IPCA, INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE

Trabalho de POO

Diogo Machado
nº26042

10 de Dezembro de 2024

Fase 2

Trabalho Prático

Programação Orientada a Objetos

Docente: Luís Ferreira

INDICE

1 Introdução **4**

2 Objetivos **4**

3 Estrutura de dados utilizada **4**
3.1 Dicionarios 4

4 Prática de boa programação **5**
4.1 Convenção de nomenclatura 5
4.2 Principios SOLID 5
4.3 Tratamento de erros 5
4.4 Utilização de Design Patterns 6
4.5 Documentação e comentários 7
4.6 Utilização do LINQ 8
4.7 Testes 9

5 Trabalho desenvolvido **11**
5.1 Diagrama de Classes 11
5.2 Estrutura da Solução 11
5.2.1 Bibliotecas (DLL) 11
5.2.2 Utilities 12
5.2.3 Business Objects 13
5.2.4 Business Layer 13
5.3 Exemplos do trabalho desenvolvido 13

6 Conclusão **16**

AMOSTRAS DE CÓDIGO

1 Exemplo de nomenclatura correta 5
2 Exemplo de tratamento de exceções 6
3 Exemplo de Documentação 8
4 Exemplo de utilização do LINQ 9
5 Exemplo da utilização de testes 10
6 Demonstração da Interface criada 13
7 Implementação dos métodos da interface 14
8 Implementação do método de criar objeto de negócio 15

1 INTRODUÇÃO

Este documento é o relatório do trabalho da disciplina de Programação Orientada a Objetos e tem como objetivo que sejam desenvolvidas soluções em C# para problemas reais de complexidade moderada. Como tema do trabalho eu decidi desenvolver um sistema de gestão de atividades de socorro de modo a poder registar ocorrências de pedidos de ajuda e gerir equipamentos e pessoas para prestar o auxílio necessário. Com esse objetivo foi criada a solução que será apresentada neste documento.
keywords: Proteção Civil, equipamentos, INEM, enfermeiros, médicos, bombeiros;

2 OBJETIVOS

Para este trabalho, foram definidos objetivos claros que visam guiar o processo de aprendizagem e consolidação de competências essenciais no âmbito da programação e do desenvolvimento de software. Estes objetivos foram delineados de forma a proporcionar uma abordagem prática e aplicada, promovendo não só a compreensão dos conceitos fundamentais mas também a capacidade de os aplicar em contextos reais. Assim, pretende-se alcançar os seguintes propósitos:

- 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.

3 ESTRUTURA DE DADOS UTILIZADA

3.1 Dicionarios

Os dicionários em C# foram escolhidos como a estrutura de dados para o desenvolvimento deste trabalho devido à sua eficiência e flexibilidade no armazenamento e recuperação de dados baseados em pares *key-value*. Uma das maiores vantagens dos dicionários é a sua capacidade de fornecer um acesso extremamente rápido aos elementos, uma vez que internamente utilizam *hash tables*, que permitem localizar um valor diretamente a partir da sua *key*. Os dicionários eliminam a necessidade de percorrer todos os objetos para encontrar um valor, como acontece noutras estruturas de dados lineares, tornando-os ideais para cenários onde a consulta rápida é essencial. Além disso, a dimensão de um dicionário não precisa de ser previamente definida, pois este é dinâmico, redimensionando-se automaticamente à medida que novos pares *key-value* são adicionados.
Para além disso, os dicionários disponibilizam uma vasta gama de métodos predefinidos que facilitam a sua manipulação. Métodos como *Add*, *Remove*, *ContainsKey* e *TryGetValue* permitem adicionar, remover, verificar a existência de uma chave ou obter valores de forma segura e eficaz.
Esta combinação de eficiência, flexibilidade e facilidade de utilização faz dos dicionários uma das escolhas mais adequadas para problemas que envolvem a associação entre *key* e *values* em aplicações de C#.

4 PRÁTICA DE BOA PROGRAMAÇÃO

Práticas de boa programação em C# são fundamentais para garantir que o código seja eficiente, seguro, legível e fácil de manter. Estas práticas ajudam a reduzir erros, melhorar a colaboração em equipa e preparar o código para alterações futuras.

4.1 Convenção de nomenclatura

Para uma convenção de nomenclatura correta que seja de facil interpretação, ter atenção aos seguintes pontos:

- Utilizar *PascalCase* para classes, métodos e propriedades;
- Utilizar *camelCase* para variáveis e parâmetros;
- Utilizar o I como prefixo de interfaces, por exemplo **IMetodos**;

Amostra de Código 1: Exemplo de nomenclatura correta

```
1   public class Pessoa
2 {
3   #region Attributes
4   int id;                         //ID da pessoa
5   string nome;                    //Nome da pessoa
6   string contacto;                //Numero de contacto da pessoa
7   string email;                   //Email da pessoa
8   static int idCounter = 0;       //Variavel estatica que serve como
        contador dos IDs das pessoas
9   #endregion
10 }
```

4.2 Principios SOLID

Para uma fácil criação, manutenção e reutilização do código devem ser seguidos os principios ***SOLID*** que é um acrônimo que representa cinco princípios fundamentais da programação orientada a objetos (POO).

- **Single Responsibility Principle (SRP)**: Uma classe só deve ter um motivo para ser alterada, o que significa que cada classe só deve ter uma responsabilidade.
- **Open Closed Principle (OCP)**: o código deve estar aberto para extensões mas fechado para modificações.
- **Liskov Substitution Principle (LSP)**: Objetos de uma classe base devem poder ser substituídos por objetos das suas classes derivadas sem alterar o comportamento do sistema.
- **Interface Segregation Principle (ISP)**: As interfaces devem ser pequenas e específicas para facilitar a sua implementação
- **Dependency Inversion Principle (DIP)**: Depende de abstrações e não de implementações.

4.3 Tratamento de erros

O **tratamento de erros** é um aspeto essencial no desenvolvimento de software, pois garante que a aplicação possa lidar com situações inesperadas de forma robusta e confiável. Em C#, o tratamento de erros é baseado no uso de exceções, que representam condições anormais ou erros que ocorrem durante a execução de um programa. Uma abordagem adequada para lidar com essas exceções não só melhora a experiência do utilizador, como também previne falhas graves e facilita a manutenção do código.

As exceções em C# são objetos que herdam da classe base *System.Exception*. Quando ocorre um erro durante a execução, uma exceção é "lançada" (usando a *keyword throw*) e pode ser "capturada" (usando *try-catch*). Isso permite que o programador trate o erro de forma controlada, sem interromper o programa. Exemplos comuns incluem operações inválidas, acesso a recursos inexistentes ou falhas em conexões com bases de dados.

Amostra de Código 2: Exemplo de tratamento de exceções

```
1  /// <summary>
2  /// Funcao que cria um membro do INEM completo
3  /// </summary>
4  /// <param name="nomeMembro">Nome do membro</param>
5  /// <param name="contacto">Contacto do membro</param>
6  /// <param name="email">Email do membro</param>
7  /// <param name="especialidade">Especialidade do membro</param>
8  /// <returns>Se criou o membro completo com sucesso ou nao</returns>
9  public bool CriaMembroINEMCompleto(string nomeMembro, string contacto,
       string email, EspecialidadeINEM especialidade)
10 {
11   try
12   {
13     if (Validar.VerificaEmail(email) && Validar.VerificaContacto(
           contacto))
14     {
15       return instituicao.CriaMembroINEMInstCompleto(nomeMembro, contacto
             , email, especialidade);
16     }
17   }
18   catch (ValidationException e)
19   {
20     throw e;
21   }
22   catch (Exception e)
23   {
24     throw e;
25   }
26   return false
27 }
```

4.4 Utilização de Design Patterns

Os ***design patterns*** são soluções reutilizáveis para problemas recorrentes que surgem durante o desenvolvimento de software. Eles representam abordagens testadas e documentadas que facilitam a criação de sistemas robustos, flexíveis e fáceis de manter. Os padrões de *design* não são blocos de código prontos para utilização, mas sim orientações ou estruturas que podem ser adaptadas ao contexto específico do projeto.
Existem dois tipos de *design patterns* que são os mais utilizados:

- **Model-View-Controller (MVC)**: contém uma separação clara entre a lógica da aplicação, a apresentação dos dados e a interação do utilizador. Essa divisão facilita a organização do código e melhora a manutenção.
- **Arquitetura multicamadas(N-Tier)**: organiza a aplicação em várias camadas lógicas independentes. Cada camada tem uma responsabilidade específica, o que promove a separação de preocupações e a modularidade.

Embora ambos promovam a separação de responsabilidades, o **MVC** é um *design patterns* que organiza componentes dentro de uma camada (geralmente a camada de apresentação), en-

quanto o N-Tier é uma arquitetura que organiza a aplicação inteira em camadas lógicas independentes.
Ao aplicar estas arquiteturas, o trabalho será mais modular, facilitando a introdução de novas funcionalidades, a resolução de problemas e a adaptação a mudanças nos requisitos. Além disso, elas promovem boas práticas de desenvolvimento, contribuindo para a qualidade geral do sistema.
Neste trabalho foi utilizado a arquitetura N-Tier com 4 camadas:

- **Dados**: Camada que contém toda a estrutura de dados do sistema.
- **Business Layer**: Camada que contém todas as regras e validações de segurança necessárias para o acesso aos dados.
- **Business Objects**: Camada que agrupa os conceitos e entidades do domínio do negócio, representando objetos que modelam informações e comportamentos associados às regras de negócio da aplicação.
- **Frontend**: Camada que interage com a camada de negócios para exibir informações.

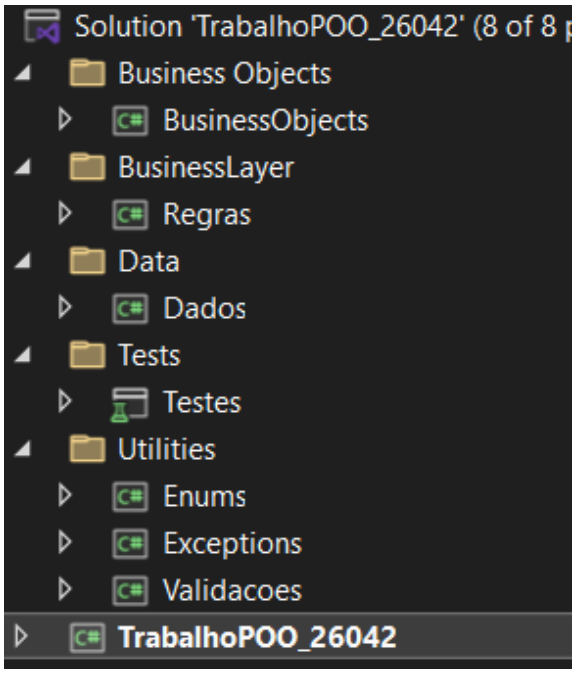

Figure 4.1: Design Pattern N-Tier

4.5 Documentação e comentários

A **documentação e os comentários** são componentes fundamentais no desenvolvimento de software, sendo essenciais para a manutenção, compreensão e continuidade de um projeto. Embora o foco principal de um programador seja escrever código funcional e eficiente, garantir que este código seja compreensível para outras pessoas (e até mesmo para o próprio autor no futuro) é igualmente importante. A ausência de uma boa documentação e de comentários claros pode transformar um código funcional em algo difícil de entender e praticamente impossível de manter.
A documentação é importante porque:

- **Facilita a compreensão do Sistema**: Documentar o código e o projeto permite que outros desenvolvedores, sejam eles novos membros da equipa ou colaboradores externos,

compreendam rapidamente a estrutura, as funcionalidades e os objetivos do sistema. Sem documentação, entender o propósito e o funcionamento de um código pode exigir um tempo desnecessariamente longo, especialmente em projetos complexos.

- **Reduz Custos de Manutenção**: A maior parte do ciclo de vida de um software envolve a sua manutenção, e um código bem documentado reduz significativamente o tempo necessário para localizar e corrigir problemas, adicionar novas funcionalidades ou fazer atualizações.
- **Promove a Continuidade do Projeto**: Em situações em que o autor original do código deixa a equipa, a documentação permite que outros programadores continuem o trabalho sem dificuldades. Ela age como um guia que esclarece decisões tomadas, padrões utilizados e os motivos por trás de certas implementações.
- **Evita Ambiguidade**: Mesmo um código bem escrito pode conter complexidades que não são imediatamente óbvias. A documentação serve para esclarecer essas partes, explicando, por exemplo, como um algoritmo específico funciona ou por que uma abordagem foi escolhida em detrimento de outra.

Amostra de Código 3: Exemplo de Documentação

```
/// <summary>
/// Funcao que cria um membro do INEM completo
/// </summary>
/// <param name="nomeMembro">Nome do membro</param>
/// <param name="contacto">Contacto do membro</param>
/// <param name="email">Email do membro</param>
/// <param name="especialidade">Especialidade do membro</param>
/// <returns>Se criou o membro completo com sucesso ou nao</returns>
public bool CriaMembroINEMCompleto(string nomeMembro, string contacto,
     string email, EspecialidadeINEM especialidade)
```

4.6 Utilização do LINQ

O **LINQ** (Language Integrated Query) é um conjunto de extensões para as linguagens .NET que introduz uma sintaxe de consulta semelhante ao SQL diretamente no código. Ele permite realizar operações sobre coleções de objetos (como listas, arrays, e dicionários), bases de dados, XML, e até mesmo fontes externas de dados. Isso elimina a necessidade de trabalhar com APIs diferentes para cada tipo de dado, unificando a experiência de desenvolvimento.
As principais caracteristicas são:

- **Sintaxe Uniforme**: LINQ oferece uma sintaxe comum para consultar diferentes tipos de dados, como objetos, bases de dados relacionais e documentos XML.
- **Fortemente tipado**:As consultas LINQ são verificadas em tempo de compilação, reduzindo erros e proporcionando maior segurança no código.
- **Composição de consultas**: As consultas podem ser criadas dinamicamente, permitindo compor e reutilizar partes de forma eficiente.
- **Legibilidade**: A sintaxe intuitiva do LINQ torna o código mais fácil de ler e entender, promovendo boas práticas de programação.

Com isto o **LINQ** oferece bastantes benefícios ao nosso código tais como:

- **Redução de Código Boilerplate**: Operações que tradicionalmente exigiriam loops e condições explícitas podem ser realizadas com poucas linhas de código.
- **Integração com o .NET**: LINQ está totalmente integrado ao .NET Framework, utilizando recursos como tipagem forte, IntelliSense e verificações em tempo de compilação.

- **Fácil manutenção**: A legibilidade do código LINQ facilita futuras alterações ou depuração.
- **Uniformidade**: Permite aplicar o mesmo padrão de consultas a diferentes fontes de dados.

No desenvolvimento de um projeto, o LINQ facilita significativamente a manipulação de dados, tornando as consultas mais rápidas de escrever e fáceis de entender. Ele é especialmente útil quando se trabalha com grandes coleções de dados ou quando se deseja unificar a lógica de manipulação entre diferentes fontes. Além disso, promove um código mais limpo e alinhado com as boas práticas de programação, sendo uma ferramenta indispensável em projetos que visam eficiência e organização.

Amostra de Código 4: Exemplo de utilização do LINQ

```
        /// <summary>
/// Funcao que transforma os equipamentos em texto
/// </summary>
/// <returns>Texto que se pretende apresentar</returns>
public override string ToString()
{
  if (equipamentos.Count == 0)
  {
    return "Nao tem equipamentos nesta lista";
  }
  string infoEquip = string.Join("\n", equipamentos.Select(equip =>
      equip.ToString()));
  return infoEquip;
}
```

4.7 Testes

Os **testes** desempenham um papel fundamental no desenvolvimento de software, garantindo a qualidade, confiabilidade e manutenção do código. Eles são essenciais para identificar e corrigir erros, validar funcionalidades e assegurar que o sistema atenda aos requisitos especificados. Além disso, ajudam a evitar regressões e a promover boas práticas de programação.

- **Garantia de Qualidade**: Os testes são o principal mecanismo para assegurar que o software funcione conforme esperado. Eles permitem que os desenvolvedores validem se as funcionalidades implementadas cumprem os requisitos e detectem problemas antes que o sistema seja entregue ao cliente ou implantado em produção.
- **Prevenção de regressões**: Conforme o software evolui, com a adição de novas funcionalidades ou correções de bugs, há sempre o risco de introduzir erros em partes já desenvolvidas. Os testes, especialmente os automatizados, ajudam a identificar essas regressões, permitindo que os desenvolvedores façam alterações com maior confiança.
- **Redução de Custos a Longo Prazo**: Embora implementar testes possa parecer um esforço adicional, eles economizam tempo e recursos no longo prazo. Identificar e corrigir um erro durante o desenvolvimento é significativamente menos custoso do que fazer isso após o software ser implementado, onde os custos de manutenção e o impacto nos usuários são muito maiores.
- **Facilidade da Manutenção**: Sistemas bem testados são mais fáceis de manter. Os testes atuam como uma documentação funcional do sistema, fornecendo exemplos claros de como o código deve se comportar em diferentes cenários. Isso é particularmente útil em projetos de longo prazo ou em equipas com alta rotatividade, onde novos desenvolvedores precisam entender rapidamente o funcionamento do sistema.

- **Benefícios na Experiência do Utilizador**:Erros em software podem impactar negativamente a experiência do utilizador, causando insatisfação, perda de confiança e até prejuízos financeiros. Os testes ajudam a evitar problemas críticos, garantindo que o sistema seja estável e ofereça uma experiência fluida e confiável.

Os testes não são apenas uma etapa do desenvolvimento, mas uma prática essencial para garantir a qualidade e sustentabilidade de qualquer projeto de software. Eles promovem confiança, reduzem custos e riscos, e permitem que as equipes entreguem soluções robustas e alinhadas às expectativas dos clientes. Investir em testes desde o início do desenvolvimento não é apenas uma boa prática, mas uma abordagem indispensável para o sucesso a longo prazo.

Amostra de Código 5: Exemplo da utilização de testes

```
namespace Testes
{
  /// <summary>
  /// Class que contem os testes
  /// </summary>
  [TestClass]
  public sealed class Testes
  {
    /// <summary>
    /// Teste ao construtor da classe Bombeiro
    /// </summary>
    [TestMethod]
    public void Constructor_ParametrosValidos_Bombeiro()
    {
      ///Arrange
      string nome = "Joao";
      string contacto = "912345678";
      string email = "joni@gmail.com";
      PatenteBombeiro patente = PatenteBombeiro.OFICIAL;

      ///Act
      Bombeiro bombeiro = new Bombeiro(nome, contacto, email, patente);


      ///Assert
      Assert.AreEqual(nome, bombeiro.Nome);
      Assert.AreEqual(contacto, bombeiro.Contacto);
      Assert.AreEqual(email, bombeiro.Email);
      Assert.AreEqual(patente, bombeiro.Patente);
      Assert.AreNotEqual(0, bombeiro.Id);
    }

    [TestMethod]
    public void
        Constructor_ParametrosInvalidos_EsperaExcecaoContactoInvalido()
    {
      ///Arrange
      string nome = "Joao";
      string contacto = "9123456758";
      string email = "joni@gmail.com";
      PatenteBombeiro patente = PatenteBombeiro.OFICIAL;

      ///Act & Assert
      try
      {
        Bombeiro bombeiro = new Bombeiro(nome, contacto, email, patente)
            ;
        Assert.Fail("Exception esperada nao foi lancada");
```

```
47          }
48          catch (ValidationException e)
49          {
50              Assert.AreEqual(Erro.CONTACTO_INVALIDO, e.erro);
51          }
52      }
53  }
54 }
```

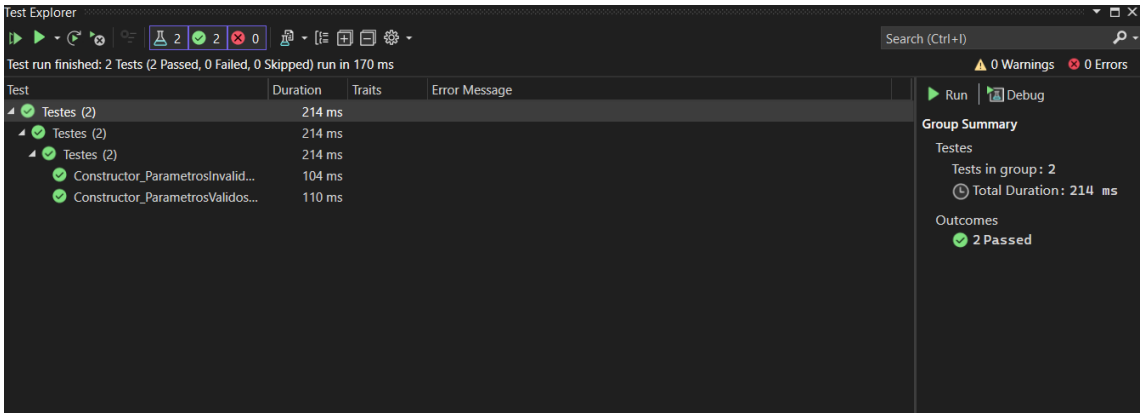

Figure 4.2: Execução dos testes

5 Trabalho Desenvolvido

5.1 Diagrama de Classes

Neste diagrama de classes conseguimos perceber como está estruturado o trabalho desenvolvido. Conseguimos concluir que o programa contém uma camada de dados com todas as estruturas que representam os dados da instituição de proteção civil. De seguida temos a camada de regras que é a camada que o utilizador necessita acessar para conseguir manipular os dados, e consegue isso através da *main* do programa. Com isso ainda contemos ficheiros com funcoes de validacões de alguns dados, como *email* e contacto, ficheiro com enumerados, com alguns enumerados como patente do bombeiro, especialidade do membro do INEM e código de erros, e também um ficheiro com as *exceptions* que serão lançadas no caso de alguns dados estarem inválidos na criação de um objeto. Por fim contém o ficheiro com alguns testes do programa.

5.2 Estrutura da Solução

5.2.1 Bibliotecas (DLL)

- **Dados.dll**
  - ***Pessoa.cs***: Definição de atributos e propriedades de pessoa.
  - ***Bombeiro.cs***: Definição de atributos e propriedades de bombeiro. Herda de pessoa.
  - ***Bombeiros.cs***: Classe de agregação de ***Bombeiro.cs***, que contém os métodos para gestão do dicionário.
  - ***MembroINEM.cs***: Definição de atributos e propriedades de Membro do INEM. Herda de pessoa.
  - ***EquipaINEM.cs***: Classe de agregação de ***MembroINEM.cs***, que contém os métodos para gestão do dicionário.

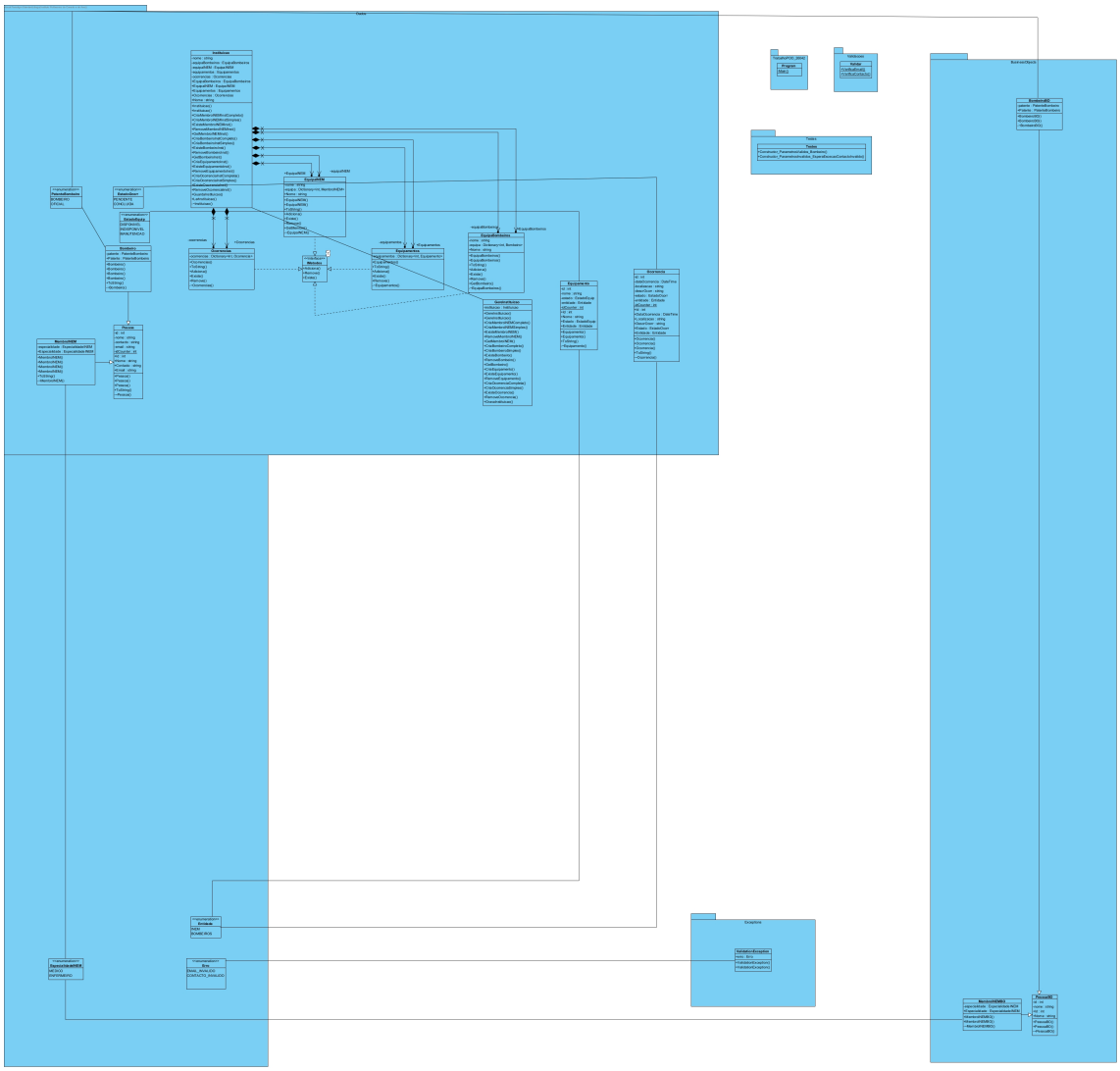

Figure 5.1: Diagrama de Classes

- ***Equipamento.cs***: Definição de atributos e propriedades de equipamento.
- ***Equipamentos.cs***: Classe de agregação de ***Equipamento.cs***, que contém os métodos para a gestão do dicionário.
- ***Ocorrência.cs***: Definição de atributos e propriedades de ocorrência.
- ***Ocorrências.cs***: Classe de agregação de ***Ocorrência.cs***, que contém os métodos para gestão do dicionário.
- ***IMetodos.cs***: Interface com os métodos utilizados para manipular os dicionários.
- ***Instituicao.cs***: Classe de agregação com os dicionários com todos os objetos que dizem respeito a uma instituição.

5.2.2 Utilities

- *Validacoes.dll*
  - ***Validacoes.cs***: Definição dos métodos que dizem respeito à validação de dados que serão inseridos na classe pessoa, nomeadamente o email e contacto.
- *Exceptions.dll*
  - ***ValidationException.cs***: Definição das exceções que serão lançadas no caso dos

dados que se validam em ***Validacoes.cs*** serem inválidos.

- *Enums.dll*
  - ***Enumerados.cs***: Definição dos enumerados que serão utilizados para alguns atributos das classes do ***Dados.dll*** e para controlo de erros no ***Exceptions.dll***.

5.2.3 Business Objects

- *BusinessObjects.dll*
  - ***PessoaBO.cs***: Definição de atributos de uma pessoa e propriedades que dizem respeito às regras de negócio.
  - ***BombeiroBO.cs***: Definição de atributos de um bombeiro e propriedades que dizem respeito às regras de negócio. Herda de PessoaBO.
  - ***MembroINEMBO.cs***: Definição de atributos de um membro do INEM e propriedades que dizem respeito às regras de negócio. Herda de PessoaBO.

5.2.4 Business Layer

- *Regras.dll*
  - ***GereInstituicao.cs***: Definição dos métodos da *Business Layer* de modo que o utilizador consiga obter informações dos ***Dados.dll*** sem ter acesso direto.

5.3 Exemplos do trabalho desenvolvido

Para os métodos de manipulação de dicionários dos dados foi criada uma interface com os metodos que seriam necessários:

Amostra de Código 6: Demonstração da Interface criada

```
1  namespace Dados
2  {
3    /// <summary>
4    /// Interface que contem as funcoes que serao utilizadas em todas as
         classes com listas
5    /// </summary>
6    /// <typeparam name="T">O tipo de objetos que os metodos irao operar</
         typeparam>
7    internal interface IMetodos<T>
8    {
9      #region Methods
10     /// <summary>
11     /// Cabecalho da funcao que adiciona um objeto numa lista
12     /// </summary>
13     /// <param name="item">Objeto que se pretende adicionar</param>
14     /// <returns>Se adicionou com sucesso ou nao</returns>
15     bool Adiciona(T item);
16
17     /// <summary>
18     /// Cabecalho da funcao que remove um determinado objeto de uma
           lista
19     /// </summary>
20     /// <param name="id">ID do objeto que se pretende remover</param>
21     /// <returns>Se removeu com sucesso ou nao</returns>
22     bool Remove(int id);
23
24     /// <summary>
25     /// Cabecalho da funcao que verifica se um determinado objeto existe
            na lista
```

```
26      /// </summary>
27      /// <param name="id">ID do obejto que se pretende verificar se
            existe</param>
28      /// <returns>Se existe ou nao</returns>
29      bool Existe(int id);
30      #endregion
31    }
32  }
```

Após isso foram implementados os respetivos métodos nas classes de agregação, tal como se pode observar neste exemplo:

Amostra de Código 7: Implementação dos métodos da interface

```
1   namespace Dados
2   {
3     [Serializable]
4     /// <summary>
5     /// Purpose: Class EquipaBombeiros que contem um dicionario de
          bombeiros e o nome da  equipa
6     /// Created by: diogo
7     /// Created on: 11/13/2024 7:52:17 PM
8     /// </summary>
9     /// <remarks></remarks>
10    /// <example></example>
11    public class EquipaBombeiros : IMetodos<Bombeiro>
12    {
13
14
15      /// <summary>
16      /// Funcao que adiciona bombeiro no dicionario da equipa de
            bombeiros
17      /// </summary>
18      /// <param name="bombeiro">Bombeiro que se pretende adicionar</param
            >
19      /// <returns>Se adicionou com sucesso ou nao</returns>
20      public bool Adiciona(Bombeiro bombeiro)
21      {
22        if(bombeiro == null || Existe(bombeiro.Id))
23        {
24          return false;
25        }
26        equipa.Add(bombeiro.Id, bombeiro);
27        return true;
28      }
29
30      /// <summary>
31      /// Funcao que verifica se existe um determinado bombeiro na equipa
            de bombeiros
32      /// </summary>
33      /// <param name="id">ID do bombeiro que se pretende verificar se
            existe</param>
34      /// <returns>Se existe ou nao</returns>
35      public bool Existe(int id)
36      {
37        return equipa.ContainsKey(id);
38      }
39
40
41      /// <summary>
42      /// Funcao que remove um determinado bombeiro da equipa de bombeiros
43      /// </summary>
```

```
44      /// <param name="id">ID do bombeiro que se pretende remover da
            equipa</param>
45      /// <returns>Se removeu com sucesso ou nao</returns>
46      public bool Remove(int id)
47      {
48        if(Existe(id))
49        {
50          equipa.Remove(id);
51          return true;
52        }
53        return false;
54      }
55    }
56  }
```

Para além disso, no caso dos Bombeiros e membros do INEM foi criada uma função que cria o seu respetivo objeto de negócio, de modo a que possam ser enviadas ao utilizador as informações de que ele necessita:

Amostra de Código 8: Implementação do método de criar objeto de negócio

```
1   namespace Dados
2   {
3     [Serializable]
4     /// <summary>
5     /// Purpose: Class EquipaBombeiros que contem um dicionario de
          bombeiros e o nome da  equipa
6     /// Created by: diogo
7     /// Created on: 11/13/2024 7:52:17 PM
8     /// </summary>
9     /// <remarks></remarks>
10    /// <example></example>
11    public class EquipaBombeiros : IMetodos<Bombeiro>
12    {
13
14      /// <summary>
15      /// Funcao que devolve as informacoes de um bombeiro da equipa de
            bombeiros
16      /// </summary>
17      /// <param name="idBombeiro">Id do bombeiro</param>
18      /// <returns>Informacoes do bombeiro</returns>
19      public BombeiroBO GetBombeiro(int idBombeiro)
20      {
21        equipa.TryGetValue(idBombeiro, out Bombeiro bombeiro);
22        BombeiroBO bombeiroBO = new BombeiroBO(bombeiro.Id, bombeiro.
              Nome, bombeiro.Patente);
23        return bombeiroBO;
24      }
25    }
26  }
```

6 CONCLUSÃO

Neste relatório, foi desenvolvido um sistema para gestão de atividades de socorro, com o objetivo de consolidar conhecimentos no paradigma de programação orientada a objetos e aplicar boas práticas de desenvolvimento de software. Ao longo do trabalho, foram explorados conceitos fundamentais como a utilização de dicionários, princípios SOLID, *design patterns* e validações com LINQ, bem como a importância da documentação e dos testes para a robustez e manutenção do sistema. A aplicação da arquitetura N-Tier foi essencial para a separação de responsabilidades e facilitou a manutenção e evolução do código. Além disso, os testes implementados foram determinantes para assegurar a confiabilidade do sistema.

Em suma, o projeto cumpriu os objetivos propostos, não só fortalecendo a compreensão teórica dos conceitos abordados, mas também permitindo a sua aplicação prática em um contexto próximo ao real. Este trabalho reforça a relevância das boas práticas na programação e a necessidade de continuar a aprimorar as minhas competências no desenvolvimento de soluções robustas e eficientes.