INSTITUTO POLITÉCNICO
DO CÁVADO E DO AVE
ESCOLA SUPERIOR DE TECNOLOGIA

RELATÓRIO DE TRABALHO PRÁTICO

Sistema de Gestão de Crise de Saúde Pública

JOEL MARTINS | **JOSÉ MATOS**

ALUNO Nº 17439 | **ANUNO Nº 19334**

Trabalho realizado sob a orientação de:
Luís Ferreira

Linguagens de Programação II

Licenciatura em Engenharia de Sistemas Informáticos

Barcelos, maio de 2020

Índice

Lista de Figuras

1 Introdução

Neste sistema será dedicado a implementar todos os conhecimentos aprendidos na unidade curricular de Linguagem de Programação II.

A programação para a criação de um sistema que permita gerir as pessoas infetadas numa situação de crise de saúde publica, foi escrito na linguagem C# (C-Sharp), Linguagem de Programação Orientada a Objetos, na plataforma .NET (dotNet), uma plataforma de execução de aplicações Windows. Esta plataforma inclui um sistema de execução virtual designado CLR (Common Language Runtime) e uma vasta biblioteca de classes com diversas funcionalidades, desde a apresentação dos dados no ecrã em modo consola até ao acesso a base de dados.

O código será compilado numa linguagem intermediária designada por MSIL (Microsoft Itermediate Language) que está em conformidade com a especificação CLI (Common Language Infrasstructure), o resultado é um ficheiro executável (.exe ou .dll).

Quando o programa em C# é executado, o Assembly é carregado no CLR que, caso as condições estejam satisfeitas, executará a compilação JIT (Just In Time) para converter o código MSIL em instruções nativas da máquina.

1.1 Criação do Sistema

Na situação de crise de saúde pública o paradigma da programação orientada por objetos, iremos usar classes que é a apresentação de uma estrutura de um objeto com os seus atributos e métodos.

No nosso trabalho começamos por identificar as classes, atribuindo-lhes um conjunto de atributos e métodos. Classes essas que serão base de classes derivadas que favorece a reutilização do código.

Neste quesito é onde sentimos grandes dificuldades em fazer a transição da Linguagem de Programação C para este novo mundo de programação orientado a objetos, no entanto acreditamos que estamos bem direcionados para nos embrenhar nesta linguagem.

Devido a esta dificuldade tentamos manter a nossa solução o mais simples possível, no entanto demonstrando que conseguimos efetivamente dominar as bases da programação em C#.

1.2 Segunda Fase do Trabalho

Nesta segunda fase do trabalho, trocamos a utilização de “Arrays” para “Listas”, conferindo outra simplicidade ao manuseamento dos dados existentes no programa.

Com Listas temos, por exemplo, a facilidade de adicionar dados sem que seja necessário percorrer toda a lista à procura de um espaço livre para a inserção desse dado, e aliás, nem a preocupação de verificar se existe espaço livre suficiente é necessária, tornando o código mais limpo, mais eficiente e mais prático. Também a procura, a alteração ou eliminação de um dado específico se torna muito mais simples.

Foram adicionadas várias exceções para evitar que o nosso programa colapse inesperadamente e sem qualquer aviso prévio, o que pode ocorrer muito facilmente quando, por exemplo, é pedido um número ao utilizador e este insere uma palavra.

Primeiramente decidimos fazer a escrita das listas em ficheiros de texto, no nosso caso, do tipo CSV, para que nos fosse mais fácil a visualização, e caso necessário a edição, daquilo que efetivamente estávamos a escrever nos ditos ficheiros. Achamos por bem manter este código no trabalho final para demonstrar a nossa capacidade de o manusear, visto que efetivamente é mais trabalhoso. Posteriormente, adicionamos as funções necessárias para poder fazer a leitura e escrita das listas em ficheiros Binários.

Fizemos melhoramentos em várias funções, como por exemplo a função que retornava a quantidade de pessoas que representavam um caso suspeito com idade igual àquela inserida pelo utilizador, também à função que devolve o número de casos suspeitos quanto ao género, este também inserido pelo utilizador.

Adicionamos uma função para a contagem de casos com pessoas infetadas. Esta função faz uma travessia da lista de casos suspeitos, verificando quais representam uma pessoa infetada, contabilizando cada um deles e devolve o somatório total.

Foi também adicionada uma função para que o utilizador possa atualizar um caso suspeito, isto é, poderá alterar o estado de infetado para verdadeiro ou falso.

Nesta segunda fase do trabalho, conseguimos fazer uma tímida aproximação às expressões Lambda e ao LINQ (Language Integrated Query), na qual verificamos que, apesar de um pouco confuso e de difícil "leitura", se consegue reduzir uma função/método a apenas uma expressão, otimizando a aplicação.

2 Desenvolvimento do sistema

2.1 Diagrama de classes

No Paradigma de Programação Orientada por Objetos, o diagrama de classes é uma estrutura que permite ver como estão descritas as classes, os seus atributos e os seus métodos, bem como a relação entre os objetos.

Para a resolução do sistema definimos o diagrama de classes em infra:

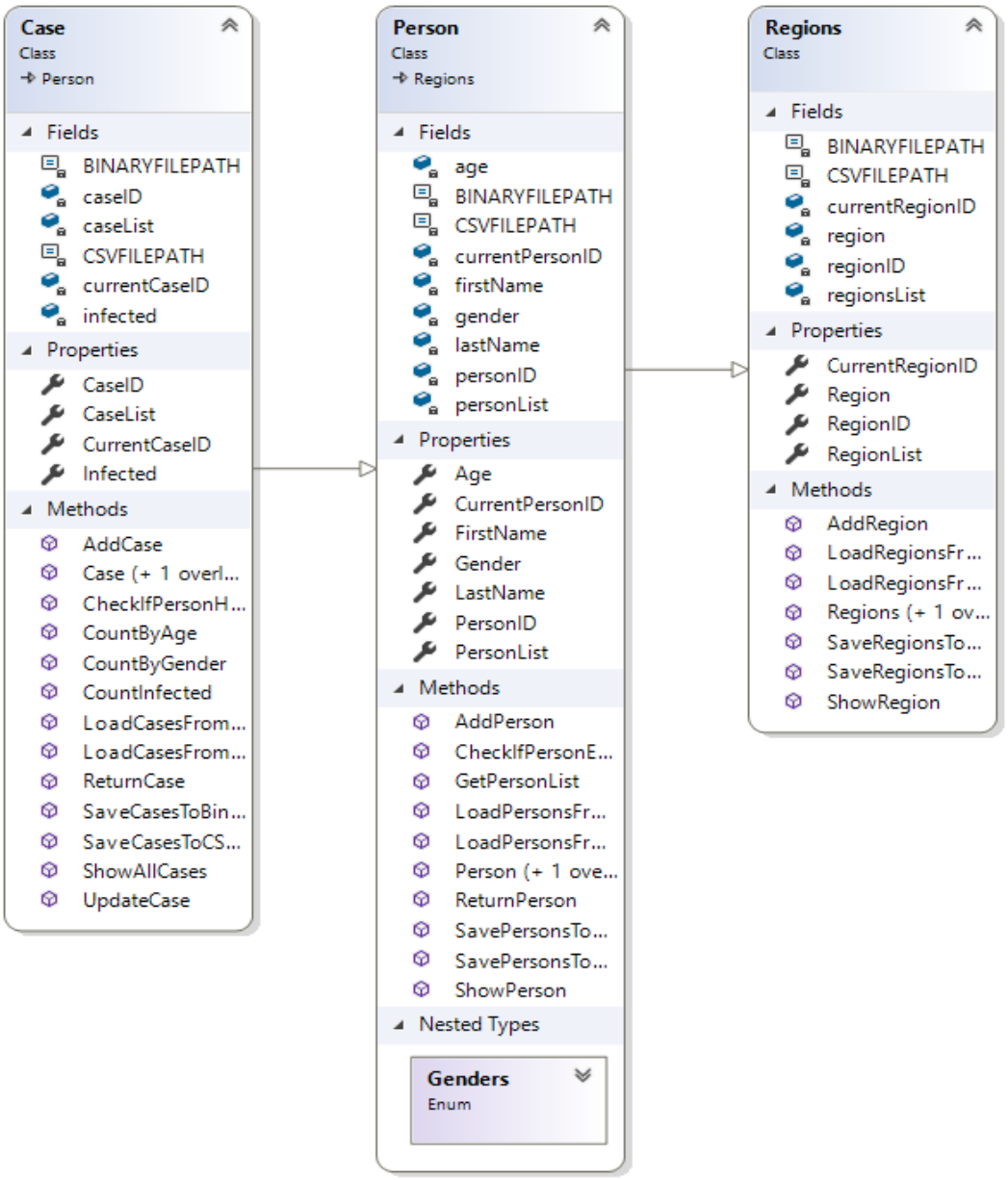

Figura 1 – Diagrama de Classes

2.1.1 Classe Person

Nesta classe, é onde é definido o que é uma pessoa, e todas as funções relacionadas com pessoas.

Ao fazer o registo de uma pessoa, terá de ser inserido o primeiro nome, último nome, género, idade e o "id" da região a que esta pessoa pertence, sendo o "id" único atribuído automaticamente. De seguida, na Figura 2 mostramos um exemplo de código para adicionar uma pessoa à lista de pessoas.

```
Person personExtra1 = new Person("Helder", "Cunha", Person.Genders.M, 33, 7);
defaultPerson.AddPerson(personExtra1);
```

Figura 2 – Adicionar Pessoa

Como se pode verificar é utilizada a função "AddPerson". Esta é uma função que está presente na classe Person, que recebe por parâmetro uma pessoa e a adiciona à lista de pessoas, como demonstra a Figura 3.

```
public bool AddPerson(Person newPerson)
{
    foreach (var person in personList)
    {
        if (newPerson.personID == person.PersonID)
        {
            return false;
        }
    }
    personList.Add(newPerson);
    return true;
}
```

Figura 3 – Função que adiciona pessoa à lista

Como se pode verificar na Figura 3, apesar de o "id" de uma pessoa ser automaticamente incrementado, isto quer dizer que será virtualmente impossível existirem dois "id" iguais, fazemos uma travessia da lista para verificar se existe um "id" igual ao inserido, conferindo outra robustez à nossa solução.

2.1.2 Classe Case

Nesta classe é definido o que deverá ser um caso suspeito de uma pessoa infetada. Aqui é apenas atribuído um "id" único a cada caso, o "id" da pessoa de que se suspeita a infeção e é guardada a informação se esta pessoa está ou não infetada. Na Figura 4 mostramos o construtor que faz este processamento.

```
public Case(int personId, bool infected)
{
    this.caseID = Interlocked.Increment(ref currentCaseID);
    this.PersonID = personId;
    this.infected = infected;
}
```

Figura 4 – Construtor de um caso suspeito

Assim como na classe Person, temos também nesta classe uma função que adicionará um caso à lista de casos.

Tal como referido na introdução, nesta segunda fase do trabalho fizemos uma aproximação às expressões Lambda e ao LINQ, na seguinte figura podemos visualizar este tipo de código:

```
public Case ReturnCase(int personId)
{
    var caso = caseList.Find(x => x.PersonID == personId);

    return caso;
}
```

Figura 5 – Lambda e LINQ

Esta função é declarada do tipo "Case", isto quer dizer que terá de retornar uma variável do mesmo tipo, ou quanto muito terá de retornar nula, ou seja, nada. Resumindo o que vemos na Figura 5: é recebido um "id" de uma pessoa por parâmetro, de seguida atribuímos à variável "caso" a procura na lista de casos, em que "x" toma o "id" de pessoa e compara com o "id" recebido, se for encontrado o "id" igual, então a variável "caso" ficará com o objeto caso correspondente com o "id" procurado. Esta função será então uma alternativa a por exemplo o método "foreach", que iria percorrer todos os casos, verificando se cada um deles teria o "id" igual ao procurado.

2.1.3 Classe Regions

Nesta classe é definido para cada região o seu nome e o "id" único. Visto não haver nada de novo em relação a outras classes, aproveitamos para demonstrar, na seguinte figura, o processo de ler dados a partir de um ficheiro binário e guardar esses dados na lista de regiões:

```
public bool LoadRegionsFromBinaryFile()
{
    try
    {
        FileStream fs = new FileStream(BINARYFILEPATH, FileMode.Open, FileAccess.Read);
        BinaryFormatter bin = new BinaryFormatter();

        if (File.Exists(BINARYFILEPATH))
        {
            regionsList = (List<Regions>)bin.Deserialize(fs);
        }
        else
        {
            fs = File.Create(BINARYFILEPATH);
            bin.Serialize(fs, regionsList);
            currentRegionID = 0;
        }

        fs.Close();

        Regions aux = regionsList.Last();
        currentRegionID = aux.regionID;

        return true;
    }
    catch (Exception e)
    {
        Console.WriteLine("Erro: " + e.Message);
    }

    return false;
}
```

Figura 6 – Ler ficheiro binário

Como se pode observar, na variável "fs" é guardado o caminho do ficheiro a ser lido, a especificação para o Sistema Operativo saber como deve abrir o ficheiro e a definição de acesso a esse mesmo ficheiro, neste caso, apenas de leitura.

De seguida é criada a variável "bin" do tipo *BinaryFormatter,* este tipo de variável serve para serializar ou desserializar, isto é, prepara os dados para que possam ser lidos ou gravados no ficheiro binário.

Fazemos a verificação de que realmente o ficheiro existe, e de seguida toda a informação contida no ficheiro é “despejada” para a lista de regiões.

Na seguinte figura demonstramos o código de como é guardar uma lista de dados num ficheiro binário:

```
public bool SaveRegionsToBinaryFile()
{
    try
    {
        FileStream fs = new FileStream(BINARYFILEPATH, FileMode.Create);
        BinaryFormatter bin = new BinaryFormatter();
        bin.Serialize(fs, regionsList);
        fs.Close();

        return true;
    }
    catch (Exception e)
    {

        Console.WriteLine("Erro: " + e.Message);
    }
    return false;
}
```

Figura 7 – Escrita em ficheiro binário

Tal como demonstrado na *Figura 7*, o processo de gravar os dados num ficheiro binário não difere muito da sua leitura, sendo a grande diferença o tipo de acesso que o Sistema Operativo vai ter sobre o ficheiro, neste caso será de escrita.

2.2 Exceções

Exceções representam erros que podem ocorrer durante a execução da aplicação. Tal como observado na *Figura 6* e *Figura 7*, a implementação de exceções aquando da leitura ou escrita de um ficheiro é extremamente importante porque poderá acontecer um erro completamente inesperado. Com a implementação das exceções conseguimos proceder ao tratamento destes imprevistos, informando o utilizador, e até mesmo o próprio programador, de qual foi o erro encontrado.

Na figura seguinte, demonstramos um exemplo de outro tipo de exceções muito usual:

```
try
{
    Console.Write("Qual o ID da pessoa com suspeitas de virus: ");
    personId = Convert.ToInt32(Console.ReadLine());
}
catch (FormatException e)
{
    Console.WriteLine("Erro: " + e.Message);
}
```

Figura 8 - Exceção

Neste caso, a variável “personId” é do tipo inteiro, pedimos a introdução de dados ao utilizador, o que acontece frequentemente é o utilizador enganar-se ou mesmo propositadamente e insere dados que não sejam do tipo inteiro. Acontecer isto sem a utilização de uma exceção, levaria a que o programa fechasse inesperadamente. Fazendo o tratamento desta exceção, podemos enviar uma mensagem para o ecrã dizendo o tipo de erro encontrado e uma breve descrição, algo do género de: “Erro: Cadeia de caracteres de entrada com formato incorreto”. Relembro que o mostrado na figura é apenas um exemplo, e não o devido tratamento que se deverá dar a uma exceção.

3 Conclusão

Em consideração final, pensamos ter demonstrado que conseguimos implementar o conteúdo lecionado em aula, apresentando uma solução simples, mas que está assente sobre os alicerces da Programação Orientada a Objetos.

Tivemos alguma dificuldade em fazer a transição da linguagem C para C#, principalmente na definição de classes, no entanto achamos ter atingido o objetivo, mas claro, estamos cientes que temos ainda muito espaço para melhoramento.

Inicialmente achávamos que a escrita em ficheiro de texto seria mais simples, apenas pelo facto de poder visualizar o conteúdo. Mais tarde descobrimos que a escrita em ficheiros binários se torna muito mais simples e intuitiva.

Com esta segunda fase aprendemos também novas formas de guardar os dados em memória, mais propriamente as listas, que transferem uma enorme simplicidade em comparação com os arrays.

Finalmente, concluímos que toda a pesquisa e “batalha” com este desafio nos levou a ficar com mais vontade de aprofundar os conhecimentos sobre C#.

Bibliografia

Ferreira, Luis: C# Essencial Linguagens de Programaçao & Programação Orientada a Objetos . Versao 6.0, IPCA, março 2017.

Henriques, Jorge & Trigo, Antonio: Aprenda a programar com C#. 1ª Ediçao, Lisboa, janeiro de 2018.