Programação Orientada a Objetos

Relatório de Trabalho Prático

Arthur Fellipe Cerqueira Gomes - 24200

Júlia Dória Rodrigues - 24204

Engenharia de Sistemas Informáticos

Novembro de 2023

Afirmo por minha honra que não recebi qualquer apoio não autorizado na realização deste trabalho prático. Afirmo igualmente que não copiei qualquer material de livro, artigo, documento web ou de qualquer outra fonte exceto onde a origem estiver expressamente citada.

Arthur Fellipe Cerqueira Gomes - 24200

Júlia Dória Rodrigues - 24204

Índice

Lista de Figuras

Introdução

Este trabalho aborda o desenvolvimento de um sistema em C# dedicado à gestão de reservas em residências estudantis. A fase inicial do projeto foca na identificação e implementação das classes essenciais, além da definição das estruturas de dados fundamentais.

O sistema visa atender às necessidades de estudantes em busca de acomodação, proporcionando uma experiência online que simplifica a pesquisa de quartos disponíveis. A plataforma oferece o cadastro de usuários, sendo eles clientes, proprietários, gestores ou funcionários. O projeto de Gestão de Reservas poderá ser capaz ainda de listar quartos disponíveis, demonstrando ainda as características de cada uma das acomodações e cada utilizador terá funções especificas dentro do sistema.

O processo de reserva de quartos/camas para estudantes permitirá que eles solicitem, cancelem e modifiquem suas reservas de forma direta e eficiente. A essência do sistema reside na centralização das opções de alojamento em uma plataforma online, visando facilitar a busca por moradias estudantis na cidade.

A documentação gerada não só descreve as classes e estruturas de dados, mas também explica a lógica subjacente a cada componente, fornecendo a base necessária para o desenvolvimento completo subsequente do sistema de gestão de reservas em residências estudantis em C#.

Estrutura do Código

O código fonte da aplicação contém os seguintes ficheiros:

- Program.cs
- Utilizador.cs
- Morador.cs
- Funcionario.cs
- Gestor.cs
- Quarto.cs
- Reserva.cs
- ControladorUtilizador.cs
- ControladorMorador.cs
- ControladorQuarto.cs
- MenuResidencia.cs

As classes Funcionário e Gestor, subclasses de Utilizador, ainda não foram implementadas em sua integridade, tendo lógica similar à subclasse Morador

Classes e Subclasses

1. Program.cs

A classe Program contém o método Main, que serve como ponto de entrada para a execução do programa. O código dentro do método realiza as seguintes operações:

Inicia a criação dos ficheiros JSON para utilizadores e moradores, chamando os métodos CriarFicheiroJson("utilizador.json") e CriarFicheiroJson("morador.json"). Esses métodos são responsáveis por garantir a existência dos ficheiros JSON ou criar ficheiros vazios, se necessário.

Em seguida, chama o método CriarListaDeQuartos() para gerar uma lista de quartos com informações aleatórias. Esta lista é armazenada na variável Quarto.listaDeQuartos.

Finalmente, chama o método estático ExibirMenu() para exibir e operar o menu principal da aplicação. Este método contém um loop interativo que permite que o utilizador escolha diversas opções para interação com o sistema, como registar utilizadores, listar moradores, criar reservas, entre outras. O loop continua até que o utilizador escolha a opção de sair.

2. Utilizador.cs

O código define uma classe pública chamada Utilizador, que representa todo e qualquer pessoa que faça registo na aplicação.

Dentro da classe, há uma lista pública estática chamada listaDeUtilizadores que armazena todos os utilizadores criados. As propriedades da classe, todas definidas como protegidas, incluem informações típicas de um utilizador, como ID, nome, email, etc. A propriedade ultimoId é auxiliar e usada para manter o último ID de utilizador.

O código utiliza o atributo [JsonProperty("NomePropriedade")] para mapear propriedades C# para chaves específicas durante a serialização ou desserialização de JSON.

```
public static List<Utilizador> listaDeUtilizadores = new List<Utilizador>(); // Variável que guarda a lista de utilizadores

[JsonProperty("UtiId")]
protected int UtiId;  // Propriedade que guarda o id do utilizador
[JsonProperty("NomeUti")]
protected string NomeUti; // Propriedade que guarda o nome do utilizador
[JsonProperty("Email")]
protected string Email; // Propriedade que guarda o email do utilizador
[JsonProperty("Password")]
protected string Password; // Propriedade que guarda a password do utilizador
[JsonProperty("DataNascimento")]
protected DateTime DataNascimento; // Propriedade que guarda a data de nascimento do utilizador
[JsonProperty("Morada")]
protected string Morada; // Propriedade que guarda a morada do utilizador
[JsonProperty("CodigoPostal")]
protected string CodigoPostal; // Propriedade que guarda o código postal do utilizador
[JsonProperty("Localidade")]
protected string Localidade; // Propriedade que guarda a localidade do utilizador
[JsonProperty("ContactoTelefone")]
protected string ContactoTelefone; // Propriedade que guarda o contacto telefónico do utilizador
[JsonProperty("DocIdentificacao")]
protected string DocIdentificacao; // Propriedade que guarda o documento de identificação do utilizador
[JsonProperty("TipoDocIdentificacao")]
protected string TipoDocIdentificacao; // Propriedade que guarda o tipo de documento de identificação do utilizador
[JsonProperty("IBAN")]
protected string IBAN; // Propriedade que guarda o IBAN do utilizador
[JsonProperty("TipoUtilizador")]
protected string TipoUtilizador; // Propriedade que guarda o tipo de utilizador
[JsonProperty("IsAtivo")]
protected bool IsAtivo; // Propriedade que indica se o utilizador está ativo ou não
[JsonProperty("DataRegisto")]
protected DateTime DataRegisto; // Propriedade que guarda a data de registo do utilizador

protected static int ultimoId = 0; // Variável que guarda o último id de utilizador
```

Figura 1 - Classe Utilizador

O construtor da classe inicializa um objeto Utilizador com as respetivas propriedades e define que alguns valores já serão inicializados com valores padrão. Nomeadamente, estabelece que a data de registo padrão é a data atual e que o utilizador inicia com estado inativo.

```
public Utilizador(int utiId,
                  string nomeUti,
                  string email,
                  string password,
                  DateTime dataNascimento,
                  string morada,
                  string codigoPostal,
                  string localidade,
                  string contactoTelefone,
                  string docIdentificacao,
                  string tipoDocIdentificacao,
                  string iban,
                  string tipoUtilizador,
                  bool isAtivo = false, // Por defeito, o utilizador não está ativo
                  DateTime dataRegisto = default(DateTime)) // Por defeito, a data de registo é a data atual

{
    UtiId = utiId;
    NomeUti = nomeUti;
    Email = email;
    Password = password;
    DataNascimento = dataNascimento;
    Morada = morada;
    CodigoPostal = codigoPostal;
    Localidade = localidade;
    ContactoTelefone = contactoTelefone;
    DocIdentificacao = docIdentificacao;
    TipoDocIdentificacao = tipoDocIdentificacao;
    IBAN = iban;
    TipoUtilizador = tipoUtilizador;
    IsAtivo = isAtivo;
    DataRegisto = (dataRegisto == default(DateTime)) ? DateTime.Now : dataRegisto; // Se a data de registo não for definida, assume a data atual
}
```

Figura 2 - Construtor Utilizador

Há também métodos Get e Set para cada propriedade, permitindo acesso e modificação desses atributos.

```
#region Getters e Setters
3 references
public int GetUtiId()
{
    return UtiId;
}

0 references
public void SetUtiId(int utiId)
{
    UtiId = utiId;
}
```

Figura 3 - Get/Set Utilizador

3. Morador.cs

O código define uma subclasse chamada Morador, que herda da classe base Utilizador, e representa um morador de uma residência. A propriedade *listaDeMoradores* é uma lista estática que armazena todos os moradores criados. A propriedade *IsAdimplente*, específica da subclasse, indica se o morador está adimplente ou não.

```
public class Morador : Utilizador
{
    public static List<Morador> listaDeMoradores = new List<Morador>(); // Variável que guarda a lista de moradores

    [JsonProperty("isAdimplente")]
    protected bool IsAdimplente; // Propriedade que indica se o morador está adimplente ou não
```

Figura 4 - SubClasse Morador

O construtor da classe Morador chama o construtor da classe base Utilizador, passando os parâmetros necessários e inicializando a propriedade específica *IsAdimplente* com valor verdadeiro por defeito, assim como *tipoUtilizador* passa a receber a string padrão "Morador".

```
public Morador(
int utiId,
string nomeUti,
string email,
string password,
DateTime dataNascimento,
string morada,
string codigoPostal,
string localidade,
string contactoTelefone,
string docIdentificacao,
string tipoDocIdentificacao,
string iban,
bool isAtivo,
DateTime dataRegisto,
bool isAdimplente = true) // O morador é adimplente por defeito
: base(utiId, nomeUti, email, password, dataNascimento, morada, codigoPostal, localidade, contactoTelefone, docIdentificacao, tipoDocIdentificacao, iban, "Morador", isAtivo, dataRegisto)
{
    IsAdimplente = isAdimplente;
}
#endregion
```

Figura 5 - Construtor Morador

Há métodos Get e Set para obter e modificar a propriedade *IsAdimplente*.

```
#region Getters e Setters
1 reference
public bool GetIsAdimplente()
{
    return IsAdimplente;
}
0 references
public void SetAdimplente(bool isAdimplente)
{
    IsAdimplente = isAdimplente;
}
#endregion
```

Figura 6 - Get/Set Morador

4. Quarto.cs

O código define uma classe chamada Quarto, que representa um quarto em uma residência. A propriedade listaDeQuartos é uma lista estática que armazena todos os quartos criados.

A classe possui propriedades protegidas para armazenar informações sobre o quarto, como QuartoId (indicando o ID do quarto), TipoQuarto (indicando o tipo de quarto), Andar (indicando o andar onde o quarto está localizado), Capacidade (indicando a capacidade de pessoas do quarto), PrecoRenda (indicando o preço da renda do quarto) e Disponibilidade (indicando se o quarto está disponível).

O código utiliza o atributo [JsonProperty("NomePropriedade")] para mapear propriedades C# para chaves específicas durante a serialização ou desserialização de JSON.

```
public class Quarto
{
    public static List<Quarto> listaDeQuartos = new List<Quarto>(); // Variável que guarda a lista de quartos da residência

    [JsonProperty("quartoId")]
    protected int QuartoId; // Propriedade que indica o id do quarto
    [JsonProperty("tipoQuarto")]
    protected string TipoQuarto; // Propriedade que indica o tipo de quarto (individual, duplo, triplo, etc.)
    [JsonProperty("andar")]
    protected int Andar; // Propriedade que indica o andar onde se localiza o quarto
    [JsonProperty("capacidade")]
    protected int Capacidade; // Propriedade que indica a capacidade de pessoas do quarto
    [JsonProperty("precoRenda")]
    protected float PrecoRenda; // Propriedade que indica o preço da renda do quarto
    [JsonProperty("disponibilidade")]
    protected bool Disponibilidade; // Propriedade que indica se o quarto está disponível ou não
```

Figura 7 - Classe Quarto

O construtor da classe Quarto é responsável por inicializar essas propriedades com os valores passados como parâmetros.

```
public Quarto(int quartoId, string tipoQuarto, int andar, int capacidade, float precoRenda, bool disponibilidade)
{
    QuartoId = quartoId;
    TipoQuarto = tipoQuarto;
    Andar = andar;
    Capacidade = capacidade;
    PrecoRenda = precoRenda;
    Disponibilidade = disponibilidade;
}
```

Figura 8 - Construtor Quarto

Métodos Get e Set são fornecidos para aceder e modificar as propriedades do quarto.

```
#region Getters e Setters
1 reference
public int GetQuartoId()
{
    return QuartoId;
}
0 references
private void SetQuartoId(int quartoId)
{
    QuartoId = quartoId;
}
```

Figura 9 - Get/Set Quarto

5. Reserva.cs

O código apresenta uma classe chamada Reserva, que representa uma reserva de um quarto em uma residência. A classe possui propriedades públicas para armazenar informações sobre a reserva, como ReservaId (indicando o ID da reserva), QuartoId (indicando o ID do quarto reservado), UtiId (indicando o ID do utilizador que fez a reserva), DataEntrada (indicando a data de entrada na reserva), DataSaida (indicando a data de saída da reserva), PrecoCaucao (indicando o preço da caução da reserva), IsAtivo (indicando se a reserva está ativa ou não) e DataReserva (indicando a data da reserva).

O código utiliza o atributo [JsonProperty("NomePropriedade")] para mapear propriedades C# para chaves específicas durante a serialização ou desserialização de JSON.

```
public class Reserva
{
    [JsonProperty("reservaId")]
    1 reference
    public int ReservaId { get; set; } // Propriedade que indica o id da reserva
    [JsonProperty("quartoId")]
    1 reference
    public int QuartoId { get; set; } // Propriedade que indica o id do quarto reservado
    [JsonProperty("utiId")]
    1 reference
    public int UtiId { get; set; } // Propriedade que indica o id do utilizador que fez a reserva
    [JsonProperty("dataEntrada")]
    1 reference
    public DateTime DataEntrada { get; set; } // Propriedade que indica a data de entrada na reserva
    [JsonProperty("dataSaida")]
    1 reference
    public DateTime DataSaida { get; set; } // Propriedade que indica a data de saída da reserva
    [JsonProperty("precoCaucao")]
    1 reference
    public float PrecoCaucao { get; set; } // Propriedade que indica o preço da caução da reserva
    [JsonProperty("isAtivo")]
    1 reference
    public bool IsAtivo { get; set; } // Propriedade que indica se a reserva está ativa ou não
    [JsonProperty("dataReserva")]
    1 reference
    public DateTime DataReserva { get; set; } // Propriedade que indica a data da reserva
```

Figura 10 - Classe Reserva

Os métodos de obtenção e modificação das propriedades são criados de forma automática, com a sintaxe { get; set; }.

O construtor da classe Reserva é responsável por inicializar essas propriedades com os valores passados como parâmetros. Ele também define um valor padrão para DataReserva como a data atual, caso não seja especificada.

```
#region Construtor
0 references
public Reserva(
    int reservaId,
    int quartoId,
    int utiId,
    DateTime dataEntrada,
    DateTime dataSaida,
    float precoCaucao,
    bool isAtivo,
    DateTime dataReserva = default(DateTime)) // A data da reserva é a data atual por defeito
{
    ReservaId = reservaId;
    QuartoId = quartoId;
    UtiId = utiId;
    DataEntrada = dataEntrada;
    DataSaida = dataSaida;
    PrecoCaucao = precoCaucao;
    IsAtivo = isAtivo;
    DataReserva = (dataReserva == default(DateTime)) ? DateTime.Now : dataReserva; // Se a data da reserva não for especificada, é a data atual
}
#endregion
```

Figura 11 - Construtor Reserva

6. ControladorUtilizador.cs

A classe ControladorUtilizador atua como um controlador para operações relacionadas aos utilizadores.

Método CriarUtilizador(): Este método estático permite criar um novo utilizador. Ele solicita informações do utilizador através da consola, como nome, email, password, data de nascimento, morada, etc. Depois de obter essas informações, um novo objeto Utilizador é criado com base nos dados fornecidos e é chamada a função para adicioná-lo à lista de utilizadores.

```
public static Utilizador CriarUtilizador()
{
    Console.WriteLine("Por favor, forneça as informações do utilizador:");

    int ultimoId = Utilizador.GetUltimoId(); // Obter o último id de utilizador
    int utiId = ++ultimoId; // Incrementar a variável de classe com o último id do utilizador e atribuí-lo para o objeto
    Utilizador.SetUltimoId(utiId); // Guardar o último id de utilizador

    Console.Write("Nome: ");
    string nomeUti = Console.ReadLine();

    Console.Write("Email: ");
    string email = Console.ReadLine();

    Console.Write("Password: ");
    string password = Console.ReadLine();

    Console.Write("Data de Nascimento (DD-MM-YYYY): ");
    DateTime dataNascimento;
    if (DateTime.TryParseExact(Console.ReadLine(), "dd-MM-yyyy", CultureInfo.InvariantCulture, DateTimeStyles.None, out dataNascimento)) // Verificar se a data de nascimento está no formato correto
    {
        Console.WriteLine();
    }
    else
    {
        Console.WriteLine("Formato de data inválido.");
    }
```

Figura 12 - Classe ControloadorUtilizador

```
    Console.Write("Morada: ");
    string morada = Console.ReadLine();

    Console.Write("Código Postal: ");
    string codigoPostal = Console.ReadLine();

    Console.Write("Localidade: ");
    string localidade = Console.ReadLine();

    Console.Write("Contacto Telefone: ");
    string contactoTelefone = Console.ReadLine();

    Console.Write("Documento de Identificação: ");
    string docIdentificacao = Console.ReadLine();

    Console.Write("Tipo de Documento de Identificação: ");
    string tipoDocIdentificacao = Console.ReadLine();

    Console.Write("IBAN: ");
    string iban = Console.ReadLine();

    Console.Write("Tipo de Utilizador: ");
    string tipoUtilizador = Console.ReadLine();

    // Cria um novo objeto Utilizador
    Utilizador utilizador = new Utilizador(
        utiId, nomeUti, email, password, dataNascimento, morada, codigoPostal, localidade, contactoTelefone, docIdentificacao, tipoDocIdentificacao, iban, tipoUtilizador);

    AdicionarUtilizador(utilizador); // Adicionar utilizador à lista de utilizadores
    return utilizador;
}
```

Figura 13 - Classe ControladorUtilizador 2

Método AdicionarUtilizador(Utilizador novoUtilizador): Este método adiciona um utilizador à lista de utilizadores, mas antes de adicionar, verifica se um utilizador com o mesmo documento de identificação já existe na lista. Se o utilizador não existir, é adicionado à lista e a lista é salva em um arquivo JSON chamado "utilizador.json". Além disso, dependendo do tipo de utilizador, um objeto correspondente (como Morador) é criado.

```
public static void AdicionarUtilizador(Utilizador novoUtilizador)
{
    // Verificar se o utilizador já existe na lista
    if (Utilizador.listaDeUtilizadores.Exists(Utilizador =>    //Função lambda
        novoUtilizador.GetDocIdentificacao() == Utilizador.GetDocIdentificacao() &&
        novoUtilizador.GetTipoDocIdentificacao() == Utilizador.GetTipoDocIdentificacao()))
    {
        Console.WriteLine("O utilizador já existe na lista.");
    }
    else
    {
        Utilizador.listaDeUtilizadores.Add(novoUtilizador);
        SalvarListaFicheiro("utilizador.json"); // Salvar lista de utilizadores no ficheiro
        Console.WriteLine("Utilizador adicionado com sucesso.");

        // Verificar o tipo de utilizador e criar o objeto correspondente
        if (novoUtilizador.GetTipoUtilizador() == "Morador" || novoUtilizador.GetTipoUtilizador() == "morador")
        {
            ControladorMorador.CriarMorador(novoUtilizador);
        }
        /* A SER IMPLEMENTADO
        else if (novoUtilizador.GetTipoUtilizador() == "Funcionário")
        {
            Funcionario.CriarFuncionario(novoUtilizador);
        }
        else if (novoUtilizador.GetTipoUtilizador() == "Gestor")
        {
            Gestor.CriarGestor(novoUtilizador);
        }*/
    }

}
```

Figura 14 - Método AdicionarUtilizador

Método ImprimirListaDeUtilizadores(): Este método imprime a lista de utilizadores na consola. Ele carrega a lista de utilizadores a partir do arquivo JSON "utilizador.json" e, em seguida, exibe as informações de cada utilizador na consola.

Figura 15 - Método ImprimirListaDeUtilizador

Métodos JSON (CriarFicheiroJson, SalvarListaFicheiro, CarregarListaDeUtilizadores): Esses métodos são responsáveis por operações relacionadas à manipulação de ficheiros JSON. Eles lidam com a criação de um ficheiro JSON vazio, salvar a lista de utilizadores no ficheiro JSON e carregar a lista de utilizadores do ficheiro JSON, respetivamente.

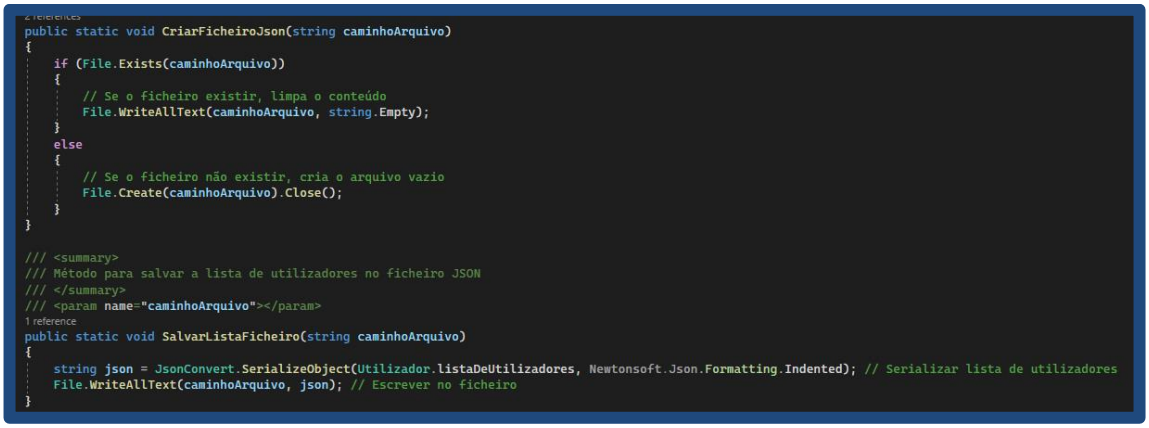

Figura 16 - Serialização JSON

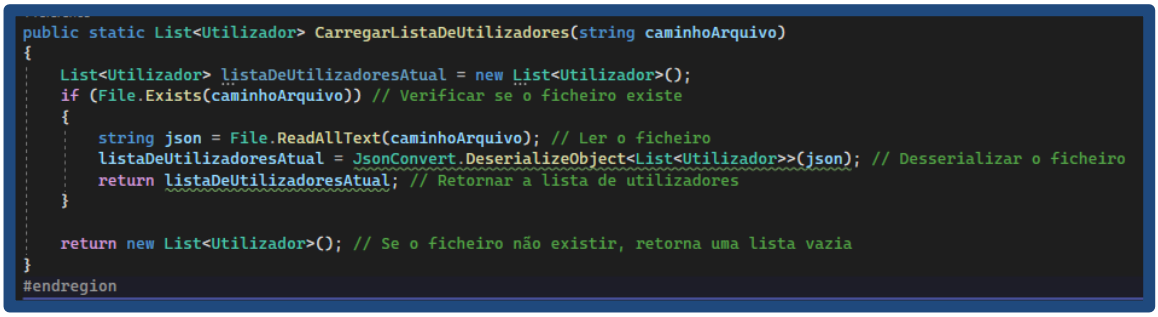

Figura 17 - Serialização JSON 2

7. ControladorMorador.cs

A classe ControladorMorador atua como um controlador para operações relacionadas aos moradores.

Método CriarMorador(Utilizador utilizador): Este método estático recebe um objeto Utilizador como parâmetro e utiliza suas informações para criar um novo objeto Morador. O morador é então adicionado à lista de moradores por meio do método AdicionarMorador, e o próprio morador é retornado.

```
public static Morador CriarMorador(Utilizador utilizador)
{
    Morador morador = new Morador(
                utilizador.GetUtiId(),
                utilizador.GetNomeUti(),
                utilizador.GetEmail(),
                utilizador.GetPassword(),
                utilizador.GetDataNascimento(),
                utilizador.GetMorada(),
                utilizador.GetCodigoPostal(),
                utilizador.GetLocalidade(),
                utilizador.GetContactoTelefone(),
                utilizador.GetDocIdentificacao(),
                utilizador.GetTipoDocIdentificacao(),
                utilizador.GetIBAN(),
                utilizador.GetIsAtivo(),
                utilizador.GetDataRegisto());
    AdicionarMorador(morador); // Adicionar morador à lista de moradores
    return morador;

}
```

Figura 18 - Classe Controlador Morador

Método AdicionarMorador(Morador novoMorador): Este método adiciona um morador à lista de moradores, verificando antes se um morador com o mesmo documento de identificação já existe na lista. Se o morador não existir, ele é adicionado à lista e a lista é salva em um arquivo JSON chamado "morador.json".

```
public static void AdicionarMorador(Morador novoMorador)
{
    // Verificar se o morador já existe na lista através do documento de identificação
    if (Morador.listaDeMoradores.Exists(morador => //Função lambda
            novoMorador.GetDocIdentificacao() == morador.GetDocIdentificacao() &&
            novoMorador.GetTipoDocIdentificacao() == morador.GetTipoDocIdentificacao()))
    {
        Console.WriteLine("O morador já existe na lista.");
    }
    else
    {
        Morador.listaDeMoradores.Add(novoMorador); // Adicionar morador à lista de moradores
        SalvarListaFicheiro("morador.json"); // Salvar lista de moradores no ficheiro
    }
```

Figura 19 - Método AdicionarMorador

Método ImprimirListaDeMoradores(): Este método imprime a lista de moradores na consola. Ele carrega a lista de moradores a partir do arquivo JSON "morador.json" e, em seguida, exibe as informações de cada morador na consola.

Figura 20 - Método ImprimirListaDeMoradores

Métodos JSON (SalvarListaFicheiro, CarregarListaDeMoradores): Esses métodos são responsáveis por operações relacionadas à manipulação de ficheiros JSON. Eles lidam com a serialização e desserialização da lista de moradores, permitindo salvar e carregar informações de moradores de e para um ficheiro JSON.

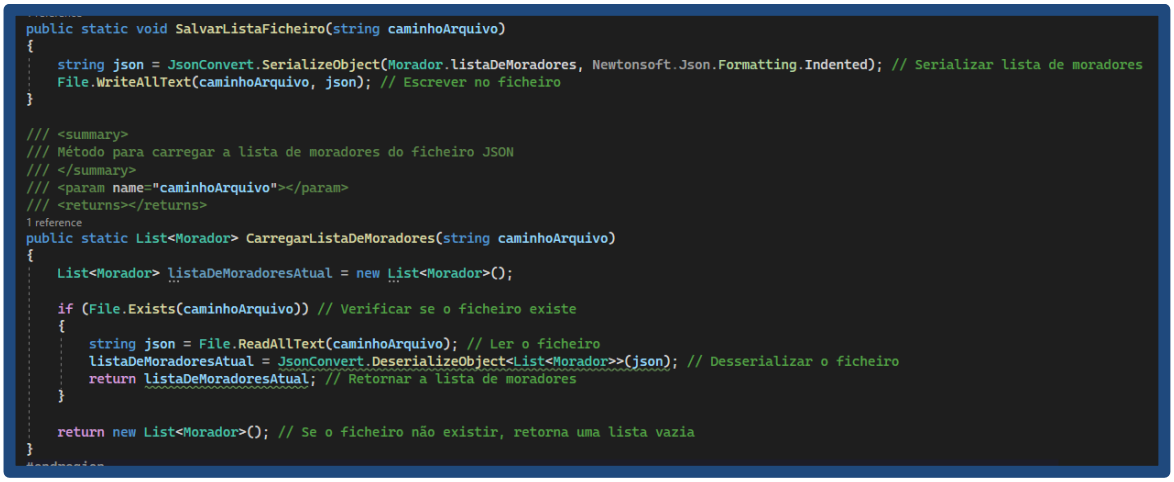

Figura 21 - Serialização JSON 3

8. ControladorQuarto.cs

A classe ControladorQuarto é um controlador que gerencia operações relacionadas aos quartos de uma residência.

Método CriarListaDeQuartos(): Este método estático é responsável por criar e retornar uma lista de quartos. Ele itera pelos andares (de 0 a 2) e pelos números dos quartos (de 1 a 19), atribuindo informações aleatórias para cada quarto, como tipo de quarto, capacidade, preço

base, preço de renda e disponibilidade, produzidas por métodos auxiliares. Os quartos são criados e adicionados à lista listaDeQuartos. A lista completa é, então, retornada.

```
public static List<Quarto> CriarListaDeQuartos()
{

    // Itera pelos andares (0 a 2) e pelos números dos quartos (1 a 19)
    for (int i = 0; i <= 2; i++)
    {
        for (int j = 1; j < 20; j++)
        {
            // Obtém informações aleatórias para criar um quarto
            string tipoQuarto = ObterTipoQuartoAleatorio();
            int capacidade = ObterCapacidadePorTipoQuarto(tipoQuarto);
            float precoBase = ObterPrecoBasePorTipoQuarto(tipoQuarto);
            float precoRenda = CalcularPrecoRenda(i, precoBase);
            bool disponibilidadeAleatoria = ObterDisponibilidadeAleatoria();

            // Cria um objeto Quarto com as informações obtidas
            Quarto quarto = new Quarto(
                j + i * 100, // ID único do quarto
                tipoQuarto,
                i, // Andar
                capacidade,
                precoRenda,
                disponibilidadeAleatoria
            );

            Quarto.listaDeQuartos.Add(quarto); // Adiciona o quarto à lista de quartos
        }
    }

    return Quarto.listaDeQuartos; // Retorna a lista de quartos criada
}
```

Figura 22 - Classe ControladorQuarto

Método ImprimirListaDeQuartos(List<Quarto> listaDeQuartos): Este método imprime na consola as informações detalhadas de cada quarto presente na lista fornecida como argumento. As informações incluem o ID do quarto, tipo, andar, capacidade, preço da renda e disponibilidade.

```
public static void ImprimirListaDeQuartos(List<Quarto> listaDeQuartos)
{
    Console.WriteLine("Lista de Quartos");
    Console.WriteLine("-----------------------------------------------");
    foreach (Quarto quarto in listaDeQuartos)
    {
        Console.WriteLine($"ID: {quarto.GetQuartoId().ToString("D3")}, Tipo: {quarto.GetTipoQuarto()}, Andar: {quarto.GetAndar()}, Capacidade: {quarto.GetCapacidade()}
    }
}
```

Figura 23 - Método ImprimirListaDeQuartos

Métodos auxiliares de Lista de Quartos:

ObterTipoQuartoAleatorio(): Retorna aleatoriamente um tipo de quarto entre "Individual", "Duplo" e "Studio".

```
private static string ObterTipoQuartoAleatorio()
{
    string[] tiposQuarto = { "Individual", "Duplo", "Studio" }; // Array com os tipos de quarto
    Random random = new Random(); // Objeto Random para gerar números aleatórios
    int indiceAleatorio = random.Next(tiposQuarto.Length); // Gera um número aleatório entre 0 e o tamanho do array
    return tiposQuarto[indiceAleatorio]; // Retorna o tipo de quarto correspondente ao índice gerado
}
```

Figura 24 - Método ObterTipoQuartoAleatório

ObterCapacidadePorTipoQuarto(string tipoQuarto): Retorna a capacidade correspondente a um tipo de quarto (1 para "Individual" e "Studio", 2 para "Duplo").

```
private static int ObterCapacidadePorTipoQuarto(string tipoQuarto)
{
    switch (tipoQuarto)
    {
        case "Individual":
        case "Studio":
            return 1;
        case "Duplo":
            return 2;
        default:
            return 0; // Caso de fallback, caso seja um tipo de quarto desconhecido
    }
}
```

Figura 25 - Método ObterCapacidadePorTipoQuarto

ObterPrecoBasePorTipoQuarto(string tipoQuarto): Retorna o preço base correspondente a um tipo de quarto.

```
private static float ObterPrecoBasePorTipoQuarto(string tipoQuarto)
{
    switch (tipoQuarto)
    {
        case "Individual":
            return 220;
        case "Duplo":
            return 400;
        case "Studio":
            return 350;
        default:
            return 0; // Caso de fallback, caso seja um tipo de quarto desconhecido
    }
}
```

Figura 26 - Método ObterPrecoBasePorTipoQuarto

CalcularPrecoRenda(int andar, float precoBase): Calcula e retorna o preço da renda de um quarto com base no andar e no preço base, aplicando um acréscimo de 5% por andar.

```
private static float CalcularPrecoRenda(int andar, float precoBase)
{
    return precoBase * (1.0f + 0.05f * andar); // Retorna o preço de renda com base no preço base e no andar, com um acréscimo de 5% por andar
}
```

ObterDisponibilidadeAleatoria(): Retorna aleatoriamente true ou false para indicar a disponibilidade de um quarto.

```
private static bool ObterDisponibilidadeAleatoria()
{
    Random random = new Random();
    return random.NextDouble() < 0.5; // Retorna true ou false aleatoriamente
}
```

Figura 27 - Método ObterDisponibilidadeAleatoria

9. MenuResidencia.cs

A classe MenuResidencia em C# representa um menu interativo para interação com as funcionalidades relacionadas à gestão de uma residência.

Método ExibirMenu(): Este método estático contém um loop que exibe o menu da residência. O usuário pode escolher entre várias opções, e a lógica de switch-case trata cada opção de acordo com as seguintes funcionalidades:

Opção 1 - Registar Utilizador: Chama o método CriarUtilizador() para criar e registar um novo utilizador.

Opção 2 - Alterar Registo de Utilizador: A lógica para esta opção ainda não foi implementada no código.

Opção 3 - Listar Utilizadores: Chama o método ImprimirListaDeUtilizadores() para exibir na consola a lista de utilizadores.

Opção 4 - Listar Moradores: Chama o método ImprimirListaDeMoradores() para exibir na consola a lista de moradores.

Opção 5 - Criar Reserva: A lógica para esta opção ainda não foi implementada no código.

Opção 6 - Alterar Reserva: A lógica para esta opção ainda não foi implementada no código.

Opção 7 - Cancelar Reserva: A lógica para esta opção ainda não foi implementada no código.

Opção 8 - Listar Quartos: Chama o método ImprimirListaDeQuartos(listaDeQuartos) para exibir na consola a lista de quartos.

Opção 9 - Sair: Termina o loop, encerrando o programa.

Após a execução de cada opção, o utilizador é solicitado a pressionar Enter para limpar o ecrã e retornar ao menu principal, mantendo a interatividade do utilizador com o sistema.

Documentação (Docfx)

O código-fonte do projeto foi devidamente documentado utilizando a ferramenta DocFX. A documentação inclui comentários incorporados diretamente no código-fonte, seguindo o formato de documentação XML suportado pelo C#. Esses comentários foram estruturados de acordo com as convenções do DocFX para garantir uma documentação clara e coesa.

Github

O código-fonte e a documentação completos do sistema de gestão de reservas em residências estudantis desenvolvido como parte deste trabalho acadêmico estão disponíveis no seguinte repositório do GitHub: https://github.com/juliadoriar/TP_POO_24200_24204.git