Relatório do trabalho da disciplina de Programação Orientada a Objetos

Relatório do Projeto
Gestão de Condomínios

Tiago Barroso Fontes - 33222

Licenciatura em Engenharia de Sistemas Informáticos

Novembro de 2025

ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Índice

Lista de Figuras

1. Introdução

O presente capítulo introduz o trabalho desenvolvido. É apresentada a motivação e o enquadramento do projeto, os objetivos definidos pela unidade curricular, e, por fim, a estrutura do documento.

1.1. Motivação e Enquadramento

O presente projeto, desenvolvido no âmbito da Unidade Curricular de Programação Orientada a Objetos (POO), centra-se na aplicação prática do Paradigma Orientado a Objetos. O objetivo primordial consiste na "análise de problemas reais simples e a aplicação do Paradigma Orientado a Objetos na implementação de possíveis soluções".

O problema selecionado para resolução foi o tema "Gestão de condomínios: sistema que permita fazer a gestão de condomínios de uma empresa". A motivação passa por criar um núcleo de software modular em C#, que modele as entidades e relações de um sistema de gestão de condomínios, focando-se em conceitos como "condomínios", "condóminos (proprietários, inquilinos)", "permilagens", "quotas" e "despesas".

1.2. Objetivos

Os objetivos deste trabalho alinham-se com os definidos pela unidade curricular, pretendendo:

- Consolidar conceitos basilares do Paradigma Orientado a Objetos;
- Analisar um problema real de complexidade moderada;
- Desenvolver capacidades de programação em C#;
- Aplicar os pilares de POO, como Herança, Encapsulamento e Abstração;
- Assimilar o conteúdo da Unidade Curricular através de um projeto prático.

1.3. Estrutura do documento

O presente relatório encontra-se estruturado em quatro capítulos. O Capítulo 1 introduz o projeto, a sua motivação e objetivos. O Capítulo 2 detalha a análise e o desenho da solução, apresentando a arquitetura de classes, as relações de herança e agregação, a estrutura das classes essenciais e as estruturas de dados selecionadas. O Capítulo 3 descreve a implementação dos serviços e da lógica de negócio, bem como as estratégias de persistência de dados e tratamento de exceções. Por fim, o Capítulo 4 apresenta as conclusões do trabalho.

2. Análise e Desenho da solução

Este capítulo é dedicado à exposição do desenho da solução. Procede-se à análise da arquitetura, à detalhação das classes e suas relações, e à justificação das estruturas de dados selecionadas para o projeto.

2.1. Arquitetura Lógica e Relações

A solução foi arquitetada com base em duas relações fundamentais do POO: Herança e Agregação.

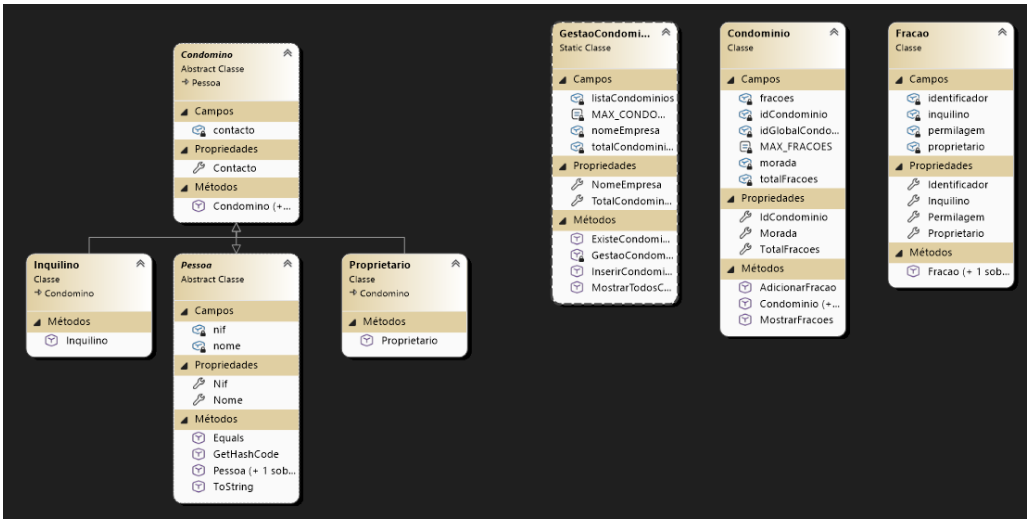


Figura 1 - Diagrama de Classes

Conforme se pode observar na Figura 1, as relações são:

- **Relação de Herança**: Foi estabelecida uma hierarquia de *Pessoa* para especializar os tipos de intervenientes. A classe *Pessoa* é a base abstrata; a classe *Condomino* herda de *Pessoa* e, por sua vez, as classes *Proprietario* e *Inquilino* herdam de Condomino.
- **Relação de Agregação**: A Figura 1 ilustra também que a classe estática *GestaoCondominios* "tem vários" *Condominio*. Por sua vez, a classe *Condominio* agrega múltiplas *Fracao*. Finalmente, a *Fracao* agrega um *Proprietario* e, opcionalmente, "tem um" *Inquilino*.

2.2. Estrutura de Classes

Apresenta-se, de seguida, a estrutura de dados e a implementação essencial de cada classe identificada, focando-se no seu estado (atributos, propriedades e construtores), conforme definido nos ficheiros de código-fonte.

2.2.1. Classe Pessoa

A classe Pessoa serve como classe base abstrata. Agrega os atributos comuns a todos os indivíduos no sistema (nome e nif).

```
public Pessoa()
{
    nome = String.Empty;
    nif = String.Empty;
}

1 referência
public Pessoa(string nome, string nif)
{
    this.nome = nome;
    this.nif = nif;
}
#endregion

#region Properties
1 referência
public string Nome
{
    get { return nome; }
    set { nome = value; }
}

1 referência
public string Nif
{
    get { return nif; }
    set { nif = value; }
}
#endregion
```

Figura 2 - Implementação da Classe *Pessoa*

2.2.2. Classe Condomino

Esta classe abstrata representa um membro genérico do condomínio. Herda de *Pessoa* e adiciona a informação de *contacto*. Os seus construtores utilizam a palavra-chave *base* para passar a informação relevante (nome, nif) ao construtor da classe-pai (*Pessoa*).

```
public abstract class Condomino : Pessoa
{
    #region Attributes
    string contacto;
    #endregion

    #region Methods

    #region Constructors

    0 referências
    public Condomino() : base()
    {
        this.contacto = String.Empty;
    }

    2 referências
    public Condomino(string nome, string nif, string contacto) : base(nome, nif)
    {
        this.contacto = contacto;
    }
    #endregion

    #region Properties
    0 referências
    public string Contacto
    {
        get { return contacto; }
        set { contacto = value; }
    }
    #endregion

    #endregion
}
```

Figura 3 - Implementação da Classe *Condomino*

2.2.3. Classes Proprietario e Inquilino

Estas são as classes concretas que herdam de *Condomino* e representam os papéis de "proprietários" e "inquilinos".

```
public class Proprietario : Condomino
{
    #region Methods
    #region Constructors

    2 referências
    public Proprietario(string nome, string nif, string contacto) : base(nome, nif, contacto)
    {
    }
    #endregion
    #endregion
}
```

Figura 4 - Implementação da Classe *Proprietario*

```
public class Inquilino : Condomino
{
    #region Methods
    #region Constructors

    1 referência
    public Inquilino(string nome, string nif, string contacto) : base(nome, nif, contacto)
    {
    }
    #endregion
    #endregion
}
```

Figura 5 - Implementação da Classe *Inquilino*

2.2.4. Classe Fracao

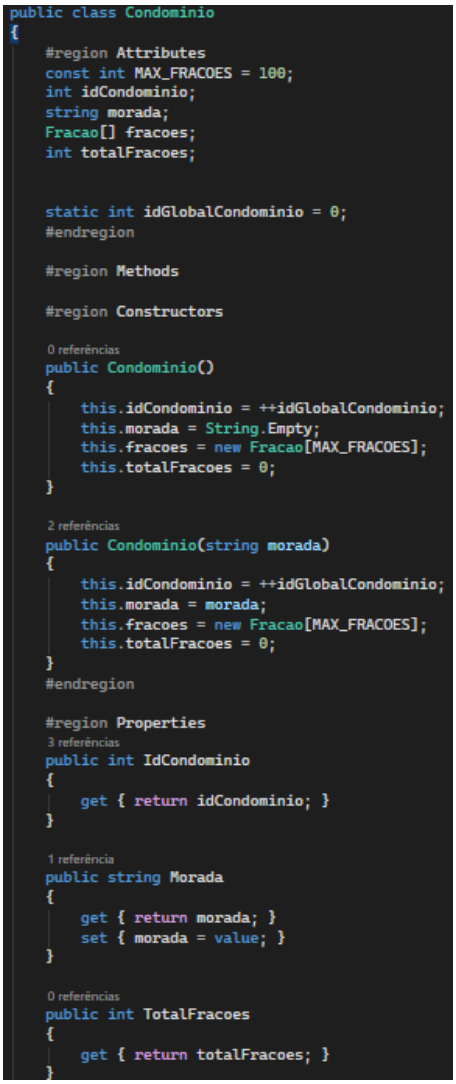
A classe Fracao modela uma unidade individual. Agrega o Proprietario e o Inquilino.

```
public class Fracao
{
    #region Attributes
    string identificador;
    double permilagem;
    Proprietario proprietario;
    Inquilino inquilino;
    #endregion

    #region Methods

    #region Constructors
    0 referências
    public Fracao()
    {
        identificador = String.Empty;
        permilagem = 0;
        proprietario = null;
        inquilino = null;
    }

    3 referências
    public Fracao(string id, double permilagem, Proprietario p)
    {
        this.identificador = id;
        this.permilagem = permilagem;
        this.proprietario = p;
        this.inquilino = null;
    }
    #endregion

    #region Properties
    1 referência
    public string Identificador
    {
        get { return identificador; }
        set { identificador = value; }
    }

    0 referências
    public double Permilagem
    {
        get { return permilagem; }
        set { permilagem = value; }
    }

    1 referência
    public Proprietario Proprietario
    {
        get { return proprietario; }
        set { proprietario = value; }
    }

    1 referência
    public Inquilino Inquilino
    {
        get { return inquilino; }
        set { inquilino = value; }
    }
    #endregion

    #endregion
}
```

Figura 6 - Implementação da Classe *Fracao*

2.2.5. Classe Condominio

Esta é a classe central, que representa os "condomínios". Agrega todas as Fracoes e gere os seus dados, como a morada. Utiliza uma variável estática idGlobalCondominio para garantir que cada novo condomínio tenha um identificador único.



```
public class Condominio
{
    #region Attributes
    const int MAX_FRACOES = 100;
    int idCondominio;
    string morada;
    Fracao[] fracoes;
    int totalFracoes;


    static int idGlobalCondominio = 0;
    #endregion

    #region Methods

    #region Constructors

    0 referências
    public Condominio()
    {
        this.idCondominio = ++idGlobalCondominio;
        this.morada = String.Empty;
        this.fracoes = new Fracao[MAX_FRACOES];
        this.totalFracoes = 0;
    }

    2 referências
    public Condominio(string morada)
    {
        this.idCondominio = ++idGlobalCondominio;
        this.morada = morada;
        this.fracoes = new Fracao[MAX_FRACOES];
        this.totalFracoes = 0;
    }
    #endregion

    #region Properties
    3 referências
    public int IdCondominio
    {
        get { return idCondominio; }
    }

    1 referência
    public string Morada
    {
        get { return morada; }
        set { morada = value; }
    }

    0 referências
    public int TotalFracoes
    {
        get { return totalFracoes; }
    }
```

Figura 7 - Implementação da Classe *Condominio*

2.2.6. Classe GestaoCondominios

Para centralizar a gestão de todos os condomínios da empresa, optou-se pela criação de uma classe estática, GestaoCondominios. Esta classe é responsável por conter a lista de todos os condomínios e fornecer métodos para os manipular. A sua inicialização é efetuada num construtor estático, garantindo que os dados da empresa são preparados antes da primeira utilização.

```
public static class GestaoCondominios
{
    #region Attributes
    const int MAX_CONDOMINIOS = 50;
    static string nomeEmpresa;
    static int totalCondominios;
    static Condominio[] listaCondominios;
    #endregion

    #region Methods

    #region Constructors

    0 referências
    static GestaoCondominios()
    {
        nomeEmpresa = "Empresa de Gestão de Condomínios, SA";
        totalCondominios = 0;
        listaCondominios = new Condominio[MAX_CONDOMINIOS];
    }
    #endregion

    #region Properties
    1 referência
    public static string NomeEmpresa
    {
        get { return nomeEmpresa; }
        set { nomeEmpresa = value; }
    }

    1 referência
    public static int TotalCondominios
    {
        get { return totalCondominios; }
    }
    #endregion
```

Figura 8 - Implementação da Classe *GestaoCondominios*

2.3. Estruturas de dados

Para a gestão das coleções de objetos (as relações 1-para-Muitos), foi selecionada a estrutura de dados de arrays (vetores) de tamanho fixo, complementada por um controlo manual do número de elementos.

Esta abordagem foi implementada nas classes Condominio e GestaoCondominios. A gestão da capacidade máxima e do total de elementos inseridos é controlada internamente nessas classes.

- *GestaoCondominios* utiliza a *estrutura Condominio[] listaCondominios* (com *MAX_CONDOMINIOS* = 50) e um contador int *totalCondominios*.
- *Condominio* utiliza a estrutura *Fracao[] fracoes* (com *MAX_FRACOES* = 100) e um contador *int totalFracoes*.

Conclusão

Conclui-se que o desenho da solução, detalhado no Capítulo 2, se encontra completo. A identificação e estruturação das classes, aplicando o pilar da Herança (*Pessoa* > *Condomino* > *Proprietario*/*Inquilino*), demonstrou ser uma abordagem eficaz para modelar o problema da gestão de condomínios.

A abordagem modular permitiu definir uma arquitetura de classes organizada e de fácil manutenção. Optou-se pela utilização de arrays de tamanho fixo para a gestão de coleções, seguindo os exemplos de código disponibilizados nas aulas.

Esta estrutura constitui uma base sólida para a implementação dos serviços, onde serão desenvolvidos os métodos para a lógica de negócio e a persistência de dados.