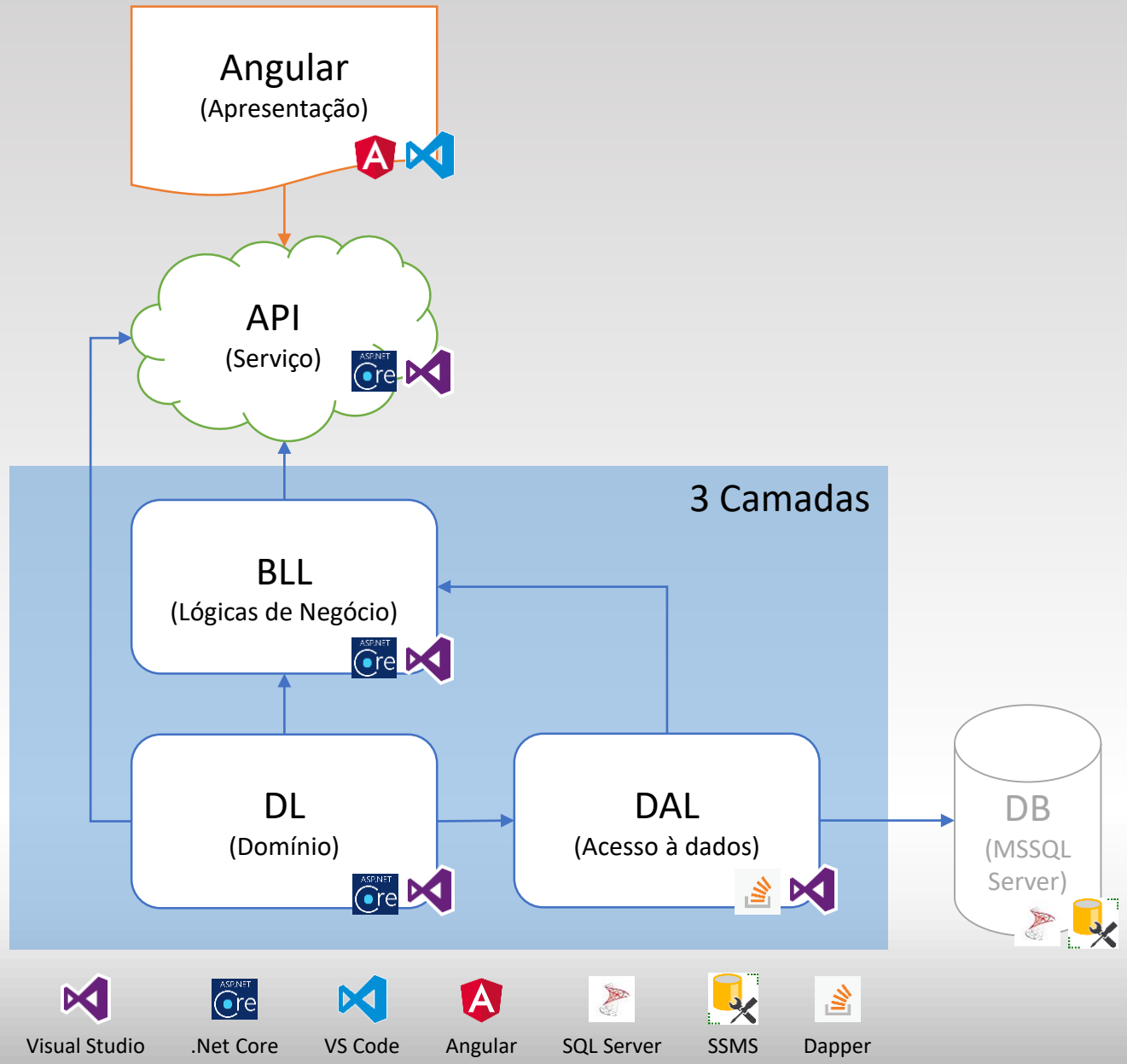
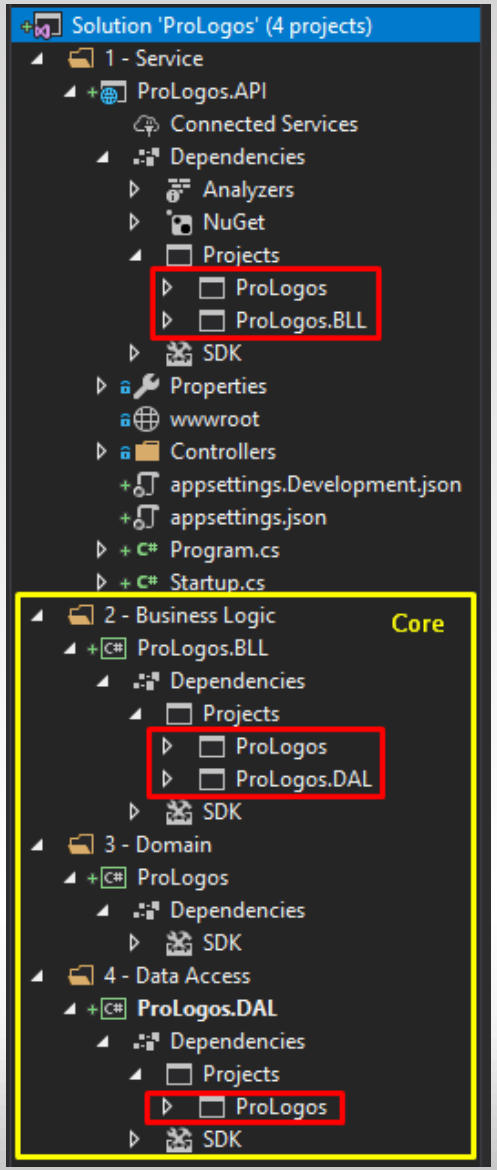
Visão Geral do Projeto



Descrição das responsabilidades

1) DL – Camada de Domínio
   Camada onde são definidos os nossos modelos (entidades) e as suas validações.
2) DAL – Camada de Acesso à Dados
   Camada onde a comunicação com o banco de dados, mapeamento de tabelas e colunas são realizados.
3) BLL – Camada de Lógicas de Negócios
   Camada onde as lógicas de negócio são definidas, toda a inteligência da aplicação estará nessa camada.
4) API – Camada de Serviços
   Basicamente deixa os métodos da camada de lógicas de negócios disponíveis on-line. Esta API poderá ser consumida por qualquer tipo de client, sejam eles browsers, smartphones e etc.
5) Angular – Camada de Apresentação
   Esta é a interface gráfica que o usuário vai ver e interagir, é esta camada que irá consumir os recursos de nossa API.
6) DB – MSSQL Server
   Banco de dados onde as informações do sistema serão salvas, acessada diretamente pela camada DAL.

1. Estrutura do Projeto

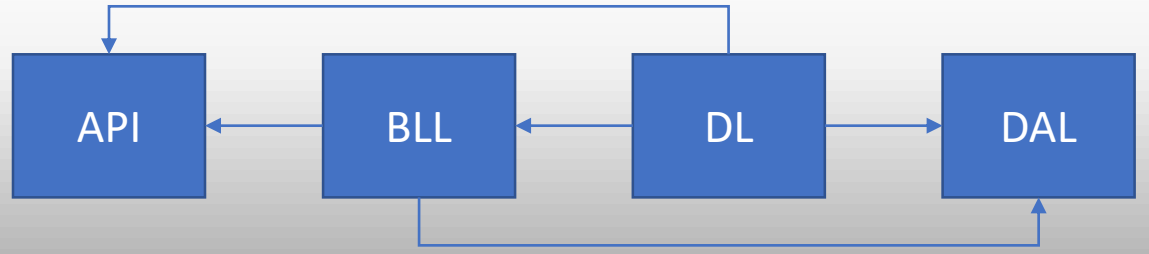


Descrição das responsabilidades

Criamos a solution, as pastas para organização e os projetos que farão parte do back-end do sistema. Além disso nós definimos as referencias entre os nossos projetos .

Estrutura

- Solution
  - 1 - Service Layer
    - ProLogos.API
  - 2 - Business Logic Layer
    - ProLogos.BLL
  - 3 -Domain Layer
    - ProLogos
  - 4 - Data Access Layer
    - ProLogos.DAL



4 Pilares da POO (Programação Orientada à Objetos)

Abstração

A abstração consiste em um dos pontos mais importantes dentro de qualquer linguagem Orientada a Objetos. Como estamos lidando com uma representação de um objeto real (o que dá nome ao paradigma), temos que imaginar o que esse objeto irá realizar dentro de nosso sistema. São três pontos que devem ser levados em consideração nessa abstração: Identidade, propriedades e métodos.

Encapsulamento

O encapsulamento é uma das principais técnicas que define a programação orientada a objetos. Se trata de um dos elementos que adicionam segurança à aplicação em uma programação orientada a objetos pelo fato de esconder as propriedades, criando uma espécie de caixa preta.
A maior parte das linguagens orientadas a objetos implementam o encapsulamento baseado em propriedades privadas, ligadas a métodos especiais chamados getters e setters, que irão retornar e definir o valor da propriedade, respectivamente. Essa atitude evita o acesso direto a propriedade do objeto, adicionando uma outra camada de segurança à aplicação.

Herança

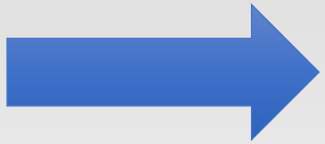
O reuso de código é uma das grandes vantagens da programação orientada a objetos. Muito disso se dá por uma questão que é conhecida como herança. Essa característica otimiza a produção da aplicação em tempo e linhas de código.

Para entendermos essa característica, vamos imaginar uma família: a criança, por exemplo, está herdando características de seus pais. Os pais, por sua vez, herdam algo dos avós, o que faz com que a criança também o faça, e assim sucessivamente

Polimorfismo

Outro ponto essencial na programação orientada a objetos é o chamado polimorfismo. Na natureza, vemos animais que são capazes de alterar sua forma conforme a necessidade, e é dessa ideia que vem o polimorfismo na orientação a objetos. Como sabemos, os objetos filhos herdam as características e ações de seus “ancestrais”. Entretanto, em alguns casos, é necessário que as ações para um mesmo método seja diferente. Em outras palavras, o polimorfismo consiste na alteração do funcionamento interno de um método herdado de um objeto pai.
Como um exemplo, temos um objeto genérico “Eletrodoméstico”. Esse objeto possui um método, ou ação, “Ligar()”. Temos dois objetos, “Televisão” e “Geladeira”, que não irão ser ligados da mesma forma. Assim, precisamos, para cada uma das classes filhas, reescrever o método “Ligar()”.

Classes Vs Objetos



Classe

Objeto

Objeto nada mais é do que uma instância de uma classe, este objeto possuirá as ações e características que foram definidas na classe.

2. Propriedades

```
1   using ProLogos.Base;
2   using System;
3
4   namespace ProLogos
5   {
6       public class Produto : BaseDomain
7       {
8           public string Codigo { get; set; }
9           public string Descricao { get; set; }
10          public decimal SaldoEstoque { get; set; }
11          public decimal ValorCompra { get; set; }
12          public decimal ValorVenda { get; set; }
13          public DateTime DataCadastro { get; set; }
14          public bool Ativo { get; set; }
15      }
16  }
17
```

As propriedades podem ser consideradas como as características que um objeto possui.

Como podemos perceber ao lado temos uma classe **Produto** que possui as seguintes propriedades (características**): Codigo, Descricao, SaldoEstoque, ValorCompra, ValorVenda, DataCadastro e Ativo**.

Podemos também traçar um paralelo mais genérico com um ser humano, por exemplo, ao definir uma classe **Pessoa** poderíamos adicionar as seguintes propriedades: **CorDosOlhos, Altura, Peso, Nome e etc.**

Getters e Setters

```
0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
1 public int Numero1 { get; set; }

  private int _numero2;
  0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
2 public int Numero2
  {
      get { return _numero2; }
      set { _numero2 = value; }
  }

3 private int _numero3;
  0 references | 0 changes | 0 authors, 0 changes | 0 exceptions
  public int Numero3 { get => _numero3; set => _numero3 = value; }
```

Getters e Setters são usados para encapsular propriedades. Os três exemplos teriam o mesmo efeito em nossa aplicação (São a mesma coisa), entretanto em nossas aplicações acabaremos tendo que utilizar os 3 formatos de acordo com o que se é necessário fazer.

3. Construtores

```
0 references | Luiz R. A. Coelho, 4 hours ago | 1 author, 1 change | 0 exceptions
public Marca(string nome, string segmento = null)
{
    Nome = nome;
    Segmento = segmento;
    DataCadastro = DateTime.Now;
}
```

Os construtores são executados sempre que um novo objeto é instanciado, ou seja, é possível programar ações ou obrigar que algumas informações necessárias sejam preenchidas na própria concepção do objeto.

4. Métodos

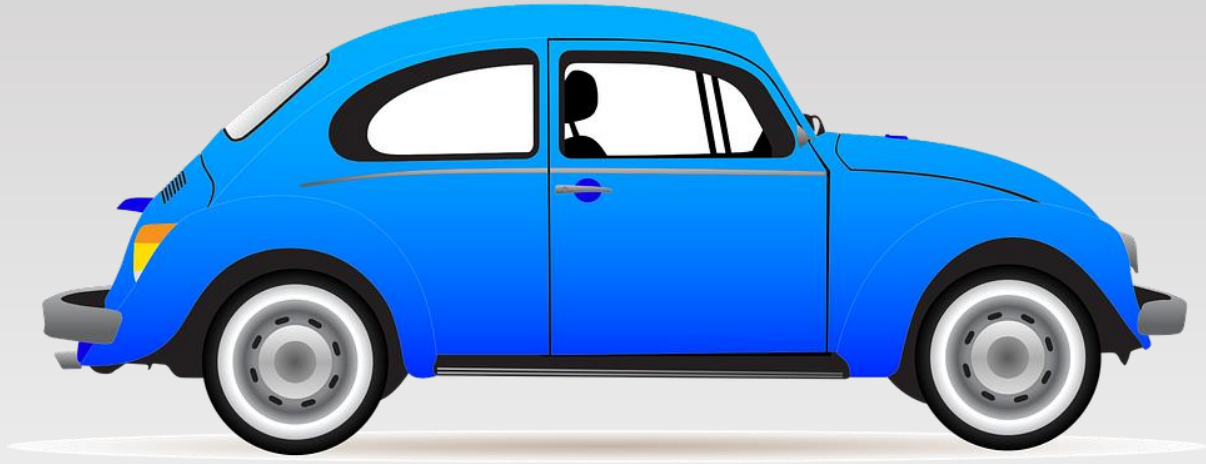
```
private void SetDescricao(string descricao)
{
    if (string.IsNullOrEmpty(descricao))
        throw new ArgumentNullException(nameof(Descricao), "O campo código deve ser preenchido.");
    else if (descricao.Length > 30)
        throw new ArgumentOutOfRangeException(nameof(Descricao), "O campo código deve ter até 30 caracteres.");

    _descricao = descricao;
}
```

Os métodos podem ser considerados como as ações que um objeto possui.

Como podemos perceber aacima temos uma classe **Produto** que possui o seguinte método (ação**): SetDescricao()**, o intuito deste método é verificar se uma descrição foi preenchida e se possui 30 caracteres ou menos, caso tudo ocorra bem o valor é atribuído à variável _descrição, caso contrário uma exceção é lançada.

Podemos também traçar um paralelo mais genérico com um ser humano, por exemplo, ao definir uma classe **Pessoa** poderíamos adicionar os seguintes métodos: **Andar, Piscar, Respirar e etc.**

Tipos primitivos Vs complexos

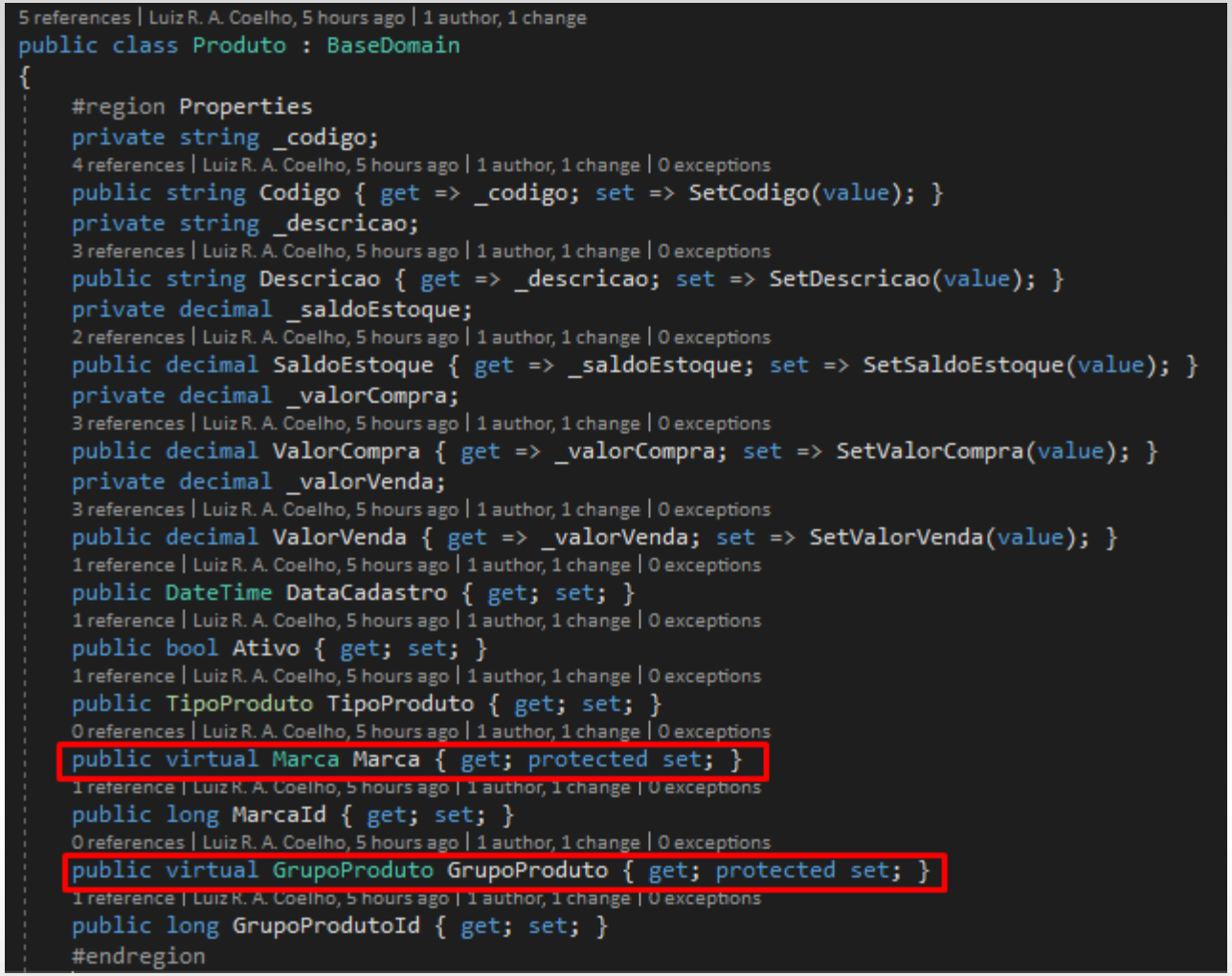


Primitivo

Complexo

As classes podem ser formadas por tipos primitivos (**int, decimal, double, string e etc.**) ou por tipos complexos. Temos alguns tipos complexos provenientes da própria linguagem como por exemplo **DateTime** e outros que nós mesmos podemos adicionar.
Trocando em miúdos, uma classe pode se constituída não somente de tipos primitivos, mas também de outras classes.

5. Relacionamentos



```
5 references | Luiz R. A. Coelho, 5 hours ago | 1 author, 1 change
public class Produto : BaseDomain
{
    #region Properties
    private string _codigo;
    4 references | Luiz R. A. Coelho, 5 hours ago | 1 author, 1 change | 0 exceptions
    public string Codigo { get => _codigo; set => SetCodigo(value); }
    private string _descricao;
    3 references | Luiz R. A. Coelho, 5 hours ago | 1 author, 1 change | 0 exceptions
    public string Descricao { get => _descricao; set => SetDescricao(value); }
    private decimal _saldoEstoque;
    2 references | Luiz R. A. Coelho, 5 hours ago | 1 author, 1 change | 0 exceptions
    public decimal SaldoEstoque { get => _saldoEstoque; set => SetSaldoEstoque(value); }
    private decimal _valorCompra;
    3 references | Luiz R. A. Coelho, 5 hours ago | 1 author, 1 change | 0 exceptions
    public decimal ValorCompra { get => _valorCompra; set => SetValorCompra(value); }
    private decimal _valorVenda;
    3 references | Luiz R. A. Coelho, 5 hours ago | 1 author, 1 change | 0 exceptions
    public decimal ValorVenda { get => _valorVenda; set => SetValorVenda(value); }
    1 reference | Luiz R. A. Coelho, 5 hours ago | 1 author, 1 change | 0 exceptions
    public DateTime DataCadastro { get; set; }
    1 reference | Luiz R. A. Coelho, 5 hours ago | 1 author, 1 change | 0 exceptions
    public bool Ativo { get; set; }
    1 reference | Luiz R. A. Coelho, 5 hours ago | 1 author, 1 change | 0 exceptions
    public TipoProduto TipoProduto { get; set; }
    0 references | Luiz R. A. Coelho, 5 hours ago | 1 author, 1 change | 0 exceptions
    public virtual Marca Marca { get; protected set; }
    1 reference | Luiz R. A. Coelho, 5 hours ago | 1 author, 1 change | 0 exceptions
    public long MarcaId { get; set; }
    0 references | Luiz R. A. Coelho, 5 hours ago | 1 author, 1 change | 0 exceptions
    public virtual GrupoProduto GrupoProduto { get; protected set; }
    1 reference | Luiz R. A. Coelho, 5 hours ago | 1 author, 1 change | 0 exceptions
    public long GrupoProdutoId { get; set; }
    #endregion
```

Trabalhar com relacionamento em nossos modelos, nada mais é do que criar uma classe se utilizando de outras classes, neste exemplo da classe **Produto** podemos ver que ela possui uma propriedade **Marca** e outra **GrupoProduto**, ambas tipos complexos.

Um ponto que vale ser ressaltado neste caso é a presença das propriedades **MarcaId** e **GrupoProdutoId**. Estas propriedades têm como intuito prover uma funcionalidade para o banco de dados chamado **Fluent Api**.

Nos aprofundaremos neste conceito mais à frente.

6. Enumeradores

```
namespace ProLogos.Enums
{

    public enum TipoProduto
    {
        ProdutoAcabado = 1,
        MateriaPrima = 2
    }
}
```

Enumeradores têm como objetivo tratar um valor inteiro simples como um nome que faça sentido em nossa aplicação.

Neste exemplo de enumerador temos dois valores: ProdutoAcabado que equivale à 1 e MateriaPrima que equivale à 2.

Quando formos definir um tipo de produto usaremos a seguinte sintaxe: **TipoProduto.ProdutoAcabado**, entretanto, no banco de dados será salvo o valor 1.