02

Herança e Polimorfismo

Agora que já aprendemos a fazer as operações básicas da loja, vamos implementar as vendas.

Toda venda é feita para um usuário (relacionamento many to one com o usuário)

```
public class Venda
{
    public virtual int Id { get; set; }

    public virtual Usuario Cliente { get; set; }
}
```

Cada venda possui diversos produtos e um produto pode participar de várias vendas, o que caracteriza um relacionamento many to many. Para representar o many to many, colocaremos uma lista de produtos como propriedade da venda e faremos sua inicialização no construtor da classe:

```
public class Venda
{
    public virtual int Id { get; set; }

    public virtual Usuario Cliente { get; set; }

    public virtual IList<Produto> Produtos { get; set; }

    public Venda()
    {
        this.Produtos = new List<Produto>();
    }
}
```

Vamos agora mapear a venda do sistema:

```
<class name="Venda">
    <id name="Id">
        <generator class="identity"/>
    </id>
    <many-to-one name="Cliente" column="ClienteId"/>
</class>
```

No banco de dados, para representarmos um relacionamento many to many, utilizamos uma tabela intermediária que guarda os ids das entidades participantes.

Como vimos anteriormente, toda vez que queremos mapear um uma lista, utilizamos a tag `bag` , porém dessa vez estamos mapeando um relacionamento many to many e por isso, devemos informar qual é o nome da tabela de relacionamento. Nesse exemplo, utilizaremos a tabela `Venda_Produtos` :

```
<bag name="Produtos" table="Venda_Produtos">

</bag>
```

Dentro da tag `bag`, precisamos, também, informar qual é o nome da coluna que representa a chave estrangeira para a venda na tabela de relacionamento:

```
<bag name="Produtos" table="Venda_Produtos">
    <key column="VendaId"/>
</bag>
```

Além disso, devemos informar ao NHibernate que a lista representa um relacionamento do tipo many to many, fazemos isso através da tag `many-to-many`. Dentro dessa tag, devemos informar qual é o nome da coluna que representa a chave estrangeira para o produto na tabela de relacionamento, além disso, precisamos informar qual é a classe da outra entidade que participa do relacionamento:

```
<bag name="Produtos" table="Venda_Produtos">
    <key column="VendaId"/>
    <many-to-many column="ProdutoId" class="Produto"/>
</bag>
```

Criando Vendas

Agora que conseguimos mapear os produtos e as vendas, vamos começar a vender produtos para os clientes!

Quando vendemos um produto, precisamos inicialmente saber para qual cliente estamos vendendo:

```
ISession session = // abre a session
ITransaction transacao = session.BeginTransaction();
Venda venda = new Venda();
Usuario cliente = session.Get<Usuario>(1);
venda.Cliente = cliente;
```

Depois de definirmos para quem estamos vendendo, informaremos que o cliente comprará os produtos com ids 1 e 2, por exemplo.

```
Produto p1 = session.Get<Produto>(1);
Produto p2 = session.Get<Produto>(2);
```

Para relacionar os produtos `p1` e `p2` com a venda, precisamos apenas adicioná-los na lista de produtos, o NHibernate cuidará da sincronização com o banco de dados!

```
venda.Produtos.Add(p1);
venda.Produtos.Add(p2);
```

Agora que já criamos a venda, vamos gravá-la no banco de dados, encerrar a transação e fechar a sessão:

```
session.Save(venda);
transacao.Commit();
session.Close();
```

O código completo para a criação da venda fica da seguinte forma:

```
ISession session = // abre a session
ITransaction transacao = session.BeginTransaction();
Venda venda = new Venda();
Usuario cliente = session.Get<Usuario>(1);
venda.Cliente = cliente;
Produto p1 = session.Get<Produto>(1);
Produto p2 = session.Get<Produto>(2);
venda.Produtos.Add(p1);
venda.Produtos.Add(p2);
session.Save(venda);
transacao.Commit();
session.Close();
```

Venda para empresas

Nossa loja cresceu muito e agora atende também a revendedores, ou seja, nós agora vendemos produtos no atacado e no varejo atendendo empresas e cidadãos comuns, mas nosso modelo atual não atende essa nova regra de trabalho. Precisamos fazer uma modificação para conseguir identificar se o cliente é pessoa física ou pessoa jurídica.

Pessoas físicas possuem todos os atributos que definimos para a entidade `Usuario` e um atributo chamado `CPF` .

```
public class PessoaFisica
{
    public virtual int Id { get; set; }
    public virtual string Nome { get; set; }
    public virtual string CPF { get; set; }
}
```

Já as pessoas jurídicas possuem, além dos atributos do Usuario, um atributo `CNPJ` .

```
public class PessoaJuridica
{
    public virtual int Id { get; set; }
    public virtual string Nome { get; set; }
    public virtual string CNPJ { get; set; }
}
```

Esse modelo atende nossa loja, porém teremos de replicar todas as funcionalidades do `Usuario` para essas duas novas entidades, o que dificulta a manutenção do código. Se precisarmos alterar a lógica do cliente, precisaremos alterar para os dois tipos de cliente. Além disso, para listarmos todos os clientes da loja, precisaríamos de dois `selects` , um na entidade `PessoaFisica` e outro na `PessoaJuridica` .

Na orientação a objetos, quando queremos resolver esse tipo de problema, utilizamos o Polimorfismo. Para aproveitarmos a implementação existente na classe `Usuario`, faremos com que as entidades `PessoaJuridica` e `PessoaFisica` herdem da entidade `Usuario`.

Os modelos ficarão da seguinte forma:

```
public class PessoaFisica : Usuario
{
    public virtual string CPF { get; set; }
}

public class PessoaJuridica : Usuario
{
    public virtual string CNPJ { get; set; }
}
```

Como todos os clientes do sistema devem ser uma instância ou de `PessoaFisica` ou de `PessoaJuridica`, não queremos permitir que a classe `Usuario` seja instanciada, ela, portanto, será uma classe abstrata:

```
public abstract class Usuario
{
    public virtual int Id { get; set; }
    public virtual string Nome { get; set; }
}
```

Agora precisamos informar ao NHibernate que o usuário é abstrato, ou seja, ele não precisa de uma tabela própria no banco de dados. Essa configuração é feita através do atributo `abstract` da tag `class` do arquivo de mapeamento do usuario:

```
<hibernate-mapping>
  <class name="Usuario" abstract="true">
    <!-- mapeamento dos atributos -->
  </class>
</hibernate-mapping>
```

Agora que já definimos as entidades, precisamos mapeá-las no NHibernate. Existem duas estratégias principais para o mapeamento de Herança:

- Tabela Única: O NHibernate utilizará uma única tabela que conterá todas as propriedades mapeadas pela classe pai ou por suas filhas.
- Uma Tabela por subclasse: Nesse mapeamento, o NHibernate cria uma tabela que armazena os dados da classe pai e uma para para cada uma de suas filhas. As tabelas que representam as classes filhas tem um relacionamente do tipo one to one com a tabela da classe pai.

Tabela Única

O mapeamento da hierarquia de classes do `Usuario` é feito no `Usuario.hbm.xml`.

Como estamos armazenando dados de várias entidades em uma única tabela, precisamos diferenciar os registros de alguma forma. Para isso criaremos um campo na tabela `Usuario` chamado `Tipo`, que será do tipo string. Esse campo conterá um

valor que discrimina qual é a entidade representada pela linha da tabela, o valor padrão para esse campo é o nome completo da classe da entidade. O mapeamento desse atributo é feito através da tag `discriminator` .

```
<discriminator column="Tipo" type="System.String" />
```

Essa tag deve ficar logo após o mapeamento do `Id` da entidade. O `Usuario.hbm.xml` fica com o seguinte conteúdo:

```
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
assembly="Loja" namespace="Loja.Entidades">
  <class name="Usuario" abstract="true">
    <id name="Id">
      <generator class="identity"/>
    </id>
    <discriminator column="Tipo" type="System.String" />
    <property name="Nome"/>
  </class>
</hibernate-mapping>
```

Agora precisamos fazer o mapeamento da `PessoaFisica` e da `PessoaJuridica` . Essas entidades serão mapeadas dentro da tag `subclass` que ficará dentro da tag `class` do usuário:

```
<hibernate-mapping>
  <class Name="Usuario">
    <!-- mapeamento do usuário -->

    <subclass name="PessoaFisica">
      <!-- mapeamento da pessoa física -->
    </subclass>
    <subclass name="PessoaJuridica">
      <!-- mapeamento da pessoa jurídica -->
    </subclass>
  </class>
</hibernate-mapping/>
```

Agora só precisamos mapear a propriedade `CPF` na pessoa física e `CNPJ` na pessoa jurídica:

```
<hibernate-mapping>
  <class Name="Usuario">
    <!-- mapeamento do usuário -->

    <subclass name="PessoaFisica">
      <property name="CPF"/>
    </subclass>
    <subclass name="PessoaJuridica">
      <property name="CNPJ"/>
    </subclass>
  </class>
</hibernate-mapping/>
```

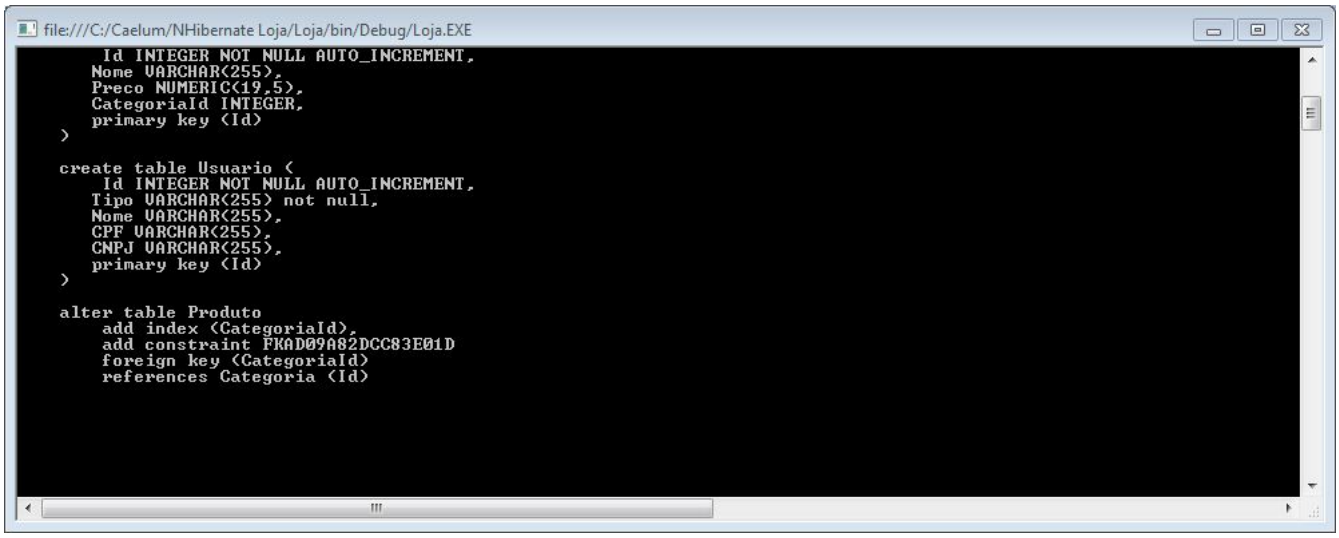
O mapeamento final de nosso novo modelo fica da seguinte forma:

```
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
assembly="Loja" namespace="Loja.Entidades">
  <class name="Usuario" abstract="true">
    <id name="Id">
      <generator class="identity"/>
    </id>
    <discriminator column="Tipo" type="System.String" />

    <property name="Nome"/>

    <subclass name="PessoaFisica">
      <property name="CPF"/>
    </subclass>
    <subclass name="PessoaJuridica">
      <property name="CNPJ"/>
    </subclass>
  </class>
</hibernate-mapping>
```

Quando gerarmos as tabelas com o NHibernate, teremos:



Por fim, vamos inserir dois usuários em nosso banco de dados, um do tipo `PessoaFisica` e outro do tipo `PessoaJuridica`.

```
ISession session = NHibernateHelper.AbreSession();
ITransaction transacao = session.BeginTransaction();

PessoaFisica murilo = new PessoaFisica();
murilo.Nome = "Murilo";
murilo.CPF = "123.456.789.00";
session.Save(murilo);

PessoaJuridica caelum = new PessoaJuridica();
caelum.Nome = "Caelum";
caelum.CNPJ = "123.456/0001-09";
session.Save(caelum);

transacao.Commit();
session.Close();
```

Uma tabela por subclasse

O mapeamento para uma tabela por subclasse é muito parecido com o que fizemos anteriormente para uma única tabela, porém utilizaremos a tag `joined-subclass` ao invés de `subclass` e não precisaremos do campo discriminante, pois os registros já estarão em tabelas distintas.

```
<joined-subclass name="PessoaFisica">
  <property name="CPF"/>
</joined-subclass>
<joined-subclass name="PessoaJuridica">
  <property name="CNPJ"/>
</joined-subclass>
```

Dentro da tabela com as informações da subclasse, o NHibernate precisa colocar uma chave estrangeira para a tabela da classe pai, informamos o nome dessa chave através do atributo `column` da tag `key`. Essa tag deve ser a primeira filha da `joined-subclass`:

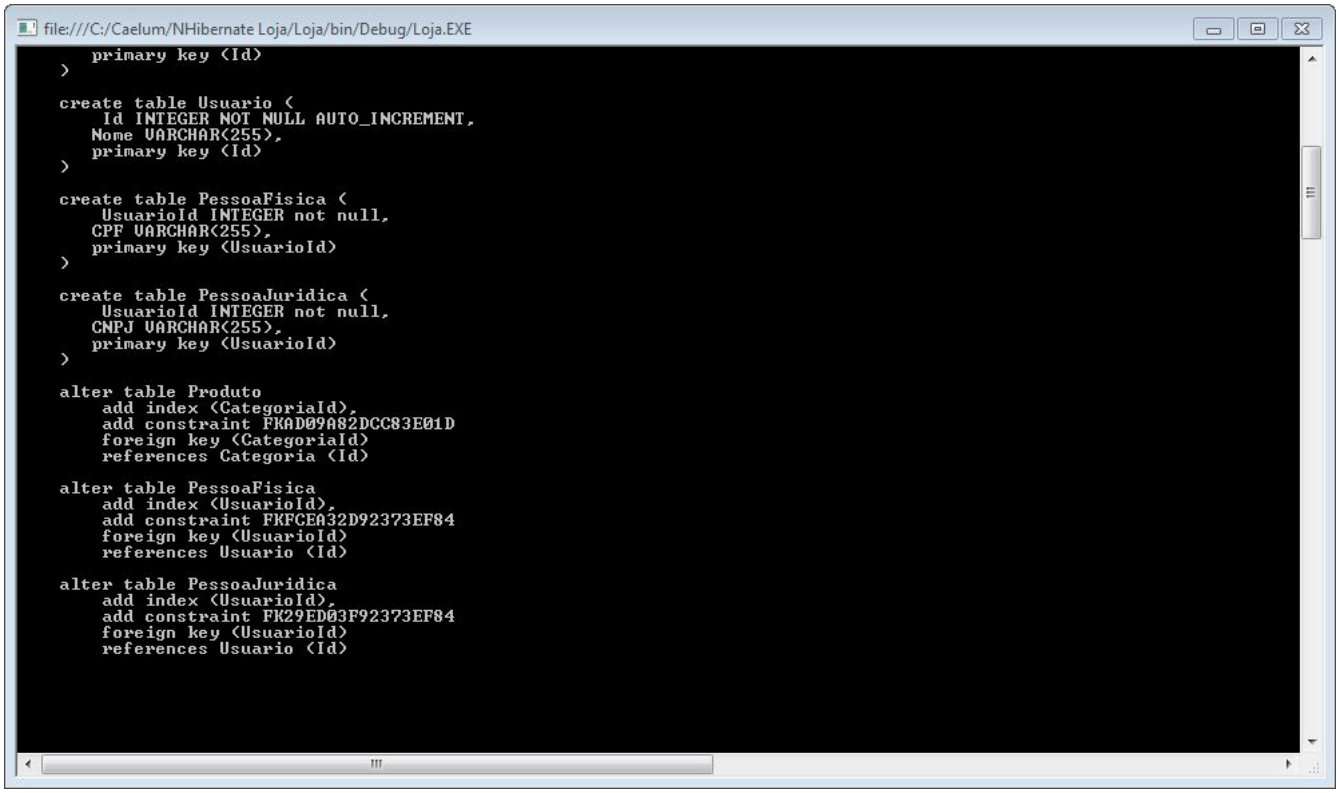
```
<joined-subclass name="PessoaFisica">
  <key column="UsuarioId"/>
  <property name="CPF"/>
</joined-subclass>
<joined-subclass name="PessoaJuridica">
  <key column="UsuarioId"/>
  <property name="CNPJ"/>
</joined-subclass>
```

O `Usuario.hbm.xml` ficará com o seguinte código:

```
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
assembly="Loja" namespace="Loja.Entidades">
  <class name="Usuario" abstract="true">
    <id name="Id">
      <generator class="identity"/>
    </id>
    <property name="Nome"/>

    <joined-subclass name="PessoaFisica">
      <key column="UsuarioId"/>
      <property name="CPF"/>
    </joined-subclass>
    <joined-subclass name="PessoaJuridica">
      <key column="UsuarioId"/>
      <property name="CNPJ"/>
    </joined-subclass>
  </class>
</hibernate-mapping>
```

Vamos gerar novamente as tabelas.



Quando inserirmos novamente a pessoa física, o NHibernate executará um insert na tabela `Usuario` e um na tabela `PessoaJuridica`. O mesmo acontecerá quando inserirmos a pessoa jurídica.