Universidade Estadual do Norte do Paraná

***Campus* Luiz Meneghel**

Persistência de dados com Java e Hibernate Annotations

André Roberto Ortoncelli

Carlos Alberto Jóia Lazarin

Bandeirantes, 2009

HIBERNATE ANNOTATIONS

Introdução

Nas linguagens de programação modernas como Java, o conceito de orientação a objetos esta cada vez mais difundido, os dados são manipulados no formato de objetos, porém na maioria das vezes são persistidos em bancos de dados relacionais, pois os bancos de dados orientados a objetos não estão tão desenvolvidos quanto os relacionais, devido a falta de robustez e eficiência.

Como uma alternativa para esse obstáculo, foram desenvolvidas as ferramentas de mapeamento objeto/relacional (MOR), e dentre essas ferramentas se destaca o Hibernate.

O Hibernate é um framework para mapeamento objeto/relacional em Java, que abstrai o código SQL da aplicação, permitindo, entre outra coisas, modificar a base de dados para outro SGBD (Sistema Gerenciador de Banco de Dados) sem modificar uma linha de código Java.

O Hibernate Annotations foi criado para que o numero de arquivos necessários para o mapeamento dos objetos fossem reduzidos, já que sem o pacote Annotations é necessário criar um arquivo de mapeamento para cada tabela da base de dados utilizada, e com ele basta adicionar anotações a classe Java.

1. Mapeamento Objeto/Relacional

A maneira mais comuns de se armazenar dados é em bases de dados relacionais, porém as linguagens orientadas a objeto vêm se desenvolvendo muito e torna-se necessário que a interação entre os bancos de dados relacionais ocorra da maneira mais funcional e simples possível.

Para que essa comunicação ocorra é necessário converter objetos em tabelas e tabelas em objetos, e muitas vezes os dados não são compatíveis (os tipos de dados de uma linguagem não são compatíveis com os do banco de dados).

O mapeamento Objeto/Relacional faz a transformação entre objetos e linhas de tabelas como a ilustra a figura 1, com um exemplo de armazenamento da cidade que contem uma objeto estado, nas tabelas estado e cidade, sendo que a tabela cidade possui uma chave estrangeira de estado.

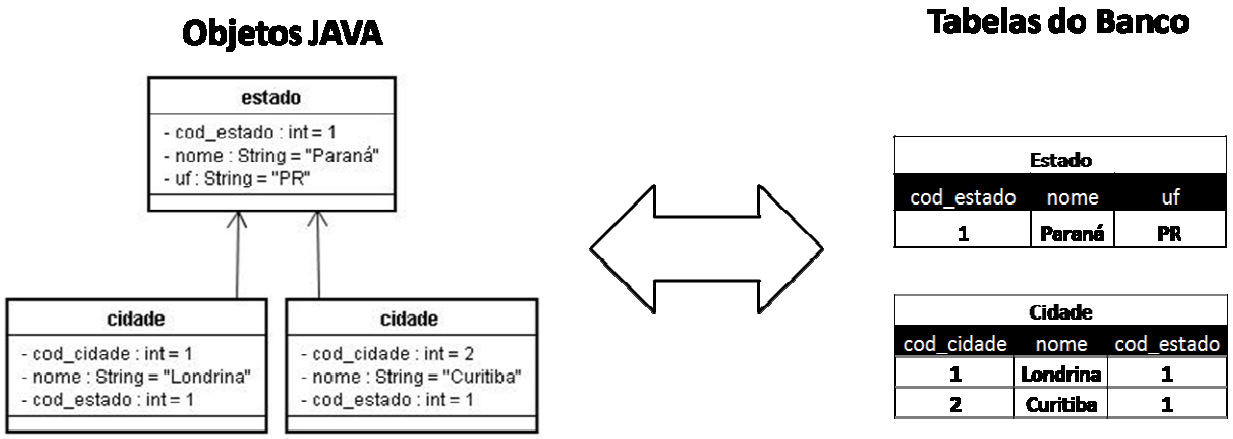


Figura 1: Mapeamento Objeto/Relacional

2. Hibernate

O Hibernate é um framework *open source* de mapeamento objeto/relacional desenvolvido em Java, ou seja, ele transforma objetos definidos pelo desenvolvedor em dados tabulares de uma base de dados, portanto com ele o programador se livra de escrever uma grande quantidade de código de acesso ao banco de dados e de SQL.

Se comparado com a codificação manual e SQL, o Hibernate é capaz de diminuir 95% das tarefas relacionadas a persistência.

2.1 Vantagens do Hibernate

A utilização de código SQL dentro de uma aplicação agrava o problema da independência de plataforma de banco de dados e complica, em muito, o trabalho de mapeamento entre classes e banco de dados relacional.

O Hibernate abstrai o código SQL da nossa aplicação e permite escolher o tipo de banco de dados enquanto o programa está rodando, permitindo mudar sua base sem alterar nada no seu código Java.

Além disso, ele permite criar suas tabelas do banco de dados de um jeito bem simples, não se fazendo necessário todo um design de tabelas antes de desenvolver seu projeto que pode ser muito bem utilizado em projetos pequenos.

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.

2.2 Arquitetura

A arquitetura do Hibernate é formada basicamente por um conjunto de interfaces. A figura 2 ilustra 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. As interfaces do Hibernate ilustradas na pela Figura 1 podem ser classificadas como:

- Interfaces chamadas pela aplicação para executar operações básicas do *CRUD* (***c**reate, **r**etreive, **u**pdate, **d**elete*). Essas são as principais interfaces de dependência entre a lógica de negócios/controle da aplicação e o Hibernate. Estão incluídas *Session*, *Transaction* e *Query*.
- Interfaces chamadas pela infra-estrutura da aplicação para configurar o Hibernate, mais especificamente *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.

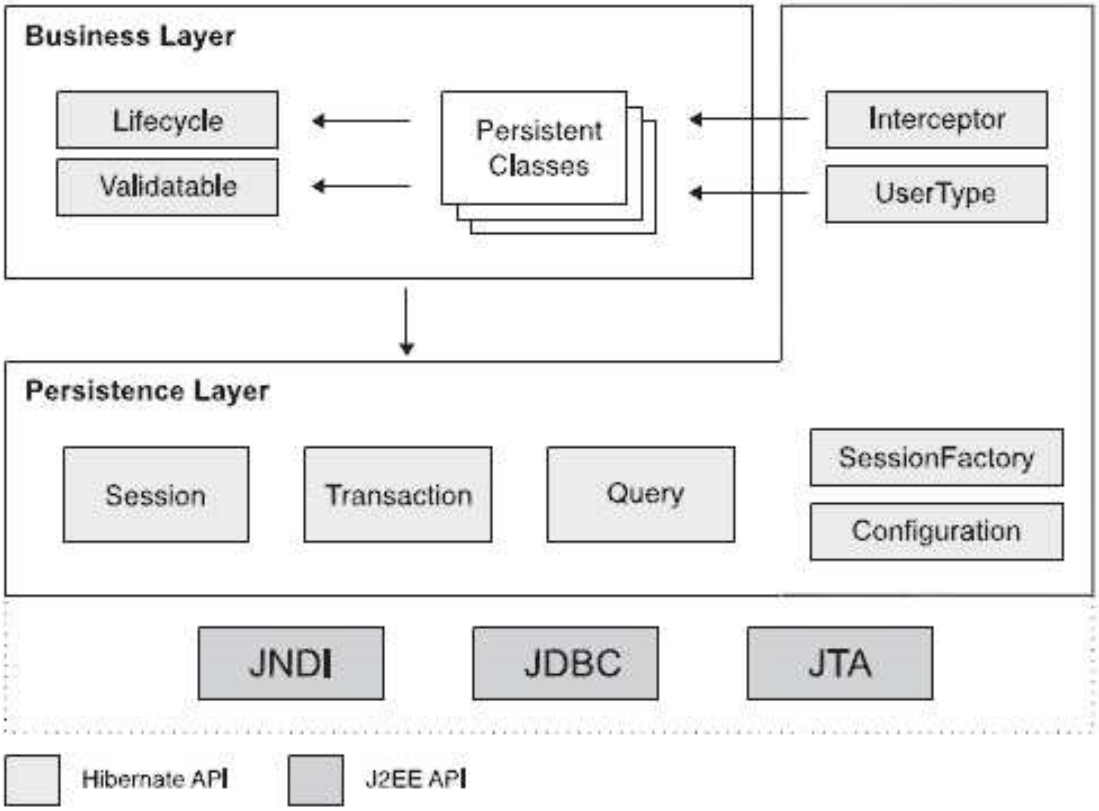


Figura 2: Arquitetura do Hibernate.

As interfaces mais importantes ilustradas na figura acima são *Session, Session Factory, Transaction, Query* e *Configuration.*

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*[1]. Um objeto *Session* possui um *cache* local de objetos recuperados na sessão. Com ele é possível criar, remover, atualizar e recuperar objetos persistentes.

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.

Configuration (org.hibernate.Configuration)

Um objeto *Configuration* é utilizado para realizar as configurações de inicialização do Hibernate. Com ele, definem-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.

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.

---

1. *Thread safe:* Dizemos que uma classe é *thread safe* quando está pronta para ter uma instância utilizada entre várias *threads* concorrentemente.

Interfaces Criteria e Query

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

2.3 Ciclo de Vida

2.3.1 Estados de instância

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 objeto *Session* do Hibernate é o contexto persistente.

- **Transiente:** A instância não é, e nunca foi associada com nenhum contexto persistente. Não possui uma identidade persistente (valor da primary key).
- **Persistente:** A instância está atualmente associada a um contexto persistente. Possui uma identidade persistente (valor da primary key) e, talvez, correspondente a um registro no banco de dados. Para um contexto persistente em particular, o Hibernate *garante* que a identidade persistente é equivalente a identidade Java (na localização em memória do objeto).
- ***Destached*:** A instância foi associada com um contexto persistente, porém este contexto foi fechado, ou a instância foi serializada por outro processo. Possui uma identidade persistente, e, talvez, corresponda a um registro no banco de dados. Para instâncias desatachadas, o Hibernate não garante o relacionamento entre identidade persistente e identidade Java

A Figura 3 ilustra o ciclo de vida de persistência do Hibernate.

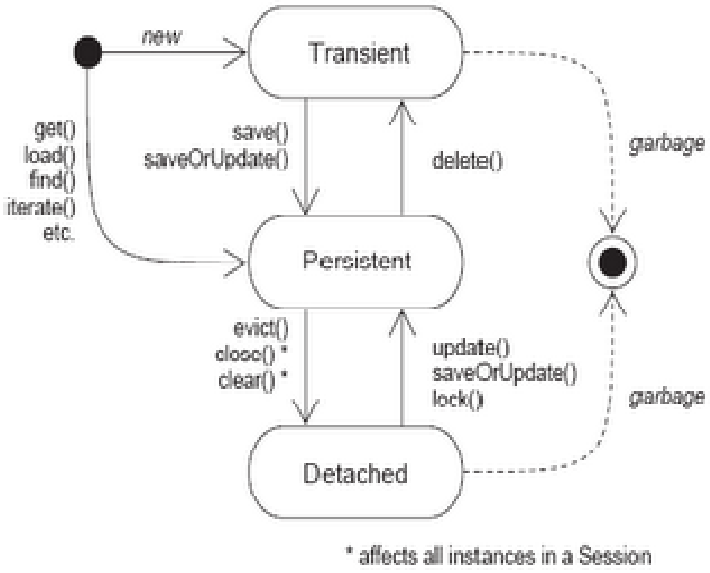


Figura 3: Ciclo de vida

2.4 Download do Hibernate

O Hibernate esta atualmente na versão 3, e seu download pode ser facilmente realizado na pagina oficial do hibernate (www.hibernate.org), onde também podem ser encontrados entre outras coisas a documentação oficial do Hibernate.

Na pagina oficial do Hibernate há um link denominado download, localizado no menu da lateral esquerdo do site, clique nesse link para visualizar as opções de downloads do Hibernate.

Nessa apostila iremos fazer uso do pacotes do hibernate-core, e do hibernate-annotations.

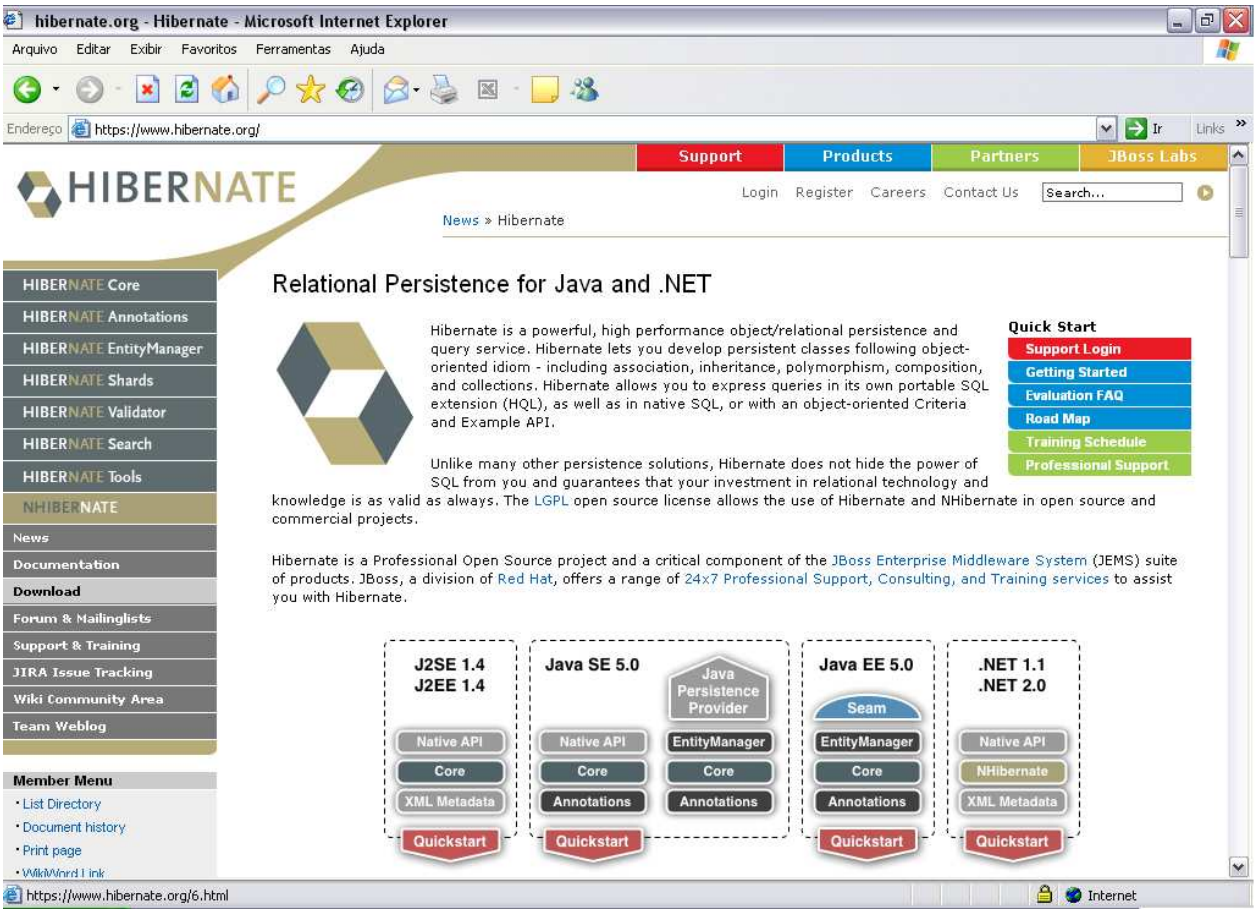


Figura 4: Site oficial do Hibernate

2.5 Adicionando o Hibernate a uma aplicação Java

Essa apostila mostrara como adicionar o Hibernate a uma aplicação Java utilizando a IDE NetBens. É importante ressaltar que a partir da versão 6.5 do NetBens as bibliotecas do Hibernate e do Hiberante Annotations já vem por padrão junto com a IDE.

Se você estiver utilizando uma versão do NetBens anterior a 6.5, você deve descompactar os arquivos .rar do *hibernate-core* e do *hiberante-annotations* em duas pastas distintas.

Da pasta que contem os arquivos do *hibernate-core* devem ser importados os arquivos *hibernate3.jar*, *hibernate-tensting.jar* e todos os arquivos .jar que estiverem na pasta *lib* e em suas subpastas.

E do *hibernate-annotations* deve ser importado o arquivo *hibernate-annotations.jar* e todos os arquivos *.jar* que estiverem na pasta *lib* e em suas subpastas.

Para adicionar esses arquivos a aplicação, clique com o botão direito do mouse em Bibliotecas (*Library*) e depois clique em Adicionar JAR (*Add JAR/Folder*).

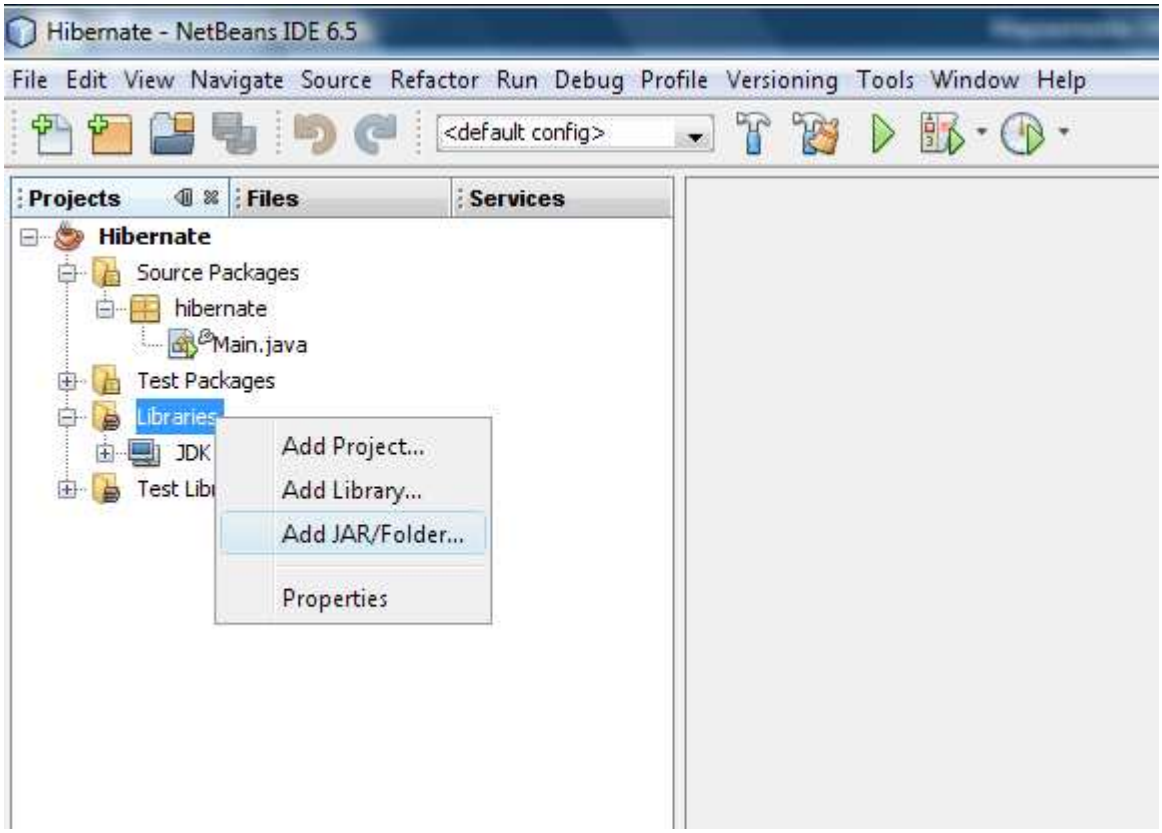


Figura 5: Adicionando arquivos .jar

O próximo passo é selecionar os arquivos necessários na janela que ira aparecer e adicioná-los.

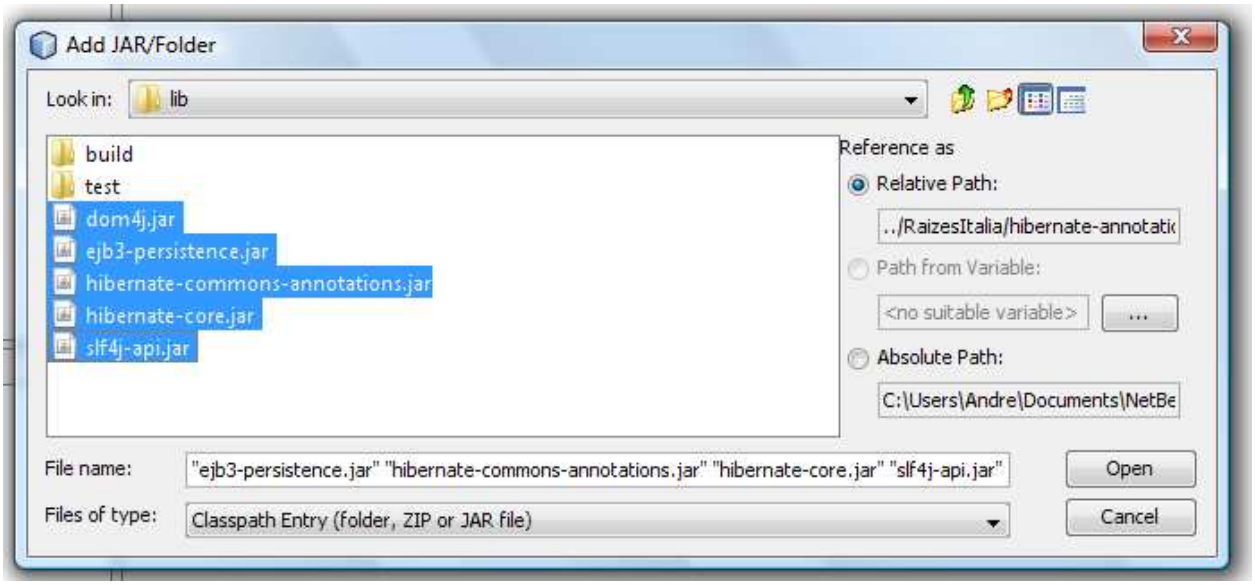


Figura 6: Adicionando arquivos .jar do Hibernate

Se a sua versão do NetBens for a 6.5 ou superior, você não precisa importar os arquivos .jar, basta adicionar as bibliotecas Hibernate e HibernateAnnotations, clicando com o botão direito do mouse em Bibliotecas (*Library*) e depois clique em Adicionar Biblioteca (*Add Library*).

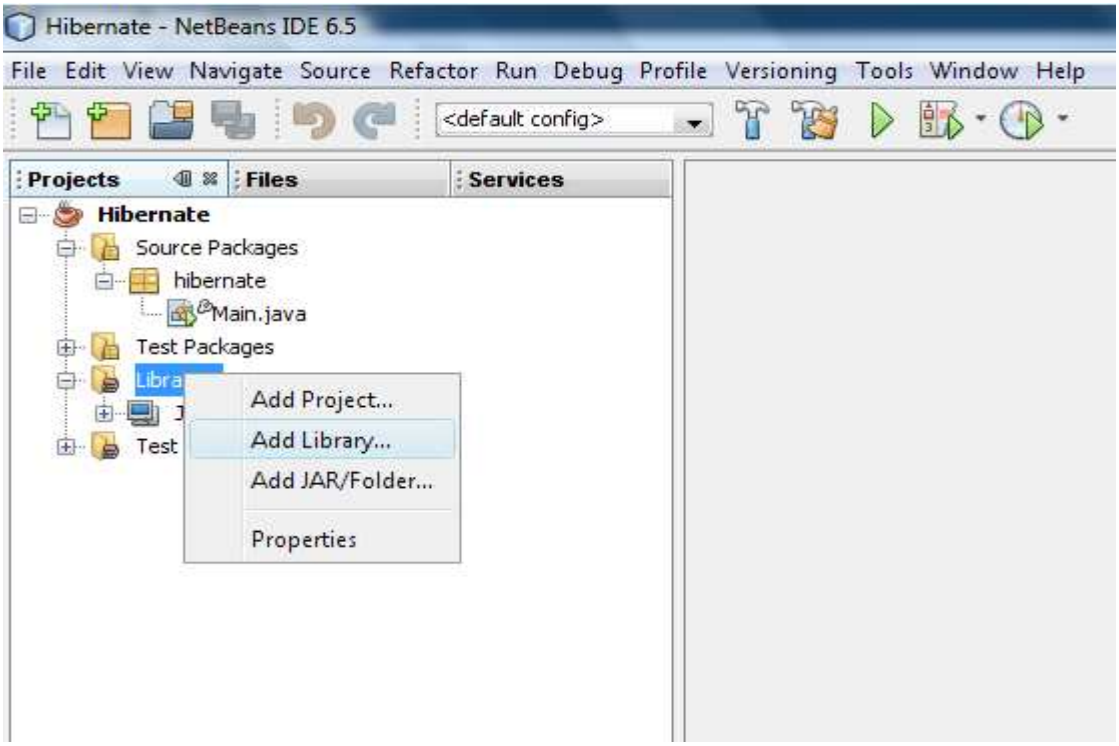


Figura 7: Adicionando bibliotecas

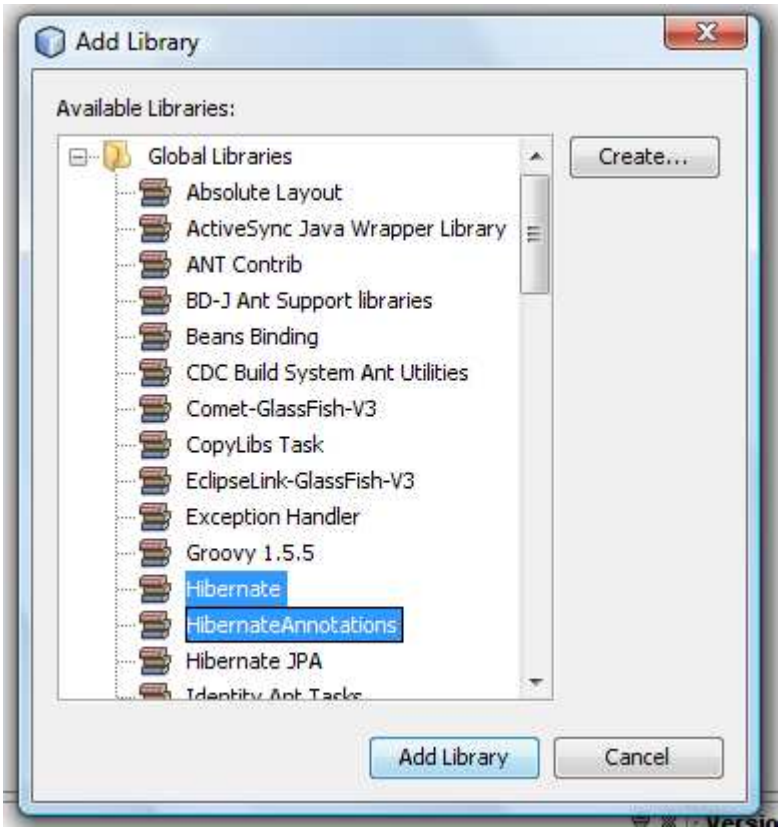


Figura 8: Adicionando as bibliotecas do Hiberante

2.6 Configuração

Antes de começar a trabalhar com Hibernate é necessário configurar a conexão entre o Hibernate e a base de dados, o que pode se feito de três maneiras:

- Instanciar um objeto através de *org.hibernate.cfg.Configuration* para configuração e a adicionar a ele as propriedades necessárias de maneira programática.
- Através de um arquivo *.properties* indicando os arquivos de configuração programaticamente.
- Criar um arquivo XML, denominado de *hibernate.cfg.xml*, com as propriedades de inicialização e a localização do arquivos de mapeamentos, ou das classes com as anotações se você estiver usado o pacote Hibernate Annotations.

Nessa apostila nos iremos utilizar o arquivo configuração de *hiberante.cfg.xml*, por ser a forma mais simples de configuração.

Um modelo de arquivo de configuração será listado a seguir, é importante lembrar que esse arquivo deve ficar fora de qualquer pacote existente na aplicação, portanto utilizando a IDE NetBens o coloque dentro de src.

```
<?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>
      <session-factory>
        <property name="connection.url">
          jdbc:postgresql://localhost:5432/pessoa
        </property>
        <property name="connection.driver_class">
            org.postgresql.Driver
        </property>
        <property name="connection.username"> postgres </property>
        <property name="connection.password"> info </property>
        <property name="dialect">
              org.hibernate.dialect.PostgreSQLDialect
        </property>
        <mapping class="negocios.Cidade" />
        <mapping class="negocios.Estado" />
      </session-factory>
    </hibernate-configuration>
```

Código do arquivo de configuração hibernate.cfg.xml

O modelo listado acima representa a estrutura básica de um arquivo de mapeamento.

O arquivo começa normalmente com a definição do DTD, e depois pelo elemento raiz <hiberante-configuration> seguido pelo elemento <session-factory>, que é onde se inicia a conexão com o banco de dados.

As configurações ficam nos elementos <property>, onde cada um possui um elemento name, indicando sua funcionalidade, agora veremos todas as funcionalidas dos atributos name do nosso arquivo de configuração:

- **connection.url:** especifica a URL de conexão com o banco de dados, no nosso caso utilizamos a base de dados pessoa.
- **connection.driver_class:** nome da classe do driver JDBC utilizado, no caso o driver é o PostgreSQL.

- **connection.username:** o nome de usuário com o qual deve-se conectar ao banco de dados.
- **connection.password:** a senha com a qual deve-se conectar ao banco de dados.
- **dialect:** especifica o dialeto com o qual o hibernate ira se comunicar com a base de dados, ou seja, a utilização do dialeto SQL especifico para cada base de dados.

No arquivo de configuração também existe o elemento <mapping class>, que relaciona a classe tabela mapeada, com a sua respectiva classe Java.

Se não estivéssemos utilizando o hibernate annotations, ao invés de utilizarmos o elemento <mapping class> utilizaríamos o elemento <mapping resource> que indiaria um arquivo de mapeamento para cada tabela.

No exemplo foi utilizado o banco de dados PostgreSQL, mas a maioria dos bancos e dados mais populares também são suportados pelo Hibernate (como DB2, Oracle, MySQL, ...) basta somente modificar os atributos connection.url, connection.driver_class, e dialect, assim em apenas três linhas se pode modificar o banco de dados utilizado.

É importante ressaltar que se deve também importar o driver JDBC do banco de dados junco com as bibliotecas do projeto.

2.7 Mapeamento

Como os bancos e dados não entendem dados orientados a objetos, a solução utilizada pelo Hibernate é utilizar um identificador não natural, assim o banco de dados é capaz de compreender os objetos e montar seus relacionamentos.

Assim o mapeamento consistem em relacionar cada campo de uma tabela da base de dados a uma variável de uma classe, e também montar os identificadores não naturais.

No Hibernate, há duas maneiras de fazer o mapeamento, via XML, ou utilizando anotações.

2.7.1 Mapeamento via XML

No mapeamento via XML são criados arquivos XML que devem ter a extensão .hbm.xml, e também devem ser referenciados no arquivo de configuração.

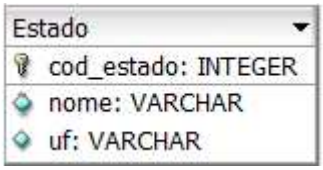
A desvantagem desse tipo de mapeamento é que para cada tabela da base de dados deve ser criado um arquivo de mapeamento e uma classe *POJO*.

Esse processo é mais trabalhoso que o mapeamento utilizando anotações, porem, há ferramentas, como o *XDoclet*, que são utilizadas para gerar os mapeamentos.

2.7.2 Mapeamento via Anotações

Com o mapeamento via anotações, não é necessário criar nenhum arquivo XML (ou em qualquer outro formato) para fazer o mapeamento, basta somente colocar as anotações (annotations) na classe *POJO* relacionada a tabela.

Abaixo veremos um exemplo simples, do mapeamento de uma tabela Estado, utilizando anotações.



Modelagem da tabela estado

1. POJO: São classes com apenas metodos simles (get e set).

```java
@Entity
@Table(name="estado")
public class Estado
{

    @Id
    @SequenceGenerator( name = "cod_estado", sequenceName = "cod_estado_seq", allocationSize = 1 )
    @GeneratedValue( strategy = GenerationType.SEQUENCE, generator = "cod_estado" )
    @Column (name = "cod_estado", nullable = false)
    private int codEstado;
    @Column (name="nome")
    private String nome;
    @Column (name="uf")
    private String uf;
```

Código do mapeamento da tabela estado

Para que o código acima funcione corretamente, devem ser adicionados a ele os construtores e métodos get e set para cada variável.

Também se pode ver que há varias anotações no código anterior, agora veremos para que serve cada uma delas:

@Entity: declara a classe como uma entidade, ou seja, uma classe persistente.

@Table: define qual tabela da base de dados será mapeada.

@Id: define qual campo será usado como identificador.

@GeneretedValue: identifica a estratégia de geração de identificadores, no nosso caso utilizamos sequences.

@SequenceGenerator: como utilizamos a sequence para gerar o identificador, essa anotação serve para referenciar o sequence ao id.

@Column: define qual coluna da tabela será mapeada.

2.8 Relacionamentos

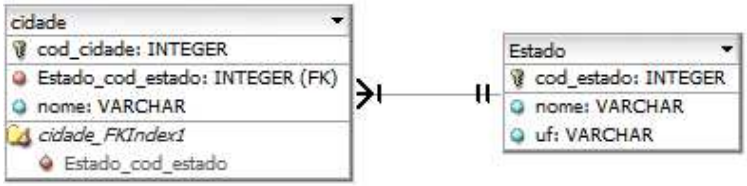
Com o Hibernate é possível mapear todos os relacionamentos existentes entre as tabelas de um banco de dados relacional.

A seguir veremos como fazer o mapeamento de cada um dos relacionamentos.

2.8.1 One-to-many

Aqui veremos como mapear um relacionamento One-to-Many, entre as tabelas estado e cidade.

Como se pode observar no diagrama abaixo, para cada estado, poderá haver varias cidades, porem para cada cidade pode pertencer a apenas um estado.



Modelagem das tabelas cidade e estado

```
@Entity
@Table(name="estado")
@SuppressWarnings ("serial")
public class Estado implements Serializable
{

    @Id
    @SequenceGenerator( name = "cod_estado", sequenceName = "cod_estado_seq", allocationSize = 1 )
    @GeneratedValue( strategy = GenerationType.SEQUENCE, generator = "cod_estado" )
    @Column (name = "cod_estado", nullable = false)
    private int codEstado;
    @Column (name="nome")
    private String nome;
    @Column (name="uf")
    private String uf;
    @OneToMany(mappedBy="estado", fetch = FetchType.LAZY)
    @Cascade(CascadeType.SAVE_UPDATE)
    private Collection <Cidade> cidade;
```

Código Fonte de mapeamento da tabela estado

Nesse código a anotação *@OneToMany*, que indica que se trata de um relacionamento um para muitos, e também pode-se observar os atributos *cascade* e *mappedBy,* dentro dessa anotação.

O atributo *cascade* indica como as alterações na entidade serão refletidas na entidade relacionada no banco de dados, enquanto que o atributo *mappedBy* indica a propriedade que tem o relacionamento com a tabela do lado n (no nosso caso, cidade).

2.8.2 Many-to-one

Nesta seção estudaremos o mapeamento do relacionamento *Many-to-one*, entre as tabelas cidade e estado, utilizando o mesmo diagrama da seção anterior.

```java
@Entity
@Table(name="cidade")
@SuppressWarnings ("serial")
public class Cidade
{
    @Id
    @SequenceGenerator( name = "cod_cidade", sequenceName = "cod_cidade_seq", allocationSize = 1 )
    @GeneratedValue( strategy = GenerationType.SEQUENCE, generator = "cod_cidade" )
    @Column (name = "cod_cidade", nullable = false)
    private int codCidade;
    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name="cod_estado", insertable=true, updatable=true)
    @Fetch(FetchMode.JOIN)
    @Cascade(CascadeType.SAVE_UPDATE)
    private Estado estado;
    @Column (name = "nome")
    private String nome;
```

Código Fonte de mapeamento da tabela cidade

No código acima, a anotação *@ManyToOne*, é utilizada para indicar que se trata de um relacionamento de muitos para um, também se pode perceber que existem os atributos *joinColumn, fetch,* e *cascade* (que já foi visto no topico anterior).

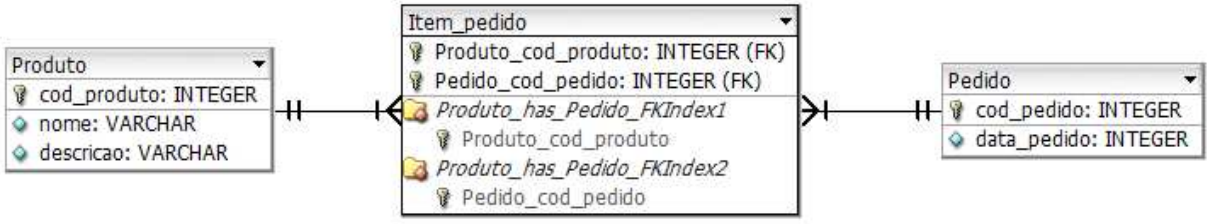
O atributo *JoinColumn* é usado para especificar a coluna que contem a chave estrangeira de cada relacionamento, e a anotação *fetch* com *FetchMode.JOIN*, indica que as consultas vão ser feitas através de um *join*, se no lugar de *FetchMode.JOIN* for colocado *FetchMode.SELECT* a consulta vai ser feita por dois *selects*.

Já quando o atributo *fetch* recebe *FetchType .EAGER* indica que sempre que o objeto pai for buscado na base de dados, seu conteúdo também será trazido, se no lugar de *FetchType.EAGER* for colocado *FetchType.LAZY*, o conteúdo do objeto pai só será trazido a primeira vez que o objeto pai for trazido da base de dados.

2.8.3 Many-to-many

Aqui veremos o mapeamento do relacionamento *Many-to-many*, entre as tabelas pedido e produto.

O diagrama abaixo representa que no relacionamento entre as tabelas pedido e produto. Um pedido pode conter vários produtos, e um produto pode estar em vários pedidos, e pelo fato de se tratar de um relacionamento de muitos para muitos, é gerada uma terceira tabela denomina item_pedido.



Modelagem das tabelas produto, item_pedido e pedido

```
@Entity
@Table(name="pedido")
public class Pedido implements Serializable {

    @Id
    @SequenceGenerator( name = "cod_pedido", sequenceName = "cod_pedido_seq",allocationSize = 1 )
    @GeneratedValue( strategy = GenerationType.SEQUENCE, generator="cod_pedido")
    @Column(name="cod_pedido", nullable = false)
    private int codPedido;
    @Column (name = "data_pedido")
    @Temporal(TemporalType.DATE)
    private Date dataPedido;
    @ManyToMany(fetch=FetchType.LAZY)
    @JoinTable(name="Item_pedido", joinColumns={@JoinColumn(name="cod_pedido")},
                        inverseJoinColumns={@JoinColumn(name="cod_produto")})
    @Cascade(CascadeType.ALL)
    private Collection <Produto> produto;
```

Código Fonte de mapeamento da tabela pedido

```
@Entity
@Table(name="Produto")
public class Produto implements Serializable{

    @Id
    @SequenceGenerator( name = "cod_produto", sequenceName = "cod_produto_seq",allocationSize = 1 )
    @GeneratedValue( strategy = GenerationType.SEQUENCE, generator="cod_produto")
    @Column(name="cod_produto", nullable = false)
    private int codProduto;
    @Column (name = "nome")
    private String nome;
    @Column (name = "descricao")
    private String descricao;
    @ManyToMany(fetch=FetchType.LAZY)
    @JoinTable(name="Item_pedido", joinColumns={@JoinColumn(name="cod_produto")},
                          inverseJoinColumns={@JoinColumn(name="cod_pedido")})
    @Cascade(CascadeType.ALL)
    private Collection<Pedido> pedido;
```

Código Fonte de mapeamento da tabela produto

```java
@Embeddable
public class ItemPedidoPK implements Serializable {

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name="cod_produto")
    @Cascade(CascadeType.ALL)
    private Produto produto;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "cod_pedido")
    @Cascade(CascadeType.ALL)
    private Pedido pedido;
```

Código Fonte da classe ItemPedidoPK

```java
@Entity
@Table(name="Item_pedido")
public class ItemPedido implements Serializable {

    @EmbeddedId
    private ItemPedidoPK chaveComposta; //chave composta da tabela
```

Código Fonte da classe ItemPedido

Quando se mapeia um relacionamento de muitos para muitos se usa a anotação *@ManyToMany*, nas classes que tem o relacionamento, no caso pedido e pessoa, e como a tabela gerada por esse relacionamento possui criar uma classe (um novo tipo de dado em Java), que depois será referenciado como chave primaria.

No nosso caso a classe itemPedido foi criada por causa do relacionamento de muitos para muitos entre Pedido e Produto, e a classse ItemPedidoPK foi criada para ser o tipo da chave composta de ItemPedido.

Na classe ItemPedido, pode se observar que há duas anotações *@ManyToOne*, um entre ItemPedido e Pedido, e um entre ItemPedido e Produto.

E na classe, há uma anotação *@EmbeddedId*, que indica que a chave primaria da tabela ItemPedido é composta.

Nas classes Pedido e Produto pode-se perceber que existe a anotação *JoinTable* que indica quais são as chaves primarias que vão formar a chave composta da tabela gerada pelo relacionamento *ManyToMany*, e também seu nome.

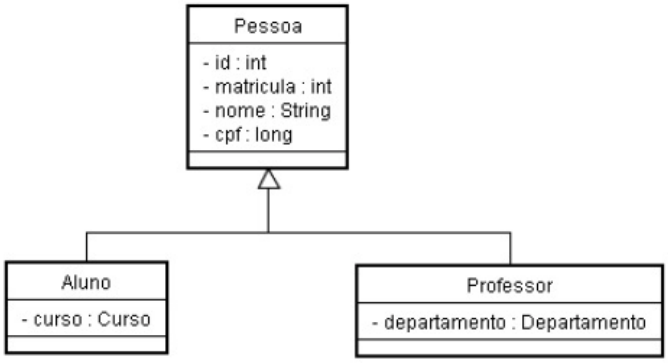
Na classe Pedido há também a anotação *@Temporal*, que deve ser usada toda vez que trabalhar com atributos do tipo date no Hibernate.

2.9 Herança

Como uma boa *engine* de mapeamento O/R o Hibernate trás suporte para herança e polimorfismo para consultas e persistência dos dados. 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. Em cada classe são definidas colunas para todas as propriedades da classe, inclusive as propriedades herdadas;
- **Tabela por Hierarquia:** todas as classes são mapeadas em uma única tabela;
- **Tabela por Sub-Classe:** mapeia cada tabela, inclusive a classe mãe, para tabelas diferentes.

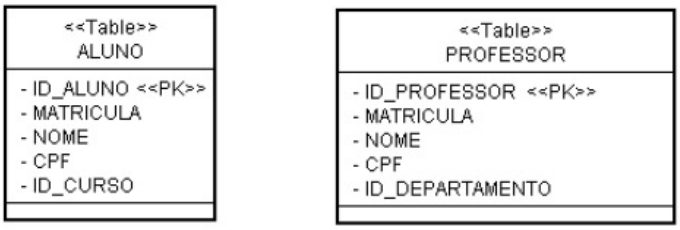
A seguir, como exemplo, está ilustrado na figura abaixo a hierarquia entre Pessoa, Aluno e Professor. É importante salientar que os exemplos a seguir foram extraídos de apostila Fernandes e Lima (2007).



Modelagem da hierarquia da classe Pessoa

A partir da hierarquia apresentada acima iremos exemplificar os três tipos de herança suportadas pelo Hibernate. Os exemplos utilizam *Annotations* para mapeamento, no qual pertence ao escopo desta apostila.

2.9.1 Tabela Por Classe Concreta



Modelagem Tabela por Classe Concreta

Na estratégia tabela por classe concreta, onde são criadas duas tabelas independentes, uma para cada classe filha da classe Pessoa, mapeia-se a classe mãe usando a anotação *@MappedSuperclass* e seus atributos como já apresentado anteriormente.

```
1
2  import javax.persistence.Column;
3  import javax.persistence.GeneratedValue;
4  import javax.persistence.GenerationType;
5  import javax.persistence.Id;
6  import javax.persistence.MappedSuperclass;
7
8
9
10 @MappedSuperclass
11 public class Pessoa {
12
13     @Id
14     @GeneratedValue(strategy = GenerationType.SEQUENCE)
15     @Column(name="id_pessoa")
16     private int id;
17     @Column(name="matricula")
18     private int matricula;
19     @Column(name="nome")
20     private String nome;
21     @Column(name="cpf")
22     private long cpf;
23
24     //Métodos getters e setters...
```

Código Fonte da classe Pessoa no mapeamento tabela por classe concreta

Para mapear as subclasses através desta estratégia, utiliza-se a anotação *@Inheritance* informando através do atributo *strategy* a estratégia de tabela por classe

concreta (valor *InheritanceType.TABLE_PER_CLASS*). O Verifica-se nos códigos apresentados que apenas os atributos específicos de cada classe precisam ser mapeados.

```
1
2  import javax.persistence.Entity;
3  import javax.persistence.FetchType;
4  import javax.persistence.Inheritance;
5  import javax.persistence.InheritanceType;
6  import javax.persistence.JoinColumn;
7  import javax.persistence.ManyToOne;
8  import javax.persistence.Table;
9
10 //Anotação que informa que a classe mapeada é persistente
11 @Entity
12 @Table(name="professor", schema="anotacoes")
13 @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
       public class Professor extends Pessoa{
15
16     @ManyToOne(fetch = FetchType.EAGER)
17     @JoinColumn(name = "id_departamento")
18     private Departamento departamento;
19
20     //Métodos getters e setters
```

Código fonte da classe Professor no mapeamento tabela por classe concreta

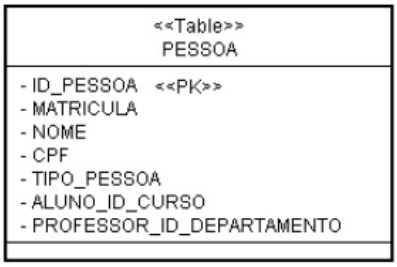
```
1  import javax.persistence.Entity;
2  import javax.persistence.FetchType;
3  import javax.persistence.Inheritance;
4  import javax.persistence.InheritanceType;
5  import javax.persistence.JoinColumn;
6  import javax.persistence.ManyToOne;
7  import javax.persistence.Table;
8
9  @Entity
10 @Table(name="aluno", schema="anotacoes")
11 @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
   public class Aluno extends Pessoa{
13
14     @ManyToOne(fetch = FetchType.EAGER)
15     @JoinColumn(name = "id_curso")
16     private Curso curso;
17
18     //Métodos getters e setters...
```

Código fonte da classe Aluno no mapeamento tabela por classe concreta

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 subclasses são mapeadas em tabelas diferentes, portanto uma associação polimórfica para a classe mãe não seria possível através de chave estrangeira.

Outro problema dessa abordagem é que se uma propriedade é mapeada na superclasse, o nome da coluna deve ser o mesmo em todas as tabelas das subclasses. Dessa forma, várias colunas diferentes de tabelas distintas compartilham da mesma semântica, podem 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.

2.9.2 Tabela Por Hierarquia



Modelagem Tabela por Hierarquia

Em relação à tabela por hierarquia, o mapeamento deve ser feito como mostrado nos exemplos abaixo, onde todos os atributos da classe mãe e das classes filhas são armazenados em uma única tabela. Os mapeamentos de todas as classes são feitos na 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 0, 1 e 2. Esses valores são atribuídos automaticamente pelo framework. O valor 0 representa um objeto persistido do tipo Pessoa, o tipo 1 representa um objeto do tipo Aluno, já o tipo 2, um objeto do tipo Professor. O mapeamento da classe Pessoa, classe mãe, é feito utilizando a anotação *@Inheritance* recebendo o atributo *strategy* com o valor

InheritanceType.SINGLE_TABLE, informando que as três classes serão mapeadas em uma única tabela.

```
2  import javax.persistence.Column;
3  import javax.persistence.DiscriminatorColumn;
4  import javax.persistence.DiscriminatorType;
5  import javax.persistence.DiscriminatorValue;
6  import javax.persistence.Entity;
7  import javax.persistence.GeneratedValue;
8  import javax.persistence.GenerationType;
9  import javax.persistence.Id;
10 import javax.persistence.Inheritance;
11 import javax.persistence.InheritanceType;
12 import javax.persistence.Table;
13
14 @Entity
15 @Table(name = "pessoa", schema = "anotacoes")
16 @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
17 @DiscriminatorColumn(
18 name = "tipo_pessoa",
19 discriminatorType = DiscriminatorType.INTEGER
20 )
21 @DiscriminatorValue("0")
   public class Pessoa {
23
24     @Id
25     @GeneratedValue(strategy = GenerationType.SEQUENCE)
26     @Column(name="id_pessoa")
27     private int id;
28     @Column(name="matricula")
29     private int matricula;
30     @Column(name="nome")
```

Código fonte da classe Pessoa no mapeamento tabela por hierarquia

A anotação *@DiscriminatorColumn* é utilizada para informar a coluna especial que identificará de que tipo será o objeto persistido. O nome da coluna é definido pelo atributo *name* e o seu tipo pelo atributo *discrimitatorType*, que no caso é *Integer*. Por fim, a anotação *@DiscriminatorValue* é utilizada para definir o valor assumido pela coluna tipo_pessoa no caso de se persistir um objeto do tipo Pessoa. Os atributos da classe são mapeados das maneiras apresentadas anteriormente.

Os mapeamentos das subclasses Aluno e Professor são feitos como mostrado nos exemplos abaixo. Essas classes devem ser mapeadas com anotação *@Entity* e

com a anotação *@DiscriminatorValue* para definir o valor assumido pela coluna tipo_pessoa no caso de se persistir um objeto do tipo Aluno ou Professor.

```
1
2  import javax.persistence.DiscriminatorValue;
3  import javax.persistence.Entity;
4  import javax.persistence.FetchType;
5  import javax.persistence.JoinColumn;
6  import javax.persistence.ManyToOne;
7
8  @Entity
9  @DiscriminatorValue("2")
   public class Professor extends Pessoa{
11     @ManyToOne(fetch = FetchType.EAGER)
12     @JoinColumn(name = "professor_id_departamento")
13     private Departamento departamento;
14     //Métodos getters e setters
15     //...
16 }
```

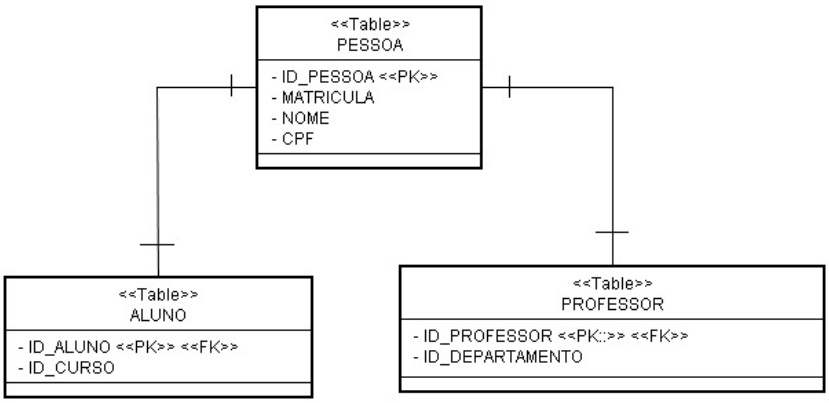
Código fonte da classe Professor no mapeamento tabela por hierarquia

```
1
2  import javax.persistence.DiscriminatorValue;
3  import javax.persistence.Entity;
4  import javax.persistence.FetchType;
5  import javax.persistence.JoinColumn;
6  import javax.persistence.ManyToOne;
7
8  @Entity
9  @DiscriminatorValue("1")
   public class Aluno extends Pessoa{
11     @ManyToOne(fetch = FetchType.EAGER)
12     @JoinColumn(name = "aluno_id_curso")
13     private Curso curso;
14     //Métodos getters e setters
15     //...
16 }
```

Código fonte da classe Aluno no mapeamento tabela por hierarquia

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 subclasses, pois esse mesmo atributo para outra subclasse será nulo.

2.9.3 Tabela Por Subclasse



Modelagem Tabela por Subclasse

A terceira estratégia consiste em mapear cada classe em uma tabela diferente. Nessa estratégia as tabelas filhas contêm apenas colunas que não são herdadas e suas chaves primárias são também chaves estrangeiras para a tabela mãe. O código fonte abaixo apresenta o mapeamento da superclasse Pessoa, onde a estratégia de mapeamento da herança é definida como JOINED, através da anotação *Inheritance(strategy=InheritanceType.JOINED).*

```java
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity
@Table(name = "pessoa", schema = "anotacoes")
@Inheritance(strategy=InheritanceType.JOINED)
public class Pessoa {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    @Column(name="id_pessoa")
    private int id;
    @Column(name="matricula")
    private int matricula;
    @Column(name="nome")
    private String nome;
    @Column(name="cpf")
    private long cpf;

    //Métodos getters e setters...
```

Código fonte da classe Pessoa no mapeamento tabela por subclasse

Os mapeamentos das subclasses Aluno e Professor são feitos como mostrado a seguir. Essas classes devem ser mapeadas com anotação *@Entity* e com a anotação *@PrimaryKeyJoinColumn* para definir qual coluna da tabela filha corresponde à coluna chave primária na tabela mãe, que devem ter o mesmo valor.

```
1
2  import javax.persistence.Entity;
3  import javax.persistence.FetchType;
4  import javax.persistence.JoinColumn;
5  import javax.persistence.ManyToOne;
6  import javax.persistence.PrimaryKeyJoinColumn;
7  import javax.persistence.Table;
8
9  @Entity
10 @Table(name="professor", schema="anotacoes")
11 @PrimaryKeyJoinColumn(name = "id_professor")
   public class Professor extends Pessoa{
13
14     @ManyToOne(fetch = FetchType.EAGER)
15     @JoinColumn(name = "id_departamento")
16     private Departamento departamento;
17     //Métodos getters e setters
18     //...
19
20 }
```

Código fonte da classe Professor no mapeamento tabela por subclasse

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, utilizando a coluna definida na anotação *@PrimaryKeyJoinColumn*. Uma grande vantagem dessa estratégia é que o modelo de relacionamento é totalmente normalizado.

```
1
2  import javax.persistence.Entity;
3  import javax.persistence.FetchType;
4  import javax.persistence.JoinColumn;
5  import javax.persistence.ManyToOne;
6  import javax.persistence.PrimaryKeyJoinColumn;
7  import javax.persistence.Table;
8
9  @Entity
10 @Table(name="aluno", schema="anotacoes")
11 @PrimaryKeyJoinColumn(name = "id_aluno")
   public class Aluno extends Pessoa{
13
14     @ManyToOne(fetch = FetchType.EAGER)
15     @JoinColumn(name = "id_curso")
16     private Curso curso;
17     //Métodos getters e setters
18     //...
19 }
```

Código fonte da classe Aluno no mapeamento tabela por subclasse

2.10 Consultas

Há três formas de realizar consultas com o Hibernate:

- *Criteria Query API* (para montar buscas programaticamente);
- *Hibernate Query Language (HQL);*
- *SQL Query* Nativo*;*

A maioria das suas necessidades deve ser suprida com as duas primeiras alternativas, o resto, você sempre pode usar SQL para resolver.

2.10.1 Criteria Query API

A *Criteria Query API* é um conjunto de classes para a montagem de queries em código Java. Como tudo é definido programaticamente, ganha-se 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.

O código a seguir exemplifica sua utilização:

```
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();
```

O código consiste basicamente em:

- Abrir uma sessão;
- Iniciar uma transação e começar a acessar o banco de dados;
- Criar uma *query* chamando o método *createCriteria(Class class)*, passando como parâmetro a classe que vai ser pesquisada, que no nosso caso é Tuma.
- Para finalizar podemos chamar o método *list()*, que retorna um "List" com os objetos resultantes da *query*.

Pode-se ainda usufruir da interface *Criteria* e seus métodos como, por exemplo, adicionar paginação de resultados.

```
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();
```

Para mais detalhes sobre os métodos da API, e o que é possível fazer a partir dela acesse:

https://www.hibernate.org/hib_docs/v3/api/org/hibernate/Criteria.html

Restrictions

Através da interface *Restrictions* é possível adicionar restrições aos objetos do tipo *Criteria* através do método *add().* Alguns métodos dessa interface exemplificados são:

- Restrictions.eq(“name”, ”Shin”);
- Restrictions.ne(“name”, ”NoName”);
- Restrictions.like(“name”, “Sa%”);
- Restrictions.ilike(“name”, “sa%”);
- Restrictions.isNull(“name”);
- Restrictions.gt(“price”,new Double(30.0));
- Restrictions.between(“age”, new Integer(2), new Integer(10));
- Restrictions.or(criterion1, criterion2);
- Restrictions.disjunction().

Abaixo está exemplificada sua utilização:

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

Para maiores informações de detalhamento das funcionalidades dos métodos da interface acesse *API Restrictions* em:

https://www.hibernate.org/hib_docs/v3/api/org/hibernate/criterion/Restrictions.html

2.10.2 Hibernate Query Language (HQL)

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.

```
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();
```

Há algumas observações sobre o código acima:

- O uso do “as”, que serve para “apelidar” uma classe no 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;
- O acesso as propriedades usando o operador “.” (ponto), utiliza-se sempre esse operador para acessar as propriedades;
- 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*. Com HQL insere-se 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.

Para maiores informações e detalhamento sobre a utilização do HQL acesse a documentação oficial do em: https://www.hibernate.org/5.html.

2.10.3 SQL Query Nativo

A execução de consultas *SQL* nativas é controlada pela interface *SQL Query*, que é obtida chamando *Session.createSQLQuery().* Para exemplificar sua utilização, observe o trecho de código abaixo:

```
sessao.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").list();

OU

sessao.createSQLQuery("SELECT * FROM CATS").list();
```

Ambos retornarão um *List* de *arrays* de *Object (Object[])* com valores escalares por cada coluna da tabela CATS. O Hibernate usará *ResultSetMetaData* para deduzir a ordem atual e os tipos dos valores escalares retornados. Para evitar o overhead ao se usar *ResultSetMetaData* ou simplesmente para ser mais explícito no que é retornado pode-se usar *addScalar()* como está ilustrado no código abaixo:

```
//query de consulta
sessao.createSQLQuery("SELECT * FROM CATS")
//colunas e tipos de retorno explícito
.addScalar("ID", Hibernate.LONG)
.addScalar("NAME", Hibernate.STRING)
.addScalar("BIRTHDATE", Hibernate.DATE)
```

Para maiores informações e detalhamento sobre a utilização do *SQL* nativo para realizar consultas com o Hibernate acesse a documentação em oficial: https://www.hibernate.org/5.html.

3. Conclusão

O Hibernate alem de ser um ótimo framework *open-source* para o mapeamento objeto/relacional, é também a solução mais utilizado hoje em dia.

A principal vantagem do Hibernate é mudança do paradigma estruturado para o de orientação a objetos, eliminado trabalho repetitivo e tedioso.

A desvantagem do Hibernate é o fato de ocorrer uma pequena perca de desempenho, por causa da existência de uma camada intermediaria entre a aplicação e a base de dados, porem em muitos casos as vantagens do Hibernate superam suas desvantagem.

Para sistemas que tem a maior parte da lógica da aplicação na base de dados ou fazem uso extensivo de *stored procedures* e *triggers,* o hibernate pode não ser a melhor opção, mas para aplicações que tem a maior parte da lógica na aplicação Java, os considerados modelos de objetos “ricos”, Hibernate sem dúvida é a melhor opção.

Referencias

Apostila Caelum FJ-21. Java para desenvolvimento Web. Disponível em: http://www.caelum.com.br – acesso em 30/06/2009.

Apostila GUJ - Introdução ao Hibernate 3. Por Maurício Linhares. Disponível em: http://www.guj.com.br/content/articles/hibernate/intruducao_hibernate3_guj.pdf - acesso em 30/06/2009.

BAUER, C. KING, G. Hibernate in Action. Manning Publications. United States of America, 2005.

FERNANDES, R. G.; LIMA, G. A. F. Hibernate com Anotações. 2007. Disponível em: *http://www.j2eebrasil.com.br/index* - acesso em 30/06/2009.

GONÇALVES. E, Desenvolvendo Aplicações Web com JSP, Servlets, JavaServer Faces, Hibernate, EJB 3 Persistence e AJAX, chap. 22, Editora Ciência Moderna.

Hibernate Documentation – Disponível em: https://www.hibernate.org/5.html - acesso em 30/06/2009

UMLAUF. S, FILIPINI. C, "Hibernate dicas e trutques", Mundo Java, No. 10, ano II, pp. 11.