Hibernate com Anotações

Por: Raphaela Galhardo Fernandes
Gleydson de A. Ferreira Lima

_____

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

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

Natal, Maio de 2007

Sumário

1. Conceitos Gerais de Persistência

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

2. Mapeamento Objeto Relacional

Seção Escrita Por: Juliano Rafael Sena de Araújo

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

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

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

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

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

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

2.1 Mapeamento objeto relacional

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

2.1.1 Mapeamento de um objeto com tipos primitivos

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

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

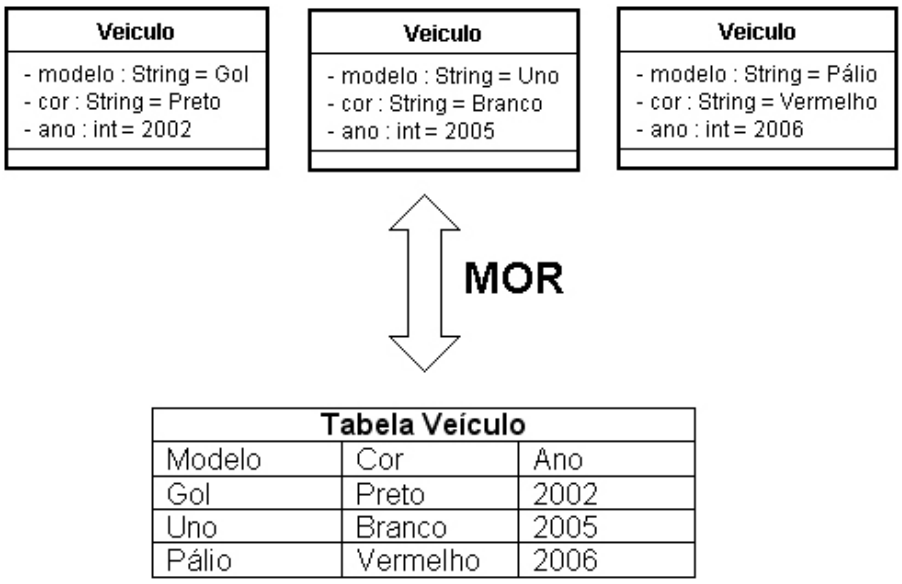

Figura 1 - Exemplo de mapeamento de tipos básicos

2.1.2 Mapeamento de objetos que contém uma coleção de objetos

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

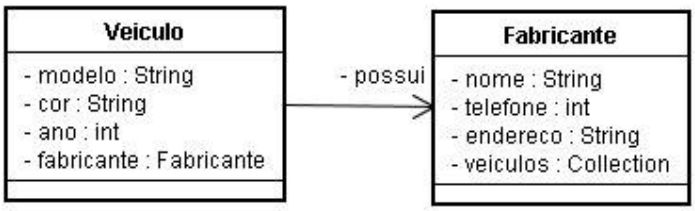

Figura 2 - Relacionamento entre as classes Fabricante e Veiculo

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

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

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

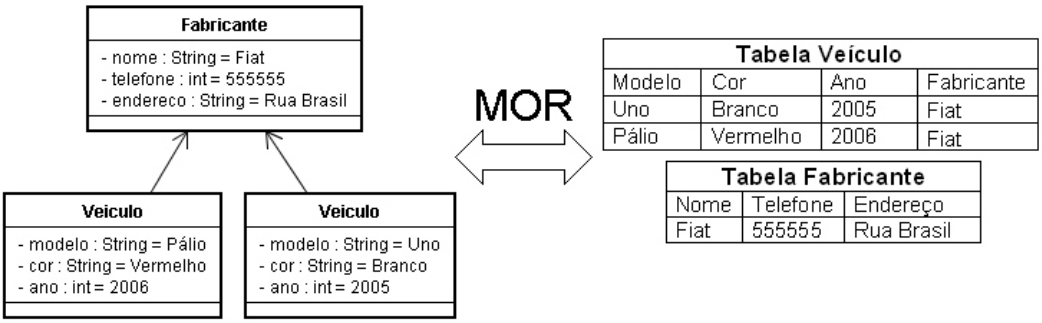

Figura 3 – Exemplo de mapeamento 1 para N

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

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

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

Esse tipo de mapeamento é muito utilizado em aplicações em que os objetos fazem muita referência a outros objetos.

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

3. Introdução ao Hibernate

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

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

4. Arquitetura

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

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

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

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

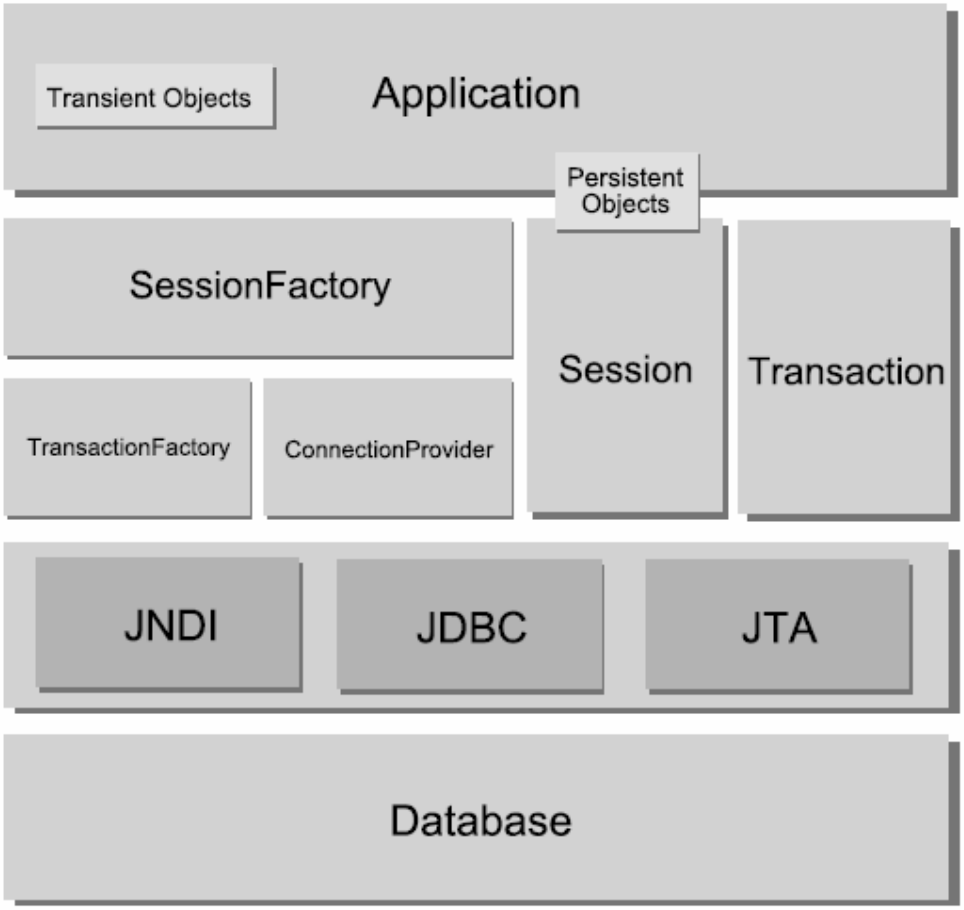

Figura 4 - Arquitetura do *Hibernate*

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

4.1 Session (org.hibernate.Session)

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

4.2 SessionFactory (org.hibernate.SessionFactory)

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

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

4.3 Configuration (org..hibernate.Configuration)

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

4.4 Transaction (org.hibernate.Transaction)

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

4.5 Interfaces Criteria e Query

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

5. Classes Persistentes

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

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

public class Pessoa {

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

    public Pessoa(){}

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

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

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

    public String getNome () { return nome; }
    public void setNome (Stringnome) {
        this.nome = nome;
    }

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

Listagem 1 – Exemplo de Classe POJO

Considerações:

- O *Hibernate* requer que toda classe persistente possua um construtor padrão sem argumentos, assim, o *Hibernate* pode instanciá-las simplesmente chamando `Construtor.newInstance();`
- Observe que a classe `Pessoa` apresenta métodos *setters* e *getters* para acessar ou retornar todos os seus atributos. O *Hibernate* persiste as propriedades no estilo *JavaBeans*, utilizando esses métodos;
- A classe `Pessoa` possui um atributo `id` que é o seu identificador único. É importante que, ao utilizar *Hibernate,* todos os objetos persistentes possuam um identificador e que eles sejam independentes da lógica de negócio da aplicação.

5.1 Identidade/Igualdade entre Objetos

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

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

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

5.2 Escolhendo Chaves Primárias

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

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

satisfazer as seguintes propriedades:

- Única;
- Nunca ser nula;
- Constante.

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

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

O *Hibernate* apresenta vários mecanismos internos para a geração de chaves primárias. Veja a Tabela 1.

Tabela 1 - Mecanismo de Geração de Chaves Primárias

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

6. Objetos Persistentes, Transientes e *Detached*

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

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

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

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

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

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

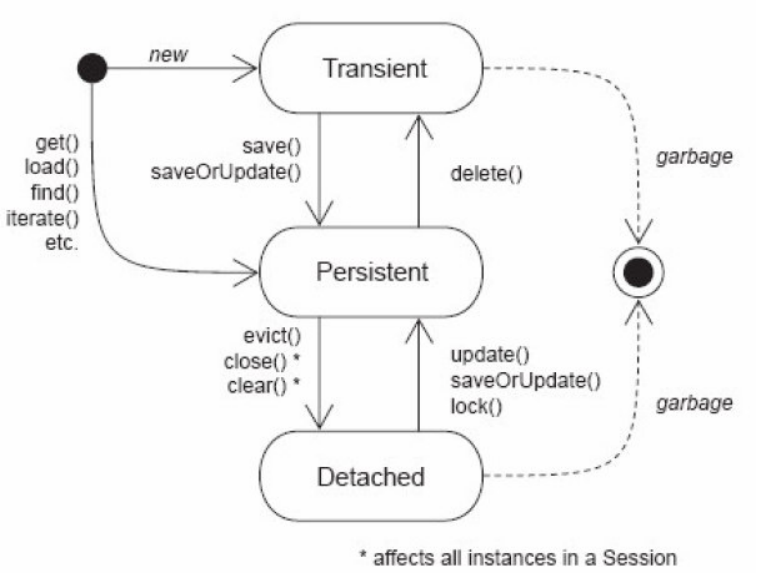

Figura 5: Ciclo de Vida - Persistência

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

7. Como Utilizar o *Hibernate Annotation*

Para utilizar o *Hibernate Annotation*, primeiramente, é preciso copiar sua versão atual do site http://hibernate.org, disponível em um arquivo compactado. Então, este arquivo deve ser descompactado e seu conteúdo consiste em um conjunto de arquivos JARs. Esses arquivos devem ser copiados para o diretório das libs de sua aplicação. Os arquivos `hibernate-annotations.jar` e `lib/ejb3-persistence.jar` também devem referenciados no *classpath* da aplicação, juntamente com a classe do *driver* do banco de dados utilizado.

8. Introdução a Anotações (*Annotations*)

Como já citado, o mapeamento objeto relacional utilizando *Hibernate* pode ser feito a partir de anotações. As anotações podem ser definidas como metadados que aparecem no código fonte e são ignorados pelo compilador. Qualquer símbolo em um código Java que comece com uma @ (arroba) é uma anotação. Este recurso foi introduzido na linguagem Java a partir da versão Java SE 5.0. Em outras palavras, as anotações marcam partes de objetos de forma que tenham algum significado especial.

A Listagem 2 e a Listagem 3 apresentam exemplo de um tipo de anotação denominado de **TesteAnotacao** e do seu uso em um método qualquer (**metodoTeste**), respectivamente. O exemplo é meramente ilustrativo, de forma que o tipo de anotação definido não agrega nenhum significado especial ao código fonte.

```
public @interface TesteAnotacao{
      ...
}
```

Listagem 2 – Exemplo ilustrativo para a criação de um tipo de anotação

```
@TesteAnotacao
public void metodoTeste{
      ...
}
```

Listagem 3 – Exemplo ilustrativo para o uso de um tipo de anotação

As anotações podem possuir nenhum, um ou mais de um elementos em sua definição. Um exemplo de tipo de anotação sem nenhum atributo já foi visto nas Listagem 2 e na Listagem 3. Se um tipo de anotação possui um único elemento, como definido e ilustrado na Listagem 4, no momento de se utilizar a anotação, um valor para este atributo deve ser passo entre parênteses, como mostrado na Listagem 5

```
public @interface TesteAnotacao{
      String elemento1();
}
```

Listagem 4 – Exemplo ilustrativo para a criação de um tipo de anotação

```
@TesteAnotacao("Definindo valor")
public void metodoTeste{
      ...
}
```

Listagem 5 – Exemplo ilustrativo para o uso de um tipo de. Anotação

No caso caso em que o tipo da anotação ser definido com mais de um elemento, como mostrado na Listagem 6, na hora em que a anotação for usada, para atribuir valores para seus elementos é preciso passá-los também entre parênteses, porém definindo o nome do elemento e o seu valor, como exemplo ilustrado na Listagem 7.

```
public @interface TesteAnotacao{
      String elemento1();
      int elemento2();
}
```

Listagem 6 – Exemplo ilustrativo para a criação de um tipo de anotação

```
@TesteAnotacao(elemento1 = "Definindo valor 1", elemento2 = 25)
public void metodoTeste{
      ...
}
```

Listagem 7 – Exemplo ilustrativo para o uso de um tipo de anotação

8.1 Tipos de Anotações Suportadas pelo Java 5.0

Dois tipos de anotações são suportadas pelo Java 5.0: anotações simples e meta anotações. As anotações simples são usadas apenas para agregar significado especial ao código fonte, mas não para criar algum tipo de anotação. Já as meta anotações são aquelas utilizadas para definir tipos de anotações.

Não é o intuito desta seção detalhar cada um dos tipos de anotações existentes no Java 5.0, mas apenas citá-las e apresentar os seus significados.

As anotações simples do Java 5.0 são três:

- **@Override**: indica que o método que possui esta anotação deve ter a sua implementação sobrescrevendo a de sua superclasse. Se o método anotado não tiver implementação, o compilador gerará um erro;

- **@Deprecated**: permite indicar que algum elemento do código fonte está em desuso, ou com uso depreciado. A anotação servirá para que o compilador alerte o programador quanto a utilização de um elemento em desuso;
- **@Suppresswarnings**: esta anotação indica ao compilador que os alertas (*warnings*) no elemento anotado não sejam informados ao programador.

Já as meta anotações são:

- **@Target**: indica em que nível dos elementos da classe a anotação será aplicada, por exemplo: a qualquer tipo; a um atributo, a um método, a um construtor, etc;
- **@Retention**: indica onde e quanto tempo a anotação será considerada. Por exemplo: se apenas a nível de código fonte e ignoradas pelo compilador; se apenas pelo compilador em tempo de compilação e não pela JVM; ou consideradas pela JVM apenas em tempo de execução;
- **@Documented**: indica que o elemento anotado deve ser documentado através de alguma ferramenta javadoc;
- **@Inherited**: indica que a classe anotada com este tipo é automaticamente herdada. Por exemplo, se uma anotação é definida a partir de **@Inherited**, uma classe anotada por esta nova anotação irá herdar automaticamente todos os seus elementos.

Mais informações sobre estes tipos de anotações a partir do artigo "*An Introduction to Java Annotation*" por M. M. Islam Chisty (Outrubro, 2005).

Inicialmente, o mapeamento objeto relacional com *Hibernate* era feito a partir de um conjunto de configurações em arquivos XMLs. Com o surgimento das anotações no Java SE 5.0, o *framework Hibernate* anexou este recurso, permitindo que as classes Java fossem mapeadas a partir de anotações, simplificando o seu uso. A próxima seção apresentará um exemplo simples do uso da persistência de uma classe com *Hibernate Annotations*.

9. Primeiro Exemplo com *Hibernate Annotations*

Inicialmente, para que o *Hibernate* soubesse como carregar e armazenar objetos de classes persistentes, eram utilizados apenas arquivos de mapeamentos XML. Dessa forma, era possível informar que tabela do banco de dados se refere uma dada classe persistente e quais colunas na tabela são referentes a quais atributos da classe. Com o surgimento das anotações no Java 5.0, tornou-se possível substituir os arquivos XML para o mapeamento objeto relacional. Através do uso de um conjunto de anotações no código fonte das classes mapeadas.

Neste primeiro exemplo, será apresentado como realizar o mapeamento objeto relacional da classe **Aluno** (Classe ilustrada na Figura 6) utilizando anotações. Dessa forma, informando ao *Hibernate* que tabela no banco de dados a representa e quais atributos correspondem as quais colunas desta tabela.

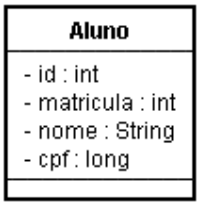

Figura 6 - Classe Aluno

A classe Java que representa a entidade **Aluno** está ilustrada na Listagem 8.

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

public class Aluno {

	//Atributos da classe
	private int id;
	private int matricula;
	private String nome;
	private long cpf;

	//Construtor padrão
	public Aluno(){}

	//Métodos getters e setters
	public long getCpf() { return cpf; }
	public void setCpf(long cpf) { this.cpf = cpf; }
```

```
	public int getId() { return id; }
	public void setId(int id) { this.id = id; }

	public int getMatricula() { return matricula; }
	public void setMatricula(int matricula) {
		this.matricula = matricula;
	}

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

Listagem 8 - Classe de Domínio: Aluno

Para os exemplos ilustrados neste material, a base de dados foi feita utilizando o PostgreSQL 8.2, que pode ser baixado no site www.posgresql.org. A classe **Aluno** foi mapeada para a tabela **aluno** presente no esquema **anotações** do banco criado denominado **jeebrasil**. O script para a criação da tabela **aluno** pode ser visto na Listagem 9.

```
CREATE TABLE anotacoes.aluno
(
  id_aluno integer NOT NULL, -- Identificador da tabela
  matricula integer NOT NULL, -- Matrícula do aluno
  nome character(40) NOT NULL, -- Nome do aluno
  cpf bigint NOT NULL, -- CPF do aluno
  CONSTRAINT pk_aluno PRIMARY KEY (id_aluno),
  CONSTRAINT un_cpf_aluno UNIQUE (cpf),
  CONSTRAINT un_matricula_aluno UNIQUE (matricula)
)
WITHOUT OIDS;
ALTER TABLE anotacoes.aluno OWNER TO postgres;
COMMENT ON COLUMN anotacoes.aluno.id IS 'Identificador da tabela';
COMMENT ON COLUMN anotacoes.aluno.matricula IS 'Matrícula do aluno';
COMMENT ON COLUMN anotacoes.aluno.nome IS 'Nome do aluno';
COMMENT ON COLUMN anotacoes.aluno.cpf IS 'CPF do aluno';
```

Listagem 9 – Script para a Criação da Tabela aluno

Para o mapeamento com anotações das entidades, serão utilizadas tanto anotações do pacote `javax.persistence.*`, como do pacote `org.hibernate.annotations.*`.

Todas as classes persistentes mapeadas com anotações são declaradas usando a anotação **@Entity**, aplicada em nível de classes, como mostrado na Listagem 10. Observa-se que com o uso de anotações, não há mais a necessidade de se utilizar arquivos de mapeamento XML adicionais.

Quando o nome da classe é diferente do nome da tabela para a qual é mapeada é necessário informar na anotação **@Table** qual o nome da tabela, usando o atributo **name**. No caso do mapeamento da classe **Aluno**, não havia a necessidade de se informar o nome da tabela, pois ela e a classe possuem o mesmo nome. Como a tabela pertence a um esquema do banco de dados (**anotações**), no mapeamento da classe, também é necessário informar em que esquema a tabela mapeada se encontra, utilizando o atributo **schema** da anotação **@Table**. No caso, a linha do código fonte `@Table(name="aluno", schema="anotacoes")` está informando o nome e o esquema da tabela para a qual está mapeada a classe aluno.

A chave primária da tabela é mapeada na classe através da anotação **@Id**. O valor atribuído à chave primária pode ser dado tanto pela aplicação quanto por um mecanismo do *Hibernate* que o gere automaticamente. A anotação @**GeneratedValue** permite a definição automática para o valor do identificador, utilizando um dos mecanismos de geração apresentados anteriormente. Neste caso, utilizou-se a estratégia a partir de uma seqüência, como feito na linha de código `@GeneratedValue(strategy = GenerationType.SEQUENCE)`. Dessa forma, na hora de persistir uma linha na tabela **aluno**, o *Hibernate* vai pegar como valor para a chave primária o próximo valor disponível por uma seqüência padrão chamada **hibernate_sequence**. Deve-se salientar que o programador deverá criar uma seqüência com este nome na base de dados. Ainda em relação ao identificador, como o nome da coluna mapeada é diferente do nome do atributo, é necessário utilizar a anotação **@Column** informando o nome da coluna, através do atributo **name**. Neste exemplo, o nome da coluna mapeada para o identificador é **id_aluno**, mapeada da seguinte forma: `@Column(name="id_aluno")`.

Observa-se que nos demais atributos da classe não há nenhuma anotação de mapeamento. Isso pode ser feito quando o nome do atributo é igual ao nome da coluna da tabela mapeada, de forma que não há a necessidade de mapeamento explícito.

Por fim, para se utilizar as anotações para mapeamento das classes, é preciso importá-las de algum lugar, neste caso do pacote `javax.persistence`, como mostrado no início do código fonte da Listagem 10.

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

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

//Anotação que informa que a classe mapeada é persistente
@Entity
//Informando nome e esquema da tabela mapeada
@Table(name="aluno", schema="anotacoes")
public class Aluno {

	//Definição da chave primária
	@Id
	//Definição do mecanismo de definição da chave primária
	@GeneratedValue(strategy = GenerationType.SEQUENCE)
	//Informa o nome da coluna mapeada para o atributo
	@Column(name="id_aluno")
	private int id;
	private int matricula;
	private String nome;
	private long cpf;

	public void Aluno(){}

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

Listagem 10 – Mapeamento da classe Aluno com anotações

Depois de criar todas as classes persistentes com seus respectivos mapeamentos com anotacoes, deve-se realizar algumas configurações do *Hibernate*, mostradas na seção seguinte.

10. Configurando o Hibernate

Pode-se configurar o Hibernate de três maneiras distintas:

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

Será apresentada a configuração a partir do arquivo `hibernate.cfg.xml`. Um exemplo deste arquivo de configuração pode ser visto na Listagem 11. Vários parâmetros podem ser configurados. Basicamente, deve-se configurar:

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

```
<?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>

      <!-- properties -->
      <property name="connection.driver_class">
          org.postgresql.Driver
      </property>
      <property name="connection.url">
          jdbc:postgresql://localhost:5432/jeebrasil
      </property>
      <property name="dialect">
            org.hibernate.dialect.PostgreSQLDialect
      </property>
      <property name="show_sql">true</property>
```

```
      <property name="connection.username">postgres</property>
      <property name="connection.password">postgres</property>
      <property name="connection.pool_size">10</property>

      <!-- mapping classes -->
      <mapping class="br.com.jeebrasil.hibernate.anotacoes.dominio.Aluno"/>

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

Listagem 11 - Arquivo de Configuração hibernate.cfg.xml

Resumindo as descrições das propriedades a serem configuradas:

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

O *Hibernate* trabalha com dialetos para um grande número de bancos de dados, tais como: *DB2, MySQL, Oracle, Sybase, Progress, PostgreSQL, Microsoft SQL Server, Ingres, Informix* entre outros. Possíveis valores para os dilates estão presentes na Tabela 2.

Tabela 2 - Possíveis valores de dialetos

DB2 – org.hibernate.dialect.DB2Dialect
HypersonicSQL – org.hibernate.dialect.HSQLDialect
Informix – org.hibernate.dialect.InformixDialect
Ingres – org.hibernate.dialect.IngresDialect
Interbase – org.hibernate.dialect.InterbaseDialect
Pointbase – org.hibernate.dialect.PointbaseDialect
PostgreSQL – org.hibernate.dialect.PostgreSQLDialect
Mckoi SQL – org.hibernate.dialect.MckoiDialect
Microsoft SQL Server – org.hibernate.dialect.SQLServerDialect
MySQL – org.hibernate.dialect.MySQLDialect
Oracle (any version) – org.hibernate.dialect.OracleDialect
Oracle 9 – org.hibernate.dialect.Oracle9Dialect
Progress – org.hibernate.dialect.ProgressDialect
FrontBase – org.hibernate.dialect.FrontbaseDialect
SAP DB – org.hibernate.dialect.SAPDBDialect
Sybase – org.hibernate.dialect.SybaseDialect
Sybase Anywhere – org.hibernate.dialect.SybaseAnywhereDialect

Já no final do arquivo `hibernate.cfg.xml` é onde devem ser informados os arquivos das classes mapeadas que o *Hibernate* deve processar. Se alguma classe não for definida neste local, a mesma não poderá ser persistida utilizando os mecanismos do *Hibernate*.

Para este exemplo, apenas uma classe foi mapeada. Se existissem outras classes persistentes, as mesmas deveriam ter linhas semelhantes a `<mapping class="br.com.jeebrasil.hibernate.anotacoes.dominio.Aluno"/>`, informando o seu nome para que possam ser persistidas através do *Hibernate*.

11. Manipulando Objetos Persistentes

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

O código fonte exibido na Listagem 12 mostra a criação e persistência de um objeto do tipo **Aluno**. Inicialmente alguns imports de objetos são feitos para se utilizar a persistência com o *Hibernate*. Em seguida, dentro do método **main**, cria-se um objeto que conterá as configurações definidas do arquivo `hibernate.cfg.xml`. A partir deste arquivo, uma sessão para a persistência de objetos é aberta, juntamente com uma transação. Então, criou-se um objeto do tipo **Aluno**, atribuiu-se valores para seus atributos e, por fim, invocou-se o método

save do objeto **Session**. Com isso, uma linha na tabela **aluno** é adicionada de acordo com os valores definidos no objeto. Assim, a persistência desejada é realizada e deve-se liberar tanto a transação quanto a sessão com o *Hibernate*.

```
package br.com.jeebrasil.hibernate.anotacoes.testes;
//Imports de elementos para o uso do Hibernate funcionar
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.cfg.Configuration;
//Import da classe de domínio persistida
import br.com.jeebrasil.hibernate.anotacoes.dominio.Aluno;

public class Listagem12 {
      public static void main(String[] args) {

            //Cria objeto que receberá as configurações
            Configuration cfg = new AnnotationConfiguration();
            //Informe o arquivo XML que contém a configurações
            cfg.configure(
      "/br/com/jeebrasil/hibernate/anotacoes/conf/hibernate.cfg.xml");
            //Cria uma fábrica de sessões.
            //Deve existir apenas uma instância na aplicação
            SessionFactory sf = cfg.buildSessionFactory();

            // Abre sessão com o Hibernate
            Session session = sf.openSession();
            //Cria uma transação
            Transaction tx = session.beginTransaction();

            // Cria objeto Aluno
            Aluno aluno = new Aluno();
            aluno.setNome("Raphaela Galhardo Fernandes");
            aluno.setMatricula(200027803);
            aluno.setCpf(1234567898);

            session.save(aluno); // Realiza persistência
```

```
            tx.commit(); // Finaliza transação
            session.close(); // Fecha sessão
      }
}
```

Listagem 12 - Exemplo de Persistência da Classe Aluno

O código para a criação de um objeto **SessionFactory** deve ser chamado uma única vez durante a execução da aplicação. Objetos deste tipo armazenam os mapeamentos e configurações do *Hibernate* e são muito pesados e lentos de se criar.

A Listagem 13 apresenta o resultado da execução do código fonte presente na Listagem 12. No caso, o *Hibernate* buscou um valor para a chave primária da tabela, utilizando a seqüência padrão **hibernate_sequence**. Em seguida, o comando **INSERT** foi executado para inserir uma linha na tabela **aluno** de acordo com as informações atribuídas ao objeto montado.

```
Hibernate: select nextval ('hibernate_sequence')
Hibernate: insert into anotacoes.aluno
      (matricula, nome, cpf, id_aluno) values (?, ?, ?, ?)
```

Listagem 13 – Resultado da Execução do Código da Listagem 12

A Tabela 3 apresenta alguns dos métodos que podem ser invocados a partir do objeto **Session**.

Tabela 3 - Métodos invocados a partir do objeto `Session`

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

Em relação ao método **saveOrUpdate**, uma questão que se pode formular é a seguinte: "Como o *Hibernate* sabe se o objeto em questão já existe ou não no banco de dados, ou seja, se ele deve ser criado ou atualizado?". A resposta é simples: se o valor atribuído ao identificador for diferente de zero, o objeto deve ter

sua linha na tabela atualizada (deve-se garantir que o valor do identificador do objeto se refere a um valor da chave primária da tabela). Caso contrário, uma nova linha será adicionada à tabela.

A Listagem 14 e a Listagem 15 apresentam exemplos dos métodos invocados a partir do objeto **Session**.

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

      // Busca objeto aluno da base de dados com chave primária = 1
      Aluno aluno = (Aluno) session.get(Aluno.class, 1);
      // Atualiza informação de matrícula.
      aluno.setMatricula(200027807);

      // Como o identificador do objeto aluno é diferente de 0,
      // a sua matrícula é atualizada já que foi alterada
      session.saveOrUpdate(aluno);

      tx.commit();
      session.close();
//...
```

Listagem 14 - Exemplo de Busca e Atualização de um Objeto Aluno

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

      Aluno aluno = new Aluno();
      // Existe linha na tabela aluno com chave primária = 2
      aluno.setId(2);

      // Deleta aluno com id = 2 da tabela.
      // Somente necessária informação do seu identificador
      session.delete(aluno);

      tx.commit();
      session.close();
//...
```

Listagem 15 - Exemplo de Remoção de Objeto Aluno

Os resultados das execuções dos códigos presentes na Listagem 14 e na Listagem 15 podem ser vistos na Listagem 16 e na Listagem 17, respectivamente.

```
Hibernate: select aluno0_.id_aluno as id1_0_0_, aluno0_.matricula as
      matricula0_0_, aluno0_.nome as nome0_0_, aluno0_.cpf as cpf0_0_
      from anotacoes.aluno aluno0_ where aluno0_.id_aluno=?
Hibernate: update anotacoes.aluno set matricula=?, nome=?, cpf=?
      where id_aluno=?
```

Listagem 16 – Resultado da Execução do Código da Listagem 14

```
Hibernate: delete from anotacoes.aluno where id_aluno=?
```

Listagem 17 – Resultado da Execução do Código da Listagem 15

12. Mapeando Atributos da Classe

Esta seção destina-se a apresentar alguns exemplos de mapeamentos de atributos de uma classe. Considerando a classe **Cliente** exibida na Figura 7, composta de um conjunto de atributos, onde um deles (**id**) corresponde à chave primária da tabela correspondente. Dessa forma, pode-se criar uma tabela **cliente** na base de dados a partir do script mostrado na e Listagem 18.

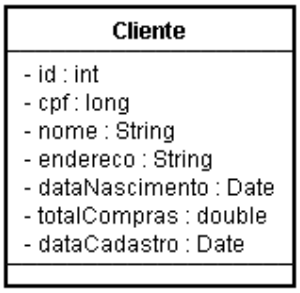

Figura 7 - Classe Cliente

```
CREATE TABLE anotacoes.cliente
(
  id_cliente integer NOT NULL,
  cpf bigint NOT NULL,
  nome character(40) NOT NULL,
  endereco character(100),
  total_compras numeric(16,2),
```

```
  data_cadastro timestamp without time zone NOT NULL,
  data_nascimento date
)
WITHOUT OIDS;
ALTER TABLE anotacoes.cliente OWNER TO postgres;
```

Listagem 18 – Script para a Criação da Tabela cliente

A classe **Cliente**, ilustrada pela Listagem 19, possui exemplos de mapeamentos de atributos a partir das anotações: **@Column**, **@Transient**, **@Temporal**. As anotações **@Id** e **@GeneratedValue** também são utilizadas e servem para mapear a chave primária da tabela correspondente e definir como o seu valor será gerado, respectivamente, como explicado anteriormente.

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

import javax.persistence.*;

@Entity
@Table(name="cliente", schema="anotacoes")
public class Cliente {

	@Transient
	private int temporaria;

	@Id
	@GeneratedValue(strategy = GenerationType.SEQUENCE)
	@Column(name="id_cliente")
	private int id;
	@Column(unique=true, nullable=false,
			insertable=true, updatable=true)
	private long cpf;
	@Column(nullable=false, length=40,
			insertable=true, updatable=true)
	private String nome;
	@Column(length=100)
	private String endereco;
	@Column(name="total_compras", precision=2)
	private double totalCompras;
```

```
        @Column(name="data_nascimento", nullable=false)
        @Temporal(TemporalType.DATE)
        private Date dataNascimento;
        @Column(name="data_cadastro")
        @Temporal(TemporalType.TIMESTAMP)
        private Date dataCadastro;

        public Cliente(){}

        //Métodos getters e setters
}
```

Listagem 19 – Mapeamento da Classe Cliente com Anotações

A seguir uma breve descrição da utilização destas anotações:

- Anotação **@Transient**: informa que o atributo mapeado não possui correspondente na tabela mapeada pela classe, ou seja, não é um atributo persistente;

- Atributos da anotação **@Column** utilizados:

    - **name**: nome da coluna na tabela do banco de dados que representa o atributo;
    - **unique**: indica se a coluna na tabela que representa o atributo possui a restrição de unicidade ou não. Por padrão, assume o valor **false**;
    - **nullable**: indica se a coluna na tabela que representa o atributo pode assumir valor nulo ou não. Por padrão, assume o valor **true**, indicando que pode assumir valores nulos;
    - **length**: informa o tamanho máximo assumido pelo valor da coluna na tabela;
    - **precision**: informa a precisão decimal dos possíveis valores para a coluna mapeada pelo atributo;
    - **insertable**: indica se o atributo será inserido no momento da inserção de uma linha na tabela. Por padrão assume o valor **true**;
    - **updatable**: indica se o atributo será atualizado no momento da atualização de uma linha na tabela. Por padrão assume o valor **true**.

- Anotação **@Temporal**: utilizada para mapeamento de datas e hora. Recebe um valor como argumento que pode ser:
    - **TemporalType.DATE**: usado para mapear datas;
    - **TemporalType.TIME**: usado para mapear hora;
    - **TemporalType.TIMESTAMP**: usado para mapear datas e hora.

13. Associações

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

Nas seções seguintes, serão apresentados exemplos de mapeamentos com anotações para cada um dos tipos de relacionamentos citados.

13.1 Associações 1-n (one-to-many)

Para exemplificar o relacionamento 1-n, considere o relacionamento entre a entidade **Centro** e a entidade **Universidade** da Figura 8. O relacionamento diz que uma universidade possui um conjunto de *n* centros e um centro está associado a apenas uma única universidade. Considere as classes de domínio Java de uma universidade e de um centro, respectivamente, mostradas na Listagem 20 e na Listagem 21. Ambas as classes já possuem o seu mapeamento via anotações inserido.

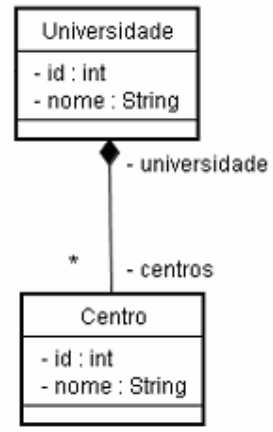

Figura 8 - Relacionamento entre Centro e Universidade

```java
package br.com.jeebrasil.hibernate.anotacoes.dominio;
import java.util.Collection;
import javax.persistence.*;

import org.hibernate.annotations.Cascade;
import org.hibernate.annotations.CascadeType;

@Entity @Table(schema="anotacoes")
public class Universidade {

	@Id
	@GeneratedValue(strategy = GenerationType.SEQUENCE)
	@Column(name="id_universidade")
	private int id;
	private String nome;

	@OneToMany(mappedBy="universidade", fetch = FetchType.LAZY)
	@Cascade(CascadeType.ALL)
	private Collection<Centro> centros;

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

Listagem 20 - Classe de Domínio: Universidade

```java
package br.com.jeebrasil.hibernate.anotacoes.dominio;
import javax.persistence.*;

import org.hibernate.annotations.Cascade;
import org.hibernate.annotations.CascadeType;
import org.hibernate.annotations.Fetch;
import org.hibernate.annotations.FetchMode;

@Entity @Table(schema="anotacoes")
public class Centro {

	@Id @GeneratedValue(strategy = GenerationType.SEQUENCE)
	@Column(name="id_centro")
	private int id;
	private String nome;
```

```
	@ManyToOne(fetch = FetchType.EAGER)
	@JoinColumn(name="id_universidade",
	                    insertable=true, updatable=true)
	@Fetch(FetchMode.JOIN)
	@Cascade(CascadeType.SAVE_UPDATE)
	private Universidade universidade;  //Métodos getters e setters
	//...
}
```

Listagem 21 - Classe de Domínio: Centro

As classes de domínio **Universidade** e **Centro** são mapeadas para as tabelas **universidade** e **centro**, que podem ser criadas na base de dados a partir dos scripts apresentados na Listagem 22 e na Listagem 23, respectivamente.

```
CREATE TABLE anotacoes.universidade
(
  id_universidade integer NOT NULL, -- Identificador da tabela
  nome character(100) NOT NULL, -- Nome da universidade
  CONSTRAINT pk_universidade PRIMARY KEY (id_universidade)
)
WITHOUT OIDS;
ALTER TABLE anotacoes.universidade OWNER TO postgres;
COMMENT ON COLUMN anotacoes.universidade.id_universidade IS
'Identificador da tabela';
COMMENT ON COLUMN anotacoes.universidade.nome IS 'Nome da
universidade';
```

Listagem 22 – Script para a Criação da Tabela universidade

```
CREATE TABLE anotacoes.centro
(
  id_centro integer NOT NULL, -- Identificador da tabela
  nome character(100) NOT NULL, -- Nome do centro
  id_universidade integer NOT NULL,
            -- Identificador da universidade a que o centro pertence
  CONSTRAINT pk_centro PRIMARY KEY (id_centro),
  CONSTRAINT fk_centro_universidade FOREIGN KEY (id_universidade)
      REFERENCES anotacoes.universidade (id_universidade) MATCH SIMPLE
```

```
      ON UPDATE NO ACTION ON DELETE NO ACTION
)
WITHOUT OIDS;
ALTER TABLE anotacoes.centro OWNER TO postgres;
COMMENT ON COLUMN anotacoes.centro.id_centro IS 'Identificador da
tabela';
COMMENT ON COLUMN anotacoes.centro.nome IS 'Nome do centro';
COMMENT ON COLUMN anotacoes.centro.id_universidade IS 'Identificador
da universidade a que o centro pertence';
```

Listagem 23 – Script para a Criação da Tabela centro

A classe de domínio **Universidade** é a que possui um mapeamento do tipo 1-n através do atributo coleção de **centros**. O seu mapeamento foi feito na Listagem 20 a partir da anotação **@OneToMany**. Como a coleção conterá objetos do tipo **Centro**, então está classe também deverá ser uma classe persistente da aplicação. Na anotação **@OneToMany**, existe um atributo denominado **mappedBy** que deverá receber como valor o nome do atributo na classe **Centro** (classe dos tipos de objetos da coleção) que se refere à classe **Universidade** (onde há o mapeamento 1-n). Em outras palavras, a tabela **centro** possui uma chave estrangeira para a tabela **universidade**, representada pelo atributo **Universidade universidade** da classe **Centro**, que corresponderá ao atributo **mappedBy** da anotação **@OneToMany**, ou seja, **mappedBy="universidade"**.

Já o atributo **fecth** indica quando o conteúdo do atributo será trazido da base de dados. Pode assumir dois valores:

- **FetchType.EAGER**: sempre que o objeto "pai" for trazido da base de dados, o atributo mapeado com **fetch=FetchType.EAGER** fará com que o seu conteúdo também seja trazido;
- **FetchType.LAZY**: sempre que o objeto "pai" for trazido da base de dados, o atributo mapeado com **fetch=FetchType.LAZY** fará com que o seu conteúdo somente seja trazido quando acessado pela primeira vez.

A anotação **@Cascade**, também utilizada no mapeamento da coleção **centros**, serve para indicar com que ação em cascata o relacionamento será tratado, ou seja, especifica quais operações deverão ser em cascata do objeto pai para o objeto associado. Por exemplo, pode assumir alguns dos valores abaixo:

- **CascadeType.PERSIST**: os objetos associados vão ser inseridos automaticamente quando o objeto "pai" for inserido;
- **CascadeType.SAVE_UPDATE**: os objetos associados vão ser inseridos ou atualizados automaticamente quando o objeto "pai" for inserido ou atualizado;
- **CascadeType.REMOVE**: os objetos associados ao objeto "pai" vão ser removidos, quando o mesmo for removido;
- **CascadeType.REPLICATE**: Se o objeto for replicado para outra base de dados, os filhos também serão;
- **CascadeType.LOCK**: Se o objeto for reassociado com a sessão persistente, os filhos também serão;
- **CascadeType.REFRESH**: Se o objeto for sincronizado com a base de dados, os filhos também serão;
- **CascadeType.MERGE**: Se um objeto tiver suas modificações mescladas em uma sessão, os filhos também terão;
- **CascadeType.EVICT**: Se o objeto for removido do cache de primeira nível, os filhos também serão;
- **CascadeType.ALL**: junção de todos os tipos de cascade.

Para ilustrar o efeito da anotação **@Cascade(CascadeType.ALL)**, considere o exemplo da Listagem 24, onde, inicialmente, um objeto **Universidade** é criado. Em seguida, dois objetos da classe **Centro** também são criados, recebem valores para seu atributo **nome** e são associados ao objeto **universidade**, que é posteriormente persistido.

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

      Centro centro1 = new Centro();
      centro1.setNome("Centro de Tecnologia");
      centro1.setUniversidade(univ);

      Centro centro2 = new Centro();
      centro2.setNome("Centro de Humanas");
      centro2.setUniversidade(univ);

      univ.setCentros(new HashSet<Centro>());
```

```
      univ.getCentros().add(centro1);
      univ.getCentros().add(centro2);

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

Listagem 24 – Exemplo de Persistência OneToMany. Efeito da anotação @Cascade(CascadeType.ALL)

A Listagem 25 apresenta o resultado da persistência do objeto **universidade** presente na Listagem 24. Observa-se, que a partir do atributo c**ascade** com valor **CascadeType.ALL**, inicialmente é inserida uma linha na tabela **universidade** e em seguida duas linhas na tabela **centro**. Vale salientar, que na hora de inserir as linhas na tabela **centro**, como os objetos que as representavam foram associados ao objeto **universidade**, automaticamente, o *Hibernate* atribui como valor de suas chaves estrangeiras, o valor gerado para a chave primária da linha criada na tabela **universidade**.

```
Hibernate: insert into anotacoes.Universidade
      (nome, id_universidade) values (?, ?)
Hibernate: insert into anotacoes.Centro
      (nome, id_universidade, id_centro) values (?, ?, ?)
Hibernate: insert into anotacoes.Centro
      (nome, id_universidade, id_centro) values (?, ?, ?)
```

Listagem 25 – Resultado da Execução do Código da Listagem 24

O atributo **centros** da classe **Universidade** foi mapeado com a anotação **@OneToMany** e com o atributo **fecth=FetchType.EAGER**. A Listagem 26 apresenta um exemplo de consulta a uma linha na tabela **universidade** com valor de chave primária igual a 100. Na Listagem 27 e na Listagem 28 estão os resultados da consulta considerando o atributo **fecth** da coleção definido como **FetchType.EAGER** e como **FetchType.LAZY**, respectivamente.

```
//...
      Session session = sf.openSession();


      //Consulta de uma linha na tabela universidade
      //com valor de chave primária = 100
      Universidade univ =
                  (Universidade)session.get(Universidade.class, 100);


      session.close();
//...
```

Listagem 26 – Consulta para Ilustrar o Uso do Atributo fecth

```
Hibernate: select universida0_.id_universidade as id1_3_1_,
      universida0_.nome as
      nome3_1_, centros1_.id_universidade as id3_3_,
      centros1_.id_centro as id1_3_, centros1_.id_centro as id1_1_0_,
      centros1_.nome as nome1_0_,
      centros1_.id_universidade as id3_1_0_
from anotacoes.Universidade universida0_
left outer join anotacoes.Centro centros1_ on
      universida0_.id_universidade=centros1_.id_universidade
where universida0_.id_universidade=?
```

Listagem 27 – Resultado da Execução do Código da Listagem 26 com fecth=FetchType.EAGER

```
Hibernate: select universida0_.id_universidade as id1_3_0_,
      universida0_.nome as nome3_0_
from anotacoes.Universidade universida0_
where universida0_.id_universidade=?
```

Listagem 28 – Resultado da Execução do Código da Listagem 26 com fecth=FetchType.LAZY

A partir destes resultados, é possível observar que realmente com o valor **FetchType.EAGER**, o SQL gerado pelo *Hibernate* na consulta de uma universidade realiza um **left outer join** com a tabela **centro** já trazendo os dados da coleção, podendo ser desnecessário. Por outro lado, utilizando **FetchType.LAZY**, a consulta retorna apenas os dados referentes à universidade, de forma que se fossem necessários os dados da coleção, bastaria acessar o atributo que a representa no objeto **universidade**.

A Listagem 29 apresenta um exemplo com a coleção **centros** mapeada com **FetchType.LAZY**. Neste exemplo, a mesma consulta à universidade de chave primária igual a 100 é feita e em seguida um acesso ao atributo **centros**, através da linha do código fonte **univ.getCentros().iterator()**. O resultado pode ser visto na Listagem 30, em que duas SQLs são geradas, uma gerada no momento da consulta à linha na tabela **universidade** de identificador igual a 100 e a outra no momento em que se itera a coleção **centros** do objeto anteriormente recuperado.

```
//...
      Session session = sf.openSession();

      //Consulta de uma linha na tabela universidade
      //com valor de chave primária = 100.
      //fetch=FetchType.LAZY. Não traz dados da coleção centros
      Universidade univ =
                  (Universidade)session.get(Universidade.class, 100);
      //Acesso à coleção centros, de forma que os dados serão buscados
      univ.getCentros().iterator();

      session.close();
//...
```

Listagem 29 – Consulta para Ilustrar o Uso do Atributo fecth=FetchType.LAZY

```
Hibernate: select universida0_.id_universidade as id1_3_0_,
      universida0_.nome as nome3_0_ from anotacoes.Universidade
      universida0_ where universida0_.id_universidade=?

Hibernate: select centros0_.id_universidade as id3_1_,
      centros0_.id_centro as id1_1_, centros0_.id_centro as id1_1_0_,
      centros0_.nome as nome1_0_, centros0_.id_universidade as
      id3_1_0_ from anotacoes.Centro centros0_
where centros0_.id_universidade=?
```

Listagem 30 – Resultado da Execução do Código da Listagem 29

13.2 Associações n-1 (many-to-one)

O relacionamento n-1 será apresentado a partir do relacionamento existente entre as classes **Centro** e **Universidade**, mostrado também na Figura 8. Neste caso, o relacionamento está presente no mapeamento da classe **Centro**, como mostrado na Listagem 21, através da anotação **@ManyToOne**. Para facilitar o entendimento, o trecho de mapeamento many-to-one do atributo **universidade** da classe **Centro** pode ser visto também na Listagem 31.

```
//...

@Entity @Table(schema="anotacoes")
public class Centro {
	//...

	@ManyToOne(fetch = FetchType.EAGER)
	@JoinColumn(name="id_universidade",
			insertable=true, updatable=true)
	@Fetch(FetchMode.JOIN)
	@Cascade(CascadeType.SAVE_UPDATE)
	private Universidade universidade;  //Métodos getters e setters
	//...
}
```

Listagem 31 - Mapeamento ManyToOne

A anotação **@ManyToOne** também possui o atributo **fetch**, que possui o mesmo comportamento apresentado anteriormente. A anotação **@JoinColumn** é utilizada para informar qual o nome da coluna que corresponde à chave estrangeira do mapeamento, no caso, **name="id_universidade"**. Nesta anotação também são aceitáveis os atributos **insertable** e **updatable** que se assumirem **true** indica que o atributo deve ser inserido (**insertable**) ou atualizado (**updatable**) no momento em que o objeto que possui o relacionamento é inserido ou atualizado, respectivamente. O atributo do relacionamento não será inserido se **insertable = false** e não será atualizado se **updatable = false**.

Outra anotação utilizada é a **@Fecth**, que define como o atributo mapeado será recuperado da base de dados. Pode assumir três valores:

- **FetchMode.JOIN**: utiliza **outer join** para carregar entidades ou coleções mapeadas;
- **FetchMode.SELECT**: utiliza um novo **select** para carregar entidades ou coleções mapeadas;
- **FetchMode.SUBSELECT**: utiliza uma consulta **subselect** adicional para carregar coleções adicionais. Não permitido para mapeamentos **ManyToOne**.

A Listagem 32 apresenta uma consulta à base de dados do centro com valor de chave primária igual a 110. A Listagem 33 apresenta o resultado da consulta com o mapeamento do atributo **universidade** da classe **Centro** utilizando a anotação **@Fecth** recebendo com valor **FetchMode.JOIN**. Pode-se observar que uma única consulta é feita, realizando um **JOIN** entre as tabelas **centro** e **universidade**. Já na Listagem 34, o resultado da consulta com o uso da anotação **@Fetch** com o valor **FetchMode.SELECT** resulta em duas consultas **SELECT** à base de dados, uma para buscar as informações do centro de chave primária igual a 110 e a outra para buscar as informações da universidade associada.

```
//...
      Session session = sf.openSession();

      //Consulta de uma linha na tabela centro
      //com valor de chave primária = 110
      Centro centro = (Centro)session.get(Centro.class, 110);
      session.close();
//...
```

Listagem 32 – Consulta para Ilustrar o Uso da Anotação @Fecth

```
Hibernate: select centro0_.id_centro as id1_1_1_, centro0_.nome
      as nome1_1_, centro0_.id_universidade as id3_1_1_,
      universida1_.id_universidade as id1_3_0_, universida1_.nome as
      nome3_0_
from anotacoes.Centro centro0_
left outer join anotacoes.Universidade
      universida1_ on
      centro0_.id_universidade=universida1_.id_universidade
where centro0_.id_centro=?
```

Listagem 33 – Resultado da Execução do Código da Listagem 32 com @Fetch(FetchMode.JOIN)

```
Hibernate: select centro0_.id_centro as id1_1_0_, centro0_.nome as
      nome1_0_, centro0_.id_universidade as id3_1_0_
from anotacoes.Centro centro0_ where centro0_.id_centro=?

Hibernate: select universida0_.id_universidade as id1_3_0_,
      universida0_.nome as nome3_0_
from anotacoes.Universidade universida0_
where universida0_.id_universidade=?
```

Listagem 34 – Resultado da Execução do Código da Listagem 32 com @Fetch(FetchMode.SELECT)

13.3 Associações n-n (many-to-many)

O relacionamento n-n será feito a partir do relacionamento entre as entidades **Departamento** e **Curso** mostrado na Figura 9.

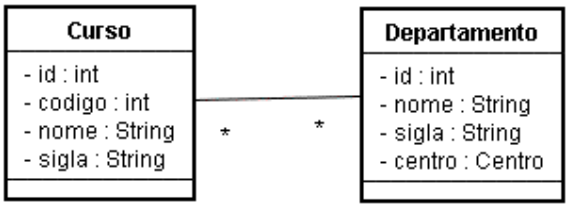

Figura 9 - Relacionamento n-n entre Curso e Departamento

Um relacionamento n-n implica em existir uma nova tabela para mapear o relacionamento no banco de dados. Vamos denominar essa nova tabela como **DEPARTAMENTO_CURSO**, como mostrado na Figura 10. Dessa forma, um departamento possui uma coleção de cursos e um curso uma coleção de departamentos. A existência dessas coleções nas classes de domínio é opcional. Por exemplo, pode ser que em um sistema real não seja necessário saber todos os departamentos de determinado curso, mas se for realmente necessário, o *Hibernate* apresenta outros mecanismos para a obtenção desta informação.

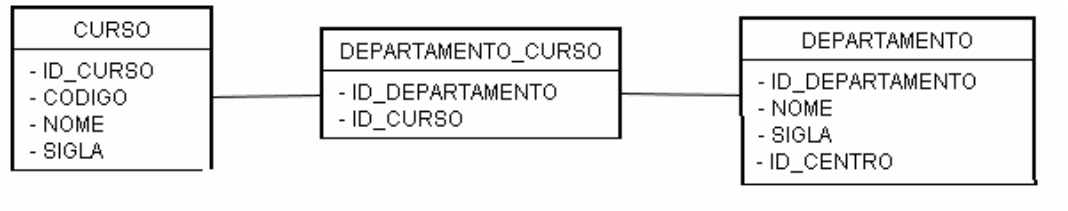

Figura 10 - Tabela de relacionamento DEPARTAMENTO_CURSO

Para criar as tabelas **curso**, **departamento** e **departamento_curso**, pode-se utilizar os scripts presentes na Listagem 35, na Listagem 36 e na Listagem 37, respectivamente.

```
CREATE TABLE anotacoes.curso
(
  id_curso integer NOT NULL,
  codigo integer NOT NULL,
  nome character varying NOT NULL,
  sigla character(6) NOT NULL,
  CONSTRAINT pk_curso PRIMARY KEY (id_curso),
  CONSTRAINT uk_codigo_curso UNIQUE (codigo)
)
WITHOUT OIDS;
ALTER TABLE anotacoes.curso OWNER TO postgres;
```

Listagem 35 - Script para a Criação da Tabela curso

```
CREATE TABLE anotacoes.departamento
(
  id_departamento integer NOT NULL,
  nome character varying NOT NULL,
  sigla character(6) NOT NULL,
  id_centro integer NOT NULL,
  CONSTRAINT pk_departamento PRIMARY KEY (id_departamento),
  CONSTRAINT fk_departamento_centro FOREIGN KEY (id_centro)
      REFERENCES anotacoes.centro (id_centro) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)
WITHOUT OIDS;
ALTER TABLE anotacoes.departamento OWNER TO postgres;
```

Listagem 36 - Script para a Criação da Tabela departamento

```
CREATE TABLE anotacoes.departamento_curso
(
  id_departamento integer NOT NULL,
  id_curso integer NOT NULL,
  CONSTRAINT fk_dc_curso FOREIGN KEY (id_curso)
      REFERENCES anotacoes.curso (id_curso) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT fk_dc_departamento FOREIGN KEY (id_departamento)
      REFERENCES anotacoes.departamento (id_departamento) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)
WITHOUT OIDS;
ALTER TABLE anotacoes.departamento_curso OWNER TO postgres;
```

Listagem 37 - Script para a Criação da Tabela departamento_curso

As classes Java das entidades **Departamento** e **Curso** estão ilustradas nas Listagem 38 e Listagem 39, respectivamente. Ambas as classes possuem mapeamentos de coleções com o relacionamento n-n, que é feito a partir da anotação **@ManyToMany**. Para este mapeamento, foi preciso também do uso da anotação **@JoinTable** para informar qual a tabela intermediária entre as entidades **Departamento** e **Curso**.

A anotação **@JoinTable** possui o atributo **name** que é onde é informado o nome da tabela intermediária que representa o mapeamento n-n, no caso, a tabela **departamento_curso**. Possui também o atributo **schema**, onde dever ser informado o esquema da base de dados em que a tabela se encontra. Já o atributo **joinColumns** recebe como valor uma anotação **@JoinColumn**, informando qual o nome da coluna na tabela intermediária que representa a classe onde se está fazendo o mapeamento n-n. Em relação ao mapeamento na tabela **Departamento**, a coluna na tabela intermediária **departamento_curso** que a representa se chama **id_departamento**. Por fim, no atributo **inverseJoinColumns** deve ser informada qual a coluna que representa a outra entidade no relacionamento n-n. No mapeamento da classe **Departamento**, a outra classe do relacionamento é a classe **Curso**, portanto, a coluna na tabela intermediária que a representa é **id_curso**.

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

```
import java.util.Collection;
import javax.persistence.*;
import org.hibernate.annotations.Cascade;
import org.hibernate.annotations.CascadeType;

@Entity
@Table(name="departamento", schema="anotacoes")
public class Departamento {

	@Id
	@GeneratedValue(strategy = GenerationType.SEQUENCE)
	@Column(name="id_departamento")
	private int id;

	@Column(nullable=false, insertable=true, updatable=true)
	private String nome;
	@Column(nullable=false, insertable=true,
			updatable=true, length=6)
	private String sigla;

	@ManyToOne(fetch=FetchType.EAGER)
	@JoinColumn(name="id_centro", insertable=true, updatable=true)
	@Cascade(CascadeType.SAVE_UPDATE)
	private Centro centro;

	@ManyToMany(fetch=FetchType.LAZY)
	@JoinTable(name="departamento_curso", schema="anotacoes",
			joinColumns=@JoinColumn(name="id_departamento"),
			inverseJoinColumns=@JoinColumn(name="id_curso"))
	private Collection<Curso> cursos;

	//Métodos setters e getters

}
```

Listagem 38 - Classe de Domínio: Departamento

Na classe **Curso** também há o mapeamento de um relacionamento n-n com a coleção **departamentos**. A explicação é semelhante ao mapeamento da coleção **cursos** na classe **Departamento**.

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

import java.util.Collection;
import javax.persistence.*;

@Entity
@Table(name="curso", schema="anotacoes")
public class Curso {

	@Id
	@GeneratedValue(strategy = GenerationType.SEQUENCE)
	@Column(name="id_curso")
	private int id;
	@Column(unique=true, nullable=false,
			insertable=true, updatable=true)
	private int codigo;
	@Column(nullable=false, insertable=true,
				updatable=true, length=6)
	private String nome;
	@Column(nullable=false, insertable=true, updatable=true)
	private String sigla;

	@ManyToMany(fetch=FetchType.LAZY)
	@JoinTable(name="departamento_curso", schema="anotacoes",
		joinColumns={@JoinColumn(name="id_curso")},
		inverseJoinColumns={@JoinColumn(name="id_departamento")})
	private Collection<Departamento> departamentos;

	//Métodos setters getters
}
```

Listagem 39 - Classe de Domínio: Curso

O código presente na Listagem 40 cria um instância de um objeto **Departamento**. Em seguida recupera um objeto persistente da classe **Centro** com identificador igual a **110** e outro da classe **Curso** com identificador igual a **220**. O objeto **Curso** é adicionado na coleção de **cursos** do objeto **Departamento** criado. Por fim, o departamento é persistido.

```
//...
tx.begin();
```

```
Departamento depart = new Departamento();
depart.setNome("DEPARTAMENTO 1");
depart.setSigla("DEP.01");

//Busca objeto centro na base de dados com
//identificador igual a 110
depart.setCentro((Centro)session.get(Centro.class, 110));

//Busca objeto curso na base de dados com
//identificador igual a 220
Curso curso = (Curso)session.get(Curso.class, 220);
depart.setCursos(new ArrayList<Curso>());
//Adiciona curso na coleção do departamento
depart.getCursos().add(curso);

//Persiste departamento
session.save(depart);

tx.commit();
//...
```

Listagem 40 – Exemplo com Mapeamento Usado a Anotação @ManyToMany: Persistência de Departamento

O SQL gerado é apresentado na Listagem 41. Observa-se que **SELECT**s na tabela **centro**, na tabela **curso** e na seqüência **hibernate_seq** são realizados. Depois, é feito e um INSERT na tabela **departamento** e em seguida, uma inclusão de linha na tabela de relacionamento **departamento_curso**, devido ao mapeamento n-n na classe **Departamento**.

```
Hibernate: select centro0_.id_centro as id1_1_1_, centro0_.nome as
      nome1_1_, centro0_.id_universidade as id3_1_1_,
      universida1_.id_universidade as id1_5_0_, universida1_.nome as
      nome5_0_
from anotacoes.Centro centro0_
left outer join anotacoes.Universidade universida1_
      on centro0_.id_universidade=universida1_.id_universidade
where centro0_.id_centro=?
```

```
Hibernate: select curso0_.id_curso as id1_3_0_, curso0_.codigo as
      codigo3_0_, curso0_.nome as nome3_0_, curso0_.sigla as sigla3_0_
from anotacoes.curso curso0_
where curso0_.id_curso=?

Hibernate: select nextval ('hibernate_sequence')

Hibernate: insert into anotacoes.departamento
      (nome, sigla, id_centro, id_departamento) values (?, ?, ?, ?)

Hibernate: insert into anotacoes.departamento_curso
      (id_departamento, id_curso) values (?, ?)
```

Listagem 41 – Resultado da Execução do Código Presente na Listagem 40

13.4 Associações n-n com Atributos

Imagine que seria necessário guardar a data em que foi feita a associação entre um determinado curso e um determinado departamento, ou seja, necessário ter um novo atributo na tabela **departamento_curso**. Dessa forma, esta tabela não seria formada apenas pelos identificadores das tabelas **curso** e **departamento**, mas sim também pela data. Para essa situação, os mapeamentos dos relacionamentos com a anotação **@ManyToMany** nas classes **Departamento** e **Curso** não resolveriam o problema. Então, deve-se criar uma nova classe **DepartamentoCurso**, como mostrado no diagrama da Figura 11, formada por uma chave primária composta por dois elementos e por um atributo que representa a data.

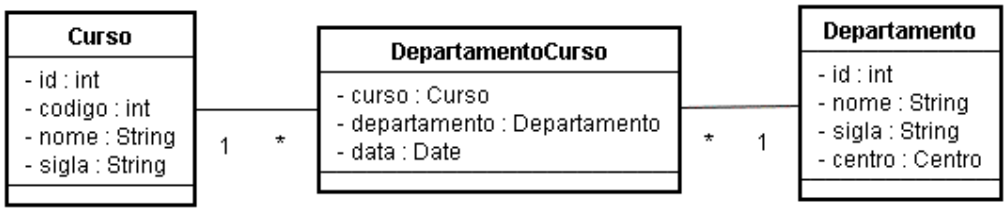

Figura 11 - Mapeamento n-n com Atributo

Para adicionar a coluna **data** à tabela **departamento_curso**, basta utilizar o script apresentado na Listagem 42.

```
ALTER TABLE anotacoes.departamento_curso ADD COLUMN
      data time without time zone;
ALTER TABLE anotacoes.departamento_curso ALTER COLUMN
      data SET STORAGE PLAIN;
```

Listagem 42 - Script para a Adição da Coluna data na Tabela departamento_curso

Com a adição da coluna data, deve ser feito o mapeamento da tabela **departamento_curso** como mostrado na Listagem 43 a partir da classe **DepartamentoCurso**. Pode-se observar que a chave composta é representada pelo atributo **chaveComposta** do tipo **DepartamentoCursoPK** e mapeada pela anotação **@EmbeddedId**.

A classe **DepartamentoCursoPK** representa a chave primária composta e é mapeada como mostrado na seção seguinte.

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

import java.util.Date;
import javax.persistence.*;

@Entity
@Table(name="departamento_curso", schema="anotacoes")
public class DepartamentoCurso {

      @EmbeddedId
      private DepartamentoCursoPK chaveComposta;

      @Column(name="data")
      @Temporal(TemporalType.DATE)
      private Date data;

      //Métodos getters e setters
}
```

Listagem 43 - Classe de Domínio: DepartamentoCurso

13.4.1 Composite-id

Para mapear uma chave composta, deve-se criar uma classe que a

represente. Esta classe deve ser mapeada com a anotação **@Embeddable**, como mostrado na Listagem 44 no mapeamento da chave composta **DepartamentoCursoPK**. Dentro desta classe deve-se mapear os atributos que compõem a chave composta das maneiras vistas anteriormente. É importante ressaltar que a classe que representa a chave composta deve implementar a interface **Serializable**.

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

import java.io.Serializable;
import javax.persistence.*;

@Embeddable
public class DepartamentoCursoPK implements Serializable{

	@ManyToOne(fetch = FetchType.EAGER)
	@JoinColumn(name="id_departamento")
	private Departamento departamento;

	@ManyToOne(fetch = FetchType.EAGER)
	@JoinColumn(name = "id_curso")
	private Curso curso;
	//Métodos getters e setters
}
```

Listagem 44 - Classe de Domínio: DepartamentoCursoPK

Para exemplificar o relacionamento n-n com atributos, observe o exemplo da Listagem 45. Primeiro um departamento e um curso são buscados na base de dados, ambos com identificadores iguais a 1. Em seguida, cria-se uma instância de um objeto da classe **DepartamentoCursoPK** que representa a chave composta. Os valores que compõem a chave, curso e departamento, são atribuídos. Finalmente, cria-se um objeto da classe **DepartamentoCurso** que representa a tabela de relacionamento entre as entidades, define-se a sua chave composta e a data de criação, persistindo-o na base de dados.

```
//...
tx.begin();

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

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

//Cria Chave Composta
DepartamentoCursoPK dcID = new DepartamentoCursoPK();
dcID.setDepartamento(d);
dcID.setCurso(c);

//Define relacionamento com atributo
DepartamentoCurso dc = new DepartamentoCurso();
dc.setChaveComposta(dcID);
dc.setData(new Date());

session.save(dc);

tx.commit();
//...
```

Listagem 45 – Exemplo com Mapeamento de Chave Composta

O resultado da execução do código presente na Listagem 45 pode ser visto na Listagem 46.

```
Hibernate: select departamen0_.id_departamento as id1_4_2_,
      departamen0_.nome as nome4_2_, departamen0_.sigla as sigla4_2_,
      departamen0_.id_centro as id4_4_2_, centro1_.id_centro as
      id1_1_0_, centro1_.nome as nome1_0_, centro1_.id_universidade as
      id3_1_0_, universida2_.id_universidade as id1_6_1_,
      universida2_.nome as nome6_1_
from anotacoes.departamento departamen0_
left outer join anotacoes.Centro centro1_ on
      departamen0_.id_centro=centro1_.id_centro left outer join
      anotacoes.Universidade universida2_ on
      centro1_.id_universidade=universida2_.id_universidade
where departamen0_.id_departamento=?

Hibernate: select curso0_.id_curso as id1_3_0_, curso0_.codigo as
      codigo3_0_, curso0_.nome as nome3_0_, curso0_.sigla as sigla3_0_
from anotacoes.curso curso0_ where curso0_.id_curso=?

Hibernate: insert into anotacoes.departamento_curso
      (data, id_departamento, id_curso) values (?, ?, ?)
```

Listagem 46 – Resultado da Execução do Código Presente na Listagem 45

13.5 Associações 1-1 (one-to-one)

Considere o relacionamento 1-1 entre as entidades **Universidade** e **Endereco** mostrado na Figura 12. Uma universidade tem um único endereço e um endereço pertence apenas a uma única universidade.

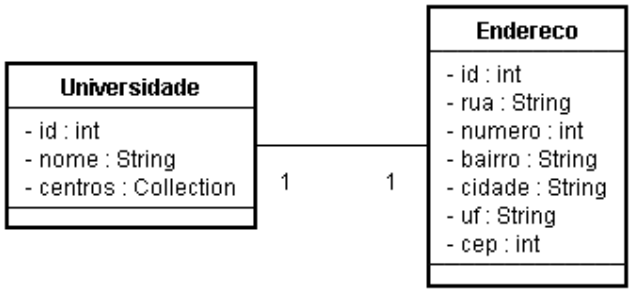

Figura 12 - Exemplo de Relacionamento 1-1

A Listagem 20 e a Listagem 47 apresentam as classes Java para as entidades **Universidade** e **Endereco**, respectivamente. Na classe **Endereco**, o relacionamento 1-1 com a classe **Universidade** é mapeado através da anotação **@OneToOne** e da anotação **@JoinColumn** usada para informar o nome da coluna que representa a chave estrangeira para a tabela **universidade**.

```
package br.com.jeebrasil.hibernate.anotacoes.dominio;
import javax.persistence.*;

@Entity
@Table(schema="anotacoes")
public class Endereco {

	@Id
	@GeneratedValue(strategy = GenerationType.SEQUENCE)
	@Column(name="id_endereco")
	private int id;

	private String rua;
	private int numero;
	private String bairro;
	private String cidade;
```

```
        private String uf;
        private int cep;

        @OneToOne
        @JoinColumn(name="id_universidade")
        private Universidade universidade;

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

Listagem 47 - Classe de Domínio: Endereço

A tabela **endereço** pode ser criada a partir do script exibido na Listagem 48.

```
CREATE TABLE anotacoes.endereco
(
  id_endereco integer NOT NULL,
  rua character varying(200) NOT NULL,
  numero integer NOT NULL,
  bairro character(30) NOT NULL,
  cidade character(30) NOT NULL,
  uf character(2) NOT NULL,
  cep integer NOT NULL,
  id_universidade integer NOT NULL,
  CONSTRAINT endereco_pkey PRIMARY KEY (id_endereco),
  CONSTRAINT fk_universidade_endereco FOREIGN KEY (id_universidade)
      REFERENCES anotacoes.universidade (id_universidade) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)
WITHOUT OIDS;
ALTER TABLE anotacoes.endereco OWNER TO postgres;
```

Listagem 48 - Script para a Criação da Tabela endereco

Para finalizar o mapeamento 1-1 entre as classes **Universidade** e **Endereço**, deve ser incluído na classe **Universidade** o atributo **endereco** também mapeado com a anotação **@OneToOne**, como mostrado na Listagem 49. No mapeamento 1-1 na classe **Universidade**, como em sua tabela não há chave estrangeira para a tabela **endereco**, deve-se informar qual atributo na classe **Universidade** refere-se ao mapeamento 1-1 do outro lado, no caso o atributo

universidade, informado pelo atributo **mappedBy** da anotação **@OneToOne**. Também é utilizada a anotação **@Cascade** passando como argumento **CascadeType.ALL**, informando que no momento que se inserir, atualizar ou remover uma universidade, a operação deve ser cascateada para a coluna da tabela.

```
//...
      @OneToOne(mappedBy="universidade")
      @Cascade(CascadeType.ALL)
        private Endereco endereco;
//...
```

Listagem 49 – Inclusão de Atributo endereco na classe Universidade

Para ilustrar o uso deste mapeamento, considere o exemplo mostrado na Listagem 50, onde, incialmente, objetos da classe **Endereco** e **Universidade** são criados. Em seguida, associa-se a universidade ao endereço e o endereço a universidade. Por fim, o objeto **universidade** é persistido na base de dados.

A Listagem 51 apresenta o resultado da execução do código presente na Listagem 50. Observa-se a universidade é persistida na base de dados e em seguida, o endereço também fazendo a associação com a universidade criada.

```
//...
      tx.begin();

      Endereco endereco = new Endereco();
      endereco.setRua("Av. Senador Salgado Filho");
      endereco.setNumero(1034);
      endereco.setBairro("Lagoa Nova");
      endereco.setCidade("Natal");
      endereco.setUf("RN");
      endereco.setCep(59067710);

      Universidade universidade = new Universidade();
      universidade.setNome("Universidade Teste");

      endereco.setUniversidade(universidade);
      universidade.setEndereco(endereco);

      session.save(universidade);

        tx.commit();
//...
```

Listagem 50 – Exemplo para Ilustrar Mapeamento 1-1

```
Hibernate: select nextval ('hibernate_sequence')
Hibernate: select nextval ('hibernate_sequence')
Hibernate: insert into anotacoes.Universidade
      (nome, id_universidade) values (?, ?)
Hibernate: insert into anotacoes.Endereco
      (rua, numero, bairro, cidade, uf, cep,
      id_universidade, id_endereco) values (?, ?, ?, ?, ?, ?, ?, ?)
```

Listagem 51 – Resultado da Execução do Código Presente na Listagem 50

14. Definição de Chave Primária Utilizando uma Seqüência Própria

Nos mapeamentos apresentados até o momento, a definição dos valores das chaves primárias foi feita apenas utilizando a seqüência **hibernate_sequence**. O exemplo presente na Listagem 52 mostra o mapeamento da chave primária da tabela **cliente** utilizando a seqüência de nome **cliente_seq**. Para isso, utilizou-se

as anotações **@SequenceGenerator** e **@GeneratedValue**.

Na anotação **@SequenceGenerator** define-se qual seqüência criada na base de dados será utilizada para o mapeamento. No caso, nomeia-se a seqüência com o nome **SEQ_CLIENTE** (**name = "SEQ_CLIENTE"**).

Para definir qual seqüência será utilizada no mapeamento da chave primária, utiliza-se a anotação **@GeneratedValue**, onde é informado que a definição do valor do atributo será através de seqüências (**strategy = GenerationType.*SEQUENCE***) e o nome da mesma (**generator = "SEQ_CLIENTE"**).

```
package br.com.jeebrasil.hibernate.anotacoes.dominio;
import javax.persistence.*;

@Entity
@Table(name="cliente", schema="anotacoes")
public class Cliente {

	@Id
	@SequenceGenerator(name = "SEQ_CLIENTE",
			sequenceName = "anotacoes.cliente_seq")
	@GeneratedValue(  strategy = GenerationType.SEQUENCE,
			  generator = "SEQ_CLIENTE")
	@Column(name="id_cliente")
	private int id;

	//...
}
```

Listagem 52 – Mapeamento de Chave Primária Usando Seqüência Própria

15. Coleções

O mapeamento de coleções já foi apresentado no contexto dos relacionamentos através da anotação **@OneToMany**. Essa seção está destinada a apresentar o mapeamento dos quatro tipos de coleções existentes (*Set*, *Bag*, *List* e *Map*), porém formadas por tipos primitivos.

15.1 Set

Para exemplificar o mapeamento de um **Set** de tipos primitivos considere o mapeamento de uma coleção de **String's** para armazenar os telefones de um **Aluno**, como visto na Figura 13. Então, na classe **Aluno** deve-se inserir o atributo **telefones** com seu mapeamento anotado, como mostrado na Listagem 53.

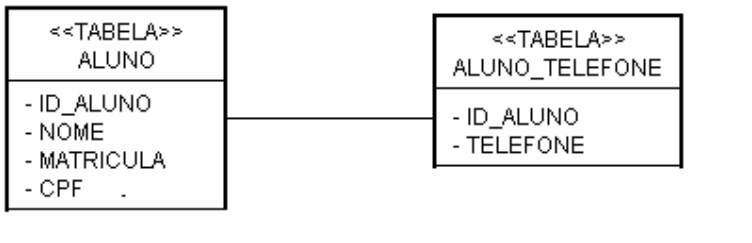

Figura 13 - Coleção de telefones para Aluno: Set

Para mapear esta coleção foi utilizada a anotação **@CollectionOfElements**. A anotação **@JoinTable** é utlizada para informar o nome da tabela e esquema onde a coleção dos elementos mapeados serão armazendos, no caso através dos atributos **name="aluno_telefone"** e **schema="anotacoes"**. Na anotação **@JoinTable** também é informada o nome da coluna que representa a classe onde a coleção está sendo mapeada, no caso a classe **Aluno**. Para informar esta coluna, utiliza-se o atributo **joinColumns** que recebe como valor a anotação **@JoinColumn** com o nome da coluna, no caso **id_aluno**. Por fim, informa-se a coluna que representará a informação do telefone utilizando a anotação **@Column.**

Os telefones do aluno são armazenados na tabela **aluno_telefone**. Para o banco de dados está tabela é separada da tabela **aluno**, mas para o *Hibernate*, ele cria a ilusão de uma única entidade.

```
//...
import org.hibernate.annotations.CollectionOfElements;
//...
	@CollectionOfElements
	@JoinTable(name = "aluno_telefone", schema = "anotacoes",
		joinColumns = @JoinColumn(name = "id_aluno"))
	@Column(name = "telefone")
	private Set<String> telefones;
//...
```

Listagem 53 – Atributo telefones Inserido na Classe Aluno (Set)

Um **Set** não pode apresentar elementos duplicados, portanto a chave primária de **aluno_telefone** consiste nas colunas **id_aluno** e **telefone**.

O script para a criação da tabela **aluno_telefone** está presente na Listagem 54.

```
CREATE TABLE anotacoes.aluno_telefone
(
  id_aluno integer NOT NULL,
  telefone character varying(12) NOT NULL
)
WITHOUT OIDS;
ALTER TABLE anotacoes.aluno_telefone OWNER TO postgres;
```

Listagem 54 - Script para a Criação da Tabela aluno_endereco

Dado o mapeamento da coleção de telefones de **Aluno**, observe o exemplo mostrado na Listagem 55. Neste exemplo, cria-se um objeto **Aluno** que terá valores para seus atributos definidos, inclusive dois números de telefone inseridos em sua coleção de **String's**. Por fim, o objeto **Aluno** é persistido. O resultado é mostrado na Listagem 56, onde primeiro é inserida uma linha na tabela **aluno** e posteriormente duas linhas na tabela **aluno_telefone**.

```
//...
      tx.begin();

      Aluno aluno = new Aluno();
      aluno.setMatricula(12846822);
      aluno.setNome("João Maria Costa Neto");

      aluno.setTelefones(new HashSet<String>());
      aluno.getTelefones().add("32222529");
      aluno.getTelefones().add("32152899");

      session.save(aluno);

      tx.commit();
//...
```

Listagem 55 - Exemplo de Mapeamento de Set

```
Hibernate: select nextval ('hibernate_sequence')
Hibernate: insert into anotacoes.aluno
      (matricula, nome, cpf, id_aluno) values (?, ?, ?, ?)
Hibernate: insert into anotacoes.aluno_telefone
      (id_aluno, telefone) values (?, ?)
Hibernate: insert into anotacoes.aluno_telefone
      (id_aluno, telefone) values (?, ?)
```

Listagem 56 - Resultado da Execução do Código Presente na Listagem 55

15.2 List

Um **List** é uma coleção ordenada que pode conter elementos duplicados. O mapeamento de uma lista requer a inclusão de uma coluna de índice na tabela do banco de dados. A coluna índice define a posição do elemento na coleção, como visto na Figura 14, a coluna índice é representada pela coluna **posicao** da tabela **aluno_telefone**. Dessa maneira, o *Hibernate* pode preservar a ordenação da coleção quando recuperada do banco de dados e for mapeada como um List.

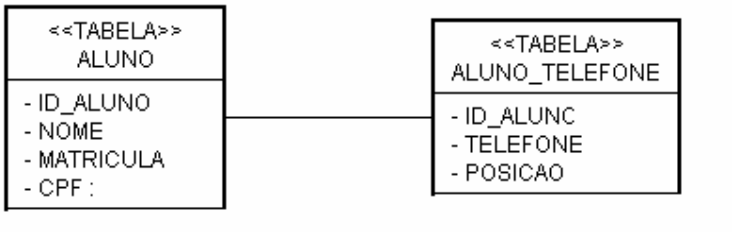

Figura 14 - Coleção de telefones para Aluno: List

Para mapear a coleção de telefones da classe **Aluno**, define o atributo **telefones** como mostrado na Listagem 57. A difereça é a inclusão da definição do índice através da anotação **@IndexColumn(name = "posição")**, informando a coluna que ele representa na tabela.

```
//...
import org.hibernate.annotations.CollectionOfElements;
import org.hibernate.annotations.IndexColumn;
//...
      @CollectionOfElements
      @JoinTable(name = "aluno_telefone", schema = "anotacoes",
```

```
            joinColumns = @JoinColumn(name = "id_aluno"))
      @Column(name = "telefone")
      @IndexColumn(name = "posicao")
      private List<String> telefones;
//...
```

Listagem 57 – Atributo telefones Inserido na Classe Aluno (List)

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

Para exemplificar, considera-se o exemplo presente na Listagem 58, com resultado de execução presente na Listagem 59.

```
//...
      tx.begin();

      Aluno aluno = new Aluno();
      aluno.setMatricula(12846822);
      aluno.setNome("João Maria Costa Neto");

      aluno.setTelefones(new ArrayList<String>());
      aluno.getTelefones().add("32222529");

      session.save(aluno);

      tx.commit();
//...
```

Listagem 58 - Exemplo de Mapeamento de List

```
Hibernate: select nextval ('hibernate_sequence')
Hibernate: insert into anotacoes.aluno
      (matricula, nome, cpf, id_aluno) values (?, ?, ?, ?)
Hibernate: insert into anotacoes.aluno_telefone
      (id_aluno, posicao, telefone) values (?, ?, ?)
```

Listagem 59 - Resultado da Execução do Código Presente na Listagem 58

15.3 Map

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

O mapeamento de um **Map** é semelhante ao de um **List**, onde o índice de posição passa a ser a chave. Veja a Figura 14. A Listagem 60 apresenta a coleção de telefones sendo mapeada como um **Map**. A chave primária da tabela **aluno_telefone** passa a ser as colunas **id_aluno** e **chave**, também permitindo a presença de telefones duplicados na coleção. A diferença dos outros mapeamentos é que a chave é mapeada através da anotação **@MapKey(columns = {@Column(name = "chave")}).**

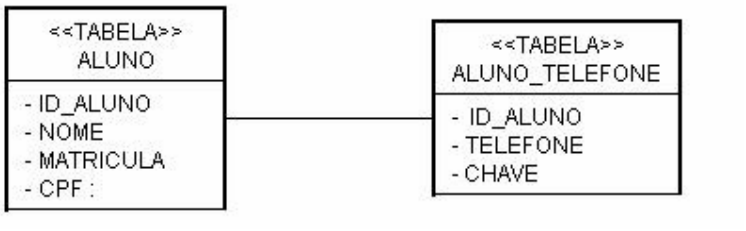

Figura 15 - Coleção de telefones para Aluno: Map

```
//...
import org.hibernate.annotations.CollectionOfElements;
import org.hibernate.annotations.MapKey;
//...
	@CollectionOfElements
	@JoinTable(name = "aluno_telefone", schema = "anotacoes",
		joinColumns = @JoinColumn(name = "id_aluno"))
	@Column(name = "telefone")
	@MapKey(columns = {@Column(name = "chave")})
	private Map<String, String> telefones;
//...
```

Listagem 60 – Atributo telefones Inserido na Classe Aluno (Map)

Para exemplificar, considera-se o exemplo presente na Listagem 61, com resultado de execução presente na Listagem 62.

```
//...
	tx.begin();
```

```
      Aluno aluno = new Aluno();
      aluno.setMatricula(12846822);
      aluno.setNome("João Maria Costa Neto");

      aluno.setTelefonesMap(new HashMap<String, String>());
      aluno.getTelefonesMap().put("1", "2234346");
      aluno.getTelefonesMap().put("2", "2342342");

      session.save(aluno);

      tx.commit();
//...
```

Listagem 61 - Exemplo de Mapeamento de Map

```
Hibernate: select nextval ('hibernate_sequence')
Hibernate: insert into anotacoes.aluno
      (matricula, nome, cpf, id_aluno) values (?, ?, ?, ?)
Hibernate: insert into anotacoes.aluno_telefone
      (id_aluno, chave, telefone) values (?, ?, ?)
Hibernate: insert into anotacoes.aluno_telefone
      (id_aluno, chave, telefone) values (?, ?, ?)
```

Listagem 62 - Resultado da Execução do Código Presente na Listagem 61

15.4 Bag

Um **Bag** consiste em uma coleção desordenada que permite elementos duplicados. Em Java não há implementação de um **Bag**, contudo o *Hibernate* fornece um mecanismo de que um **List** em Java simule o comportamento de um **Bag**. Pela definição de um **List**, uma lista é uma coleção ordenada, contudo o *Hibernate* não preserva a ordenação quando um **List** é persistido com a semântica de um **Bag**. Para usar o **Bag**, a coleção de telefones deve ser definida com o tipo **List**. Um exemplo de mapeamento de um **Bag** pode ser visto na Figura 16 e na Listagem 63.

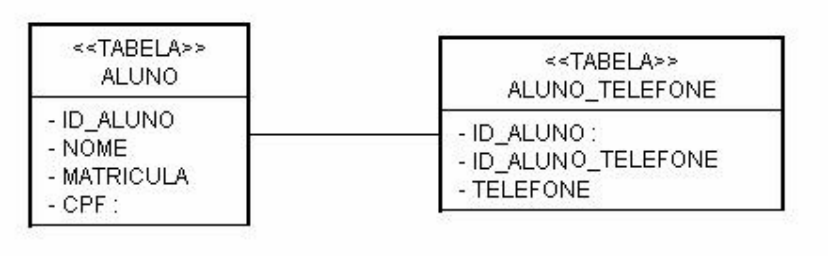

Figura 16 - Coleção de telefones para Aluno: Bag

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

Para o exemplo, a chave substituta é representada pela coluna **id_aluno_telefone** da table **aluno_telefone**. É necessário informar como os seus valores serão gerados, neste caso, é utlizada uma seqüência. A anotação **@SequenceGenerator** serve para definir o nome de uma seqüência no banco de dados. O atributo **name** informa como será denominada a seqüência, já o atributo **sequenceName** informa qual seqüência se está utilizando.

A anotação **@CollectionId** é utlizada para definir a chave substituta. No atributo **columns** informa-se qual o nome da coluna na tabela; o atributo **generator** informa qual seqüência está sendo utilizada para gerar seus valores (no caso **SEQ_HIBERNATE** definida na anotação **@SequenceGenerator**); e o atributo **type** é utilizado para informar o tipo desta coluna, no caso **integer**.

```
//...
import org.hibernate.annotations.CollectionId;
import org.hibernate.annotations.CollectionOfElements;
import org.hibernate.annotations.Type;
//...
      @CollectionOfElements
      @JoinTable( name = "aluno_telefone", schema = "anotacoes",
                  joinColumns = @JoinColumn(name = "id_aluno"))
      @Column(name = "telefone")
      @SequenceGenerator( name = "SEQ_HIBERNATE",
                          sequenceName = "hibernate_sequence")
      @CollectionId(columns = @Column (name = "id_aluno_telefone"),
                    generator = "SEQ_HIBERNATE",
```

```
                    type = @Type(type = "integer"))
      private List<String> telefones;
//...
```

Listagem 63 – Atributo telefones Inserido na Classe Aluno (Bag)

Para exemplificar, considera-se o exemplo presente na Listagem 58, onde a coleção agora está mapeada com semântica de **Bag**, com resultado de execução presente na Listagem 64. Observa-se que a seqüência **hibernate_sequence** é invocada duas vezes: uma para o valor da chave primária do objeto **Aluno** e outra para o valor da chave substitua da coleção de telefones.

```
Hibernate: select nextval ('hibernate_sequence')
Hibernate: insert into anotacoes.aluno
      (matricula, nome, cpf, id_aluno) values (?, ?, ?, ?)
Hibernate: select nextval ('hibernate_sequence')
Hibernate: insert into anotacoes.aluno_telefone
      (id_aluno, id_aluno_telefone, telefone) values (?, ?, ?)
```

Listagem 64 - Resultado da Execução do Código Presente na Listagem 58

16. Herança

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

- Tabela por classe concreta: cada classe concreta é mapeada para uma tabela diferente no banco de dados. 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.

Observe o exemplo de herança apresentado na Figura 17. Neste caso, as classes **Aluno** e **Professor** herdam da classe **Pessoa**, ou seja, são seus filhos.

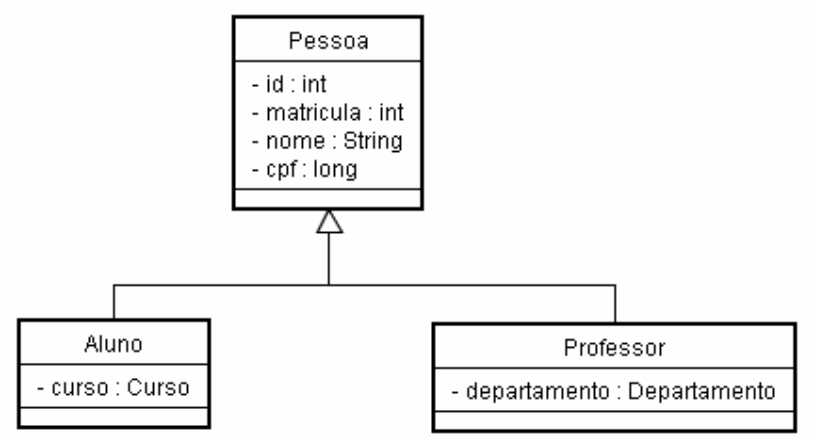

Figura 17 – Herança

A Figura 18, a Figura 19 e a Figura 20 mostram as tabelas que devem ser criadas para as estratégias de mapeamento tabela por classe concreta, tabela por hierarquia e tabela por sub-classe, respectivamente.

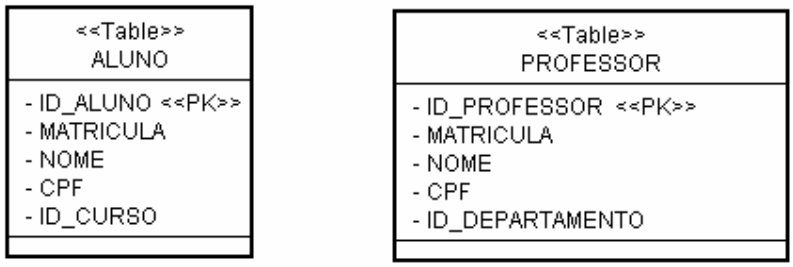

Figura 18 - Tabela por Classe Concreta

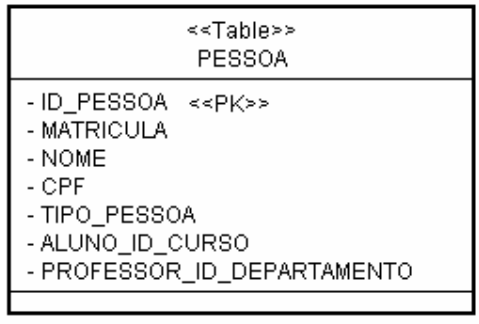

Figura 19 -Tabela por Hierarquia

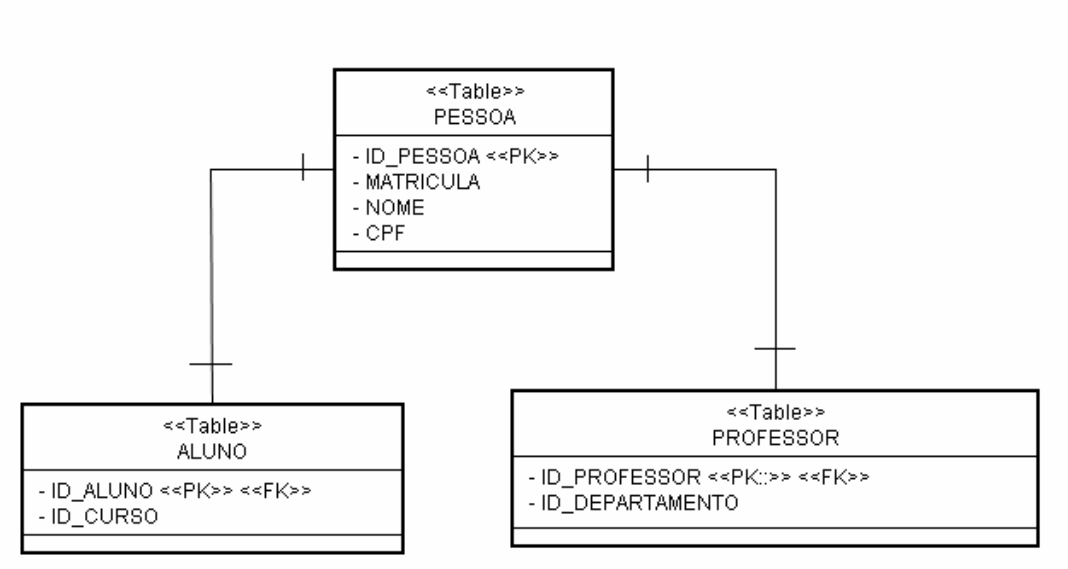

Figura 20 - Tabela por Sub-Classe

16.1 Tabela Por Classe Concreta

Em relação à 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 em seções anteriores. O mapeamento da classe **Pessoa** pode ser visto na Listagem 65.

```
package br.com.jeebrasil.hibernate.anotacoes.heranca.dominio;

import javax.persistence.Column;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.MappedSuperclass;

@MappedSuperclass
public class Pessoa {

	@Id
	@GeneratedValue(strategy = GenerationType.SEQUENCE)
	@Column(name="id_pessoa")
	private int id;
	private int matricula;
```

```
	private String nome;
	private long cpf;

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

Listagem 65 – Mapeamento da Classe Pessoa: 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***). A Listagem 66 e a Listagem 67 apresentam os mapeamentos das classes **Aluno** e **Professor**, respectivamente, que herdam da classe **Pessoa**. Verifica-se que apenas os atributos específicos de cada classe precisam ser mapeados.

```
package br.com.jeebrasil.hibernate.anotacoes.heranca.dominio;

import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

import br.com.jeebrasil.hibernate.anotacoes.dominio.Curso;

@Entity
@Table(name="aluno", schema="anotacoes")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Aluno extends Pessoa{

	@ManyToOne(fetch = FetchType.EAGER)
	@JoinColumn(name = "id_curso")
	private Curso curso;

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

Listagem 66 – Mapeamento da Classe Aluno: Tabela Por Classe Concreta

```java
package br.com.jeebrasil.hibernate.anotacoes.heranca.dominio;

import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

import br.com.jeebrasil.hibernate.anotacoes.dominio.Departamento;



//Anotação que informa que a classe mapeada é persistente
@Entity
@Table(name="professor", schema="anotacoes")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Professor extends Pessoa{

	@ManyToOne(fetch = FetchType.EAGER)
	@JoinColumn(name = "id_departamento")
	private Departamento departamento;

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

Listagem 67 – Mapeamento da Classe Professor: 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 sub-classes 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.

Considere o exemplo mostrado na Listagem 68, em que um objeto do tipo **Aluno** é criado, tem valores atribuídos para os atributos herdados e específico e, por fim, é persistido. O resultado é exibido na Listagem 69, onde uma linha na tabela **aluno** é inserida.

```
//...
      tx.begin();

      Aluno aluno = new Aluno();

      //Definição de atributos herdados
      aluno.setCpf(228765287);
      aluno.setNome("Claudia Dantas Pedroza");
      aluno.setMatricula(12345);

      //Definição de atributo específico
      Curso curso = (Curso) session.get(Curso.class, 220);
      aluno.setCurso(curso);

      //Persistência do objeto aluno
      session.save(aluno);

      tx.commit();
//...
```

Listagem 68 – Exemplo de Persistência: Tabela Por Classe Concreta

```
Hibernate: select curso0_.id_curso as id1_3_0_,
      curso0_.codigo as codigo3_0_, curso0_.nome as nome3_0_,
      curso0_.sigla as sigla3_0_ from anotacoes.curso curso0_
where curso0_.id_curso=?
Hibernate: select nextval ('hibernate_sequence')
Hibernate: insert into anotacoes.aluno
      (matricula, nome, cpf, id_curso, id_pessoa)
      values (?, ?, ?, ?, ?)
```

Listagem 69 – Resultado da Execução do Código Presente na Listagem 68

16.2 Tabela Por Hierarquia

Em relação à tabela por hierarquia, o mapeamento deve ser feito como mostrado nas listagens abaixo, onde todos os atributos da classe mãe e das classes filhas são armazenadas 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***, infomando que as três classes serão mapeadas em uma única tabela.

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.

```
package br.com.jeebrasil.hibernate.anotacoes.heranca.dominio;

import javax.persistence.*;

@Entity
@Table(name = "pessoa", schema = "anotacoes")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
      name = "tipo_pessoa",
      discriminatorType = DiscriminatorType.INTEGER
      )
@DiscriminatorValue("0")
public class Pessoa {
```

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

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

Listagem 70 – Mapeamento da Classe Pessoa: Tabela Por Hierarquia

Os mapeamentos das subclasses **Aluno** e **Professor** sao feitos como mostrado na Listagem 71 e na Listagem 72, respectivamente. 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**.

```
package br.com.jeebrasil.hibernate.anotacoes.heranca.dominio;

import javax.persistence.*;

import br.com.jeebrasil.hibernate.anotacoes.dominio.Curso;

@Entity
@DiscriminatorValue("1")
public class Aluno extends Pessoa{

	@ManyToOne(fetch = FetchType.EAGER)
	@JoinColumn(name = "aluno_id_curso")
	private Curso curso;

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

Listagem 71 – Mapeamento da Classe Aluno: Tabela Por Hierarquia

```
package br.com.jeebrasil.hibernate.anotacoes.heranca.dominio;
```

```java
import javax.persistence.*;

import br.com.jeebrasil.hibernate.anotacoes.dominio.Departamento;

@Entity
@DiscriminatorValue("2")
public class Professor extends Pessoa{

	@ManyToOne(fetch = FetchType.EAGER)
	@JoinColumn(name = "professor_id_departamento")
	private Departamento departamento;

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

Listagem 72 – Mapeamento da Classe Professor: Tabela Por Hierarquia

Considere o exemplo mostrado na Listagem 73, em que um objeto do tipo **Professor** é criado, tem valores atribuídos para os atributos herdados e específico e, por fim, é persistido. O resultado é exibido na Listagem 74, onde uma linha na tabela **professor** é inserida.

```java
//...
	tx.begin();

	Professor prof = new Professor();

	//Definição de atributos herdados
	prof.setCpf(2342523);
	prof.setNome("Pedro Maia Ferreira");
	prof.setMatricula(23523);

	//Definição de atributo específico
	Departamento departamento = new Departamento();
	departamento.setId(210);
	prof.setDepartamento(departamento);

	//Persistência do objeto professor
	session.save(prof);
```

```
//...
```

Listagem 73 – Exemplo de Persistência: Tabela Por Hierarquia

```
Hibernate: select nextval ('hibernate_sequence')
Hibernate: insert into anotacoes.pessoa
      (matricula, nome, cpf, professor_id_departamento,
      tipo_pessoa, id_pessoa) values (?, ?, ?, ?, 2, ?)
```

Listagem 74 – Resultado da Execução do Código Presente na Listagem 73

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

16.3 Tabela Por SubClasse

A terceira estratégia, como já citada, 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. A Listagem 75 apresenta o mapeamento da superclasse **Pessoa**, onde a estratégia de mapeamento da herança é definida como **JOINED**, através da anotação **@Inheritance(strategy=InheritanceType.*JOINED*)**.

```
package br.com.jeebrasil.hibernate.anotacoes.heranca.dominio;

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;
	private int matricula;
	private String nome;
	private long cpf;
	//Métodos getters e setters
	//...
}
```

Listagem 75 – Mapeamento da Classe Pessoa: Tabela Por SubClasse

Os mapeamentos das subclasses **Aluno** e **Professor** sao feitos como mostrado na Listagem 76 e Listagem 77, respectivamente. 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.

```
package br.com.jeebrasil.hibernate.anotacoes.heranca.dominio;

import javax.persistence.*;
import br.com.jeebrasil.hibernate.anotacoes.dominio.Curso;

@Entity
@Table(name="aluno", schema="anotacoes")
@PrimaryKeyJoinColumn(name = "id_aluno")
public class Aluno extends Pessoa{

	@ManyToOne(fetch = FetchType.EAGER)
	@JoinColumn(name = "id_curso")
	private Curso curso;
	//Métodos getters e setters
	//...
}
```

Listagem 76 – Mapeamento da Classe Aluno: Tabela Por SubClasse

```
package br.com.jeebrasil.hibernate.anotacoes.heranca.dominio;

import javax.persistence.*;
import br.com.jeebrasil.hibernate.anotacoes.dominio.Departamento;

@Entity
@Table(name="professor", schema="anotacoes")
@PrimaryKeyJoinColumn(name = "id_professor")
public class Professor extends Pessoa{

	@ManyToOne(fetch = FetchType.EAGER)
	@JoinColumn(name = "id_departamento")
	private Departamento departamento;
	//Métodos getters e setters
	//...
}
```

Listagem 77 – Mapeamento da Classe Professor: 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.

A Listagem 78 apresenta um exemplo de persistência de um objeto da classe **Professor**. Observa-se no resultado da execução na Listagem 79, que é inserida uma linha na tabela **pessoa** com os valores dos atributos herdados da classe **Pessoa** e uma outra linha na tabela **professor** com apenas os valores específicos da classe.

```
//...
	tx.begin();

	Professor prof = new Professor();

	//Definição de atributos herdados
	prof.setCpf(2342523);
```

```
      prof.setNome("Pedro Maia Ferreira");
      prof.setMatricula(23523);

      //Definição de atributo específico
      Departamento departamento = new Departamento();
      departamento.setId(210);
      prof.setDepartamento(departamento);

      //Persistência do objeto professor
      session.save(prof);
//...
```

Listagem 78 – Exemplo de Persistência: Tabela Por SubClasse

```
Hibernate: select nextval ('hibernate_sequence')
Hibernate: insert into anotacoes.pessoa
      (matricula, nome, cpf, id_pessoa) values (?, ?, ?, ?)
Hibernate: insert into anotacoes.professor
      (id_departamento, id_professor) values (?, ?)
```

Listagem 79 – Resultado da Execução do Código Presente na Listagem 78

17. Transações

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

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

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

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

1º) Ler o saldo da conta corrente A (xA)
2º) Calcular o débito de R$ 150,00 da conta corrente A (dA = xA – 150,00)

3º) Gravar na base de dados o novo saldo da conta corrente A (xA = dA)

4º) Ler o saldo da conta corrente B (xB)

5º) Calcular o crédito de R$ 150,00 na conta corrente B (dB = xB + 150,00)

6º) Gravar na base de dados o novo saldo da conta corrente B (xB = dB)

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

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

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

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

- **Atomicidade:** o banco de dados deve garantir que todas as transações sejam indivisíveis.

- **Consistência:** após a execução de uma transação, o banco de dados deve continuar consistente, ou seja, deve continuar com um estado válido.

- **Isolamento:** mesmo que várias transações ocorram paralelamente (ou concorrentemente), nenhuma transação deve influenciar nas outras. Resultados parciais de uma transação não devem ser "vistos" por outras transações executadas concorrentemente.

- **Durabilidade:** após a finalização de uma transação, todas as alterações feitas por ela no banco de dados devem ser duráveis, mesmo havendo falhas no sistema após a sua finalização.

17.1 Modelos de Transações

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

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

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

- ***Chained Transaction***. Também conhecido como Modelo de Transações Encadeadas, esse modelo tem como objetivo desfazer as operações de uma transação em caso de erro com a menor perda de trabalho possível. Uma

transação encadeada consiste em um conjunto de sub-transações executadas seqüencialmente, em que à medida que as sub-transações vão sendo executadas, são validadas e não podem mais ser desfeitas. Os resultados do conjunto de transações só serão visíveis ao final da execução de todas elas.

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

17.2 Transações e Banco de Dados

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

Os limites das operações de uma transação devem ser demarcados. Assim, é possível saber a partir de qual operação a transação é iniciada e em qual operação ela finalizada. Ao final da execução da última operação que pertence à transação, todas as alterações no banco de dados realizadas pelas operações que compõe a transação devem ser confirmadas, ou seja, um *commit* é realizado. Se houver algum erro durante a execução de algumas das suas operações, todas as operações da transação que já foram executadas devem ser desfeitas, ou seja, um *rollback* é realizado. A Figura 21 ilustra esses conceitos.

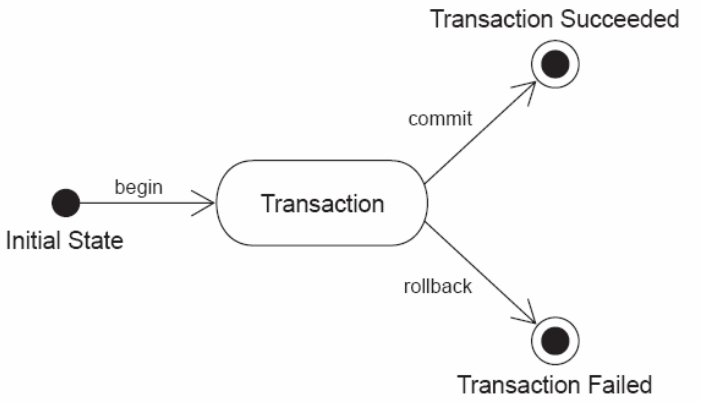

Figura 21 - Estados do sistema durante uma transação

17.3 Ambientes Gerenciados e Não Gerenciados

As seções seguintes referem-se às definições dos conceitos relacionados a

transações JDBC e JTA, onde aparecem os termos ambientes gerenciados e não gerenciados. Esta seção destina-se a explicar sucintamente o que são esses termos.

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

17.4 Transações JDBC

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

Uma transação JDBC é controlada pelo gerenciador de transações SGBD e geralmente é utilizada por ambientes não gerenciados. Utilizando um *driver* JDBC, o início de uma transação é feito implicitamente pelo mesmo. Embora alguns bancos de dados necessitem invocar uma sentença "**begin transaction**" explicitamente, com a API JDBC não é preciso fazer isso. Uma transação é finalizada após a chamada do método `commit()`. Caso algo aconteça de errado, para desfazer o que foi feito dentro de uma transação, basta chamar o método `rollback()`. Ambos, `commit()` e `rollback()`, são invocados a partir da conexão JDBC.

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

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

17.5 Transações JTA

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

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

17.6 API para Transações do Hibernate

A interface `Transaction` fornece métodos para a declaração dos limites de uma transação. A Listagem 80 apresenta um exemplo de uso de transações com a interface `Transaction`.

A transação é iniciada a partir da invocação ao método `session.beginTransaction()`. No caso de um ambiente não gerenciado, uma transação JDBC na conexão JDBC é iniciada. Já no caso de um ambiente gerenciado, uma nova transação JTA é criada, caso não exista nenhuma já criada. Casa já existe uma transação JTA, essa nova transação une-se a existente.

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

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

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

```
Session session = sessions.openSession();
Transaction tx = null;
```

```
try {

      tx = session.beginTransaction();
      acaoExecutada();
      tx.commit();

} catch (Exception e) {
      if (tx != null) {
            try {
                  tx.rollback();
            } catch (HibernateException he) {
                  //log he and rethrow e
            }
      }
} finally {

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

Listagem 80 - Usando a Interface Transaction do Hibernate

17.7 Flushing

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

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

O *Hibernate* possui um modo *flush* que pode ser definido a partir do

comando `session.setFlushMode()`. Este modo pode assumir os seguintes valores:

- **`FlushMode.AUTO`:** valor padrão. Faz com que o *Hibernate* não realize o processo de *flushing* antes de todas as consultas, somente realizará se as mudanças dentro da transação alterar seu resultado.
- **`FlushMode.COMMIT`:** especifica que os estados dos objetos em memória somente serão sincronizados com a base de dados ao final da transação, ou seja, quando o método `commit`() é chamado.

- **`FlushMode.NEVER`:** especifica que a sincronização só será realizado diante da chamada explícita ao método `flush()`.

17.8 Níveis de Isolamento de uma Transação

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

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

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

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

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

- ***Nonrepeatable Read*** (Leitura que não pode ser repetida): uma transação lê mais de uma vez um mesmo dado e constata que há valores distintos em cada leitura. Por exemplo, uma transação **A** lê uma linha do banco; uma

transação **B** modifica essa mesma linha e é finalizada (*commit*) antes que a transação **A**; a transação **A** lê novamente esta linha e obtém dados diferentes.

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

Existem quatro níveis de isolamento da transação em SQL. Eles se diferenciam de acordo com a ocorrência ou não dos fenômenos anteriormente descritos, como mostrado na Tabela 4.

Tabela 4 - Níveis de Isolamento da Transação em SQL

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

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

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

efetivará suas mudanças considerando a linha da tabela anteriormente lida. Caso contrário, a segunda transação irá ignorar a atualização caso a linha tenha sido excluída ou aplicará a sua atualização na versão atualizada da linha.

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

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

17.9 Configurando o nível de isolamento

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

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

Para configurá-lo basta incluir a linha presente na Listagem 81 no arquivo de configuração *.cfg.xml. Neste exemplo, o nível de isolamento foi definido como *Repeatable Read*.

```
<property name=" hibernate.connection.isolation">4</property>
```

Listagem 81 – Configuração do Nível de Isolamento

18. Concorrência

Em algumas situações pode acontecer que duas ou mais transações que ocorrem paralelamente leiam e atualizem o mesmo dado. Considerando que duas transações leiam um mesmo dado **x** quase que simultaneamente. Ambas as transações vão manipular esse mesmo dado com operações diferentes e atualizá-lo

na base de dados. Para exemplificar, a Listagem 82 apresenta um exemplo de duas transações concorrentes manipulando o mesmo dado **x**.

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

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

Listagem 82 – Exemplo de Transações Concorrentes

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

- **Pessimista**: utilizar o controle pessimista significa que se uma transação T1 lê um dado e tem a intenção de atualizá-lo, esse dado será bloqueado (nenhuma outra transação poderá lê-lo) até que T1 o libere, normalmente após a sua atualização.

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

18.1 Lock Otimista

Para ilustrar o gerenciamento do tipo otimista, um exemplo é dado a partir das Figura 22 e Figura 23.

O problema é mostrado na Figura 22, onde, inicialmente, duas transações (ilustradas por *Thread 1* e *Thread 2*) acessam um mesmo dado na base de dados (`SELECT Dado`), uma seguida da outra. Logo em seguida, a primeira transação (*Thread 1*) atualiza este dado na base de dados e depois quem também o atualiza é a segunda transação (*Thread 2*). Nesta abordagem otimista, acontece que a atualização do dado feita pela segunda transação sobrescreve a atualização realizada pela primeira, ou seja, a atualização feita pela primeira transação é perdida.

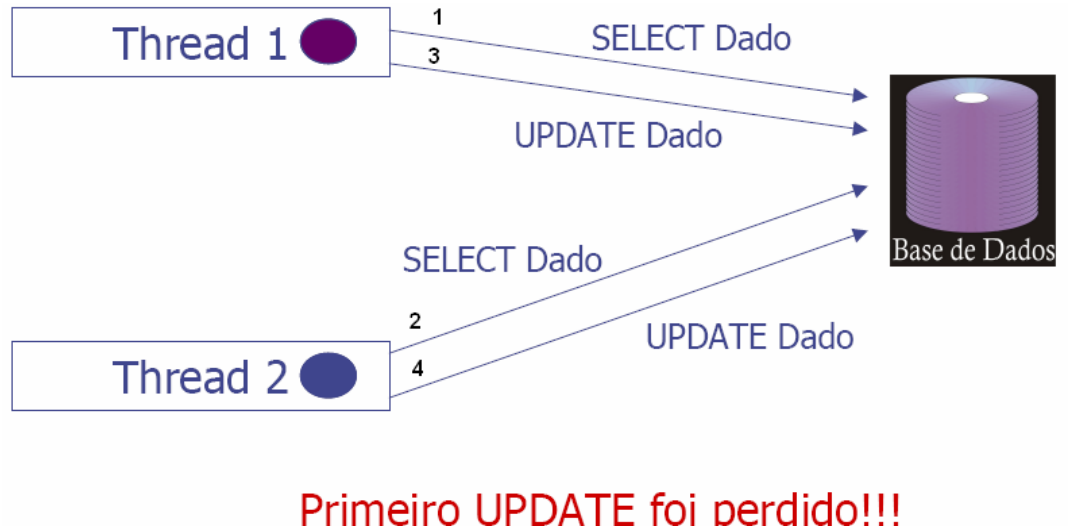

Figura 22 - Loocking Otimista: Atualização Sobrescrita

Para resolver o problema descrito anteriormente com a abordagem otimista, pode-se utilizar o conceito de *Version Number*, que é um padrão utilizado para versionar numericamente os dados de uma linha de uma tabela na base de dados. Por exemplo, na Figura 23, também, inicialmente, duas transações (ilustradas por *App 1* e *App 2*) acessam um mesmo dado na base de dados (`SELECT Dado`), uma seguida da outra. Com isso, esse mesmo dado nas duas transações são rotulados com a versão atual dele na base de dados, no caso, *Versão 1*. Logo em seguida, a segunda transação atualiza este dado. Quando a atualização vai ser feita, é verificado se a versão do dado na transação corresponde à versão dele na base de dados. Nesta primeira atualização, a versão da transação é 1 e a da base de dados também. Como elas são iguais, a atualização é efetivada e a versão do dado na base de dados passa a ser a *Versão 2*. Por fim, a primeira transação vai também atualizar este mesmo dado. Dessa forma, também é feita uma comparação entre as versões do dado na transação e na base de dados. Neste caso, a versão na transação é a 1 e na base de dados é 2, ou seja, as versões não correspondem.

Assim, um erro é disparado e a atualização desejada pela primeira transação não é concretizada, evitando que a atualização feita pela segunda transação não seja desfeita.

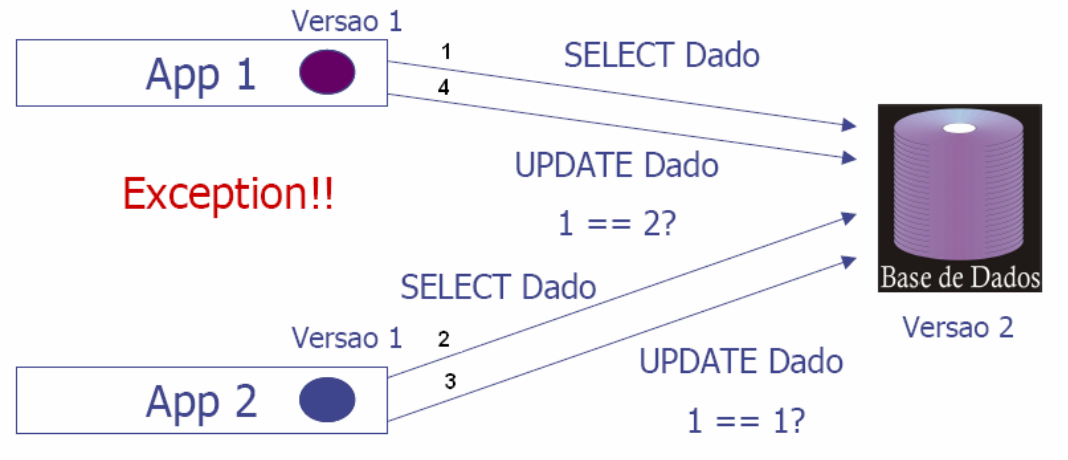

Figura 23 - Loocking Otimista: Abodargem com *Version Number*

Com o Hibernate, uma forma de utilizar o versionamento dos dados é utilizar a anotação **@Version** no mapeamento das tabelas. Para exemplificar o seu uso, considera-se a classe **ContaCorrente** presente na Listagem 83 que será mapeada para a tabela **conta_corrente** na base de dados. Dentre os diversos atributos da classe tá o atributo denominado **versao** do tipo inteiro que irá guardar a versão atual das linhas da tabela.

```
package br.com.jeebrasil.hibernate.anotacoes.dominio;
import javax.persistence.*;

@Entity
@Table(name = "conta_corrente", schema = "anotacoes")
public class ContaCorrente {

	//Atributo utilizado para o versionamento
	@Version
	@Column(name = "versao")
	private int versao;

	//Demais atributos
	@Id
```

```
	@Column(name = "id_conta_corrente")
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int id;
	private double saldo;
	//Outros atributos
	//...

	public int getVersao(){
		return versao;
	}
	public void setVersao(int versao){
		this.versao = versao;
	}
	//Outros métodos getters e setters
	//...
}
```

Listagem 83 - Classe de Domínio: ContaCorrente. Primeira Estratégia Locking Otimista

A tabela **conta_corrente** também deve ter uma coluna para onde esse atributo **versao** será mapeado, como mostrado no script de criação da tabela na Listagem 84. De acordo com o mapeamento feito na classe **ContaCorrente**, toda vez que uma determinada linha for atualizada na base de dados, a sua coluna **versao** será incrementada de uma unidade, indicando a nova versão dos dados.

```
CREATE TABLE anotacoes.conta_corrente
(
  id_conta_corrente integer NOT NULL,
  saldo numeric(16,2) NOT NULL,
  versao integer NOT NULL,
  CONSTRAINT pk_conta_corrente PRIMARY KEY (id_conta_corrente)
)
WITHOUT OIDS;
ALTER TABLE anotacoes.conta_corrente OWNER TO postgres;
```

Listagem 84 - Script para a Criação da Tabela conta_corrente

Considerando o exemplo apresentado na Listagem 85, em que um objeto **ContaCorrente** é criado e persistido na base de dados, observa-se o resultado presente na Figura 24 da inserção do mesmo como linha da tabela correspondente. No caso, a linha foi inserida e a versão atual da mesma assumiu o valor zero, correspondendo à primeira versão.

```
//...
Transaction tx = session.beginTransaction();

ContaCorrente conta = new ContaCorrente();
conta.setSaldo(1000);

//Persistência do objeto conta
session.save(conta);

tx.commit();
//...
```

Listagem 85 – Persistindo Objeto com Atributo *Version Number*

	id_conta_corrente [PK] integer	saldo numeric(16,2)	versao integer
1	65	1000.00	0
*			

Figura 24 - Resultado da Execução do Código Presente na Listagem 85

Caso a próxima operação seja uma atualização do objeto que corresponde a esta linha, como mostrado na Listagem 86, verifica-se o resultado na Figura 25, onde os dados da conta corrente foram atualizados. Também teve-se a coluna **versao** sendo incrementada de uma unidade, atualizando a informação de versionamento da linha da tabela.

```
//...
Transaction tx = session.beginTransaction();

ContaCorrente conta = (ContaCorrente)
            session.get(ContaCorrente.class, 65);
//Modificando valor do saldo
conta.setSaldo(12250);
```

```
//Atualizacao  do objeto conta corrente
session.saveOrUpdate(conta);

tx.commit();
//...
```

Listagem 86 – Atualizando Objeto com Atributo *Version Number*

	id_conta_corrente [PK] integer	saldo numeric(16,2)	versao integer
1	65	12250.00	1
*			

Figura 25 - Resultado da Execução do Código Presente na Listagem 86

Outra forma de implementar o *lock* otimista é utilizando o atributo que representa a versão do dado como sendo do tipo *timestamp*. Neste caso, a classe **ContaCorrente** do exemplo anterior ao invés de ter um atributo inteiro para guardar a versão do dado, teria uma atributo do tipo **java.util.Date** para guardar o instante no tempo da última atualização. Neste caso, a classe de domínio com o mapeamento seria equivalente à mostrada na Listagem 87. Neste exemplo, a tabela **conta_corrente** deve conter uma coluna do tipo *timestamp* denominada **data_ultima_atualizacao**.

```
package br.com.jeebrasil.hibernate.anotacoes.dominio;
import javax.persistence.*;
import java.util.Date;

@Entity
@Table(name = "conta_corrente", schema = "anotacoes")
public class ContaCorrente {

	//Atributo utilizado para o versionamento
	@Version
	@Column(name = "data_ultima_atualizacao")
	@Temporal(TemporalType.TIMESTAMP)
	private Date data;

	//Demais atributos
```

```
	@Id
	@Column(name = "id_conta_corrente")
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int id;
	private double saldo;
	//Outros atributos
	//...

	public Date getData(){
		return data;
	}
	public void setData(Date Data){
		this.data = data;
	}
	//Outros métodos getters e setters
	//...
}
```

Listagem 87 - Classe de Domínio: ContaCorrente. Segunda Estratégia Locking Otimista

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

Com essa última abordagem, quando uma determinada linha vai ser atualizada, o *Hibernate* verifica se os dados dessa linha correspondem aos mesmos dados que foi recuperado. Caso afirmativo, a atualização é efetuada. Para isso, no mapeamento da tabela, deve-se utilizar o atributo **optimisticLock = OptimisticLockType.*ALL*** da anotação **@org.hibernate.annotations.Entity** (Além da anotação **javax.persistence.Entity**), como mostrado na Listagem 91.

```
package br.com.jeebrasil.hibernate.anotacoes.dominio;
import javax.persistence.*;

@Entity
@Table(name = "conta_corrente", schema = "anotacoes")
@org.hibernate.annotations.Entity(
	optimisticLock = OptimisticLockType.ALL , dynamicUpdate = true)
```

```
public class ContaCorrente {

	//Atributos da classe
	@Id
	@Column(name = "id_conta_corrente")
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int id;
	private double saldo;
	//Outros atributos
	//...

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

Listagem 88 - Classe de Domínio: ContaCorrente. Terceira Estratégia Locking Otimista

Dessa maneira, quando uma linha dessa tabela fosse atualizada, o SQL equivalente gerado para a atualização seria como o exibido na Listagem 89. Neste exemplo, considera-se a atualização do saldo para R$ 1.500,00 de uma determinada conta de saldo R$ 1.000,00.

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

Listagem 89 - Exemplo com Lockin Otimista: optimisticLock = OptimisticLockType.ALL

18.2 Lock Pessimista

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

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

Considerando o exemplo da Listagem 90, onde um determinado aluno é consultado na base de dados e tem seu nome atualizado. Neste caso, não há um bloqueio ao dado, então qualquer outra transação pode acessar este mesmo dado concorrentemente e modificá-lo, de forma que poderá ocorrer uma incosistência dos dados. Na Listagem 91, há um exemplo de uso do *lock* pessimista para resolver este problema, bastando passar a constante **LockMode.UPGRADE** como terceiro argumento do método **get** do objeto **Session**.

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

Listagem 90 - Exemplo de Transação sem Lock

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

Listagem 91 - Exemplo de Transação com Lock Pessimista: LockMode.UPGRADE

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

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

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

- **Lock.WRITE**: Obtida automaticamente quando o *Hibernate* realiza alguma inserção ou atualização na base de dados.

19. Caching

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

Existem três tipos principais de *chache*:

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

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

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

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

Pode ser também que o mecanismo de persistência opte por implementar identidade no escopo do processo, de forma que a identidade do objeto seja equivalente à identidade do banco de dados. Assim, se a consulta a dados em transações que executam concorrentemente for feita a partir de identificadores

de banco de dados iguais, o resultado também será o mesmo objeto Java. Outra forma de se proceder é retornar os dados em forma de novos objetos. Assim, cada transação teria seu próprio objeto Java representando o mesmo dado no banco.

No escopo de *cluster*, é necessário haver comunicação remota, em que os dados são sempre manipulados por cópias. Em geral, utiliza-se o *JavaGroups*, que consiste em uma plataforma utilizada como infra-estrutura para a sincronização do *cache* no no *cluster*.

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

19.1 Arquitetura de Cache com Hibernate

A Figura 26 apresenta a arquiteura de cache com o Hibernate. Existem dois níveis de *caches*. O primeiro nível encontra-se em nível de uma sessão (**Session**), ou em nível de transação de banco de dados ou até mesmo de uma transação de aplicação. Este nível não pode ser desabilitado e garante a identifidade do objeto dentro da sessão/transação. Dessa forma, dentro de uma mesma sessão, é garantida que se a aplicação requisitar o mesmo objeto persistente duas vezes, ela receberá de volta à mesma instância, evitando um tráfego maior na base de dados.

Com o uso do *cache* em nível de transação, nunca poderão existir diferentes representações do mesmo registro de banco de dados ao término de uma transação, tendo apenas um objeto representando qualquer registro. Mudanças realizadas em um objeto, dentro de uma transação, são sempre visíveis para qualquer outro código executado dentro da mesma transação.

Quando são feitas chamadas através dos métodos: **load()**, **find()**, **list()**, **iterate()** ou **filter()** o conteúdo do objeto é primeiramente procurado no nível de cache.

O segundo nível de *cache* é opcional e pode abranger o nível do escopo do processo ou do cluster. É um *cache* de estado e não de instâncias persistentes. Por ser opcional, é preciso configurar este nível de cache para ser usado na aplicação. Cada classe que precisar ter objetos em cache possui uma configuração individual, pois cada uma terá particularidade de mapeamento.

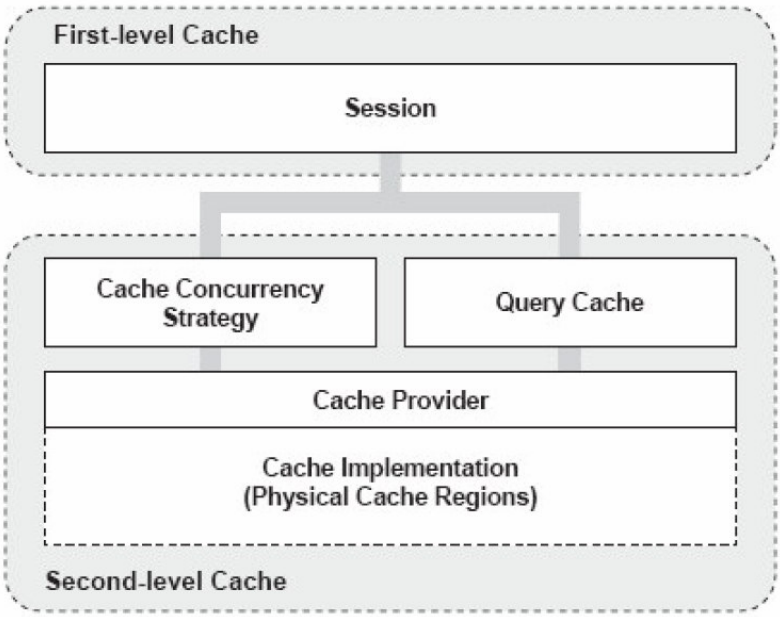

Figura 26 - Arquitetura de Cache

O *Cache Provider* é utilizado para informar qual política de cache de segundo nível será utilizada na aplicação. Essa política é definida no arquivo **hibernate.cfg.xml** a partir da propriedade **hibernate.cache.provider_class** e pode assumir algum dos valores abaixo:

- EHCache (`org.hibernate.cache.EhCacheProvider`): usado para o gerenciamento de caches no escopo do processo. Possui uma boa documentação e é fácil de configurar. Além disso, suporta cache de consultas que será apresentado na próxima sessão;
- OSCache (`org.hibernate.cache.OSCacheProvider`): usado para o gerenciamento de caches no escopo do processo ou do cluster;
- SwarmCache (`org.hibernate.cache.SwarmCacheProvider`): usado para o gerenciamento de caches no escopo do cluster. Não suporta cache de consultas.

- JBossCache (`org.hibernate.cache.TreeCacheProvider`): usado para o gerenciamento de caches no escopo do cluster. É um serviço de replicação por cluster transacional. Suporta cache de consultas.

19.2 Mapeando Cache

Quando se utiliza o segundo nível de *cache*, é importante utilizar uma estratégia para gerenciar o nível de concorrência dos dados. A estratégia de concorrência é uma mediadora, responsável por armazenar e recuperar os dados no *cache*. Com o Hibernate, é possível utilizar uma de quatro estratégias:

- Transacional: utilizado em ambiente JTA (gerenciável) e para manter dados em cache são mais consultados do que atualizados. Garante o nível de isolamento da transação até o nível *repeatable-read*, em que o valor é mantido durante toda a transação.

- Leitura/Escrita (*Read-Write*): utilizada se a aplicação precisar atualizar os dados que estão em cache. Não deve ser utilizada quando o nível de isolamento da transação for do tipo *Serializable*. Utilizado em ambientes que não utilizam *cluters*.

- *Nonstrict Read-Write*: utilizada se os dados em cache dificilmente sejam atualizados, ou seja, se for quase que improvável duas transações atualizarem o mesmo dado simultaneamente. É importante também a utilização de um nível rígido de isolamento da transação. Não garante que o estado do cache seja sincronizado com o do banco de dados;

- *Read-Only*: utilizada se o cache for aplicado a dados da aplicação que somente são lidos, ou seja, nunca são modificados. Estratégia simples, segura em nível de escopo de cluster e apresenta melhor performance.

A Tabela 5 apresenta quais níveis de cache são suportados por quais políticas de cache.

Cache Provider	read-only	nonstrict-read-write	read-write	transactional
EHCache	X	X	X	
OSCache	X	X	X	
SwarmCache	X	X		
JBossCache	X			X

Tabela 5 – Políticas X Níveis de Cache

19.3 Utilizando o Cache

Toda vez que um objeto é consultado no banco através do Hibernate e dos métodos **get** e **load**, o resultado da consulta é armazenado no primeiro nível de chache, ou seja, no cache do objeto **Session**.

Considerando o exemplo presente no diagrama da Figura 27, onde objetos da classe **Categoria** são atualizados na base de dados raramente. Então, pode-se utilizar *cache* na classe **Categoria**.

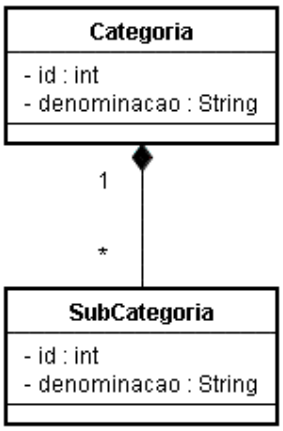

Figura 27 - Exemplo de Uso de Cache

Primeiramente, para exemplificar, considere o mapeamento da classe **Categoria** apresentado na Listagem 96. Neste caso, a classe foi mapeada considerando apenas o cache de primeiro nível (habilitado por padrão), ou seja, mapeada utilizando as anotações já apresentadas nesta apostila.

```
@Entity
@Table(schema = "anotacoes")
public class Categoria {

	@Id
	@SequenceGenerator(allocationSize = 1, name="SEQ")
	@GeneratedValue(  strategy = GenerationType.AUTO,
	                  generator = "SEQ")
	private int id;
	private String denominacao;

	@OneToMany(mappedBy = "categoria", fetch = FetchType.LAZY)
	private Collection<SubCategoria> subCategorias;

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

Listagem 92 - Mapeamento de Cache

Considera-se o código fonte apresentado na Listagem 93. Neste código, um objeto **Categoria** de valor de chave primária igual a **1** é recuperado três vezes do banco de dados. Em cada recuperação, o objeto é atribuído a três objetos diferentes **Categoria cat1**, **Categoria cat2** e **Categoria cat3**, onde **cat1** e **cat2** são recuperados dentro da mesma sessão, **session1**, e **cat3** é recuperado dentro de uma outra sessão, **session2**, aberta após o fechamento de **session1**.

```
//...

Session session1 = sf.openSession();
System.out.println("ABRIU SESSION 1\n");

System.out.println("Consulta cat1 (id = 1) -- Objeto session1 ");
Categoria cat1 = (Categoria) session1.get(Categoria.class, 1);
```

```
System.out.println("Consulta cat1 (id = 1) -- Objeto session1 ");
Categoria cat2 = (Categoria) session1.get(Categoria.class, 1);

session1.close();
System.out.println("\nFECHOU SESSION 1\n");

System.out.println("ABRIU SESSION 2\n ");
Session session2 = sf.openSession();

System.out.println("Consulta cat3 (id = 1) -- Objeto session2 ");
Categoria cat3 = (Categoria) session2.get(Categoria.class, 1);

session2.close();
System.out.println("\nFECHOU SESSION 2");

//...
```

Listagem 93 – Exemplificando o Uso de Cache

A Listagem 94 apresenta o resultado após a execução do código presente na Listagem 93. Pode-se observar, que dentro da primeira sessão (**session1**), a linha correspondete à tabela **categoria** com valor de chave primária igual a **1** é recuperada duas vezes (representado pelos objetos **cat1** e **cat2**), porém o Hibernate só faz a consulta no banco uma única vez (representada por apenas um SQL gerado durante a existência de **session1**. Em seguida, a primeira sessão é fechada (**session1.close()**) e uma segunda sessão é aberta (**sessao2**). Dessa forma, quando é necessário consultar mais uma vez a linha na tabela **categoria** que possui chave primária igual a **1**, como a consulta está dentro de outra sessão, um novo comando SQL é gerado, já que até o momento não há cache para o objeto **Session session2**.

```
ABRIU SESSION 1

Consulta cat1 (id = 1) -- Objeto session1
Hibernate: select categoria0_.id as id0_0_, categoria0_.denominacao as
      denomina2_0_0_ from anotacoes.Categoria categoria0_
      where categoria0_.id=?
Consulta cat1 (id = 1) -- Objeto session1
```

```
FECHOU SESSION 1

ABRIU SESSION 2

Consulta cat3 (id = 1) -- Objeto session2
Hibernate: select categoria0_.id as id0_0_, categoria0_.denominacao as
      denomina2_0_0_ from anotacoes.Categoria categoria0_
      where categoria0_.id=?

FECHOU SESSION 2
```

Listagem 94 - Resultado da Execução do Código Presente na Listagem 93: Cache em Primeiro Nível

Para se utilizar o artifício do cache em segundo nível, ou seja, em nível do processo ou de cluster, é necessário habilitá-lo quando se usa Hibernate. Para exemplificar, será utilizado o *Provider EHCache*, que mantém cache em nível de processo. Para isso, é necessário incluir o **jar ehcache.jar** no classpath da aplicação.

Deve-se informar ao Hibernate que se está utilizando o cache de segundo nível, para isso, é preciso incluir as linhas apresentadas na Listagem 95 no arquivo de configuração **hibernate.cfg.xml**. A primeira propriedade habilita o cache em segundo nível, já a segunda, informa qual o *provider* utilizado.

```
<property name="cache.use_second_level_cache">true</property>
<property name="cache.provider_class">
      net.sf.ehcache.hibernate.EhCacheProvider
</property>
```

Listagem 95 – Configurando o Cache no Arquivo de Configuração *.cfg.xml

Feitas estas alterações, é preciso informar quais entidades terão suas informações armazenadas no cache de segundo nível. Para isso, é necessário inserir anotações extras no mapeamento dessas entidades.

Para exemplificar, considera-se o mapeamento de *cache* na classe **Categoria**, como mostrado na Listagem 96. Para isso, utiliza-se a anotação **@Cache**, onde a estratégia para o nível de isolamento é definida pelo atributo **usage** e pode assumir um dos seguintes valores:

- **CacheConcurrencyStrategy.NONE*;***
- ***Ca*cheConcurrencyStrategy.*READ_WRITE;***
- **CacheConcurrencyStrategy.*READ_ONLY;***
- **CacheConcurrencyStrategy.*TRANSACTIONAL;***
- **CacheConcurrencyStrategy.*NONSTRICT_READ_WRITE.***

```
@Entity
@Table(schema = "anotacoes")
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Categoria {

	@Id
	@SequenceGenerator(allocationSize = 1, name="SEQ")
	@GeneratedValue(  strategy = GenerationType.AUTO,
	                  generator = "SEQ")
	private int id;
	private String denominacao;

	@OneToMany(mappedBy = "categoria", fetch = FetchType.LAZY)
	@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
	private Collection<SubCategoria> subCategorias;

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

Listagem 96 - Mapeamento de Cache na Classe Categoria

Observa-se que a classe **Categoria** possui uma coleção de subcategorias (atributo **subCategorias**). O *cache* não se extende a coleções. Se for necessário aplicar *cache* também em seu conteúdo, deve-se também utilizar a anotação **@Cache** no atributo que represente a coleção, como também foi feito na Listagem 96.

Além de incluir as anotações de cache nas classes, também é necessário configurar as propriedades de classe para cada classe. Essa configuração é feita em um arquivo **ehcache.xml** que deve se encontrar no classpath da aplicação, como apresentado na Listagem 97.

No caso, se for desejado configurar as propriedades de cache para todas as classes anotadas com **@Cache**, é preciso incluir essas propriedades na tag

<cache> no arquivo **ehcache.xml**. Se para alguma classe não for inserida esta tag, o Hibernate considerará a configuração padrão definida na tag **<defaultCache>**.

```
<ehcache>

      <diskStore path="java.io.tmpdir"/>

      <cache
      name="br.com.jeebrasil.hibernate.anotacoes.dominio.Categoria"
        maxElementsInMemory="500"
        eternal="false"
        timeToIdleSeconds="300"
        timeToLiveSeconds="600"
        overflowToDisk="false"
      />

     <defaultCache
        maxElementsInMemory="1000"
        eternal="false"
        timeToIdleSeconds="120"
        timeToLiveSeconds="120"
        overflowToDisk="true"
     />

</ehcache>
```

Listagem 97 – Arquivo de Configuração ehcache.xml

Abaixo a explicação das configurações presentes nas tags **<cache>** e **<defaultCache>** no arquivo **ehcache.xml**:

- **maxElementsInMemory**: número máximo permitido de objetos que podem ficar armazenados em memória;
- **eternal**: se configurado para **true**, significa que o cache nunca vai expirar;
- **timeToIdleSeconds**: tempo em segundos que um objeto pode permanecer inutilizado no cache;
- **timeToLiveSeconds**: tempo em segundos que um objeto pode ficar em cache;

- **overflowToDisk**: se configurado para **true** e caso o limite de objetos em memória seja superior ao definido pela propriedade **maxElementsInMemory**, as informações serão armazenadas em disco.

A configuração **<diskStore path="java.io.tmpdir"/>** informa em que local serão armazenados os objetos se o número máximo permitido de objetos que podem ficar armazenados em memória ultrapassar o definido na propriedade **maxElementsInMemory**. Para esse armazenamento, a propriedade **overflowToDisk** deve ter sido configurada como **true**.

Para exemplificar, considere o resultado da execução do código presente na Listagem 93 apresentado na Listagem 98. Observa-se que o cache de segundo nível foi considerado e apenas um comando SQL foi gerado dentro da primeira sessão (**session1**) ao se consultar a categoria de chave primária igual a **1**.

```
ABRIU SESSION 1

Consulta cat1 (id = 1) -- Objeto session1
Hibernate: select categoria0_.id as id0_0_, categoria0_.denominacao
      as denomina2_0_0_ from anotacoes.Categoria categoria0_
      where categoria0_.id=?
Consulta cat1 (id = 1) -- Objeto session1

FECHOU SESSION 1

ABRIU SESSION 2

Consulta cat3 (id = 1) -- Objeto session2

FECHOU SESSION 2
```

Listagem 98 - Resultado da Execução do Código Presente na Listagem 93: Cache em Segundo Nível