INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE, ESCOLA SUPERIOR DE TECNOLOGIA

# Licenciatura em Engenharia de Sistemas Informáticos

José António da Cunha Alves
nº27967

17 de Novembro de 2024


Fase 2

Trabalho Prático UC Programação Orientada a Objetos

Docente: Luís Ferreira

# CONTEÚDOS

# AMOSTRAS DE CÓDIGO

# 1 ENUNCIADO

## 1.1 Motivação

Pretende-se que sejam desenvolvidas soluções em C# para problemas reais de complexidade moderada. Serão identificadas classes, definidas estruturas de dados e implementados os principais processos que permitam suportar essas soluções. Pretende-se ainda contribuir para a boa redação de relatórios.

## 1.2 Objetivos

- Consolidar conceitos basilares do Paradigma Orientado a Objetos;
- Analisar problemas reais;
- Desenvolver capacidades de programação em C#;
- Potenciar a experiência no desenvolvimento de software;
- Assimilar o conteúdo da Unidade Curricular.

## 1.3 Problema a Explorar

(vii) Comércio eletrónico: sistema que permita a gestão de uma loja online. *keywords* : **produtos, categorias, garantias, stocks, clientes, campanhas, vendas, marcas.**

# 2 Introdução

Este documento é uma descrição do trabalho realizado na primeira fase do trabalho prático da unidade curricular de Programação Orientada a Objetos. Neste trabalho, é proposto desenvolver um programa que torne possível a gestão de uma loja online, tendo como termos indispensáveis produtos, garantias, vendas, clientes, categorias, stocks, campanhas e marcas. Assim sendo, é necessário que o programa contemple funções que permitam todo o tipo de operações de gestão, desde criação de ficha de produto/cliente, bem como a gestão dos stocks dos respetivos produtos, datas de fim de garantias, datas de venda, entre outros. Esta primeira fase contempla apenas a definição de classes indispensáveis ao projeto, bem como as funcionalidades básicas de gestão das mesmas.
Este trabalho tem como objetivos: a consolidação de conceitos basilares do Paradigma Orientado a Objetos; a análise de problemas reais, neste caso, de gestão de uma loja; o desenvolvimento de capacidades de programação em C#; o potenciamento da experiência no desenvolvimento de software e a assimilação do conteúdo lecionado na Unidade Curricular em questão.

Todo o código fonte e respetiva documentação podem ser encontrados no seguinte repositório **GitHub**.

# 3 Revisão de Leitura

- Programação Orientada a Objetos – Material das aulas;
- Documentação – *Doxygen Quick Reference*;
- Qualidade do Código – *Clean Code – A Handbook of Agile Software Craftsmanship*, de Robert C. Martin.
- *.NET Coding Conventions* – Microsoft.

# 4 Convenções de Nomenclatura Utilizadas (*Microsoft .NET Coding Conventions*)

## 4.1 Classes e Interfaces

- Classes e tipos públicos devem ter nomes em ***PascalCase***.
- Interfaces devem começar com a letra "I", seguida de um nome em ***PascalCase*** (ex: *IList-Management, IClient*).

## 4.2 Métodos

- Deve utilizar-se ***PascalCase*** para métodos públicos e internos (ex: *AddClient, Remove-ProductFromSale*).

## 4.3 Propriedades e Campos

- Propriedades públicas e internas em ***PascalCase*** (ex: *MakeList, ClientList*).
- Atributos privados e variáveis de instância usam ***camelCase*** e um prefixo '_' (ex: *_id, _durantionInYears* ).
- Constantes e campos *readonly* podem ser nomeados em ***PascalCase*** (ex: *MaxProducts*).

**Amostra de Código 1: A Classe Cliente**

```
1
2     public class Client
3   {
4     #region Attributes
5     int _clientID;
6     string _name;
7     string _contact;
8     static int _clientCount=0;
9     #endregion
10  }
```

## 4.4 Variáveis Locais e Parâmetros

- Deve utilizar-se ***camelCase*** para variáveis locais e parâmetros (ex: *campList*).
- Devem escolher-se nomes descritivos para melhorar a clareza, evitando abreviações excessivas.

# 5 Padrões e Convenções de Estilo (*Microsoft .NET Coding Conventions*)

## 5.1 Espaçamento e Identação

- Identação com 4 espaços (não usar tabulações), para manter o padrão da maioria dos editores de C#.

## 5.2 Colocação de Chavetas

- Devem usar-se chavetas de abertura '{' na linha seguinte à declaração (*if, for, while*), assim como as chavetas de fecho.

## 5.3 Comentários e XML Documentation

- Devem comentar-se métodos e classes utilizando **comentários XML (///)**. Descreva parâmetros e valor de retorno, incluindo exceções lançadas.
- Devem utilizar-se tags XML como <*summary*>, <*param*>, <*returns*>, <*exception*>, para fornecer documentação completa.

## 5.4 Nomes de Ficheiros

- Ficheiros de código devem ser nomeados de acordo com a classe pública principal que ele contém. Por exemplo, a classe *Client* deve estar no ficheiro *Client.cs*.

## 5.5 Tratamento de Excecões

- Devem utilizar-se exceções claras e específicas.
- Deve evitar-se capturar exceções genéricas sem tratamento adequado.

**Amostra de Código 2: Exemplo de Utilização de Padrões e Convenções de Estilo**

```
1
2   /// <summary>
3  /// Create a Client and add it to the store's list of clients.
4  /// </summary>
5  /// <param name="name">Client's Name</param>
6  /// <param name="contact">Client's Contact</param>
7  /// <returns>True - Client Successfully created and added to the list.</
       returns>
8  /// <returns>Exception - An error occurred in the process.
9  public static bool CreateClientInStore(string name, string contact)
10 {
11    try
12    {
13      if (BestSale_Validations.BestSale_Validations.ValidatePhoneNumber(
            contact))
14      {
15        bool aux = Client.CreateClientFromNameContact(name, contact, out
              Client newClient);
16        aux = Store.InsertClientInStore(newClient);
17        return aux;
18      }
19      return false;
20    }
21    catch(Exceptions.InvalidPhoneNumberException invalidPhoneNumber)
22    {
23      throw invalidPhoneNumber;
24    }
25    catch (Exception excep)
26    {
27      throw (excep);
28    }
29 }
```

# 6 UTILIZAÇÃO DE *LINQ*

*LINQ* (*Language Integrated Query*) é um recurso da linguagem C# que permite a consulta e manipulação de coleções de dados de forma consistente e expressiva. Este recurso unifica a forma de consultar diferentes estruturas de dados, sejam *arrays*, *listas*, bases de dados, entre outros, utilizando sintaxe similar à de consultas SQL.

## 6.1 Principais Características do *LINQ*

### *6.1.1 Integração com a Linguagem*

LINQ é integrado diretamente na linguagem C#, permitindo a execução de consultas diretamente no código, sem necessidade de *strings* SQL separadas ou outras linguagens externas.

### *6.1.2 Suporte a Diversas Fontes de Dados*

- O LINQ pode ser utilizado para trabalhar com:
  - **Coleções em memória**: como *List<T>*, *Array*, *Dictionary<TKey, TValue>* (via *LINQ to Objects*).
  - **Bases de Dados**: como o SQL Server (via *LINQ to SQL* ou *Entity Framework*).
  - **XML**: consulta e manipulação de dados em formato XML (via *LINQ to XML*).

### *6.1.3 Sintaxe Declarativa*

- Permite descrever **o que** se quer fazer (o resultado desejado) em vez de **como** fazer (detalhes de implementação).
- Exemplos de operadores: *where*, *select*, *order by*, *group by*.

### *6.1.4 Strongly Typed*

- O compilador verifica a consulta *LINQ* no momento da compilação, ajudando a evitar erros, identificando-os antes da execução do programa.

## 6.2 *Benefícios do LINQ*

- **Consistência**: Uma única abordagem para consultar diferentes estruturas de dados.
- **Legibilidade**: A sintaxe declarativa facilita a compreensão do código.
- **Segurança de Tipos**: Deteção de erros aquando da compilação.
- **Redução de Código**: Evita códigos complexos e repetitivos.

## 6.3 Limitações

- Pode ser menos eficiente em alguns casos específicos, dependendo do contexto e da estrutura de dados.
- Em bases de dados, a tradução para SQL pode gerar consultas menos otimizadas se não for bem configurada.

O *LINQ* é amplamente usado em C# devido à sua simplicidade e flexibilidade na manipulação de dados.

**Amostra de Código 3: Utilização de Função Lambda**

```
1
2 public decimal TotalPrice()
3 {
4   return _prods.Sum(p => p.Price);
5
6 }
```

A utilização desta *lambda function* (utilizando o operador '=>'), indica à função *Sum()* que, para cada Produto *p* em *_prods*, deve utilizar o valor da propriedade *Price*.

# 7 O PADRÃO N-TIER

O padrão **N-Tier** (ou arquitetura por camadas) é uma abordagem arquitetural para a construção de sistemas de software que organiza a aplicação em várias camadas lógicas, cada uma com responsabilidades distintas. Este padrão promove a separação de preocupações, facilitando a manutenção, escalabilidade e reutilização do código.
O "N" no nome refere-se ao número de camadas, que pode variar conforme a complexidade do sistema. Contudo, os sistemas mais comuns baseiam-se em 3 camadas principais. No caso deste programa, **temos as seguintes camadas**:

- **Camada de Apresentação** *Front End* - BestSale
- **Camada de Regras de Negócio** *Business Layer* - Business_Layer
- **Camada de Dados** *Back End* - Data_BestSale

Temos ainda algumas **camadas secundárias**, que atravessam as restantes. São elas:

- **Camada de Exceções** *Exceptions Layer* - Exceptions
- **Camada de Objetos de Negócio** *Business Objects* - Business_Object
- **Camada de Validações** *Validations* - BestSale_Validations. Esta não é visível no *Front End*, mas apenas nas duas restantes camadas.

## 7.1 Vantagens da Arquitetura N-Tier

A arquitetura N-Tier apresenta diversas vantagens que a tornam uma escolha popular no desenvolvimento de sistemas. Em primeiro lugar, promove a **separação de responsabilidades**, pois cada camada assume uma função bem definida, reduzindo o acoplamento entre diferentes partes do sistema. Além disso, oferece **escalabilidade**, uma vez que permite escalar camadas individuais conforme necessário, como adicionar servidores específicos apenas para a camada de dados. Outra vantagem significativa é a **facilidade de manutenção**, pois alterações realizadas numa camada, como a substituição da base de dados, não afetam diretamente as demais camadas. Por fim, a arquitetura N-Tier **facilita os testes**, já que cada camada pode ser testada de forma isolada, permitindo identificar e corrigir problemas com maior eficiência.

# 8 UTILIZAÇÃO DE *SOLID*

***SOLID*** é um conjunto de cinco **princípios de design** de software orientado a objetos que ajudam a criar sistemas mais robustos, flexíveis e fáceis de manter. O primeiro princípio, ***Single Responsibility Principle***, estabelece que uma classe deve ter apenas uma responsabilidade, ou seja, deve haver apenas uma razão para ela sofrer alterações. O segundo, ***Open/Closed Principle***, afirma que o software deve ser aberto para extensão, mas fechado para modificação, permitindo adicionar funcionalidades sem alterar o código existente. O terceiro, ***Liskov Substitution Principle*** , sugere que os subtipos devem ser substituíveis pelos seus tipos base sem alterar o comportamento do sistema. O quarto princípio, ***Interface Segregation Principle*** , diz que uma classe não deve ser obrigada a implementar interfaces que não utiliza, promovendo interfaces mais específicas e menores. Por fim, o ***Dependency Inversion Principle*** defende que módulos de alto nível não devem depender de módulos de baixo nível; ambos devem depender de abstrações, reduzindo o acoplamento e melhorando a flexibilidade do sistema. Esses princípios, juntos, promovem um design de software mais organizado e sustentável.
Neste projeto, tentou-se ao máximo seguir estes princípios. Os que têm presença mais notória são o ***Single Responsibility Principle*** e o ***Open/Closed Principle***.

# 9 RESPEITAR A *.NET CLS*

A *Common Language Specification (CLS)* é uma componente fundamental da plataforma .NET que define regras e convenções destinadas a garantir a **interoperabilidade entre linguagens de programação diferentes** dentro do ecossistema .NET. A CLS define uma base comum de funcionalidades que todas as linguagens compatíveis com .NET devem suportar, permitindo que código escrito numa linguagem seja usado noutra sem problemas de compatibilidade. Por exemplo, linguagens como C#, VB.NET e F# podem interagir e partilhar bibliotecas graças à conformidade com a CLS. Algumas destas regras incluem a utilização de tipos de dados padrão e a proibição de recursos específicos de linguagem que possam não ser suportados por outras. Assim, a CLS desempenha um papel crucial na criação de bibliotecas reutilizáveis e na garantia de que componentes escritos em linguagens diferentes possam ser integrados de forma fluída dentro de uma mesma aplicação. Neste projeto, por exemplo, foi utilizado o tipo *decimal* para definir os preços dos produtos (valores financeiros) devido à sua maior precisão e respeitando a CLS.

# 10 TESTES UNITÁRIOS

Os testes unitários em .NET são uma prática essencial para garantir a qualidade e fiabilidade do código. Estes verificam o comportamento de pequenas unidades de código, como métodos ou classes, de forma isolada. Neste projeto, utilizei o **MSTest** para fazer alguns testes da classe *Client*.

**Amostra de Código 4: Teste Unitário do construtor de Client, quando é indicado um contacto inválido**

```
1
2  [TestMethod]
3 public void Constructor_InvalidContact_ThrowsInvalidPhoneNumberException
      ()
4 {
5   //Arrange
6   var name = "Jose Alves";
7   var invalidContact = "123456789";
8
9   //Act & Assert
10  try
11  {
12    var client = new Client(name, invalidContact);
13    Assert.Fail("Expected InvalidPhoneNumberException not thrown");
14  }
15  catch (InvalidPhoneNumberException exception)
16  {
17    Assert.AreEqual("Invalid Phone Number", exception.Message);
18  }
```

# 11 TRABALHO DESENVOLVIDO

## 11.1 Diagrama de Classes

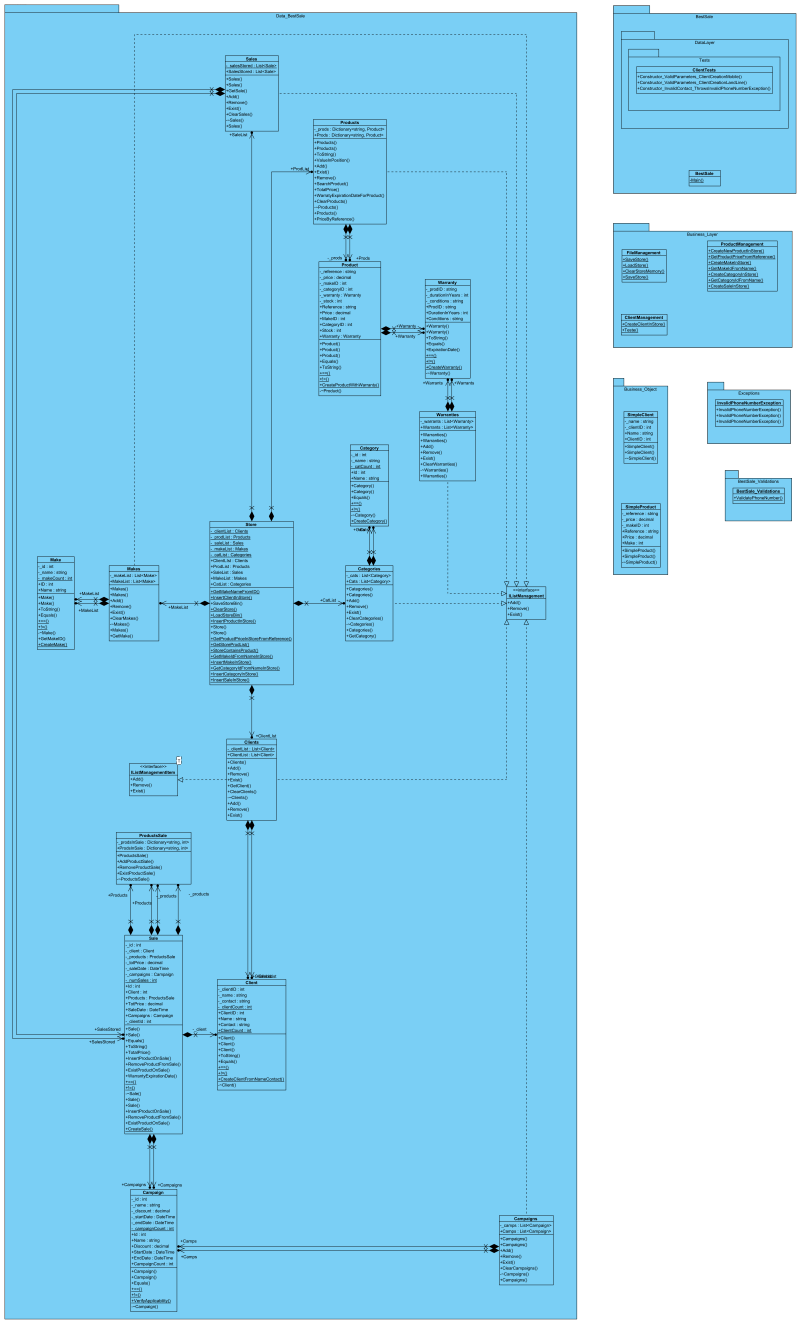

Figure 11.1: Diagrama de Classes

## 11.2 Estrutura de Projetos

### *11.2.1 Bibliotecas (.dll)*

- *Data_BestSale.dll*
  - ***Campaign.cs*** – Definição de atributos de campanha, propriedades e métodos para gestão das mesmas.
  - ***Campaigns.cs*** – Classe de agregação de ***Campaign.cs***, que contém os métodos para gestão da pluralidade.
  - ***Category.cs*** – Definição de atributos de categoria, propriedades e métodos para gestão das mesmas.
  - ***Categories.cs*** – Classe de agregação de ***Category.cs*** que contém os métodos para gestão da pluralidade.
  - ***Client.cs*** – Definição de atributos de cliente, propriedades e métodos para gestão dos mesmos.
  - ***Clients.cs*** – Classe de agregação de ***Client.cs***, que contém os métodos para a gestão da pluralidade.
  - ***Make.cs*** – Definição de atributos de marca, propriedades e métodos para gestão das mesmas.
  - ***Makes.cs*** – Classe de agregação de ***Make***, que contém os métodos para gestão da pluralidade.
  - ***Product.cs*** – Definição de atributos de produto, propriedades e métodos para gestão dos mesmos.
  - ***Products.cs*** – Classe de agregação de ***Product.cs***, que contém os métodos para gestão da pluralidade.
  - ***Sale.cs*** – Definição de atributos de venda e propriedades e métodos para gestão das mesmas.
  - ***Sales.cs*** – Classe de agregação de ***Sales***, que contém métodos para a gestão da pluralidade.
  - ***Store.cs*** – Definição de atributos de loja, propriedades e métodos para gestão da mesma.
  - ***Warranty.cs*** – Definição de atributos de garantia, propriedades e métodos para gestão das mesmas.
  - ***Warranties.cs*** - Classe de agregação de ***Warranty.cs***, que contém métodos para a gestão da pluralidade.
  - ***IListManagement.cs*** – Interface que mostra como implementar as funções de gestão de listas. (*Adicionar*, *Remover*, *Existe*).
- *Business_Object.dll*
  - ***SimpleProduct.cs*** – Definição de atributos de um cliente simples e propriedades. Serve para agilizar a circulação de dados entre o *back end* e a *business layer*.
- *BestSale_Validations.dll*
  - ***BestSale_Validations.cs*** – Definição dos métodos que permitem fazer validações de dados inseridos.
- *Exceptions.dll*
  - ***InvalidPhoneNumberException.cs*** – Definição da exceção apresentada quando um contacto telefónico inserido não respeita o padrão definido.

### *11.2.2 Business Layer*

- *Business_Layer.dll*

- ***ClientManagement.cs*** – Contém os métodos intermediários de gestão de clientes, que permitem a comunicação entre o *back end* e a *front end*.
- ***ProductManagement.cs*** – Contém os métodos intermediários de gestão de produtos, que permitem a comunicação entre o *back end* e a *front end*. Contém também os métodos intermediários que dizem respeito a marcas e categorias.
- ***FileManagement.cs*** – Contém os métodos intermediários de gestão de ficheiros, que permitem a comunicação entre o *back end* e a *front end*.

### *11.2.3 Front End*

- ***BestSale.cs*** – Classe Principal.

## 11.3 Observações

No interface desenvolvido, visto que a versão de .NET que estava a utilizar era ligeiramente mais antiga, não me foi possível utilizar generalizações do tipo *T<Item>*. Assim sendo, tive de recorrer a outro método, o que tornou o código mais repetitivo e, de certa forma, inutilizou a utilização do interface, pois envolve fazer vários testes em todas as implementações do interface. Seguem dois exemplos:

**Amostra de Código 5: Implementação do Interface**

```
1
2   public class Products : IListManagement
3   {
4     /// This method inserts a product in a list of products.
5     public bool Add(object obj)
6     {
7       if (obj == null) return false;
8       var aux=obj as Product;
9       if (Exist(aux.Reference))
10      {
11        if (obj is Product)
12        {
13          _prods.Add((Product)obj);
14          return true;
15        }
16      }
17      return false;
18    }
19    /// Method used to verify if a product is on a products' list, given
          its Reference.
20    public bool Exist(object obj)
21    {
22      if (obj == null) return false;
23      if (obj is string)
24      {
25        foreach (Product p in _prods)
26        {
27          if (p.Reference == (string)obj) return true;
28        }
29      }
30      return false;
31    }
32    /// Method used to remove a product from a Products' list.
33    public bool Remove(object obj)
34    {
35      if (obj == null) return false;
36      Product p = (Product)obj;
```

```
37        if (Exist(p.Reference))
38        {
39          _prods.Remove(p);
40          return true; ///Product removed successfully
41        }
42        return false; ///Product was not removed.
43      }
44    }
```

Aquando da passagem de um projeto único para a implementação do *design pattern N-Tier*, tentei fazer pelo menos um exemplo de um interface utilizando *T<Item>*. Surgiu assim o interface *IListManagementItem*.

**Amostra de Código 6: Implementação do Interface "T<Item>"**

```
1
2   public class Clients : IListManagementItem<Client>
3   {
4     static List<Client> _clientList;
5     /// <summary>
6     /// Method to add a client to a clients' list.
7     /// </summary>
8     public bool Add(Client client)
9     {
10      if(client==null || Exist(client))
11      {
12        return false;
13      }
14      _clientList.Add(client);
15      return true;
16    }
17    /// <summary>
18    /// Method to remove a client from the store's client list.
19    /// </summary>
20    public bool Remove(Client client)
21    {
22      if (client == null || !(Exist(client)))
23      {
24        return false;
25      }
26      _clientList.Remove(client);
27      return true;
28    }
29    /// <summary>
30    /// Method to check if a client is listed on a clients' list
31    /// </summary>
32    public bool Exist(Client client)
33    {
34      foreach(Client _client in _clientList)
35      {
36        if(_client.ClientID==client.ClientID)
37        {
38          return true;
39        }
40      }
41      return false;
42    }
43  }
```

Mais ainda, apesar de, para tornar o código mais modular e melhor organizado , se deva separar as classes o mais modularmente possível, optei por colocar dentro da classe *ProductManagement* também as funções que permitam a interação entre *front end* e *back end* no que toca à criação e pesquisa de marcas, categorias e garantias (indo um pouco contra os princípios do *SOLID*). Esta opção foi tomada, visto que, num programa com esta pequena dimensão, a classe em questão não se torna caótica e, ao mesmo tempo, evita-se criar classes com apenas dois ou três métodos. Mais ainda, tendo em conta que, para criar um produto, é necessário ter também os registos destas restantes vertentes, faz sentido que estas possam também ser manipuladas nesta classe. Ainda assim, todas as partes estão distinguidas com a sua devida *#region*, como se vê no exemplo seguinte:

**Amostra de Código 7: Classe ProductManagement.cs**

```
1
2   public class ProductManagement
3   {
4       #region Products
5
6     /// Method used to create and add a product to a store.
7     public static bool CreateNewProductInStore(string reff, decimal
          price, int makeID, int categoryID, int warrantyDuration, string
          warrantyConditions)
8     {
9       try
10      {
11        Product prod = Product.CreateProductWithWarranty(reff, price,
              makeID, categoryID, warrantyDuration, warrantyConditions);
12        Store.InsertProductInStore(prod);
13        return true;
14      }
15      catch(Exception excep)
16      {
17        throw excep;
18      }
19    }
20
21    /// Mtehod that returns the price of a certain product, given its
          reference.
22    public static decimal GetProductPriceFromReference(string reference)
23    {
24      return Store.GetProductPriceInStoreFromReference(reference);
25    }
26    #endregion
27
28    #region Make
29
30    /// This method creates a new make and inserts it on the store's
          list of makes.
31    public static bool CreateMakeInStore(string name)
32    {
33      Make.CreateMake(name, out Make newMake);
34      return Store.InsertMakeInStore(newMake);
35    }
36
37    /// This method returns the ID of a make in the store's list, given
          its name.
38    public static int GetMakeIdFromName(string name)
39    {
40      return Store.GetMakeIdFromNameInStore(name);
```

```
41      }
42
43      #endregion
44
45      #region Category
46
47      /// This method creates and inserts a Category in a store's list.
48      public static bool CreateCategoryInStore(string name)
49      {
50        Category.CreateCategory(name, out Category newCategory);
51        return Store.InsertCategoryInStore(newCategory);
52      }
53
54      /// This method returns the ID of a Category in the store's list,
            given its name.
55      public static int GetCategoryIdFromName(string name)
56      {
57        return Store.GetCategoryIdFromNameInStore(name);
58      }
59      #endregion
60  }
```

# 12 Estruturas de Dados

As estruturas de dados são fundamentais na programação, responsáveis por organizar, armazenar e manipular informaçâo de forma eficiente. Estas permitem que os dados sejam acedidos e processados com rapidez, o que é essencial para a execução de algoritmos e resolução de problemas complexos.
Neste sentido, ao longo do projeto, houve uma evolução nas estruturas de dados escolhidas para cada situação. Numa abordagem inicial, de experimentação, foram implementadas estruturas básicas, como *arrays*. Posteriormente, optei por implementar listas *List<T>*, o que tornou o programa mais eficiente. Ainda assim, o acesso aos dados nesta lista, apesar de eficiente no sentido do espaço de armazenamento utilizado, não era o mais rápido.
Assim sendo, nesta abordagem final, algumas dessas listas mais críticas foram substituidas por *Dictionary(TKey, TValue)*. Estas estruturas permitem um acesso muito mais rápido aos dados visto que, dada a chave, o acesso ao valor é praticamente imediato.

## 12.1 Vantagens de *Dictionary(TKey, TValue)*

- **Acesso rápido e eficiente** a valores associados a **chaves únicas**.
- **Flexibilidade na escolha das chaves**.
- **Facilidade de atualização**. Alterar ou adicionar valores num Dictionary é muito simples. Se a chave já existir, o valor é reescrito; caso contrário, a entrada é adicionada.
- **Verificação eficiente de existência**. Métodos como *ContainsKey* e *TryGetValue* permitem verificar de forma eficiente se uma chave existe no dicionário, evitando exceções ou operações desnecessárias.
- **Iteração conveniente**.É possível iterar sobre todas as chaves, todos os valores ou os pares chave-valor do dicionário de forma conveniente.
- **Ideal para mapeamentos e associações**. Excelente em casos onde é necessário relacionar identificadores *keys* e informações detalhadas *values*.
  Um exemplo disso é $Referência \rightarrow Produto$

# 13 Conclusão

O trabalho em questão foi desenvolvido com sucesso, respeitando a maioria dos requisitos estabelecidos. Apesar de, num modo geral, o programa se encontrar num estado embrionário, sem aplicação real imediata, é um bom exercício de aplicação de princípios e padrões que visam a melhoria da qualidade do código e da aplicação em geral, nomeadamente no que toca a manutenção e escalabilidade do projeto.

Neste projeto, pude aproximar-me de vários dos padrões vigentes na área, bem como aplicar a maioria dos conceitos aprendidos em sala de aula e fora dela, pelo que penso que os objetivos foram cumpridos, apesar de ter ainda um longo caminho de aprendizagem e melhoria pela frente.

Devo realçar que este projeto foi também bom a nível pessoal, visto que acho que finalmente consegui encontrar vontade e prazer na programação e desenvolvimento de soluções de *software*, o que espero vir a tornar-se uma mais-valia nos projetos seguintes, permitindo-me crescer enquanto programador.