IPCA

Escola superior de Tecnologia

Projeto de Programação Orientada a Objetos

Autores:
Diogo Bernardo ***21144***
Pedro Perez ***18623***
João Ribeiro ***23795***
Henrique Senra ***21154***

Docentes:
Célio Carvalho

submissão do projeto para
o curso de Engenharia de Sistemas Informáticos

ESI

Escola Superior de Tecnologia
Bitbucket

December 23, 2022

IPCA

Resumo

Bitbucket

Engenharia de Sistemas Informáticos

Projeto de Programação Orientada a Objetos

por Diogo Bernardo **21144**
Pedro Perez **18623**
João Ribeiro **23795**
Henrique Senra **21154**

Trabalho em cargo da disciplina de Programação Orientada a Objetos.

Objetivos do projeto

Apresentamos em síntese os nossos objetivos na realização do projeto:

- Apresentar um programa de fácil utilização.
- Apresentar uma solução para o tema que nos foi proposto, Smart Campus.
- Sedimentar o nosso conhecimento a nível do C#.
- Projetar o nosso trabalho em grupo.
- Aprender a utilizar ferramentas que facilitam o trabalho em grupo.

Contents

Chapter 1

Classes

Neste exemplo, para representar Device,no C#, precisamos criar uma classe. Dentro do C# a declaração da classe é feita utilizando-se a palavra class seguida do nome da classe que queremos implementar:

```
internal class Device
{
    /// <summary>
    ///
    /// </summary>
    public IPAddress? iP { get; set; }

    /// <summary>
    ///
    /// </summary>
    public DateTime LogedDate { get; set; }

    /// <summary>
    ///
    /// </summary>
    public string? Os { get; set; }

    /// <summary>
    ///
    /// </summary>
    public string? HostName { get; set; }

    /// <summary>
    ///
    /// </summary>
    public string? Browser { get; set; }

    /// <summary>
    ///
    /// </summary>
    public double IpLocationLat { get; set; }

    /// <summary>
    ///
    /// </summary>
    public double IpLocationLon { get; set; }
}
```

Chapter 2

Construtores

Um construtor é um método cujo nome é igual ao nome de seu tipo, com o objetivo de criar e inicializar objetos sendo apartir do mesmo que se criem instâncias de uma classe.

Neste exemplo podemos ver em "new Address" o construtor a ser chamado.

```
public static Address GenExample1() => new Address("4720-000", "Portugal
    ",
"Braga", DateTime.Now,"Rua exemplo", 20,"Amares");
```

Chapter 3

Encapsulamento

O encapsulamento é o processo de ocultar ou esconder os membros de uma classe do acesso exterior usando modificadores de acesso. O encapsulamento fornece uma maneira de preservar a integridade do estado dos dados. Em vez de definir campos públicos devemos definir campos de dados privados.

```
internal class MbRef
{
    /// <summary>
    ///
    /// </summary>
    public DateTime ExpiryDate { get; set; }

    /// <summary>
    ///
    /// </summary>
    public int Value { get; private set; }

    /// <summary>
    ///
    /// </summary>
    public MbRef()
    {
        ExpiryDate = (DateTime.Now).AddDays(1);
        Value = GenerateMbReference();
    }

    /// <summary>
    ///
    /// </summary>
    private static int GenerateMbReference() =>
        // !TODO communicate with related services to get mb reference.
        (new Random()).Next(0, 999999999);
}
```

Chapter 4

Herança

A herança permite criar novas classes que podem ser reutilizadas e/ou estendidas. Classes cujos membros são herdados são chamadas de classes mãe, e classes que herdam esses membros são chamadas de classes filho.

Neste exemplo, podemos ver a classe Invoice a herdar da classe Payment.

```
internal sealed class Invoice : Payment
```

Neste exemplo, podemos ver a classe Client a herdar da classe Person e da interface ILogin.

```
internal sealed class Client : Person, ILogin
```

Chapter 5

Interfaces

Uma interface, no paradigma da orientação a objetos, é um tipo de classe que contém apenas as assinaturas de métodos, propriedades e eventos.
A implementação dos membros é feita por uma classe concreta ou struct que implementa a interface.
Neste exemplo, podemos ver um exemplo de uma interface, ILogin.

```
using static Data.DataBase;

namespace Host.Login;

/// <summary>
///
/// </summary>
internal interface ILogin
{
    #region data

    /// <summary>
    ///
    /// </summary>
    internal Task<string?> PassHashGetAsync
    {
        get => CmdExecuteQueryAsync<string>(
            "SELECT hashedpassword FROM logindata WHERE username='db4'")
    ;
    }

    #endregion

    #region methods

    private protected async Task<int> CreateNewUserAsync(string username
    ,
                                                         string
    passwordHash)
    {
        try
        {
            return await CmdExecuteNonQueryAsync(
                $"INSERT INTO logindata(username,hashedpassword) VALUES"
     +
                $"('{username}','{passwordHash}')");
        }
        catch
        {
            throw new Exception("User already exists");
        }
    }
```

```
    /// <summary>
    ///
    /// </summary>
    /// <returns></returns>
    private protected LoginStatus LogIn();

    /// <summary>
    ///
    /// </summary>
    /// <returns></returns>
    private protected LoginStatus LogOut();

    /// <summary>
    ///
    /// </summary>
    /// <returns></returns>
    internal static List<LoginAttempt>? GetLoginHistory() { return
    default; }

    #endregion
}
```

Chapter 6

Classe abstrata

A classe abstrata é um tipo de classe que somente pode ser herdada e não instanciada, de certa forma pode se dizer que este tipo de classe é uma classe conceitual que pode definir funcionalidades para que as suas subclasses possam implementa-las de forma não obrigatória.

Neste exemplo, temos Person como uma classe abstrata.

```
internal abstract class Person : Gym
{
    /// <summary>
    ///
    /// </summary>
    /// <param name="firstName"></param>
    /// <param name="lastName"></param>
    /// <param name="gender"></param>
    /// <param name="dateOfBirth"></param>
    /// <param name="nif"></param>
    /// <param name="address"></param>
    internal Person(string firstName, string lastName, Gender gender,
                    DateTime dateOfBirth, int nif, Address address,
                    string email, LoginData loginData)
    {
        FirstName = firstName;
        LastName = lastName;
        Gender = gender;
        DateOfBirth = dateOfBirth;
        Nif = nif;
        Email = email;

        LoginData = loginData;

        Addresses = new List<Address>();
        Addresses.Add(address);
    }
}
```

Chapter 7

Exceções

Uma exceção é uma forma de lidar com situações alternativas. Em C# é utilizado o try, catch e finally para lidar com as situações alternativas.

Neste exemplo, podemos ver uma exceção em ClientException.

```
internal class ClientException : UserException
{
    /// <summary>
    ///
    /// </summary>
    /// <param name="message"></param>
    internal ClientException(string message) : base(message)
    {
    }
}

/// <summary>
///
/// </summary>
internal class InvalidClientDataException : ClientException
{
    /// <summary>
    ///
    /// </summary>
    /// <param name="message"></param>
    internal InvalidClientDataException(string message) : base(message)
    {

    }
}
```

Chapter 8

LINQ e Lambda

8.0.1 LINQ

```
/// <summary>
    ///
    /// </summary>
    /// <param name="clientId"></param>
    /// <returns></returns>
    internal static async Task<List<CreditCard>?>
    GetClientCreditCards(Guid clientId)
    {
        try
        {
            var values = await GetCLientCreditCardsFromDb(clientId);
            var ccList = new List<CreditCard>();

            foreach (var (line, cc) in from line in values
                                        where values is not null
                                        let cc = new CreditCard()
                                        select (line, cc))
            {
                foreach (var val in from column in line
                                    where line is not null
                                    where column.Value is not null
                                    select column)
                {
                    switch (val.Key)
                    {
                        case 1:
                            cc.CcNum = (UInt64)val.Value;
                            break;
                        case 3:
                            cc.ExpiryDate = (DateTime)val.Value;
                            break;
                        case 4:
                            cc.InsertedDate = (DateTime)val.Value;
                            break;
                        case 5:
                            cc.SecurityCode = (string)val.Value;
                            break;
                        case 6:
                            cc.CcName = (string)val.Value;
                            break;
                    }
                }
                ccList.Add(cc);
            }

            return ccList;
        }
```

```
        catch (DataBaseException e)
        {
            Log.Error(e);
            return default;
        }
        catch (Exception e)
        {
            Log.Error(e);
            return default;
        }
    }
}
```

8.0.2 Lambda

O lambda é representado pelo token => que tem como objetivo colocar o código mais curto e limpo.

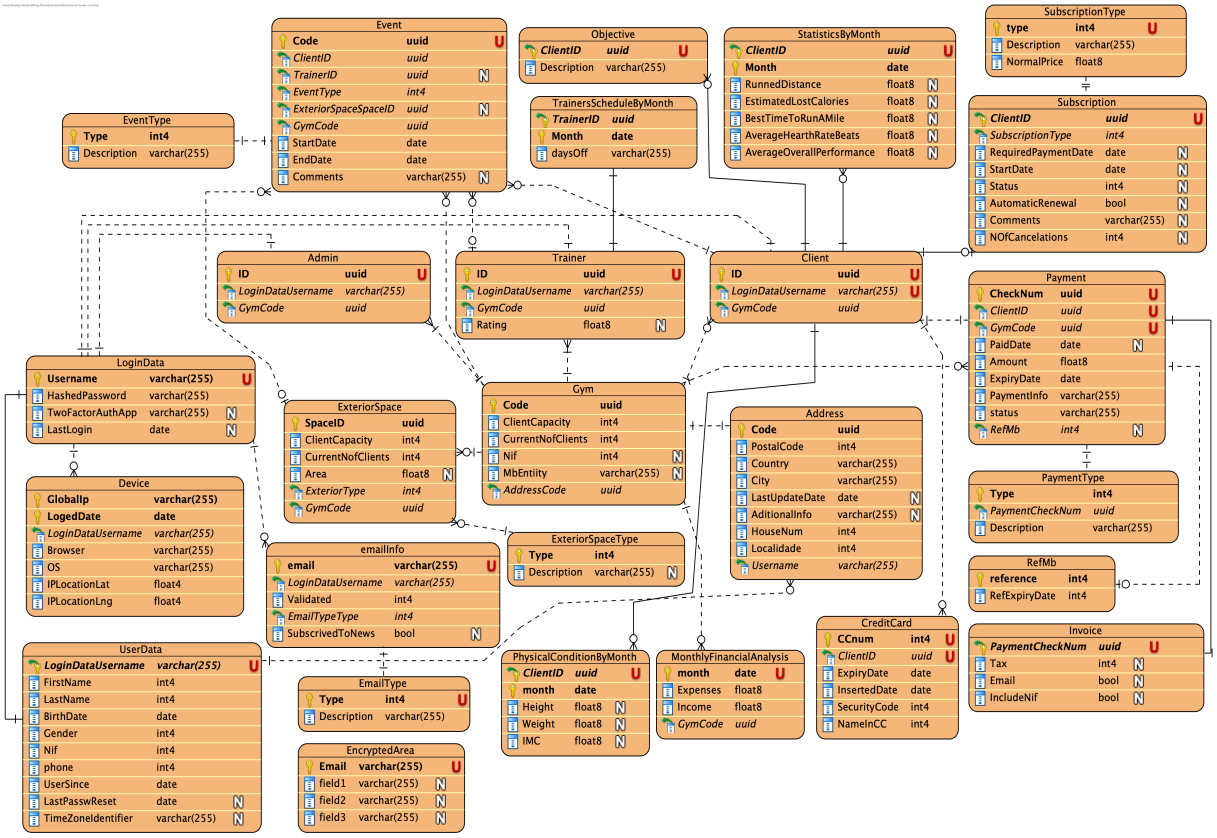
Aqui podemos ver um exemplo do => a ser usado no nosso projeto.

```
private static async Task<bool> SomeBankingServiceAsync() =>
        // SIMULATED CODE
        await Task<bool>.Run(() => true);
```

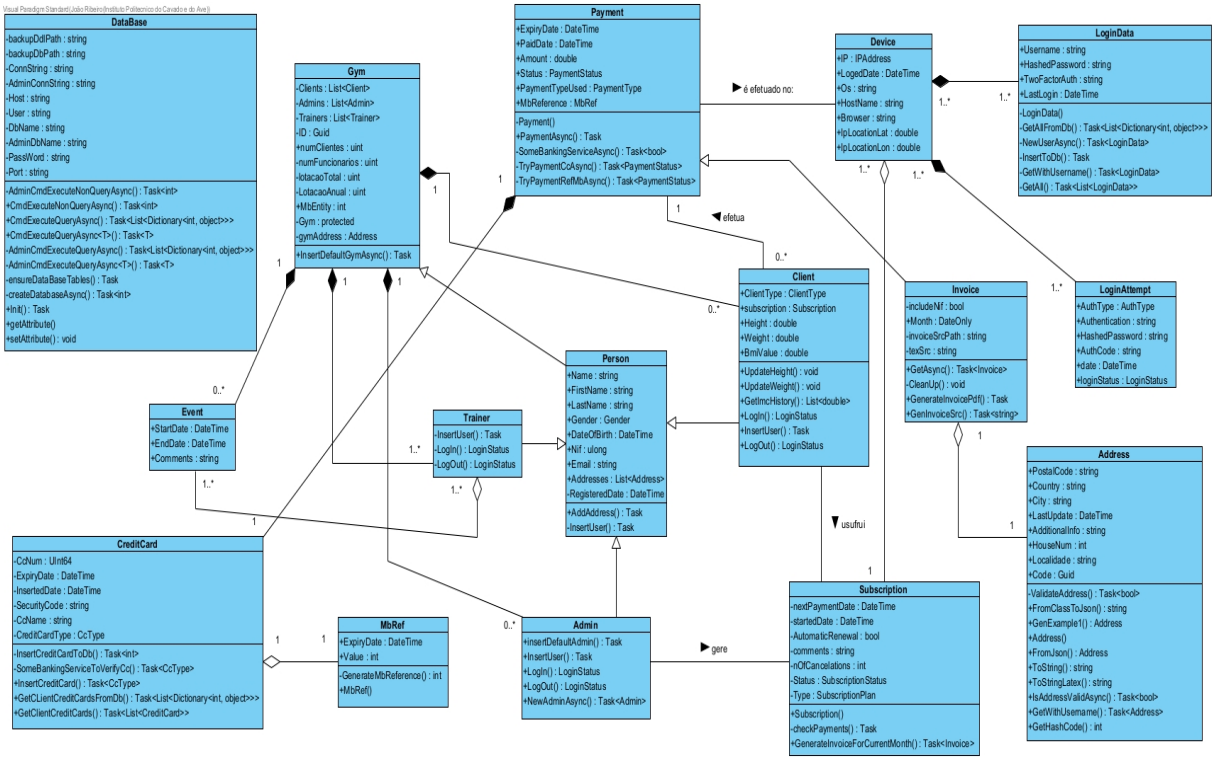
Chapter 9

Diagramas

9.1 Diagrama ER



9.2 Diagrama de Classes



Chapter 10

Conclusão

Com este trabalho concluimos que este projeto foi bastante importante para a nossa aprendizagem e para elevar o nosso trabalho em grupo. Aprendemos a trabalhar com a linguagem C# e a organizar melhor as tarefas, dividindo-as pelo grupo.