ESCOLA
SUPERIOR
DE TECNOLOGIA
IPCA

Programação Orientada a Objetos

Trabalho prático realizado por Gonçalo Costa

a26024@alunos.ipca.pt

LESI

Professor: Luís Ferreira

Índice

Introdução:

No âmbito da disciplina Programação Orientada a Objetos, 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.

O tema escolhido foi acerca de um sistema de gestão de um UCCI (unidade de cuidados continuados intensivos).

Objetivos do projeto:

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

Primeira Fase:

Na primeira fase é pedido a identificação de classes, a implementação essencial de cada classe e a sua estrutura de dados a utilizar.

Foram criadas 3 DLL´s para a divisão de pessoas de produtos e de equipamentos respetivamente

Na DLL *persons*, abordamos todas as pessoas constituintes pessoas da UCCI, como os funcionários e diferentes tipos de funcionários (diretores, enfermeiro, auxiliar) e os utentes.

Na DLL *product*, foram abordados todo o tipo de produtos da UCCI, sejam produtos de limpeza/higiene ou farmácia.

Na DLL *equipment*, são abordados todo o tipo equipamento usados na UCCI, sejam medicinais, como os concentradores de oxigénio ou os desfibriladores, ou de lazer, como televisões, projetores e de outros tipos.

Classes:

Foram criadas 21 classes e 1 interface, sendo que dessas 21 classes, 10 são de armazenamento e gestão de "*array",* nomeadamente as classes Pessoas, Funcionários, Médicos, Auxiliares, Enfermeiros, Utentes, Equipamentos, Produtos, Medicamentos, Limpezas.

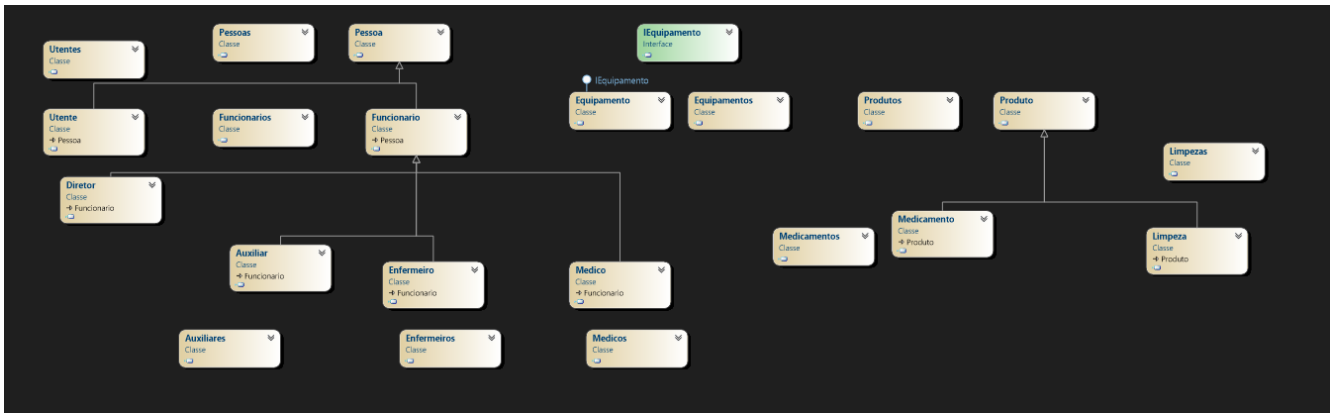


Figura1: diagrama de classes da gestão de uma UCCI

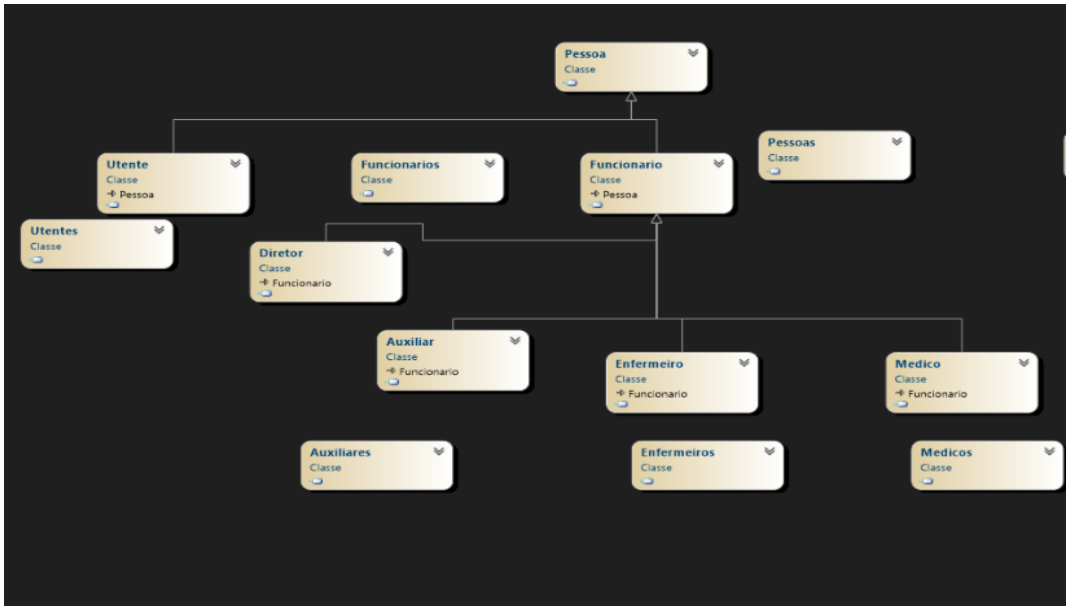


Figura1a: diagrama de classes da secção Persons

Na figura1a, podemos observar todas as classes que constituem a *DLL* que irá definir todas as pessoas que vão estar presentes na UCCI, sejam eles utentes, funcionários (médicos, diretor, enfermeiros e auxiliares).

A classe *Pessoa* é o elemento principal e estrutural deste sistema.

Esta classe desempenha um papel central ao armazenar dados essenciais como nome, apelido, idade, NIF e número de identificação no SNS.

Tais atributos estabelecem uma base sólida de informações compartilhadas por outras entidades especializadas no sistema.

Além disso, a estruturação da classe Pessoa permite uma extensão coerente para outras classes mais especializadas, como Funcionário e Utente(herança).

A classe *Pessoas* permite armazenar e gerir um array de Pessoas, onde contém métodos de inserção e remoção do objeto no array.

A classe *Funcionário* é uma extensão da classe Pessoa, destinada a representar indivíduos que desempenham papéis específicos na UCCI. Ela herda características essenciais da classe Pessoa, mas também introduz atributos próprios, como código, cardo, data de entrada na UCCI e contato.

A sua estrutura permite a distinção entre diferentes funções e responsabilidades dentro do ambiente do sistema, facilitando a atribuição de tarefas, a gestão de hierarquias e a organização administrativa. Essa classe serve como base para a criação de uma estrutura mais ampla que gerência e controla o pessoal envolvido nesta UCCI.

A classe *Funcionários* permite armazenar e gerir um array de Funcionários, onde contém métodos de inserção e remoção do objeto no array.

A classe *Utente* é uma extensão da classe Pessoa, direcionada a pessoa que recebem os serviços da UCCI. Herda as propriedades básicas da classe Pessoa, como nome, apelido, idade, NIF e número de identificação no SNS, mas também inclui características específicas, como o contacto do familiar e a sua chegada a UCCI e a avaliação da sua situação clínica.

Dessa forma, a classe Utente amplia as informações da classe Pessoa ao incluir dados específicos, viabilizando uma abordagem mais eficaz e personalizada para atender às necessidades individuais dos usuários.

A classe *Utentes* permite armazenar e gerir um array de Utentes, onde contém métodos de inserção e remoção do objeto no array.

A classe *Médico* é uma extensão da classe Funcionário. Além das informações gerais de um funcionário, ela inclui detalhes sobre a especialidade médica, organização dos médicos da UCCI.

A classe *Médicos* permite armazenar e gerir um array de Médicos, onde contém métodos de inserção e remoção do objeto no array.

A classe Auxiliar é uma extensão da classe Funcionário, representando membros da equipa de trabalho da UCCI. Além das informações padrões de um funcionário, ela pode conter dados relacionados à função de auxílio prestada dentro do sistema, como códigos específicos e condição de trabalho.

A classe *Auxiliares* permite armazenar e gerir um array de auxiliares, onde contém métodos de inserção e remoção do objeto no array.

A classe *Enfermeiro* é uma extensão da classe Funcionário, focalizando profissionais de enfermagem. Ela agrega informações essenciais sobre enfermeiros, além dos detalhes comuns de um funcionário, incluindo um código de identificação e informações sobre a condição profissional.

A classe *Enfermeiros* permite armazenar e gerir um array de Enfermeiros, onde contém

métodos de inserção e remoção do objeto no array.

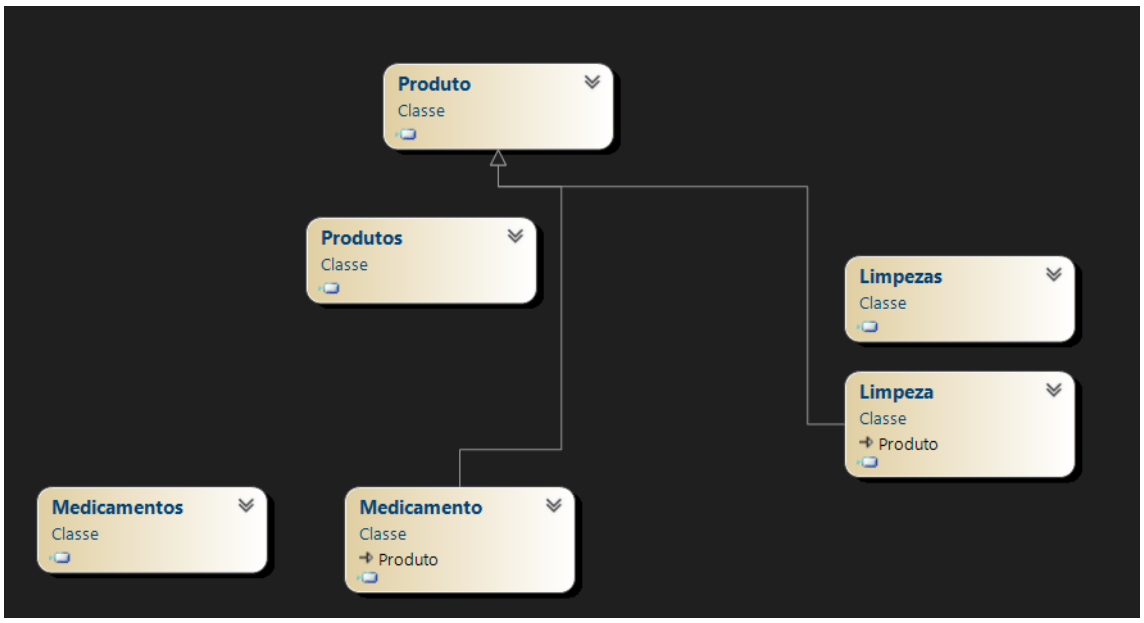


Figura1b: diagrama de classes da secção Product

Na figura1b, podemos observar todas as classes que constituem a *DLL* que irá definir todos os produtos que vão estar presentes na UCCI, sejam eles medicamentos ou produtos de limpeza/higiene.

A classe Produto é a classe principal desta DLL onde são apresentados todo o tipo de produto utilizados na UCCI. Nesta classe contem atributos como nome, código e o código de fornecedor.

A classe *Produtos* permite armazenar e gerir um array de Produtos, onde contém métodos de inserção e remoção do objeto no array.

A classe *Medicamento* é um herdeiro da classe Produtos, destinada a apresentar todos os medicamentos da UCCI. Ela herda características essenciais da classe Produto, mas também introduz atributos próprios, como o seu código de medicamentos e a sua marca.

A classe *Medicamentos* permite armazenar e gerir um array de Medicamento, onde contém métodos de inserção e remoção do objeto no array.

A classe *Limpeza* é um herdeiro da classe Produtos, destinada a apresentar todos os produtos de limpeza/ higiene pessoal da UCCI. Ela herda características essenciais da classe Produto, mas também introduz atributos próprios, como a sua referência e uma descrição de utilidade.

A classe *Limpezas* permite armazenar e gerir um array de Limpeza, onde contém métodos de inserção e remoção do objeto no array.

Figura1c: diagrama de classes da secção Equipment

Na figura1c, podemos observar todas as classes que constituem a *DLL* que irá definir todos os equipamentos da UCCI, sejam eles medicinais ou de normal funcionamento da UCCI.

A classe *Equipamento* é a classe principal desta DLL onde são apresentados todo o tipo de equipamentos utilizados na UCCI. Nesta classe contem atributos como nome, código, descrição, carga e o seu estado (bom ou avariado).

A classe *Equipamentos* permite armazenar e gerir um array de Equipamento, onde contém métodos de inserção e remoção do objeto no array.

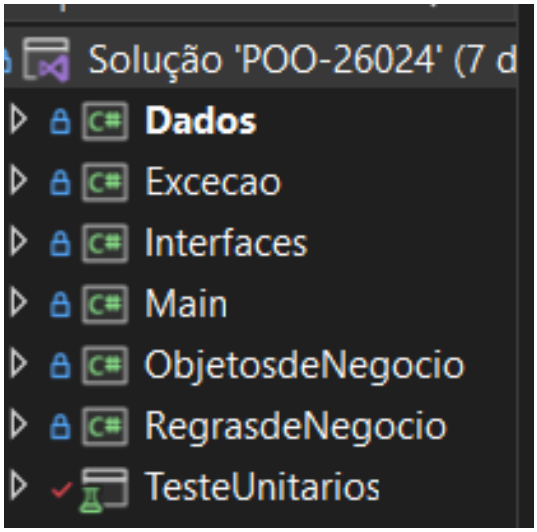
A interface *IEquipamentos* indica que a classe Equipamento deve ter a propriedade Estado.

Segunda Fase

Para esta segunda fase, foram feitas bastantes alterações, seja a nível de estruturas, objetivos iniciais do trabalho.

Primeiramente, a nível de classes, foram removidas algumas classes, pois foi entendido que não é o mais importante definir classes, ou um bom sistema de gestão, neste caso, de uma "UCCI".

Foi abordada, como pretendido pelo professor, a programação por camadas, onde estão divididos por etapas diferentes, como "objetos de negócio", "dados", "exceções", "interfaces", "Regras de Negócio", "InputOutput".



Para esta segunda fase, também foram usadas novas estruturas de dados, *collections*, neste projeto, *Lists.* Foi abordado a utilização de exceções e como devem ser utilizadas corretamente.

Diagramas de Classes

Figura: Diagrama de Classe das Exceções

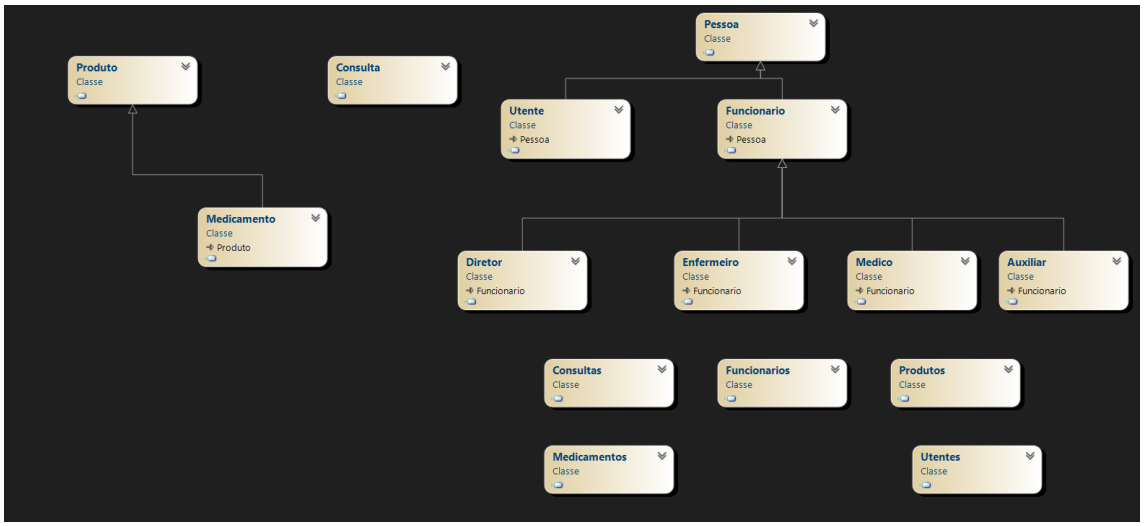


Figura: Diagrama de classes

Comparando a fase 1, podemos verificar a adição de exceções e a alteração do número de classes, algumas foram adicionadas, outras removidas.

Estruturas de dados

Existiu uma mudança da primeira fase para a segunda a nível de Estruturas de dados, na primeira fase, foram usados as estruturas de dados *Arrays*. Para a segunda fase, foram todas transformadas para *Lists*.

As *Lists* oferecem flexibilidade, permitindo adicionar e remover objetos sem redimensionamento explícito, enquanto os *Arrays* têm tamanho fixo. Além disso, as *Lists* gerem a memória de forma autonoma, ajustando-se conforme necessário, ao contrário dos *Arrays*, que exigem realocações manuais, o que pode ser menos eficiente em termos de desempenho e manutenção de código.

```
public class Funcionarios
{
    private static List<Medico> medicoslist;
    private static List<Enfermeiro> enfermeirosList;
    private static List<Auxiliar> auxiliaresList;
```

Figura: Utilização de List

Implementação:

Neste projeto, desenvolveu-se um sistema de gestão de uma UCCI, onde tenta-se mostrar um pouco do que se passa “dentro” de uma UCCI.

A verdade é que o principal objetivo, nunca foi fazer um grande sistema de gestão de uma UCCI, mas sim bom código em C#, código “claro”, simples e com muita qualidade.

No mesmo é possível inserir bastantes “atores”, sejam eles, “utentes”, “enfermeiros”, “médicos”, “auxiliares”, as “consultas externas” que a UCCI apenas recebe e respeita ordens e até “medicamentos”, produtos fundamentais para um bom funcionamento de um UCCI.

Além das inserções, é possível manipular de outras formas os objetos, seja pelas listagens, remoção ou outros métodos específicos para cada ator.

Foi implementada a programação por camadas “*NTier”*, onde todo o código está organizado em diferentes camadas, com o objetivo de melhor abstração.

As Regras de negócio definem o que é possível obter ou não na camada de Input/Output, onde a mesma terá de comunicar com as Regras de negócio e depois a mesma irá comunicar com a camada de dados.

As mesmas filtram também métodos na sua camada e só ira para o Input/output o que ela mesmo permitirem.

Exceções

Exceções são elementos cruciais para lidar com situações inesperadas ou erros durante a execução do programa.

As exceções representam erros que podem surgir durante a execução do programa e permitem que sejam capturadas, tratadas de maneira controlada, garantindo a estabilidade do código e do programa.

Foram criadas exceções desde exceções genéricas para atores até exceções de acontecimentos específicos.

```
/// <summary>
/// Exceção genérica para Utentes.
/// </summary>
/// <seealso cref="System.Exception" />
9 referências
public class UtenteException : Exception
{
    0 referências
    public UtenteException() { }

    3 referências
    public UtenteException(string message) : base(message) { }
}
```

Figura: Exceção genérica

```
/// <summary>
/// Exceção para escrita de ficheiro.
/// </summary>
/// <seealso cref="System.Exception" />
23 referências
public class EscritaFicheiro : Exception
{
    0 referências
    public EscritaFicheiro() { }

    10 referências
    public EscritaFicheiro(string message) : base(message) { }
}
```

Figura: Exceção para situação de escrita de ficheiro

Testes Unitários

Foram realizados testes para a inserção de utentes e consultas:

```
[TestMethod]
0 referências
public void InsereUtente()
{
    Utente utente1 = new Utente(924070650, new DateTime(2023,2,20), "Estavel", 1, "Joao", "Ferreira", 65, 123456789, 1001);
    Utente utente2 = new Utente(924070650, new DateTime(2023, 2, 20), "Estavel", 1, "Joao", "Ferreira", 65, 123456789, 1001);
    Utente utente3 = new Utente(924070650, new DateTime(2023, 2, 20), "Estavel", 1, "Joao", "Ferreira", 65, 123456789, 1001);
    Utente utente4 = new Utente(924070650, new DateTime(2023, 2, 20), "Estavel", 1, "Joao", "Ferreira", 65, 123456789, 1001);
    Utente copiaUtente1 = utente1;
    bool resultado;
    try
    {
        bool resultado1 = Utentes.InsereUtenteLista(utente1);
        bool resultado2 = Utentes.InsereUtenteLista(utente2);
        bool resultado3 = Utentes.InsereUtenteLista(utente3);
        bool resultado4 = Utentes.InsereUtenteLista(utente4);
        bool copiaresultado1 = Utentes.InsereUtenteLista(copiaUtente1);

        bool remocao = Utentes.RemoverUtentes();

        Assert.IsTrue(remocao);
    }
    catch (Exception )
    {
    Assert.Fail("ERRO");
    }
```

Figura: Inserção de Utentes

```
public void InsereConsultas()
{
    Consulta consulta1 = new Consulta(1,  new DateTime(2023, 12, 21), 123456789, 987654321, "Hospital Regional", 20 );
    Consulta consulta2 = new Consulta(1, new DateTime(2023, 12, 21), 123456789, 987654321, "Hospital Regional", 20);
    Consulta consulta3 = new Consulta(1, new DateTime(2023, 12, 21), 123456789, 987654321, "Hospital Regional", 20);
    Consulta consulta4 = new Consulta(1, new DateTime(2023, 12, 21), 123456789, 987654321, "Hospital Regional", 20);
    Consulta copiaConsulta1 = consulta1;

    bool resultado;
    try
    {
        bool resultado1 = Consultas.InsereConsultaLista(consulta1);
        bool resultado2 = Consultas.InsereConsultaLista(consulta2);
        bool resultado3 = Consultas.InsereConsultaLista(consulta3);
        bool resultado4 = Consultas.InsereConsultaLista(consulta4);
        bool copiaresultado1 = Consultas.InsereConsultaLista(copiaConsulta1);

        bool remocao = Consultas.RemoverConsultas();

        Assert.IsTrue(remocao);
    }
    catch (Exception)
    {
        Assert.Fail("ERRO");
    }
}
```

Figura: Adição de Consultas

O objetivo dos testes foi induzir em erro, ou seja, com a inserção incorreta, onde, como era pretendido, os testes falharam, como mostra a imagem abaixo

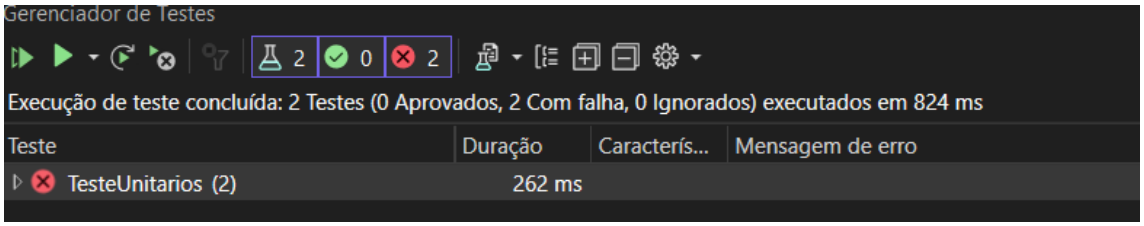


Figura: Testes falharam, como pretendido

Conclusão:

Com a realização deste projeto, foi possível atingir todos os conteúdos abordados nas aulas de POO, que resultou na evolução da capacidade lógica de pensamento, e na capacidade de programação em C Sharp.

Nenhum projeto é um verdadeiro projeto sem desafios, sem obstáculos, com isto, é importante referir a evolução notória nas capacidades de desenvolvimento, que resultaram de várias e longas pesquisas, sempre que existia um problema e sempre que pretendia-se avançar e evoluir para um próximo patamar na capacidade de programação.

Em suma, apenas se pode retirar bons motivos ao falar deste projeto e o quanto o mesmo me fez evoluir enquanto desenvolvedor/programador. Projetos difíceis/ desafiadores.

Gonçalo Cardoso Ferreira da Costa

Número: 26024

LESI-IPCA

Bibliografia:

- GitHub do Professor Luís G. Ferreira: LESI-POO-2023-2024;
- Sebenta de C Sharp by lufer;
- Canal de Youtube “Bro Code” como complementar em C Sharp;