Instituto Politécnico do Cávado e do Ave

Escola Superior de Tecnologia

Programação Orientada a Objetos

Licenciatura em Engenharia de Sistemas Informáticos

Trabalho Prático

Gestão de Obras

Hugo Tiago Mendes Cruz – a23010

novembro de 2023

Resumo

Este projeto foi desenvolvido para atender as necessidades de gestão no setor de construção, focado na organização de 4 elementos fundamentais na construção civil, sendo esses os clientes, equipas de cada projeto, inventario e os diferentes projetos.

As principais funcionalidades incluem a criação e gestão de projetos onde é possível atribuindo equipas e clientes aos projetos, consumir material do inventario entre outros. Para desenvolver este sistema utilizei arrays, onde garantiu a eficiência do sistema.

Entre as melhorias futuras destacam-se: cada projeto ira ter uma lista de material consumido, remover funcionários do projeto e o cálculo automático do projeto. O sistema demostra se pratico atualmente, mas com estas implementações ficaria ainda mais robusto.

Índice

Índice de Figuras

1. Introdução

O presente trabalho surge no âmbito da cadeira de Programação Orientada a Objetos (POO), onde tem como objetivo consolidar, aplicar e expandir os conhecimentos adquiridos ao longo do semestre. Ao longo do desenvolvimento do projeto, procurou-se explorar de forma aprofundada os pilares fundamentais de POO, nomeadamente o encapsulamento, heranças e o polimorfismo, assegurando que o código desenvolvido segue boas práticas de programação, estas práticas incluem a utilização de estruturas de dados de forma eficiente.

O projeto de Gestão de Obras, foi idealizado com uma definição clara das responsabilidades das classes, e uma estrutura que permite a escalabilidade e manutenção do código. Por fim este trabalho não reflete só no conhecimento adquirido ao longo da cadeira, mas também a capacidade de transpor esse conhecimento para um contexto real de desenvolvimento, onde contribui para a formação de umas bases solidas na Programação Orientada a Objetos.

1.1 Problema

No setor de construção, a gestão de múltiplos projetos (Obras) apresenta desafios significativos em termos de organização e eficiência. Gerir dados de clientes, funcionários, materiais e a organização de projetos pode tornar-se complexo e sujeito a erros, espacialmente quando se trabalha em grandes volumes de informações. O projeto de Gestão de Obras foi desenvolvido para responder a essa necessidade, propondo uma solução que permite centralizar o registo, consulta e atualização de dados relacionados às obras.

O sistema organiza informações sobre clientes, funcionários, inventario de materiais, estados dos projetos entre outros, de maneira a tornar a gestão de uma obra mais eficiente. Esta centralização de dados permite evitar erros comuns, otimizar recursos e melhorar as decisões a tomar ao longo do processo de gerir diversas obras.

2. Arquitetura e Design

2.1 Arquitetura e Design

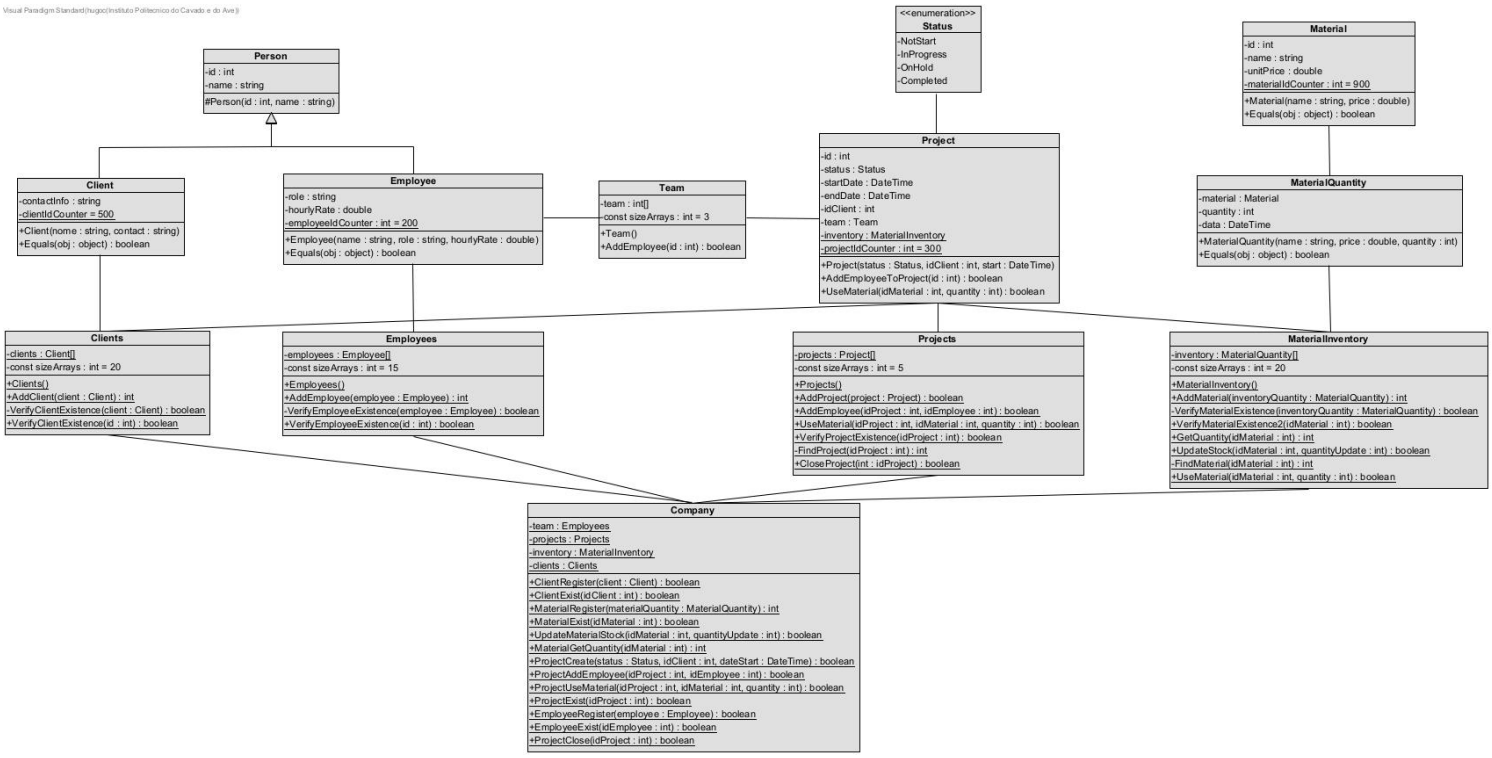


Figura 1 – Diagrama de classes

2.2 Estrutura e Principais Responsabilidades das Classes

2.2.1 Person

Classe que representa uma pessoa de forma simples, com atributos básicos como id e name. É a superclasse para **Client** e **Employee**, de forma a evitar duplicação de código.

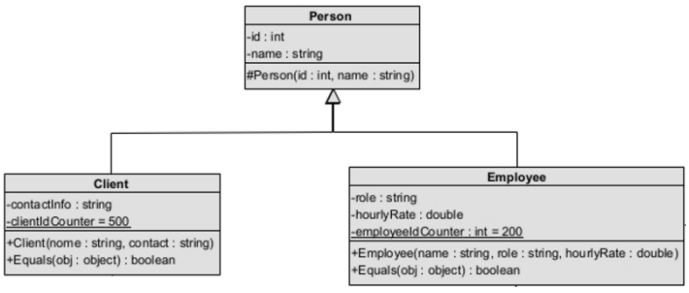


Figura 2 - Herança

2.2.2 Client

Representa um cliente da empresa incluindo informações de contacto e métodos para comparar clientes. Armazenado na classe **Clients**, que se trata se de uma "lista" **única** responsável por armazenar todos os clientes que a empresa contém. Para manipular esta lista a mesma contém alguns métodos como: Adicionar novos clientes verificar a existência de um cliente através do seu id. Numa futura fase tenho como objetivo implementar outros métodos de forma a evitando a falta de funcionalidades no sistema.

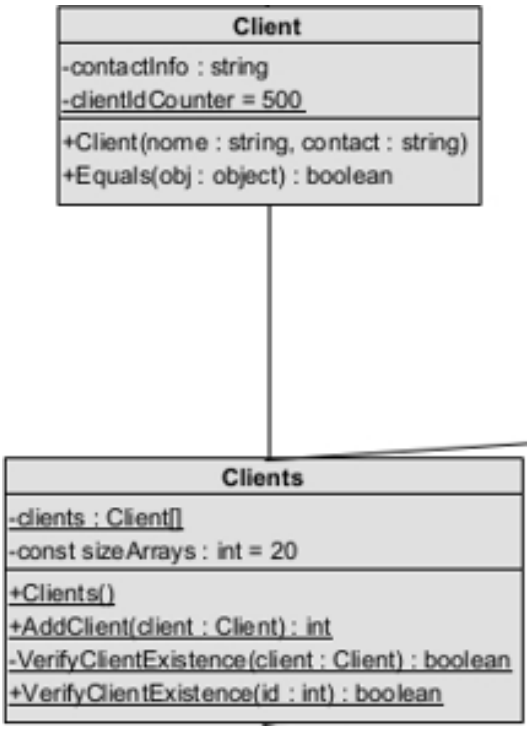


Figura 3 – Clients

2.2.3 Employee

Representa um funcionário da empresa incluindo informações de quanto ganha a hora e qual o seu cargo e métodos para comparar funcionarios. Armazenado na classe **Employees**, que se trata se de uma "lista" **única** responsável por armazenar todos os funcionários que a empresa contém. Para manipular esta lista a mesma contém alguns métodos como: Adicionar novos empregados verificar a existência de um funcionário através do seu id. Numa futura fase tenho como objetivo implementar outros métodos de forma a evitando a falta de funcionalidades no sistema.

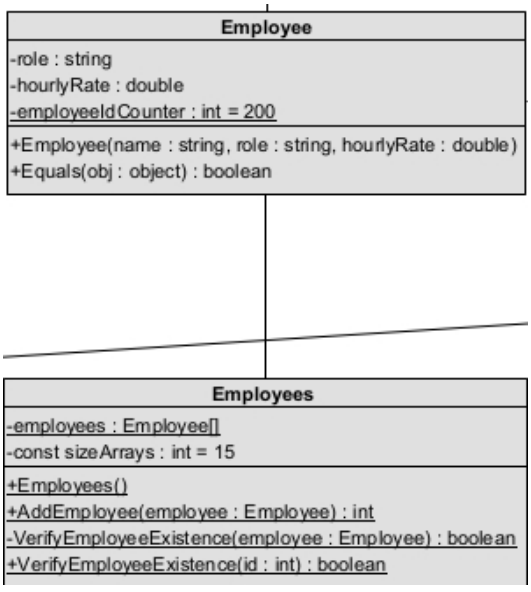


Figura 4 – Employees

2.2.4 Project

Representa uma obra com atributos como **status**, **startDate**, **endDate.** Cada obra apenas pode contém um cliente, que no nosso sistema é armazenado apenas o id do mesmo. Cada projeto ira conter uma equipa de 3 funcionários que é armazenado na classe **Team,** onde apenas serão armazenados o id de cada funcionário. Esta classe também contém uma ligação muito importante com o **MaterialInventory** será através disso possível gerir o stock de material. Toda esta informação irá ser armazenada numa lista única de **Projects.**

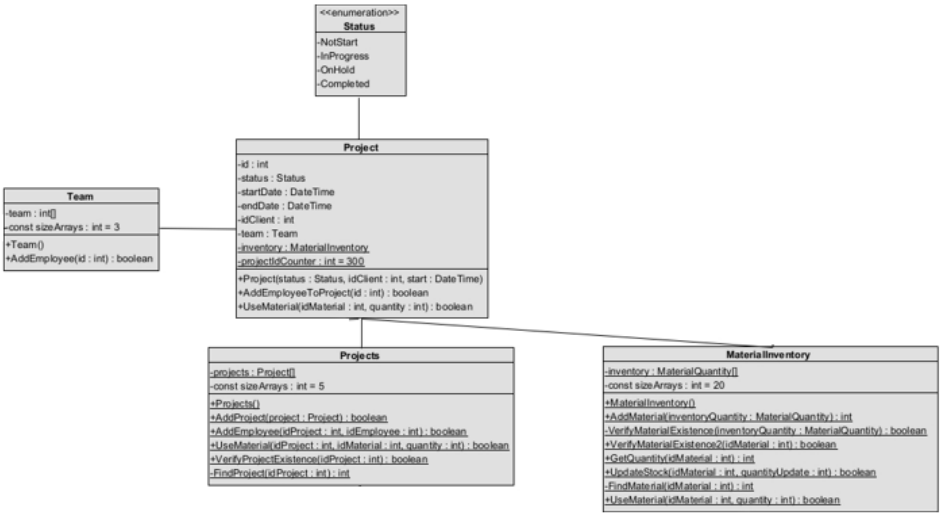


Figura 5 - Projects

2.2.5 Material

Representa um material necessário para os projetos, com atributos como **name** e **unitPrice**. Cada material é utilizado na classe **MaterialQuantity**, que associa o material à sua quantidade específica e à data de registo, permitindo um controlo detalhado do uso ao longo do tempo. A classe **MaterialInventory** centraliza a gestão do stock de materiais da empresa, funcionando como um repositório **único** de materiais, com quantidades e datas associadas. Esta classe é responsável por adicionar, verificar e atualizar o stock disponível, assegurando que a empresa possui os materiais necessários para os projetos em andamento, otimizando a organização e o controlo do inventário.

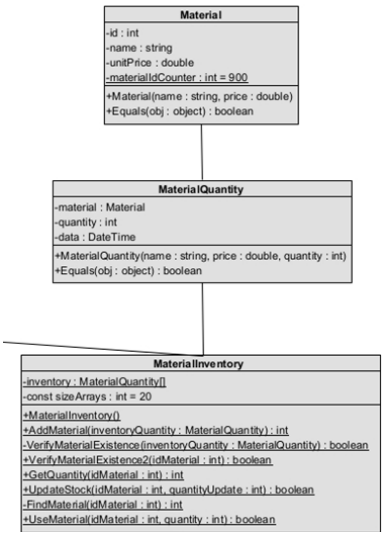


Figura 6 - Materials

2.2.6 Company

Classe central que coordena todos os elementos do sistema, com métodos para registar clientes, empregados, projetos e gerenciar o inventário de materiais. É a camada de alto nível que permite a execução de funcionalidades complexas.

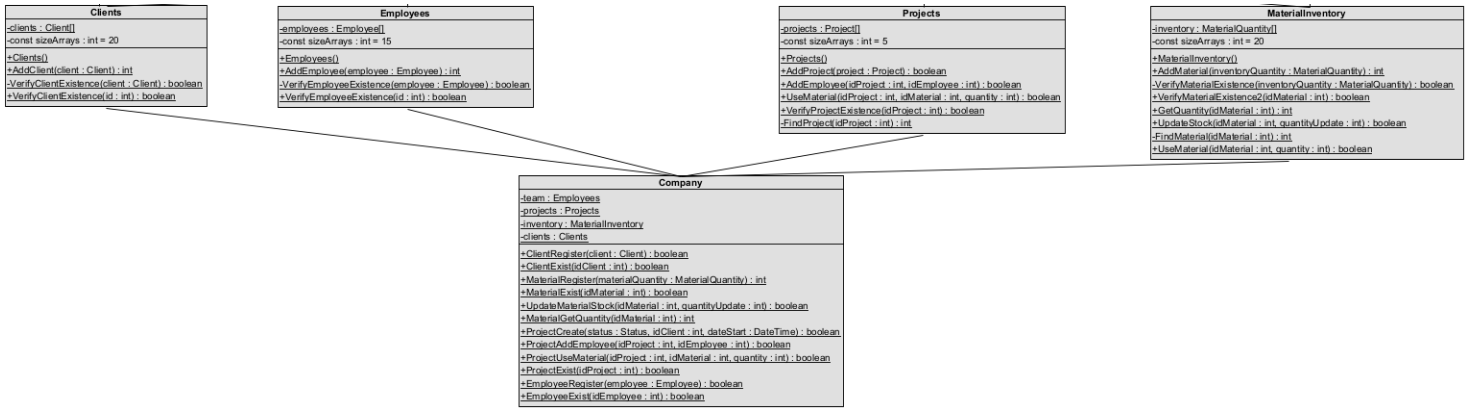


Figura 7 - Company

3. Implementação

3.1 Principais funcionalidades

É na criação de um projeto que tudo começa. Para criar um projeto é necessário existir um cliente na nossa lista única, caso não exista o projeto não é criado, ou temos de criar um cliente. Para isso utilizei este método:

```
public static int AddClient(Client client)
{
    if (!VerifyClientExistence(client))
    {
        for (int i = 0; i < clients.Length; i++)
        {
            while (clients[i] == null)
            {
                clients[i] = client;
                return clients[i].Id;
            }
        }
    }
    return -10;
}
```

Ira verificar se ele já existe para evitar duplicação de clientes, caso não exista corre casa a casa da nossa estruturas de dados até encontrar uma casa nula. Quando encontrar coloca o cliente na respetiva casa e retorna o id do mesmo.

Figura 8 – Adiciona um Cliente

Agora que temos o novo cliente pudemos criar o nosso projeto e inserir lo numa lista única de projetos, onde utilizei um método semelhante.

```
public Project(Status status, int idClient, DateTime start)
{
    Id = projectIdCounter++;
    Status = status;
    startDate = start;
    EndDate = new DateTime();
    IdClient = idClient;
    team = new Team();
}
```

Figura 10 - Cria um projeto

```
public static int AddProject(Project project)
{
    for (int i = 0; i < projects.Length; i++)
    {
        if (VerifyProjectExistence(project.Id))
        {
            while (projects[i] == null)
            {
                projects[i] = project;
                return projects[i].Id;
            }
        }
    }
    return -10;
}
```

Figura 9 – Adiciona Project

Visto que já temos o nosso projeto necessitamos de atribuir quais são os funcionários que iram trabalhar nesse projeto. Para isso utilizamos a seguinte funcionalidade:

```
public bool AddEmployee(int id)
{
    for (int i = 0; i < team.Length; i++)
    {
        while (team[i] == 0)
        {
            team[i] = id;
            return true;
        }
    }

    return false;
}
```

Figura 11 – Adiciona Funcionário a um projeto

Por fim conseguimos consumir o material necessário e concluir o projeto.

```
public static bool CloseProject(int idProject)
{
    if (VerifyProjectExistence(idProject))
    {
        int posicao = FindProject(idProject);
        projects[posicao].EndDate = DateTime.Now;
        projects[posicao].Status = Status.Completed;
        return true;
    }

    return false;
}
```

Figura 13 - Termina o projeto

```
public static bool UseMaterial(int idMaterial, int quantity)
{
    if (VerifyMaterialExistence(idMaterial))
    {
        int position = FindMaterial(idMaterial);

        if (inventory[position].Quantity >= quantity)
        {
            inventory[position].Date = DateTime.Now;
            inventory[position].Quantity -= quantity;
            return true;
        }
    }

    return false;
}
```

Figura 12 – Usa o material

3.2 Decisões técnicas

Nesta fase do trabalho, optei por utilizar arrays como estrutura de dados principal para armazenar e organizar informações sobre clientes, funcionários, projetos e materiais. A escolha dos arrays baseia-se na simplicidade e eficiência dessa estrutura, onde utilizei a abordagem sugerida em aula.

```
public class Project
{
    #region Attributes

    int id;
    Status status;
    DateTime startDate;
    DateTime endDate;
    int idClient;
    Team team;
    static MaterialInventory usedMaterials;
    static int projectIdCounter = 300;

    #endregion
```

Figura 15 – Classe Project

```
public class Projects
{
    #region Attributes

    const int sizeArrays = 5;
    static Project[] projects;

    #endregion
```

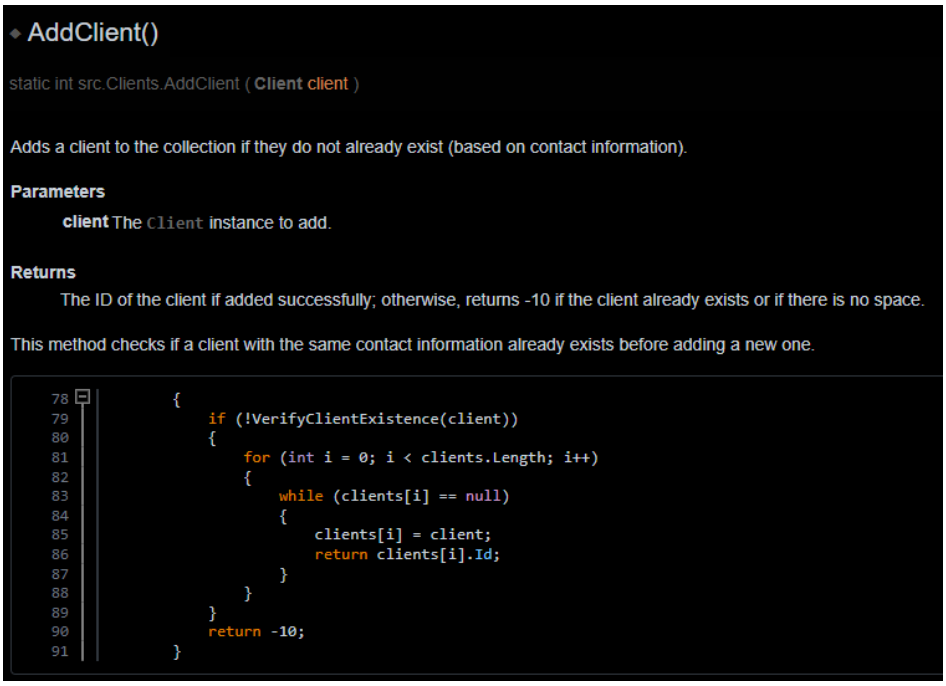
Figura 14 – Classe Projects

3.3 Futuras Implementações

- Cada projeto ira ter uma lista de material consumido.
- Cálculo Automático de Custos do Projeto
- Atualizar Informações do Clientes e Empregado
- Remover funcionários do projeto
- Listar Material consumido no projeto através do id
- Listar Projetos, Clientes e Funcionários

3.4 Código documentado XML

A documentação com XML envolve adicionar **comentários XML** diretamente nos métodos, classes e atributos do código, descrevendo o que cada parte do código faz, quais são os parâmetros, os valores de retorno e outros detalhes importantes. Isso é especialmente útil, pois facilita a compreensão do código.



AddClient()

static int src.Clients.AddClient (**Client client**)

Adds a client to the collection if they do not already exist (based on contact information).

Parameters

client The `Client` instance to add.

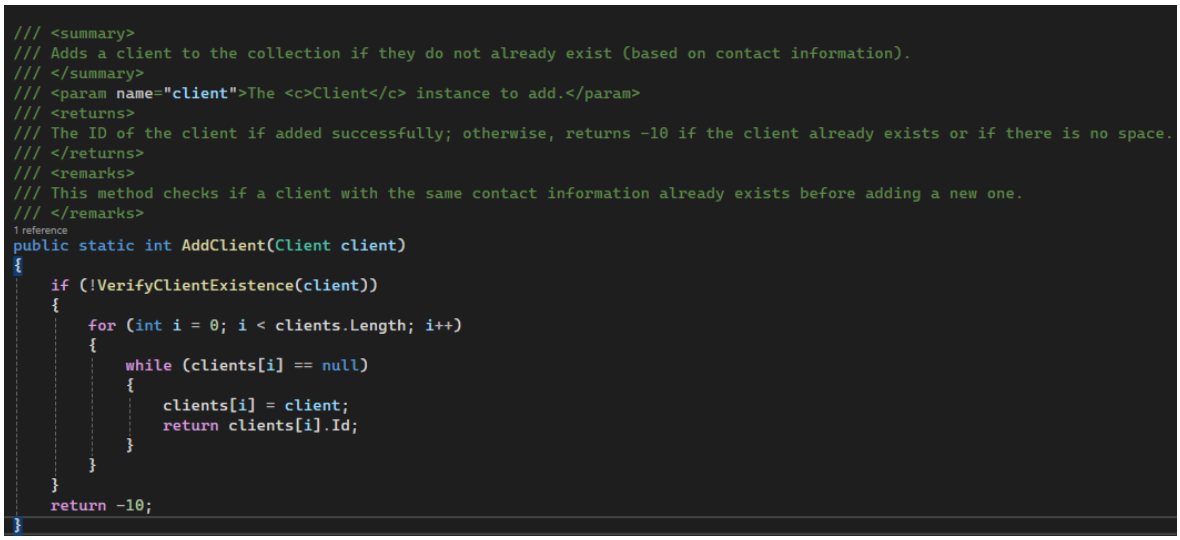
Returns

The ID of the client if added successfully; otherwise, returns -10 if the client already exists or if there is no space.

This method checks if a client with the same contact information already exists before adding a new one.

```
78      {
79          if (!VerifyClientExistence(client))
80          {
81              for (int i = 0; i < clients.Length; i++)
82              {
83                  while (clients[i] == null)
84                  {
85                      clients[i] = client;
86                      return clients[i].Id;
87                  }
88              }
89          }
90          return -10;
91      }
```

Figura 16 – Exemplo XML



```
/// <summary>
/// Adds a client to the collection if they do not already exist (based on contact information).
/// </summary>
/// <param name="client">The <c>Client</c> instance to add.</param>
/// <returns>
/// The ID of the client if added successfully; otherwise, returns -10 if the client already exists or if there is no space.
/// </returns>
/// <remarks>
/// This method checks if a client with the same contact information already exists before adding a new one.
/// </remarks>
1 reference
public static int AddClient(Client client)
{
    if (!VerifyClientExistence(client))
    {
        for (int i = 0; i < clients.Length; i++)
        {
            while (clients[i] == null)
            {
                clients[i] = client;
                return clients[i].Id;
            }
        }
    }
    return -10;
}
```

Figura 17- Exemplo XML

4. Princípios OOP aplicados

```
public string ContactInfo
{
    set
    {
        if (value.Length >= 9)
        {
            contactInfo = value;
        }
    }
    get { return contactInfo; }
}
```

Figura 19 - Encapsulamento

```
public class Client : Person
{
```

Figura 18- Herança

```
private static bool VerifyEmployeeExistence(Employee employee)
{
    for (int i = 0; i < employees.Length; i++)
    {
        if (employee.Equals(employees[i]))
        {
            return true;
        }
    }
    return false;
}


2 references
public static bool VerifyEmployeeExistence(int idEmployee)
{
    for (int i = 0; i < employees.Length; i++)
    {

        if (employees[i] == null)
        {
            break;
        }

        if (employees[i].Id == idEmployee)
        {
            return true;
        }
    }
    return false;
}
```

Figura 20- Polimorfismo

5. Conclusão

O desenvolvimento do projeto de Gestão de Obras proporcionou aprofundar os conceitos de Programação Orientada a Objetos (POO). Foi possível explorar pilares fundamentais como encapsulamento, herança e polimorfismo, onde foram aplicados de maneira eficaz para criar um sistema estruturado, escalável e eficiente. O foco nas boas práticas de programação e na utilização de estruturas de dados adequadas assegurou que o projeto se mantivesse organizado e funcional.

O sistema desenvolvido atende a necessidade do setor de construções focado na organização de múltiplos projetos. A inclusão de funcionalidades adicionais como cálculo automático de custos, listagem detalhada de materiais e a melhoria das operações de atualização e remover iram garantir que o sistema fica ainda mais completo, conseguindo satisfazer a necessidades de algumas construtoras.

Em resumo, este projeto ajudou-me a compreender alguns conceitos fundamentais na programação, mas também permitiu aplicá-los em contexto pratico. Além de consolidar o conhecimento adquirido, o projeto abriu portas para utilizá-lo em cenários futuros, demonstrando o quanto é importante para o desenvolvimento de aplicações mais robustas.

6. Referências

https://www.youtube.com/watch?v=6ac6Hdn4-p0

https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/base

https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/inheritance

https://forums.visual-paradigm.com/t/static-method/10327

https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/oop

https://www.w3schools.com/cs/cs_oop.php

https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/