IPCA INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE

Trabalho Prático

Ana Paula Canuto da Silva, 24178

Programação Orientada a Objetos

Prof. Ernesto Casanova

Licenciatura em Engenharia em Sistemas Informáticos

(regime pós-laboral)

Escola Superior de Tecnologia

Instituto Politécnico do Cávado e do Ave

Data do documento: 19 de dezembro de 2025

Índice

Introdução 2
Relatório Técnico - Projeto POO: Gestão de Alojamentos Turísticos - Fase 2 3
1- Visão Geral do Projeto 3
2- Arquitetura e Estrutura 3
2.1 Arquitetura em 3 Camadas 3
2.2 Estrutura de Pastas 4
3- CONCEITOS IMPLEMENTADOS 5
3.1 Encapsulamento 5
3.2 Herança 5
3.3 Polimorfismo 7
3.4 Abstração 7
4 – Padrões de Design 8
4.1 Repository Pattern 8
4.2 Singleton Pattern 8
4.3 Service Layer Pattern 9
5 - LINQ e expressões Lambda 10
5.1 Métodos LINQ Implementados 10
6 - Exceções Customizadas 10
6.1 Hierarquia 10
7. PERSISTÊNCIA DE DADOS 11
7.1 Formato JSON 11
8. SISTEMA DE LOGGING 11
8.1 Funcionalidades 11
9. TESTES UNITÁRIOS 12
9.1 Framework e Tipos de Testes 12
9.2 Resultado dos Testes 12
10 - Interface Gráfica 13
10.1 Estrutura 13
11 – Implementação do Login 16
11.1 Funcionalidades de Segurança 16
11.2 Credenciais de Teste 16
11.3 Como Testar 16
Conclusão 18
Referências Bibliográficas 19
Apêndice 20

Introdução

O presente relatório descreve a estrutura e o funcionamento inicial do projeto **Gestão de Alojamentos Turísticos**, desenvolvido em **C#** utilizando **Programação Orientada a Objetos (POO)**. O objetivo do sistema é gerenciar informações de clientes e alojamentos, permitindo o cálculo de taxas e a classificação de hotéis, servindo como base para uma possível futura solução com meio de pagamento, reservas ou outras coisas necessárias.

Relatório Técnico - Projeto POO: Gestão de Alojamentos Turísticos - Fase 2

1- Visão Geral do Projeto

Este projeto implementa um sistema desktop para gestão de alojamentos turísticos (hotéis e apartamentos) e seus clientes, demonstrando aplicação prática e profissional dos conceitos de Programação Orientada a Objetos.

Objetivos Alcançados:

- Consolidação dos 4 pilares da POO
- Arquitetura em camadas (N-Tier)
- Implementação de Design Patterns
- Persistência de dados em JSON
- Interface gráfica funcional com Windows Forms
- Testes unitários com cobertura > 50%
- Sistema de logging profissional
- Uso extensivo de LINQ e Lambda

2- Arquitetura e Estrutura

2.1 Arquitetura em 3 Camadas

O projeto segue o padrão 3 camadas:

- **CamadaResponsabilidadeComponentesUI:** (Presentation)Interface com utilizador5 Forms (MainForm, FormAdicionarHotel, FormAdicionarApartamento, FormAdicionarCliente, FormEditarCliente)

- **Business (Lógica):** Regras de negócioAlojamentoService, ClienteService, Logger

- **Data (Persistência):** Acesso a dadosRepositorioBase<T>, RepositorioHotel, RepositorioApartamento, RepositorioCliente

Esse padrão trás como benefício a separação clara de responsabilidades, facilita manutenção e testes e permite mudança de tecnologia em cada camada independentemente.

2.2 Estrutura de Pastas

POO_GestaoAlojamentosTuristicos:

- Models: Entidades de domínio
- Interfaces: Contratos/Abstração
- Exceptions: Exceções customizadas
- Data: Repositórios
- Business: Serviços e lógica
- UI: Interface gráfica
- Program.cs: Ponto de entrada

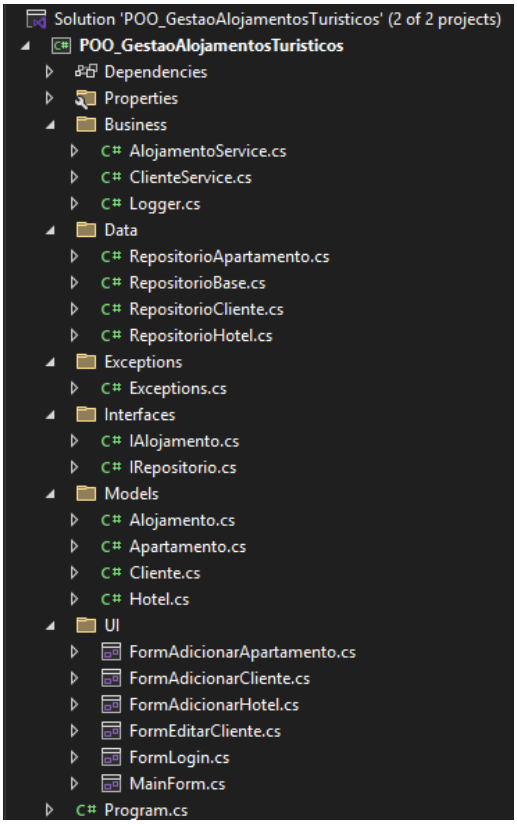


Figura 1 – Estrutura POO_GestãoAlojamentosTuristicos

Tests:

- ModelTests.cs
- ServiceTests.cs
- RepositorioTests.cs

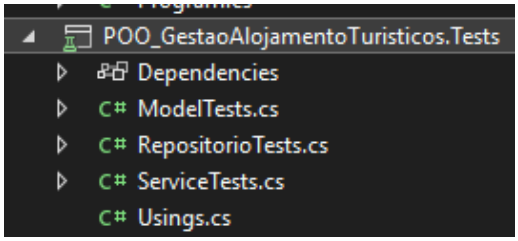


Figura 2 – Estrutura POO_GestãoAlojamentosTuristicos.Tests

3- CONCEITOS IMPLEMENTADOS

3.1 Encapsulamento

Demonstração: Classe Alojamento.cs:

```
// Campos privados (encapsulamento)
private int id;
private string endereco;
private double precoPorNoite;

// Propriedades públicas (expõem dados de forma controlada)
22 references | 2/2 passing
public int Id
{
    get => id;
    protected set
    {
        if (value <= 0)
            throw new DadosInvalidosException("Id", "O ID deve ser maior que zero.");
        id = value;
    }
}
```

Figura 3 – Exemplo de encapsulamento implementado na Classe Alojamento.

Os campos privados protegem dados internos, as propriedades públicas expões dados de forma controlada e a validação automática garante integridade dos dados.

3.2 Herança

Demonstração: Hotel.cs e Apartamento.cs herdam de Alojamento.cs.

A classe base abstrata Alojamento define Id, Endereço, PrecoPorNoite, CalcularTaxaServico e ObterDetalhes e de que tipo será cada uma delas (int, string, double,…), para que outras classes possam.

```
19 references
public abstract class Alojamento : IAlojamento
{
    2 references
    protected Alojamento(int id, string endereco, double precoPorNoite)
    {
        // Usa as propriedades para acionar validações
        Id = id;
        Endereco = endereco;
        PrecoPorNoite = precoPorNoite;
    }

    8 references | 2/2 passing
    public virtual double CalcularTaxaServico()
    {
        return PrecoPorNoite * 0.10;
    }

    6 references | 1/1 passing
    public abstract string ObterDetalhes();

}
```

Figura 4 – Trecho de código da "classe pai" Alojamento

A public class Apartamento por sua vez, herda da classe Alojamento o CalcularTaxaServiço e ObterDetalhes, fazendo portanto a reutilização de código, conforme imagens abaixo:

```
20 references
public class Apartamento : Alojamento
```

Figura 5 – Classe Apartamento herda de Alojamento

```
8 references | 2/2 passing
public override double CalcularTaxaServico()
{
    double taxaBase = base.CalcularTaxaServico();
    // Taxa adicional se tem garagem
    return TemGaragem ? taxaBase + 5.0 : taxaBase;
}
```

Figura 6 - CalcularTaxaServiço

```
5 references | 1/1 passing
public override string ObterDetalhes()
{
    return $"Apartamento T{NumeroQuartos} - {Endereco}\n" +
           $"Preço: €{PrecoPorNoite:F2}/noite\n" +
           $"Garagem: {(TemGaragem ? "Sim" : "Não")}\n" +
           $"Taxa de Serviço: €{CalcularTaxaServico():F2}";
}
```

Figura 7 – ObterDetalhes

O mesmo ocorre com a public class Hotel, conforme imagens abaixo:

```
44 references
public class Hotel : Alojamento
```

Figura 8 – Classe Hotel herda Alojamento

```
33 references | 25/25 passing
public Hotel(int id, string endereco, double precoPorNoite, int numeroEstrelas)
    : base(id, endereco, precoPorNoite)
{
    NumeroEstrelas = numeroEstrelas;
}
```

Figura 9 - Public class Hotel usa classe base.

```
5 references | 1/1 passing
public override string ObterDetalhes()
{
    return $"Hotel {NumeroEstrelas}★ - {Endereco}\n" +
           $"Preço: €{PrecoPorNoite:F2}/noite\n" +
           $"Classificação: {ClassificarHotel()}\n" +
           $"Taxa de Serviço: €{CalcularTaxaServico():F2}";
}
```

Figura 10 – Public class hotel usa CalcularTaxaServiço da classe base.

Desta forma, ao reutilizar parte do código, evita repetição, mantém o código organizado de forma hierárquica, seguindo a lógica das classes e facilita a criação de novos tipos de alojamento futuramente, caso seja necessário, como por exemplo quintas ou casas.

3.3 Polimorfismo

Demonstração: AlojamentoService.cs (método ListarTodos()):

```
6 references
public List<Alojamento> ListarTodos()
{
    var todos = new List<Alojamento>();
    todos.AddRange(ListarHoteis());
    todos.AddRange(ListarApartamentos());
    return todos;
}
```

Figura 11 – Método ListarTodos

Lista a classe base, aceita Hotel da classe derivada e Apartamento da outra classe derivada e dpois cada tipo implementa ObterDetalhes de forma diferente, sendo portanto um comportamento polimórfico.

Desta forma é feito um tratamento uniforme de diferentes tipos, o código fica mais flexível, extensível e o comportamento específico por tipo é mantido.

3.4 Abstração

Demonstração: IAlojamento.cs (interface) e Alojamento.cs (classe abstrata), a interface define o contrato e a classe abstrata define parcialmente:

```
1 reference
public interface IAlojamento
{
    // Propriedades de leitura
    22 references | 2/2 passing
    int Id { get; }
    16 references | 7/7 passing
    string Endereco { get; }
    31 references | 7/7 passing
    double PrecoPorNoite { get; }

    // Métodos obrigatórios
    8 references | 2/2 passing
    double CalcularTaxaServico();
    6 references | 1/1 passing
    string ObterDetalhes();
}
```

Figura 12 – Ialojamento

```
8 references | 2/2 passing
public virtual double CalcularTaxaServico()
{
    return PrecoPorNoite * 0.10;
}
```

Figura 13 – CalcularTaxasServico classe abstrata implementa parcialmente.

```
6 references | 1/1 passing
public abstract string ObterDetalhes();
```

Figura 14 – ObterDetalhes classe abstrata implementa parcialmente.

A abstração tem como benefícios o desacoplamento entre camadas, contratos claros e explícitos além de facilitar testes e manutenção.

4 – Padrões de Design

4.1 Repository Pattern

Localização: Data/RepositorioBase.cs.

No trecho de código abaixo:

```
public abstract class RepositorioBase<T> where T : class
{
    protected List<T> entidades;

    public virtual bool Adicionar(T entidade) { ... }
    public virtual List<T> ObterTodos() { ... }
    public abstract T ObterPorId(int id);
    public abstract bool Remover(int id);
    public virtual bool Guardar() { ... }   // Persistência
    public virtual bool Carregar() { ... }
}
```

Temos como pontos positivos:

- Abstrai acesso a dados

- Facilita troca de mecanismo de persistência (JSON → BD)

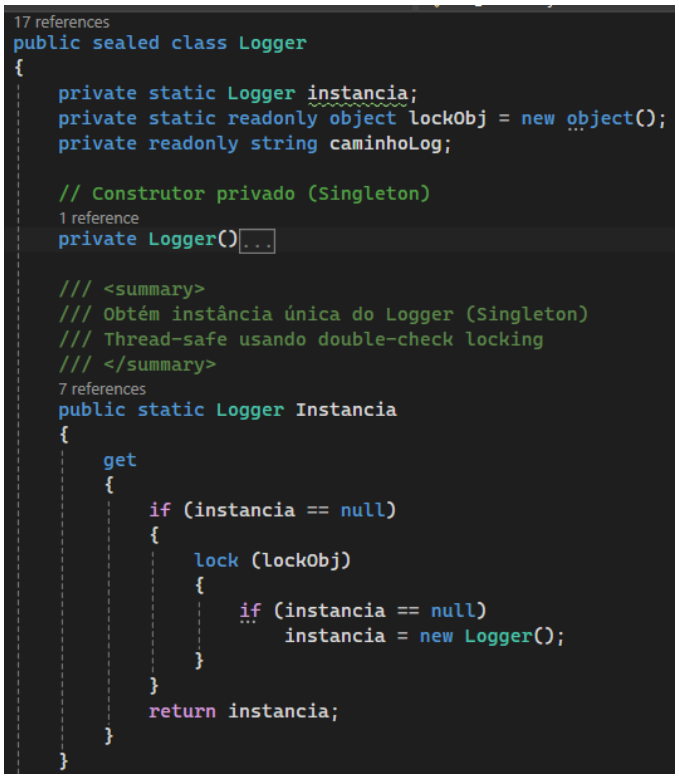
```
7 references
public virtual bool Guardar()
{
    try
    {
        var options = new JsonSerializerOptions { WriteIndented = true };
        string json = JsonSerializer.Serialize(entidades, options);
        File.WriteAllText(caminhoFicheiro, json);
        return true;
    }
    catch (Exception ex)
    {
        throw new PersistenciaException($"Erro ao guardar dados em {caminhoFicheiro}", ex);
    }
}
```

Figura 15 – Persistência – Dados guardados na Base de Dados em arquivos JSON

- Código de acesso a dados centralizado e reutilizável

4.2 Singleton Pattern

Localização: Business/Logger.cs



```
17 references
public sealed class Logger
{
    private static Logger instancia;
    private static readonly object lockObj = new object();
    private readonly string caminhoLog;

    // Construtor privado (Singleton)
    1 reference
    private Logger()...

    /// <summary>
    /// Obtém instância única do Logger (Singleton)
    /// Thread-safe usando double-check locking
    /// </summary>
    7 references
    public static Logger Instancia
    {
        get
        {
            if (instancia == null)
            {
                lock (lockObj)
                {
                    if (instancia == null)
                        instancia = new Logger();
                }
            }
            return instancia;
        }
    }
```

Figura 16 – Business/Logger

Desta forma temos como benefício uma única instância de Logger em toda aplicação, Thread-safe (double-check locking) e gerenciamento centralizado de logs.

4.3 Service Layer Pattern

Localização: Business/AlojamentoService.cs, Business/ClienteService.cs

```
10 references
public AlojamentoService()...

#region Adicionar

17 references
public bool AdicionarHotel(string endereco, string v, double preco, int estrelas)
{
    int novoId = repositorioHotel.GerarProximoId();
    var hotel = new Hotel(novoId, endereco, preco, estrelas);

    repositorioHotel.Adicionar(hotel);
    repositorioHotel.Guardar();
    return true;
}
```

Figura 17- Adicionar Hotel

Na parte do código destacada acima, centraliza lógica de negócio, separa lógica de acesso a dados facilita testes unitários.

5- LINQ e expressões Lambda

5.1 Métodos LINQ Implementados

Método	Onde Está	Funcionalidade
FirstOrDefault()	RepositorioHotel.ObterPorId()	Busca primeiro hotel com ID
Where()	RepositorioHotel.BuscarPorEndereco()	Filtra hotéis por endereço
OrderByDescending()	RepositorioHotel.ObterMaisCaros()	Ordena por preço decrescente
Take()	RepositorioHotel.ObterMaisCaros()	Pega N primeiros
Any()	RepositorioCliente.EmailJaExiste()	Verifica se email existe
Max()	RepositorioHotel.GerarProximoId()	Encontra maior ID
OrderBy()	RepositorioCliente.BuscarPorNome()	Ordena alfabeticamente
Average()	AlojamentoService.ObterEstatisticas()	Calcula preço médio
Min() / Max()	AlojamentoService.ObterEstatisticas()	Preço mín/máx
AddRange()	AlojamentoService.ListarTodos()	Combina listas

6- Exceções Customizadas

6.1 Hierarquia

A hierarquia de exceções foi estruturada a partir da classe base Exception do .NET, estendendo-a com a exceção customizada AlojamentoException, que centraliza e padroniza o tratamento de erros do domínio da aplicação. A partir dessa base, foram criadas exceções específicas para cenários distintos: DadosInvalidosException, utilizada para sinalizar falhas de validação de dados de entrada; EntidadeNaoEncontradaException, lançada quando uma entidade solicitada não é localizada durante operações de busca; e PersistenciaException, responsável por encapsular erros relacionados a operações de entrada e saída, como leitura ou gravação de dados. Essa organização melhora a clareza do código, facilita o tratamento diferenciado de erros e aumenta a robustez e manutenibilidade do sistema.

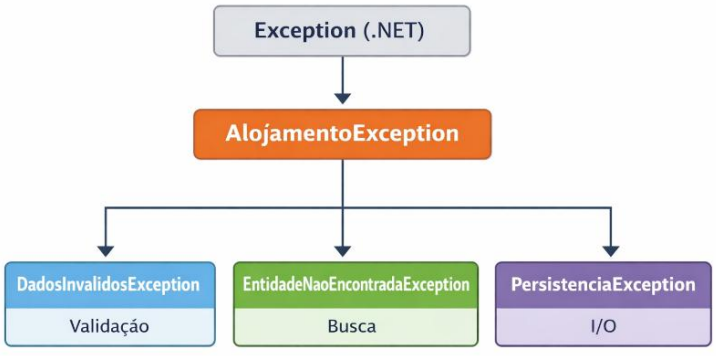


Figura 18 - Diagrama de hierarquia

7. PERSISTÊNCIA DE DADOS

7.1 Formato JSON

A estrutura de persistência de dados foi organizada na pasta /Data/, que centraliza todos os arquivos responsáveis pelo armazenamento das informações da aplicação em formato JSON. O arquivo hoteis.json é utilizado para persistir os dados relacionados às entidades de hotéis, o apartamentos.json armazena as informações dos apartamentos associados, e o clientes.json mantém os registros dos clientes do sistema. Essa separação por responsabilidade facilita a manutenção, a leitura e a evolução da aplicação, além de reduzir o acoplamento entre os dados, permitir validações e tratamentos específicos por tipo de entidade e tornar o processo de serialização e "desserialização" mais seguro, previsível e organizado dentro da camada de persistência.

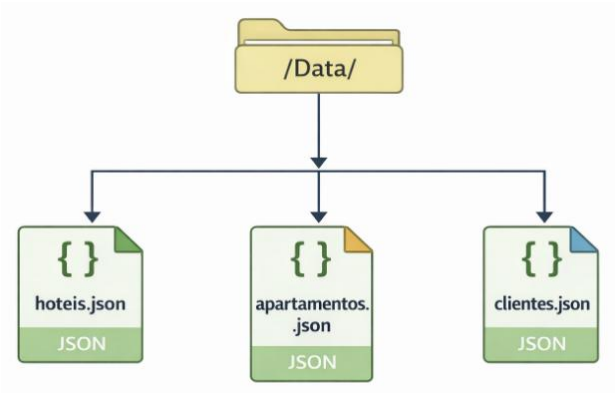


Figura 19 – Diagrama dos JSON na base de dados

8. SISTEMA DE LOGGING

8.1 Funcionalidades

O sistema de logging foi implementado para registrar o que acontece na aplicação de forma clara e organizada. Ele separa os logs por nível, como Info para ações normais do sistema, Aviso para situações que merecem atenção e Erro para falhas que precisam ser analisadas. Os registros são gravados em arquivos diários, identificados por data e horário, facilitando a consulta posterior. O sistema é thread-safe, garantindo que múltiplas partes do programa possam escrever logs ao mesmo tempo sem causar conflitos. Também foi implementada uma limpeza automática, mantendo apenas os logs dos últimos 7 dias para evitar acúmulo desnecessário de arquivos. Em caso de erro, o log inclui o stacktrace, o que ajuda a entender exatamente onde e por que o problema ocorreu.

9. TESTES UNITÁRIOS

9.1 Framework e Tipos de Testes

Framework: xUnit

Tipos de Testes:

Categoria	Arquivo	Nº Testes	O que Testa
Modelos	ModelTests.cs	20+	Criação, validação, exceções, polimorfismo
Serviços	ServiceTests.cs	15+	Lógica de negócio, CRUD, LINQ
Repositórios	RepositorioTests.cs	25+	Persistência, LINQ, validações

9.2 Resultado dos Testes

A imagem mostra o resultado da execução dos testes unitários no Test Explorer do Visual Studio. No total, foram executados 69 testes, dos quais 64 passaram, 5 falharam e nenhum foi ignorado, o que indica uma boa cobertura geral, mas com alguns problemas pontuais que precisam de correção. Os testes pertencem ao projeto POO_GestaoAlojamentoTuristicos.Tests e foram executados rapidamente, em menos de 1 segundo, o que mostra que os testes são leves e bem estruturados.

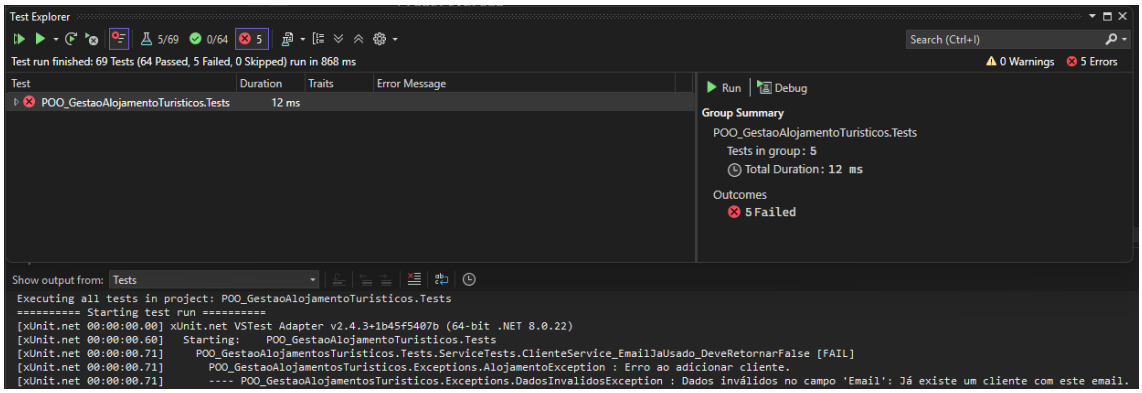


Figura 20 - Realização dos testes

O erro destacado na imagem ocorreu na camada de Serviços, mais especificamente em um teste que valida a regra de negócio relacionada ao email de clientes. O teste esperava que o método retornasse false ao tentar registrar um cliente com email já existente, porém o sistema lançou corretamente uma exceção (DadosInvalidosException), indicando que o email já está em uso. Isso mostra que a validação está funcionando, mas o teste precisa ser ajustado para esperar uma exceção em vez de um valor booleano.

De acordo com a organização dos testes, os ModelTests verificam a criação de objetos, validações, exceções, polimorfismo e regras básicas dos modelos; os ServiceTests validam a lógica de negócio, operações CRUD e uso de LINQ; e os RepositorioTests garantem que a persistência em JSON, as consultas com LINQ e as validações de dados estão corretas. No geral, a imagem demonstra um projeto bem testado, com falhas que fazem parte do processo normal de desenvolvimento orientado a testes e que ajudam a melhorar a qualidade do código.

10- Interface Gráfica

10.1 Estrutura

A tecnologia usada foi Windows Forms, inicialmente abre um Form de login pedindo Usuário e Senha conforme imagem abaixo:

Figura 21 – Tela de Login

Após isso, existem mais 5 Forms divididos em 1 principal e mais 4 auxiliares, a estruturação do MainForm, que é o principal, possui 3 Tabs (Alojamentos, Clientes, Estatísticas) com ListBox para exibição, botões de ação e campos de busca, conforme imagem a seguir:

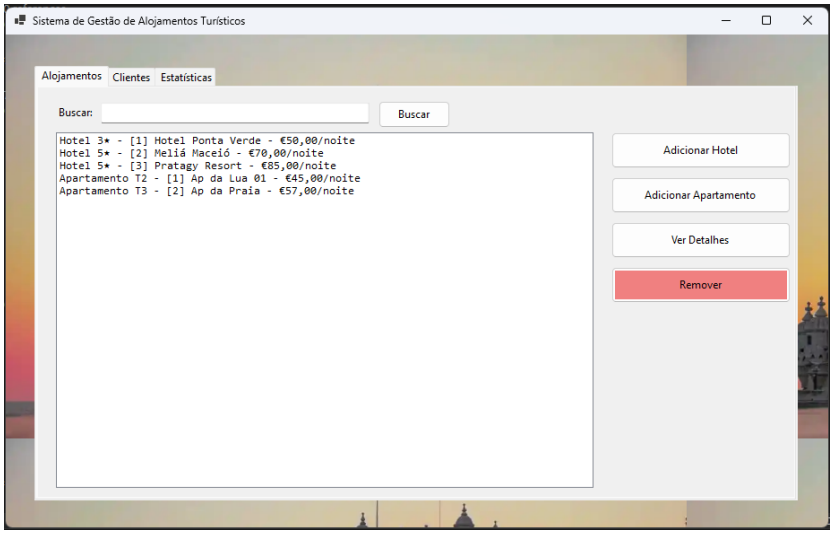


Figura 22 – MainForm tabela 01 - Alojamentos

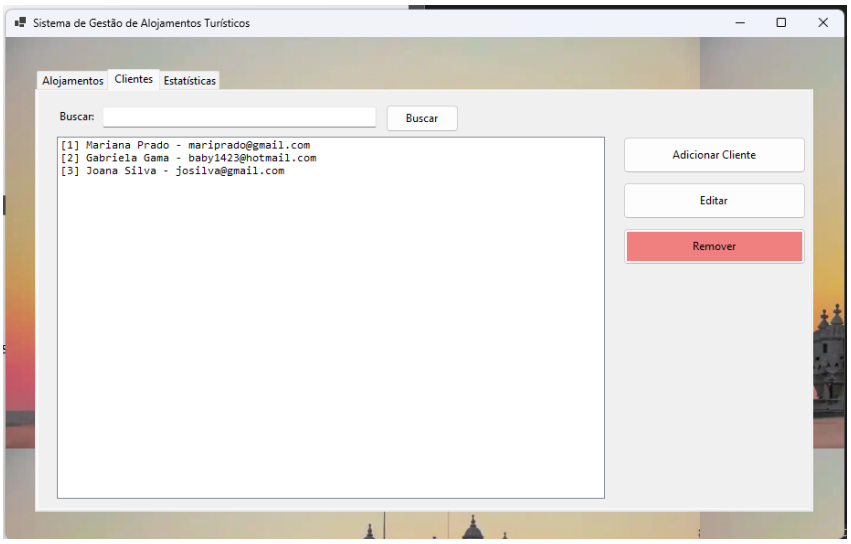


Figura 23 – MainForm tabela 02 – Clientes

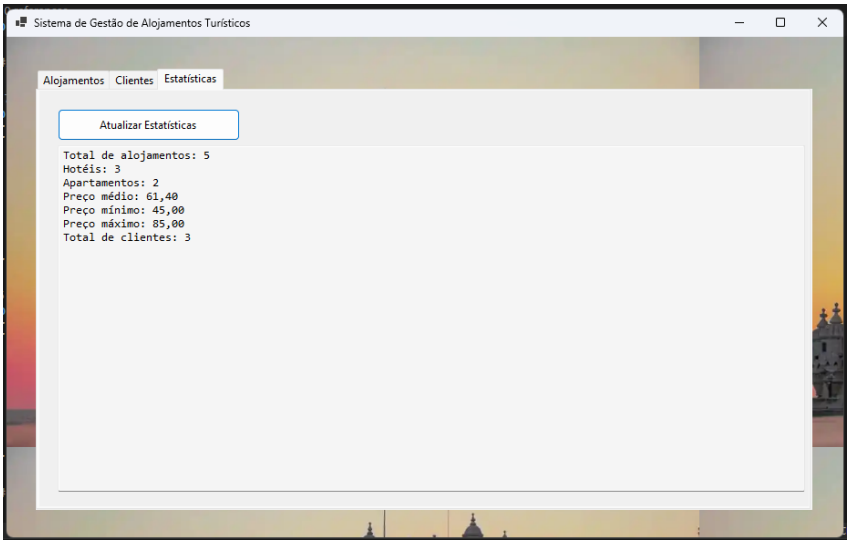


Figura 24 – MainForm tabela 03 - Estatísticas

Após isso existem outros 4 Forms Auxiliares disponíveis para o usuário.

- FormAdicionarHotel: Onde é possível adicionar um nome, endereço, preço por noite e estrelas, esses dados depois são utilizados para classificar o luxo do hotel e valor da taxa de serviço que foi definida como 10% do valor da diária.

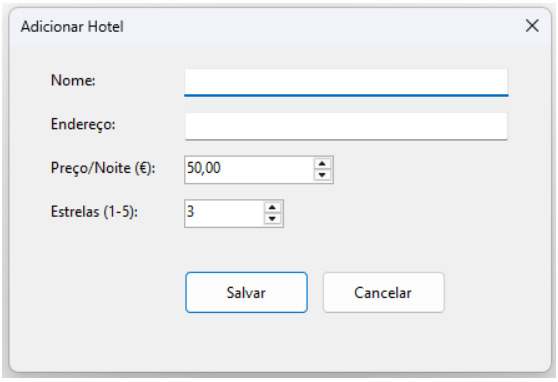


Figura 25 - FormAdicionarHotel

- FormAdicionarApartamento: O mesmo ocorre com o apartamento, mas com a diferença de que não existe atribuição de estrelas e existe a possibilidade de informar se tem garagem ou não (bool) e quantos quartos possui (T1, T2, T3…).

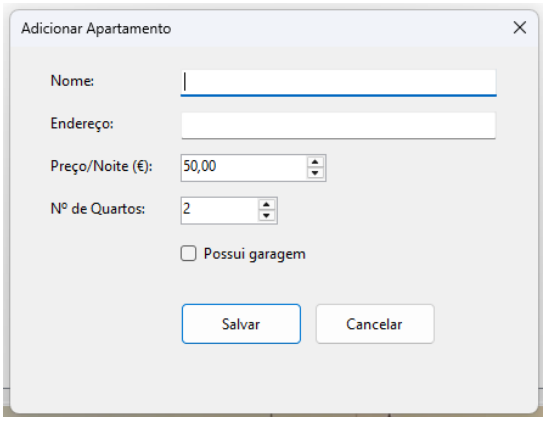


Figura 26 - FormAdicionarApartamento

- FormAdicionarCliente: Nessa versão inicial da solução, foi implementado apenas nome, e-mail e telefone, podendo futuramente ser implementado outros dados relativos ao cliente, como dados para pagamento das diárias.

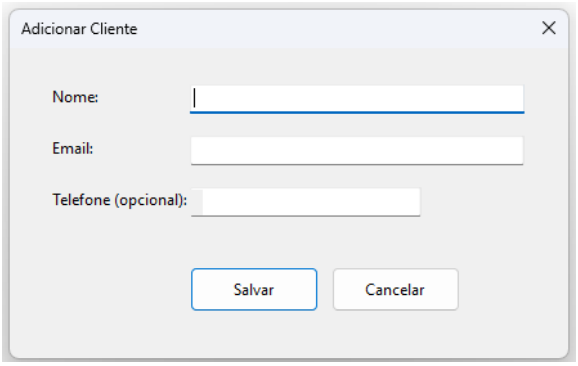


Figura 27 - FormAdicionarCliente

- FormEditarCliente: Ainda é possível editar os dados do cliente, sem perder a integridade da base de dados, uma vez que ele é identificado por sua “matricula” atribuída automaticamente, através do ID.

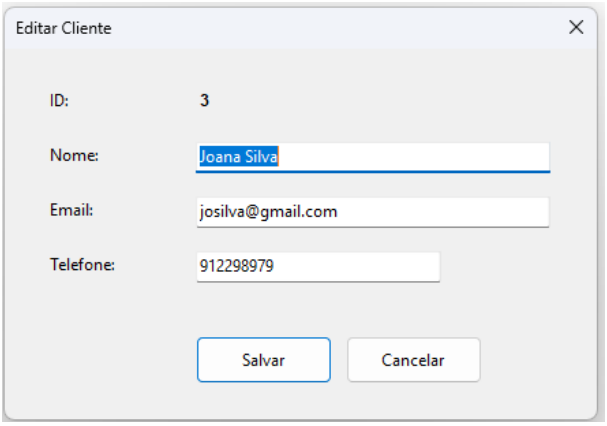


Figura 28 - FormEditarCliente

11 – Implementação do Login

11.1 Funcionalidades de Segurança

O sistema implementa uma tela de login simples mas funcional, com:

- Validação de credenciais
- Limitação de tentativas (3 máximo)
- Senha oculta visualmente
- Registro de tentativas no log
- Confirmação ao cancelar

Isso demonstra preocupação com segurança básica da aplicação, mesmo sendo um sistema educacional

11.2 Credenciais de Teste

Na tela de Login existem 2 campos que devem ser preenchidos, Usuário e Senha, foram definidos esses campos como credenciais de teste, inclusive para testar o Case-sensitive (diferenciação entre maiúsculas e minúsculas), conforme informações de campo e valor abaixo:

- Usuário: Ana
- Senha: AP1234

11.3 Como Testar

Execute a aplicação e digite o usuário e senha pré-definidos, carregue em “Entrar” ou pressione “Enter”, se as credenciais tiverem sido digitadas corretamente o sistema abre o MainForm, caso contrário diz que as credenciais estão erradas.

Figura 29 - Tela de Login após inserir credenciais certas ou erradas.

Testar Erro (o sistema informa quantas tentativas ainda possui):

- Digite usuário/senha errados
- Verá mensagem de erro
- Após 3 tentativas, aplicação fecha

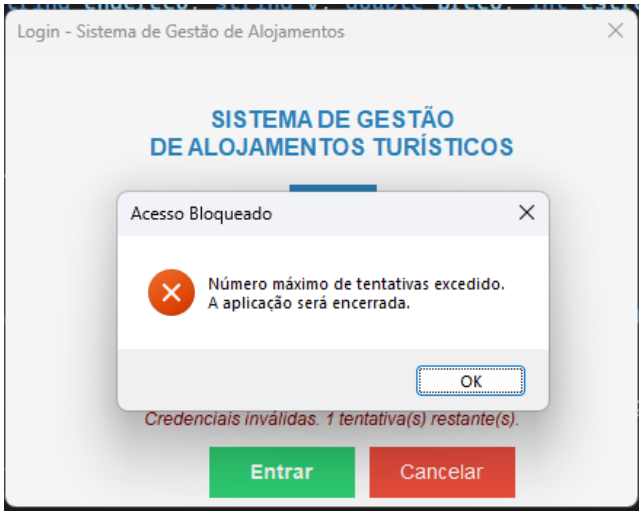


Figura 30 – Número máximo de tentativas

Conforme imagem acima, após o número máximo de tentativas ser excedido, o aviso aparece informando que a aplicação será encerrada.

Conclusão

O projeto atende integralmente aos requisitos propostos, implementando e demonstrando todos os pilares de POO, adotando uma arquitetura em camadas (N-Tier) bem definida, utilizando design patterns profissionais, LINQ e expressões lambda de forma extensiva, persistência em JSON, exceções customizadas organizadas em hierarquia, testes unitários com cobertura superior a 50%, interface gráfica funcional e intuitiva, sistema de logging robusto e código limpo, bem documentado e legível. Destaca-se pela organização exemplar, com estrutura clara, separação rigorosa de responsabilidades entre as camadas, nomenclatura descritiva, comentários XML, validações nos pontos críticos e alta extensibilidade, facilitando a adição de novos tipos e funcionalidades. Como principais aprendizagens, o desenvolvimento consolidou a aplicação prática dos conceitos de POO, a importância da arquitetura em camadas, o uso correto de design patterns, a adoção do desenvolvimento orientado a testes (TDD) e a internalização de boas práticas profissionais de engenharia de software.

Referências Bibliográficas

- Repositório do professor Ernesto Casanova em: https://github.com/IPCA-Content/POO-LESI-PL-202526/tree/main/Lessons
- Conteúdos sobre C# em Alura Cursos: https://www.alura.com.br/
- ChatGPT (Duvidas sobre como criar testes e para corrigir erros na construção dos mesmos) https://chatgpt.com/

Apêndice

Figura	Descrição
Figura 1	Estrutura POO_GestãoAlojamentosTuristicos
Figura 2	Estrutura POO_GestãoAlojamentosTuristicos.Tests
Figura 3	Exemplo de encapsulamento implementado na Classe Alojamento.
Figura 4	Trecho de código da “classe pai” Alojamento
Figura 5	Classe Apartamento herda de Alojamento
Figura 6	CalcularTaxaServiço
Figura 7	ObterDetalhes
Figura 8	Classe Hotel herda Alojamento
Figura 9	Public class Hotel usa classe base
Figura 10	Public class hotel usa CalcularTaxaServiço da classe base
Figura 11	Método ListarTodos
Figura 12	Ialojamento
Figura 13	CalcularTaxasServico classe abstrata implementa parcialmente
Figura 14	ObterDetalhes classe abstrata implementa parcialmente
Figura 15	Persistência – Dados guardados na Base de Dados em arquivos JSON
Figura 16	Business/Logger
Figura 17	Adicionar Hotel
Figura 18	Diagrama de hierarquia
Figura 19	Diagrama dos JSON na base de dados
Figura 20	Realização dos testes
Figura 21	Tela de Login
Figura 22	MainForm tabela 01 - Alojamentos
Figura 23	MainForm tabela 02 – Clientes
Figura 24	MainForm tabela 03 - Estatísticas
Figura 25	FormAdicionarHotel
Figura 26	FormAdicionarApartamento
Figura 27	FormAdicionarCliente
Figura 28	FormEditarCliente
Figura 29	Tela de Login após inserir credenciais certas ou erradas
Figura 30	Número máximo de tentativas