

Introdução ao Hibernate 3

Maurício Linhares

Mapeie o seu modelo de objetos diretamente para o banco de dados, de uma forma simples e se livrando de vez da SQL

O que é o Hibernate?

O [Hibernate](#) é uma ferramenta de mapeamento objeto/relacional para Java. Ela transforma os dados tabulares de um banco de dados em um grafo de objetos definido pelo desenvolvedor. Usando o Hibernate, o desenvolvedor se livra de escrever muito do código de acesso a banco de dados e de SQL que ele escreveria não usando a ferramenta, acelerando a velocidade do seu desenvolvimento de uma forma fantástica.

Mas o framework não é uma boa opção para todos os tipos de aplicação. Sistemas que fazem uso extensivo de stored procedures, triggers ou que implementam a maior parte da lógica da aplicação no banco de dados, contando com um modelo de objetos “pobre” não vai se beneficiar com o uso do Hibernate. Ele é mais indicado para sistemas que contam com um modelo rico, onde a maior parte da lógica de negócios fica na própria aplicação Java, dependendo pouco de funções específicas do banco de dados.

Antes de você seguir em frente...

Antes de começar você deve fazer o download de uma versão estável do Hibernate 3, de preferência a versão 3.0.5, com a qual este artigo foi testado. Você pode encontrar os arquivos dela aqui:

https://sourceforge.net/project/showfiles.php?group_id=40712&package_id=127784

Depois de fazer o download, adicione o arquivo hibernate3.jar e os seguintes arquivos da pasta “lib” do download ao seu classpath (ou ao classpath da sua IDE):

```
ehcache-1.1.jar  
jta.jar  
xml-apis.jar  
commons-logging-1.0.4.jar  
c3p0-0.8.5.2.jar  
asm-attrs.jar  
log4j-1.2.9.jar  
dom4j-1.6.jar  
antlr-2.7.5H3.jar  
cglib-2.1.jar  
asm.jar  
jdbc2_0-stdext.jar  
xerces-2.6.2.jar  
commons-collections-2.1.1.jar
```

O banco de dados escolhido para este artigo é o MySQL, versão 4.1, mas os scripts SQL pra gerar as tabelas podem ser facilmente adaptados para outros bancos. Usando o MySQL você também vai ter que colocar o driver JDBC dele no seu classpath, ele pode ser baixado no seguinte endereço (a versão utilizada no artigo foi a 3.1.8):

<http://dev.mysql.com/downloads/connector/j/3.1.html>

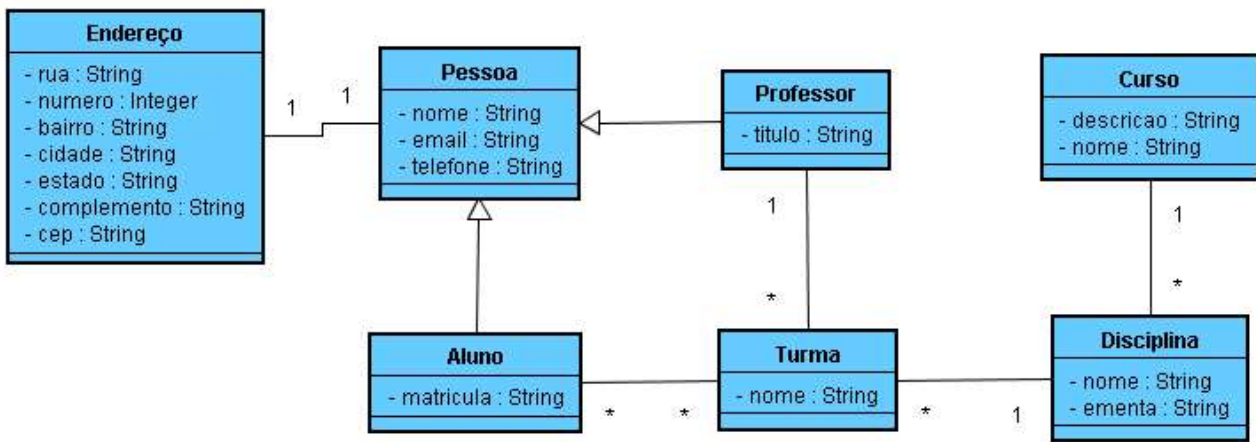
Além do driver do MySQL, você também vai ter que adicionar os arquivos .JAR que estão na pasta “lib” do arquivo do driver, o “aspectjrt.jar” e o “aspectjtools.jar”.

Você também deve criar o banco de dados, usando o arquivo “banco.sql” que está junto com os outros anexos desse artigo.

Neste artigo você vai conhecer um pouco sobre os mapeamentos do Hibernate e a HQL, a linguagem de buscas do framework. O artigo assume que você já tenha um conhecimento sobre a linguagem Java, bancos de dados e XML, além de ser capaz de montar um ambiente de desenvolvimento Java com todas as configurações necessárias.

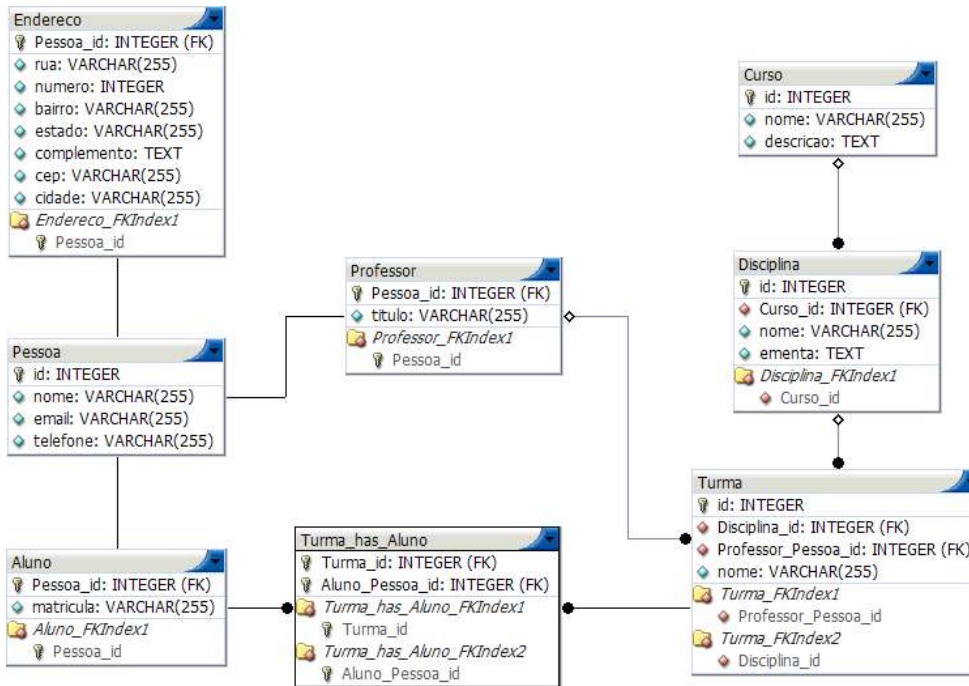
Definindo os objetos do modelo e as tabelas do banco

O nosso modelo é simples, ele modela um cadastro de alunos em uma universidade. Nele nós temos as classes, Pessoa, Aluno, Professor, Endereço, Turma, Disciplina e Curso, que se relacionam como mostrado no seguinte diagrama:



A classe **Pessoa**, que tem um **Endereço**, é a classe base para as classes **Aluno** e **Professor**. Um **Aluno** pode estar em várias **Turmas**, assim como um **Professor**. Cada **Turma** pode ter vários **Alunos** e apenas um **Professor**. A **Turma** tem ainda uma **Disciplina**, que pode conter várias **Turmas** e que tem um **Curso**, que também pode ter várias **Disciplinas**.

Com um modelo de objetos criado, vejamos como essas classes poderiam ser transformadas em tabelas, neste diagrama:



Como você já deve ter percebido, não há nada demais com esse modelo de banco de dados, as tabelas estão interligadas normalmente usando chaves primárias e chaves estrangeiras e uma tabela de relacionamento, no caso da relação N:N entre Aluno e Turma (use o script para criar as tabelas que está nos materiais anexos ao artigo). As tabelas tem as mesmas propriedades das classes e os mesmos relacionamentos.

Dando nome aos bois

Quando estiver mapeando as suas classes do modelo para o banco de dados, tente usar os mesmos nomes das classes e de suas propriedades, isso vai evitar várias dores de cabeça, como não saber se o nome “daquela” propriedade é todo em maiúsculas, ou se tem um “_” (underline) entre um nome e outro. Além do que, se os nomes forem iguais, você não precisa repetir os nomes das tabelas e dos relacionamentos no mapeamento do Hibernate.

Agora que já temos as nossas classes e elas foram mapeadas para o banco, podemos começar a trabalhar com os mapeamentos do Hibernate, pra fazer com que aquelas tabelas do banco se transformem nas nossas classes do modelo, quando fizermos acesso ao banco de dados.

Mapeando as classes para as tabelas

Antes de começar a fazer os mapeamentos do Hibernate, temos um conceito do banco de dados que precisa ser revisto, a identidade. Para um banco de dados, o modo de diferenciar uma linha das outras, é usando chaves, de preferência chaves não naturais, como as colunas “id” que nós criamos para nossas tabelas, mas no nosso modelo orientado a objetos, a identidade não é encontrada dessa forma. Em Java, nós definimos a identidade dos objetos sobrescrevendo o método “Object.equals(Object object)”, do modo que nos convier. A implementação “default” deste método, define a identidade através da posição de memória ocupada pelo objeto.

Não podemos usar o método “equals()” no banco de dados, porque o banco de dados não sabe que temos objetos, ele só entende tabelas, chaves primárias e estrangeiras. A solução é adicionar aos nossos objetos um identificador não natural, como os que nós encontramos no banco de dados, porque assim o banco de dados e o próprio Hibernate vão ser capazes de diferenciar os objetos e montar os seus relacionamentos. Nós fazemos isso adicionando uma propriedade chamada “id” do tipo Integer a todas as nossas classes, como no exemplo de código a seguir:

```
//Listagem do arquivo Pessoa.java

public class Pessoa {

    private String nome;

    private String email;

    private String telefone;

    private Endereco endereco;

    private Integer id;

    //métodos getters e setters das propriedades

}
```

Poderíamos ter escolhido o tipo primitivo "int" para a propriedade, mas trabalhando com objetos, o mapeamento e a resolução se um objeto existe ou não no banco de dados torna-se mais simples para o Hibernate. Outra observação importante são os métodos "getters" e "setters" para as propriedades dos objetos. O Hibernate não precisa de que as propriedades sejam definidas usando a nomenclatura dos JavaBeans, mas isso já é uma boa prática comum na comunidade, além de existirem outros frameworks e tecnologias que exigem essa nomenclatura (como a Expression Language dos JSPs), então nós definimos métodos "getters" e "setters" para todas as propriedades nos nossos objetos.

Conhecendo o arquivo "hbm.xml" e mapeando um relacionamento 1:1

Cortemos a conversa fiada e vamos iniciar a construção dos mapeamentos. O primeiro mapeamento abordado é o da classe Pessoa e do seu relacionamento com a classe Endereco. No nosso modelo, um Endereco tem apenas uma Pessoa e uma Pessoa tem apenas um Endereco (perceba que o arquivo .java e o nome da classe Java que modela Endereco não tem o "ç", ficou Endereco, para ficar com o mesmo nome da tabela). Vejamos o mapeamento para a classe Pessoa (o arquivo "Pessoa.hbm.xml"):

Nomes de arquivos e extensões

É uma boa prática usar um arquivo de mapeamento para cada classe e usar como extensão do arquivo ".hbm.xml" para diferenciar de outros arquivos XML utilizados na aplicação.

```
<?xml version="1.0" encoding="UTF-8"?>

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

<hibernate-mapping>

<class name="Pessoa">

    <!-- Identificador da classe -->

    <id name="id">
        <generator class="increment"/>
    </id>

    <!-- Propriedades da classe -->

    <property name="nome"/>
    <property name="telefone"/>
    <property name="email"/>

    <!-- Relacionamento da classe -->

    <one-to-one
        name="endereco"
        class="Endereco"
        cascade="save-update"/>

</class>

</hibernate-mapping>
```

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

O arquivo de mapeamento é um arquivo XML que define as propriedades e os relacionamentos de uma classe para o Hibernate, este arquivo pode conter classes, classes componentes e queries em HQL ou em SQL. No nosso exemplo, temos apenas uma classe sendo mapeada no arquivo, a classe Pessoa. O arquivo XML começa normalmente com as definições da DTD e do nó raiz, o `<hibernate-mapping>`, depois vem o nó que nos interessa neste caso, `<class>`.

No nó `<class>` nós definimos a classe que está sendo mapeada e para qual tabela ela vai ser mapeada. O único atributo obrigatório deste nó é `"name"`, que deve conter o nome completo da classe (com o pacote, se ele não tiver sido definido no atributo `"package"` do nó `<hibernate-mapping>`), se o nome da classe for diferente do nome da tabela, você pode colocar o nome da tabela no atributo `"table"`, no nosso exemplo isso não é necessário.

Seguindo em frente no nosso exemplo, temos o nó `<id>` que é o identificador dessa classe no banco de dados. Neste nó nós definimos a propriedade que guarda o identificador do objeto no atributo `"name"`, que no nosso caso é `"id"`, se o nome da coluna no banco de dados fosse diferente da propriedade do objeto, ela poderia ter sido definida no atributo `"column"`. Ainda dentro deste nó, nós encontramos mais um nó, o `<generator>`, este nó guarda a informação de como os identificadores (as chaves do banco de dados) são gerados, existem diversas classes de geradores, que são definidas no atributo `"class"` do nó, no nosso caso o gerador usado é o `"increment"`, que incrementa um ao valor da chave sempre que insere um novo objeto no banco, esse gerador costuma funcionar normalmente em todos os bancos.

Os próximos nós do arquivo são os `<property>` que indicam propriedades simples dos nossos objetos, como Strings, os tipos primitivos (e seus wrappers), objetos Date, Calendar, Locale, Currency e outros. Neste nó os atributos mais importante são `"name"`, que define o nome da propriedade, `"column"` para quando a propriedade não tiver o mesmo nome da coluna na tabela e `"type"` para definir o tipo do objeto que a propriedade guarda. Normalmente, o próprio Hibernate é capaz de descobrir qual é o tipo de objeto que a propriedade guarda, não sendo necessário escrever isso no arquivo de configuração, ele também usa o mesmo nome da propriedade para acessar a coluna, se o atributo não tiver sido preenchido. Nós definimos as três propriedades simples da nossa classe, `"nome"`, `"email"` e `"telefone"`.

O último nó do arquivo, `<one-to-one>`, define o relacionamento de 1-para-1 que a classe Pessoa tem com a classe Endereço. Este nó tem os atributos `"name"`, que define o nome da propriedade no objeto (neste caso, `"endereco"`), `"type"` que define a classe da propriedade e `"cascade"` que define como o Hibernate deve `"cascatear"` as ações feitas nesta classe para a classe relacionada, como atualizações, inserções e exclusões de registro. No nosso caso o `"cascade"` foi escolhido como `"save-update"` para que quando uma classe pessoa for inserida ou atualizada no banco, a propriedade `"endereco"` também seja inserida ou atualizada.

Com isso, terminamos o mapeamento da nossa classe Pessoa, mas este mapeamento faz referência a uma outra classe o Hibernate ainda não conhece, a classe Endereço. Vejamos então como fica o mapeamento desta classe (o arquivo é o `"Endereco.hbm.xml"`):

```
<?xml version="1.0" encoding="UTF-8"?>  
  
<!DOCTYPE hibernate-mapping  
  PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  
  
<hibernate-mapping>  
  
  <class name="Endereco">  
  
    <id name="id"  
      column="Pessoa_id">  
      <generator class="foreign">  
        <param name="property">pessoa</param>  
      </generator>  
    </id>  
  
    <property name="bairro"/>  
    <property name="cidade"/>  
    <property name="complemento"/>  
    <property name="estado"/>  
    <property name="numero"/>
```

```
<property name="rua"/>
<property name="cep"/>

<one-to-one
    name="pessoa"
    class="Pessoa"
    constrained="true"/>

</class>

</hibernate-mapping>
```

Como você agora já conhece a estrutura básica do arquivo de mapeamento, vejamos as diferenças deste arquivo para o "Pessoa.hbm.xml". A primeira coisa que você deve estar notando, é que o atributo "class" do nó <generator> não é "increment", é "foreign" e agora também temos um corpo do nó, com o valor "pessoa". Isso acontece devido ao tipo de relacionamento que nós criamos no banco de dados, entre as tabelas Endereco e Pessoa. Também existe mais um nó dentro de <generator> que não existia no mapeamento anterior, o <param>, esse nó serve para passar um parâmetro para a classe geradora do identificador, que neste caso é o nome da propriedade "dona" deste objeto, "pessoa".

Para garantir que cada Endereco pertença a apenas uma Pessoa, fizemos com que a chave primária de Endereco (Pessoa_id) fosse também a chave estrangeira que a liga a Pessoa, desse modo, garantimos que cada Endereco pertence a apenas uma Pessoa e vice-versa. Outra novidade é que também tivemos que colocar o nome da coluna da chave já que ela não é igual a o nome da propriedade da classe, a propriedade se chama "id" e a coluna "Pessoa_id", assim, tivemos que declarar o atributo "column" do nó <id>.

Continuando no mapeamento, encontramos as propriedades simples da classe, que são declaradas de modo idêntico as que nós vimos no mapeamento anterior, com o nó <property> e com o atributo "name" contendo o nome da propriedade. Mais uma vez não indicamos o nome da coluna (porque os nomes são iguais as propriedades) nem o tipo, já que o Hibernate pode descobrir isso sozinho.

No nosso modelo, o relacionamento entre Pessoa e Endereco é bidirecional, o que quer dizer que é sempre possível navegar de um objeto para o outro, pois Pessoa aponta para Endereco e Endereco aponta para Pessoa. Para simbolizar isso no mapeamento, nós temos que adicionar o nó que também existe no mapeamento de Pessoa, <one-to-one>. Como você já percebeu, os atributos continuam os mesmos, "name" como sendo o nome da propriedade na classe e "class" como sendo o nome da classe dessa propriedade. Mas nós temos um atributo que não existia na relação anterior, "constrained", que nessa relação vai significar que existe uma relação entre a chave primária de Endereco e de Pessoa, avisando ao Hibernate que um Endereco não pode existir sem que exista uma Pessoa, assim, mesmo que o banco de dados não garantisse a integridade referencial do sistema, o próprio Hibernate garantiria.

Mapeando herança

Continuando o trabalho de mapear o nosso modelo para o Hibernate, vamos para a herança que nós encontramos entre Pessoa, Aluno e Professor. Pessoa é a classe pai, da qual Aluno e Professor herdam as propriedades e comportamentos.

No Hibernate existem diversas formas de se fazer o mapeamento de uma relação de herança. Usando uma tabela para cada classe filha (a classe pai não teria tabela, as propriedades comuns se repetiriam nas tabelas filhas), usando uma tabela para todas as classes (discriminadores definiriam quando é uma classe ou outra), usando uma tabela para cada uma das classes (uma para a classe pai e mais uma para cada classe filha) ou até mesmo uma mistura de todas essas possibilidades (disponível apenas na versão 3 do Hibernate). Ficou confuso? Não se preocupe, neste artigo nós vamos abordar apenas a mais comum e mais simples de ser mantida, uma tabela para cada classe.

No nosso banco de dados, temos uma tabela Pessoa e outras duas tabelas, Aluno e Professor, que estão relacionadas a professor através de suas chaves primárias a tabela Pessoa, garantindo assim que não existam Alunos ou Professores com o mesmo identificador. Vejamos então o mapeamento da classe Professor (o arquivo "Professor.hbm.xml"):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
```

```
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <joined-subclass name="Professor" extends="Pessoa">
        <key column="Pessoa_id"/>
        <property name="titulo"/>
        <set name="turmas"
            inverse="true">
            <key column="Pessoa_Professor_id"/>
            <one-to-many class="Turma"/>
        </set>
    </joined-subclass>
</hibernate-mapping>
```

Nesse mapeamento vemos um nó que nós ainda não conhecíamos, `<joined-subclass>`, que indica o mapeamento de herança usando uma tabela para cada classe, porque para retornar o objeto o Hibernate precisa fazer um "join" entre as duas tabelas. O atributo "name" é o nome da classe mapeada e o atributo "extends" recebe o nome da classe pai (neste caso, Pessoa), se o nome da classe não fosse igual ao da tabela, poderíamos adicionar o nome da tabela no atributo "table" (que foi omitido porque o nome da classe é igual ao nome da tabela).

Seguindo no mapeamento, percebemos mais um nó desconhecido, `<key>`. Este nó avisa ao Hibernate o nome da coluna que guarda a chave primária da tabela, que também é a chave estrangeira que liga a tabela Professor a tabela Pessoa. Neste nó nós adicionamos o atributo "column" e colocamos o nome da coluna que guarda a chave, que é "Pessoa_id".

O mapeamento continua com a declaração de uma propriedade e com um nó que nós também não conhecemos, `<set>`, que pode simbolizar um relacionamento 1:N ou N:N. Este nó vai ser explicado mais tarde neste artigo.

Passando para o mapeamento da classe Aluno, percebemos que não existe muita diferença:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <joined-subclass name="Aluno" extends="Pessoa">
        <key column="Pessoa_id"/>
        <property name="matricula"/>
        <set name="turmas"
            table="Turma_has_Aluno"
            inverse="true">
            <key column="Turma_id"/>
            <many-to-many class="Turma"/>
        </set>
    </joined-subclass>
</hibernate-mapping>
```

A classe é declarada do mesmo modo que Professor, usando o nó `<joined-subclass>` e usando os mesmos atributos que foram usados no mapeamento anterior. O nó `<key>` também foi incluído do mesmo modo, com o nome da coluna da chave primária/estrangeira da tabela, mais uma declaração de propriedade e mais um nó `<set>`, nada além do que já foi visto.

Mapeando associações 1:N e N:N

Você já conheceu o nó `<set>` nos mapeamentos anteriores, vamos entender agora como ele funciona e como utilizá-lo para tornar o acesso aos objetos associados ainda mais simples, mais uma vez sem nenhuma linha de SQL. Uma "set" ou "conjunto" representa uma coleção de objetos não repetidos, que podem ou não estar ordenados (dependendo da implementação da interface `java.util.Set` escolhida). Quando você adiciona um nó deste tipo em um arquivo de mapeamento, você está indicando ao Hibernate que o seu objeto tem um relacionamento 1:N ou N:N com outro objeto. Vejamos o exemplo da classe `Professor`, que tem um relacionamento 1:N com a classe `Turma`:

```
<joined-subclass name="Professor" extends="Pessoa">
    <key column="Pessoa_id"/>
    <property name="titulo"/>
    <set name="turmas"
        inverse="true">
        <key column="Pessoa_Professor_id"/>
        <one-to-many class="Turma"/>
    </set>
</joined-subclass>
```

No nó `<set>` o primeiro atributo a ser encontrado é "name" que assim como nos outros nós, define o nome da propriedade que está sendo tratada, já o outro atributo, "inverse", define como o Hibernate vai tratar a inserção e retirada de objetos nessa associação. Quando um lado da associação define o atributo "inverse" para "true" ele está indicando que apenas quando um objeto for inserido do "outro lado" da associação ele deve ser persistido no banco de dados.

Para entender melhor o atributo "inverse" pense em como está definido o mapeamento da classe `Professor`. Lá ele está definido para "true", significando que apenas quando uma `Turma` adicionar um professor, o relacionamento vai ser persistido no banco de dados. Em código:

```
Professor professor = new Professor();
Turma turma = new Turma();

turma.setProfessor(professor);
professor.getTurmas().add(turma);
```

Se apenas o `Professor` adicionando a `Turma` a sua coleção de `Turmas`, nada iria acontecer ao banco de dados. Isso parece não ter muito sentido, mas se você prestar bem atenção, o Hibernate não tem como saber qual dos dois lados foi atualizado, desse modo ele vai sempre atualizar os dois lados duas vezes, uma para cada classe da relação, o que seria desnecessário. Usando "inverse" você define de qual lado o Hibernate deve esperar a atualização e ele vai fazer a atualização apenas uma vez. Lembre-se sempre de adicionar os objetos dos dois lados, pois em Java os relacionamentos não são naturalmente bidirecionais.

Voltando a o nó `<set>`, percebemos que dentro dele ainda existem mais dois nós, `<key>` e `<one-to-many>`. O nó `<key>` representa a coluna da tabela relacionada (neste caso, `Turma`) que guarda a chave estrangeira para a classe `Professor`, nós adicionamos o atributo "column" e o valor é o nome da coluna, que neste caso é "Pessoa_Professor_id". No outro nó, `<one-to-many>`, nós definimos a classe a qual pertence essa coleção de objetos, que é `Turma` (lembre-se sempre de usar o nome completo da classe, com o pacote).

Depois do relacionamento um-para-muitos (1:N) vejamos agora como mapear um relacionamento muitos-para-muitos (N:N). Em um banco de dados relacional, este tipo de associação faz uso de uma "tabela de relação", que guarda as chaves estrangeiras das duas tabelas associadas, como ocorre no nosso exemplo, onde tivemos que criar uma tabela "Turma_has_Aluno", para poder mapear o relacionamento N:N entre as classes `Turma` e `Aluno` no banco de dados.

Como você já sabe, este tipo de associação também é definida usando o nó `<set>`, vejamos então o nosso exemplo:

```
<joined-subclass name="Aluno" extends="Pessoa">
    <key column="Pessoa_id"/>
    <property name="matricula"/>
    <set name="turmas"
        table="Turma_has_Aluno"
        inverse="true">
        <key column="Aluno_Pessoa_id"/>
        <many-to-many class="Turma" column="Turma_id"/>
    </set>
</joined-subclass>
```

Os atributos do nó `<set>` neste mapeamento são parecidos com os que nós vimos no mapeamento da classe Professor, mas eles contêm algumas informações adicionais graças ao tipo da associação. Primeiro, temos um novo atributo no nó `<set>`, o `"table"`. Como nós havíamos falado antes, para uma associação N:N você precisa de uma "tabela de relacionamento" e o atributo `"table"` guarda o nome desta tabela. Mais uma vez o atributo `"inverse"` foi marcado como `"true"` indicando que apenas as ações feitas do "outro lado" da associação vão ser persistidas no banco de dados.

Dentro do nó, nós encontramos mais dois nós, `<key>`, que vai conter, no atributo `"column"`, o nome da coluna que guarda a chave estrangeira para a tabela Aluno e o nó `<many-to-many>`, que contêm a classe dos objetos da coleção, no atributo `"class"` e o nome da coluna na "tabela de relacionamento" que referencia a chave primária da tabela Turma, no atributo `"column"`.

Vamos agora terminar de mapear as nossas outras classes do nosso modelo e tornar as associações bidirecionais, como manda o nosso modelo de classes. Começando pela classe Turma:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="Turma">
        <id name="id">
            <generator class="increment"/>
        </id>
        <property name="nome"/>
        <many-to-one
            name="professor"
            class="Professor"
            column="Professor_Pessoa_id"/>
        <many-to-one
            name="disciplina"
            class="Disciplina"
            column="Disciplina_id"/>
        <set name="alunos"
            table="Turma_has_Aluno">
            <key column="Turma_id"/>
            <many-to-many
                class="Aluno"
                column="Aluno_Pessoa_id"/>
        </set>
    </class>
</hibernate-mapping>
```

A maior parte do código já é conhecida, mas ainda existem algumas coisas que precisam ser esclarecidas. Turma é o lado "um" de dois relacionamentos um-para-muitos, um com a classe Professor e outro com a classe Disciplina, para tornar essa associação bidirecional nós vamos utilizar o nó `<many-to-one>`, que modela o lado "um" de uma associação um-para-muitos (1:N). Neste nó, nós inserimos os atributos `"name"`, que recebe o nome da propriedade, `"class"`, que recebe o nome da classe da propriedade e `"column"`, que recebe o nome da coluna nesta tabela que guarda a chave estrangeira para a outra tabela do relacionamento.

O resto do mapeamento é idêntico aos outros mapeamentos que nós já estudamos, a única diferença é o lado da associação que ele está modelando. Os dois outros mapeamentos, de Disciplina e Curso, não trazem nenhuma novidade para o nosso estudo, portanto esteja livre para estudar os arquivos junto com o resto do material de apoio.

Cuidado com a descoberta automática de tipos

O Hibernate realmente facilita a vida do desenvolvedor com a descoberta automática dos tipos das propriedades, mas em alguns momentos você vai ter que definir o tipo da propriedade, para que o Hibernate não entre em conflito com o banco.

Um tipo `java.util.Date`, por exemplo, pode simbolizar tanto um tipo "DATE" quanto um "TIMESTAMP" em um banco de dados, por isso, você deve avisar ao Hibernate o que ele está mapeando, se é um "DATE" ou um "TIMESTAMP", no atributo `"type"` da propriedade. Esse mesmo cuidado também deve ser tomado para CLOBs, BLOBs e a classe `java.util.Calendar`.

Configurando o Hibernate 3

A engine do Hibernate pode ser configurada de três modos diferentes, instanciando um objeto de configuração (**`org.hibernate.cfg.Configuration`**) e inserindo as suas propriedades programaticamente, usando um arquivo `.properties` com as suas configurações e indicando os arquivos de mapeamento programaticamente ou usando um arquivo XML (o `"hibernate.cfg.xml"`) com as propriedades de inicialização e os caminhos dos arquivos de mapeamento. Vejamos como configurar o Hibernate para o nosso projeto:

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <property name="hibernate.dialect">
            org.hibernate.dialect.MySQLDialect
        </property>
        <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver
        </property>
        <property name="hibernate.connection.url">
            jdbc:mysql://localhost/hibernate?autoReconnect=true
        </property>
        <property name="hibernate.connection.username">
            root
        </property>
        <property name="hibernate.connection.password">
        </property>

        <!-- Configuração do c3p0 -->

        <property name="hibernate.c3p0.max_size">10</property>
        <property name="hibernate.c3p0.min_size">2</property>
        <property name="hibernate.c3p0.timeout">5000</property>
        <property name="hibernate.c3p0.max_statements">10</property>
        <property name="hibernate.c3p0.idle_test_period">3000</property>
    </session-factory>
</hibernate-configuration>
```

```
<property name="hibernate.c3p0.acquire_increment">2</property>

<!-- Configurações de debug -->

<property name="show_sql">true</property>
<property name="hibernate.generate_statistics">true</property>
<property name="hibernate.use_sql_comments">true</property>

<mapping resource="Curso.hbm.xml"/>
<mapping resource="Disciplina.hbm.xml"/>
<mapping resource="Turma.hbm.xml"/>
<mapping resource="Pessoa.hbm.xml"/>
<mapping resource="Aluno.hbm.xml"/>
<mapping resource="Professor.hbm.xml"/>
<mapping resource="Endereco.hbm.xml"/>

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

Na documentação do Hibernate você pode verificar todas as opções de propriedades que podem ser utilizadas e seus respectivos resultados, por isso nós vamos nos ater ao que é importante para começarmos a trabalhar. Vejamos as propriedades:

hibernate.dialect: é a implementação do dialeto SQL específico do banco de dados a ser utilizado.

hibernate.connection.driver_class: é o 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.

Essa segunda parte do arquivo são as configurações do "pool" de conexões escolhido para a nossa aplicação. No nosso exemplo o pool utilizado é o C3P0, mas você poderia utilizar qualquer um dos pools que são oferecidos no Hibernate ou então usar um DataSource do seu servidor de aplicação. Na terceira parte, estão algumas opções para ajudar a debugar o comportamento do Hibernate, a propriedade "show_sql" faz com que todo o código SQL gerado seja escrito na saída default, "hibernate.generate_statistics" faz com que o Hibernate gere estatísticas de uso e possa diagnosticar uma má performance do sistema e "hibernate.use_sql_comments" adiciona comentários ao código SQL gerado, facilitando o entendimento das queries.

A última parte do arquivo é onde nós indicamos os arquivos de mapeamento que o Hibernate deve processar antes de começar a trabalhar, se você esquecer de indicar um arquivo de mapeamento de qualquer classe, essa classe não vai poder ser persistida pela engine do Hibernate. Outro detalhe importante, é que quando você usa mapeamentos com Herança, o mapeamento pai deve sempre vir antes do filho.

Persistência em cascata

Os relacionamentos entre os objetos no Hibernate podem ser tratados com ações "em cascata". Isso significaria, no nosso exemplo, que você poderia inserir um objeto Pessoa no banco de dados e depois associar um Endereco a ele, sem ter de indicar ao Hibernate que este objeto deve ser persistido no banco de dados. Esse tipo de ação é definido pelo atributo "cascade" nos nós de relacionamento (<set>, <many-to-one>, etc). Ele pode assumir os seguintes valores:

- ✓ "none" : o Hibernate ignora a associação
- ✓ "save-update" : ele vai inserir ou atualizar automaticamente os objetos associados, quando o objeto "raiz" for inserido ou atualizado
- ✓ "delete" : ele vai deletar os objetos associados ao objeto raiz
- ✓ "all" : uma junção de "delete" e "save-update"
- ✓ "all-delete-orphan" : o mesmo que "all", mas o Hibernate vai deletar qualquer objeto que tiver sido retirado da associação
- ✓ "delete-orphan" : faz com que o Hibernate veja se o objeto ainda faz parte da associação, se ele não fizer, vai ser deletado

Entrando em Ação

Agora que você já está com o Hibernate configurado e pronto para funcionar, vamos entender o mecanismo de persistência dele. Para o Hibernate, existem três tipos de objetos, objetos "transient" (transientes), "detached" (desligados) e "persistent" (persistentes). Objetos "transient" são aqueles que ainda não tem uma representação no banco de dados (ou que foram excluídos), eles ainda não estão sobre o controle do framework e podem não ser mais referenciáveis a qualquer momento, como qualquer objeto normal em Java. Objetos "detached" têm uma representação no banco de dados, mas não fazem mais parte de uma sessão do Hibernate, o que significa que o seu estado pode não estar mais sincronizado com o banco de dados. Objetos "persistent" são os objetos que tem uma representação no banco de dados e que ainda fazem parte de uma transação do Hibernate, garantindo assim que o seu estado esteja sincronizado com o banco de dados (nem sempre, claro).

No Hibernate, assim como no JDBC, existem os conceitos de sessão e transação. Uma sessão é uma conexão aberta com o banco de dados, onde nós podemos executar queries, inserir, atualizar e deletar objetos, já a transação é a demarcação das ações, uma transação faz o controle do que acontece e pode fazer um rollback, assim como uma transação do JDBC, se forem encontrados problemas.

Edite o arquivo de configuração do Hibernate (hibernate.cfg.xml) com as suas informações específicas (nome de usuário, senha, URL de conexão, etc), coloque ele na raiz do seu classpath, junto com as classes compiladas e os arquivos de mapeamento, porque nós vamos colocar o Hibernate pra funcionar (tenha certeza de que o seu classpath está configurado corretamente, com todos os arquivos necessários).

Primeiro, vamos criar uma classe para configurar e abrir as sessões do Hibernate, o código é simples:

```
//Arquivo HibernateUtility.java

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtility {

    private static SessionFactory factory;

    static {

        try {

            factory = new Configuration().configure().buildSessionFactory();

        } catch (Exception e) {

            e.printStackTrace();
            factory = null;

        }

    }

    public static Session getSession() {

        return factory.openSession();

    }

}
```

O bloco estático (as chaves marcadas como "static") instancia um objeto de configuração do Hibernate (org.hibernate.cfg.Configuration), chama o método configure() (que lê o nosso arquivo hibernate.cfg.xml) e depois que ele está configurado, criamos uma SessionFactory, que é a classe que vai ficar responsável por abrir as sessões de trabalho do Hibernate. Faça um pequeno teste pra ter certeza de que tudo está funcionando:

```
//Arquivo Teste.java
```

```
public class Teste {  
  
    public static void main(String[] args) {  
  
        Session sessao = HibernateUtility.getSession(); //Abrindo uma sessão  
        Transaction transaction = sessao.beginTransaction(); //Iniciando uma transação  
  
        Curso curso = new Curso(); //Instanciando um objeto transiente  
  
        curso.setNome("Desenvolvimento de Software"); //Preenchendo as propriedades do objeto  
        curso.setDescricao("Curso só pra programadores");  
  
        sessao.save(curso); //Transformando o objeto transiente em um objeto  
                           //persistente no banco de dados  
  
        transaction.commit(); //Finalizando a transação  
        sessao.close(); //Fechando a sessão  
  
    }  
}
```

Se ele não lançou nenhuma exceção, o seu ambiente está configurado corretamente. Vamos entender agora o que foi que aconteceu neste código, primeiro nós inicializamos a *SessionFactory*, dentro do bloco estático na classe *HibernateUtility*, depois que ela é inicializada, o método *getSession()* retorna uma nova sessão para o código dentro do *main()*. Após a sessão ter sido retornada, nós iniciamos uma transação, instanciamos um objeto *Curso*, preenchemos as suas propriedades e chamamos o método *save()* na sessão. Após o método *save()*, finalizamos a transação e fechamos a sessão. Acabamos de inserir um registro no banco de dados sem escrever nenhuma linha de SQL, apenas com a chamada de um método.

O código de aplicações que usam o Hibernate costumam mostrar esse mesmo comportamento, abrir sessão, iniciar transação, chamar os métodos *save()*, *update()*, *get()*, *delete()*, etc, fechar a transação e depois a sessão, um comportamento muito parecido com o de aplicações JDBC comuns, a diferença é que não escrevemos nem uma linha sequer de SQL para tanto.

Nomes das classes

Na maioria dos exemplos, nós não incluímos os “imports” nem usamos os nomes completos das classes para facilitar a leitura, veja aqui alguns dos nomes completos das classes que você vai encontrar:

- ✓ *Session* = *org.hibernate.Session*
- ✓ *SessionFactory* = *org.hibernate.SessionFactory*
- ✓ *Configuration* = *org.hibernate.cfg.Configuration*
- ✓ *Transaction* = *org.hibernate.Transaction*
- ✓ *Query* = *org.hibernate.Query*
- ✓ *Criteria* = *org.hibernate.Criteria*
- ✓ *Criterion* = *org.hibernate.criterion.Criterion*
- ✓ *Restrictions* = *org.hibernate.criterion.Restrictions*

Fazendo pesquisas no banco usando o Hibernate

Agora que você já sabe mapear e configurar a fera, podemos passar para uma das partes mais importantes do funcionamento do framework, as pesquisas. Existem três meios de se fazer buscas usando o Hibernate, usando a sua linguagem própria de buscas, a *Hibernate Query Language (HQL)*, usando a sua **Criteria Query API** (para montar buscas programaticamente) e usando SQL puro. A maioria das suas necessidades deve ser suprida com as duas primeiras alternativas, o resto, você sempre pode usar SQL pra resolver.

A HQL é uma extensão da SQL com alguns adendos de orientação a objetos, nela você não vai referenciar tabelas, vai referenciar os seus objetos do modelo que foram mapeados para as tabelas do banco de dados. Além disso, por fazer pesquisas em objetos, você não precisa selecionar as “colunas” do banco de dados, um *select* assim: “*select * from Turma*” em HQL seria simplesmente “*from Turma*”, porque não estamos mais pensando em tabelas e sim em objetos.

A Criteria Query API é um conjunto de classes para a montagem de queries em código Java, você define todas as propriedades da pesquisa chamando os métodos e avaliações das classes relacionadas. Como tudo é definido programaticamente, você ganha todas as funcionalidades inerentes da programação orientada a objetos para montar as suas pesquisas e ainda garante a completa independência dos bancos de dados, pois a classe de "dialeto SQL" do seu banco vai se encarregar de traduzir tudo o que você utilizar.

Antes de ver os exemplos propriamente ditos, vejamos como criar um objeto que implemente a interface Query ou Criteria. Para instanciar um desses objetos, você vai ter que abrir uma sessão do Hibernate, como nós já vimos na listagem do arquivo "Teste.java". Vejamos como ficaria o código:

```
Session sessao = HibernateUtility.getSession();
Transaction tx = sessao.beginTransaction();

Query select = sessao.createQuery("from Turma");
List objetos = select.list();

System.out.println(objetos);

tx.commit();
sessao.close();
```

O código ainda segue aquela mesma seqüência da listagem anterior, abrir uma sessão, iniciar uma transação e começar a acessar o banco de dados. Para criar uma query, nós chamamos o método `createQuery(String query)` na sessão, passando como parâmetro o String que representa a query. Depois disso, podemos chamar o método `list()`, que retorna um "List" com os objetos resultantes da query. Usando a Criteria API, o código ficaria desse modo:

```
Session sessao = HibernateUtility.getSession();
Transaction tx = sessao.beginTransaction();

Criteria select = sessao.createCriteria(Turma.class);
List objetos = select.list();

System.out.println(objetos);

tx.commit();
sessao.close();
```

Veja que apenas a linha onde nós criávamos a query mudou. Agora, em vez de chamar o método `createQuery(String query)`, nós chamamos o método `createCriteria(Class clazz)`, passando como parâmetro a classe que vai ser pesquisada, que no nosso caso é Tuma.

Outro complemento importante do Hibernate é o suporte a paginação de resultados. Para paginar os resultados, nós chamamos os métodos `setFirstResult(int first)` e `setMaxResults(int max)`. O primeiro método indica de qual posição os objetos devem começar a ser carregados, o segundo indica o máximo de objetos que devem ser carregados. Estes métodos estão presentes tanto na interface "Query" quanto na interface "Criteria". Vejamos como nós deveríamos proceder para carregar apenas as dez primeiras turmas do nosso banco de dados:

```
Session sessao = HibernateUtility.getSession();
Transaction tx = sessao.beginTransaction();

Criteria select = sessao.createCriteria(Turma.class);
select.setFirstResult(0);
select.setMaxResults(10);

List objetos = select.list();

System.out.println(objetos);

tx.commit();
sessao.close();
```

Veja que nós chamamos os métodos `setFirstResult()` e `setMaxResults()` **antes** de listar os objetos da query e lembre-se também que a contagem de resultados (assim como os arrays) começa em zero, não em um.

Uma propriedade específica da HQL, que foi herdada do JDBC, é o uso de parâmetros nas queries. Assim como no JDBC, os parâmetros podem ser numerados, mas também podem ser nomeados, o que simplifica ainda mais o uso e a manutenção das queries, porque a troca de posição não vai afetar o código que as usa. Vejamos um exemplo do uso de parâmetros:

```
Session sessao = HibernateUtility.getSession();
Transaction tx = sessao.beginTransaction();

Query select = sessao.createQuery("from Turma as turma where turma.nome = :nome");
select.setString("nome", "Jornalismo");

List objetos = select.list();

System.out.println(objetos);

tx.commit();
sessao.close();
```

Nesse nosso novo exemplo, tivemos mais algumas adições a query HQL. A primeira é o uso do "as", que serve para "apelidar" uma classe na nossa expressão (do mesmo modo do "as" em SQL), ele não é obrigatório, o código poderia estar "from Turma turma ..." e ele funcionaria normalmente. Outra coisa a se notar, é o acesso as propriedades usando o operador "." (ponto), você sempre acessa as propriedades usando esse operador. E a última parte é o parâmetro propriamente dito, que deve ser iniciado com ":" (dois pontos) para que o Hibernate saiba que isso é um parâmetro que vai ser inserido na query. Você insere um parâmetro nomeado, usando o método "set" correspondente ao tipo de parâmetro, passando primeiro o nome com o qual ele foi inserido e depois o valor que deve ser colocado.

E se você também não gosta de ficar metendo código SQL no meio do seu programa, porque deveria meter código HQL? O Hibernate facilita a externalização de queries HQL (e até SQL) com os nós `<query>` e `<sql-query>` nos arquivos de mapeamento. Vamos adicionar uma query no mapeamento da classe turma (o arquivo "Turma.hbm.xml"):

```
<query name="buscarTurmasPeloNome">
  <![CDATA[from Turma t where t.nome = :nome]]>
</query>
```

Essas linhas adicionadas no arquivo de mapeamento vão instruir o Hibernate a criar um `PreparedStatement` para esse select, fazendo com que ele execute mais rápido e possa ser chamado de qualquer lugar do sistema. Você deve preencher o atributo "name" com o nome pelo qual esta query deve ser chamada na aplicação e no corpo do nó você deve colocar o código da query, de preferência dentro de uma tag CDATA, pra evitar que o parser XML entenda o que está escrito na query como informações para ele. Veja como executar uma "named query":

```
Session sessao = HibernateUtility.getSession();
Transaction tx = sessao.beginTransaction();

Query select = sessao.getNamedQuery("buscarTurmasPeloNome");
select.setString("nome", "Jornalismo");

List objetos = select.list();

System.out.println(objetos);

tx.commit();
sessao.close();
```

A única diferença para as outras listagens é que em vez de escrevermos a query dentro do código Java, ela ficou externalizada no arquivo de mapeamento. Para instanciar o objeto, chamamos o método `getNamedQuery(String name)`, passando como parâmetro o nome da query definido no atributo "name" no arquivo XML.

Adicionando restrições as pesquisas

Vamos ver agora como adicionar restrições as nossas pesquisas no Hibernate. Os exemplos vão seguir uma dinâmica simples, com uma versão em HQL e como o mesmo exemplo poderia ser feito usando a API de Criteria.

A HQL e a API de Criteria suportam todos os operadores de comparação da SQL (\leq , $>$, $<>$, \leq , \geq , $=$, *between*, *not between*, *in* e *not in*), vejamos um exemplo em HQL:

```
from Aluno al where al.endereco.numero >= 50
```

O uso dos operadores de comparação segue a mesma sintaxe da linguagem SQL, usando Criteria, essa mesma query poderia ser expressa da seguinte maneira:

```
Criteria select = sessao.createCriteria(Aluno.class);  
  
select.createCriteria("endereco")  
    .add( Restrictions.ge( "numero", new Integer(10) ) );
```

Nós criamos a o objeto Criteria usando a classe Aluno, mas a nossa restrição é para a propriedade "numero" da classe Endereco que está associada a classe Aluno, então temos que "descer" um nível, para poder aplicar a restrição a propriedade "numero" de "endereco". Usando Criteria, você pode usar os métodos estáticos da classe "org.hibernate.criterion.Restrictions" para montar expressões.

Os operadores lógicos (e os parênteses) também estão a sua disposição e em HQL eles continuam com a mesma sintaxe do SQL. Você pode usar *or*, *and* e parênteses para agrupar as suas expressões. Vejamos um exemplo:

```
from Endereco end where  
    ( end.rua in ("Epitácio Pessoa", "Ipiranga") )  
    or ( end.numero between 1 and 100 )
```

Passando para Criteria:

```
Criteria select = sessao.createCriteria(Endereco.class);  
String[] ruas = {"Epitácio Pessoa", "Ipiranga"};  
  
select.add(  
  
    Restrictions.or(  
  
        Restrictions.between("numero", new Integer(1), new Integer(100) ),  
        Restrictions.in( "rua", ruas )  
  
    )  
);
```

O código usando Criteria é muito mais longo do que a expressão em HQL e também é menos legível. Procure utilizar Criteria apenas quando for uma busca que está sendo montada em tempo de execução, se você pode prever a busca, use "named queries" e parâmetros para deixar o eu código mais limpo e mais simples.

Outra parte importante das pesquisas são as funções de pesquisa em Strings. Você pode usar "LIKE" e os símbolos "%" e "_", do mesmo modo que eles são utilizados em SQL. Vejamos um exemplo:

```
from Curso c where c.nome like "Desenvolvimento%"
```

Em Criteria:

```
Criteria select = sessao.createCriteria(Curso.class);  
select.add( Restrictions.like("nome", "Desenvolvimento", MatchMode.START) );
```

Além disso, você ainda pode ordenar as suas pesquisas, em ordem ascendente ou descendente. Como nos exemplos a seguir:

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

Em Criteria:

```
Criteria select = sessao.createCriteria(Aluno.class);  
select.addOrder( Order.asc("nome") );
```

Inicialização tardia (lazy-loading)

Quando você faz uma pesquisa em uma tabela no banco de dados, não necessariamente pesquisa também pelas tabelas que estão relacionadas, porque você pode estar carregando informações que não são necessárias neste cenário da sua aplicação. O Hibernate também não inicializa os relacionamentos 1:N e N:N automaticamente, porque ele pode estar caindo nesse caso de carregar objetos que não são necessários neste caso.

Existem dois modos de se definir a inicialização de relacionamentos 1:N e N:N, o primeiro é definir o atributo “outer-join” da associação (um <set>, por exemplo) para “true”. Isso vai fazer com que a coleção de objetos seja “sempre” inicializada completamente antes de devolver o objeto. Mas isso só funciona para buscas feitas usando a Criteria API, ou os métodos “get()” e “load()”.

O outro modo (e mais indicado) é fazer uma “left join” entre os objetos, em uma busca HQL, assim você só vai carregar a coleção quando ela for necessária. Vejamos como fazer isso no nosso modelo, carregando as Disciplinas de um Curso:

```
from Curso curso  
left join fetch curso.disciplinas  
where curso.nome = :nome
```

Veja que nós adicionamos “left join fetch” a query e depois indicamos a coleção que deveria ser inicializada, neste caso “curso.disciplinas”. Assim, quando essa busca for executada, vai retornar os objetos Curso com as suas respectivas Disciplinas intanciadas.

Conclusão

O Hibernate é um framework incrível, que facilita o desenvolvimento de aplicações que acessam bancos de dados, fazendo com que o programador se preocupe mais com o seu modelo de objeto e seus comportamentos, do que com as tabelas do banco de dados. O Hibernate também evita o trabalho de escrever dúzias de código repetido para fazer as mesmas coisas, como “inserts”, “selects”, “updates” e “deletes” no banco de dados, além de ter duas opções para se montar buscas complexas em grafos de objetos e ainda uma saída para o caso de nada funcionar, usar SQL.

Além de mecanismo de mapeamento objeto/relacional, o Hibernate também pode trabalhar com um sistema de cache das informações do banco de dados, aumentando ainda mais a performance das aplicações. O esquema de cache do Hibernate é complexo e totalmente extensível, existindo diversas implementações possíveis, cada uma com suas próprias características. E junto com isso, ele também tem um gerenciador de versões próprio.

Entretanto, como já foi dito no início do artigo, ele não é a solução perfeita para todos os problemas, aplicações que fazem uso intenso de stored procedures, triggers, user defined functions e outras extensões específicas dos bancos de dados, normalmente não vão se beneficiar do mecanismo oferecido pelo Hibernate. Além disso, aplicações que tem uma alta frequência de “movimentos em massa”, como seqüências de “inserts”, “updates” e “deletes” terminaria por perder performance, graças a instanciação e carregamento de diversos objetos na memória.

Este artigo é apenas um aperitivo para o que o Hibernate pode realmente fazer por você, consulte a referência oficial, leia os JavaDocs e, se puder, compre o livro do Gavin King e do Christian Bauer, Hibernate em Ação (originalmente “Hibernate in Action”), é um adendo valioso a sua biblioteca, não apenas pelo conteúdo relacionado ao Hibernate, mas pelas boas práticas de desenvolvimento e arquitetura de sistemas.

Maurício Linhares (mauricio.linhares@gmail.com) é estudante de Desenvolvimento de Software para a Internet no [CEFET-PB](#), de Comunicação Social (com habilitação em Jornalismo) na [UFPB](#) e membro do grupo Comunidade Livre Paraíba (<http://www.softwarelivre.pb.gov.br/>).

Referências e recursos

- ✓ [Hibernate In Action](#). Bauer, Christian. King, Gavin. Editora Manning, 2004
- ✓ [Hibernate: A Developer's Notebook](#). Elliot, James. Editora O'Reilly, 2004
- ✓ [Documentação do Hibernate 3](#), vários autores.

- ✓ [DbDesigner](#) (ferramenta de criação de bancos de dados MySQL)
- ✓ [Jude](#) (ferramenta de modelagem UML Java)