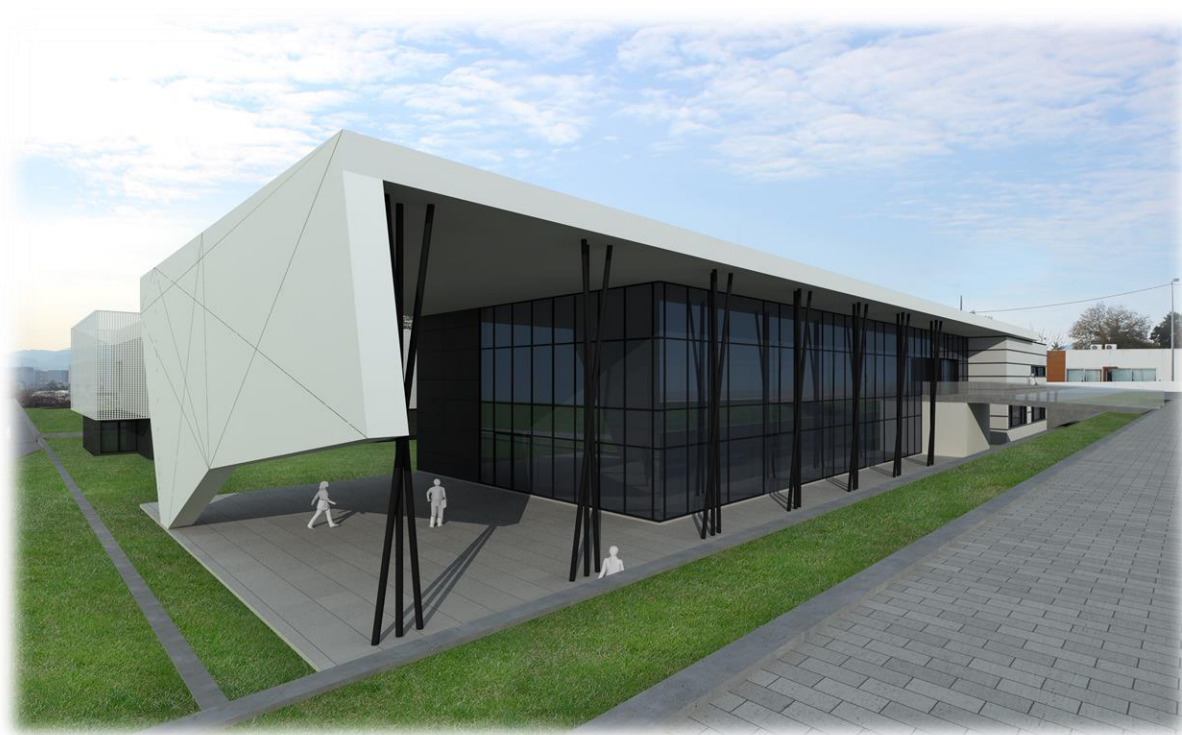


**Escola Superior de Tecnologia
Engenharia de Sistemas Informáticos**

Tiago André Gomes Carneiro | N° 28002



Docente | Ernesto Carlos Casanova Ferreira

Índice

Introdução	3
Fase 1	4
Classes Identificadas.....	4
Classe Users	4
Classe Accommodation	4
Classe Reservation.....	4
Classe Client.....	4
Classe CheckIn.....	5
Estruturas de dados a utilizar.....	5
Listas	5
Fase 2	6
Camada de Apresentação (UI)	7
Camada de Lógica (BLL - Business Logic Layer).....	8
Camada de Acesso a Dados (DAL - Data Access Layer).....	8
Camada de Dados (Data Layer).....	9
Herança em Programação Orientada a Objetos	10
Abstração em Programação Orientada a Objetos	12
Encapsulamento em Programação Orientada a Objetos	14
Testes Unitários	15
Conclusão	17

Introdução

Neste trabalho prático, realizado no âmbito da unidade curricular de Programação Orientada a Objetos, pretendo consolidar conceitos fundamentais do Paradigma Orientado a Objetos, analisar problemas reais, desenvolver capacidades de programação em C#, potenciar a experiência no desenvolvimento de software e assimilar o conteúdo da Unidade Curricular.

O tema selecionado envolve o desenvolvimento de uma aplicação relacionada com a Gestão de Alojamentos Turísticos. Esta aplicação será projetada para oferecer funcionalidades essenciais, incluindo a gestão de reservas, a consulta e o registo de alojamentos disponíveis, o processo de check-in e check-out de clientes, e a administração das informações dos hóspedes. Além disso, a aplicação permitirá ao administrador aprovar ou rejeitar check-ins, fornecendo uma plataforma eficiente e intuitiva para a gestão de alojamentos turísticos.

Fase 1

Classes Identificadas

Classe Users

A classe Users representa os utilizadores do sistema, que podem ser administradores ou clientes. Esta classe armazena informações básicas, incluindo o nome de utilizador, a palavra-passe e o tipo de utilizador. O sistema utiliza esta classe para direccionar o utilizador para o painel adequado (administração ou cliente) após o login. O administrador pode ver e aprovar reservas, enquanto o cliente pode fazer novas reservas e visualizar informações sobre os alojamentos.

Classe Accommodation

Esta classe representa os alojamentos disponíveis no sistema, contendo informações como nome, tipo (por exemplo, quarto duplo ou apartamento), capacidade e preço. A classe Accommodation facilita a listagem e visualização dos alojamentos disponíveis, permitindo que o cliente selecione e visualize detalhes ao fazer uma reserva.

Classe Reservation

A classe Reservation gere o processo de registo de uma reserva e organiza todas as operações relacionadas com o ato de reservar, incluindo a verificação da disponibilidade do alojamento e a criação da reserva. Esta classe centraliza a lógica necessária para registar e consultar reservas, facilitando o processo para os utilizadores.

Classe Client

A classe Client representa os clientes que utilizam o sistema para fazer reservas. Ela armazena os dados de contacto e preferências do cliente, facilitando o processo de reserva. Cada cliente pode ter uma ou mais reservas associadas.

Classe CheckIn

A classe CheckIn representa o processo de entrada do cliente no alojamento reservado. Esta classe armazena o estado do check-in (pendente, aprovado ou recusado) e permite ao administrador aprovar ou rejeitar o check-in com base nas reservas realizadas. Esta funcionalidade é importante para controlar o acesso dos clientes aos alojamentos.

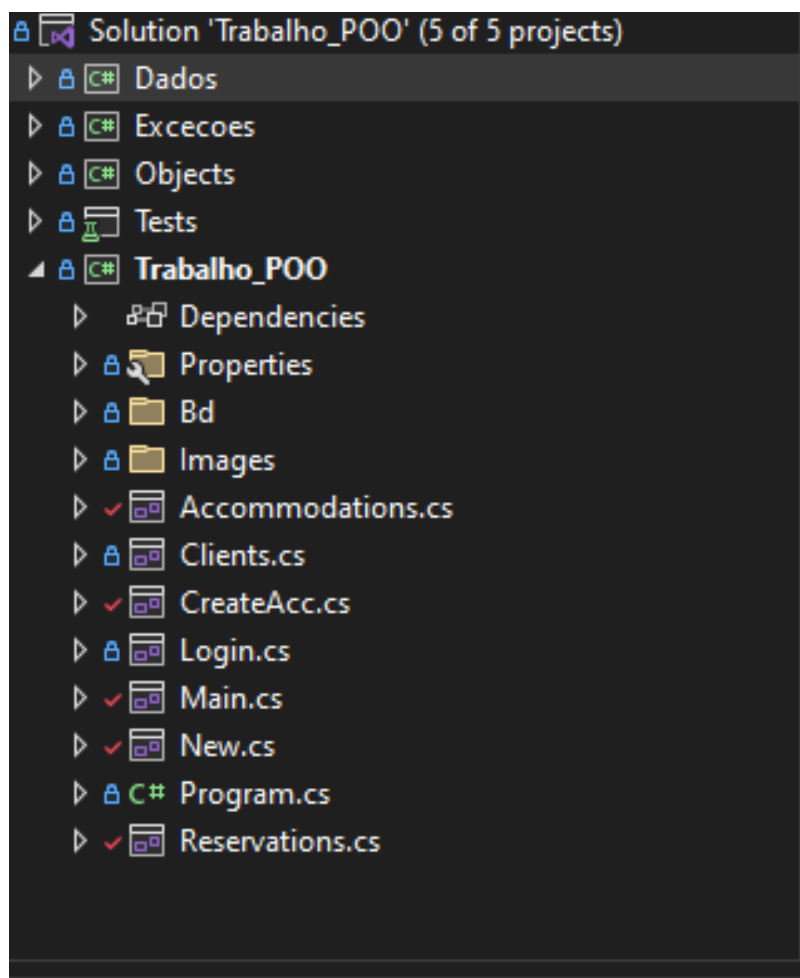
Estruturas de dados a utilizar

Listas

Foram extensivamente utilizadas para manter coleções dinâmicas de objetos. Cada classe possui uma lista correspondente para armazenar os seus elementos associados. Estas listas são específicas para cada instância.

Foram também usadas listas estáticas, para manter informações compartilhadas entre todas as instâncias da classe tornando-se assim uma lista geral da classe

Fase 2

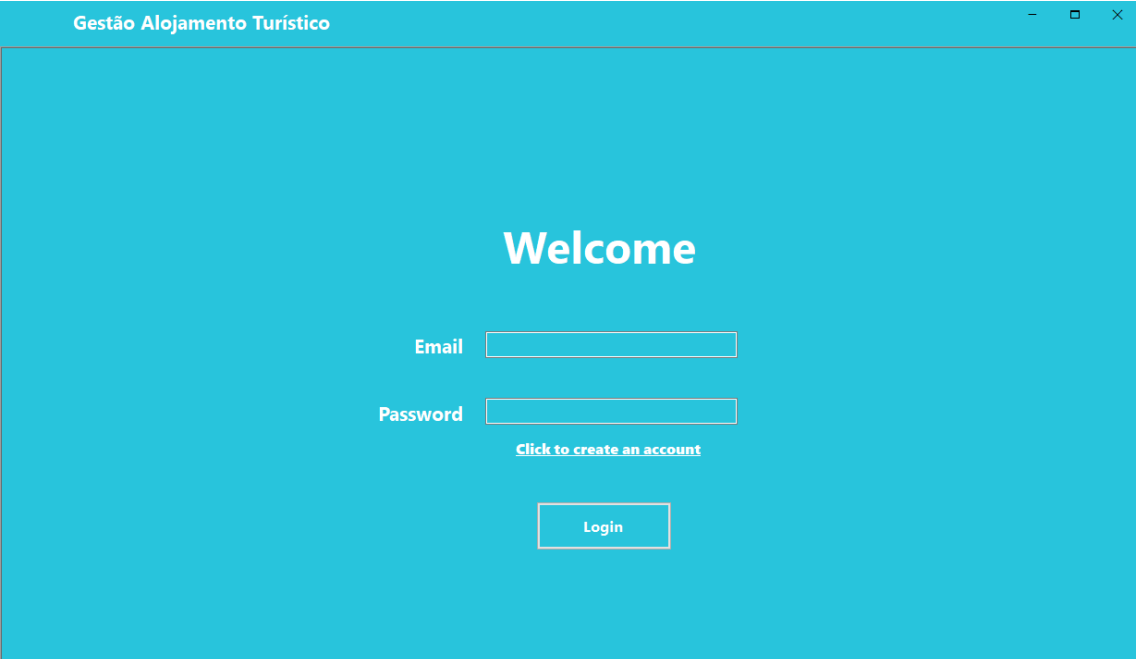


Camada de Apresentação (UI)

A camada de apresentação será responsável pela interface do utilizador. Ela interage diretamente com o utilizador e apresenta as informações ao utilizador através de formulários.

Funcionalidades:

- **Login:** Validar utilizador e redirecionar para o painel apropriado (Admin ou Cliente).
- **Painel de Admin:** Permitir que o administrador visualize e aprove reservas, gerir alojamentos e utilizadores.
- **Painel de Cliente:** Permitir que o cliente faça novas reservas, consulte reservas existentes e visualize os alojamentos disponíveis.



The screenshot shows a web application window with the title "Gestão Alojamento Turístico". The background is a solid teal color. In the center, the word "Welcome" is displayed in a large, white, sans-serif font. Below "Welcome", there are two input fields: the first is labeled "Email" and the second is labeled "Password". Both labels are in a small, white, sans-serif font. Below the "Password" field, there is a link that says "Click to create an account" in a small, white, sans-serif font. At the bottom center, there is a white rectangular button with the word "Login" in a small, black, sans-serif font.

Camada de Lógica (BLL - Business Logic Layer)

Esta camada contém toda a lógica, incluindo as regras e operações específicas do sistema, como validação de utilizadores, reservas e gestão dos dados.

Classes principais:

- **Users:** Responsável por manter o estado da sessão do utilizador.
- **Reservations:** Lida com a criação e gestão das reservas.
- **Accommodations:** Gere as operações dos alojamentos, como validação e carregamento de dados.

Camada de Acesso a Dados (DAL - Data Access Layer)

A camada de acesso a dados lida com a manipulação dos dados armazenados nos ficheiros **.txt**. Ela abstrai as operações de leitura e escrita e oferece funções para carregar e guardar os dados.

Exemplo de código para a camada de acesso a dados:

```
namespace Dados
{
    14 references
    public class Users : User
    {
        [Attributes]
        [Constructor]

        #region Methods
        /// <summary>
        /// Load all users from the file and add them to the list.
        /// </summary>
        /// <param name="users">The list to store users loaded from the file.</param>
        /// <returns>True if users are successfully loaded, otherwise false.</returns>
        1 reference
        public static bool LoadUsers(out List<User> users)
        {
            users = new List<User>();

            try
            {
                if (!File.Exists(filePath))
                    throw new FileNotFoundException("Arquivo de usuários não encontrado.");

                var lines = File.ReadLines(filePath);

                foreach (var line in lines)
                {
                    var user = BuildUser(line);
                    if (user != null)
                    {
                        users.Add(user);
                    }
                }

                return users.Count > 0;
            }
            catch (Exception ex)
            {
                throw new Exception("Ocorreu um erro: " + ex.Message);
            }
        }
    }
}
```


Camada de Dados (Data Layer)

Nesta camada, os dados são armazenados em ficheiros .txt. Esses ficheiros simulam uma base de dados simples, mas com funcionalidades limitadas.

Ficheiros principais:

- **Users.txt:** Contém informações sobre os utilizadores.
- **Accommodations.txt:** Contém detalhes sobre os alojamentos disponíveis.
- **Reservations.txt:** Contém as reservas feitas pelos clientes.

Exemplo de conteúdo dos ficheiros txt:

Accommodations.txt

d35a971c,Teste1,Apartamento,Famalicao,200,10,True,BeachsideVilla.jpeg 804277a9,Teste2,Apartment,Famalicao,200,20,False,BeachsideVilla.jpeg

Herança em Programação Orientada a Objetos

A **herança** é um dos pilares fundamentais da programação orientada a objetos (POO), permitindo que uma classe "filha" herde atributos e métodos de uma classe "base". Esta característica é útil para reutilização de código, melhorar a organização e facilitar a manutenção do sistema.

Classe Base: User

A classe **User** é uma classe que serve como base para outras classes de utilizadores no sistema. Ela define os atributos comuns a todos os tipos de utilizadores (como **ID**, **Nome**, **Email**, **Data de Nascimento**, **Password**, e **Role**).

Benefícios da Estrutura de Herança:

1. **Reutilização de Código:** A classe **User** armazena as propriedades comuns de todos os utilizadores, o que evita a duplicação de código.
2. **Facilidade de Expansão:** Se no futuro precisar de adicionar novos tipos de utilizadores, como "Administrador", "Cliente", ou "Gerente", basta criar novas classes que herdem de **User** e implementar a lógica.

```
13 references
public abstract class User
{
    #region Attributes
    3 references
    public Guid Id { get; protected set; }
    4 references
    public string Name { get; protected set; }
    5 references
    public string Email { get; protected set; }
    2 references
    public DateTime DataNascimento { get; protected set; }
    3 references
    public string Password { get; protected set; }
    2 references
    public string Role { get; protected set; }
    #endregion

    Properties

    #region Constructor
    /// <summary>
    /// Initializes a new instance of the <see cref="User"/> class with the specified parameters.
    /// </summary>
    /// <param name="id">The unique identifier of the user.</param>
    /// <param name="name">The name of the user.</param>
    /// <param name="email">The email address of the user.</param>
    /// <param name="dataNascimento">The birth date of the user.</param>
    /// <param name="password">The password for the user.</param>
    /// <param name="role">The role assigned to the user.</param>
    1 reference
    public User(Guid id, string name, string email, DateTime dataNascimento, string password, string role)
    {
        Id = id;
        Name = name;
        Email = email;
        DataNascimento = dataNascimento;
        Password = password;
        Role = role;
    }
    #endregion
}
```

Classe Filha: Users

A classe Users herda de User e implementa funcionalidades específicas relacionadas ao gerenciamento de utilizadores, como o carregamento de dados dos utilizadores a partir de um ficheiro, a validação de campos, e a criação de novos utilizadores.

1. Atributos e Construtor

A classe Users tem um atributo estático users que mantém uma lista de objetos Users. Isto permite manipular a lista de utilizadores sem a necessidade de uma instância da classe. Além disso, o construtor valida os dados do utilizador (nome, email, data de nascimento, etc.).

2. Métodos:

- **LoadUsers(out List<User> users):** Carrega todos os utilizadores a partir de um ficheiro .txt.
- **BuildUser(string line):** Cria um objeto Users a partir de uma linha do ficheiro.
- **AddUser(Users user):** Adiciona um novo utilizador à lista e ao ficheiro. Verifica se o utilizador já existe antes de adicioná-lo.
- **ValidateUserExists(string email):** Verifica se já existe um utilizador com o mesmo email na lista.
- **ValidateFields():** Valida os campos do utilizador (nome, email, etc.).

```
namespace Dados
{
    14 references
    public class Users : User
    {
        #region Attributes
        private static List<Users> users = new List<Users>();
        private const string filePath = "C:\\Projeto_POO_28882-dev\\Projeto_POO_28882-dev\\Trabalho_POO\\Bd\\Users.txt";
        #endregion

        [Constructor]

        #region Methods
        /// <summary>
        /// Load all users from the file and add them to the list.
        /// </summary>
        /// <param name="users">The list to store users loaded from the file.</param>
        /// <returns>True if users are successfully loaded, otherwise false.</returns>
        1 reference
        public static bool LoadUsers(out List<User> users)
        {
            users = new List<User>();

            try
            {
                if (!File.Exists(filePath))
                    throw new FileNotFoundException("Arquivo de usuários não encontrado.");

                var lines = File.ReadLines(filePath);

                foreach (var line in lines)
                {
                    var user = BuildUser(line);
                    if (user != null)
                    {
                        users.Add(user);
                    }
                }

                return users.Count > 0;
            }
            catch (Exception ex)
            {
                throw new Exception("Ocorreu um erro: " + ex.Message);
            }
        }
    }
}
```

Abstração em Programação Orientada a Objetos

A **abstração** permite representar conceitos complexos de forma simplificada, ocultando detalhes desnecessários e focando apenas nas funcionalidades essenciais.

Classe Abstrata: Accommodation

Criamos a classe abstrata **Accommodation** para encapsular os atributos e comportamentos básicos de um alojamento. Ela atua como uma base para todas as classes que representam alojamentos.

```
5 references
public abstract class Accommodation
{
    #region Attributes
    7 references
    public Guid AccommodationID { get; set; }
    9 references
    public string Name { get; set; }
    7 references
    public string Type { get; set; }
    8 references
    public string Location { get; set; }
    8 references
    public decimal Price { get; set; }
    8 references
    public int Capacity { get; set; }
    9 references
    public bool Available { get; set; }
    4 references
    public string Image { get; set; }
    #endregion

    Constructor

    #region Methods
    /// <summary>
    /// Sets the availability status of the accommodation.
    /// </summary>
    /// <param name="availability">New availability status.</param>
    1 reference
    public void SetAvailability(bool availability)
    {
        Available = availability;
    }

    /// <summary>
    /// Abstract method to get the availability status of the accommodation.
    /// </summary>
    /// <returns>Returns true if the accommodation is available, false otherwise.</returns>
    1 reference
    public abstract bool GetAvailability();
    #endregion
}
```

Classe Concreta: Accommodations

A classe **Accommodations** herda de **Accommodation** e implementa os métodos abstratos definidos na classe base. Ela adiciona comportamentos específicos para os alojamentos.

```
22 references
public class Accommodations : Accommodation
{
    Attributes

    Constructor

    Properties

    #region Methods
    /// <summary>
    /// Gets the availability status of the accommodation.
    /// </summary>
    /// <returns>True if the accommodation is available, otherwise false.</returns>
    1 reference
    public override bool GetAvailability()
    {
        return Available;
    }
}
```

Encapsulamento em Programação Orientada a Objetos

O **encapsulamento** é um dos pilares fundamentais da programação orientada a objetos (POO), ao lado da **abstração**, **herança** e **polimorfismo**. O encapsulamento visa esconder os detalhes internos de uma classe e expor apenas a interface necessária para interagir com ela. Este princípio promove a proteção dos dados, tornando a manipulação de objetos mais segura e controlada.

Classe Reservation

Na classe Reservation, os atributos foram definidos como private ou protected, restringindo o acesso direto aos dados.

Para permitir o acesso controlado, foram utilizadas propriedades com os métodos get e set.

```
8 references
public abstract class Reservation
{
    #region Attributes
    3 references
    public Guid ReservationID { get; protected set; }
    2 references
    public Guid ClientID { get; protected set; }
    2 references
    public string ClientName { get; protected set; }
    4 references
    public Guid AccommodationID { get; protected set; }
    2 references
    public string AccommodationName { get; protected set; }
    7 references
    public DateTime CheckInDate { get; protected set; }
    7 references
    public DateTime CheckOutDate { get; protected set; }
    3 references
    public decimal TotalPrice { get; protected set; }
    4 references
    public string ReservationStatus { get; protected set; }
    #endregion

    Properties

    Constructor

    Methods
}
```

Testes Unitários

```
[TestInitialize]
0 references
public void Setup()
{
    user = new Users(
        id: Guid.NewGuid(),
        name: "Teste",
        email: "teste@gmail.com",
        dataNascimento: new DateTime(2004,2,11),
        password: "teste123",
        role: "Client"
    );

    accommodation = new Accommodations(
        accommodationId: Guid.NewGuid(),
        name: "Beachside Villa",
        type: "Villa",
        location: "Miami Beach",
        price: 250.00m,
        capacity: 6,
        available: true,
        image: ""
    );

    reservation = new Reservations(
        reservationId: Guid.NewGuid(),
        clientId: user.Id,
        clientName: user.Name,
        accommodationId: accommodation.AccommodationID,
        accommodationName: accommodation.Name,
        checkInDate: DateTime.Now.AddDays(1),
        checkOutDate: DateTime.Now.AddDays(7),
        totalPrice: 1750.00m,
        reservationStatus: "Confirmed"
    );
}

[TestMethod]
0 references
public void LoadUsersTest()...

[TestMethod]
0 references
public void AddUsersTest()...

[TestMethod]
0 references
public void LoadReservationsTest()...
}
```

TestInitialize - Setup()

O método `Setup()` é executado antes de cada teste. Ele é marcado com o atributo `[TestInitialize]`, o que significa que o conteúdo desse método será executado automaticamente antes de cada método. Este método prepara os dados necessários para os testes seguintes.

1. **Utilizador (user):** Cria uma instância de `Users` com valores específicos (como ID, nome, e-mail, data de nascimento, palavra-passe e função).
2. **Alojamento (accommodation):** Cria uma instância de `Accommodations` representando um alojamento com detalhes como ID, nome, tipo, localização, preço, capacidade e disponibilidade.
3. **Reserva (reservation):** Cria uma instância de `Reservations` para representar uma reserva, incluindo detalhes como ID da reserva, ID e nome do cliente, ID e nome da acomodação, datas de check-in e check-out, preço total e estado da reserva.

LoadUsersTest

Este método testa a funcionalidade de carregar os utilizadores. Ele chama o método `LoadUsers` da classe `Users`, que procura os utilizadores do ficheiro `Users.txt` e carrega-os para a lista `users`.

Objetivo: Verificar se a função `LoadUsers` devolve `true`, o que indica que a operação de carregar os utilizadores foi bem-sucedida.

AddUsersTest

Este método testa a funcionalidade de adicionar um utilizador. Ele chama o método `AddUser` da classe `Users`, passando a instância de `user` como argumento. Este método adiciona um novo utilizador ao ficheiro `"Users.txt"`.

Objetivo: Verificar se a função `AddUser` devolve `true`, indicando que o utilizador foi adicionado com sucesso.

Conclusão

O projeto Gestão de Alojamentos Turísticos foi desenvolvido com o objetivo de criar um sistema eficiente e intuitivo para o registo, consulta, reserva e gestão de alojamentos turísticos. Durante o desenvolvimento, foram aplicados conceitos de programação orientada a objetos (POO) para criar um sistema de fácil manutenção.

Principais Funcionalidades Desenvolvidas

1. **Gestão de Utilizadores:** O sistema permite o registo e autenticação de utilizadores (clientes e administradores). Os utilizadores podem consultar e reservar alojamentos disponíveis.
2. **Gestão de Alojamentos:** Administradores podem adicionar, editar e remover alojamentos. Cada alojamento inclui informações detalhadas como tipo, localização, capacidade e preço.
3. **Reservas de Alojamentos:** Os utilizadores podem consultar a disponibilidade dos alojamentos e realizar reservas. O sistema calcula o preço total com base nas datas de check-in e check-out.
4. **Administração de Reservas:** Os administradores podem ver, alterar o estado ou cancelar reservas, facilitando a gestão das operações.

Tecnologias Utilizadas

- **C#:** Linguagem de programação principal, que foi usada para desenvolver a lógica do sistema.
- **Windows Forms:** Para a criação da interface gráfica, proporcionando uma interação fácil e intuitiva.

Melhorias e Funcionalidades Futuras

- **Integração com sistemas de pagamento:** Implementação de um sistema de pagamento online para processar as reservas e garantir a segurança das transações.
- **Aplicação Mobile:** Desenvolver uma versão mobile do sistema para aumentar a acessibilidade dos clientes.