INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE
ESCOLA SUPERIOR DE TECNOLOGIA
ENGENHARIA DE SISTEMAS INFORMÁTICOS

GONÇALO SANTOS

GESTÃO DE CENTRO DE SAÚDE: RELATÓRIO FINAL

Barcelos
2024

GONÇALO SANTOS

GESTÃO DE CENTRO DE SAÚDE: RELATÓRIO FINAL

Relatório apresentado como requisito parcial para avaliação na Unidade Curricular de **Programação Orientada a Objetos** do Curso de Engenharia de Sistemas Informáticos do Instituto Politécnico do Cávado e do Ave. O tema explorado é a **Gestão de Centro de Saúde**, com enfoque na implementação de soluções para gerir consultas, médicos e entre outros.

Prof. Dr. Luís G. Ferreira.

Barcelos
2024

Sumário

Páginas

1 Abreviaturas **6**

2 Introdução **7**

3 Domínio do Problema **8**

4 Desenho e Implementação **10**
4.1 Desenho 10
4.2 Implementação 10
4.2.1 Classes Principais 10
4.2.2 Classes Secundárias 14

5 Apêndices **17**

6 Contexto **19**

7 Desenho e Implementação **20**
7.1 Desenho 20
7.2 Implementação 20
7.2.1 Medico 20
7.2.2 Consulta: 25
7.3 Camadas do Projeto 32
7.3.1 N-Tier 33
7.3.2 MVC 35
7.4 Bibliotecas 36
7.5 Logs 37
7.5.1 Método `Log` 37
7.5.2 Método `CriarLog` 38
7.6 Testes 38

8 Problemas do Projeto **39**
8.1 MVC 39
8.2 Ficheiro Geral 39
8.3 GitHub 39
8.4 GitHub 40

9 GitHub **40**
9.1 Análise de Desempenho do Repositório 40
9.2 Impacto do Repositório 42

10 Referências **43**

Lista de Figuras

1 Diagrama de contexto 8
2 TentarAdicionarMedico 23
3 Exemplo de implementação da interface `ICompareTo` 24
4 Implementação do método `ValidarObjetoConsulta`. 26
5 Implementação do método `EncontraDisponibilidadeMedico`. 27
6 Método FindConsulta 30
7 Método ObterConsulta 31
8 Camadas do projeto 32
9 Esquema N-Tier 35
10 Método `Log`: Registo de exceções no ficheiro de log. 37
11 Método `CriarLog`: Notificação e encapsulamento do registo de exceções. 38
12 Testes Unitários 39
13 Guardar todas as classes num ficheiro 40
14 Gráficos de Clones e Visitantes do Repositório 41
15 Conteúdo Popular do Repositório 41

Lista de Tabelas

1 Códigos de erro associados à classe `Medico`. 21
2 Códigos de erro associados à classe `ListaMedicos`. 21
3 Códigos de erro associados à classe `RegrasMedicoException`. 22
4 Códigos de erro associados à classe `ConsultaException`. 28
5 Códigos de erro associados à classe `ListaConsultaException`. 29
6 Códigos de erro associados à classe `RegrasConsultaException`. 29

1 Abreviaturas

- CS – Centro de Saúde
- NIF – Número de Identificação Fiscal
- CRM – Conselho Regional de Medicina
- POO – Programação Orientada a Objetos
- DLL – Dynamic Link Libraries

2 Introdução

Este relatório apresenta o desenvolvimento de uma aplicação orientada a objetos, implementada em C#, com o objetivo de realizar a gestão eficaz de um **Centro de Saúde**. Este trabalho foi realizado no âmbito da Unidade Curricular de **Programação Orientada a Objetos (POO)** do curso de **Engenharia de Sistemas Informáticos** do IPCA.

O trabalho foi estruturado em duas fases principais:

1. **Fase 1:** Estrutura de Classes identificadas, implementação essencial das classes, estruturas de dados a utilizar, relatório do trabalho desenvolvido até à data.

2. **Fase 2:** Implementação final das classes e serviços, aplicação demonstradora dos serviços implementados, relatório final do trabalho realizado.

O projeto reforça a aplicação prática dos conceitos do paradigma orientado a objetos, destacando-se pela utilização de **herança**, **encapsulamento**, **abstração** e **modularidade**. Além disso, reflete o empenho em criar um sistema robusto e alinhado com as necessidades de um ambiente real.

3 Domínio do Problema

Neste projeto, será desenvolvido um sistema de software que visa gerir um centro de saúde. O sistema abrange o registo e gestão de entidades essenciais, como médicos, pacientes, staff, assim como a gestão de camas, consultas e diagnósticos.

Para ilustrar, apresenta-se o diagrama de contexto na Figura 1.

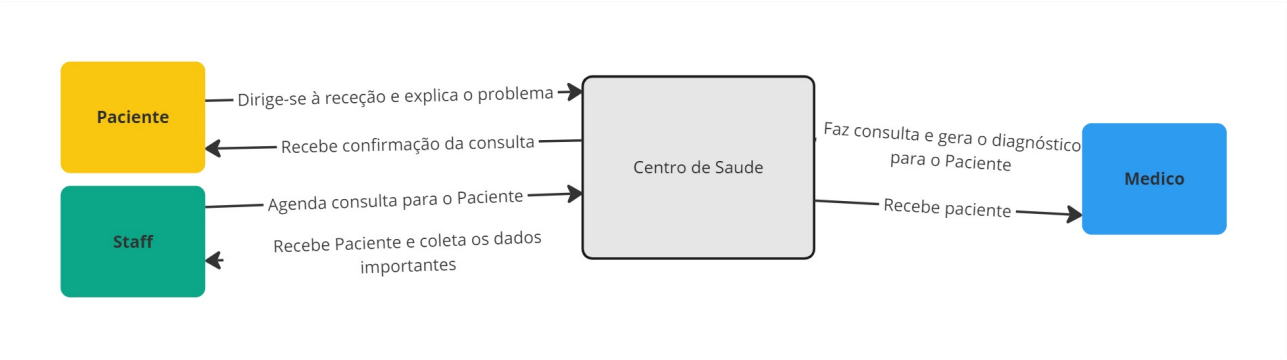

Figura 1. *Diagrama de contexto*

Fase1

4 Desenho e Implementação

4.1 Desenho

Nesta secção, mostramos o diagrama de classes que serve como base para a implementação e o próximo tópico explora as interações entre os vários elementos do sistema, tais como o Centro de Saúde, Médicos, Pacientes, Staff e Consultas, etc. A partir deste modelo de classes, avançamos para a codificação em C#, seguindo boas práticas de programação e organização de código para garantir a robustez e eficiência do sistema.

Uma vez que o diagrama é grande, solicitamos que visualize o Anexo 1.

Adicionalmente, abordaremos as decisões técnicas tomadas, a implementação dos principais métodos e a forma como estes métodos suportam as funcionalidades pretendidas, como o registo e gestão de pacientes, consultas, gestão do staff e entre outros.

4.2 Implementação

Inicialmente, definimos as classes principais, métodos e atributos necessários para cumprir os requisitos identificados. Foram utilizadas técnicas de modelação orientadas a objetos, e o sistema foi desenhado para ser modular, permitindo uma clara separação de responsabilidades e fácil manutenção.

4.2.1 Classes Principais

Estas são as classes primárias do projeto.

1. Classe CentroDeSaude (CS): É a classe principal do projeto, responsável por armazenar a maioria das informações e é o centro do projeto. A classe é composta por: `nome (string)`, `telefone (string)`, `morada (string)`, `staffs (Staffs)`, `medicos (Medicos)`, `pacientes (Pacientes)`, `camas (Camas)`, `consultas (Consultas)`.

É possível criar vários `CS's` e definir diferentes médicos, pacientes, staff, camas e consultas. A classe irá contar com alguns métodos:

- **AdicionarMedico**: Método responsável por criar um Médico e adicioná-lo à lista de Médicos do CS específico. Leva como parâmetros o nome do Médico, a data de nascimento, o NIF, a especialidade, o CRM e a morada.
- **ListarMedicos**: Método responsável por listar todos os Médicos do CS.
- **AdicionarPaciente**: Método cuja função é criar um Paciente e adicioná-lo à lista de Pacientes do respetivo CS. Tem como parâmetros o nome do Paciente, a data de nascimento, o NIF, a morada e o número de telefone.
- **ListarPacientes**: Método que lista todos os pacientes de um determinado CS.
- **AdicionarStaff**: Método responsável por criar um Staff e adicioná-lo à lista de Staffs do CS específico. Leva como parâmetros o nome do Staff, a data de nascimento, o NIF, a morada, o número do Staff e a categoria.
- **ListarStaff**: Método responsável por listar todos os Staffs do CS.
- **CriarCama**: Método que cria um valor de camas, indicadas por parâmetro, e são guardadas na lista respetiva de Camas do CS.

- **ListarCamas**: Método responsável por listar todas as Camas do CS específico, mostrando o seu `número`, se estão ocupadas e, se sim, por quem.
- **EncontrarCamaDisponivel**: Este método tem duas funções, se receber um valor por parâmetro, vai procurar qual é a cama que corresponde ao valor indicado, se encontrar devolve a Cama, se não encontrar ou se não for indicado nenhum valor como parâmetro, o método tem como objetivo procurar na lista de camas qual é a primeira Cama disponível e se encontrar, devolve a Cama disponível.
- **ListarConsultas**: Este método tem duas funções, se receber alguma lista de Consultas por parâmetro, lista essa lista de consultas indicada, se não, lista todas as Consultas registadas no CS.
- **AgendarConsulta**: Neste método é dos métodos principais do projeto, é nele que se concentram quase todas as classe existentes, recebe como parâmetros, data de inicio da consulta e a data do fim da mesma, o NIF do Paciente, o CRM do Medico, o numero de Staff do Staff que estiver na receção, este staff vai ser o responsável por marcar e validar a consulta, e o tipo de Consulta a ser realizada, esta pode ser do seis tipos, planeamento familiar ou saúde adultos ou saúde materna ou saúde infantil ou recurso/reforço ou teleconsulta. É neste método que ele realiza uma procura nas listas para encontrar as devidas entidades e de seguida criar a consulta.
- **AtualizarConsulta**: Este método recebe como argumento um id de uma Consulta uma descrição feita a partir do Médico, esta descrição é a descrição do diagnóstico feito pelo Médico, e uma possível data de fim de internamento. O método procura na lista de consultas a consulta com o mesmo id, depois de encontrar, vai atualizar o estado da Consulta através do método AtualizarConsulta(), se não tiver descrição, atualiza para Consulta não realizada, se tiver descrição, cria um Diagnóstico e anexa-o à consulta através da função AdicionarDiagnostico(), de seguida, se tiver uma data de fim de internamento, o paciente é internado através da função AdicionarPaciente(), aonde encontra uma Cama Disponível para ser internado.
- **AgendarExame**: Quem agenda um exame é um médico, caso um paciente queira agendar um exame, ele terá que primeiro marcar uma consulta com um médico especializado e só depois poderá, se necessário, efetuar o exame que pretende. Para agendar um exame é necessário o CRM do Médico, uma vez que será ele quem vai marcar o exame, o idConsulta (int), o tipo de exame (string), o numeroStaff (int) que será o responsável por fazer o exame e a data em que o exame será marcado (DateTime). Através do idConsulta, o CRM e o numeroStaff conseguimos ter acesso à Consulta, ao Medico e ao Staff, através das suas respetivas funções de procura. Depois, são validados todos os atributos, se foram ou não encontrados, depois disso, é verificado se o Staff tem a categoria "Exames", se não tiver, tem que ser alocado outro staff que tenha essa categoria, caso seja tudo aprovado, é então agendada a consulta através do método AdicionarExame(), que recebe o **tipo** de exame, o **Paciente**, o **Medico**, o **Staff** e a **data**.
- **RealizarExame**: Para encontrar o exame a fazer, o paciente necessita fornecer o seu **número** de paciente e a **data** em que o exame, isto porque um paciente só tem um exame marcado numa hora, não é possível o mesmo paciente ter dois exames na mesma hora. O método recebe como argumento o número de paciente, a data do exame e o resultado do exame, depois de ser confirmado o exame, é atualizado o **resultado** ao respetivo exame, através da função RealizarExame()

2. Classe Pessoa: Esta é a classe pai das classes Medico, Staff e Paciente, tem como atributos o nome (string), a data de nascimento (DateTime), o NIF (int) e a morada (Morada). Não tem em si definido nenhum método nem override, apenas tem definidas as propriedades e dois construtores, um vazio e outro com os parâmetros correspondentes aos atributos.

3. Classe Medico: Esta classe é filha da classe Pessoa, herda os seus atributos e ainda tem mais dois, o CRM (string) e a especialidade (Especialidade). Esta classe é responsável por criar um Médico por vez através do construtor definido, tem as propriedades definidas e não tem nenhum método nem override definido.

4. Classe Staff Esta classe é filha da classe Pessoa, herda os seus atributos e ainda tem mais dois, o numeroStaff (int) e a categoria (Categoria). Esta classe é responsável por criar um Staff por vez através do construtor definido, tem as propriedades definidas e não conta com nenhum método nem override definido.

5. Classe Paciente: Esta classe é filha da classe Pessoa, herda os seus atributos e ainda tem mais dois, o numeroStaff (int) e a categoria (Categoria). Esta classe é responsável por criar um Staff por vez através do construtor definido, tem as propriedades definidas e não conta com nenhum método nem override definido.

6. Classe Morada: Esta classe foi criada com o intuito de ajudar na organização das moradas de cada entidade, é responsável por armazenar o **CP** (int), a **morada** (string), a **cidade** (string) e o **numeroPorta** (int). Apenas estão definidos dois construtores, um deles vazio e outro com os atributos correspondentes, também foi reescrito a função .ToString(), para assim fazer uma listagem organizada de cada atributo e facilitar a visualização da informação.

7. Classe Especialidade: Esta classe ainda não está 100% elaborada, foi pensado em transformar a classe num ENUM, porem, como um CS poderia acrescentar novas especialidades a novos ou antigos médicos, foi tomada a decisão de criar uma classe, para já, a classe apenas guarda o nome da especialidade (string), mas serão adicionados novos atributos no futuro. A classe apenas conta com um construtor com o atributo correspondente, e assim como a devida propriedade do atributo.

8. Classe Categoria: A classe categoria é responsável por organizar as funções de cada staff, uma vez que podem ser criadas necessidades a cada dia, foi optada por uma classe para as categorias, em vez de um ENUM. A classe é responsável por guardar o nome da **categoria** (string) e o **nível hierárquico** do staff (ENUM NivelHierarquico), o NivelHierarquico pode ser estagiário, júnior ou sénior. Um caso prático, entra um novo staff no CS, que irá ser secretário, é então atribuída a categoria de "Secretario" a nível "júnior" e assim adiante consoante a experiência do staff. A classe conta com um construtor que guardará os atributos respetivos, as devidas propriedades de cada atributo e também um override do método ToString(), para facilitar a visualização dos atributos da classe.

9. Classe Cama: Esta classe é responsável por criar uma cama de cada vez, para isso é necessário o **número** da cama (int), **ocupada** (bool) e **diagnostico** (Diagnostico), nesta classe não são feitas nenhumas operações a não ser a criação do objeto através do construtor com todos os atributos e as propriedades definidas para cada atributo.

10. Classe Consulta: A classe Consulta é umas das classes principais pois é a classe que irá envolver quase todas as classes do projeto, tem com atributos **totalConsultas** (static int), usado para definir automaticamente os ids das consultas e saber quantas consultas existem, o **idConsulta** (int), a **dataInicio** da consulta (DateTime), a **dataFim** da consulta (DateTime), o **paciente** (Paciente), o **medico** (Medico),

o **staff** (Staff), o **diagnostico** (Diagnostico), o **tipoConsulta** (ENUM TIPOCONSULTA) e o **estado** (ENUM ESTADO). Começando pelo tipoConsulta, como já foi explicado em cima, ele pode ser de seis tipos, planeamento familiar ou saúde adultos ou saúde materna ou saúde infantil ou recurso/reforço ou teleconsulta. O estado pode ser de 4 tipos, agendada (momento em que é marcada a consulta), concluída (quando o paciente sai do internamento ou quando é definido pelo Médico), não realizada (quando o paciente não veio à consulta e então não se realizou) ou em processo (quando é gerado o diagnostico e o paciente está ainda em internamento ou durante o tempo da consulta). Em relação a passar os objetos todos em vez de apenas um campo, como isto é, a classe principal, ela já vai receber um objeto inteiro, antecipadamente será só solicitado o identificador de cada identidade como visto anteriormente na classe CentroDeSaude. Esta classe a pesar de ter os dois ENUM's definidos conta também com dois construtores, um vazio e outro com os atributos correspondentes, à exceção do idConsulta que é incrementado automaticamente dentro do construtor. Também estão definidas todas as propriedades para os atributos. E por ultimo, foram criados dois métodos, o AdicionarDiagnostico() que recebe com parâmetro a descrição do diagnostico e uma possível data de fim do internamento, caso seja necessário, dentro do método ele cria um novo diagnostico, passa como argumento o objeto atual a descrição e a data final de internamento, depois de criado e armazenado na consulta o diagnostico é alteado o estado da consulta para em processo caso a data fim internamento tenha valor, uma vez que o paciente foi internado ou alterado para concluída caso a data não tenha valor, por isso o paciente depois da consulta foi embora. No método AtualizarConsulta() que recebe como parâmetro o estado, simplesmente é atualizado o estado da consulta para o recebido no parâmetro. Foi implementado adicionalmente o método AdicionarExame() que recebe como parâmetro o **tipo** de exame (string), o **paciente** (Paciente), o **medico** (Medico), o **staff** (Staff) e a **data** do exame (DateTime), este método simplesmente cria um exame com os parâmetros referidos e adiciona aos atributos o exame criado.

11. Classe Diagnóstico: A classe Diagnóstico está ligada diretamente à classe Consulta, uma vez que um diagnostico só pode existir se o paciente for visto por um médico e isso acontece depois ser feita a consulta. Esta classe tem como atributos **consulta** (Consulta), a consulta aonde será inserido o diagnostico, a **descrição** (string), **fimInternamento** (DateTime), que vai definir se o Paciente precisa de ser internado ou não, dependendo se recebe ou não valor, e a **dataDiagnostico** (DateTime?) que será correspondente à data de criação do diagnostico. Esta classe conta com dois construtores, um vazio e outro com os atributos respetivos, para além de não ter nenhum método e nenhum override, tem propriedades para todos os atributos.

12. Classe Exame: Esta classe foi pensada para apenas ser utilizada durante uma consulta, caso seja necessário realizar algum exame, é então agendado um exame por parte do Médico. Para agendar um exame é necessário o **tipo** de Exame (string), o **paciente** (Paciente), o **medico** (Medico), o **staff** que vai realizar o exame (Staff), **data** do exame (DateTime), a classe conta também com o **idExame** (int), o **estado** (ENUM ESTADO) o **resultado** (string), e um contador do número de Exames (static int) realizado no CS. Aqui só estão definidas as propriedades para cada atributo, dois construtores, um vazio e outro com os atributos necessários para agendar um exame e um método chamado RealizarExame(), que tem como argumentos o **resultadoExame** (string), ele alterar o resultado do objeto atual para o resultado passado por parâmetro e o estado do exame como "Concluída".

4.2.2 Classes Secundárias

Entende-se por classes secundárias classes criadas para o auxílio durante a implementação do projeto em C#, uma vez que todos os CS's permitem fazer uma gestão de consultas, staffs, médicos, pacientes, camas e entre várias outras atividades e entidades relacionadas, foram criadas 4 classes para registar e gerir todas as **Consultas**, **Staffs**, **Pacientes**, **Medicos** e **Camas**.

- **Class Consultas**:

  A classe Consultas é composta por um atributo consultas do tipo `List<Consulta>`, isto significa que será guardada uma lista de consultas com todas as 'consultas' no singular criadas. Este método é constituído por um construtor que cria a `List` não tem propriedades, mas tem vários métodos:

  * **CriarConsulta**: Este método é responsável por criar uma Consulta e adicioná-la à lista de Consultas. Para criar uma consulta é necessário a `dataInicio (DateTime)`, a `dataFim (DateTime)`, o `paciente (Paciente)`, o `medico (Medico)`, o `staff (Staff)` e o `tipoConsulta (ENUM TIPOCONSULTA)`.
  * **EncontrarConsultasPaciente**: Caso um Paciente queira saber todas as suas consultas, é possível através deste método. Basta só dar o seu NIF, o método cria uma lista local de Consultas e procura através do `FindAll` todas as consultas em que o NIF do Paciente seja correspondente. Depois disso, se encontrar alguma consulta ou várias, retorna a lista, senão, retorna `null`.
  * **ListarConsultas**: Este método tem duas funções. Ambas com o mesmo objetivo, mostrar todas as consultas. Pode ser através da chamada do método sem argumentos, que vai listar todas as consultas registadas no CS ou através de uma lista de consultas passada por parâmetro. O método irá listar todos os atributos registados nas consultas passadas.
  * **EncontrarConsulta**: Neste método é passado um id de uma consulta por parâmetro. O método procura em todas as consultas e retorna uma variável do tipo Consulta e retorna a mesma.
  * **EncontrarConsultaExame**: À semelhança do método `EncontrarConsulta`, este método também devolve uma variável do tipo Consulta. Mas neste caso é necessário o NIF do paciente e a data do Exame. O método utiliza estes dois argumentos para procurar uma consulta que o paciente seja o mesmo e que a data seja igual. Esta função também pode validar se a consulta foi realizada com sucesso ou não e retorna `null` em caso de seja.

- **Class Staffs**:

  A classe Staffs é composta por um atributo staff do tipo `List<Staff>`, isto significa que será guardada uma lista de staff com todos os 'staffs' no singular criados. Este método é constituído por um construtor que cria a `List`, não tem propriedades, mas tem vários métodos:

  * **AdicionarStaff**: Este método é responsável por criar e guardar um Staff. Para isso é necessário o `nome (string)`, a `data de nascimento (DateTime)`, o `NIF (int)`, o `número identificador de Staff (int)`, a `morada (Morada)` e a `categoria (Categoria)`.
  * **EncontrarStaff**: Através do `número identificador de staff`, este método procura na lista de staffs o staff com o mesmo número e se encontrar, retorna o objeto de Staff.
  * **RemoverStaff**: Este método é responsável por remover um staff da lista de staffs. Recebe então o seu `número identificador`, procura na lista de staffs o staff e se conseguir encontrar, remove-o da lista.

    - **ListarStaff**: Este método lista todos os staffs registados no CS e lista todos os atributos da classe.

- **Class Pacientes**:

  A classe Pacientes é composta por um atributo pacientes do tipo `List<Paciente>`, isto significa que será guardada uma lista de pacientes com todos os 'pacientes' no singular criados. Este método é constituído por um construtor que cria a `List`, não tem propriedades, mas tem vários métodos:

    - **AdicionarPaciente**: Para ser criado um Paciente, é necessário o `nome (string)`, a `data de nascimento (DateTime)`, o `NIF (int)`, o `número de telefone (int)`, a `morada (Morada)`. Depois de ter estes atributos todos, é criado o Paciente e se for possível, é então adicionado à lista de pacientes.
    - **EncontrarPaciente**: Através do NIF do paciente, é possível encontrar um paciente. O método procura então na lista de todos os pacientes o paciente com o mesmo NIF e devolve, caso encontre, o objeto total.
    - **RemoverPaciente**: Este método é responsável por remover um paciente da lista de pacientes. Recebe então o seu NIF, procura na lista de pacientes o paciente e se conseguir encontrar, remove-o da lista.
    - **ListarPacientes**: Este método lista todos os pacientes registados no CS e lista todos os atributos da classe.

- **Class Medicos**:

  A classe Medicos tem como seu encargo guardar todos os médicos do CS, então tem como seu atributo uma `List<Medico>`, uma lista de 'medicos' no singular. Este método é constituído por um construtor que cria a `List`, não tem propriedades, mas tem vários métodos:

    - **AdicionarMedico**: Para ser criado um Medico, é necessário o `nome (string)`, a `data de nascimento (DateTime)`, o `NIF (int)`, a `morada (Morada)`, a `especialidade (Especialidade)` e o `CRM (string)`. Depois de ter estes atributos todos, é verificado se já existe algum Medico com o mesmo CRM. Se não tiver, é criado o Medico e se for possível, é então adicionado à lista de medicos.
    - **EncontrarMedico**: Através do CRM do medico, é possível encontrar um medico. O método procura então na lista de todos os medicos, o medico com o mesmo CRM e devolve, caso encontre, o objeto total.
    - **RemoverMedico**: Este método é responsável por remover um medico da lista de medicos. Recebe então o seu CRM, procura na lista de medicos o medico e se conseguir encontrar, remove-o da lista.
    - **ListarMedicos**: Este método lista todos os medicos registados no CS e lista todos os atributos da classe.
    - **AdicionarCama**: Este método recebe um número identificador de uma cama e se deseja, tenta criar uma cama, se conseguir, adiciona a Cama à lista de camas e acaba.
    - **LiberarCama**: Este método é utilizado depois de uma paciente receber alta, recebe então o número da cama, encontra a cama através do método `EncontrarCama()`, atualiza o valor do atributo Ocupada para false e o Diagnostico para false.
    - **EncontrarCama**: Através do número da cama, é possível encontrar uma cama. O método procura então na lista de todas as camas, a cama com o mesmo número e devolve, caso encontre, o objeto total.

* **EncontrarCamaDisponivel**: Este método é responsável por devolver uma Cama que esteja livre. Existem duas opções, encontrar uma cama disponível ou, através do número da cama que será passado por parâmetro, verificar se essa cama está ocupada e retornar-la. Começa por verificar se o número da cama é diferente de 0, se for, procura a cama através da função `EncontrarCama()`, se não encontrar, retorna `null`. Se encontrar, verifica se a cama está ocupada, se não estiver, retorna essa Cama e se estiver sai do condicional e encontra em todas as camas a primeira cama disponível, caso encontre, retorna essa Cama.
* **RemoverCama**: Este método é responsável por remover uma cama da lista de camas. Recebe então o seu número, procura na lista de camas a cama correspondente e se conseguir encontrar, remove-a da lista.
* **ListarCamas**: Este método lista todas as camas registadas no CS e lista todos os atributos da classe.

5 Apêndices

– **Dicionário de Classes**:

1. Cama - Representa uma cama específica, aonde será alocado um paciente.
2. Camas - Gera uma lista de camas dentro do centro de saúde.
3. Categoria - Enumera as diferentes categorias dos funcionários, como secretario, exame, entre outros.
4. CentroDeSaude - Representa o centro de saúde, englobando consultas, exames e outros serviços/entidades.
5. Consulta - Contém informações de uma consulta entre paciente e médico, incluindo datas, diagnósticos, exame, entre outros.
6. Consultas - Gera uma lista de objetos do tipo consulta.
7. Diagnostico - Regista o diagnóstico feito durante uma consulta, incluindo a data e a descrição, caso necessário.
8. Especialidade - Enumera as diferentes especialidades dos médicos, como cardiologia, pediatria, etc.
9. Exame - Representa um exame indicado por um médico associado a uma consulta.
10. Medico - Guarda informações sobre um médico, como nome e especialidade, etc.
11. Medicos - Gera uma lista de médicos.
12. Morada - Representa um endereço, incluindo detalhes como rua, cidade e código postal.
13. Paciente - Representa um paciente com informações pessoais e de contacto.
14. Pacientes - Gera uma lista de objetos do tipo paciente.
15. Pessoa - Classe base para Staff, Medico e Paciente, contendo atributos básicos de uma pessoa.
16. Staff - Representa os recursos humanos do centro de saúde.
17. Staffs - Gera uma lista de objetos do tipo staff.

Fase2

6 Contexto

Após apresentar a Fase 1 do projeto ao professor responsável, Luís G. Ferreira, no seu gabinete, recebi um feedback bastante incisivo. O que inicialmente considerei como uma implementação sólida não estava, de facto, alinhado com os objetivos e os conceitos fundamentais abordados na disciplina. A complexidade desnecessária e a multiplicidade de classes criadas não refletiam uma aplicação prática e aprofundada dos princípios de Programação Orientada a Objetos (POO).

Perante esta situação, após uma análise detalhada e uma discussão produtiva com o professor, foi-me sugerido reiniciar o projeto. O objetivo seria focar apenas em explorar uma única classe do projeto, de modo a aplicar todos os conceitos abordados de forma sistemática e rigorosa. Neste contexto, a classe "Medico"foi escolhida como objeto central de estudo. Através dela, foram criadas diversas subclasses, permitindo uma exploração mais abrangente das técnicas de POO, como herança, encapsulamento e polimorfismo, entre outras.

Após a conclusão desta etapa e validação dos resultados, a metodologia foi expandida para as outras classes, permitindo uma implementação gradual e consistente, respeitando os mesmos princípios e desenvolvimento.

Agora será apresentado o trabalho realizado através da classe "Medico".

7 Desenho e Implementação

7.1 Desenho

Nesta secção, apresentamos o diagrama de classes que constitui a base para a implementação do sistema, detalhando as interações entre os diversos elementos que o compõem. Este modelo serve como ponto de partida para a codificação em C#, na qual foram seguidas as melhores práticas de programação e organização de código, com o objetivo de assegurar a robustez e eficiência do sistema.

Dado que o diagrama possui uma dimensão significativa, remetemos o leitor para o Anexo 2, onde poderá ser consultado com maior detalhe.

Adicionalmente, serão exploradas as decisões técnicas tomadas ao longo do desenvolvimento, bem como a implementação dos métodos principais e o modo como estes suportam as funcionalidades previstas.

7.2 Implementação

Durante esta revisão da Fase 2, foram alterados alguns atributos e metódos das classes, em baixo serão apresentadas as justificações para isso acontecer.

7.2.1 Medico

Para a classe Medico, foram alterados os atributos, o CRM passou a ser o identificador do objeto. Para o bom funcionamento, foram criadas mais 5 subclasses, todas com funções distintas, miniMedico, validações, exceções, lista e regras.

Para além disso, a classe conta com 4 métodos, '**CriaMedico**', usado para criar um novo objeto, '**EditaMedico**', usado para editar atributos de um objeto já criado, interface para '**CompareTo**', usado para obter qual é o maior nome do médico e override para '**ToString**', usado para visualizar melhor os atributos durante o debug.

MiniMedico: A classe `MiniMedico` foi desenvolvida com base nos conceitos apresentados nas aulas. A necessidade de restringir o acesso à informação para utilizadores sem permissões adequadas resultou na criação desta classe, cujo objetivo principal é reduzir a quantidade de dados exibidos. Esta classe contém apenas dois atributos: o CRM do médico e o respetivo nome.

Mais justificações

A utilização desta subclasse é fundamental para garantir a privacidade e segurança dos dados. Ao limitar a informação partilhada, evitamos expor dados sensíveis ou desnecessários a utilizadores sem autorização. Além disso, proporciona maior eficiência no acesso e manipulação de dados. Por exemplo, caso seja necessário apenas o nome do médico, não faz sentido transferir todo o conjunto de dados associados ao objeto `Medico`. A subclasse `MiniMedico` permite, assim, otimizar o processo ao disponibilizar apenas os atributos estritamente necessários para o contexto em questão.

Validações: Esta subclasse foi desenvolvida com o propósito de validar todos os atributos da classe `Medico`. Abrange validações relacionadas com diferentes aspetos, como valores, tamanhos, datas e objetos. A sua implementação visa simplificar o trabalho do programador, permitindo que, sempre que seja necessário validar um atributo, se possa simplesmente chamar a subclasse e o método de validação correspondente.

A subclasse `Validações` é declarada como `static`, uma vez que não é necessário criar instâncias da mesma. O objetivo é fornecer métodos utilitários de validação, promovendo uma maior organização e

reutilização de código. As validações estão estruturadas com base em códigos de erro que, posteriormente, são tratados e apresentados nas `Exceptions`, garantindo uma gestão eficiente e clara de erros.

Exceções: As exceções foram implementadas individualmente para cada subclasse, uma vez que cada uma pode ter os seus próprios casos específicos de erro. Estas foram abordadas durante as aulas e têm como principal objetivo sinalizar e reportar falhas que possam ocorrer durante a execução do programa.

Cada falha está associada a um código de erro que permite identificar o problema de forma clara e precisa. No entanto, caso o código de erro não seja reconhecido, será apresentada uma mensagem genérica de erro desconhecido. Relativamente à classe `Medico`, foram implementadas exceções tanto para a lista de médicos como para a própria classe e para as regras, garantindo uma gestão robusta e estruturada de potenciais erros.

MedicoException: As exceções associadas à classe `Medico` estão organizadas por códigos de erro, conforme apresentado na Tabela 1.

Código de Erro	Descrição
-1	Nome Inválido
-2	Idade Incorreta
-3	NIF Inválido
-4	Morada Inválida
-5	CRM não existente
-6	Especialidade Inválida
-7	Objeto Medico nulo
-8	Objeto MiniMedico nulo

Tabela 1. *Códigos de erro associados à classe `Medico`.*

ListaMedicosException: As exceções relacionadas com a lista de médicos estão organizadas na Tabela 2.

Código de Erro	Descrição
-11	Lista nula
-12	Lista vazia
-13	CRM duplicado
-14	CRM não existente

Tabela 2. *Códigos de erro associados à classe `ListaMedicos`.*

RegrasMedicoException: As exceções relacionadas com as regras de médicos estão organizadas na Tabela 2.

Código de Erro	Descrição
-21	Sem Permissão

Tabela 3. *Códigos de erro associados à classe `RegrasMedicoException`.*

Lista de Medico: Esta subclasse destaca-se como uma das mais importantes dentro do sistema, uma vez que, sempre que existe um objeto de negócio, este deve ser armazenado de forma estruturada. Neste caso, os objetos relacionados são armazenados na lista gerida pela subclasse `ListaMedicos`. Esta lista integra diversos métodos que facilitam a manipulação dos dados, permitindo uma implementação eficiente e organizada.

Todos os métodos da `ListaMedicos` são desenvolvidos seguindo, no mínimo, três princípios fundamentais da metodologia SOLID, garantindo que o código seja modular, reutilizável e fácil de manter. A seguir, apresentam-se os métodos implementados nesta subclasse, acompanhados de uma breve descrição da sua funcionalidade:

`ExisteCRM: bool` | Verifica se um CRM específico já está registado na lista. Retorna `true` caso o CRM exista e `false` caso contrário.

`AdicionarMedico: int` | Adiciona um novo médico à lista. Retorna um código de erro caso a operação falhe ou `1` em caso de sucesso.

`AtualizarMedico: int` | Atualiza as informações de um médico já existente na lista, com base no CRM. Retorna um código de erro caso a operação falhe ou `1` em caso de sucesso.

`RemoverMedico: int` | Remove um médico da lista com base no CRM. Retorna `1` em caso de sucesso ou um código de erro caso não encontre o médico.

`ListarMedicos: void` | Lista todos os médicos registados na lista, mostrando os atributos relevantes para consulta. **PS:** Usado apenas para testes

`FindMedico: Medico` | Procura um médico na lista através do CRM e retorna o objeto completo do tipo `Medico`.

`ObterMedico: MiniMedico` | Retorna um único médico, apresentando apenas os atributos limitados da classe `MiniMedico`.

`ObterMedicoFiltro: List<Medico>` | Retorna uma lista de médicos com base em critérios de filtragem definidos. Inclui toda a informação de cada médico.

`ObterMiniMedicoFiltro: List<MiniMedico>` | Retorna uma lista de médicos, mas com informações reduzidas, limitando-se ao `CRM` e `nome`. Este método é útil para restringir o acesso à informação total.

`OrganizarMedicosAlfabeticamente: int` | Ordena os médicos da lista alfabeticamente com base no `nome`. Retorna `1` em caso de sucesso ou um código de erro se a operação falhar.

`GuardarMedicosFicheiro: int` | Guarda a lista de médicos num ficheiro binário. Retorna `1` em caso de sucesso ou um código de erro se a operação não for bem-sucedida.

`LerMedicosFicheiro: int` | Lê os dados de médicos de um ficheiro binário e carrega-os para a lista. Retorna `1` em caso de sucesso ou um código de erro caso a leitura falhe.

Como foi possível analisar, estão presentes métodos que controlam a informação que é fornecida. Por exemplo, os métodos `ObterMedicoFiltro` e `ObterMiniMedicoFiltro`. Enquanto o primeiro disponibiliza todos os dados associados ao objeto `Medico`, o segundo fornece apenas informações limitadas, como o `CRM` e o `nome`, de modo a promover a segurança e privacidade dos dados.

Regras de Médicos: Com o papel mais importante no sistema, surgem as regras, cuja responsabilidade é controlar o acesso da primeira camada (apresentação e solicitação de dados) à terceira camada (armazenamento de dados). Estas regras são implementadas de forma a assegurar que apenas utilizadores com permissões adequadas possam realizar operações específicas no sistema.

Neste caso, foi implementado um algoritmo do tipo `switch-case`, onde todos os métodos recebem como um dos parâmetros a permissão do utilizador (`PERMISSOES`), que pode assumir três valores: `None`, `Low` ou `High`. Cabe ao programador definir quais permissões têm acesso a cada funcionalidade.

Abaixo, é analisado o método `TentarAdicionarMedico`, representado na Figura 2.

```
public static int TentarAdicionarMedico(PERMISSOES perm, Medico medico)
{
    switch (perm)
    {
        case PERMISSOES.None:
            return -21;
        case PERMISSOES.Low:
        case PERMISSOES.High:
            try
            {
                int res = Medicos.AdicionarMedico(medico);
                if (res != 1)
                    return res;
            }
            catch (ListaMedicosException lme)
            {
                throw lme;
            }
            return 1;
        default:
            return -21;
    }
}
```

Figura 2. *TentarAdicionarMedico*

Descrição do Código:

- **Parâmetros:** O método recebe dois parâmetros:
  - `perm`: Representa a permissão do utilizador (`PERMISSOES`).
  - `medico`: Objeto do tipo `Medico` que se pretende adicionar à lista.
- **Estrutura** `switch-case`**:** A permissão do utilizador é avaliada de acordo com os seguintes casos:
  - `PERMISSOES.None`: O acesso é negado, e o método retorna o código de erro `-21`, indicando que o utilizador não tem permissão para adicionar um médico.
  - `PERMISSOES.Low ou High`: É permitido prosseguir com a operação de adição. O método tenta adicionar o objeto `medico` à lista, invocando o método `AdicionarMedico` da classe `Medicos`.
    - Caso o método `AdicionarMedico` retorne um código de erro diferente de `1`, o mesmo é devolvido ao utilizador.
    - Em caso de sucesso, o método retorna `1`, indicando que o médico foi adicionado corretamente.

  - `default`: Qualquer permissão inválida também retorna o código de erro `-21`.

- **Tratamento de exceções:** Caso ocorra uma exceção do tipo `ListaMedicosException`, a mesma é lançada novamente para ser tratada noutra parte do sistema.

Objetivo do Método: O método `TentarAdicionarMedico` foi desenhado para garantir que apenas utilizadores autorizados possam adicionar novos médicos ao sistema. Este mecanismo de controlo por permissões reforça a segurança e a integridade do sistema, prevenindo acessos ou alterações não autorizadas.

ICompareTo: Para facilitar a comparação entre objetos da classe `Medico`, foi criada a interface `ICompareTo`, que implementa o método `CompareTo`. Este método tem como objetivo comparar dois objetos do tipo `Medico`, avaliando os seus nomes em ordem alfabética. A implementação está descrita a seguir e é detalhada para garantir que os objetos sejam ordenados de forma consistente e precisa.

```
public int CompareTo(Medico outroMedico)
{
    int res = ValidarMedico.ValidarObjetoMedico(outroMedico);
    if (res != 1)
        throw new MedicoException(res);

    string nome1 = this.Nome.ToLower();
    string nome2 = outroMedico.Nome.ToLower();

    int minLength;

    if (nome1.Length > nome2.Length)
    {
        minLength = nome1.Length;
    }
    else
    {
        minLength =nome2.Length;
    }

    for (int i = 0; i < minLength; i++)
    {
        if (nome1[i] < nome2[i])
            return -1; // `this.Nome` é maior
        else if (nome1[i] > nome2[i])
            return 1;  // `outroMedico.Nome` é maior
    }

    if (nome1.Length < nome2.Length)
        return -1; // Mais curto vem primeiro
    else if (nome1.Length > nome2.Length)
        return 1;  // Mais longo vem depois

    return 0;
}
```

Figura 3. *Exemplo de implementação da interface `ICompareTo`*

Explicação do Código:

- **Validação do Objeto:** O método começa por validar o objeto `outroMedico`, utilizando o método `ValidarObjetoMedico`. Caso o objeto não seja válido, é lançada uma `MedicoException` com o respetivo código de erro.

- **Comparação de Nomes:** Os nomes dos médicos, `this.Nome` e `outroMedico.Nome`, são convertidos para minúsculas (`ToLower()`) para assegurar uma comparação insensível a maiúsculas/minúsculas.
- **Determinação do Comprimento Mínimo:** O comprimento do menor nome é calculado (`minLength`) para evitar problemas ao aceder índices fora do intervalo em nomes de diferentes tamanhos.
- **Comparação Caractere a Caractere:** Um loop percorre os caracteres dos dois nomes até o comprimento mínimo, comparando-os. O método retorna:
  - `-1` se o nome do objeto atual (`this.Nome`) vier antes alfabeticamente.
  - `1` se o nome do `outroMedico` vier antes.
- **Comprimento dos Nomes:** Caso os caracteres até o comprimento mínimo sejam iguais, o método verifica o comprimento dos nomes:
  - `-1` se o nome do objeto atual (`this.Nome`) for mais curto.
  - `1` se o nome do `outroMedico` for mais curto.
- **Resultado Final:** Se os nomes forem idênticos em conteúdo e comprimento, o método retorna `0`, indicando que são iguais.

7.2.2 Consulta:

A classe `Consulta` é responsável pelo registo de todas as consultas realizadas nos Centros de Saúde (CS). Em relação à implementação realizada na fase anterior, foram efetuadas alterações nos seguintes atributos: o paciente passou a ser identificado apenas pelo seu `nif`, em vez de incluir o objeto completo; o médico passou a ser identificado pelo `crm`; o staff pelo `numeroStaff`; e o diagnóstico pelo `idDiagnostico`. Estas alterações foram essenciais para evitar o transporte desnecessário de todas as informações associadas aos objetos. Em conformidade com as boas práticas de programação, apenas os identificadores dos objetos são agora utilizados, promovendo maior eficiência e segurança no tratamento dos dados.

Tal como na classe apresentada anteriormente, foram desenvolvidas mais cinco subclasses associadas à classe `Consulta`: `MiniConsulta`, `Validações`, `Exceções`, `Lista` e `Regras`. Além disso, foram implementados dois novos métodos principais:

`CriaConsulta`: responsável pela criação de um novo objeto da classe `Consulta`. `AdicionarDiagnostico`: permite associar um diagnóstico específico a uma consulta já existente. **Nota:** Como a primeira classe foi definida e estruturada de forma robusta, as classes subsequentes seguem a mesma lógica estrutural, tanto no que diz respeito aos métodos como às subclasses. Assim, é esperado que alguns trechos de texto sejam semelhantes.

MiniConsulta: Com o avanço no desenvolvimento da aplicação, surgiu a necessidade de criar uma versão simplificada da classe `Consulta`, denominada `MiniConsulta`. Esta subclasse reúne apenas os atributos essenciais, a saber:

- `idConsulta`: identificador único da consulta.
- `crm`: identificador do médico responsável pela consulta.
- `nif`: identificador do paciente associado à consulta.
- `dataI`: data de início da consulta.
- `dataF`: data de término da consulta.

Validações: As subclasses `ValidarConsulta e ValidarListaConsulta` foram desenvolvidas com o objetivo de validar todos os atributos da classe `Consulta e da Lista`. Estas subclasses abrangem validações relacionadas com diversos aspetos, tais como valores, intervalos de datas, identificadores (como o `idConsulta`, `crm`, e `nif`) e outros atributos essenciais. A implementação visa facilitar o trabalho do programador, permitindo que, sempre que seja necessário validar um atributo ou conjunto de atributos, seja possível apenas chamar estas subclasses e o respetivo método de validação.

As subclasses de `Validações` são definidas como `static`, uma vez que não há necessidade de criar instâncias da mesma. O principal propósito é disponibilizar métodos utilitários para a validação de dados, promovendo uma maior organização, reutilização e simplicidade no código.

Além disso, as validações são organizadas e associadas a códigos de erro específicos, que são posteriormente tratados nas `Exceptions`. Este mecanismo garante uma gestão clara e eficiente de erros, assegurando que qualquer problema encontrado seja rapidamente identificado e devidamente reportado.

Validação do Objeto Consulta O método `ValidarObjetoConsulta` é responsável por validar se um objeto da classe `Consulta` é válido, garantindo que os seus atributos respeitam os critérios estabelecidos. Este método desempenha um papel crucial na consistência de dados do sistema. O código do método é apresentado na Figura 4.

```
public static int ValidarObjetoConsulta(Consulta consulta)
{
    if (consulta == null) return -109;
    int res = ValidarCamposConsulta(consulta.DataInicio,
        consulta.DataFim,
        consulta.NIF,
        consulta.CRM,
        consulta.NumeroStaff,
        consulta.IdDiagnostico,
        consulta.TipoConsulta,
        consulta.Estado);
    if (res != 1) return res;

    return 1;
}
```

Figura 4. *Implementação do método `ValidarObjetoConsulta`.*

Explicação do Código:

- **Verificação de nulidade:** Inicialmente, o método verifica se o objeto `Consulta` é nulo. Caso seja, retorna o código de erro `-109`, sinalizando que o objeto não foi instanciado.
- **Validação dos campos:** Se o objeto não for nulo, o método recorre a `ValidarCamposConsulta` para validar os atributos específicos da consulta, como `DataInicio`, `DataFim`, `CRM`, `NIF`, entre outros.
- **Código de erro:** Caso algum atributo não passe no processo de validação, o código de erro correspondente é devolvido, permitindo uma gestão eficiente dos erros.

- **Validação bem-sucedida:** Se todas as verificações forem aprovadas, o método retorna `1`, indicando que o objeto `Consulta` é válido.

Verificação de Disponibilidade do Médico O método `EncontraDisponibilidadeMedico` foi desenvolvido para verificar a disponibilidade de um médico num determinado intervalo de tempo. Esta funcionalidade é essencial para evitar conflitos de agendamento no sistema. O código é apresentado na Figura 5.

```
public static int EncontraDisponbibilidadeMedico(MiniConsulta mini)
{
    int res;
    List<int> consultas = TodasConsultas.EncontraTodasConsultasMedico(mini.CRM);
    if (consultas.Count == 0)
        return 1; // tem disp
    foreach (int idConsulta in consultas)
    {
        //Dinicio
        res = ValidarListaConsulta.ComparaIntervalosData(mini.DataInicio, TodasConsultas.ObterConsulta(idConsulta));
        if (res == -113)
            return -115; //para informar que o problema está na DataIncio
        if (res != 1)
            return res;
        //Dfim
        res = ValidarListaConsulta.ComparaIntervalosData(mini.DataFim, TodasConsultas.ObterConsulta(idConsulta));
        if (res == -113)
            return -116; //para informar que o problema está na DataFim
        if (res != 1)
            return res;
    }
    return 1; //não existe sobreposições
}
```

Figura 5. *Implementação do método `EncontraDisponibilidadeMedico`.*

Explicação do Código:

- **Obtenção de consultas associadas:** O método começa por utilizar `EncontraTodasConsultas Medico` para recuperar todas as consultas associadas ao CRM do médico. Caso nenhuma consulta seja encontrada, considera-se que o médico está disponível, retornando `1`.
- **Validação dos intervalos de datas:** Para cada consulta encontrada:
  - A data de início (`DataInicio`) da nova consulta é comparada com os intervalos das consultas existentes através do método `ComparaIntervalosData`. Em caso de conflito, é retornado o código `-115`.
  - A mesma verificação é efetuada para a data de fim (`DataFim`). Caso exista conflito, é devolvido o código `-116`.
- **Resultado final:** Se nenhum conflito for identificado, o método retorna `1`, indicando que o médico está disponível no intervalo solicitado.

Importância das Validações Os métodos apresentados desempenham um papel essencial na integridade e robustez do sistema:

- **Consistência de Dados:** Garante que apenas objetos e atributos válidos são aceites e processados, reduzindo o risco de inconsistências.
- **Gestão Eficiente de Erros:** A utilização de códigos de erro facilita a identificação e o tratamento dos problemas, contribuindo para uma experiência de utilizador mais fluida.
- **Segurança:** Previne a manipulação de objetos inválidos ou incompletos, protegendo o sistema de falhas inesperadas.

- **Modularidade:** A centralização das validações em métodos específicos promove a reutilização de código e simplifica a manutenção do sistema.

Exceções Tal como referido anteriormente, as exceções foram criadas para quase todas as subclasses e classes. Na parte das consultas, foram desenvolvidas três exceções principais: `ConsultaException`, `ListaConsultaException` e `RegrasConsultaException`. Estas exceções estão organizadas por códigos de erro, apresentados nas tabelas seguintes.

ConsultaException: As exceções associadas à classe `Consulta` estão descritas na Tabela 4.

Código de Erro	Descrição
-100	Id da Consulta Inválido
-101	Data de Início Inválida
-102	Data de Fim Inválida
-103	NIF Inválido
-104	CRM Inválido
-105	Número do Staff Inválido
-106	ID de Departamento Inválido
-107	Tipo de Consulta Inválido
-108	Estado de Consulta Inválido
-109	Objeto Consulta nulo
-110	Objeto MiniConsulta nulo

Tabela 4. *Códigos de erro associados à classe `ConsultaException`.*

ListaConsultaException: As exceções associadas à classe `ListaConsultaException` estão descritas na Tabela 5.

Código de Erro	Descrição
-111	Lista nula
-112	Lista vazia
-113	A data está dentro do intervalo de outra consulta
-114	A consulta está duplicada
-115	A data do Inicio está dentro do intervalo de outra consulta
-116	A data do Fim está dentro do intervalo de outra consulta

Tabela 5. *Códigos de erro associados à classe `ListaConsultaException`.*

RegrasConsultaException: As exceções associadas à classe `RegrasConsultaException` estão descritas na Tabela 6.

Código de Erro	Descrição
-141	Sem Permissão

Tabela 6. *Códigos de erro associados à classe `RegrasConsultaException`.*

Lista de Consultas: A lista que armazena todas as consultas do projeto é manipulada através da subclasse *TodasConsultas*. Esta subclasse fornece vários métodos que permitem a manipulação, gestão e validação das consultas registadas. Estes métodos são descritos abaixo:

- **ExisteConsulta:** Este método verifica se uma consulta já existe na lista, retornando um valor booleano (`true` se existir, `false` caso contrário).
- **AdicionarConsulta:** Responsável por adicionar uma consulta à lista. Retorna um valor inteiro indicando o sucesso (`1`) ou um código de erro, caso não seja possível adicionar.
- **RemoverConsulta:** Permite remover uma consulta existente da lista. O método devolve um inteiro, sendo `1` o código de sucesso e outros valores correspondendo a erros.
- **FindConsulta:** Encontra e devolve um objeto do tipo `Consulta` com base no identificador fornecido. Este método é privado, garantindo que a busca de consultas seja encapsulada dentro da classe.

```
private static Consulta FindConsulta(int id)
{
    int res;
    try
    {
        res = ValidarConsulta.ValidarIdConsulta(id);
        if (res != 1)
            throw new ConsultaException(res);

        Consulta consulta = null;
        foreach (Consulta c in _consultas)
        {
            if (c.IdConsulta.Equals(id))
            {
                consulta = c;
                break;
            }
        }
        res = ValidarConsulta.ValidarObjetoConsulta(consulta);
        if (res == 1)
            return consulta;
        throw new ConsultaException(res);
    }
    catch (ListaConsultaException)
    {
        throw;
    }
}
```

Figura 6. *Método FindConsulta*

Explicação do Código:

- **Tipo de Método:** Este método é privado, isto significa que a só pode ser utilizado dentro daquela classe.
- **Validação do ID da Consulta:** O método começa por validar o `idConsulta` fornecido, utilizando o método `ValidarConsulta.ValidarIdConsulta`. Caso o ID não seja válido, é lançada uma `ConsultaException` com o respetivo código de erro.
- **Procura na Lista de Consultas:** O método percorre a lista de consultas (`_consultas`) utilizando um ciclo `foreach`. Caso o `idConsulta` corresponda ao ID de uma consulta, essa consulta é guardada na variável `consulta`.
- **Validação do Objeto Consulta:** Após encontrar a consulta, o método valida o objeto `Consulta` com `ValidarConsulta.ValidarObjetoConsulta`. Se o objeto for válido, a consulta é devolvida. Caso contrário, é lançada uma exceção com o respetivo código de erro.
- **Tratamento de Exceções:** Qualquer erro que ocorra durante o processo é capturado e lançado para ser tratado por métodos superiores.

- **ObterConsulta:** Este método devolve um objeto do tipo `MiniConsulta`, que contém apenas informações essenciais da consulta: `idConsulta`, `CRM`, `NIF`, `dataInicio` e `dataFim`. Este método é útil para reduzir a utilização de dados desnecessários ou confidenciais.

```
public static MiniConsulta ObterConsulta(int id)
{
    try
    {
        Consulta c = FindConsulta(id);
        MiniConsulta mini = new MiniConsulta(c.IdConsulta, c.CRM, c.NIF, c.DataInicio, c.DataFim);
        int res = ValidarConsulta.ValidarObjetoMiniConsulta(mini);
        if (res != 1)
            throw new ConsultaException(res);
        return mini;
    }
    catch (ConsultaException)
    {
        throw;
    }
    catch (ListaConsultaException)
    {
        throw;
    }
}
```

Figura 7. *Método ObterConsulta*

Explicação do Código:

- **Tipo de Método:** Este método já é publico, o que significa que todas as entidades do projeto podem acedê-lo, ele é responsável por auxiliar o método FindConsulta, na questão de privacidade dos dados.
- **Procura da Consulta:** O método começa por chamar o método `FindConsulta`, passando o `idConsulta` para encontrar a consulta na lista.
- **Criação da MiniConsulta:** Após encontrar a consulta, é criado um objeto `MiniConsulta`, que armazena apenas os dados relevantes: `idConsulta`, `CRM`, `NIF`, `dataInicio` e `dataFim`.
- **Validação da MiniConsulta:** O objeto `MiniConsulta` é validado utilizando `ValidarConsulta`. *ValidarObjetoMiniConsulta*. Se a validação falhar, é lançada uma `ConsultaException`.
- **Tratamento de Exceções:** Caso ocorram erros durante a execução, como consultas não encontradas ou objetos inválidos, as exceções são capturadas e lançadas para métodos superiores.
- **Retorno:** Caso todos os passos sejam concluídos com sucesso, o objeto `MiniConsulta` é retornado.

Benefícios:

- **Encapsulamento:** O método `FindConsulta` é privado, garantindo que a procura de consultas está encapsulada dentro da classe.
- **Redução de Dados Expostos:** Ao utilizar o método `ObterConsulta`, apenas os dados essenciais da consulta são devolvidos, promovendo uma maior segurança e eficiência.
- **Reutilização de Código:** Ambos os métodos utilizam validações centralizadas, o que facilita a manutenção e melhora a consistência do sistema.

- **EncontraTodasConsultasPaciente:** Devolve uma lista com os identificadores (`idConsulta`) de todas as consultas associadas a um paciente, identificado pelo `NIF`.
- **EncontraTodasConsultasMedico:** Devolve uma lista com os identificadores (`idConsulta`) de todas as consultas associadas a um médico, identificado pelo `CRM`.
- **FindConsultaParam:** Procura uma consulta com base em parâmetros específicos e devolve um código inteiro que representa o estado ou o sucesso da operação.

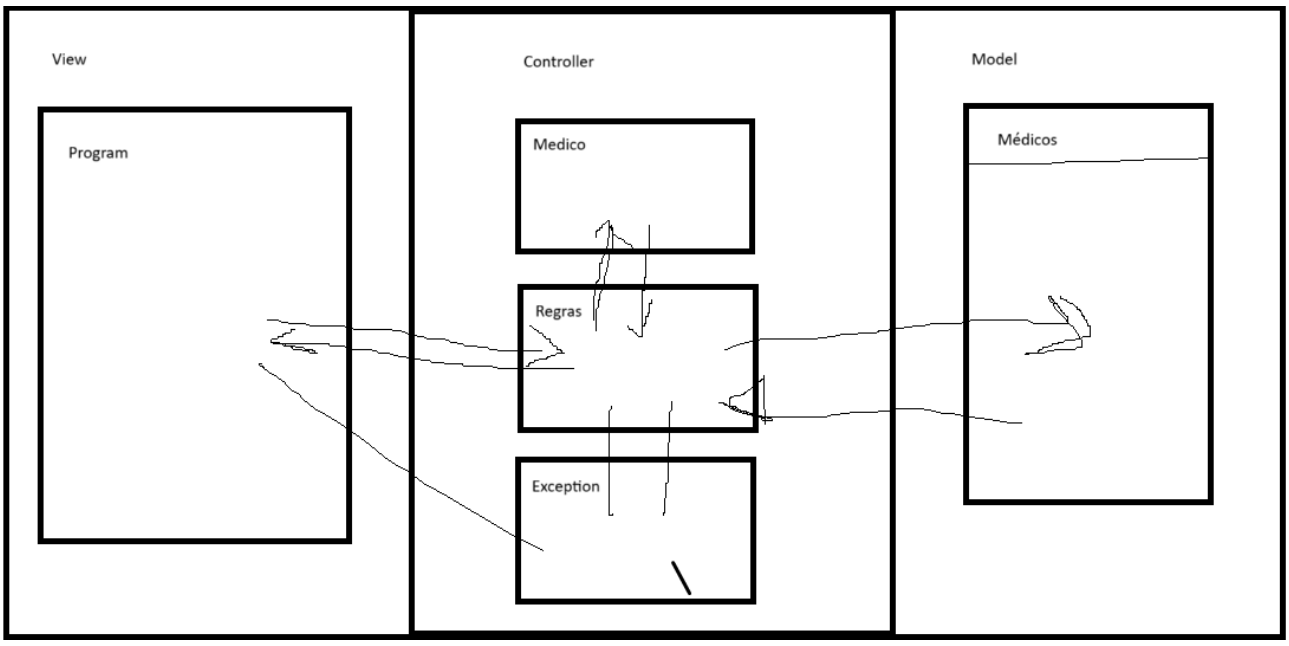

Figura 8. *Camadas do projeto*

- **GuardarConsultasFicheiro:** Este método guarda todas as consultas da lista num ficheiro externo, garantindo a persistência dos dados. Retorna `1` para sucesso ou um código de erro em caso de falha.
- **LerConsultasFicheiro:** Lê todas as consultas a partir de um ficheiro externo e carrega-las na lista em memória. Retorna `1` para sucesso ou um código de erro em caso de falha.

Regras para Consulta: Assim como visto anteriormente para a class *Médico*, a subclasse *RegrasConsulta* é responsável por fazer a ligação entre a 1ª camada e a 3ª camada, o cliente e os dados. Nesta subclasse estão todos os métodos relacionados à gestão da lista e da classe *Consulta*, aqui estão alguns exemplos:

- **ObterConsultasMedico:** Permite obter todas as consultas associadas a um médico específico com base no CRM.
- **ObterConsultasPaciente:** Permite obter todas as consultas associadas a um paciente com base no NIF.
- **TentaCriarConsulta:** Realiza todas as validações e tenta criar uma nova consulta, verificando disponibilidade e prevenindo conflitos de agendamento.
- **TentarAdicionarConsulta:** Adiciona uma consulta existente à lista de consultas, verificando permissões e integridade do objeto.
- **TentaRemoverConsulta:** Remove uma consulta identificada pelo `idConsulta`, caso o utilizador tenha as permissões necessárias.

7.3 Camadas do Projeto

Durante as aulas foram exploradas diversas metodologias que poderiam ser utilizadas para estruturar e organizar o projeto. Entre as opções estudadas, destacaram-se o modelo N-Tier e o padrão MVC. Para compreender melhor a implementação destas camadas em C#, foi elaborado, com o auxílio do professor, o seguinte esquema ilustrativo:

O esquema apresentado ilustra a comunicação entre as diferentes camadas do projeto, evidenciando o fluxo autorizado de informações. As setas no diagrama representam os caminhos de interação permitidos, garantindo o cumprimento das regras de acesso e validações.

Exemplo 1 - Criação de um objeto do tipo Médico: Um utilizador com permissões adequadas solicita a criação de um objeto `Medico` à camada de **Regras**. Contudo, antes de proceder à criação, as **Regras** enviam os dados à subcamada de **Validações** (este bloco foi adicionado posteriormente e não está refletido no esquema inicial). As **Validações** analisam os dados fornecidos e, caso estejam corretos, as **Regras** pedem à classe **Medico** para criar o objeto. Se o objeto for criado com sucesso, ele é devolvido à camada de **Regras**, que por sua vez o envia ao utilizador. Caso ocorra algum erro durante o processo, um código de erro é retornado, indicando o problema.

Exemplo 2 - Adição de um Médico à lista: Após obter um objeto `Medico`, o utilizador pode desejar adicioná-lo à lista de médicos. Neste caso, ele solicita à camada de **Regras** para realizar esta operação. As **Regras** enviam o pedido à subcamada de **Validações**, onde os dados do objeto são validados. Se os dados forem válidos, a camada de **Regras** solicita à classe **Medicos** que guarde o objeto na lista. Se a operação for bem-sucedida, a classe **Medicos** retorna um inteiro indicando sucesso. Caso contrário, é devolvido um código de erro que, tal como no exemplo anterior, percorre a camada de **Regras** antes de chegar ao utilizador.

Importância do Esquema: Este esquema exemplifica a organização lógica do projeto e a separação de responsabilidades entre as camadas. Cada camada tem um papel bem definido, o que facilita a manutenção, a escalabilidade e o cumprimento de boas práticas de desenvolvimento. Além disso, o fluxo de comunicação controlado reduz a ocorrência de erros e melhora a segurança e a integridade dos dados ao longo do sistema.

Em baixo será apresentado o trabalho realizado em cada uma das camadas.

7.3.1 N-Tier

O modelo N-Tier, também conhecido como arquitetura em camadas, foi escolhido para organizar o projeto de forma modular e escalável. Este modelo divide a aplicação em várias camadas lógicas, onde cada uma tem responsabilidades bem definidas, promovendo a separação de preocupações e facilitando a manutenção e evolução do sistema.

No contexto deste projeto, as camadas implementadas foram:

- **Camada de Lógica do Negócio (BL):** Nesta camada foram implementadas todas as regras de negócio do projeto, como validações, permissões e a lógica subjacente às operações realizadas. É responsável por assegurar a integridade das operações realizadas e o cumprimento das regras previamente definidas, como verificar a disponibilidade de médicos para consultas e garantir que apenas utilizadores autorizados podem aceder a determinadas informações.

- **Camada de Acesso a Dados (DAL):** Esta camada lida diretamente com o armazenamento e recuperação de dados, abstraindo a interação com os ficheiros ou uma base de dados. No projeto, a camada de acesso a dados permite guardar e ler informações como médicos, pacientes e consultas de ficheiros, garantindo a persistência das informações.

- **Camada Transversal (CT):** É composta por um conjunto de operações, ferramentas e funcionalidades que são utilizadas de forma comum por todas as camadas do projeto. Esta camada desempenha um papel fundamental ao fornecer serviços e utilitários que promovem a consistência, reutilização de código e a manutenção do sistema.

  Exemplo de Componentes da Camada Transversal:

Validações: Métodos que validam a integridade e a coerência dos dados antes de serem processados por outras camadas. Estes métodos incluem validações de valores, tamanhos, formatos e integridade de objetos. Exemplo: A validação de um NIF ou CRM antes de criar ou modificar um objeto.

Gestão de Exceções: Classes responsáveis por centralizar a criação e a gestão de exceções específicas para o projeto, garantindo que os erros sejam tratados de forma consistente em todas as camadas.

Vantagens do Modelo N-Tier:

- **Manutenção Facilitada:** A separação de camadas facilita a manutenção, permitindo modificar ou melhorar uma camada sem impactar as outras.
- **Reutilização de Código:** A lógica de negócio e as operações de acesso a dados podem ser reutilizadas por diferentes interfaces de utilizador.
- **Escalabilidade:** O sistema pode ser expandido adicionando novas funcionalidades em camadas específicas sem a necessidade de reestruturar toda a aplicação.

Desafios do Modelo N-Tier: Apesar das suas vantagens, o modelo N-Tier também apresenta alguns desafios, como o aumento da complexidade inicial do desenvolvimento e a necessidade de comunicação eficiente entre as camadas. Contudo, ao longo do projeto, o modelo demonstrou ser uma escolha robusta e eficaz para garantir a organização e a modularidade do sistema.

Esquema do N-Tier: O esquema apresentado na Figura 9 ilustra a estrutura do modelo N-Tier implementado no projeto. Este modelo é composto por camadas distintas que organizam e separam as responsabilidades, promovendo modularidade, reutilização de código e maior clareza na arquitetura.

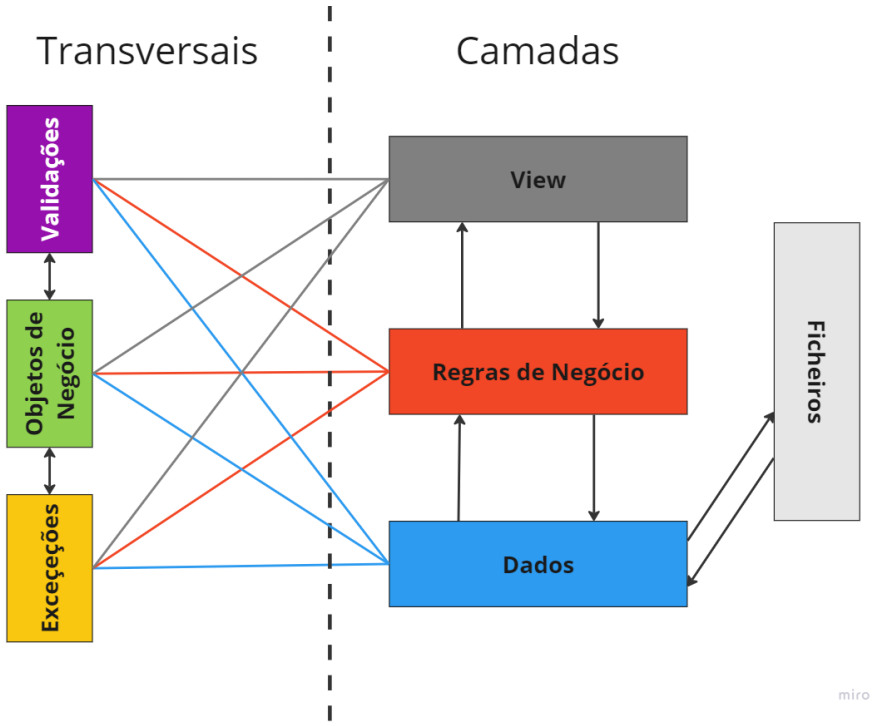

Figura 9. *Esquema N-Tier*

- Camada de View: Responsável pela interação com o utilizador, exibindo os dados e capturando entradas para serem processadas pelas camadas inferiores. Embora não tenha sido desenvolvida, está documentada. Apenas existe o program.cs aonde são feitos os testes dos métodos.
- Camada de Regras de Negócio: Centraliza a lógica de negócio do sistema. Esta camada interpreta as solicitações recebidas da camada de apresentação (View) e interage com a camada de Dados para aceder ou manipular a informação.
- Camada de Dados: Contém os métodos e estruturas necessárias para armazenar e gerir os dados, seja em memória, ficheiros ou outros meios de armazenamento.
- Camada Transversal: Inclui elementos como validações, objetos de negócio e exceções, que são utilizados de forma partilhada por todas as outras camadas. Esta camada promove consistência e organização.

O fluxo de comunicação entre as camadas é representado pelas setas no diagrama, destacando os caminhos permitidos para a troca de informações. Este modelo garante que cada camada se comunique apenas com as camadas adjacentes, mantendo a integridade e a separação de responsabilidades no sistema.

7.3.2 MVC

O modelo MVC (Model-View-Controller) é uma metodologia amplamente utilizada para organizar e estruturar aplicações, promovendo a separação das responsabilidades em três componentes principais:

- Model: Responsável pela gestão e manipulação dos dados do sistema.

- View: Trata da apresentação dos dados ao utilizador, definindo a interface visual da aplicação e interagindo diretamente com o utilizador.
- Controller: Atua como intermediário entre o Model e a View, processando as ações do utilizador, aplicando a lógica de negócio necessária e atualizando a View com os dados relevantes. Inclui as regras de Negócio.

A Tentativa de Implementação do MVC Durante o desenvolvimento do projeto, explorei a possibilidade de implementar o modelo MVC como alternativa ao N-Tier. Apesar do esforço e do interesse em compreender este modelo, a implementação revelou-se mais desafiadora do que o esperado. Mesmo após estudar os ficheiros disponibilizados pelo professor no Moodle e no GitHub, não consegui entender completamente como integrar as componentes do MVC no projeto de forma eficaz.

Embora tenha tentado avançar com a implementação, a falta de clareza na abordagem e a complexidade percebida levaram-me a optar pelo modelo N-Tier, que me parecia mais adequado e acessível no momento. Reconheço, no entanto, o potencial do MVC e a sua aplicabilidade em projetos de maior dimensão e complexidade.

Abordagem Futura Este ponto será abordado de forma mais aprofundada noutra secção do relatório, onde discutirei as dificuldades encontradas.

7.4 Bibliotecas

Durante o desenvolvimento do projeto, foi criado um subprojeto exclusivo para a implementação de bibliotecas, com o objetivo de explorar este conceito fundamental do C#. Para esta abordagem, foi utilizada especificamente a classe `Medico`, permitindo um estudo aprofundado da criação e utilização de bibliotecas dinâmicas (*Dynamic Link Libraries* ou *DLLs*) dentro do contexto do projeto.

Importância das Bibliotecas no Projeto A criação de bibliotecas oferece diversos benefícios, como:

- **Reutilização de Código:** Permite que a lógica e os métodos associados a uma classe sejam utilizados em diferentes partes do sistema sem duplicação de código.
- **Modularidade:** Facilita a organização e separação de responsabilidades no projeto, contribuindo para um código mais limpo e estruturado.
- **Portabilidade:** As bibliotecas podem ser partilhadas e reutilizadas em diferentes projetos, promovendo economia de tempo e esforço no desenvolvimento.
- **Encapsulamento:** Protege o funcionamento interno das classes, expondo apenas os métodos e atributos necessários para a interação com outras partes do sistema.

Partilha de Conhecimento Com o intuito de ajudar outros colegas no entendimento e na criação de *DLLs*, produzi um vídeo onde demonstro o processo de desenvolvimento das minhas bibliotecas para o projeto. Este vídeo está disponível no seguinte link:

<https://alunosipca-my.sharepoint.com/:v:/g/personal/a27985_alunos_ipca_pt/ETk8EaDAcj9FkmMEm_fbVxQB1OLcVyyQaj11SrWB1aElaw?e=kcclF6>

7.5 Logs

O sistema inclui um módulo de registo de *logs* que captura e regista todos os erros gerados durante a execução do programa. Estes erros são registados em ficheiros de texto com o formato estruturado apresentado abaixo:

```
[19/12/2024 16:15:54] [Tipo: ListaConsultaException] [Código: -115]
[Mensagem: A data do Início está dentro do intervalo de outra consulta]

[19/12/2024 16:16:27] [Tipo: ListaConsultaException] [Código: -115]
[Mensagem: A data do Início está dentro do intervalo de outra consulta]

[19/12/2024 16:18:45] [Tipo: ListaConsultaException] [Código: -115]
[Mensagem: A data do Início está dentro do intervalo de outra consulta]

[21/12/2024 17:48:36] [Tipo: ConsultaException] [Código: -101]
[Mensagem: Data de Início Inválida]
```

Este sistema é extremamente útil para o programador, pois permite a monitorização dos erros ocorridos e a respetiva análise para correção. A seguir, apresenta-se o código que implementa esta funcionalidade.

7.5.1 Método `Log`

O método `Log` é responsável por registar os detalhes das exceções no ficheiro de log. A sua implementação está apresentada na Figura 10.

```
public static int Log(Exception ex)
{
    try
    {
        // Verifica se a exceção contém um código de erro
        int errorCode = 0;
        if (ex is ConsultaException consultaEx)
            errorCode = consultaEx.ErrorCode;
        else if (ex is ListaConsultaException listaCEx)
            errorCode = listaCEx.ErrorCode;
        else if (ex is MedicoException medicoEx)
            errorCode = medicoEx.ErrorCode;
        else if (ex is ListaMedicosException listaMedicosEx)
            errorCode = listaMedicosEx.ErrorCode;
        else if (ex is RegrasMedicosException RmedicoEx)
            errorCode = RmedicoEx.ErrorCode;
        else if (ex is RegrasConsultaException RconsultaEx)
            errorCode = RconsultaEx.ErrorCode;

        using (StreamWriter writer = new StreamWriter(_LogFilePath, true)) //vai adicionar ao texto ja existente
        {
            writer.WriteLine($"[{DateTime.Now}] [Tipo: {ex.GetType().Name}] [Código: {errorCode}] [Mensagem: {ex.Message}]");
        }
        return 1;
    }
    catch
    {
        return -1;
    }
}
```

Figura 10. *Método `Log`: Registo de exceções no ficheiro de log.*

Descrição do Código:

- **Identificação do Tipo de Exceção:** O método identifica a classe da exceção (`ConsultaException`, `ListaConsultaException`, etc.) e obtém o respetivo código de erro, caso exista.

- **Registo no Ficheiro:** Utilizando a classe `StreamWriter`, o método escreve no ficheiro os detalhes da exceção, incluindo a data, o tipo, o código de erro e a mensagem associada.
- **Gestão de Erros:** Em caso de falha no processo de registo, o método retorna o código de erro `-1`.

7.5.2 Método `CriarLog`

O método `CriarLog` é responsável por encapsular a funcionalidade do `Log`, assegurando que o utilizador seja notificado sobre o sucesso ou falha do registo da exceção. A implementação deste método está ilustrada na Figura 11.

```
public static int CriarLog(Exception ex)
{
    int res123 = Log(ex);
    if (res123 == 1)
    {
        return res123;
    }
    else
    {
        return res123;
    }
    throw ex;
}
```

Figura 11. *Método `CriarLog`: Notificação e encapsulamento do registo de exceções.*

Descrição do Código:

- **Chamadas ao Método `Log`:** Invoca o método `Log` para registar a exceção no ficheiro.
- **Notificação ao Utilizador:** Após a execução, informa o utilizador sobre o estado do registo:
  - **Sucesso:** devolve 1 `"Erro registado no ficheiro de log."`.
  - **Falha:** devolve o erro respetivo `"Falha ao registar o erro no ficheiro de log."`.
- **Encapsulamento da Exceção:** A exceção original é novamente lançada para não interromper o fluxo do programa.

7.6 Testes

Na parte dos testes, não tive muito tempo para os explorar, cheguei por fazer algo muito rápido, mas sem tempo para resolver bugs, acabei por não conseguir rodar nenhum teste, mas durante o periodo até apresentar vou melhorar esta parte.

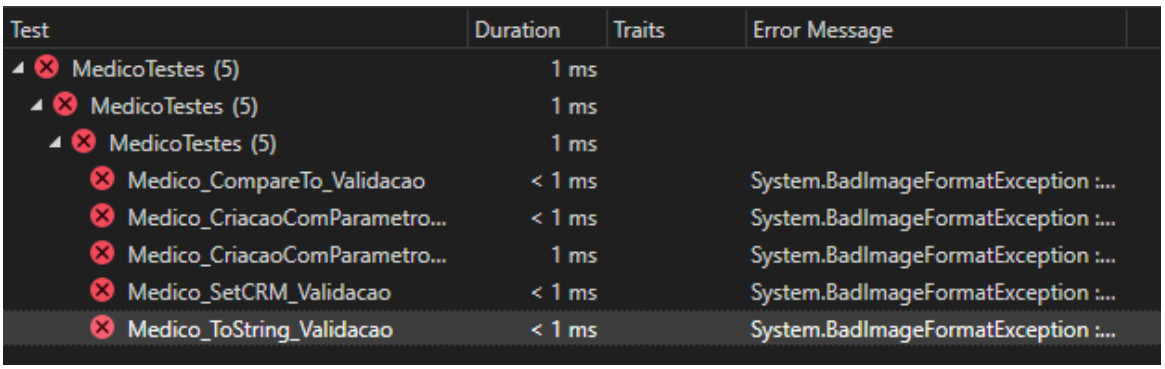

Figura 12. *Testes Unitários*

8 Problemas do Projeto

8.1 MVC

Durante o desenvolvimento do projeto, decidi tentar implementar a metodologia `MVC` (Model-View-Controller). Apesar dos esforços e de compreender o conceito teórico do `MVC`, não consegui aplicá-lo de forma funcional. Mesmo após estudar os materiais disponibilizados pelo professor no *Moodle* e no *GitHub*, enfrentei dificuldades significativas na transição do conceito teórico para a prática.

Consegui desenvolver a camada `View`, mas no que diz respeito ao `Model`, surgiram várias dúvidas sobre como organizar e guardar os dados. Questões como: "Onde devo guardar as listas?", "Como aceder a essas listas?", "Onde escrevo o código para manipulá-las?" deixaram-me confuso e atrasaram o progresso. A camada `Controller` também apresentou desafios semelhantes, uma vez que não conseguia estruturar claramente a lógica que faria a ligação entre o `Model` e a `View`.

Além disso, como este conteúdo foi introduzido já numa fase mais avançada do semestre, a cerca de duas semanas da entrega final, não tive oportunidade de visitar o gabinete do professor para esclarecer estas dúvidas. Como resultado, optei por abandonar a tentativa de implementar `MVC` e concentrar-me na aplicação da metodologia `N-Tier`, onde utilizei boas práticas para estruturar o projeto de forma organizada e funcional. Apesar do insucesso com o `MVC`, esta experiência permitiu-me ganhar uma visão mais ampla sobre diferentes metodologias de desenvolvimento.

8.2 Ficheiro Geral

Outro problema encontrado foi a implementação de um ficheiro geral para guardar e ler dados de várias classes. Desenvolvi a classe `FileManagement`, que inclui instâncias das classes `Medico` e `Consulta`. Foram criados métodos para guardar e ler dados de um ficheiro, mas mesmo assim não obtive sucesso. Ao testar com `debug`, percebi que as classes ficavam sem dados ao serem lidas e que os ficheiros criados não armazenavam a informação pretendida. Este problema revelou-se particularmente frustrante, já que a lógica parecia correta, mas a execução não produzia os resultados esperados.

8.3 GitHub

Inicialmente, utilizei o *GitHub* como ferramenta para organização do projeto. Comecei por desenvolver a classe `Medico` num repositório. No entanto, ao testar as bibliotecas e tentar integrar novas versões, surgiram problemas de compatibilidade entre *branches*, o que dificultou o progresso. Para resolver este problema, decidi criar um novo repositório e reorganizar o projeto, adotando uma estrutura semelhante à utilizada pelo professor,

```
public static int GuardarTudoFicheiro(string fileName)
{
        try
        {
            Stream stream = File.Open(fileName, FileMode.OpenOrCreate, FileAccess.ReadWrite);
            BinaryFormatter bin = new BinaryFormatter();
            FileManagement dados = new FileManagement();
            bin.Serialize(stream, dados);
            return 1;
        }
        catch (IOException e)
        {
            throw e;
        }
}
```

Figura 13. *Guardar todas as classes num ficheiro*

com projetos separados por pastas. Esta reorganização permitiu uma melhor organização do código, além de possibilitar a inclusão de recursos adicionais, como vídeos explicativos e o esquema de camadas do projeto.

8.4 GitHub

Desde o início do projeto, tomei a decisão de utilizar o *GitHub* como ferramenta de apoio. Como era o aluno com mais progressos no desenvolvimento e com maior domínio sobre `C#` no meu grupo de amigos, decidi criar um repositório público. Um aluno do terceiro ano sugeriu esta abordagem, destacando que, além de facilitar o versionamento, ajudaria a dar mais visibilidade ao meu perfil no *GitHub*.

O repositório foi estruturado de forma organizada, seguindo uma metodologia clara. A minha organização inspirou outros colegas, que utilizaram o repositório para tirar ideias sobre como implementar determinadas funcionalidades. O repositório conta ainda com recursos adicionais, como um vídeo explicativo da criação das bibliotecas e o esquema das camadas. Considero que esta decisão foi acertada, não só porque me ajudou a manter o controlo do projeto, mas também porque permitiu que outros alunos beneficiassem do trabalho desenvolvido.

9 GitHub

Desde o início do projeto, decidi publicá-lo no GitHub, dado que no meu grupo de colegas era quem estava mais avançado no desenvolvimento e tinha mais bases de programação em C#. Além disso, recebi a recomendação de um aluno do 3.º ano para disponibilizar o projeto publicamente, com o objetivo de dar mais visibilidade ao meu perfil no GitHub e contribuir para a comunidade. A decisão revelou-se bastante útil, tanto para mim quanto para os outros alunos, pois o repositório serviu como referência para ideias de implementação e organização do código.

O meu repositório foi estruturado para refletir boas práticas, incluindo uma separação clara entre projetos e recursos adicionais, como vídeos explicativos e esquemas. Abaixo, apresento uma análise detalhada das métricas do GitHub associadas ao meu projeto.

9.1 Análise de Desempenho do Repositório

Na Figura 14, podemos observar duas métricas principais:

- **Clones:** Refere-se ao número total de vezes que o repositório foi clonado (22 clones), e o número de utilizadores únicos que realizaram essas operações foi de 18. O pico de clones ocorreu a 12 de dezembro, com mais de 10 clones, indicando um interesse significativo pelo repositório nesse dia.

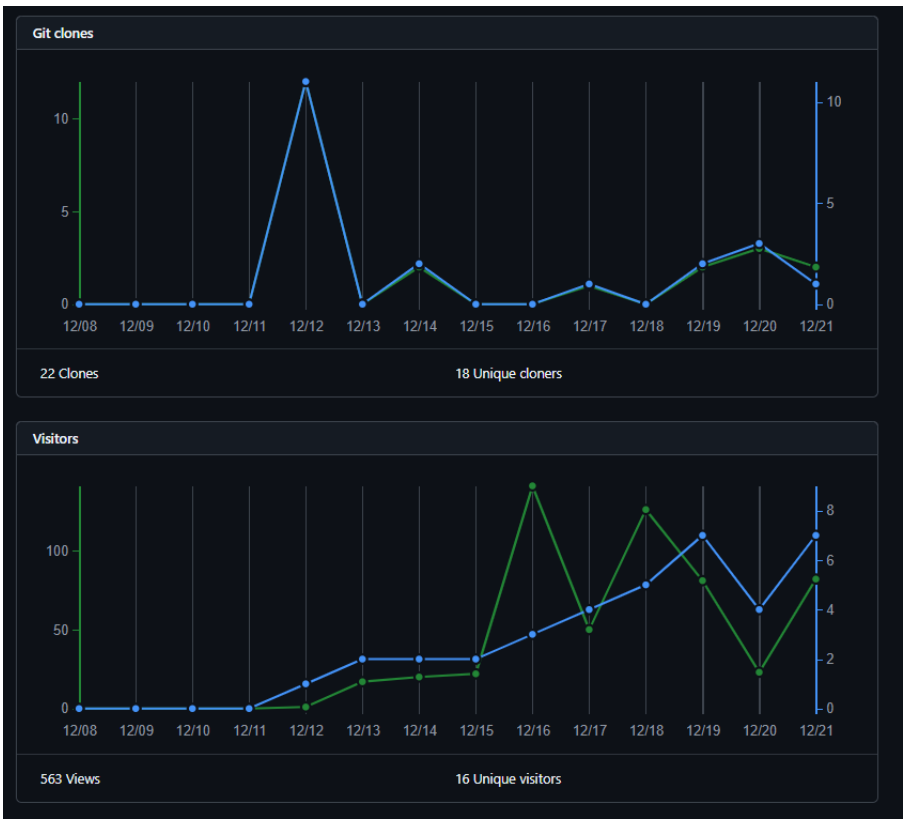

Figura 14. *Gráficos de Clones e Visitantes do Repositório*

- **Visitantes:** O número de visitantes reflete o número total de visualizações do repositório (563 visualizações), com 16 visitantes únicos. Este valor destaca que, além de clonar, muitos utilizadores exploraram o repositório online para obter informações.

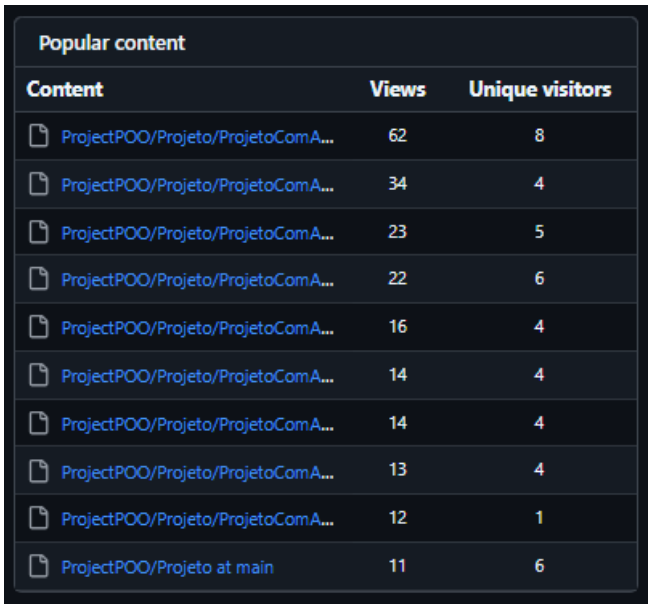

Figura 15. *Conteúdo Popular do Repositório*

Na Figura 15, é apresentado o conteúdo mais visualizado do repositório:

- O ficheiro mais visualizado foi o `ProjectPOO/Projeto/ProjetoComAsMinhasCamadas`

`/MinhasCamadas/Dados/Medicos.cs`, com 62 visualizações e 8 visitantes únicos. Este ficheiro foi relevante, pois continha os métodos implementados e demonstrava a lógica aplicada nas Listas.

- Outros ficheiros também tiveram um número significativo de visualizações, destacando o interesse em diferentes partes do projeto, como estruturas auxiliares e exemplos de funcionalidades.

9.2 Impacto do Repositório

O meu repositório ajudou vários colegas, que puderam usar as ideias partilhadas para melhorar os seus próprios projetos. A organização clara dos ficheiros e a inclusão de recursos adicionais (como vídeo e esquemas) contribuíram para que outros entendessem a abordagem adotada.

10 Referências

As referências utilizadas neste relatório estão listadas abaixo, organizadas por tipo de fonte:

Artigos Online

- Microsoft Learn (2023). *Guia de Estilos de Arquitetura: N-Tier*. Última atualização: 8 de abril de 2023. URL: <https://learn.microsoft.com/pt-pt/azure/architecture/guide/architecture-styles/n-tier>
- B. W. (2023). *Saiba como gerir coleções de dados com a Lista<T> em C#*. Última atualização: 8 de abril de 2023. URL: <https://learn.microsoft.com/pt-pt/dotnet/csharp/tour-of-csharp/tutorials/arrays-and-collections>
- Universidade Veiga de Almeida (UVA) (2024). *O que é CRM Médico e para que serve*. Última atualização: 13 de novembro de 2024. URL: <https://www.uva.br/postagens/o-que-e-crm-medico-e-para-que-serve>
- Luís Ferreira (2024). *LESI-POO-2024-2025*. Última atualização: 2024. URL: <https://github.com/luferIPCA/LESI-POO-2024-2025>

Relatórios e Livros

- Universidade do Porto (2024). *Relatório de Projeto de Software*. Porto: FEUP