

Para realizar o mapeamento 1-1 de acordo com essa estratégia, basta inserir um mapeamento <many-to-one> no mapeamento da tabela Universidade (Tabela 17), como mostrado na Tabela 39.

Tabela 39 – Relacionamento 1-1 em `Universidade.hbm.xml`: 1ª estratégia

```
...
       <many-to-one name="endereco"
                class="br.com.jeebrasil.dominio.Endereco"
                column="ID_ENDERECO"
                cascade="save-update" unique="true"/>
```

```
...
```

Veja que o atributo `cascade` foi definido como `save-update` o que implica em o objeto `Endereco` ser inserido ou atualizado automaticamente quando o objeto Universidade for inserido ou atualizado. Nesse mapeamento aparece o atributo `unique`, que quando assume valor `true` implica em ter apenas uma universidade por endereço.

A outra abordagem é ao invés de inserir o código da Tabela 39 no mapeamento de `Universidade.hbm.xml`, inserir o código mostrado na Tabela 40. A tag `<one-to-one>` define o relacionamento 1-1 que a classe `Universidade` tem com a classe `Endereco`. Os atributos desta tag não são novidades.

Tabela 40 - Relacionamento 1-1 em Universidade.hbm.xml: 2ª estratégia

```
...
       <one-to-one name="endereco"
                class="br.com.jeebrasil.dominio.Endereco"
                cascade="save-update"/>
...
```

Para utilizar a tag `<one-to-one>` no arquivo `Universidade.hbm.xml`, o mapeamento do endereço em `Endereco.hbm.xml` deve ser feito como mostrado na Tabela 41.

Tabela 41 - Relacionamento 1-1 em Endereco.hbm.xml: 2ª estratégia

```
...
<hibernate-mapping>
    <class name="br.com.jeebrasil.dominio.Endereco"  table="ENDERECO">
        <id name="id" column="ID_UNIVERSIDADE" type="int">
            <generator class="foreign">
                <param name="property">universidade</param>
            </generator>
        </id>
        <property name="rua"/>
        <property name="numero"/>
        <property name="bairro"/>
        <property name="cidade"/>
        <property name="uf"/>
        <property name="cep"/>
```

```
        <one-to-one name="universidade"
              class="br.com.jeebrasil.dominio.Universidade"
              constrained="true"/>
    </class>
</hibernate-mapping>
```

A primeira diferença para o mapeamento anterior da entidade `Endereco` é o valor do atributo `class` da tag `generator` agora é `foreign`. A tag `param` de `name` igual a `property` permite a associação 1-1 da classe `Endereco` com a classe `Universidade`, o que acontece é que um parâmetro é passado para a classe geradora do identificador, que neste caso é a propriedade `universidade`, que é da classe que se relaciona 1-1 com `Endereco`. A garantia de que um endereço pertença a uma única universidade vem do fato da chave primária de `Endereco (ID_UNIVERSIDADE)` ser também a chave estrangeira que liga a `Universidade`.

O mapeamento 1-1 em `Endereco` de `Universidade` também é feito utilizando a tag `<one-to-one>`. A única novidade da tag é o atributo `constrained` que sendo igual a true, implica em existir uma relação entre a chave primária de `Endereco` e de `Universidade`, informando ao *Hibernate* que um endereço não pode existir sem que uma universidade exista.

12. Coleções

O mapeamento de coleções já foi apresentado no contexto dos relacionamentos. Essa seção está destinada a apresentar o mapeamento dos quatro tipos de coleções existentes: *Set*, *Bag*, *List* e *Map*.

12.1 Set

Um `Set` é um conjunto que contém elementos únicos. Como já citado, o seu mapeamento é feito através da tag `set` e possui o conjunto de atributos mostrado na Tabela 42. Esses atributos são semelhantes para os demais tipos de coleções a serem apresentadas.

Tabela 42 - Coleções: Atributos

```
<set
   name="nomePropriedade”
   table="nomeTabela"
   schema="nomeEsquema"
   lazy="true|false"
```

```
   inverse="true|false"
   cascade="all|delete|save-update|delete|delete-orphan|all-delete-
            orphan"
   sort="unsorted|natural|comparatorClass"
   order-by="nome_coluna asc|desc"
   where="condição sql arbitrária">
      <key .../>
      <element .../> | <one-to-many .../> | <many-to-many .../>
</set>
```

Dos atributos ainda não apresentados neste material estão:

- `schema`: indica o nome do esquema do banco de dados;
- `sort`: a ordenação pode ser feita na classe Java.
  - `unsorted`: desordenada
  - `natural`: odenada a partir do método `equals(...);`
  - `comparatorClass`: ordenada a partir do método `compareTo();`
- `order-by`: ordenação feita no banco de dados a partir do nome da coluna informada. Pode ser ascendente ou descendente;
- `where`: para informar uma condição arbitrária para a busca da coleção.
- `element`: indica que a coleção é formada por um tipo primitivo.

Exemplos de `Set` já foram apresentados nos mapeamentos dos relacionamentos. Para exemplificar o mapeamento de um `Set` de tipos primitivos considere o mapeamento de uma coleção de `String`'s para armazenar os telefones de um `Aluno`, como visto na Figura 11. Então no mapeamento `Aluno.hbm.xml` deve ser inserido o mapeamento mostrado na Tabela 43.

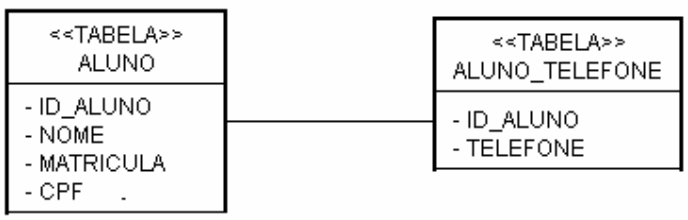

Figura 11 - Coleção de telefones para Aluno: Set

Os telefones do aluno são armazenados na tabela `ALUNO_TELEFONE`. Para o

banco de dados está tabela é separada da tabela `ALUNO`, mas para o *Hibernate*, ele cria a ilusão de uma única entidade. A tag `key` indica a chave estrangeira da entidade pai (`ID_ALUNO`). A tag `element` indica o tipo da coleção (`String`) e a coluna na qual está a informação.

Tabela 43 - Coleções: Set

```
...
<set name="telefones" lazy="true" table="ALUNO_TELEFONE">
   <key column="ID_ALUNO"/>
   <element type="java.lang.String"
          column="TELEFONE" not-null="true"/>
</set>
...
```

Um `Set` não pode apresentar elementos duplicados, portanto a chave primária de `ALUNO_TELEFONE` consiste nas colunas `ID_ALUNO` e `TELEFONE`.

Dado o mapeamento da coleção de telefones de `Aluno`, observe o exemplo mostrado na Tabela 44. Neste exemplo, cria-se um objeto `Aluno` que terá o seu nome definido e um único número de telefone inserido em sua coleção de `String's`. Por fim, o objeto `Aluno` é persistido. O resultado é mostrado na Tabela 45, onde primeiro é inserida uma linha na tabela `ALUNO` e posteriormente uma linha na tabela `ALUNO_TELEFONE`.

Tabela 44 - Exemplo de Mapeamento de Set

```
Aluno aluno = new Aluno();
aluno.setNome("João Maria Costa Neto");
aluno.setTelefones(new HashSet<String>());
aluno.getTelefones().add("32222529");

session.save(aluno);
```

Tabela 45 - Resultado da execução do código presente na Tabela 44

```
Hibernate: insert into ALUNO (MATRICULA, NOME, CPF,
        ID_CURSO, ID_ALUNO) values (?, ?, ?, ?, ?)
Hibernate: insert into ALUNO_TELEFONE (ID_ALUNO,
        TELEFONE) values (?, ?)
```

12.2 List

Um `List` é uma coleção ordenada que pode conter elementos duplicados. O mapeamento de uma lista requer a inclusão de uma coluna de índice na tabela do banco de dados. A coluna índice define a posição do elemento na coleção, como visto na Figura 12 no mapeamento da coleção de telefones da entidade Aluno como sendo um `List`. Dessa maneira, o *Hibernate* pode preservar a ordenação da coleção quando recuperada do banco de dados e for mapeada como um `List`. Observe o mapeamento da coleção telefones na Tabela 46.

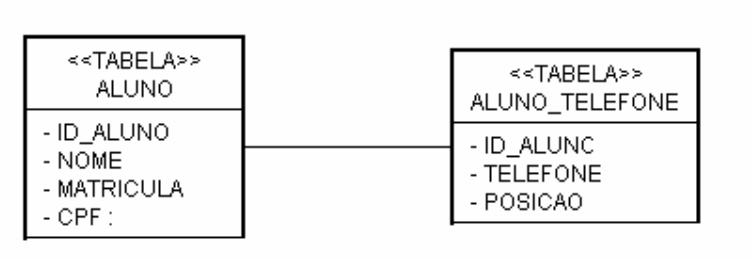

Figura 12 - Coleção de telefones para Aluno: List

Tabela 46 - Coleções: List

```
...
<list name="telefones" lazy="true" table="ALUNO_TELEFONE">
   <key column="ID_ALUNO"/>
   <index column="POSICAO"/>
   <element type="java.lang.String"
          column="TELEFONE" not-null="true"/>
</list >
...
```

Mapeando a coleção como um `List`, a chave primária da tabela `ALUNO_TELEFONE` passa a ser as colunas `ID_ALUNO` e `POSICAO`, permitindo a presença de telefones (`TELEFONE`) duplicados na coleção.

12.3 Map

`Maps` associam chaves aos valores e não podem conter chaves duplicadas. Eles diferem de Sets no fato de que `Maps` contêm chaves e valores, ao passo que `Sets` contêm somente a chave.

O mapeamento de um `Map` é semelhante ao de um `List`, onde o índice de posição passa a ser a chave. Veja a Figura 12. A Tabela 47 apresenta a coleção de

telefones sendo mapeada como um `Map`. A chave primária da tabela `ALUNO_TELEFONE` passa a ser as colunas `ID_ALUNO` e `CHAVE`, também permitindo a presença de telefones duplicados na coleção.

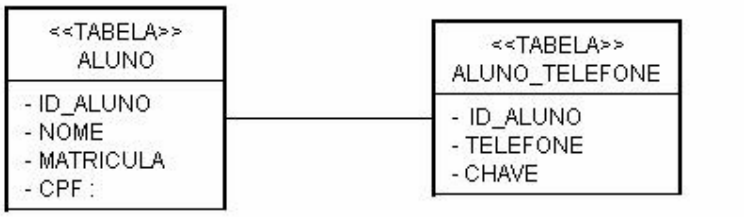

Figura 13 - Coleção de telefones para Aluno: Map

Tabela 47 - Coleções: Map

```
...
<map name="telefones" lazy="true" table="ALUNO_TELEFONE">
   <key column="ID_ALUNO"/>
   <index column="CHAVE" type="java.lang.String"/>
   <element type="java.lang.String"
          column="TELEFONE" not-null="true"/>
</map>
...
```

12.4 Bag

Um `Bag` consiste em uma coleção desordenada que permite elementos duplicados. Em Java não há implementação de um `Bag`, contudo o *Hibernate* fornece um mecanismo de que um `List` em Java simule o comportamento de um `Bag`. Pela definição de um `List`, uma lista é uma coleção ordenada, contudo o *Hibernate* não preserva a ordenação quando um `List` é persistido com a semântica de um `Bag`. Para usar o `Bag`, a coleção de telefones deve ser definida com o tipo `List`. Um exemplo de mapeamento de um `Bag` pode ser visto na Figura 14 e Tabela 48.

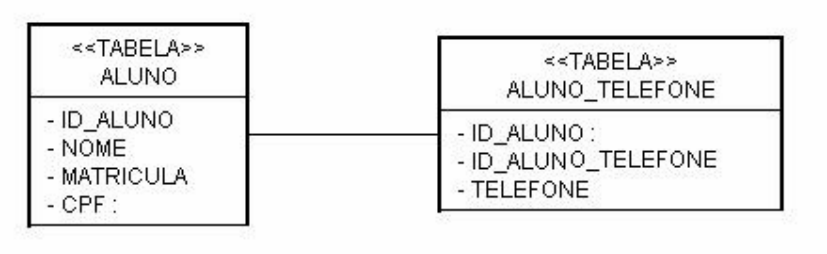

Figura 14 - Coleção de telefones para Aluno: Bag

Para simular o `Bag` como `List`, a chave primária não deve ter associação com a posição do elemento na tabela, assim ao invés da tag `index`, utiliza-se a tag `collection-id`, onde uma chave substituta diferente é atribuída a cada linha da tabela na coleção. Entretanto, o *Hibernate* não fornece nenhum mecanismo para descobrir o valor chave substituta de uma linha em particular.

Tabela 48 - Coleções: Bag

```
...
<idbag name="telefones" lazy="true" table="ALUNO_TELEFONE">
   <collection-id type="long" column="ID_ALUNO_TELEFONE">
      <generator class="sequence"/>
   </collection-id>
   <key column="ID_ALUNO"/>
   <element type="java.lang.String"
          column="TELEFONE" not-null="true"/>
</idbag>
...
```

13. Herança

O *Hibernate* fornece vários mecanismos de se realizar o mapeamento de uma relação de herança:

- Tabela por classe concreta: cada classe concreta é mapeada para uma tabela diferente no banco de dados;
- Tabela por Hierarquia: todas as classes são mapeadas em uma única tabela;
- Tabela por Sub-Classe: mapeia cada tabela, inclusive a classe pai, para tabelas diferentes.

Observe o exemplo de herança apresentado na Figura 15. Neste caso, as classes `Aluno` e `Professor` herdam da classe Pessoa, ou seja, são seus filhos.

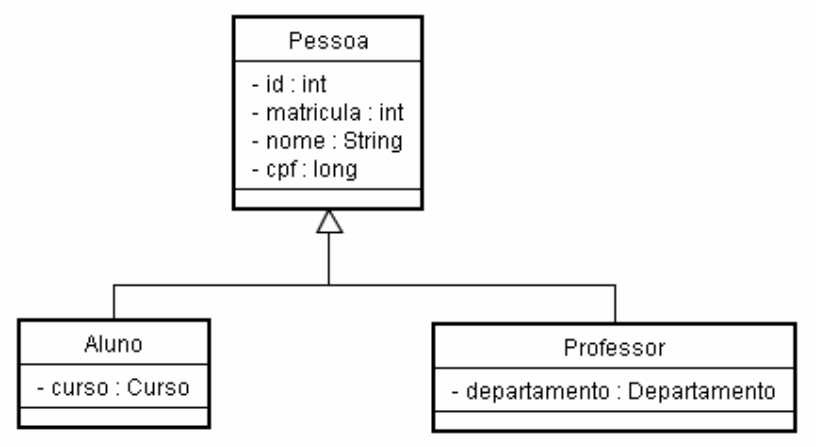

Figura 15 – Herança

As Figuras Figura 16, Figura 17 e Figura 18 mostram as tabelas que devem ser criadas para as estratégias de mapeamento tabela por classe concreta, tabela por hierarquia e tabela por sub-classe, respectivamente.

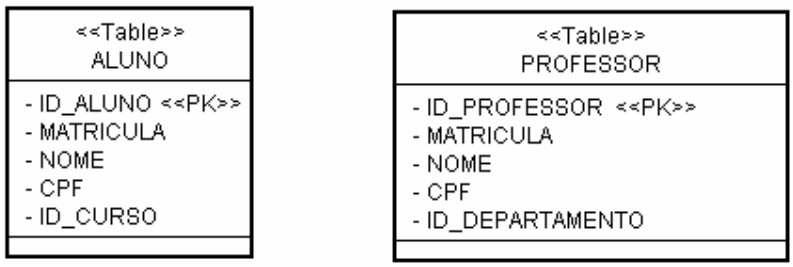

Figura 16 - Tabela por Classe Concreta

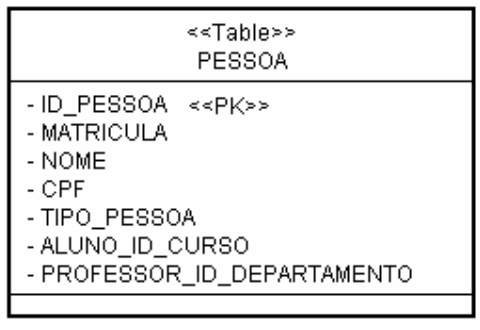

Figura 17 -Tabela por Hierarquia

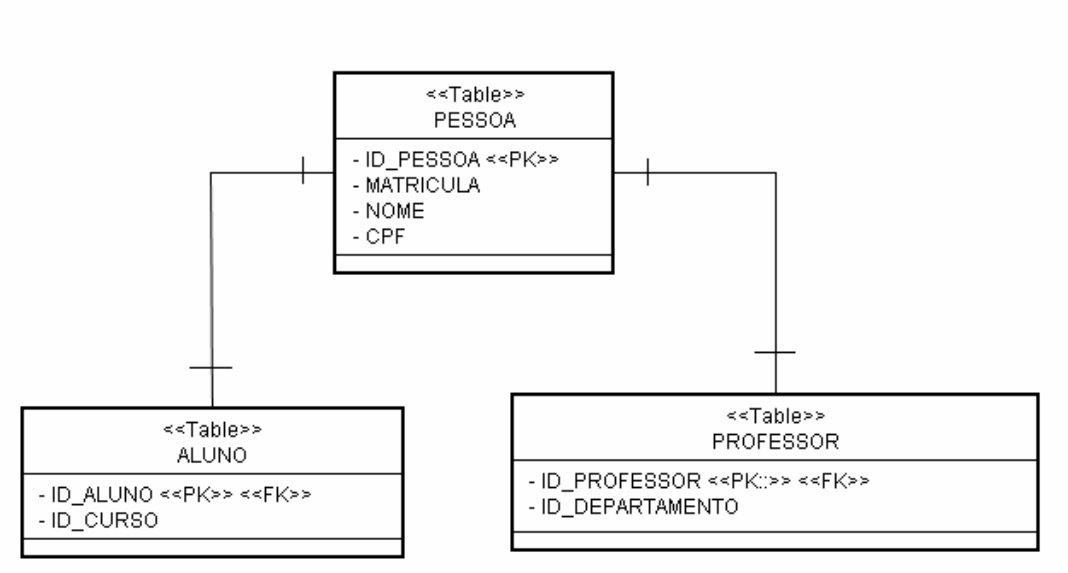

Figura 18 - Tabela por Sub-Classe

Em relação à estratégia tabela por classe concreta, o mapeamento deve ser feito aplicando os conhecimentos já estudados, pois existem duas tabelas independentes. O principal problema dessa estratégia é que não suporta muito bem associações polimórficas. Em banco de dados, as associações são feitas através de relacionamentos de chave estrangeira. Neste caso, as sub-classes são mapeadas em tabelas diferentes, portanto uma associação polimórfica para a classe mãe não seria possível através de chave estrangeira. Outro problema conceitual é que várias colunas diferentes de tabelas distintas compartilham da mesma semântica, podendo tornar a evolução do esquema mais complexo, por exemplo, a mudança de um tipo de uma coluna da classe mãe implica em mudanças nas várias tabelas mapeadas.

Em relação à tabela por hierarquia, o mapeamento deve ser feito como mostrado na Tabela 49. O mapeamento da classe `Pessoa` é feito para a tabela `PESSOA`. Para haver a distinção entre as três classes (`Pessoa`, `Aluno` e `Professor`) surge uma coluna especial (*discriminator*). Essa coluna não é uma propriedade da classe persistente, mas apenas usada internamente pelo *Hibernate*. No caso, a coluna *discriminator* é a `TIPO_PESSOA` e neste exemplo pode assumir os valores 1 e 2. Esses valores são atribuídos automaticamente pelo *Hibernate*.

Cada sub-classe tem uma tag `<subclass>`, onde suas propriedades devem ser mapeadas.

Tabela 49 - Mapeamento Herança: Tabela por Hierarquia

```
...
<hibernate-mapping>

    <class name="br.com.jeebrasil.dominio.Pessoa"  table="PESSOA"
           discriminator-value="0">

        <id name="id" column="ID_PESSOA" type="int">
            <generator class="sequence">
                  <param name="sequence">pessoa_seq</param>
            </generator>
        </id>

        <!-- Coluna Discriminante -->
        <discriminator column="TIPO_PESSOA" type="int"/>

        <!-- Propriedades comuns -->
        <property name="matricula"/>
        <property name="nome"/>
        <property name="cpf"/>

       <!-- Sub-Classes -->
       <subclass name="br.com.jeebrasil.dominio.Aluno"
                 discriminator-value="1">
         <many-to-one name="curso" column="ID_CURSO"
              class="br.com.jeebrasil.dominio.Curso"/>
      </subclass>

      <subclass name="br.com.jeebrasil.dominio.Professor"
                discriminator-value="2">
         <many-to-one name="departamento" column="ID_DEPARTAMENTO"
              class="br.com.jeebrasil.dominio.Departamento"/>
      </subclass>

    </class>
</hibernate-mapping>
```

A estratégia tabela por hierarquia é bastante simples e apresenta o melhor desempenho na representação do polimorfismo. É importante saber que restrições não nulas não são permitidas para o mapeamento de propriedades das sub-classes,

pois esse mesmo atributo para uma outra sub-classe será nulo.

A terceira estratégia, como já citada, consiste em mapear cada classe em uma tabela diferente. Para o exemplo citado, essa estratégia de mapeamento pode ser vista na Tabela 50. Nessa estratégia as tabelas filhas contêm apenas colunas que não são herdadas e suas chaves primárias são também chaves estrangeiras para a tabela mãe.

Tabela 50 - Mapeamento Herança: Tabela por Sub-Classe

```
...
<hibernate-mapping>
    <class name="br.com.jeebrasil.dominio.Pessoa"  table="PESSOA">
        <id name="id" column="ID_PESSOA" type="int">
            <generator class="increment"/>
        </id>
        <!-- Propriedades comuns -->
        <property name="matricula"/>
        <property name="nome"/>
        <property name="cpf"/>

     <!--Sub-Classes -->
     <joined-subclass name="br.com.jeebrasil.dominio.Aluno"
table="ALUNO">
         <key column="ID_ALUNO"/>
         <many-to-one name="curso" column="ID_CURSO"
                      class="br.com.jeebrasil.dominio.Curso"/>
     </joined-subclass>
     <joined-subclass name="br.com.jeebrasil.dominio.Professor"
                      table="PROFESSOR">
         <key column="ID_PROFESSOR"/>
         <many-to-one name="departamento" column="ID_DEPARTAMENTO"
              class="br.com.jeebrasil.dominio.Departamento"/>
     </joined-subclass>
    </class>
</hibernate-mapping>
```

Neste caso, por exemplo, se um objeto da classe `Aluno` é persistido, os valores das propriedades da classe mãe são persistidos em uma linha da tabela `PESSOA` e apenas os valores correspondentes à classe `Aluno` são persistidos em uma linha da tabela `ALUNO`. Em momentos posteriores essa instância de aluno persistida pode ser recuperada de um *join* entre a tabela filha e a tabela mãe. Uma

grande vantagem dessa estratégia é que o modelo de relacionamento é totalmente normalizado.

Observe que no mapeamento, o Hibernate utiliza a tag `<joined-class>` para realizar o mapeamento por sub-classe. A tag `key` declara as chaves primárias das sub-classes que são também chaves estrangeiras para a chave primária da classe mãe.

14. Transações

Uma transação é uma unidade de execução indivisível (ou atômica). Isso significa dizer que todas as etapas pertencentes a uma transação são completamente finalizadas ou nenhuma delas termina.

Para exemplificar o conceito de transações, considere um exemplo clássico de transferência entre contas bancárias: transferir R$ 150,00 da conta corrente do cliente A para a conta corrente do cliente B. Basicamente, as operações que compõem a transação são:

1º) Debitar R$ 150,00 da conta corrente do cliente A
2º) Creditar R$ 150,00 na conta corrente do cliente B

Para a efetivação da transação descrita acima, seria necessário seguir os seguintes passos:

1º) Ler o saldo da conta corrente A (xA)
2º) Calcular o débito de R$ 150,00 da conta corrente A (dA = xA – 150,00)
3º) Gravar na base de dados o novo saldo da conta corrente A (xA = dA)
4º) Ler o saldo da conta corrente B (xB)
5º) Calcular o crédito de R$ 150,00 na conta corrente B (dB = xB + 150,00)
6º) Gravar na base de dados o novo saldo da conta corrente B (xB = dB)

Caso ocorra algum problema (por exemplo: falta de energia, falha no computador, falha no programa, etc.), a execução dos passos anteriores pode ser interrompida. Se, por exemplo, houvesse interrupção logo após a execução do 3º passo, a conta A teria um débito de R$ 150,00 e ainda não teria sido creditado R$ 150,00 na conta corrente B. Neste caso, o banco de dados estaria em um estado inconsistente, afinal, R$ 150,00 teriam "sumido" da conta A sem destino. Dessa maneira, é de suma importância garantir que esses seis passos sejam totalmente executados. Caso haja alguma falha antes da conclusão do último passo, deve-se

garantir também que os passos já executados serão desfeitos, de forma que ou todos os passos são executados ou todos os passos não são executados. Para garantir a consistência do banco, esses seis passos devem ser executados dentro de uma transação, já que ela é uma unidade de execução atômica.

Resumindo, uma transação garante que a seqüência de operações dentro da mesma seja executada de forma única, ou seja, na ocorrência de erro em alguma das operações dentro da transação todas as operações realizadas desde o início podem ser revertidas e as alterações no banco de dados desfeitas, garantindo assim, a unicidade do processo. A transação pode ter dois fins: *commit* ou *rollback*.

Quando a transação sofre *commit*, todas as modificações nos dados realizadas pelas operações presentes na transação são salvas. Quando a transação sofre *rollback*, todas as modificações nos dados realizadas pelas operações presentes na transação são desfeitas.

Para que um banco de dados garanta a integridade dos seus dados deve possuir quatro características, conhecidas como **ACID:**

- **Atomicidade:** o banco de dados deve garantir que todas as transações sejam indivisíveis.
- **Consistência:** após a execução de uma transação, o banco de dados deve continuar consistente, ou seja, deve continuar com um estado válido.
- **Isolamento:** mesmo que várias transações ocorram paralelamente (ou concorrentemente), nenhuma transação deve influenciar nas outras. Resultados parciais de uma transação não devem ser "vistos" por outras transações executadas concorrentemente.
- **Durabilidade:** após a finalização de uma transação, todas as alterações feitas por ela no banco de dados devem ser duráveis, mesmo havendo falhas no sistema após a sua finalização.

14.1 Modelos de Transações

As definições do início de uma transação, de seu fim e das ações que devem ser tomadas na ocorrência de falhas são feitas através de um modelo de transação. Existem diversos modelos encontrados na literatura. Nesta seção serão abordados apenas quatro: *Flat Transactions*, *Nested Transactions*, *Chained Transactions* e *Join Transactions*.

- ***Flat Transaction.*** Modelo mais utilizado pela maioria dos Sistemas Gerenciadores de Banco de Dados (SGBD) e Gerenciadores de Transações. Conhecida como modelo de transações planas por apresentar uma única camada de controle, ou seja, todas as operações dentro da transação são tratadas como uma única unidade de trabalho.

- ***Nested Transaction.*** Este modelo, também conhecido como Modelo de Transações Aninhadas, possibilita que uma transação possa ser formada por várias sub-transações. Em outras palavras, uma única transação pode ser dividida em diversas unidades de trabalho, com cada unidade operando independente uma das outras. A propriedade de atomicidade é válida para as sub-transações. Além disso, uma transação não pode ser validada até que todas as suas sub-transações tenham sido finalizadas. Se uma transação for interrompida, todas as suas sub-transações também serão. O contrário não é verdadeiro, já que se uma sub-transação for abortada a transação que a engloba pode: ignorar o erro; desfazer a sub-transação; iniciar uma outra sub-transação.

- ***Chained Transaction***. Também conhecido como Modelo de Transações Encadeadas, esse modelo tem como objetivo desfazer as operações de uma transação em caso de erro com a menor perda de trabalho possível. Uma transação encadeada consiste em um conjunto de sub-transações executadas seqüencialmente, em que à medida que as sub-transações vão sendo executadas, são validadas e não podem mais ser desfeitas. Os resultados do conjunto de transações só serão visíveis ao final da execução de todas elas.

- ***Join Transaction***. Esse modelo permite que duas transações sejam unidas em um só, de forma que todos os recursos passam a ser compartilhados.

14.2 Transações e Banco de Dados

Uma transação de banco de dados é formada por um conjunto de operações que manipulam os dados. A atomicidade de uma transação é garantida por duas operações: *commit* e *rollback*.

Os limites das operações de uma transação devem ser demarcados. Assim, é possível saber a partir de qual operação a transação é iniciada e em qual operação ela finalizada. Ao final da execução da última operação que pertence à transação, todas as alterações no banco de dados realizadas pelas operações que

compõe a transação devem ser confirmadas, ou seja, um *commit* é realizado. Se houver algum erro durante a execução de algumas das suas operações, todas as operações da transação que já foram executadas devem ser desfeitas, ou seja, um *rollback* é realizado. A Figura 19 ilustra esses conceitos.

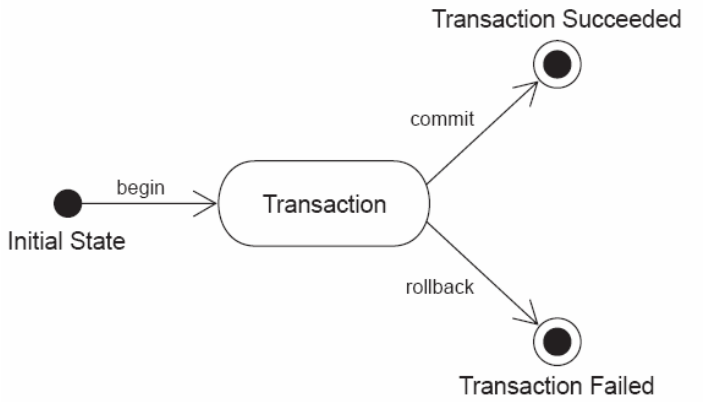

Figura 19 - Estados do sistema durante uma transação

14.3 Ambientes Gerenciados e Não Gerenciados

As seções seguintes referem-se às definições dos conceitos relacionados a transações JDBC e JTA, onde aparecem os termos ambientes gerenciados e não gerenciados. Esta seção destina-se a explicar sucintamente o que são esses termos.

Os ambientes gerenciados são aqueles caracterizados pela gerência automática de transações realizadas por algum container. Exemplos de ambientes gerenciados são componentes EJB (*Enteprise JavaBeans*) executando em servidores de aplicações (*JBoss*, *Geronimo*, etc). Já os ambientes não gerenciados são cenários onde não há nenhuma gerência de transação, como por exemplo: *Servlets*, aplicações *desktop*, etc..

14.4 Transações JDBC

A tecnologia JDBC (*Java Database Connectivity*) é um conjunto de classes e interfaces escritas em Java, ou API, que realiza o envio de instruções SQL (*Structured Query Language*) para qualquer banco de dados relacional.

Uma transação JDBC é controlada pelo gerenciador de transações SGBD e geralmente é utilizada por ambientes não gerenciados. Utilizando um *driver* JDBC, o início de uma transação é feito implicitamente pelo mesmo. Embora alguns bancos de dados necessitem invocar uma sentença "**begin transaction**"

explicitamente, com a API JDBC não é preciso fazer isso. Uma transação é finalizada após a chamada do método `commit()`. Caso algo aconteça de errado, para desfazer o que foi feito dentro de uma transação, basta chamar o método `rollback()`. Ambos, `commit()` e `rollback()`, são invocados a partir da conexão JDBC.

A conexão JDBC possui um atributo `auto-commit` que especifica quando a transação será finalizada. Se este atributo for definido como `true`, ou seja, se na conexão JDBC for invocado `setAutoCommit(true)`, ativa-se o modo de auto *commit*. O modo auto *commit* significa que para cada instrução SQL uma nova transação é criada e o *commit* é realizado imediatamente após a execução e finalização da mesma, não havendo a necessidade de após cada transação invocar explicitamente o método `commit()`.

Em alguns casos, uma transação pode envolver o armazenamento de dados em vários bancos de dados. Nessas situações, o uso apenas do JDBC pode não garantir a atomicidade. Dessa maneira, é necessário um gerenciador de transações com suporte a transações distribuídas. A comunicação com esse gerenciador de transações é feita usando JTA (*Java Transaction API*).

14.5 Transações JTA

As transações JTA são usadas em um ambiente gerenciável, onde existem transações CMT (*Container Managed Transactions*). Neste tipo de transação não há a necessidade de programação explícita das delimitações das transações, esta tarefa é realizada automaticamente pelo próprio container. Para isso, é necessário informar nos descritores dos EJBs a necessidade de suporte transacional às operações e como ele deve gerenciá-lo.

O gerenciamento de transações é feito pelo Hibernate a partir da interface `Transaction`.

14.6 API para Transações do Hibernate

A interface `Transaction` fornece métodos para a declaração dos limites de uma transação. A Tabela 51 apresenta um exemplo de uso de transações com a interface `Transaction`.

A transação é iniciada a partir da invocação ao método `session.beginTransaction()`. No caso de um ambiente não gerenciado, uma transação JDBC na conexão JDBC é iniciada. Já no caso de um ambiente

gerenciado, uma nova transação JTA é criada, caso não exista nenhuma já criada. Casa já existe uma transação JTA, essa nova transação une-se a existente.

A chamada ao método `tx.commit()` faz com que os dados em memória sejam sincronizados com a base de dados. O *Hibernate* só realiza efetivamente o *commit* se o comando `beginTransaction()` iniciar uma nova transação (em ambos ambientes gerenciado ou não gerenciado). Se o `beginTransaction()` não iniciar uma nova transação (no caso de transações JTA isso é possível), então o estado em sessão é apenas sincronizado com o banco de dados e a finalização da transação é feita de acordo com a primeira parte do código fonte que a criou.

Se ocorrer algum erro durante a execução do método `acaoExecutada()`, o método `tx.rollback()` é executado, desfazendo o que foi feito até o momento em que o erro ocorreu.

Observa-se que no final do código a sessão é finalizada a partir do comando `session.close()`, liberando a conexão JDBC e devolvendo-a para o pool de conexões.

Tabela 51 - Usando a Interface `Transaction` do Hibernate

```
Session session = sessions.openSession();
Transaction tx = null;
try {
      tx = session.beginTransaction();
      acaoExecutada();
      tx.commit();
} catch (Exception e) {
      if (tx != null) {
            try {
                  tx.rollback();
            } catch (HibernateException he) {
                  //log he and rethrow e
            }
      }
} finally {

      try {
            session.close();
      } catch (HibernateException he) {
            throw he;
      }
}
```

14.7 Flushing

Flushing é o processo de sincronizar os dados em sessão (ou em memória) com o banco de dados. As mudanças nos objetos de domínio em memória feitas dentro do escopo de uma sessão (`Session`) não são imediatamente propagadas para o banco de dados. Isso permite ao *Hibernate* unir um conjunto de alterações e fazer um número mínimo de interações com o banco de dados, ajudando a minimizar a latência na rede.

A operação de *flushing* ocorre apenas em três situações: quando é dado *commit* na transação, algumas vezes antes de uma consulta ser executada (em situações que alterações podem influenciar em seu resultado) e quando o método `Session.flush()` é invocado.

O *HIbernate* possui um modo *flush* que pode ser definido a partir do comando `session.setFlushMode()`. Este modo pode assumir os seguintes valores:

- **`FlushMode.AUTO:`** valor padrão. Faz com que o *Hibernate* não realize o processo de *flushing* antes de todas as consultas, somente realizará se as mudanças dentro da transação alterar seu resultado.
- **`FlushMode.COMMIT:`** especifica que os estados dos objetos em memória somente serão sincronizados com a base de dados ao final da transação, ou seja, quando o método `commit`() é chamado.
- **`FlushMode.NEVER:`** especifica que a sincronização só será realizado diante da chamada explícita ao método `flush()`.
-

14.8 Níveis de Isolamento de uma Transação

As bases de dados tentam assegurar que uma transação ocorra de forma isolada, ou seja, mesmo que estejam acontecendo outras transações simultaneamente, é como se ela estivesse ocorrendo sozinha.

O nível de isolamento de uma transação especifica que dados estão visíveis a uma sentença dentro de uma transação. Eles impactam diretamente no nível de acesso concorrente a um mesmo alvo no banco de dados por transações diferentes.

Geralmente, o isolamento de transações é feito usando *locking*, que significa que uma transação pode bloquear temporariamente um dado para que outras transações não o acessem no momento que ela o está utilizando. Muitos bancos de dados implementam o nível de isolamento de uma transação através do modelo de controle concorrente multi-versões (MCC – *Multiversion Concorrency Control*).

Dentre alguns fenômenos que podem ocorrer devido à quebra de isolamento de uma transação estão três:

- ***Dirty Read*** (Leitura Suja): uma transação tem acesso a dados modificados por uma outra transação ainda não finalizada que ocorre concorrentemente. Isso pode causar problema, pois pode ocorrer um erro dentro da transação que está modificando os dados e as suas alterações serem desfeitas antes de confirmadas, então é possível que a transação que acessa os dados já modificados esteja trabalhando se baseando em dados incorretos.

- ***Nonrepeatable Read*** (Leitura que não pode ser repetida): uma transação lê mais de uma vez um mesmo dado e constata que há valores distintos em cada leitura. Por exemplo, uma transação **A** lê uma linha do banco; uma transação **B** modifica essa mesma linha e é finalizada (*commit*) antes que a transação **A**; a transação **A** lê novamente esta linha e obtém dados diferentes.

- ***Phantom Read*** (Leitura Fantasma): em uma mesma transação uma consulta pode ser executada mais de uma vez e retornar resultados diferentes. Isso pode ocorrer devido a uma outra transação realizar mudanças que afetem os dados consultados. Por exemplo, uma transação **A** lê todas as linhas que satisfazem uma condição **WHERE**; uma transação **B** insere uma nova linha que satisfaz a mesma condição antes da transação **A** ter sido finalizada; a transação **A** reavalia a condição **WHERE** e encontra uma linha "fantasma" na mesma consulta feita anteriormente.

Existem quatro níveis de isolamento da transação em SQL. Eles se diferenciam de acordo com a ocorrência ou não dos fenômenos anteriormente descritos, como mostrado na Tabela 52.

Tabela 52 - Níveis de Isolamento da Transação em SQL

Nível de Isolamento	Dirty Read	Nonrepeatable	Phanton Read
Read Uncommited	SIM	SIM	SIM
Read Commited	NÃO	SIM	SIM
Repeatable Read	NÃO	NÃO	SIM
Serializable	NÃO	NÃO	NÃO

A escolha do nível de isolamento *Read Uncommited* não é recomendada

para banco de dados relacionais, já que permite ler inconsistências e informações parciais (mudanças realizadas por uma transação ainda não finalizada podem ser lidas por outra transação). Se a primeira transação não for concluída, mudanças na base de dados realizadas pela segunda transação podem deixá-la com um estado inconsistente.

Com o nível *Read Commited*, uma transação somente visualiza mudanças feitas por outras transações quando confirmadas, permitindo que transações só acessem estados consistentes do banco. No caso de uma atualização/exclusão de uma linha de alguma tabela por uma transação, pode ser que a mesma tenha acabado de ser modificada por uma transação concorrente. Nesta situação, a transação que pretende atualizar fica esperando a transação de atualização que iniciou primeiro ser efetivada ou desfeita. Se as atualizações da primeira transação forem desfeitas, seus efeitos serão desconsiderados e a segunda transação efetivará suas mudanças considerando a linha da tabela anteriormente lida. Caso contrário, a segunda transação irá ignorar a atualização caso a linha tenha sido excluída ou aplicará a sua atualização na versão atualizada da linha.

O nível *Repeatable Read* não permite que uma transação sobrescreva os dados alterados por uma transação concorrente. Uma transação pode obter uma imagem completa da base de dados quando iniciada. Este nível é ideal para a geração de relatórios, pois em uma mesma transação, um registro é lido diversas vezes e seu valor se mantém o mesmo até que a própria transação altere seu valor.

Em relação ao nível *Serializable*, ele fornece o nível de isolamento de transação mais rigoroso. Ele permite uma execução serial das transações, como se todas as transações fossem executadas uma atrás da outra. Dessa forma, pode-se perder um pouco da performance da aplicação.

14.9 Configurando o nível de isolamento

No *Hibernate* cada nível de isolamento é identificado por um número:

- 1: *Read Uncommited*
- 2: *Read Commited*
- 4: *Repeatable Read*
- 8: *Serializable*

Para configurá-lo basta incluir a linha presente na Tabela 53 no arquivo de configuração *.cfg.xml. Neste exemplo, o nível de isolamento foi definido como

Repeatable Read.

Tabela 53 – Configuração do Nível de Isolamento

```
hibernate.connection.isolation = 4
```

15. Concorrência

Em algumas situações pode acontecer que duas ou mais transações que ocorrem paralelamente leiam e atualizem o mesmo dado. Considerando que duas transações leiam um mesmo dado **x** quase que simultaneamente. Ambas as transações vão manipular esse mesmo dado com operações diferentes e atualizá-lo na base de dados. Para exemplificar, a Tabela 54 apresenta um exemplo de duas transações concorrentes manipulando o mesmo dado **x**.

No primeiro passo, ambas as transações lêem o dado **x** com o mesmo valor (2). Em seguida, T1 soma o valor **x** que leu com 1 e o valor de **x** para T1 passa a ser 3 (2 + 1). Já T2, soma o valor de **x** lido a 3 e **x** passa a ter o valor 5 (2 + 3). Por fim, ambos T1 e T2 gravarão os novos valores de **x** calculados na base de dados, respectivamente. Como não há controle de concorrência de acesso ao dado **x**, o seu valor final corresponderá a 5, ou seja, o valor calculado por T2, significando que as alterações feitas por T1 foram descartadas.

Tabela 54 – Exemplo de Transação Concorrente

```
1) Transação 1 (T1) lê x = 2
2) Transação 2 (T2) lê x = 2
3) T1 faz x = x + 1
4) T2 faz x = x + 3
5) T1 armazena o valor de x na base de dados
6) T2 armazena o valor de x na base de dados
```

Para evitar a situação descrita anteriormente, deve-se controlar o acesso concorrente ao dado, ou seja, deve-se implementar o mecanismo de *Locking*. O gerenciamento de *locking* e da concorrência pode ser feito de duas formas:

- Pessimista: utilizar o controle pessimista significa que se uma transação T1 lê um dado e tem a intenção de atualizá-lo, esse dado será bloqueado

(nenhuma outra transação poderá lê-lo) até que T1 o libere, normalmente após a sua atualização.

- Otimista: utilizar o controle otimista significa que se T1 lê e altera um dado ele não será bloqueado durante o intervalo entre a leitura e atualização. Caso uma outra transação T2 tenha lido esse mesmo dado antes de T1 o atualizá-lo tente alterá-lo em seguida na base de dados, um erro de violação de concorrência deve ser gerado.

15.1 Lock Otimista

Para ilustrar o gerenciamento do tipo otimista, um exemplo é dado a partir das Figura 20 e Figura 21.

O problema é mostrado na Figura 20, onde, inicialmente, duas transações (ilustradas por *Thread 1* e *Thread 2*) acessam um mesmo dado na base de dados (`SELECT Dado`), uma seguida da outra. Logo em seguida, a primeira transação (*Thread 1*) atualiza este dado na base de dados e depois quem também o atualiza é a segunda transação (*Thread 2*). Nesta abordagem otimista, acontece que a atualização do dado feita pela segunda transação sobrescreve a atualização realizada pela primeira, ou seja, a atualização feita pela primeira transação é perdida.

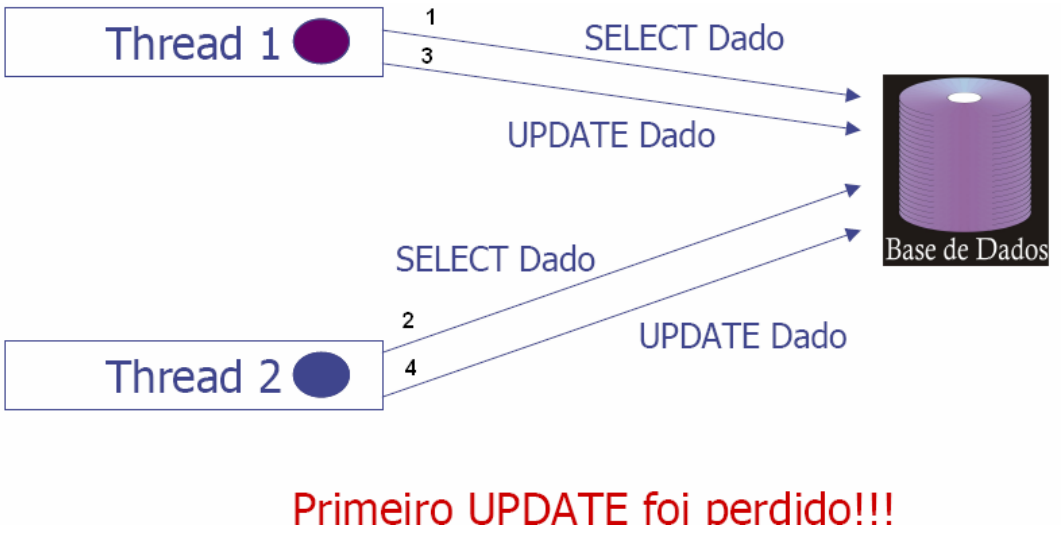

Figura 20 - Loocking Otimista 1

Para resolver o problema descrito anteriormente com a abordagem otimista, pode-se utilizar o conceito de *Version Number*, que é um padrão utilizado para versionar numericamente os dados de uma linha de uma tabela na base de dados.

Por exemplo, na Figura 21, também, inicialmente, duas transações (ilustradas por *App 1* e *App 2*) acessam um mesmo dado na base de dados (`SELECT Dado`), uma seguida da outra. Com isso, esse mesmo dado nas duas transações são rotulados com a versão atual dele na base de dados, no caso, *Versão 1*. Logo em seguida, a segunda transação atualiza este dado. Quando a atualização vai ser feita, é verificado se a versão do dado na transação corresponde à versão dele na base de dados. Nesta primeira atualização, a versão da transação é 1 e a da base de dados também. Como elas são iguais, a atualização é efetivada e a versão do dado na base de dados passa a ser a *Versão 2*. Por fim, a primeira transação vai também atualizar este mesmo dado. Dessa forma, também é feita uma comparação entre as versões do dado na transação e na base de dados. Neste caso, a versão na transação é a 1 e na base de dados é 2, ou seja, as versões não correspondem. Assim, um erro é disparado e a atualização desejada pela primeira transação não é concretizada, evitando que a atualização feita pela segunda transação não seja desfeita.

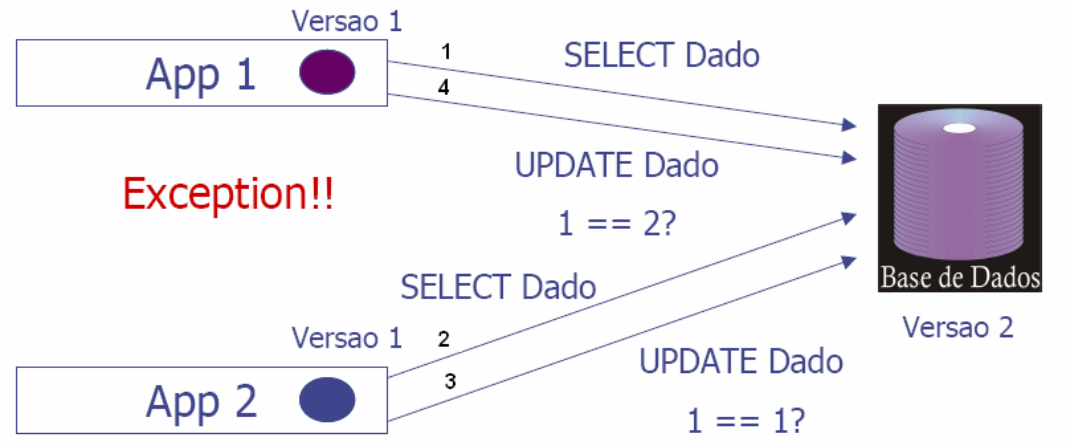

Figura 21 - Loocking Otimista 2

Com o Hibernate, uma forma de utilizar o versionamento dos dados é utilizar o elemento `version` no mapeamento das tabelas. Para exemplificar o seu uso, considera-se a classe `ContaCorrente` na Tabela 55 que será mapeada para a tabela CONTA_CORRENTE na base de dados. Dentre os diversos atributos da classe, está o atributo denominado `versao` que irá justamente guardar a versão atual das linhas da tabela. A tabela `CONTA_CORRENTE` também deve ter uma coluna para onde esse atributo `versao` será mapeado.

Tabela 55 – Classe de Domínio: ContaCorrente 1

```
package br.com.jeebrasil.dominio.ContaCorrente;
public class ContaCorrente{

      //Atributo utilizado para o versionamento
      private int versao;

      //Demais atributos
      private int id;
      private double saldo;
      private Correntista correntista;
      //Outros atributos
      //...

      public int getVersao(){
            return versao;
      }
      public setVersao(int versao){
            this.versao = versao;
      }
      //Demais métodos de acesso e modificação de dados
      //...
}
```

No arquivo de mapeamento da classe `ContaCorrente`, `ContaCorrente.hbm.xml` (ver Tabela 56), o atributo `version` é utilizado para mapear o atributo `versao` da classe `ContaCorrente` para a coluna `VERSAO` da tabela `CONTA_CORRENTE`. Com esse mapeamento, toda vez que uma determinada linha for atualizada na base de dados, a sua coluna `VERSAO` será incrementada de uma unidade, indicando a nova versão dos dados.

Tabela 56 - ContaCorrente.hbm.xml 1

```
<hibernate-mapping>
    <class name="br.com.jeebrasil.dominio.ContaCorrente"
        table="CONTA_CORRENTE">
        <id name="id" column="ID_CONTA" type="int">
            <generator class="sequence">
                  <param name="sequence">conta_seq</param>
            </generator>
```

```
        </id>

        <version name="versao" column="VERSAO"/>

        <property name="saldo" type="double" column="SALDO"/>
        <many-to-one name="correntista"
             class="br.com.jeebrasil.dominio.Correntista"
             column="ID_CORRENTISTA"/>
      <!-- Demais Atributos -->
    </class>
</hibernate-mapping>
```

Outra forma de implementar o *lock* otimista é utilizando o atributo `timestamp` no mapeamento da tabela. Neste caso, a classe `ContaCorrente` do exemplo anterior ao invés de ter um atributo inteiro para guardar a versão do dado, teria uma atributo do tipo `java.util.Date` para guardar o instante no tempo da última atualização. Neste caso, a classe de domínio seria equivalente à mostrada na Tabela 57 e seu mapeamento ao exibido na Tabela 58. Neste exemplo, a tabela `CONTA_CORRENTE` deve conter uma coluna do tipo *timestamp* denominada `DATA_ULTIMA_ATUALIZACAO`.

Tabela 57 – Classe de Domínio: ContaCorrente 2

```
package br.com.jeebrasil.dominio.ContaCorrente;
public class ContaCorrente{

      //Atributo utilizado para o versionamento
      private Date ultimaAtualizacao;

      //Demais atributos
      //...

      public Date getUltimaAtualizacao(){
            return ultimaAtualizacao;
      }
      public setUltimaAtualizacao(Date ultimaAtualizacao){
            this.ultimaAtualizacao = ultimaAtualizacao;
      }
      //Demais métodos de acesso e modificação de dados
      //...
```

```
}
```

Tabela 58 - ContaCorrente.hbm.xml 2

```
<hibernate-mapping>
    <class name="br.com.jeebrasil.dominio.ContaCorrente"
        table="CONTA_CORRENTE">
        <id name="id" column="ID_CONTA" type="int">
            <generator class="sequence">
                  <param name="sequence">conta_seq</param>
            </generator>
        </id>

       <timestamp name="ultimaAtualizacao"
                  column="DATA_ULTIMA_ATUALIZACAO"/>

       <!-- Demais Atributos -->
    </class>
</hibernate-mapping>
```

Se a tabela não possuir uma coluna para guardar a versão do dado ou a data da última atualização, com *Hibernate*, há uma outra forma de implementar o *lock* otimista, porém essa abordagem só deve ser utilizada para objetos que são modificados e atualizados em uma mesma sessão (`Session`). Se este não for o caso, deve-se utilizar uma das duas abordagens citadas anteriormente.

Com essa última abordagem, quando uma determinada linha vai ser atualizada, o *Hibernate* verifica se os dados dessa linha correspondem aos mesmos dados que foi recuperado. Caso afirmativo, a atualização é efetuada. Para isso, no mapeamento da tabela, deve-se incluir o atributo `optimistic-lock` (Tabela 59). Por exemplo, no mapeamento da classe `ContaCorrente`, na tag `class` apareceria este atributo.

Tabela 59 - ContaCorrente.hbm.xml 3

```
<hibernate-mapping>
    <class name="br.com.jeebrasil.dominio.ContaCorrente"
            table="CONTA_CORRENTE"  optimist-lock="all">

        <id name="id" column="ID_CONTA" type="int">
            <generator class="sequence">
                  <param name="sequence">conta_seq</param>
```

```
            </generator>
         </id>

        <property name="saldo" type="double" column="SALDO"/>
        <property name="descricao" type="java.lang.String"
             column="DESCRICAO"/>

        <many-to-one name="correntista"
             class="br.com.jeebrasil.dominio.Correntista"
             column="ID_CORRENTISTA"/>
       <!-- Demais Atributos -->

    </class>
</hibernate-mapping>
```

Dessa maneira, quando uma linha dessa tabela fosse atualizada, o SQL equivalente gerado para a atualização seria o exibido na Tabela 60. Neste exemplo, considera-se a atualização do saldo para R$ 1.500,00 de uma determinada conta de saldo R$ 1.000,00.

Tabela 60 – Exemplo optimist-lock="all"

```
UPDATE CONTA_CORRENTE SET SALDO = 1500
      WHERE ID_CONTA = 104 AND
            SALDO = 1000 AND
            DESCRICAO = "DESCRICAO DA CONTA" AND
            ID_CORRENTISTA = 23
```

15.2 Lock Pessimista

A estratégia de *lock* pessimista para proibir o acesso concorrente a um mesmo dado da base de dados é feita bloqueando o mesmo até que a transação seja finalizada.

Alguns banco de dados, como o *Oracle* e *PostgreSQL*, utilizam a construção SQL `SELECT FOR UPDATE` para bloquear o dado até que o mesmo seja atualizado. O *Hibernate* fornece um conjunto de modos de *lock*(constantes disponíveis na classe `LockMode`) que podem ser utilizados para implementar o *lock* pessimista.

Considerando o exemplo da Tabela 61, onde um determinado aluno é consultado na base de dados e tem seu nome atualizado. Neste caso, não há um bloqueio ao dado, então qualquer outra transação pode acessar este mesmo dado

concorrentemente e modifica-lo, de forma que poderá ocorrer uma incosistência dos dados. Na Tabela 62, há um exemplo de uso do *lock* pessimista para resolver este problema, bastando passar a constante **`LockMode.UPGRADE`** como terceiro argumento do método `get` do objeto `Session`.

Tabela 61 – Transação sem Lock

```
Transaction tx = session.beginTransaction();
Aluno aluno = (Aluno) session.get(Aluno.class, alunoId);
aluno.setNome("Novo Nome");
tx.commit();
```

Tabela 62 – Transação com Lock Pessimista: LockMode.UPGRADE

```
Transaction tx = session.beginTransaction();
Aluno aluno =
      (Aluno) session.get(Aluno.class, alunoId, LockMode.UPGRADE);
aluno.setNome("Novo Nome");
tx.commit();
```

O método `get` do objeto `Session` pode receber como terceiro argumento para implementar o *lock* pessimista as seguintes constantes:

- **Lock.NONE**: Só realiza a consulta ao banco se o objeto não estiver no *cache*[1].
- **Lock.READ**: Ignora os dados no *cache* e faz verificação de versão para assegurar-se de que o objeto em memória é o mesmo que está no banco.
- **Lock.UPDGRADE**: Ignora os dados no *cache*, faz verificação de versão (se aplicável) e obtém *lock* pessimista do banco (se suportado).
- **Lock.UPDGRADE_NOWAIT:** Mesmo que `UPGRADE`, mas desabilita a espera por liberação de *locks*, e dispara uma exceção se o *lock* não puder ser obtido. Caso especial do *Oracle* que utiliza a cláusula `SELECT ... FOR UPDATE NOWAIT` para realizar *locks*.
- **Lock.WRITE**: Obtida automaticamente quando o *Hibernate* realiza alguma inserção ou atualização na base de dados.

[1] O c*ache* é uma técnica comumente utilizada para aprimorar o desempenho da aplicação no que diz respeito ao acesso ao banco de dados. Conceito apresentado na próxima sessão.

16. Caching

O c*ache* é uma técnica comumente utilizada para aprimorar o desempenho da aplicação no que diz respeito ao acesso ao banco de dados. Com o *cache* é possível fazer uma cópia local dos dados, evitando acesso ao banco sempre que a aplicação precise, por exemplo, acessar dados que nunca ou raramente são alterados e dados não críticos. O uso do *cache* não é indicado para manipulação de dados críticos, de dados que mudam freqüentemente ou de dados que são compartilhados com outras aplicações legadas.

Existem três tipos principais de *chache*:

- **Escopo de Transação**: utilizado no escopo da transação, ou seja, cada transação possui seu próprio *cache*. Duas transações diferentes não compartilham o mesmo *cache*.

- **Escopo de Processo**: há o compartilhamento do *cache* entre uma ou mais transações. Os dados no escopo do *cache* de uma transação podem ser acessados por uma outra transação que executa concorrentemente, podendo de implicações relacionadas ao nível de isolamento.

- **Escopo de Cluster**: *cache* compartilhado por vários processos pertencentes a máquinas virtuais distintas e deve ser replicado por todos os nós do *cluster*.

Considerando o *cache* no escopo da transação, se na transação houver mais de uma consulta a dados com mesmas identidades de banco de dados, a mesma instância do objeto Java será retornada.

Pode ser também que o mecanismo de persistência opte por implementar identidade no escopo do processo, de forma que a identidade do objeto seja equivalente à identidade do banco de dados. Assim, se a consulta a dados em transações que executam concorrentemente for feita a partir de identificadores de banco de dados iguais, o resultado também será o mesmo objeto Java. Outra forma de se proceder é retornar os dados em forma de novos objetos. Assim, cada transação teria seu próprio objeto Java representando o mesmo dado no banco.

No escopo de *cluster*, é necessário haver comunicação remota, em que os dados são sempre manipulados por cópias.

Nas situações em que estratégias de MOR permitem que várias transações manipulem uma mesma instância de objeto persistente, é importante ter um controle de concorrência eficiente, por exemplo, bloqueando um dado enquanto ele não é atualizado. Utilizando o Hibernate, ter-se-á um conjunto diferente de instâncias para cada transação, ou seja, tem-se identidade no escopo da transação.

16.1 Arquitetura de Cache com Hibernate

Em construção

17. Busca de Dados

(SEÇÃO A SER MELHORADA)

A busca de dados utilizando *Hibernate* pode se dar através do uso de HQL (*Hibernate Query Languange*), *Criteria*, SLQ Nativo ou *Query By Example*, que serão apresentados neste capítulo.

17.1 SQL Nativo

SQL Nativo deve ser usado em *Hibernate* quando este não prover uma forma para realizar a consulta desejada, como, por exemplo, uma consulta hierárquica, pois *Hibernate* não suporta. Quando isto acontecer, a API JDBC pode ser utilizada diretamente.

17.2 *Hibernate Query Language* - HQL

HQL é uma linguagem de consulta semelhante à OQL (*Object Query Language*), no entanto voltada para bancos de dados relacionais. Pertence a família do SQL, podendo ser facilmente compreendida através de algum conhecimento em SQL.

HQL suporta polimorfismo, herança e associações, pode incluir na consulta, restrições nas propriedades da classe e associações, suporta ainda agregação, ordenação, paginação, subconsultas, *outer joins*, chamadas de funções SQL e projeção, que permite buscar apenas os dados necessários, não sendo necessário carregar todo o objeto e suas associações, a projeção é muito utilizada por relatórios. Como SQL, esta linguagem é utilizada apenas para consulta, isto é, não é uma linguagem de manipulação de dados (DML).

Uma simples consulta usando HQL pode ser realizada através da cláusula *from*, mostrada na Tabela 63, onde temos a consulta à lista de todos os alunos, se desejar usar `Aluno` em outra parte da consulta basta criar um "alias", neste caso usamos `aluno`, a palavra chave `as` pode ser omitida e podemos ter vários *aliases*. Por padronização é indicado usar o modelo Java de variáveis locais.

Tabela 63 – HQL: Cláusula from

```
from Aluno
from Aluno as aluno
from Aluno aluno
from Aluno as aluno, Professor as prof
```

Em HQL, como SQL, há o suporte a operadores matemáticos (+, -, * e /), operadores de comparação (<, >, <>, <=, >=, =, between, not between, in e not in), operadores lógicos (or, and e parênteses para agrupar expressões), o operador LIKE e o símbolo '%' também podem ser utilizados para pesquisas em `String`, além de poder ordenar sua pesquisa em ordem descendente ou ascendente. Vejamos alguns exemplos em HQL na Tabela 64.

Tabela 64 - Expressões HQL

```
from Aluno aluno where aluno.matricula >= 35

from Endereco end where
      ( end.rua in ("Bernardo Vieira", "Prudente de Morais") )
            or ( end.numero between 1 and 100 )

from Professor p where p.nome like "João%"

from Aluno aluno order by aluno.nome asc
```

Com HQL também é possível utilizar *joins* para associar dados de duas ou mais relações. Como por exemplo as tabelas `Curso` e `Aluno`. Para exemplificar os casos de *joins*, serão utilizados os dados abaixo das tabelas `ALUNO` e `CURSO` (Tabela 65 e Tabela 66). As consultas e resultados são apresentados as Tabela 67 e Tabela 68.

Tabela 65 - Dados da tabela ALUNO

id_aluno	nome	matricula	cpf	id_curso
1	João	123	055.352.254-33	1
2	Maria	124	321.055.321-22	2
3	Fernanda	125	254.365.654-36	1
4	André	126	354.254.256-65	1

Tabela 66 - - Dados da tabela CURSO

id_curso	código	nome	sigla	id_departamento
1	1111	CURSO 1	C1	1
2	2222	CURSO 2	C2	1
3	3333	CURSO 3	C3	1

Tabela 67 - Inner Join ou Join

```
from Curso c inner join Aluno a on c.id = a.id_curso    ou

from Curso c join Aluno a on c.id = a.id_curso
```

Resultado (omitidos matrícula e cpf de `Aluno` e código, sigla e departamento de `Curso`):

Id_curso	nome	id_aluno	nome	id_curso
1	CURSO 1	1	João	1
2	CURSO 2	2	Maria	2
1	CURSO 1	3	Fernanda	1
1	CURSO 1	4	André	1

Tabela 68 - Left outer join ou left join

```
from Curso c left outer join Aluno a on c.id = a.id_curso    ou

from Curso c left join Aluno a on c.id = a.id_curso
```

Resultado (omitido matricula e cpf de Aluno e código, sigla e departamento de Curso):

Id_curso	nome	id_aluno	nome	id_curso
1	CURSO 1	1	João	1
2	CURSO 2	2	Maria	2
1	CURSO 1	3	Fernanda	1
1	CURSO 1	4	André	1
3	CURSO 3			

As consultas HQL normalmente não explicitam os *joins*. As associações já são informadas no mapeamento das classes Java, o *Hibernate* com a informação no documento do mapeamento deduz os *joins* necessários nas tabelas. Isto facilita as escritas das consultas. Existem quatro maneiras de realizar os *joins*, são: *Join* ordinários (no *from*), *Fetch Joins* (usado para coleções), Estilos *Theta* (entre classes não associadas) e *Joins* implícitos, quando se utiliza o operador de acesso '.'.

Para enviar uma consulta HQL para o banco de dados, uma interface chamada `Query` é utilizada para representar a consulta. A Tabela 69 apresenta um objeto `Query` sendo instanciado para fazer consultas da tabela `ALUNO`. O comando

`q.list()` faz com que todos os alunos cadastrados sejam retornados, já que a HQL utilizada foi "`from Aluno`".

Tabela 69 - Obtendo o resultado de uma consulta ao banco usando Query

```
Query q = session.createQuery("from Aluno");
q.list();
```

A Tabela 70 apresenta em exemplo de consulta que retorna todos os cursos que possuem nomes começados pela letra **A**. Já a consulta da Tabela 71 retorna todos os cursos que possui departamento com identificador igual a **2**.

Tabela 70 - Exemplo com Query

```
String hql = "from Curso c where c.nome like 'A%'";
Query q = session.createQuery(hql);
q.list();
```

Tabela 71 - Exemplo com Query

```
String hql = "from Curso c where c.departamento.id = 2";
Query q = session.createQuery(hql);
q.list();
```

A Tabela 72 apresenta um exemplo de consulta com a interface `Query` no estilo `Theta`.

Tabela 72 - Exemplo com Query no Estilo Theta

```
String hql = "from Aluno a, LogRecord log where a.id_aluno =
log.id_aluno";
Query q = session.createQuery(hql);
Iterator it = q.list.iterator();
while ( i.hasNext() ) {
    Object[] obj = (Object[]) i.next();
    Aluno aluno = (Aluno) obj[0];
    LogRecord log = (LogRecord) obj[1];
}
```

Com o estilo Theta obtemos dados por meio de Projeções, que é uma maneira de se recuperar apenas alguns dados da classe mapeada. A Tabela 73 apresenta um exemplo dessa consulta.

Tabela 73 - Exemplo Consulta de Projeção

```
String hql = "select c.nome, a.nome from Curso c, Aluno a where
                        a.id_curso = c.id_curso";
Query q = session.createQuery(hql);

ArrayList resultado = new ArrayList();
Iterator it = q.list.iterator();
while ( i.hasNext() ) {

    Object[] obj = (Object[]) i.next();

    Curso curso = new Curso();
    curso.setNome(obj[0].toString());

    Aluno aluno = new Aluno();
    aluno.setNome(obj[1].toString());
    aluno.setCurso(curso);

    resultado.add(aluno);
}
```

HQL suporta ainda funções agregadas como: `avg(...)`, `sum(...)`, `min(... )`, `max(...)`, `count(*)`, `count( ...)`, `count(distinct ...)` e `count(all ...)`.

A consulta que retorna valores agregados permite que estes sejam agrupados por classes ou mesmo por atributos. A cláusula utilizada é `group by`.

A Tabela 74 mostra exemplos com funções de agregação e `group by`.

Tabela 74 - Exemplos HQL: Agregação e Group By

```
select count(*) from Aluno
select min(a.matricula), max(a.matricula) from Aluno a
select c.nome, count(c.id_departamento) from Curso c group by c.nome
```

Com HQL também é possível informar parâmetros da consulta após a criação das `Query`, basta que esses parâmetros recebam nomes. Por exemplo, no exemplo da Tabela 75, deseja-se saber todos os alunos que possuem determinado nome. Na criação da `Query`, o nome dos alunos buscados é nomeado a partir do parâmetro `:nome` que é informado posteriormente usando-se o comando `q.setString("nome","João")`. Assim, `q.list()` retornará todos os alunos de

nome João. Resumindo, nomeia-se um parâmetro através de ":<nome_parametro>" e em seguida, define-se o seu valor.

Tabela 75 - Exemplo Query: Nomeação de Parâmetros

```
Query q = createQuery("from Aluno a where a.nome = :nome");
q.setString("nome", "João");
q.list();
```

17.3 Criteria

A API *Query By Criteria* (QBC) permite realizar consultas por manipulação de critérios em tempo de execução. Esta abordagem permite definir restrições dinamicamente sem que haja manipulação de `Strings`, a sua flexibilidade e poder é bem menor do que HQL e não suporta agregações nem projeções. Nas tabelas abaixo podemos ver simples exemplo utilizando `Criteria`, o `Session` é uma fábrica de criação de objetos `Criteria`.

Um objeto `Criteria` é obtido a partir do objeto `Session`, como mostrado no exemplo da Tabela 76. Nesse exemplo todas as instâncias de alunos presentes na tabela `ALUNO` são retornadas.

Tabela 76 - Obtendo um Criteria

```
Criteria criteria = session.createCriteria(Aluno.class);
criteria.list();
```

As Tabela 77 e

Tabela **78** apresentam exemplos de consultas utilizando o objeto Criteria.

Tabela 77 – Exemplo de Criteria com Restrições

```
Criteria criteria = session.createCriteria(Aluno.class);
criteria.add(Expression.eq("nome", "Maria");
criteria.add(Order.asc(“matricula”);
criteria.list();
```

Tabela 78 – Exemplo de Criteria com Restrições Like

```
Criteria criteria = session.createCriteria(Aluno.class);
criteria.add(Expression.like("nome", "Maria%");
criteria.add(Order.asc("matricula");
criteria.list();
```

Com `Criteria` é possível consultar associações de classes usando `createCriteria()`. Podendo existir vários níveis de subcriterias. Veja um exemplo na Tabela 79.

Tabela 79 – Sub-Criteria

```
Criteria criteria = session.createCriteria(Centro.class);

Criteria subCrit = criteria.createCriteria("universidade");
subCrit.add(Expression.eq("id", 1);

criteria.list();
```

17.4 Query By Example

Uma extensão da API `Criteria`, o *Query By Example* (QBE), cria uma instância da classe utilizada com alguns valores das propriedades setados, retornando todos os objetos persistentes que possuam os valores das propriedades. Veja exemplo na Tabela 80.

Tabela 80 – Query By Example

```
Aluno aluno = new Aluno();
aluno.setNome("João");
Criteria criteria = session.createCriteria(Aluno.class);
Criteria.add(Example.create(aluno);
criteria.list();
```

17.5 Paginação

Paginação é uma forma de limitar uma consulta muito grande ao banco de dados. A paginação pode ser realizada utilizando a interface `Query` ou mesmo utilizando a API `Criteria`. Nas Tabela 81 e Tabela 82 seguem exemplos utilizando paginação com o `Query` e o `Criteria`, respectivamente. Nesses exemplos, o resultado será os alunos encontrados a partir do décimo até o quadragésimo.

Tabela 81 - Paginação Query

```
Query q = session.createQuery("from Aluno");
q.setFirstResult(10);
q.setMaxResults(30);
q.list();
```

Tabela 82 - Paginação Criteria

```
Criteria criteria = session.createCriteria(Aluno.class);
criteria.setFirstResult(10);
criteria.setMaxResults(30);
criteria.list();
```

18. Conclusões

A utilização do mecanismo de mapeamento objeto relacional pode reduzir consideravelmente o tempo de implementação de uma aplicação no que diz respeito à codificação relacionada à persistência. A utilização do *Hibernate* para realizar esse mapeamento passa a ser uma ótima escolha, já que mostrou provê grandes facilidades no desenvolvimento.

Neste material foi apresentada uma pequena parte do que o *Hibernate* oferece. Dentre outras coisas não apresentadas, o *Hibernate* também fornece mecanismos para gerenciamento de transações; mecanismos que possibilitam trabalhar com sistemas de *caches* de informações, gerenciador próprio de versões, mecanismo para validações, etc.

Caso acredite-se que há alguma perda de tempo relacionada à criação dos arquivos de mapeamento XML do *Hibernate*, ainda existe a possibilidade da utilização de ferramentas que automatizam o processo, por exemplo, o *HibernateTools*.

19. Referências

[1] Christian Bauer e Gavin King. Hibernate in Action. 2005.

[2] Grupo Hibernate. Hibernate Reference Documentation. Version 3.0.5. Obtido em http://www.hibernate.org

[3] Gleydson de Azevedo Ferreira Lima. Material Didático. 2005.

[4] Nick Heudecker. Introdução ao Hibernate.

[5] Maurício Linhares. Introdução ao Hibernate 3.

[6] Francesc Rosés Albiol. Introducción a Hibernate. 2003.

[7] Fabiano Kraemer, Jerônimo Jardel Vogt. Hibernate, um Robusto Framework de Persistência Objeto-Relacional. 2005.