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

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*).

Listing 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.

Listing 2: Exemplo de Utilização destes padrões

```
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.

Listing 3: A Classe Cliente

```
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 TRABALHO DESENVOLVIDO:

7.1 Diagrama de Classes

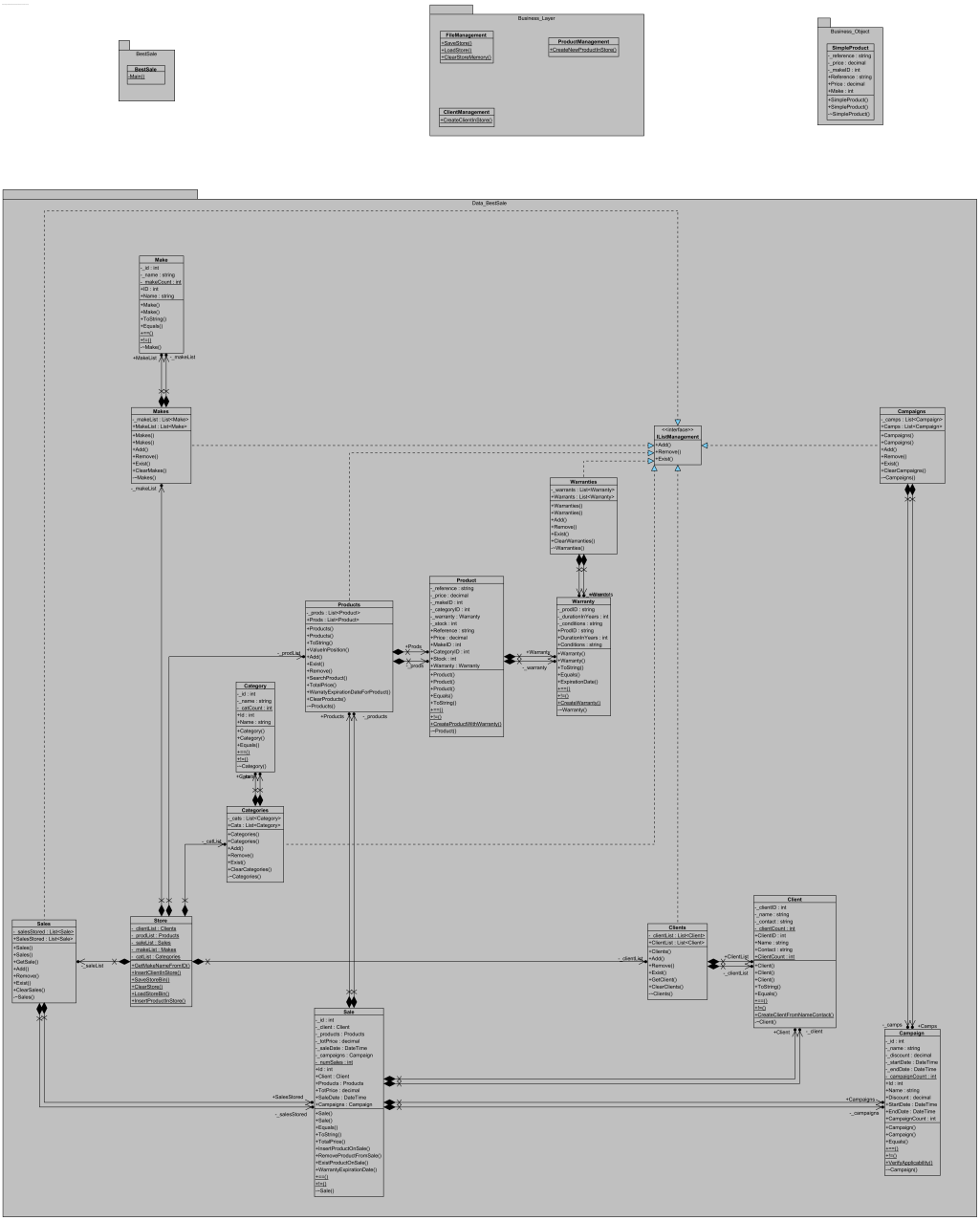

Figure 7.1: Diagrama de Classes

7.2 Estrutura de Projetos:

7.2.1 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.

7.2.2 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*.

7.2.3 Front End:

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

7.3 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 *List<T>*. 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:

Listing 4: 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
20    /// Method used to verify if a product is on a products' list, given
          its Refference.
21    public bool Exist(object obj)
22    {
23      if (obj == null) return false;
24      if (obj is string)
25      {
26        foreach (Product p in _prods)
27        {
28          if (p.Reference == (string)obj) return true;
29        }
30      }
31      return false;
32    }
33
34    /// Method used to remove a product from a Products' list.
35    public bool Remove(object obj)
36    {
```

```
37        if (obj == null) return false;
38        Product p = (Product)obj;
39        if (Exist(p.Reference))
40        {
41          _prods.Remove(p);
42          return true; ///Product removed successfully
43        }
44        return false; ///Product was not removed.
45      }
46    }
```

Listing 5: Outra Implementação do Interface

```
1
2   public class Makes : IListManagement
3   {
4     ///Method used to add a make to a list of makes.
5     public bool Add(object obj)
6     {
7       if (obj == null) return false;
8       if (obj is Make) {
9         _makeList.Add((Make)obj);
10        return true;
11      }
12      return false;
13    }
14
15    /// Method used to remove a make from a list of makes.
16    public bool Remove(object obj)
17    {
18      if (obj == null) return false;
19      var aux = obj as Make;
20      if (Exist(aux.ID))
21      {
22        _makeList.Remove((Make)obj);
23        return true;
24      }
25      return false;
26    }
27
28    /// Method used to verify if a make exists on a list of makes, given
          its ID or name.
29    public bool Exist(object obj)
30    {
31      if (obj == null) return false;
32      if (obj is int)
33      {
34        foreach (Make make in _makeList)
35        {
36          if (make.ID == (int)obj)
37          {
38            return true;
39          }
40        }
41      }
42      if(obj is string)
43      {
44        foreach (Make make in _makeList)
45        {
46          if (make.Name == (string)obj)
47          {
48            return true;
```

```
49            }
50          }
51        }
52        return false;
53      }
54    }
```

Mais ainda, apesar de, para tornar o código mais modular e melhor organizado (e indo um pouco contra os princípios do *SOLID*), 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. 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:

Listing 6: 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  }
```

7.4 Bullet Point List

- First item in a list
  - First item in a list
    - First item in a list
    - Second item in a list
  - Second item in a list
- Second item in a list

7.5 Numbered List

1. First item in a list
2. Second item in a list
3. Third item in a list

8 Interpreting a Table

8.1 The table above shows the nutritional consistencies of two sausage types. Explain their relative differences given what you know about daily adult nutritional recommendations.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent porttitor arcu luctus, imperdiet urna iaculis, mattis eros. Pellentesque iaculis odio vel nisl ullamcorper, nec faucibus

Per 50g	**Pork**	**Soy**
Energy	760kJ	538kJ
Protein	7.0g	9.3g
Carbohydrate	0.0g	4.9g
Fat	16.8g	9.1g
Sodium	0.4g	0.4g
Fibre	0.0g	1.4g

Table 8.1: Sausage nutrition.

ipsum molestie. Sed dictum nisl non aliquet porttitor. Etiam vulputate arcu dignissim, finibus sem et, viverra nisl. Aenean luctus congue massa, ut laoreet metus ornare in. Nunc fermentum nisi imperdiet lectus tincidunt vestibulum at ac elit. Nulla mattis nisl eu malesuada suscipit.

9 Reading a Code Listing

9.1 How many luftballons will be output by the Listing ?? above?

Aliquam arcu turpis, ultrices sed luctus ac, vehicula id metus. Morbi eu feugiat velit, et tempus augue. Proin ac mattis tortor. Donec tincidunt, ante rhoncus luctus semper, arcu lorem lobortis justo, nec convallis ante quam quis lectus. Aenean tincidunt sodales massa, et hendrerit tellus mattis ac. Sed non pretium nibh. Donec cursus maximus luctus. Vivamus lobortis eros et massa porta porttitor.

9.2 Identify the regular expression in Listing ?? and explain how it relates to the anti-war sentiments found in the rest of the script.

Fusce varius orci ac magna dapibus porttitor. In tempor leo a neque bibendum sollicitudin. Nulla pretium fermentum nisi, eget sodales magna facilisis eu. Praesent aliquet nulla ut bibendum lacinia. Donec vel mauris vulputate, commodo ligula ut, egestas orci. Suspendisse commodo odio sed hendrerit lobortis. Donec finibus eros erat, vel ornare enim mattis et.