



## **Relatório**

**Trabalho prático de Linguagens de Programação II**

**Aluno: André Ferreira (18865)**

**Professor: Luís Ferreira**

**Licenciatura Engenharia Sistemas Informáticos**

**Barcelos, maio de 2020**

## Resumo

Este trabalho prático, relativo à unidade curricular de **Linguagens de Programação II**, visa explorar e implementar uma solução na linguagem C# (csharp).

É pretendido demonstrar os conceitos abordados nas aulas de Linguagens de Programação II, aplicando-os numa aplicação que deve ser responsável por gerir uma eventual situação pandémica e as pessoas infetadas. Esta primeira abordagem ao projeto é muito linear, dado que existem inúmeras possibilidades de implementação.

Esta primeira fase foi realizada com recurso à linguagem C#, através da aplicação Visual Studio, sendo executada em ambiente consola.

Procurou-se aliar os conhecimentos obtidos em aula com pesquisa adicional, de forma a conseguir fornecer uma solução que cumpra as normas CLS (Common Language Specification) e também uma lógica de execução coerente.

Foi utilizado um sistema de programação em camadas do tipo NTIER, que irá ser abordado mais á frente neste documento.

Para existir a possibilidade de armazenar a informação, de forma a que a mesma não seja perdida aquando do término de execução da aplicação, foram usados ficheiros binários.

## Palavras-Chave

- CLS;
- IDE;
- DLL;
- Regex;
- Pandemia;
- Classe;
- Herança;
- Composição;
- Objetos;
- Listas;
- Métodos;
- Atributos;
- Propriedades;
- Funções;
- Debug;
- POO;
- N-TIER;
- Serialização

## Índice de Figuras

Figura 1 - Classes .....	3
Figura 2 - Composição de uma classe .....	3
Figura 3 - Exemplo de propriedade .....	4
Figura 4 - Exemplo de método .....	5
Figura 5 - Exemplo Try-Catch.....	8
Figura 6 - Diagramas de classes por camada .....	12
Figura 7 - Exemplo ReGex.....	14

## Índice

1.	Introdução .....	1
1.1.	Contextualização .....	1
1.2.	Motivação e objetivos .....	1
1.3.	Estrutura do Documento .....	1
2.	Análise ao problema.....	2
3.	POO – Programação orientada a objetos .....	3
3.1.	Classes .....	3
3.2.	Bibliotecas .....	6
3.3.	Herança e composição .....	7
4.	Exceções .....	8
5.	Armazenamento de informação.....	9
5.1.	Ficheiros binários.....	9
5.2.	Serialização.....	9
6.	Programação em camadas .....	10
6.1.	TIER Front End .....	10
6.2.	TIER Business Object .....	10
6.3.	TIER Business Logic.....	11
6.4.	TIER Data Access Object .....	11
7.	Implementação .....	12
7.1.	Classes usadas .....	12
7.2.	Bibliotecas .....	14
8.	Conclusão .....	15
9.	Webgrafia.....	16

## 1. Introdução

### 1.1. Contextualização

Este trabalho prático, relativo à unidade curricular de **Linguagens de Programação II** visa implementar em C# uma solução que permita gerir, numa situação de pandemia, pessoas infetadas. No contexto atual, em que vivemos com uma pandemia, são muitas as ideias que poderiam ser aplicadas a uma aplicação com este conceito que apesar de aparentar ser simples, é na realidade algo complexo.

### 1.2. Motivação e objetivos

Este trabalho prático, por ser parte fundamental na avaliação da unidade curricular de Linguagens de Programação II, traz sempre alguma carga motivacional extra. Objetivamente é expectável encontrar desafios, mas também soluções, soluções essas que trazem sempre uma carga de satisfação grande. Um dos objetivos alcançáveis através deste trabalho é o de cimentar os conhecimentos obtidos em aula, explorando os conteúdos já lecionados de acordo com linhas definidas para o trabalho prático. Adicionalmente o fator pesquisa terá também ele peso determinante na conclusão deste projeto.

### 1.3. Estrutura do Documento

Este documento está estruturado de forma a que seja simples a sua leitura. Existe recurso a referências de material fornecido pelo professor Luís Ferreira e/ou referências a excertos de Webgrafia.

Foram, sempre que necessário, adotadas técnicas de **debug**, que permitem fazer um seguimento de todo o código, ou de partes do mesmo, identificando situações anómalas facilmente.

## 2. Análise ao problema

Foram propostos diversos “temas” para este projeto. Atendendo ao contexto atual, em que efetivamente vemos através dos mídeas uma panóplia de situações que nós enquanto cidadãos acreditamos sempre que não foram geridas da melhor forma, foi escolhido o tema de realizar uma solução que permitisse gerir pessoas numa situação de pandemia / epidemia.

Naturalmente é algo que deve ser extremamente bem pensado, e assumir que será alvo de alterações recorrentes, mas numa fase inicial o objetivo é o de implementar algo que concretamente identifique uma pessoa, e que lhe atribua uma série de critérios relacionados com uma doença infecciosa.

Como primeira parte, a solução deve “criar” uma pessoa (pensar em objetos). Essa pessoa, deve ter de forma genérica, um nome, e um género (masculino/feminino). Naturalmente isso não chega para definir uma pessoa, razão pela qual foi adicionada a necessidade de adicionar mais informação, como a idade (calculada através da data de nascimento), contatos, entre outros.

Posteriormente existe a necessidade de criar acesso a mais informação:

- Contatos de emergência
- Doenças / Histórico de doenças
- Recuperação em ambiente hospitalar / no domicílio

Este é um projeto concretizado em linguagem C# (csharp), que é uma linguagem orientada a objetos (**POO**). Como tal, a composição de cada um dos elementos deve ser feita de forma a respeitar os conceitos da programação orientada a objetos.

### 3. POO – Programação orientada a objetos

Um dos conceitos introduzidos nas aulas de Linguagens de Programação II foi o conceito de programação orientada a objetos.

Este tipo de programação, envolve a utilização de classes. Mas o que são classes? Poderemos socorrer-nos de uma explicação no site Wikipédia,

*“Em orientação a objetos, uma classe é uma descrição que abstrai um conjunto de objetos com características similares. Mais formalmente, é um conceito que encapsula abstrações de dados e procedimentos que descrevem o conteúdo e o comportamento de entidades do mundo real, representadas por objetos. De outra forma, uma classe pode ser definida como uma descrição das propriedades ou estados possíveis de um conjunto de objetos, bem como os comportamentos ou ações aplicáveis a estes mesmos objetos.”* (Wikipédia, 2020)

Podemos então tirar algumas ilações de forma a simplificar a ideia de classes.

#### 3.1. Classes

Vamos pensar em classes como formas de bolos. Cada forma, é uma espécie de molde, permitindo criar uma série de bolos (objetos). Podemos ainda pensar numa receita como uma outra classe, onde podemos guardar uma série de procedimentos (métodos). Ligado a esta receita, poderíamos ainda visualizar uma lista de ingredientes (atributos) para a concretização do bolo.

Com este tipo de conceito, é possível evoluir o conceito de programação para algo mais concreto, criando assim objetos com mais ou menos detalhe, mas sempre concretos. As classes tem uma extensão “.cs”, como pode ser verificado na Figura 1.

```

> a C# ContactPerson.cs
> a C# DiseaseHistory.cs
> a C# EmergencyContactPerson.cs
> a C# InHome.cs
> a C# InHospital.cs
> a C# Person.cs
> a C# SickPerson.cs

```

Figura 1 - Classes

Uma classe é composta por blocos de código. Fomos instruídos a ser organizados, definindo regiões para envolver e desta forma organizar o código. Por sugestão, foram implementadas as regiões visíveis na Figura 2.

Dentro de cada região são então escritas um conjunto de variáveis, instruções e/ou procedimentos para implementar o conceito de programação orientada a objetos.

```

class Class
{
    ----- MEMBER VARIABLES -----

    ----- CONSTRUCTORS -----

    ----- PROPERTIES -----

    ----- FUNCTIONS -----

    ----- ENUMS -----
}

```

Figura 2 - Composição de uma classe



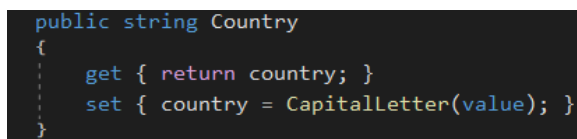
### 3.1.1.Variáveis

As variáveis são uma listagem de atributos definidos numa classe. No caso de uma pessoa por exemplo, podemos atribuir como variáveis, a altura, a cor dos olhos, a idade.

Em **POO**, as variáveis devem idealmente estar “protegidas” e inacessíveis diretamente do mundo exterior. A razão para isso é simples. Imagine-se o que seria se numa solução que envolvesse contas bancárias, alguém “manualmente”, sem qualquer tipo de validação, colocasse um valor diferente do correto/possível. Seria completamente impossível assegurar a viabilidade do processo de por exemplo levantar dinheiro. Razão pela qual as variáveis/atributos, devem estar “private” numa classe, usando-se propriedades para as trabalhar.

### 3.1.2.Propriedades

As propriedades são mecanismos que possibilitam quer a leitura quer a escrita de valores nas variáveis, de forma segura. Note-se que cada propriedade pode conter dois membros: “get;” e “set;”. Na figura 3 é visível um exemplo de uma propriedade chamada Country. Esta propriedade é responsável através do **get** de ler o valor armazenado na variável. Note-se



```
public string Country
{
    get { return country; }
    set { country = CapitalLetter(value); }
}
```

Figura 3 - Exemplo de propriedade

que por convenção as variáveis usam o chamado camelCase, enquanto que as propriedades usam PascalCase. Interessa também referir que é importante relacionar o nome da variável/atributo ao nome da propriedade.

Já foi falado o get, agora vem o set.

No set, podemos definir o valor da variável em função do valor que foi passado á propriedade. Mas esta é uma ferramenta poderosa, e podemos ainda adicionar blocos de instruções que permitem invocar funções, criar validações, etc. No exemplo da figura 3, é visível que à variável country, está a ser adicionado o valor “value”, mas este está a ser “processado” pelo método CapitalLetter.

As propriedades, devem ser públicas, permitindo assim serem usadas fora da classe. Dessa forma, o utilizador pode passar valores às variáveis, mas sempre com o “crivo” das propriedades. Podemos de forma a cimentar ainda mais o conceito, que as propriedades são uma espécie de muralha, onde para acedermos ao conteúdo das variáveis, temos de passar por um portão fortemente guardado por métodos.

### 3.1.3.Métodos

Falamos de métodos no ponto anterior como forma de processar um valor. Mas um método tem muito que se lhe diga.

Métodos são funções/procedimentos, ou se preferirmos uma expressão mais formal, são blocos de instruções que permitem executar tarefas.

A ideia de métodos é a de confinar dentro de chavetas um conjunto de instruções que podem ser invocadas diversas vezes. Desta forma conseguimos minimizar a escrita repetida de instruções, e ao mesmo tempo melhorar a manutenção do código.

Na figura 4 está um exemplo de um método implementado na solução deste trabalho prático.

Uma análise ao método, permite identificar que é um método que devolve um tipo de dados, no caso DateTime. Este método recebe como parâmetro uma “string”. No caso esta “string” traz o input do

```
public DateTime CheckDate(string date)
{
    DateTime aux;
    bool b = DateTime.TryParse(date, out aux);
    this.BirthDate = b ? aux : DateTime.Today;
    this.Age = DateToAge(this.BirthDate);
    return this.BirthDate;
}
```

*Figura 4 - Exemplo de método*

utilizador com uma data. Esta data é então processada, e parseada para uma variável auxiliar. Como forma de validação, porque o erro humano pode hesitar sempre, é feito um TryParse, que também é um método. Caso o utilizador tenha inserido uma data num formato válido, a data é então armazenada já no formato “data” na variável aux. Esta data, é então armazenada na variável birthDate, através da propriedade BirthDate.

Adicionalmente, podemos aproveitar este input para calcular a idade da pessoa. Para tal, passamos o valor da data de nascimento (já validada e armazenada na variável birthDate) como parâmetro para a função DateToAge, que é responsável por devolver a idade da pessoa, com base na data atual e a data de aniversário.

Com este simples exemplo, verificamos métodos, com métodos, realizando diversas funções, e desta forma simplificando um conjunto de operações que podem ser repetidas inúmeras vezes. De forma homóloga, também pensando numa futura manutenção do código, dado que uma implementação está centralizada num único local, é mais fácil, alterar/manter esse código.

### 3.1.4.Construtores

Falamos de atributos, propriedades e métodos, mas para criar um objeto precisamos de algo, fundamental. Um construtor.

Um construtor, como o nome implica, é um tipo especial de métodos, que criam e inicializam objetos.

Por defeito, uma classe tem um construtor. No entanto é possível criar diversos construtores. Podemos criar construtores que recebam um, dois ou vários parâmetros, de forma a criar e inicializar objetos com um dois ou vários valores já definidos.

### 3.2. Bibliotecas

Este trabalho fomenta a utilização de bibliotecas. Mas o que são bibliotecas? Para melhor compreender, pensemos em contentores. Dentro dos contentores estão caixas. Essas caixas, são as nossas classes. Os contentores, são então as bibliotecas.

Mas porquê usar bibliotecas? Porque não colocar apenas classes nas nossas soluções? A resposta é simples, mas ao mesmo tempo completa. Organização e versatilidade.

O que seria termos 100 ou 200 classes dentro da nossa solução? E se tivermos de navegar entre um conjunto de classes que servem de forma para a criação de um objeto específico, como um bolo. Faria sentido estas classes estarem junto de classes que são responsáveis por organizar stocks de armazém de ingredientes?

Ao mesmo tempo, e se criarmos um conjunto de classes, interligadas e com uma grande possibilidade de serem reutilizadas? É complicado reutilizarmos essas classes noutras soluções, quando elas estão misturadas no meio de classes que nada têm a ver.

Para dar resposta a estes dilemas, utilizam-se bibliotecas.

Como indicado acima, bibliotecas são projetos especiais que servem de contentores, para armazenar um conjunto de classes ou outros ficheiros afetos a determinada responsabilidade. Se criarmos um conjunto de classes, que em conjunto realizam um conjunto de tarefas/objetos, e as armazenarmos numa biblioteca, para reutilizarmos esse código basta invocar essa biblioteca, numa nova solução.

É também possível reutilizar código de outros programadores, recorrendo a bibliotecas por eles criadas. Mas surge aqui uma questão importante. Estas bibliotecas, ou o conteúdo das mesmas, devem ser bem estruturadas, testadas e bem implementadas. Se um dia alguém considerar usar uma biblioteca nossa, é muito importante que não surjam erros dentro da nossa biblioteca. Podemos imaginar se alguém constrói uma solução complexa, como por exemplo uma API (Application Programming Interface) para um banco, e por alguma razão alguém consegue numa caixa multibanco, levantar mais dinheiro do que aquele que tem disponível na sua conta bancária? Ou se ao invés de conseguir levantar no máximo 200 euros/levantamento, conseguir levantar 5000 euros? Efetivamente isto leva-nos a redobrar esforços para que tudo aquilo que produzimos nas nossas bibliotecas seja fiável e coerente.

### 3.3. Herança e composição

Um dos pilares da **POO**, é a possibilidade de herança. Mas o que é isto de herança?

Para melhor compreender isto, vamos seguir o exemplo de uma pessoa.

Uma classe do tipo Pessoa tem:

- Nome;
- Idade;
- Altura;
- Peso;

Agora crie-se uma classe que defina um trabalhador e uma outra que defina um desempregado. Quer o Trabalhador, quer o Desempregado, são pessoas. Dessa forma, ambos tem um nome, uma idade uma altura e um peso. Pode-se então aplicar o conceito de herança, em que a classe Trabalhador e a classe Desempregado “herdam” da classe Pessoa atributos e comportamentos. Assim, conseguimos centralizar o que é comum a vários possíveis objetos numa classe “pai”, e criar assim uma “hierarquia de classes”.

Claro que nas classes Trabalhador e Desempregado, poderemos adicionar atributos e comportamentos específicos para cada “conceito” que essas classes defendam.

No exemplo acima indicado à classe Trabalhador, poderíamos adicionar informação de entidade empregadora. Poderíamos pensar em colocar essa informação na classe Trabalhador, mas e se o trabalhador for para o desemprego? Passará a ser desempregado. Mas a classe desempregado, não poderia ter informação de emprego.

Para resolver esta confusão de ideias, recorreremos à composição de classes.

Criamos uma classe chamada Cidadão. Este cidadão herda da classe pessoa os diversos atributos e comportamentos já indicados. Paralelamente, criamos um atributo que indica se está empregado ou desempregado.

Criamos agora uma classe chamada Emprego. Nesta classe podemos definir atributos como empresa, localização da empresa, remuneração, entre outros. Mas vamo-nos abstrair da complexidade da situação profissional.

Como relacionar a classe Cidadão, com a classe emprego? Usamos a composição de classes.

Em suma, inicializamos o objeto Emprego na classe Cidadão. Desta forma dizemos que um Cidadão, tem um Emprego, o que faz todo o sentido. Conseguimos assim ligar estas duas classes sem recorrer à herança de classes.

Existe uma frase conhecida no mundo da programação que diz: *“Favorece a composição sobre a herança”*. Esta frase faz sentido, se pensarmos no exemplo acima. Com a composição resolvemos um problema, adicionando a funcionalidade de outra classe, a uma classe já existente.

## 4. Exceções

Quando um programa executa, e algo que não está previsto acontece, isso pode levar a exceções não tratadas, que regra geral fazem o referido programa crashar. Num ambiente controlado em que o programa está a ser testado, é algo que apesar de ter algum impacto, não é totalmente impactante. No entanto imagine-se que se está a fazer uma transferência bancária, ou até mesmo um pagamento, e o programa encontra uma exceção? O constrangimento por detrás da falha da aplicação nesse instante pode ser devastador.

Para dar resposta a este tipo de situações inesperadas, faz-se recurso a “Exception handlers”.

Em C# existem um conjunto de classes que herdam da classe Exception. Estas classes permitem através do uso de um try-catch “apanhar” estas exceções, e evitar assim que elas causem a falha crítica do programa em execução.

Na figura 5, está um exemplo de implementação de um try-catch. O modo de funcionamento é simples. O código que poderá ter uma vulnerabilidade, no exemplo é um “parse” de uma string para inteiro, é colocado dentro de um “try”. Este try vai tentar executar esse código. Se tudo correr bem, ou seja, se a string for convertida/parseada corretamente para inteiro, o código avança normalmente. No entanto se o processo falhar, o que acontece é uma exceção. Esta exceção é então apanhada por um catch. Os catch’s permitem apanhar inúmeros tipos de exceções.

```
try
{
    Int32.Parse(s);
    Console.WriteLine(
        "You entered valid Int32 number {0}.", s);
}
catch (FormatException)
{
    Console.WriteLine("Invalid integer number!");
}
catch (OverflowException)
{
    Console.WriteLine(
        "The number is too big to fit in Int32!");
}
```

*Figura 5 - Exemplo Try-Catch*

Neste exemplo, são usados dois tipos de exceções, logo são usados dois catch. O primeiro, apanha todas as exceções que sejam originadas por um erro de formato. A segunda, é uma exceção por overflow. Poderia ainda haver recurso a uma terceira exceção, que é a genérica.

A ideia por detrás do uso de exceções é o de irmos no sentido oposto de um funil, ou seja, começamos com as exceções mais concretas, e vamos a par de cada catch aumentando a generalização, podendo terminar então na exceção genérica.

Para o tratamento destas exceções, podem ser realizadas algumas ações, como despoletar uma mensagem de erro ao nível do utilizador, ou por exemplo realizar um log num ficheiro. Tudo isto permite em primeira instância que a aplicação não termine de forma abrupta, podendo haver perda de informação por exemplo, mas também em segunda instância manter um espécie de registo de situações que permitam ao desenvolvedor melhorar / corrigir essas situações se tal for possível.

## 5. Armazenamento de informação

Quando se trabalha com informação é sempre importante pensar no manuseamento da mesma, mas também na possibilidade de não ter que se inserir toda a informação sempre que se inicia a aplicação. A pensar nisso, existem várias soluções. Atendendo a esta fase e aos conhecimentos adquiridos à data, para este programa vai ser implementado um modelo de armazenamento de informação em ficheiros binários. Este modelo permite guardar as estruturas de dados que precisam efetivamente de ser guardadas, sendo que para tal necessitam de ser “serializados”.

### 5.1. Ficheiros binários

Ficheiros binários não são ficheiros de texto. Fazendo um comparativo entre os dois, podemos afirmar que um ficheiro binário é processado através de uma sequência de bytes enquanto que um ficheiro de texto é processado através de uma sequência de caracteres.

Num ficheiro binário os dados são então armazenados em blocos de bytes tal e qual como é feito na memória de um computador. Isto implica naturalmente que um ficheiro binário esteja de certa forma encriptado, não sendo legível pelo ser humano.

Estes ficheiros, são ficheiros com extensão “.bin”.

### 5.2. Serialização

Quando se tenta gravar informação em ficheiros binários não é possível só apontar quais as estruturas a guardar e executar. As mesmas precisam de ser serializadas ou “preparadas para escrita” em ficheiros binários. Para esse efeito, o desenvolvedor necessita de colocar um atributo logo acima da iniciação da classe, atributo esse: [Serializable]. Este atributo, permite “preparar” a classe para que a mesma seja escrita num ficheiro. Claro que se o atributo [Serializable] não for colocado, o IDE irá devolver um erro quando a classe for alvo de uma tentativa de serialização.

E se não for necessário guardar todos os membros de uma classe? Bom, para que tal seja possível, recordando que essa classe precisa de ser indicada como “serializável”, é crucial assignar aos elementos que não são necessários guardar, como por exemplo atributos calculáveis, a indicação [Non Serializable]. Isto indica que no momento da serialização, estes campos não são escritos em ficheiro.

De forma hómologa, abordando o tema da gravação, para o tema da carga de informação, proveniente de ficheiros, implica também a “deserialização” dos ficheiros binários. Pensemos no processo como um recipiente com água que é congelada (Serializada) e posteriormente descongelada novamente (deserializada).

## 6. Programação em camadas

A programação, seja ela em que linguagem for, necessita acima de tudo de organização. Um programa com alguma complexidade, fica muito difícil de ler e/ou manter se tudo estiver “ao molho”. Naturalmente poderiam ser usadas diferentes técnicas de organização, mas existe uma que pelo sentido que faz, bem como pela segurança adicional que fornece faz todo o sentido abordar. Falo claro da programação em camadas **N-TIER**.

N-TIER significa a divisão do nosso programa em N camadas, mediante o grupo de responsabilidade de cada grupo de classes. Por exemplo, ter as classes que guardam informação, junto às classes que interagem com o utilizador, além de gerar potenciais situações arriscadas, acabam por desvirtuar o conceito de código organizado e bem estruturado.

Para esta solução, foram usadas as seguintes tipologias de camadas, através do sistema N-TIER:

- ConsoleApp (Front End);
- BO (Business Object);
- BL (Business Logic);
- DAO (Data Access Object);

### 6.1. TIER Front End

Nesta camada, são colocadas todas as classes que fazem interação com o utilizador. Aqui consta naturalmente a classe Program, e outras classes que sejam necessárias para haver interação entre o utilizador e o programa.

Importa mencionar que nesta camada não existem métodos que guardem dados, ou cálculos ou ainda validações. Todos esses processos estão localizados em camadas mais baixas, limitadas de acesso.

### 6.2. TIER Business Object

Na programação POO, como está implícito, é necessário definir objetos. Essa definição é realizada sempre em classes que, neste modelo organizativo são mantidas numa classe denominada BO – Business Object.

Estas classes definem os objetos, quais os seus atributos, propriedades e construtores. Toda a informação, no entanto, não é gerida nesta camada, pois é uma camada que irá estar ligada diretamente ao front end, e também a outras camadas, pelo que o controlo de acesso á informação, caso a mesma fosse guardada neste nível seria “arriscado”.

### 6.3. TIER Business Logic

Um objeto é definido através das classes presentes na camada BO. Os diferentes pedidos quer de acesso a informação, quer de execução de métodos vem das classes presentes no Front End. Todo este fluxo, tem de ter um destino, que são as camadas mais baixas, no caso a camada de dados.

Mas para lá chegar é imperativo que a informação quando chegar à camada de dados esteja “em conformidade” quer com regras definidas pelo ambiente e natureza do programa, quer também por regras impostas pelas boas práticas da programação.

Para tal usam-se classes que gerem estes processos, situadas na camada de BL. Esta camada tem como objetivo albergar todas as classes que gerem toda a lógica do(s) processo(s). Faz ainda mais sentido, atendendo que toda a informação ou pedidos passam por esta camada obrigatoriamente para aceder a dados e no caminho inverso também.

Esta é uma camada muito importante pois se for bem pensada e implementada, assegura uma maior segurança a todos os dados que se encontram inacessíveis à camada superior.

### 6.4. TIER Data Access Object

A última camada implementada é a camada de dados. Nesta camada estão as classes que definem as estruturas de dados, bem como os métodos que gerem essas mesmas classes.

Estas classes estão inacessíveis de forma direta do Front End o que confere segurança à nossa aplicação. Nesta camada estão ainda os processos de gravação e de carregamento de dados.

Todos os pedidos realizados pelo front end, que passam pela camada de lógica, e pelas suas validações chegam então a esta camada, sendo então processados os dados.



## 7. Implementação

Já foi abordada a ideia para este projeto e mencionada na primeira fase. Esta implementação está longe de ser perfeita e ideal. É uma primeira abordagem ao conceito, e também uma das primeiras aplicações em **POO**. Naturalmente, com o avançar do tempo, esta implementação irá sofrer mutações, que irão redefinir conceitos, classes, etc. Nesta nova fase foram feitas várias alterações em relação ao “raciocínio” por detrás desta ideia. A aplicação de conceitos como o de N-TIER faz com que apesar de similar à implementação anteriormente concretizada, a mesma esteja bem mais organizada.

### 7.1. Classes usadas

Para a realização deste projeto, foram usadas as classes visíveis nas figuras abaixo. De reforçar que cada diagrama, corresponde a uma camada.

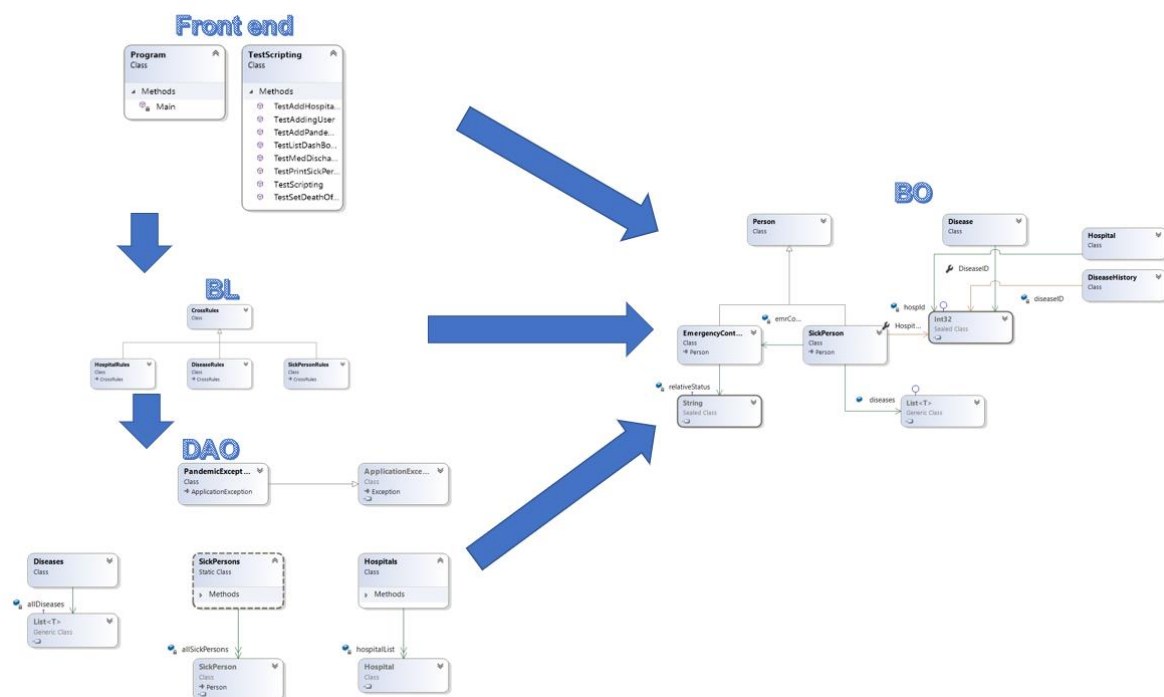


Figura 6 - Diagramas de classes por camada

No diagrama de classes, representado na Figura 6 é visível que existe uma classe “pai” do tipo **Person** que define “uma pessoa”.

As classes **SickPerson** e **EmergencyContactPerson** “herdam” de **Person**.

**SickPerson** representa uma pessoa infetada, no contexto do projeto com um vírus/doença fruto de uma epidemia/pandemia.

**EmergencyContactPerson** representa o contacto de emergência de uma pessoa que está em cuidados médicos. Naturalmente estamos a falar também de um tipo de pessoa.

A classe **ContactPerson**, não herda de **Person**. No entanto **Person** é composta por **ContactPerson**. Faz sentido atendendo que uma pessoa, tem informação de contato a si associada, como contato telefónico ou email.

Uma **SickPerson** é composta por um contato de emergência e tem um histórico de doenças, razão pela qual é composta por uma lista de objetos do tipo **DiseaseHistory**. Note-se que existe a possibilidade de determinada pessoa, que esteja doente, ter mais que uma doença.

Existe ainda uma relação para um subconjunto de classes que definem hospitais, indicando se a pessoa doente vai ficar “internada” num determinado hospital, ou se vai fazer a sua recuperação em casa.

## 7.2. Bibliotecas

Nesta implementação poderão ser usadas diversas soluções já existentes. Uma das soluções que já existe, e foi implementada, foi o recurso à biblioteca **Regular Expression Library** (REGEX)

Esta biblioteca, permite através de padrões validar o formato de determinado texto introduzido. Por exemplo, um email. Se um utilizador inserir um email do género de emailarrobaemail.pt, este email obviamente não cumpre o critério base de um endereço de correio eletrónico. Por essa razão existem padrões que em conjunto com a biblioteca REGEX permitem identificar e validar se determinado email inserido está ou não correto.

Um exemplo pode ser visualizado na Figura 6.

```
public string Email
{
    get
    {
        return email;
    }
    set
    {
        string pattern = @"^[\w+([-+.']\w+)*@\w+([-.\w+)*\.\w+([-.\w+)*])$"; //REGEX pattern for email valid
        bool b = Validate(value.ToLower(), pattern); //validates all chars of a valid email.
        email = b ? value : "no-email";
    }
}
```

Figura 7 - Exemplo ReGex

Nesta figura, é visível um padrão definido, e que permite através de um método criado de forma a poder ser reutilizado para outras “validações” Este padrão permite assim definir uma sintaxe “válida” para um email.

## 8. Conclusão

Este trabalho prático, envolveu muita pesquisa dado que é crucial assegurar que os alicerces da **POO** estão bem implementados assim como a implementação de procedimentos organizacionais de escrita de código

Creio ter seguido os pilares da **POO** que aprendi quer em ambiente letivo, quer em pesquisas na internet.

Acredito ter conseguido cimentar os conteúdos lecionados em aula, mas também despertar vontade em aprender mais, e levar este projeto a novos níveis.

Este trabalho prático teve o desafio acrescido de ser realizado com recurso a uma linguagem aprendida recentemente. Por ser uma linguagem com um conceito diferente da linguagem C, nomeadamente ser orientada a objetos, leva a que surjam dúvidas em relação a uma correta implementação. No entanto, finda esta fase, creio ter cumprido com os pontos fundamentais da **POO** e de organização das classes no projeto.

Acresce ainda a implementação de recurso a gravação e leitura de ficheiros, permitindo assim a manutenção de dados, mesmo após o término de execução do programa.

## 9. Webgrafia

- [https://pt.wikipedia.org/wiki/Classe\\_\(programa%C3%A7%C3%A3o\);](https://pt.wikipedia.org/wiki/Classe_(programa%C3%A7%C3%A3o);)
- <http://regexlib.com/>