

Licenciatura em Engenharia de Sistemas Informáticos

Programação Orientada ao Objeto

Ano letivo 2024/2025

Trabalho Prático Final

Pedro Miguel Cardoso Ribeiro

Nº 27960 – Regime Diurno

Docente:

Luís Gonzaga Martins Ferreira

Dezembro de 2024

Identificação do Aluno

Pedro Miguel Cardoso Ribeiro

Aluno número 27960, regime diurno

Licenciatura em Engenharia de Sistemas Informáticos

Docente:

Luís Gonzaga Martins Ferreira

RESUMO

Este projeto, desenvolvido em C#, visa criar um sistema de gestão de dados eficiente, explorando a boa implementação das classes e a otimização do código. São explorados, entre outros, os quatro pilares da Programação Orientada a Objetos: encapsulamento, herança, polimorfismo e abstração. É abordado como garantir um sistema com uma estrutura facilitadora à manutenção e a escalabilidade. O projeto a que se dá o presente relatório é composto por vários ficheiros cada um com a sua função, o que será referido e justificado ao longo do relatório.

ÍNDICE

1. Introdução.....	IX
1.1. Enquadramento.....	IX
1.2. Problema a resolver	IX
1.3. Objetivos	X
2. Revisão de Literatura	1
2.1. Linguagem C#.....	1
2.2. Programação Orientada a Objetos	1
2.3. UML e a Programação Orientada a Objetos.....	3
2.4. Design Patterns.....	4
2.4.1. MVC (Model-View-Controller).....	5
2.4.2. N-Tier	5
2.4.2.1. Vantagens do uso da N-Tier	7
2.5. Princípios SOLID.....	7
2.6. Testar Código	8
3. Trabalho Desenvolvido na 1ª Fase.....	9
3.1. Análise e Modelação das Classes	9
3.1.1. Utilização de UML.....	9
3.1.1.1. 1ª Versão: Primeira Visão do Projeto	10
3.1.1.2. 2ª Versão: Revisão das classes e aprimoramento da abstração e polimorfismo do Código	12
3.2. Estrutura das Classes e Lógica de Implementação.....	15
4. Análise de resultados/Reflexão (1ª Fase).....	19
4.1. Abordagens Exploradas, mas deixadas de lado:.....	19
4.2. Considerações sobre Decisões Tomadas:	20
5. Conclusão – 1ª Fase	21
6. Trabalho Realizado na 2ª Fase	23
6.1. Análise da 1ª Fase de Trabalho	23
6.2. Estruturação e Divisão em Camadas do Projeto	23
6.2.1. Estrutura Geral da Solução	24
6.3. Mudança de estrutura de dados (Array para Lists).....	25

6.4.	O uso dos identificadores	26
6.5.	Funcionamento das Classe de Gestão (Camada Regras de Negócio)	28
6.5.1.	Aplicação de Regras de Negócio	29
6.6.	As entidades presentes na camada de dados.....	31
6.6.1.	Propriedades sem set	31
6.6.1.1.	Alteração de atributos	32
6.7.	Exportação de Ficheiros	33
6.8.	Uso de Interfaces.....	34
6.9.	Passagem das Classes Light para Classes Normais	35
6.10.	Importância do Polimorfismo	35
6.11.	Exceções	36
6.11.1.	Exceções personalizadas	36
6.12.	Implementação da Interface IComparable	37
6.13.	Testes Unitários.....	38
6.13.1.	A minha Implementação dos Testes Unitários	38
6.13.2.	Como Funcionam os métodos nos Testes Unitários	38
6.13.3.	Correção no código devido aos testes unitários	40
7.	Análise de resultados/Reflexão (1ª Fase)	41
7.1.	Abordagens Exploradas, mas Deixadas de Lado.....	41
7.2.	Considerações sobre Decisões Tomadas	43
8.	Conclusão – 2ª Fase.....	45
9.	Bibliografia	47
9.1.	Bibliografia de Citações	47
9.2.	Bibliografia de Imagens	47

Lista de Figuras

Imagem 1 - As 10 linguagens de programação mais utilizadas em 2022	1
imagem 2 - Erich gamma	4
imagem 3 - Primeiro diagrama de classes, em uml	10
imagem 4 - Segundo diagrama de classes, em uml	12
imagem 5 - Demonstração de como aparece a execução dos testes unitários	39
imagem 6 – Resultado ao executar o teste unitário de adicionarconsultasuccesso	40

Lista de Excertos de código

L. Cod. 1 - exemplo de implementação dos métodos "adicionar", "remover" e "existe"	14
I. Cod. 2 – assinatura da classe medico	16
I. Cod. 3 – assinatura da classe paciente	16
I. Cod. 4 – exemplo de método que chama um método presente em outra classe	16
I. Cod. 5 – método que calcula o custo total presente na classe consulta	16
I. Cod. 6 – atributos de medico	27
I. Cod. 7 – atributos de consultasrepositorio	28
I. Cod. 8 – atributos e construtores de gestaomedicos	28
I. Cod. 9 – funcao de atualizarcustodoexame presente na gestaoexames	30
I. Cod. 10 – funcao eliminarmedico presente em gestaomedicos	31
I. Cod.11 – propriedade do identificador de consulta	31
I. Cod. 12 – funcao atualizarcustodoexame presente em gestaoexames	32
I. Cod. 13 – funcao de atualizar custo em exame	32
I. Cod. 14 – funcao de exportação presente em consultasrepositorio	33
I. Cod. 15 – funcao de exportação presente em gestaoconsultas	34
I. Cod. 16 – método compareto na classe exame	37
I. Cod. 17 – exemplo de teste unitário	39
I. Cod. 18 – antiga implementação de gestaoconsultas	42
I. Cod. 19 – função associarconsultasrepositorio presente em pessoa	43

1. Introdução

1.1. Enquadramento

No segundo ano da Licenciatura de Engenharia de Sistemas Informáticos, no Instituto Politécnico do Cávado e do Ave, no âmbito da unidade curricular de Programação Orientada ao Objeto, foi realizado como trabalho prático de semestre um projeto desenvolvido individualmente em linguagem C#, dividido em duas fases propostas em momentos distintos do período letivo do primeiro semestre. As duas fases foram realizadas utilizando documentação e elaboração de um relatório técnico desenvolvido em *DoxyGen*.

1.2. Problema a resolver

Foi-nos proposta a realização de uma solução capaz de criar e manipular dados, usando conceitos lecionados em contexto de aula e explorando mesmo novos conteúdos de forma autónoma. Ao longo do trabalho são usados e explicitados os 4 pilares principais da programação orientada ao objeto - encapsulamento, herança, polimorfismo e abstração. Inicialmente, será necessário para o projeto, formular e definir as estruturas de dados adequadas para a gestão de informações, como pacientes, médicos, consultas e diagnósticos. Na definição dessas classes, será necessária a exploração da relação entre as mesmas percebendo a aplicação na prática dos pilares da programação orientada ao objeto, verificando o respeito pelos mesmos. Após esta fase de reconhecimento da estruturação e conexão das devidas classes e do preenchimento dos devidos atributos individuais, serão implementadas as primeiras funções básicas que permitem criar, modificar e visualizar os dados associados a cada uma delas. Entre as funcionalidades a serem desenvolvidas estão, por exemplo, a inserção e remoção de pacientes e consultas. Posteriormente, a evolução do sistema irá levar à necessidade de reestruturar a forma como os dados eram geridos, passando de arrays para listas. A utilização de identificadores será também essencial para garantir a integridade e a correta manipulação dos dados. Vai ser também explorada a estruturação de um projeto em camadas, utilizando o N-Tier, serão abordados também conceitos como Testagem Unitária, exceções e entre outros.

1.3. Objetivos

O objetivo principal do presente trabalho é o aprimoramento da compreensão de estruturas de classes e dados, em linguagem C#, por meio da prática, com foco no desenvolvimento e na gestão de dados. Para alcançar esse objetivo, foram definidos os seguintes objetivos concretos:

1. **Definição e Implementação de Estruturas de Dados:** Criar classes que representem as entidades principais do sistema, assegurando que os dados sejam organizados de forma eficiente e acessível.
2. **Respeito e Exploração dos Pilares da Programação Orientada a Objetos:** Produção de um projeto que valorize a herança, o polimorfismo, a abstração e o Encapsulamento
3. **Desenvolvimento de Funções:** Implementar funções para realizar operações de inserção, remoção e atualização das informações associadas a cada classe, permitindo a manipulação dinâmica dos dados no sistema e a revisão dos conceitos associados a linguagem de programação C.
4. **Respeito aos parâmetros SOLID** e perceber o seu papel essencial na construção de uma solução eficiente, modular e de fácil manutenção.

2. Revisão de Literatura

2.1. Linguagem C#

A linguagem de programação C# é uma linguagem de alto nível desenvolvida pela Microsoft, que se destaca pela sua versatilidade e robustez. "C# is a modern, object-oriented programming language that is designed to be simple, powerful, and flexible." [1]. O C# tem uma sintaxe considerada clara, facilitadora da aprendizagem e da implementação de conceitos de programação orientada a objetos, sendo por este motivo, uma escolha popular entre desenvolvedores. Além disso, o suporte a bibliotecas e frameworks como o .NET Core e o ASP.NET amplia as suas capacidades, permitindo a criação de aplicações escaláveis e eficientes, o que leva também à popularidade associada à linguagem.

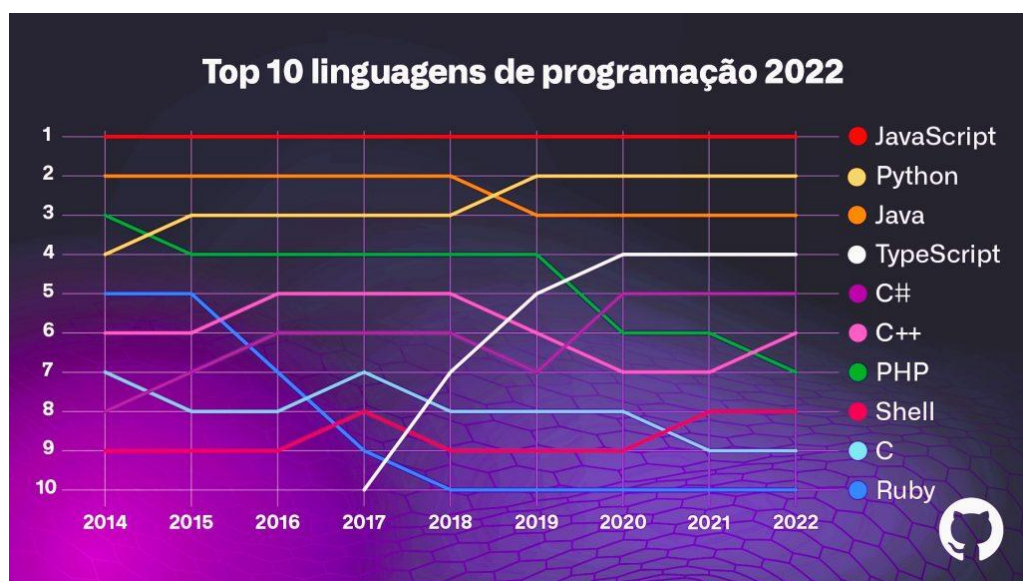


Imagem 1 - As 10 Linguagens de Programação mais utilizadas em 2022

2.2. Programação Orientada a Objetos

A Programação Orientada a Objetos (POO) é um paradigma de programação que utiliza "objetos" para a modelação de dado. Este paradigma destaca-se pela sua abordagem que permite que os desenvolvedores criem sistemas complexos de forma mais intuitiva e organizada. Os principais conceitos que fundamentam a POO incluem:

Objetos e Classes

Objetos: São instâncias de classes que representam entidades do mundo real, estes contêm atributos (dados) e métodos (comportamentos).

Classes: Servem como moldes ou uma espécie de *templates* para a criação de objetos. Uma classe define os atributos e métodos que os objetos criados a partir dela terão. Ao criar uma classe, define-se o tipo de dados que os objetos dessa classe irão conter (atributos), assim como as funcionalidades que poderão executar (métodos). São a estrutura básica para a construção de objetos na POO.

Podemos dividir as classes em três categorias principais: **classes normais**, **classes abstratas** e **interfaces**.

1. **Classes Normais:** São as classes mais comuns, podem ser instanciadas diretamente, ou seja, é possível criar objetos a partir delas. Essas classes podem ter atributos e métodos concretos, ou seja, com implementação real. Uma classe normal serve como um plano para criar objetos.
2. **Classes Abstratas:** São classes que não podem ser instanciadas diretamente. Não é possível criar objetos a partir de uma classe abstrata. Elas servem como base para outras classes. Uma classe abstrata pode ter métodos abstratos, ou seja, métodos, implementados ou não, para serem implementados pelas classes filhas.
3. **Interfaces:** São um tipo de classe especial que define um contrato ou conjunto de métodos que uma classe deve implementar. Uma interface não pode conter implementação de métodos; ela apenas define as assinaturas dos métodos. As classes que implementam uma interface são obrigadas a fornecer as implementações desses métodos, permitindo assim que diferentes classes compartilhem um conjunto comum de funcionalidades, sem a necessidade de uma hierarquia de herança.

A Programação Orientada a Objetos tem 4 pilares principais:

Encapsulamento:

O encapsulamento refere-se à prática de restringir o acesso direto aos dados de um objeto e, em vez disso, expor métodos que permitem manipular os dados. Isso protege o estado interno do objeto e garante que ele só possa ser modificado de maneiras controladas. Com o encapsulamento, é possível

criar classes que escondem os seus detalhes internos, expondo apenas o que é necessário para o funcionamento correto de cada projeto.

Herança:

A herança é um mecanismo que permite que uma classe herde características (atributos e métodos) de outra classe. Isso promove a reutilização e a otimização de código, permitindo que novos objetos sejam criados com base em objetos existentes.

Polimorfismo:

O polimorfismo permite que métodos em diferentes classes sejam chamados pelo mesmo nome, mas tenham comportamentos diferentes, dependendo do objeto com o qual se chama o método. O polimorfismo torna o código mais flexível e extensível.

Abstração:

A abstração consiste em simplificar a complexidade, ocultando os detalhes desnecessários e mostrando apenas as características essenciais de um objeto. Através da limpeza gerada pela implementação deste pilar, os desenvolvedores têm maior concentração em interações de alto nível, sem se perderem em detalhes de implementação.

2.3. UML e a Programação Orientada a Objetos

A UML (Unified Modeling Language) é uma linguagem de modelagem visual amplamente utilizada na engenharia de software para especificar, visualizar, construir e documentar sistemas. Com a padronização da sua notação gráfica, esta linguagem facilita a comunicação entre os membros de uma equipa de desenvolvimento, permitindo que os conceitos complexos de programação orientada a objetos (POO) sejam representados de forma clara e compreensível.

A UML inclui vários tipos de diagramas que são fundamentais para a modelagem de sistemas orientados a objetos, no presente projeto será explorado o diagrama de classes. Este representa as classes do sistema, os seus atributos, métodos e os relacionamentos entre elas, como herança e associação. O diagrama é

fundamental para a definição da estrutura do sistema e a organização das respetivas classes.

2.4. Design Patterns

Para o desenvolvimento de projetos com o uso de Programação Orientada a Objetos, a utilização de padrões de design é de importância extrema para alcançar um código modular, reutilizável e fácil de manter. Estes padrões fornecem diretrizes claras que ajudam a evitar problemas comuns de design e promovem uma melhor organização do sistema.

Um exemplo prático disso está na abordagem de separação de responsabilidades, aplicada através das arquiteturas MVC (Model-View-Controller) e N-Tier. Estas metodologias permitem distinguir claramente entre a lógica de negócio, o acesso a dados e a interface do utilizador.

No entanto, é importante reconhecer situações onde soluções aparentemente convenientes podem levar a “anti-patterns”, ou seja, práticas que violam princípios fundamentais de boa estruturação e que podem comprometer a escalabilidade, a segurança e a consistência do código. Ou seja, conclui-se que não existe uma receita a seguir no que toca à estruturação de código, não havendo uma metodologia certa ou errada mas sim métodos e conceitos que se aplicarmos de maneira correta no código produzido são cruciais.

Uma das figuras mais importantes no avanço dos design patterns é Erich Gamma, um dos autores do famoso livro Design Patterns: Elements of Reusable Object-Oriented Software. Segundo o mesmo: "Good designs are reusable. A reusable design is more complex than one that is disposable."



Imagem 2 - Erich Gamma

2.4.1. MVC (Model-View-Controller)

O padrão de design Model-View-Controller (MVC) é amplamente utilizado no desenvolvimento de softwares para estruturar aplicações de forma a promover principalmente a separação de responsabilidades, consequentemente facilitando a manutenção, a escalabilidade e a reutilização do código. O padrão MVC divide o sistema em três componentes principais: Model, View e Controller.

A camada Model funciona como o núcleo do sistema, responsável pela gestão dos dados e da lógica de negócio. Esta camada lida diretamente com a manipulação e o armazenamento das informações, garantindo que as regras de negócio sejam cumpridas.

A View é a camada de apresentação que exibe as informações ao utilizador. Esta camada por sua vez é responsável por capturar as interações do utilizador. Esta camada, não contém lógica de negócio, mas sim apenas código necessário para a exibição e formatação dos dados.

O Controller atua como intermediário entre a View e o Model. É nesta camada que são processadas as entradas do utilizador capturadas pela View, aplica as regras de negócio necessárias e atualiza tanto o Model quanto a View conforme apropriado.

2.4.2. N-Tier

O padrão N-Tier, padrão na qual é baseado o presente projeto, é uma arquitetura de software que promove a separação de responsabilidades ao dividir uma aplicação em múltiplas camadas lógicas. Esta abordagem é amplamente utilizada no desenvolvimento de sistemas, com a inclusão de programação orientada aos objetos, devido à sua capacidade de organizar aplicações de forma modular e escalável. A letra “N” no nome refere-se ao número de camadas, que pode variar dependendo das necessidades do sistema, sendo três o número mínimo no mais genérico modelo N-Tier.

A camada de apresentação (frontend) é responsável por interagir com o utilizador final. Nesta camada são processadas as entradas e exibidos os resultados gerados pela aplicação. Apesar de ser o ponto de contacto com o utilizador, a camada de apresentação deve ser mantida o mais independente possível da lógica de negócio, atingido consequentemente os maiores níveis de segurança possível do sistema através do distanciamento do utilizador com o sistema.

A camada de lógica de negócio é o núcleo do sistema, responsável por implementar as regras de negócio e os processos específicos da aplicação. Ela processa as entradas provenientes da camada de apresentação, aplica as validações necessárias, e comunica-se com a camada de dados para leitura ou escrita de informação.

A camada de dados é responsável por gerir o acesso e a manipulação dos dados armazenados em bases de dados ou outras fontes de informação. É aqui que residem as instruções específicas para consultar, inserir, atualizar ou eliminar dados. A camada de dados deve ser projetada de forma a minimizar o acoplamento com as outras camadas, permitindo que alterações na estrutura dos dados não afetem as restantes partes da aplicação.

O padrão N-Tier estabelece que cada camada comunique apenas com a camada adjacente. Por exemplo, a camada de apresentação não deve nunca aceder diretamente aos dados, em vez disso, interage com a camada de lógica de negócio, que por sua vez se comunica com a camada de dados. Este fluxo ordenado promove a organização e o encapsulamento.

Além das três camadas fundamentais – apresentação, lógica de negócio e dados – é comum a inclusão de uma camada de objetos de negócio em arquiteturas N-Tier. Esta camada é adjacente a todas as outras camadas referidas e é responsável por encapsular as entidades e regras do domínio da aplicação. Ela contém as classes que representam os objetos do sistema, como por exemplo, Paciente, Consulta ou Hospital, no caso de um sistema de saúde. Estas classes não apenas modelam os dados, mas também podem conter métodos que implementam comportamentos específicos.

É também usual o uso de uma camada responsável por Tratar problemas, também adjacente às restantes, onde são implementados métodos que gerem

exceções e problemas específicos do projeto, facilitando assim a gestão e a utilização do mesmo.

2.4.2.1. Vantagens do uso da N-Tier

A separação em camadas numa arquitetura N-Tier oferece diversas vantagens que tornam os sistemas mais robustos e fáceis de manter. Uma das principais vantagens é a manutenção e atualização simplificadas, pois mudanças realizadas numa camada não afetam diretamente as outras, facilitando a evolução e correção do sistema ao longo do tempo. Além disso, a escalabilidade é outro benefício significativo, uma vez que cada camada pode ser dimensionada separadamente.

Outro ponto relevante é a reutilização de componentes, especialmente na camada de lógica de negócio. Esta arquitetura promove também maior segurança ao encapsular a camada de dados, protegendo-a contra acessos diretos e garantindo que todas as interações com os dados sejam realizadas através das camadas apropriadas, reforçando a proteção do sistema.

2.5. Princípios SOLID

Os Princípios SOLID são um conjunto de cinco princípios de design de software que visam tornar o código mais compreensível, flexível e fácil de manter. O objetivo dos princípios é promover boas práticas que resultem em sistemas que possam ser facilmente alterados e expandidos sem comprometer a qualidade do software. A sigla SOLID é formada pelas iniciais dos cinco princípios, que são explicados a seguir:

S - Single Responsibility Principle (Princípio da Responsabilidade Única)

O Princípio da Responsabilidade Única afirma que uma classe deve ser responsável por apenas uma tarefa ou funcionalidade. Isso significa que cada classe deve focar em uma única preocupação ou funcionalidade do sistema. Quando uma classe assume múltiplas responsabilidades, fica mais difícil de entender e de modificar o seu código, havendo maior instabilidade correspondente ao crescimento da quantidade de código do projeto.

O - Open/Closed Principle (Princípio Aberto/Fechado)

O Princípio Aberto/Fechado diz que devemos poder adicionar novas funcionalidades à classe sem precisar alterar seu código original. Em vez de modificar a classe existente, devemos estender sua funcionalidade.

L - Liskov Substitution Principle (Princípio da Substituição de Liskov)

O Princípio da Substituição de Liskov afirma que se uma classe B é uma subclasse de A, então podemos substituir uma instância de A por uma instância de B sem que o programa se comporte de maneira inesperada.

I - Interface Segregation Principle (Princípio da Segregação de Interface)

O Princípio da Segregação de Interface estabelece que devemos criar interfaces menores e mais específicas. Isso torna as implementações mais simples e permite que os clientes usem apenas as partes da interface que realmente precisam.

D - Dependency Inversion Principle (Princípio da Inversão de Dependência)

O Princípio da Inversão de Dependência sugere que as classes de alto nível não se devem preocupar com as implementações concretas das classes de baixo nível.

2.6. Testar Código

Testar o código é uma das práticas mais importantes no desenvolvimento de software, pois garante a qualidade e confiabilidade das aplicações. Através dos testes, é possível identificar e corrigir erros de forma precoce, antes que o software seja lançado ou utilizado por outros. Testes bem elaborados não apenas verificam se o código funciona corretamente, mas também asseguram que mudanças futuras no sistema não quebrem funcionalidades existentes, promovendo uma manutenção mais eficiente.

3. Trabalho Desenvolvido na 1ª Fase

3.1. Análise e Modelação das Classes

3.1.1. Utilização de UML

No desenvolvimento do meu projeto, optei por utilizar, por ter licença académica oferecida pelo Instituto Politécnico do Cávado e do Ave, o Visual Paradigm e a linguagem UML para a definição das classes e da estrutura de interações entre elas. Essa escolha foi motivada pela necessidade de garantir uma abordagem clara e organizada na modelagem do sistema. Por ter uma notação gráfica padronizada, foi uma ferramenta facilitadora para a visualização das relações e dos comportamentos das classes, isto pois, foi possibilitada uma visualização mais dinâmica do sistema.

A gestão adequada das classes é imprescindível na programação orientada a objetos, pois classes bem definidas promovem a modularidade do sistema, e por isso, com o uso do Visual Paradigm, foi facilitada a minha decisão de quais classes criar e que relações estas teriam entre si.

Este meu estudo prévio de modulação de classes deve-se ao meu entender de que um dos pilares da programação é o pensamento crítico, pois acredito que este desempenha um papel crucial na elaboração de um projeto claro e coeso. Penso que o pensamento crítico e a necessidade de pensar bem antes de agir são fundamentais e por isso achei por bem esta projeção inicial do sistema.

3.1.1.1. 1ª Versão: Primeira Visão do Projeto

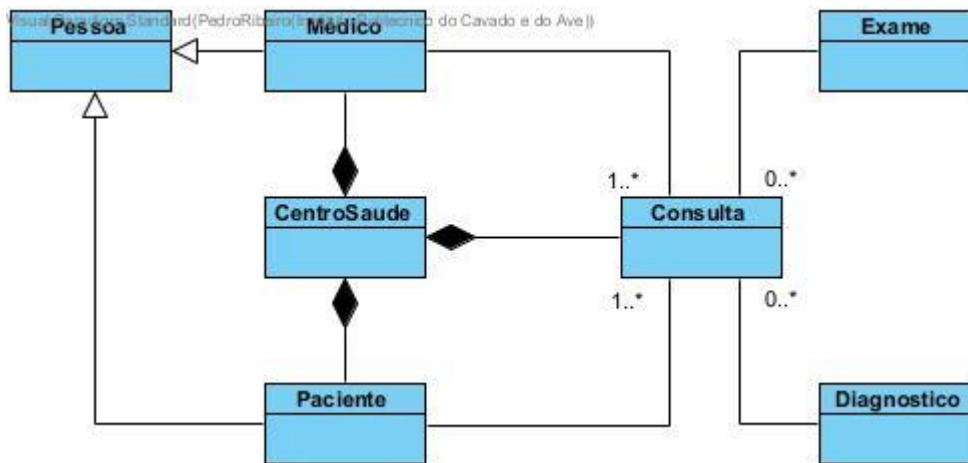


Imagem 3 - Primeiro Diagrama de Classes, em UML

Na imagem acima do diagrama de classes desenvolvido, estão representadas as diferentes classes que serão criadas e elaboradas no projeto, é possível observar também 3 tipos de interações entre essas classes diferentes.

Associação: Representada por uma linha sólida que conecta duas classes, a associação indica que existe uma relação entre elas. A multiplicidade é adicionada na extremidade da linha para indicar quantos objetos de uma classe podem estar associados a outro da outra classe. Por exemplo, a notação "1..*" ao lado da classe indica que essa classe pode ter uma ou mais instâncias associadas à classe conectada.

Composição: Representada por uma linha sólida com um losango preenchido na extremidade que aponta para a classe "pai". A composição implica uma relação onde a parte "filho" não pode existir sem a existência do "pai".

Herança: Representada por uma linha sólida com uma seta vazia na extremidade que aponta para a classe "pai", a herança indica que uma classe (a classe filha) herda características e comportamentos de outra classe (a classe pai). Essa relação permite a reutilização de código e a especialização de classes.

Abaixo explico o uso dos diferentes tipos de relações entre classes, presentes no diagrama de classes desenvolvido:

1. **Ligação entre Médico, Paciente e Pessoa:** A relação adotada entre estas classes é a herança pois o Médico e o Paciente têm características/atributos iguais, por este motivo e motivos lógicos, foi possível a criação de uma classe “pai” – a classe
2. **Relação entre Centro de Saúde e Médico, Paciente e Consulta:** Entre estas classes decidi utilizar a relação de composição. Isto deve-se ao facto de acreditar que as classes de Médico, Paciente e Consulta não podem existir sem o Centro de Saúde, pois esses profissionais e pacientes estão todos associados a uma instituição específica.
3. **Relação entre Médico e Consulta:** Logicamente um médico pode ter uma ou muitas consultas e por isso optei pela relação de associação, representando assim que um médico atende múltiplos pacientes em diferentes consultas.
4. **Relação entre Paciente e Consulta:** Assim como na relação associativa Médico e Consulta, a relação entre Paciente e Consulta é representada de modo que um paciente possa ter uma ou muitas consultas.
5. **Relações da Consulta com Exame e Diagnóstico:** Nas ligações da Consulta com Exame e Diagnóstico, optei também pela relação associativa, mas nesta situação acredito que uma consulta pode ter 0 ou muitos de ambos e não 1 ou muitos como nas relações associativas anteriormente abordadas.

3.1.1.2. 2ª Versão: Revisão das classes e aprimoramento da abstração e polimorfismo do Código

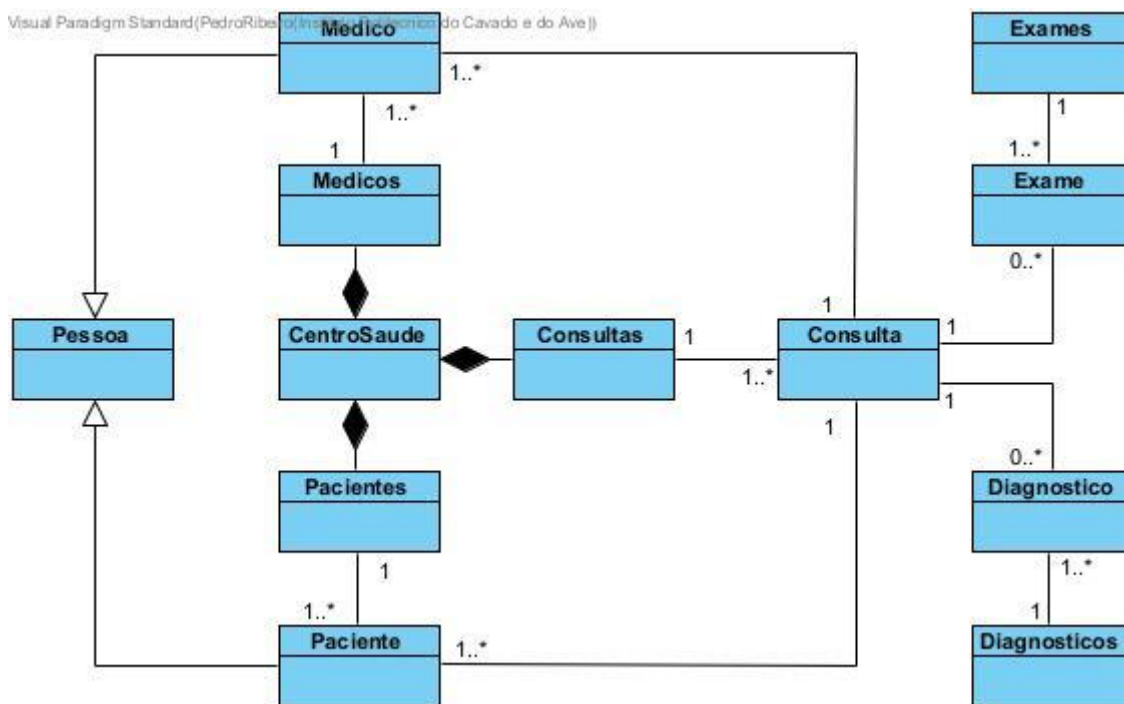


Imagem 4 - Segundo Diagrama de Classes, em UML

Como abordado na revisão da leitura, a abstração, que é um dos pilares fundamentais da Programação Orientada a Objetos, é um processo que visa a simplificação da complexidade, ao ocultar os detalhes desnecessários e ao mostrar apenas as características essenciais de um objeto. Ou seja, a abstração permite trabalhar com funcionalidades de uma maneira mais simplificada, já que a compreensão do código quer pelo próprio produtor como de terceiros é facilitada. Isso é feito, por exemplo, a partir da criação de classes que representam as diferentes entidades intuitivas do mundo real (como já elaborado na primeira versão do diagrama UML produzido) fornecendo métodos que permitem interagir com outras, mas também a partir da criação de entidades que não tem grau de intuição tão elevado, mas que facilita a gestão de um sistema.

No presente projeto explorei o uso de algumas das classes no plural para agrupar várias instâncias dos objetos da qual escolhi fazer as classes no plural (paciente, medico, diagnostico, exame, consulta). Percebi que algumas classes, começavam a ter demasiados métodos, e percebi que inicialmente pode ser tolerável mas que percebi que a longo prazo, com a evolução do projeto, não seria sustentável

a quantidade de métodos exagerada que algumas classes estavam a implementar. Resolvi então mover a responsabilidade para as novas classes, o que provocou uma melhora clara na **abstração** do código.

Antes da implementação das classes agrupadoras, cada classe possuía métodos que realizavam uma ampla gama de tarefas, e muitas dessas tarefas estavam relacionadas a manipular arrays de objetos. Notei que o código ficava muito fragmentado e desorganizado, pois as funções não estavam centralizadas em uma única classe de gestão, mas espalhadas entre diversas classes.

Antes da implementação de classes agrupadoras: Em classes como Medico, Paciente, Consulta, etc., verificava-se métodos para adicionar, remover ou verificar existência de um item existe em seus respectivos arrays de objetos. O código tornou-se difícil de manter, pois, adicionar uma nova funcionalidade, provocaria várias modificações em vários pontos do código, o que aumentava a chance de introduzir erros (e.g. mudar o tipo de dados de "Exame", exigiria a mudança de funções de gestão dos mesmos em todas as classes com arrays de "Exame"). Outro problema era a duplicação de código. Por exemplo, as mesmas operações de adição, remoção ou verificação de existência estavam espalhadas por várias classes.

Depois da implementação das classes agrupadoras: Ao introduzir as classes agrupadoras (Medicos, Consultas, Diagnosticos, Exames e Pacientes), foi possível agrupar e encapsular a lógica relacionada à gestão de entidades em uma única classe. Cada uma dessas novas classes passa a ser responsável por manipular o conjunto de objetos relacionados, ou seja, foi centralizada a gestão de um tipo específico de objeto em uma única classe.

Foi abordada também, na revisão da leitura, a importância de um dos pilares da Programação Orientada a Objetos, o polimorfismo. Este pilar é um conceito fundamental que permite que diferentes classes implementem os mesmos métodos de maneiras distintas, com base nas suas necessidades específicas. Percebi que em todas as classes agrupadoras que criei, havia métodos de adição, remoção e verificação de existência. Anteriormente em cada classe o método tinha um nome diferente, como por exemplo, "AdicionarMedico", "RemoverMedico" e "ExisteMedico", mas percebi que estaria a ignorar um dos principais pilares da programação orientada aos objetos fazendo isto, foi então que decidi renomear todos estes métodos como "Adicionar", "Remover" e "Existe" tendo assim os mesmos nomes em várias classes.

Ao fazer isto notei várias melhorias no meu código e na sua implementação. Notei uma maior facilidade de uso e uma redução da complexidade do código, pois ao usar os mesmos nomes de métodos em diferentes classes, tornou-se mais fácil entender e utilizar o código. Em vez de memorizar nomes de métodos diferentes para cada classe, passou a ser necessário usar simplesmente "Adicionar", "Remover" e "Existe" em qualquer contexto.

Exemplo de implementação dos métodos "Adicionar", "Remover" e "Existe":

```
public bool Existe(Consulta consulta)
{
    for (int i = 0; i < numConsultas; i++)
    {
        if (listaConsultas[i].Equals(consulta)) return true;
    }
    return false;
}

public bool Adicionar(Consulta consulta)
{
    if (consulta == null || Existe(consulta) || numConsultas >= maxConsultas) return false;
    listaConsultas[numConsultas++] = consulta;
    return true;
}

public bool Remover(Consulta consulta)
{
    for (int i = 0; i < numConsultas; i++)
    {
        if (listaConsultas[i].Equals(consulta))
        {
            for (int j = i; j < numConsultas - 1; j++)
            {
                listaConsultas[j] = listaConsultas[j + 1];
            }
            listaConsultas[--numConsultas] = null;
            return true;
        }
    }
}
```

L. Cod. 1 - Exemplo de implementação dos métodos "Adicionar", "Remover" e "Existe"

3.2. Estrutura das Classes e Lógica de Implementação

O desenvolvimento do sistema de gestão do Centro de Saúde envolveu a criação de classes que representam as entidades principais do problema, como Pacientes, Médicos, Consultas, Exames e Diagnósticos. O objetivo foi criar uma estrutura modular que permitisse manipular de forma eficiente os dados de cada entidade, garantindo a aplicação dos princípios da Programação Orientada a Objetos.

Classes Criadas e os Relacionamentos

Classe CentroSaude:

A classe CentroSaude representa a entidade principal. Apesar disso, percebi que seria bastante extenso e redundante ter métodos de gestão de todas as outras classes na classe CentroSaude apenas por esta ser a principal, então conectei apenas as classes Médico, Paciente, e Consulta, pois estas classes dependem logicamente da existência do Centro de Saúde. Acredito que esta composição permite uma gestão mais centralizada e organizada dos dados.

Classe Pessoa:

A classe Pessoa, pela relação de herança serve como base para as classes Médico e Paciente, representa atributos comuns - nome, data de nascimento e género. Assim, Médico e Paciente, classes-filhas, herdam as características gerais de Pessoa, mas cada uma delas possui atributos e métodos específicos.

Classe Medico e Paciente:

As classes Medico e Paciente são classes-filhas da classe Pessoa, que utilizam o conceito de herança para partilhar atributos e métodos comuns, como nome, data de nascimento e género. A herança permite que cada uma dessas classes adicione características específicas: a classe Medico inclui atributos como especialidade e número de identificação, enquanto a classe Paciente contém uma instancia da classe Diagnosticos e uma instancia da classe Consultas, tendo variáveis default para o tamanho dos arrays criados nestas instâncias a quando da criação de um objeto da classe Paciente. Esta abordagem facilita a reutilização de código e mantém a estrutura do sistema mais organizada.

```
public class Medico : Pessoa
```

L. Cod. 2 – Assinatura da classe Medico

```
public class Paciente : Pessoa
```

L. Cod. 3 – Assinatura da classe Paciente

Classe Consulta:

Esta é a classe responsável por representar uma consulta médica dentro do centro de saúde. Ela contém dados como a data da consulta, o paciente e o médico responsáveis, bem como um diagnóstico, exames realizados e o custo total da consulta. O uso da `totExames` permite, por definição, limitar o número de exames associados a uma consulta. A classe inclui métodos para adicionar e remover exames, chamando apenas os métodos criados na classe `Exames`:

```
public bool AdicionarExame(Exame exame)
{
    return exames.Adicionar(exame);
}
```

L. Cod. 4 – Exemplo de método que chama um método presente em outra classe

Além disso com esta classe é possível calcular o custo total da consulta, a partir de um método que soma à variável do custo estabelecido na própria classe o valor de saída do método que calcula o custo total dos exames presente dentro da classe `Exames`.

```
public double CalcularCustoTotal()
{
    return exames.CalcularCustoTotal() + custo;
}
```

L. Cod. 5 – Método que calcula o custo total presente na classe Consulta

Classes Exame e Diagnostico

As classes Exame e Diagnostico são relacionadas aos procedimentos entre médicos e pacientes. A classe Exame é responsável por armazenar os detalhes sobre os exames realizados, incluindo o tipo, o resultado e o custo associado. Através de seus construtores e propriedades, ela permite a criação e modificação de exames. Já a classe Diagnostico representa o diagnóstico médico realizado durante uma consulta, contendo informações como a descrição do diagnóstico e a data em que foi realizado. Essa classe também permite adicionar informações adicionais à descrição, garantindo flexibilidade no registros.

Classes Medicos, Pacientes, Consultas, Exames e Diagnosticos

As classes como Medicos, Pacientes, Consultas, Exames e Diagnosticos foram criadas para agrupar múltiplas instâncias das entidades Medico, Paciente, Consulta, Exame e Diagnostico, respetivamente. Além disso, com a aplicação do polimorfismo, e ao renomear os métodos de adição, remoção e verificação de existência com a mesma nomenclatura, a lógica de gestão de cada entidade ficou mais centralizada e facilitada, pois o código ficou simplificado.

Métodos Equals e GetHashCode

Em todas as classes além da classe CentroSaude, foram aplicados os métodos Equals e GetHashCode. Na classe CentroSaude não senti a necessidade de o fazer pois é uma classe que terá apenas um objeto já que o objetivo do projeto é a gestão de apenas um Centro de Saúde. A implementação deste método por exemplo na classe Pessoa serve de base para as verificações de igualdade em Médico e Paciente, já que esta tem como classe base a classe Pessoa. Este método simplifica a lógica de busca e comparação de dados, já que permite comparar objetos da mesma classe de forma mais intuitiva.

Criação da interface IGestão<T>

Resolvi também fazer a criação de uma interface - IGestao<T> - com o objetivo principal de proporcionar uma estrutura genérica e reutilizável que facilitasse a gestão de diversos tipos de entidades - consultas, exames, diagnósticos, pacientes e medicos - no sistema de gestão de centro de saúde.

Inicialmente, criei uma interface IGestao<obj> para organizar as operações de gestão no sistema de centro de saúde. O objetivo era implementar métodos comuns,

como Adicionar, Eliminar e Existe, aplicáveis a diferentes tipos de entidades. No entanto, ao usar `IGestao<obj>`, percebi que isso limitava a flexibilidade, pois nas classes que implementavam esta interface, o parâmetro dos métodos tinha de ser sempre do tipo `obj`.

Essa abordagem com `obj` exigia que fizesse verificações e conversões de tipo constantes, aumentando o risco de erros e tornando o código menos limpo e intuitivo. Além disso, ia contra o propósito da Programação Orientada a Objetos e da reutilização de código, que era um dos meus objetivos principais.

Assim, decidi reformular a interface como `IGestao<T>`, utilizando um tipo genérico `T` em vez de `obj`. Esta mudança permitiu-me especificar o tipo correto de dados ao implementar a interface em cada classe, eliminando a necessidade de conversões desnecessárias e assegurando que cada operação é aplicada ao tipo adequado.

4. Análise de resultados/Reflexão (1ª Fase)

4.1. Abordagens Exploradas, mas deixadas de lado:

Durante o desenvolvimento, percebi que a utilização de coleções mais avançadas, como listas, poderia facilitar a gestão de dados, especialmente quando se trata de armazenar múltiplos itens, como exames e diagnósticos, sem forçar um tamanho máximo. Contudo, optei por simplificar a estrutura inicialmente, pois foi uma estrutura de dados que não foi abordada em contexto de aula, por isso e por entender que a definição da mais adequada estrutura de dados não é o objetivo desta fase do trabalho, optei pela utilização arrays, sem recorrer a listas, o que pode ser uma área de melhoria para futuras fases do projeto.

Apesar de já ter aplicado boas práticas de programação, como o uso dos pilares fundamentais da Programação Orientada a Objetos (POO), percebi que algumas áreas poderiam ser mais otimizadas. Como por exemplo, o que já foi abordado a cima, a aplicação de coleções dinâmicas que poderia melhorar o desempenho e a flexibilidade do código.

4.2. Considerações sobre Decisões Tomadas:

Como, a meu ver, o objetivo principal da fase inicial era definir a estrutura e implementar funcionalidades centrais (como criar, modificar e visualizar dados), concentrei-me nessas funções essenciais. Embora haja várias outras funcionalidades que poderiam ser implementadas para expandir o sistema, optei por deixar essas tarefas mais específicas para a fase posterior, garantindo uma base sólida antes de adicionar complexidade.

Um dos pontos que considerei ter tomado uma boa decisão durante o desenvolvimento foi o uso de classes agrupadoras. Essa abordagem ajudou na minha aprendizagem e na organização do código, facilitando a manutenção do sistema, além de permitir a aplicação do polimorfismo de maneira eficaz. A padronização dos métodos Adicionar, Remover e Existe foi uma decisão estratégica, pois permitiu um design mais otimizado para a gestão de dados em diferentes contextos do sistema, como consultas e diagnósticos. Além disso, penso que tomei também uma decisão otimizadora em criar uma interface `IGestão<T>`, pois padronizei operações de gestão, permitindo a implementação consistente dos métodos Adicionar, Remover e Existe, independentemente da classe.

5. Conclusão – 1ª Fase

A primeira fase do projeto prático de Programação Orientada a Objetos foi essencial para a aplicação dos conceitos e princípios fundamentais desta área. O processo de implementação, desde a modelação e conceção das classes até à estruturação dos principais métodos, representou uma oportunidade importante para consolidar o conhecimento teórico abordado nas últimas aulas de Programação Orientada a Objetos. Ao longo do desenvolvimento, desafiei-me a manter um equilíbrio entre a complexidade da modelação e a simplicidade da implementação, sempre com o objetivo de tornar o código otimizado e flexível. Esta fase foi, a meu ver, marcada pelo aprofundamento de conceitos como encapsulamento, herança, polimorfismo e abstração, elementos cruciais para a estruturação de um sistema eficiente e intuitivo.

Considero um dos conceitos mais importantes que adquiri nesta fase inicial do trabalho as classes que representam um array de outras classes. A introdução de classes agrupadoras, como Médicos, Pacientes e Consultas, foi uma decisão importante para a otimização do código. Esta abordagem permitiu centralizar as operações principais, como a adição, remoção e verificação de elementos, em classes específicas. Outro ponto que sinto que foi essencial para a produção do código mais limpo e conciso possível, foi a aplicação do polimorfismo para padronizar os métodos Adicionar, Remover e Existe, consegui não só reduzir a complexidade do código, mas também simplificar a sua utilização, garantindo que a interface se mantinha consistente entre diferentes entidades.

Penso que o uso da Herança foi também interessante no meu projeto, pois com a utilização de uma classe-base Pessoa, da qual derivam Médico e Paciente, otimizei o código com a partilha de atributos e métodos comuns, evitando a duplicação de código.

Outro ponto fundamental foi a criação da interface IGestao<T>, que permitiu uma gestão mais genérica e flexível das operações de manipulação dos dados. Com a interface padronizei as funções principais para as classes que as usam.

Apesar da minha satisfação com o progresso e com a estrutura alcançada até ao momento, reconheço que existem áreas de melhoria, sobretudo no que diz respeito à robustez das funcionalidades. Focar-me nestes aspetos será essencial para a

segunda fase, onde espero explorar ainda mais a eficiência e a capacidade de expansão do sistema. Em suma, considero esta primeira fase positiva e recompensadora, permitindo-me aprender de forma prática e aprofundada sobre as boas práticas em Programação Orientada a Objetos.

6. Trabalho Realizado na 2ª Fase

6.1. Análise da 1ª Fase de Trabalho

Após a implementação da primeira fase do projeto e das novas aprendizagens relacionadas ao N-Tier e aos parâmetros SOLID, foram identificadas diversas falhas no código. Estas falhas comprometiam a organização, a escalabilidade e a manutenção do sistema, desviando-se das boas práticas recomendadas. Com a arquitetura N-Tier foi possível perceber a importância de separar a aplicação em camadas independentes, pois desta forma garantimos uma clara divisão de responsabilidades. Com a ausência de uma metodologia de estruturação utilizada na primeira fase foi possível observar a falta de separação clara entre camadas, ou seja, métodos relacionados à lógica de negócio, como a criação, remoção, e verificar existência estão misturados com a lógica de apresentação ou diretamente na camada de acesso a dados. Esta mistura cria dependências desnecessárias entre camadas, dificultando a manutenção e os testes individuais de cada camada. Essas dependências criam também um código menos seguro, pois é banalizada a importância da privacidade dos dados. Na primeira fase era feito o acesso direto a dados a partir da camada de apresentação, era possível fazer chamadas diretas a métodos que manipulam dados (ex.: criação e eliminação de consultas) em vez de haver métodos/serviços intermediários que impedem esse acesso direto. Este acesso direto quebra a independência das camadas e pode gerar inconsistências e falta de segurança nos dados.

6.2. Estruturação e Divisão em Camadas do Projeto

Por fim, depois da aprendizagem retirada de Design Patterns, no desenvolvimento do projeto, optei por uma estrutura modular e bem definida, baseada no padrão N-Tier e nos princípios de separação de responsabilidades. A organização em várias camadas permite um sistema mais fácil de manter, garantindo uma clara distinção entre cada camada.

6.2.1. Estrutura Geral da Solução

A solução está organizada em 5 projetos principais, cada um responsável por uma camada específica do sistema. A camada de **Dados**, a qual, contém as entidades, interfaces e repositórios que interagem diretamente com os dados, a camada de **ObjetosNegocio**, que engloba objetos de negócio que representam a lógica central do sistema, a **Camada RegrasNegocio**, responsável pelas regras e validações aplicadas à lógica do negócio, a camada **TrataProblemas**, que define as exceções personalizadas, a camada **TestesUnitrios** e a camada de **GestaoCentroSaude**, que é o projeto principal, responsável pela execução e pela interface com o utilizador.

A camada Dados é responsável por toda a interação com a base de dados ou qualquer fonte de dados persistentes. Aqui, adotei uma estrutura clara dividida em três partes (3 pastas). As Entidades, contém as classes que representam o elemento base de toda a base de dados, as estruturas de dados (por exemplo, Paciente, Medico e Consulta). Cada uma destas classe sem os atributos necessários à formação de objetos únicos e são basicamente o ponto de partida de todo o código. Além destes atributos nestas classes são definidos os métodos essenciais a mudanças individuais de cada objeto destas entidades, os construtores padrão e construtores com os parâmetros das classes light referentes a cada entidade. Nas interfaces, outra divisão da camada de dados, são declaradas as operações que devem ser implementadas pelos repositórios. Com estas interfaces garanto a abstração, facilitando testes unitários e futuras alterações. Por sua vez nos Repositorios, é onde fiz a Implementação das interfaces, responsáveis pela interação direta com os dados. Estas são as classes onde é possível gerir um grupo de entidades (por exemplo a classe consultasRepositorio tem como objetivo a gestão de um conjunto de consultas) , cada repositório tem também um id associado.

A Camada ObjetosNegocio, é responsável pela criação de objetos de negócio que encapsulam as regras centrais do sistema. Nesta classe são basicamente criadas versões simplificadas das entidades criadas na camada de dados (por exemplo: PacienteLight). Esta abordagem serve para transportar dados entre camadas, garantindo melhor performance e evitando a exposição desnecessária de detalhes. A camada **ObjetosNegocio** não tem dependências

para as outras camadas, respeitando assim o princípio da independência lógica.

A **Camada RegrasNegocio** centraliza todas as validações e regras específicas do negócio, desde regras de negócio como por exemplo que um paciente só pode ser adicionado na presença de um médico. Aqui, dividi a estrutura em Interfaces, que definem os métodos necessários para as regras a serem aplicadas e Servicos, que implementam as interfaces e contêm a lógica concreta de validação.

A **Camada TrataProblemas** foi criada para implementar exceções personalizadas e centralizar o tratamento de erros. Esta abordagem proporciona mensagens personalizadas e adequadas ao contexto do sistema (por exemplo: `EntidadeJaExisteException`, lançada quando se tenta adicionar uma entidade que já existe).

A **Camada GestaoCentroSaude**, atua como ponto de entrada do sistema, representando a camada de apresentação. Aqui, encontra-se o ficheiro `Program.cs`, responsável pela execução do programa e interação inicial com o utilizador.

A camada **TestesUnitários** foi implementada para garantir a qualidade e a robustez do sistema através de testes automatizados. A camada é organizada em três classes principais, cada uma correspondente a uma das áreas de funcionalidades do sistema: `GestaoConsultas`, `GestaoDiagnosticos` e `GestaoExames`. Cada uma dessas classes de teste possui métodos específicos para verificar diferentes aspetos do código, como por exemplo uma correta adição, além da validação de falhas como por exemplo a inserção de dados inválidos.

6.3. Mudança de estrutura de dados (Array para Lists)

No desenvolvimento do sistema, optei por substituir os arrays por listas, principalmente para melhorar a flexibilidade e escalabilidade do código. Embora os arrays seja, considerado por muitos a mais versátil estrutura de dados, com grande importância na manipulação de dados em cenários específicos, a decisão de migrar para listas foi tomada devido às várias vantagens que estas

oferecem no contexto específico em que me insiro, especialmente em um ambiente dinâmico e com constante necessidade de manipulação de dados.

Reconheço que as listas são estruturas mais adequadas para cenários onde o número de elementos é variável e a gestão desses elementos deve ser feita de forma mais eficiente e simples. As listas permitem adicionar, remover e buscar elementos de forma mais intuitiva e sem a necessidade de redefinir tamanhos, como ocorre com os arrays.

É importante a referência da grandiosidade dos arrays em situações específicas. Eles são eficientes em termos de uso de memória e desempenho quando a quantidade de dados é fixada e o acesso rápido por índice é necessário. Um exemplo disso lecionado pelo professor Luís Ferreira durante as aulas seria a gestão de lugares do auditório do polo de tecnologia do Instituto Politécnico do Cávado e do Ave, os lugares são limitados, e ao contrário das listas este enche e é útil que assim o seja.

6.4. O uso dos identificadores

Nesta fase do trabalho, optei também pela adição do uso de identificadores no sistema, pois considero esta uma prática profundamente otimizadora para garantir a integridade e consistência dos dados ao longo de toda a aplicação. Os identificadores são usados para representar de forma única cada entidade ou objeto, permitindo um rastreamento eficiente e uma manipulação segura dos dados.

Durante as aulas de Programação Orientada a Objetos foi muitas vezes discutido o facto de uma estrutura de dados estar presente em várias entidades não ser algo eficiente principalmente pela sua insuficiência em lidar com a mudança. A primeira conclusão retirada neste sentido foi que a presença de várias estruturas de dados (várias Lists de diferentes entidades), na mesma classe faria com que essa classe se torna-se num “monstro”, com incontáveis métodos, tornando insustentável e completamente desorganizador. Além disso a presença de duas Lists que teoricamente seriam iguais em diferentes entidades é extremamente incongruente, pois a alteração duma não implica

diretamente a alteração da outra e por essa razão foi também rapidamente esquecida essa ideia.

Após as conclusões retiradas acima foi usada a abordagem de em vez de ter uma list de dados para cada entidade teria uma instancia de uma classe agrupadora, classe que gere a list da entidade em questão. Esta abordagem é sem dúvida mais funcional que a abordagem acima referida mas ainda assim não foi a estruturação escolhida. Esta forma de organizar o código torna-se a longo prazo insustentável pois a instância de uma classe que agrua um conjunto de entidades torna-se em um atributo muito pesado, daí surgir a abordagem com identificadores.

Os usos de identificadores consistem em referir em uma entidade especifica que um dos atributos é um conjunto de outra entidade (por exemplo: o Medico tem a si associado um conjunto de Consultas) o identificador do respetivo conjunto. Assim o identificador presente nos atributos da entidade seria igual ao identificador do conjunto de entidades respetivo. Exemplo:

```
public class Medico : Pessoa
{
    #region Atributos
    /// @brief Especialidade do médico.
    string especialidade;
    /// @brief Número de ordem do médico.
    int numeroCedula;
    /// @brief variável de permissão de gestão.
    bool podeTomarDecisoes;
    /// @brief identificador do repositório de consultas.
    int consultasId;
    #endregion "
```

L. Cod. 6 – Atributos de Medico

```
"public class ConsultasRepositorio : IRepositoryo<Consulta>
{
    #region Atributos
    /// @brief Contador estático usado para gerar IDs únicos para cada
    repositório.
    static int contadorRepositorio = 0;
    /// @brief ID único atribuído ao repositório atual.
```

```

int consultasRepositorioId;
/// @brief Lista que armazena as consultas associadas a este repositório.
List<Consulta> consultas;
#endregion"

```

L. Cod. 7 – Atributos de ConsultasRepositorio

A principal vantagem de utilizar identificadores é a capacidade de garantir que, cada objeto ou entidade seja, tratado de forma única, sem depender de atributos que podem ser alterados ao longo do tempo. O identificador torna-se, assim, a chave para relacionar os objetos em diferentes camadas e repositórios, facilitando o acesso e a atualização dos dados.

6.5. Funcionamento das Classe de Gestão (Camada Regras de Negócio)

```

"public class GestaoMedicos : IGestaoMedicos
{
    #region Atributos
    MedicosRepositorio medicoRepositorio;
    #endregion
    #region Construtores
    public GestaoMedicos()
    {
        medicoRepositorio = new MedicosRepositorio();
    }
    #endregion"

```

L. Cod. 8 – Atributos e Construtores de GestaoMedicos

As classes de gestão, presentes na camada de Regras de Negócio, como a GestaoMedicos, são responsáveis por controlar as operações com as entidades do sistema, funcionando como intermediárias entre a camada de lógica de negócio e a camada de dados. Cada classe de gestão corresponde a um conjunto de dados específicos e gere as operações associadas a essas entidades.

No caso da classe `GestaoMedicos`, ela implementa a interface `IGestaoMedicos`, o que garante que os métodos definidos para gerir os médicos sejam devidamente implementados.

A classe possui um atributo chamado `medicoRepositorio`, que é responsável pela interação com os dados dos médicos. Esse repositório é onde são armazenados e manipulados os dados reais, como adicionar, remover, listar ou consultar médicos. Através deste repositório, a classe de gestão consegue realizar operações sobre os dados sem expor a complexidade da camada de persistência.

O construtor da classe `GestaoMedicos`, por sua vez, inicializa o atributo `medicoRepositorio`, criando automaticamente uma nova instância de `MedicosRepositorio` sempre que uma instância de `GestaoMedicos` é criada. Isso assegura que a classe de gestão tenha acesso aos métodos de manipulação de dados desde o momento da sua criação.

No Main, a classe `GestaoMedicos` é instanciada para realizar operações sobre os dados dos médicos. Ao criar a instância de `GestaoMedicos`, estamos a criar, internamente, uma instância do repositório de médicos, o que permite manipular os dados dos médicos de forma eficiente e organizada. Por exemplo, podemos adicionar um novo médico, listar todos os médicos ou remover um médico específico, sem precisar de detalhes sobre como os dados são guardados ou recuperados. É também possível fazer o mesmo para outras listas de médicos. Além disso como já referido pela presença de um identificador associado a cada lista é possível associar determinada lista a um determinado objeto.

6.5.1. Aplicação de Regras de Negócio

```
public bool AtualizarCustoDoExame(int examId, double novoCusto,
MedicoLight medicoLight)
{
    if (medicoLight == null) throw new
ArgumentNullException(nameof(medicoLight));
    if (!medicoLight.PodeTomarDecisooes) throw new
MedicoNaoAutorizadoException();

    if (novoCusto < 0) return false;

    try
    {
```

```

        exameRepositorio.AtualizarCustoExame(exameId, novoCusto);
        return true;
    }
    catch (ColecaoNaoInicializadaException)
    {
        throw;
    }
    catch (Exception)
    {
        throw;
    }
}

```

L. Cod. 9 – Função de AtualizarCustoDoExame presente na GestaoExames

A função `AtualizarCustoDoExame` presente na classe `GestaoExames` tem como objetivo atualizar o custo de um exame específico, identificando-o através de um `exameId` e atribuindo-lhe um novo custo, desde que algumas condições sejam cumpridas. A primeira verificação é feita através do parâmetro `medicoLight` que não pode ser nulo, o que, se verdadeiro, lança uma exceção `ArgumentNullException`, garantindo que o médico responsável pela decisão esteja presente e seja válido. Em seguida, é verificado se o médico tem autorização para tomar decisões, através da propriedade `PodeTomarDecisoes` de `medicoLight`. Caso o valor seja `false`, a exceção `MedicoNaoAutorizadoException` é lançada, o que assegura que apenas médicos com a devida permissão possam atualizar o custo do exame.

Outro ponto importante da função é a validação do `novoCusto`, garantindo que ele seja maior que zero. Caso contrário, a função retorna `false`, prevenindo a atualização com um valor inválido. Se todas as condições forem atendidas, a função tenta realizar a atualização do custo no repositório de exames utilizando o método `AtualizarCustoExame`. No entanto, há um tratamento de exceções em que, se ocorrer uma `ColecaoNaoInicializadaException`, ela é relançada, mantendo a consistência do código. Uma exceção genérica também é utilizada para capturar outras exceções que possam surgir durante a operação.

```

public bool EliminarMedico(int medicoId, int password)
{
    if (password != 0000) return false;
    try

```



```

    {
        medicoRepositorio.Remover(medicoId);
        return true;
    }
    catch (ColecaoNaoInicializadaException)
    {
        throw;
    }
    catch (Exception)
    {
        throw;
    }
}

```

L. Cod. 10 – Funcao EliminarMedico presente em GestaoMedicos

Por outro lado, a função EliminarMedico presente na classe GestaoMedicos é responsável pela eliminação de um médico do sistema, identificando-o através do medicold. Para garantir que apenas utilizadores autorizados possam realizar essa operação, é exigida uma password, e a função só prossegue se a password fornecida for igual a 0000. Caso contrário, a função retorna false, bloqueando a eliminação do médico. Como na função anterior, há um bloco de try-catch para capturar exceções que possam ocorrer durante a operação de remoção do médico. Além destas é feita também, por exemplo, a verificação da data não ser menor que a data de hoje em diversos métodos.

6.6. As entidades presentes na camada de dados

6.6.1. Propriedades sem set

```

public int Id
{
    get { return id; }
}

```

L. Cod.11 – Propriedade do Identificador de Consulta

Na camada de dados de um sistema, a utilização de propriedades sem o método set nas entidades tem como objetivo garantir maior controlo e segurança sobre os dados. Isto significa que uma propriedade é somente de leitura e não pode

ser alterada diretamente após a sua inicialização. Este tipo de abordagem garante que os dados não sejam modificados ao longo do ciclo de vida do objeto ou que sejam apenas com o uso de métodos específicos, com um controlo adicional antes de serem alterados.

6.6.1.1. Alteração de atributos

```
public bool AtualizarCustoDoExame(int exameId, double novoCusto, MedicoLight
medicoLight)
{
    if (medicoLight == null) throw new ArgumentNullException(nameof(medicoLight));
    if (!medicoLight.PodeTomarDecisooes) throw new MedicoNaoAutorizadoException();
    if (novoCusto < 0) return false;
    try
    {
        exameRepositorio.AtualizarCustoExame(exameId, novoCusto);
        return true;
    }
    catch (ColecaoNaoIniciadaException)
    {
        throw;
    }
    catch (Exception)
    {
        throw;
    }
}
```

L. Cod. 12 – Funcao AtualizarCustoDoExame presente em GestaoExames

Esta função, responsável por atualizar o custo de um determinado exame, quando acionada é redirecionada para o repositório de exames onde identifica determinado exame e redireciona para o mesmo onde através do método produzido e não do método set na propriedade é definido ou alterado.

```
public bool AtualizarCusto(double novoCusto)
{
    if (novoCusto <= 0) return false;
    custo = novoCusto;
    return true;
}
```

L. Cod. 13 – Funcao de Atualizar Custo em Exame

6.7. Exportação de Ficheiros

```
public bool ExportarConsultas(string nomeFicheiro)
{
    try
    {
        using (Stream stream = File.Open(nomeFicheiro, FileMode.Create))
        {
            BinaryFormatter bin = new BinaryFormatter();

            bin.Serialize(stream, consultasRepositorioId);
            bin.Serialize(stream, consultas);
        }
        return true;
    }
    catch (Exception ex)
    {
        throw new Exception(ex.Message);
    }
}
```

L. Cod. 14 – Funcao de Exportação presente em ConsultasRepositorio

A função `ExportarConsultas` presente na classe `ConsultasRepositorio` tem como objetivo exportar os dados das consultas para um ficheiro binário. Esta função recebe como parâmetro o nome do ficheiro onde os dados devem ser armazenados e utiliza o `BinaryFormatter` para realizar a serialização dos objetos.

O processo começa com a criação de um fluxo de dados (`Stream`) que é aberto em modo de criação de ficheiro, ou seja, caso o ficheiro já exista, ele será substituído, impedindo o acumulo de informação a medida que o programa é corrido. A seguir, a função serializa dois objetos: `consultasRepositorioId`, identificador do repositório de consultas, e `consultas`, que representa o conjunto de consultas que será exportado. A serialização converte os objetos em um formato binário, adequado para armazenamento e posterior recuperação.

É importante destacar que a classe `Consulta`, deve ser marcada com o atributo `Serializable` para permitir a serialização. Este atributo é fundamental, pois indica que os objetos dessa classe podem ser convertidos em um formato que pode ser armazenado ou transmitido. A classe `ConsultasRepositorio` ao utilizar a serialização assegura que os dados das consultas, com todos os seus detalhes, sejam gravados de forma segura e possam ser recuperados quando necessário.

A função de exportação é envolvida por um bloco try-catch, que captura qualquer exceção ocorrida durante o processo de exportação, como problemas de acesso ao ficheiro ou falhas na serialização. Caso ocorra um erro, uma exceção é lançada com a mensagem de erro específica.

A função de exportação de ficheiro assim como as restantes presentes na camada dos dados é inacessível a partir da camada de apresentação tendo uma uma função intermediária presente na camada de gestão (Regras de Negócio):

```
public bool ExportarConsultasParaFicheiro(string nomeFicheiro)
{
    try
    {
        bool sucesso = consultaRepositorio.ExportarConsultas(nomeFicheiro);
        return sucesso;
    }
    catch (Exception ex)
    {
        throw new Exception(ex.Message);
    }
}
```

L. Cod. 15 – Funcao de Exportação presente em GestaoConsultas

6.8. Uso de Interfaces

No presente projeto fiz o uso de interfaces pois reconheço que esta é uma prática fundamental na estruturação de sistemas orientados a objetos, isto pois promove a flexibilidade, o desacoplamento e a reutilização do código.

A outra grande razão que me levou à criação e aplicação de interfaces foi o facto de estas possibilitarem um contrato que as classes que a implementam devem seguir, garantindo que estas compartilhem um conjunto comum de métodos, mas sem impor uma implementação específica. Isso permite que diferentes classes possam ser alteradas ou substituídas sem afetar o restante do sistema, desde que cumpram com o contrato definido pela interface. Por exemplo a Interface criada IRepositoryo<T> é uma espécie de

contrato para todos os repositórios que tenho no meu sistema garantindo que todos este tem os métodos pretendidos.

6.9. Passagem das Classes Light para Classes Normais

Inicialmente, a passagem das classes Light para as classes normais estava a ser feita diretamente nas regras de negócio. Porém, após uma discussão de ideias com o professor, percebi que essa abordagem estava a distanciar-me do verdadeiro propósito da camada de regras de negócio. A camada de regras de negócio tem como objetivo centralizar a lógica do sistema, e não a transformação ou manipulação direta de dados, o que estava a ser feito ao converter as classes Light em classes normais nesse contexto.

Foi então que decidi alterar a forma de trabalhar com essas conversões, transferindo esse processo para a camada de dados. A partir desse momento, tanto na camada de regras de negócio quanto na camada de apresentação (GestaoCentroSaude), as classes Light passaram a ser usadas diretamente, mantendo o foco na lógica de negócio e na interface com o utilizador. A conversão entre as classes Light e as classes normais acontece exclusivamente na camada de dados.

Essa mudança permitiu-me manter a coerência da estrutura do sistema, fazendo com que a camada de dados ficasse encarregada de tratar dessas transformações de forma centralizada. Assim, o código tornou-se mais organizado, modular e alinhado com os princípios de boas práticas de design de software.

6.10. Importância do Polimorfismo

Achei relevante a importância do polimorfismo, referido na fase anterior do trabalho, pois o facto de as funções em diferentes classes terem o mesmo nome foi um fator muito facilitador para a extensão e desenvolvimento do código, pois mantive a consistência e simplicidade na implementação de funcionalidades. Ao utilizar o mesmo nome de método, foi possível centralizar a lógica nas interfaces, reduzindo a

necessidade de duplicação de código e tornando o sistema mais flexível para futuras alterações.

6.11. Exceções

Ao longo do desenvolvimento do sistema, percebi a grande importância das exceções e do seu funcionamento em cadeia, especialmente em métodos que chamam outros métodos. Ao utilizar o bloco try-catch, garanti que as exceções não fossem perdidas e que, em caso de erro, fosse possível identificar e com isso conseguia tratar adequadamente a situação. Consegui então uma gestão de erros mais robusta, permitindo que o sistema se comportasse de forma mais previsível e que os problemas fossem tratados de maneira controlada.

6.11.1. Exceções personalizadas

Durante o desenvolvimento, criei três exceções personalizadas que foram fundamentais para o tratamento de situações específicas no sistema: `EntidadeJaExisteException`, `MedicoNaoAutorizadoException` e `ColecaoNaoIniciadaException`. Cada uma delas foi criada com base na sua recorrência ao longo do código, permitindo um tratamento mais preciso e adequado para cada tipo de erro.

A **`EntidadeJaExisteException`** é lançada quando uma tentativa de adicionar uma nova entidade (como um paciente ou médico) é feita, mas a entidade já existe no sistema. A necessidade desta exceção surgiu devido a grande recorrência de possibilidade de uso da mesma e desencadeou em evitar a duplicação de dados e garantir a integridade da base de dados.

A **`MedicoNaoAutorizadoException`** é lançada quando um médico tenta realizar uma ação para a qual não tem autorização, como, por exemplo, alterar dados sem o atributo para esse fim indicado como valor true. Esta exceção foi essencial para garantir a segurança e o cumprimento das regras específicas do negócio, prevenindo que operações erradas ou não autorizadas fossem executadas.

A **`ColecaoNaoIniciadaException`** é lançada quando uma tentativa de interação com uma coleção de objetos é feita, mas a coleção não foi inicializada

corretamente. Ao capturar esse erro, o sistema é capaz de evitar falhas de execução e garante que a aplicação continue a funcionar sem interrupções.

6.12. Implementação da Interface Comparable

Na implementação do sistema de gestão de exames, utilizei a interface Comparable na classe Exame para permitir a comparação entre os objetos dessa classe com base em um critério específico, que neste caso foi o custo do exame. A implementação do método CompareTo na classe Exame torna possível ordenar uma lista de exames utilizando o método Sort() da classe List<Exame>.

```
"public int CompareTo(Exame outroExame)
{
    if (outroExame == null) return 1;
    return this.custo.CompareTo(outroExame.custo);
}"
```

L. Cod. 16 – Método CompareTo na classe Exame

Ao implementar a interface Comparable, a classe Exame passou a ter a capacidade de comparar instâncias dessa classe entre si. No método CompareTo, o critério de comparação utilizado foi o custo do exame, permitindo que a ordenação ocorra de forma natural e direta. A vantagem de usar CompareTo em conjunto com o método Sort() é que a ordenação fica automatizada, sem a necessidade de definir manualmente uma lógica de comparação a cada vez que precisarmos ordenar a lista. O método Sort() é implementado na classe ExamesRepositorio, e ele utilizará o CompareTo para realizar a ordenação de acordo com o custo do exame.

Por fim, na classe GestaoExames, a função OrdenarExamesPorCusto da classe ExamesRepositorio é chamada para ordenar os exames antes de apresentar a informação ao utilizador. Esta classe atua como um intermediário entre a camada de apresentação e a camada de dados. A função presente na GestaoExames tem como parâmetro uma instância MedicoLight, e com isso pretendi que a ordenação fosse feita apenas na presença de um médico.

6.13. Testes Unitários

Com a abordagem dos mesmos em contexto de aula, reconheci que os testes unitários são uma parte essencial no processo de desenvolvimento de um software. Com a implementação dos mesmos é possível validar o comportamento de unidades individuais de código, como funções ou métodos, de forma isolada, percebendo erros e incongruências no código. Ao testar cada parte do código enquanto está a ser escrita, os erros podem ser identificados logo no início, evitando que se propaguem para outras áreas do sistema e reduzindo significativamente os custos com a correção de falhas. Além disso, os testes unitários tornam a manutenção do código muito mais fácil.

6.13.1. A minha Implementação dos Testes Unitários

Criei um projeto de testes unitários chamado TestesUnitarios para testar as funcionalidades implementadas nas classes GestaoConsultas, GestaoDiagnosticos e GestaoExames. Cada uma dessas classes possui métodos específicos para testar diferentes partes do código e garantir o correto funcionamento dos métodos, testando tanto o sucesso como o insucesso de certas funções.

Embora tenha implementado uma boa parte dos testes, não foi testado todo o código, pois, nesta fase do trabalho, foi reconhecido que não havia necessidade de testar todas as funcionalidades em detalhe. No entanto, reconheço a conveniência do uso de testes unitários, uma vez que ajudam a garantir que cada unidade de código funcione de forma isolada e eficiente, além de facilitar a manutenção e a evolução do sistema.

6.13.2. Como Funcionam os métodos nos Testes Unitários

```
[TestMethod]
public void AdicionarDiagnosticoInsucesso_DataInvalida()
{
    GestaoDiagnosticos gestaoDiagnositcos = new GestaoDiagnosticos();

    DiagnosticoLight diagnosticoLight = new DiagnosticoLight(1, new
    DateTime(2001, 1, 1), 1);
    MedicoLight medicoLight = new MedicoLight(1, "Dr. João", 2);
```



```

        ArgumentException e = Assert.ThrowsException<ArgumentException>(() =>
gestaoDiagnositcos.AdicionarDiagnostico(diagnosticoLight, medicoLight));
    }

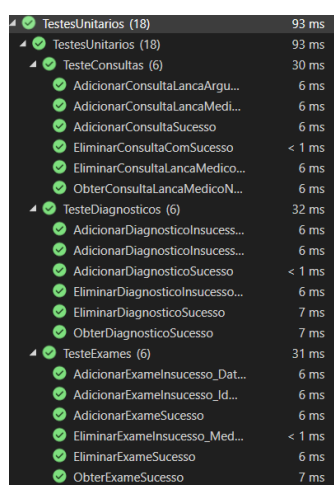
```

L. Cod. 17 – Exemplo de Teste Unitário

Para explicar o funcionamento dos testes unitários e demonstrar como foram implementados, utilizarei como exemplo o método `AdicionarDiagnosticoInsucesso_DataInvalida`, que tem como objetivo testar o comportamento do método `AdicionarDiagnostico` da classe `GestaoDiagnosticos` quando é fornecida uma data inválida para o diagnóstico.

O teste começa com a criação de instâncias dos objetos necessários, como a classe `GestaoDiagnosticos`, que é a classe que estamos a testar, e os objetos `DiagnosticoLight` e `MedicoLight`, que serão passados como parâmetros ao método que queremos testar. Neste caso, a data do `DiagnosticoLight` é configurada para uma data inválida (1 de Janeiro de 2001), com a intenção de provocar um erro, pois esta é uma data inferior à data de hoje.

A seguir, o método `AdicionarDiagnostico` é invocado, e espera-se que, devido à data inválida, seja lançada uma exceção do tipo `ArgumentException`. A linha de código `Assert.ThrowsException<ArgumentException>` é utilizada para garantir que, de facto, uma exceção desse tipo seja gerada quando o método for executado com os parâmetros errados. Como a exceção foi lançada corretamente, o teste foi considerado bem-sucedido, caso contrário, seria indicando que o comportamento esperado não foi alcançado.



Teste	Tempo
TestesUnitarios (18)	93 ms
TestesUnitarios (18)	93 ms
TesteConsultas (6)	30 ms
AdicionarConsultaLancaArgu...	6 ms
AdicionarConsultaLancaMedi...	6 ms
AdicionarConsultaSucesso	6 ms
EliminarConsultaComSucesso	< 1 ms
EliminarConsultaLancaMedico...	6 ms
ObterConsultaLancaMedicoN...	6 ms
TesteDiagnosticos (6)	32 ms
AdicionarDiagnosticoInsucess...	6 ms
AdicionarDiagnosticoInsucess...	6 ms
AdicionarDiagnosticoSucesso	< 1 ms
EliminarDiagnosticoInsucesso...	6 ms
EliminarDiagnosticoSucesso	7 ms
ObterDiagnosticoSucesso	7 ms
TesteExames (6)	31 ms
AdicionarExameInsucesso_Dat...	6 ms
AdicionarExameInsucesso_Id...	6 ms
AdicionarExameSucesso	6 ms
EliminarExameInsucesso_Med...	< 1 ms
EliminarExameSucesso	6 ms
ObterExameSucesso	7 ms

Imagem 5 - Demonstração de como aparece a execução dos Testes Unitários

6.13.3. Correção no código devido aos testes unitários

Durante a implementação dos testes unitários, percebi que a abordagem inicial, que tentava associar diretamente as consultas aos IDs dos repositórios de médicos e pacientes, não estava a funcionar corretamente. A associação estava a gerar erros nos testes, e as consultas não estavam a ser corretamente adicionadas (Problema aprofundado no tópico de Abordagens Exploradas, mas Deixadas de Lado) .

A classe `GestaoConsultas` tentava associar as consultas através das funções `AssociarRepositorioConsultasAoPaciente` e `AssociarRepositorioConsultasAoMedico`, que redirecionavam para a classe base `Pessoa`. No entanto, essa implementação não estava a garantir uma associação eficiente entre as entidades e os repositórios.

Após os testes, entendi que a lógica de associação precisava ser revista para garantir o funcionamento correto do sistema.

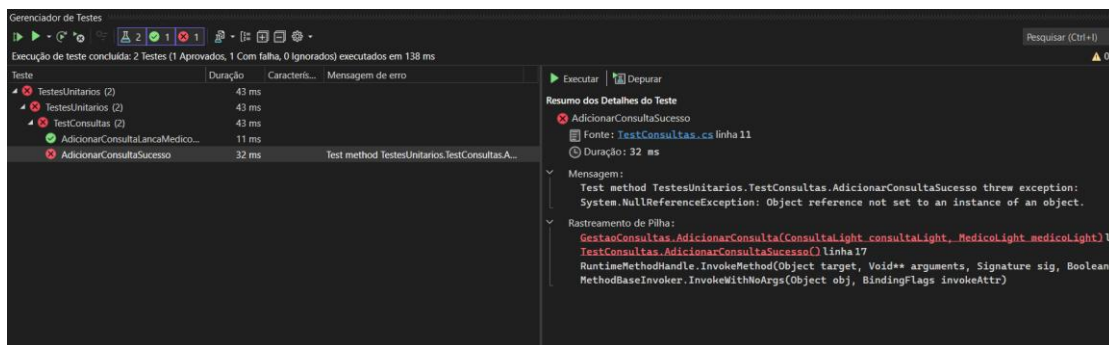


Imagem 6 – Resultado ao executar o teste unitário de `AdicionarConsultaSucesso`

7. Análise de resultados/Reflexão (1ª Fase)

7.1. Abordagens Exploradas, mas Deixadas de Lado

Durante o desenvolvimento da 2ª fase, explorei várias abordagens que, embora não tenha seguido por completo, contribuíram para a minha aprendizagem.

Um exemplo disso foi o uso de dicionários, usando chaves únicas como os identificadores (Ids). Embora essa abordagem oferecesse uma maior flexibilidade e eficiência na gestão de dados, optei por não a seguir nesta fase do projeto, pois senti que faria mais sentido o uso de listas. O Dictionary, para ser eficaz, exigiria uma estrutura adicional de chave-valor, o que adicionaria complexidade ao código. Ao invés de usar dicionários, escolhi continuar com listas, mas com um atributo adicional para representar os IDs. Isso permitiu que eu mantivesse a simplicidade e, ao mesmo tempo, associasse de forma clara os dados aos identificadores.

Outra abordagem que tentei explorar, mas deixei de lado, por motivos de não conseguir implementar, foi a associação direta da função de adicionar consulta do id do repositório em questão que as consultas estão a ser associadas aos atributos consultasId presentes nas entidades Medico e Paciente. Durante a implementação dos testes unitários a função de adicionar consulta dava um erro, além disso a própria implementação no main não adicionava corretamente as consultas. O erro dado no projeto dos Testes é abordado mais aprofundadamente no tópico presente relatório a esse fim indicado.

Como estava implementada a classe GestaoConsultas anteriormente:

```
ConsultasRepositorio consultaRepositorio;
GestaoPacientes gestaoPacientes;
GestaoMedicos gestaoMedicos;

public GestaoConsultas()
{
    consultaRepositorio = new ConsultasRepositorio();
    gestaoPacientes = new GestaoPacientes();
    gestaoMedicos = new GestaoMedicos();
}

public bool AdicionarConsulta(ConsultaLight consultaLight, MedicoLight
medicoLight)
```

```

{
    if (consultaLight == null) throw new
ArgumentNullException(nameof(consultaLight));
    if (medicoLight == null) throw new
ArgumentNullException(nameof(medicoLight));
    if (!medicoLight.PodeTomarDecisoes) throw new
MedicoNaoAutorizadoException();
    try
    {
        consultaRepositorio.AdicionarLight(consultaLight);

gestaoPacientes.AssociarRepositorioConsultasAoPaciente(consultaLight.Pacien
teId, consultaRepositorio.ConsultasRepositorioId, medicoLight);

gestaoMedicos.AssociarRepositorioConsultasAoMedico(consultaLight.MedicoId,
consultaRepositorio.ConsultasRepositorioId, medicoLight);

    }
    catch (EntidadeJaExisteException)
    {
        throw;
    }
    catch (Exception ex)
    {
        throw (ex);
    }

    return true;
}

```

L. Cod. 18 – Antiga implementação de GestaoConsultas

A intenção era que a adição da consulta ao conjunto de consultas implica-se que o identificador do conjunto de consultas fosse diretamente associado tanto ao paciente quanto ao médico. Para isso, eram chamadas duas funções específicas: `AssociarRepositorioConsultasAoPaciente` e `AssociarRepositorioConsultasAoMedico`. Estas funções estavam implementadas nas classes de repositórios de cada entidade, ou seja, a classe responsável pelos dados do paciente e a classe responsável pelos dados do médico.

Essas funções de associação, tinham como objetivo fazer com que o repositório de consultas adiciona-se os identificadores das consultas no contexto de cada paciente e médico, garantindo que a relação entre as entidades fosse devidamente estabelecida. As funções de associação, por sua vez, redirecionavam a operação para as classes de repositórios específicas, que geriam as consultas de forma organizada. O redirecionamento não era feito diretamente para as entidades de paciente ou médico, mas para a classe `Pessoa`, que era a classe base dessas entidades.

```
"public bool AssociarConsultasRepositorio(int repositorioId)
{
    consultasId = repositorioId;
    return true;
}"
```

L. Cod. 19 – Função AssociarConsultasRepositorio presente em Pessoa

7.2. Considerações sobre Decisões Tomadas

Ao longo desta fase, fiz algumas escolhas estratégicas, algumas erradas mas outras acertadas, tanto a aprendizagem retirada das escolhas erradas como o sucesso retirado das acertadas ajudaram a manter o código organizado e flexível. Penso que uma das mais importantes decisões foi utilização de Ids. Isto pois ao usar IDs em vez de instâncias diretas de objetos complexos, consegui simplificar o código e melhorar a clareza das operações, tornando o sistema mais fácil de entender e de manter. Uma outra escolha importante que fiz durante o desenvolvimento do projeto foi a utilização da arquitetura N-Tier. Esta decisão foi fundamentada, assim como abordado em contexto de aula, na sua capacidade de separar claramente as responsabilidades em diferentes camadas. Penso que a minha decisão em aplicar os conceitos da N-Tier proporcionou-me uma organização mais robusta e escalável, permitindo uma melhor manutenção e testabilidade do sistema. Decidi utilizar N-Tier e não a MVC pois acredito que a N-Tier tem um funcionamento mais intuitivo, e, além disso, estava mais familiarizado com esta arquitetura, o que facilitou a implementação e compreensão do sistema.

8. Conclusão – 2ª Fase

Nesta 2ª fase, acredito que o desenvolvimento do sistema de gestão para o Centro de Saúde evoluiu significativamente. Embora eu tenha consciência de que o código final não seja o melhor possível, acredito vivamente que consegui superar os objetivos propostos, tendo aprendido bastante ao longo do processo e melhorado as minhas capacidades de programação e design de sistemas.

Comecei por focar na implementação da estruturação do código produzido na primeira fase do projeto em camadas, seguindo o sistema de modelação N-Tier, aplicando nesta estruturação funcionalidades já criadas essenciais da aplicação (funções de gestão). A estrutura foi explorada, e as diferentes camadas do sistema foram ficando cada vez mais definidas e, a meu ver individualizadas, o que permitiu uma maior organização e modularidade no meu código. Este projeto foi imprescindível para a minha perceção da importância que têm os dados para um projeto, e consequentemente fez-me perceber a importância de um bom controlo sobre os dados. Foi também de grande interesse a elaboração de uma camada Regras de Negócio, pois inicialmente estava errado quanto a sua função, e estava a fazer nela mais do que aplicar a lógica de negócio e estava a fazer a transformação de dados. Como em qualquer projeto, a errar aprende-se, não foi diferente desta forma, consegui mudar a lógica da minha camada de regra de negócio e penso ter conseguido atingir uma solução agradável. Além disso, a adição de funcionalidades como a exportação de dados foi uma adição interessante no projeto, pois consegui perceber as diferenças mas ao mesmo tempo as semelhanças que a exportação de ficheiros tem com a programação imperativa. Penso que foi um ponto positivo do meu trabalho utilização de exceções personalizadas, pois estas permitiram-me, além de perceber a importância das exceções, fazer com que o sistema fosse mais robusto ao tratar erros de maneira específica. A ideia de criar uma camada de repositórios e usar interfaces para abstrair as operações de manipulação de dados foi uma escolha importante no meu projeto. Trabalhar com as classes Light e as classes normais também teve os seus desafios, mas consegui perceber a importância de garantir que cada camada do sistema tivesse responsabilidades bem

definidas. A separação de responsabilidades tornou o código mais limpo e fácil de entender, embora reconheça que ainda há margem para melhorar.

No geral, sinto que nesta fase do projeto consegui atingir os objetivos que me propus, embora, assim como em qualquer projeto de programação, há sempre aspetos do código que poderiam ter sido mais refinados. A experiência adquirida foi valiosa e permitiu-me aprender de forma prática muitos conceitos importantes sobre a programação orientada a objetos, a estruturação de sistemas e a aplicação de boas práticas de design. Reconheço que há sempre espaço para melhorar, mas acredito que este trabalho me permitiu crescer e ganhar uma base sólida para projetos futuros.

9. Bibliografia

Awari. "Os 4 Pilares da Programação Orientada a Objetos: Guia Completo para Iniciantes." Acesso em 28 de outubro de 2024. Disponível em:

Microsoft. (n.d.). N-tier architecture style. Azure Architecture Center. Acesso em 28 de outubro de 2024. Disponível em:

https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier?utm_source=chatgpt.com

9.1. Bibliografia de Citações

[1] - Albahari, J., & Albahari, B. (2020). *C# 9.0 in a nutshell: The definitive reference*. O'Reilly Media

[2] - Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional

9.2. Bibliografia de Imagens

Inforchannel. (2023). Linguagens de programação que mais crescem [Imagem]. Inforchannel. <https://inforchannel.com.br/2023/02/02/github-lista-as-linguagens-de-programacao-que-mais-crescem/> (acesso em: 28/10/2024).

ANEXO A

Programação Orientada a Objetos

LESI|LESIPL

Trabalho Prático

Luís G. Ferreira^{*} & Ernesto Casanova[†]

EST-IPCA
Barcelos

16 de outubro de 2024

Resumo

Este trabalho da Unidade Curricular (UC) de Programação Orientada a Objetos (POO) foca a análise de problemas reais simples e a aplicação do Paradigma Orientado a Objetos na implementação de possíveis soluções.

^{*}Email: lufer@ipca.pt; Gnh: 5

[†]Email: ecasanova@ipca.pt; Gnh: 5

1 Motivação

Pretende-se que sejam desenvolvidas soluções em C# para problemas reais de complexidade moderada. Serão identificadas classes, definidas estruturas de dados e implementados os principais processos que permitam suportar essas soluções. Pretende-se ainda contribuir para a boa redação de relatórios.

2 Objetivos

- Consolidar conceitos basilares do Paradigma Orientado a Objectos;
- Analisar problemas reais;
- Desenvolver capacidades de programação em C#;
- Potenciar a experiência no desenvolvimento de software;
- Assimilar o conteúdo da Unidade Curricular.

3 Regras do jogo

- O trabalho deverá ser feito individualmente;
- O trabalho deverá ser entregue em duas fases. Em cada fase do trabalho realizado, deverá ser submetido um ficheiro compactado, na plataforma *moodle*, cujo nome deverá conter o número do aluno e número que identifica a fase. Exemplo para a entrega da Fase 1 do aluno 1234:

– *trabalhoPOO_1234_Fase1.zip*

As datas previstas para cada cada fase e as metas a atingir em cada uma, são:

- **Fase 1 (15-11-2025)**
 - Estrutura de Classes identificadas
 - Implementação essencial das classes
 - Estruturas de dados a utilizar
 - Relatório do trabalho desenvolvido até à data
 - Cumprimento dos prazos
- **Fase 2 (19-12-2025)**
 - Implementação final das classes e serviços
 - Aplicação demonstradora dos serviços implementados
 - Relatório final do trabalho realizado
 - Cumprimento dos prazos

4 Problemas a explorar

O tema de trabalho que o aluno pretenda explorar deve ser indicado na plataforma Onedrive. O aluno pode propor o seu próprio tema ou escolher uma das seguintes sugestões:

- (i) Gerir de Atividades de Socorro: sistema que permita registar ocorrências de pedidos de ajuda e gerir equipamentos e pessoas para prestar auxílio.
keywords: Proteção Civil; Equipamentos; INEM; Enfermeiros, Medicos; Bombeiros,
- (ii) Gerir obra de construção civil: sistema que permita controlar e gerir os custos de uma determinada obra.
keywords: obras, materiais, armazéns, stocks, viaturas, serviços, mão de obra (contratada e subcontratada), orçamentos, documentos.
- (iii) Gestão de condomínios: sistema que permita fazer a gestão de condomínios de uma empresa.
keywords: condomínios, condóminos (proprietários, inquilinos), permissões, quotas, despesas, gestão de pagamentos, agendamento de reuniões, atas, documentos.
- (iv) Gestão de rendas/imóveis: sistema que permita a uma empresa/proprietário gerir os seus imóveis e as respetivas rendas mensais;
keywords: senhorios, inquilinos, imóveis (apartamento, vivenda, terreno), recibos, contratos, despesas, documentos, distritos (código e descrição nas finanças), concelhos (código e descrição nas finanças), freguesias (código e descrição nas finanças), estados.
- (v) Gestão de alojamentos turísticos: sistema que permita a gestão de alojamentos turísticos.
keywords: registos, consultas, reservas, check-in, clientes, alojamentos.
- (vi) Helpdesk: Sistema de gestão e apoio a assistências por telefone (call center).
keywords: assistência, tipo de assistência, estado da assistência, operador, cliente, produtos, problemas conhecidos, tutoriais de resolução de problema, documentos, avaliação da assistência (e.g. se o problema foi resolvido e avaliação de 1 a 10).
- (vii) Comércio eletrónico: sistema que permita a gestão de uma loja online.
keywords: produtos, categorias, garantias, stocks, clientes, campanhas, vendas, marcas.
- (viii) Gestão de jardim zoológico: sistema que permita a gestão das tarefas de um jardim zoológico.
keywords: animais, informações, assistência veterinária, alimentação, calendários, tipos de comida, limpeza de jaulas, espetáculos, bilhetes.
- (ix) Gestão de Centro de Saúde: sistema que permita gerir um Centro de Saúde.
keywords: staff, categorias, especialidades, consultas, camas, pacientes, diagnósticos, exames, custos.

5 Critérios de Avaliação

5.1 Qualidade do Relatório

- Estrutura, Clareza e Expressividade

- Capacidade de síntese e qualidade da escrita

5.2 Qualidade da solução desenvolvida

- Qualidade do código produzido: estrutura da solução, nome de ficheiros, uso de bibliotecas (DLL), Norma CLS.
- Organização e Implementação das Classes com recurso a Interfaces, Herança, classes abstractas, outras.
- Qualidade dos algoritmos aplicados, incluir testes unitários
- Estruturas de dados exploradas
- Tratamento adequado de exceções
- Tratamento adequado de logs
- Persistência de dados com recurso a ficheiros
- Programação por camadas (NTier, MVC, outras)
- Exploração de outras valências: Lambda Functions; LINQ; Windows Forms; WPF; CI-CD Pipeline (Azure DevOps)

6 Avaliação

- Fase 1 - 25% da nota final do trabalho (TP.C - Componente código)
- Fase 2 - 75% da nota final do trabalho (TP.C - Componente código)
- O trabalho será defendido individualmente - consultar metodologia e avaliação.

Bom trabalho.