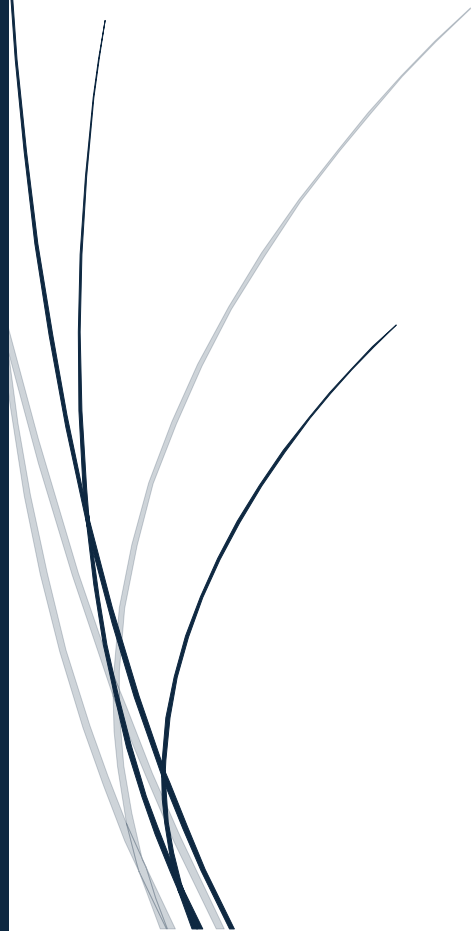
ESCOLA SUPERIOR DE TECNOLOGIA IPCA

19-12-2024

# Trabalho Prático

## Programação Orientada a Objetos



José Miguel Oliveira Pires - 27968

# Índice

# Introdução

## Enquadramento

Este projeto de avaliação faz parte da unidade curricular "Programação Orientada a Objetos" (POO), que se integra no 1º semestre do 2º ano do curso de Engenharia Informática do Instituto Politécnico do Cávado e do Ave (IPCA), em Barcelos.

## Tema Escolhido

Comércio eletrónico: sistema que permita a gestão de uma loja online.

**Palavras-chave**: produtos, categorias, garantias, stocks, clientes, campanhas, vendas, marcas.

O tema escolhido por mim foi a loja Eletrónica em Portugal, pois é algo bastante próximo há minha realidade. É um tema que domino e conheço, o que facilita a minha compreensão para uma implementação mais assertiva do projeto a desenvolver.

## Enunciado do trabalho

Trabalho_POO_ESI_2024_2025.pdf

## Objetivos

- Consolidar conceitos basilares do Paradigma Orientado a Objetos;
- Analisar problemas reais;
- Desenvolver capacidades de programação em C#;
- Potenciar a experiência no desenvolvimento de software;
- Assimilar o conteúdo da Unidade Curricular.

## Problemas a resolver

De acordo com o enunciado, este projeto de avaliação em POO, consiste no desenvolvimento de duas fases:

### Fase 1: Realizar até (15-11-2025)

- Estrutura de Classes identificadas
- Implementação essencial das classes
- Estruturas de dados a utilizar
- Cumprimento dos prazos

### Fase 2: Realizar até (19-12-2025)

- Implementação final das classes e serviços
- Aplicação demonstradora dos serviços implementados
- Relatório final do trabalho realizado
- Cumprimento dos prazos

## Pilares de POO

Os pilares da Programação Orientada a Objetos (POO) são os fundamentos que estruturam esta abordagem de programação. Estes princípios permitem organizar o código de forma modular, eficiente e reutilizável, promovendo soluções mais robustas e fáceis de manter.

Os quatro pilares principais são: abstração, encapsulamento, herança e polimorfismo.

**Abstração**

- A abstração consiste em representar apenas os aspetos mais relevantes de um objeto do mundo real, ignorando detalhes desnecessários. Desta forma, uma classe contém apenas os atributos e métodos essenciais para o problema que se pretende resolver.

**Encapsulamento**

- O encapsulamento consiste em proteger os dados de um objeto, restringindo o acesso direto aos seus atributos e métodos. Os dados são acedidos e modificados de forma controlada, geralmente através de métodos ou propriedades.
- **Vantagens**: Proteção dos dados, controlo de acesso e facilidade de manutenção.

**Herança**

- A herança permite que uma classe herde atributos e métodos de outra classe, chamada classe base. Assim, a classe derivada pode reutilizar código e adicionar ou modificar comportamentos.
- **Vantagens**: Reutilização de código e organização hierárquica.

**Polimorfismo**

- O polimorfismo permite que objetos de diferentes classes derivadas sejam tratados de forma uniforme, mas executem comportamentos específicos, dependendo da classe a que pertencem. Isto promove flexibilidade no código.

# SOLID

Solid é um conjunto de cinco princípios fundamentais para a Programação Orientada a Objetos, sendo eles:

**S** - Single Responsibility Principle (SRP) - Princípio da Responsabilidade Única

**O** - Open/Closed Principle (OCP) - Princípio Aberto/Fechado

**L** - Liskov Substitution Principle (LSP) - Princípio da Substituição de Liskov

**I** - Interface Segregation Principle (ISP) - Princípio da Segregação de Interfaces

**D** - Dependency Inversion Principle (DIP) - Princípio da Inversão de Dependência

Todos eles são essenciais para criar um software mais robusto, testável e de fácil manutenção. Aplica-los não é obrigatório, mas são uma mais-valia. Durante o meu projeto tentei aplicar alguns deles, sendo eles o Single Responsibility Principle e o Dependency Inversion Principle.

## Single Responsibility Principle

Em todo o meu projeto tentei respeitar ao máximo o princípio da Responsabilidade Única, que se reflete no facto de cada ação dentro do meu projeto seja única e realize uma única tarefa.

## Dependency Inversion Principle

Apliquei também o Princípio da Inversão de Dependência que significa que durante a realização de trabalho nenhuma classe ficou dependente de outra. Nenhuma Implementação, depende de uma segunda.

# Desenvolvimento

## Estrutura de Classes identificadas

Neste projeto utilizei como classes Categoria, Produto, Categoria, Cliente, Compra e Campanha.

## Objetos de Negócios

### Classe Produto

A classe Produto representa os produtos disponíveis para compra e armazena as suas informações principais, como nome, preço e quantidade.

**Principais Funcionalidades:**

- **Construtores**: Criam instâncias de produtos com os atributos necessários, como nome, telemóvel, morada, contribuinte e idCliente.
- **Método CriaProduto**: Método estático que permite criar e retornar instâncias de produtos de forma padronizada.

### Classe Categoria

A classe Categoria organiza os produtos em categorias específicas, como Telemóveis, Portáteis ou Monitores.
Permite identificar e gerir as categorias existentes no sistema.

**Principais Funcionalidades:**

- **Construtores**: Criam instâncias de categorias e adicionam-nas a uma lista global de categorias.
- **Método CriaCategoria**: Método estático que permite criar e retornar instâncias de categorias de forma padronizada.

## Classe Cliente

A classe Cliente representa os clientes do sistema, armazenando as suas informações pessoais e identificadores únicos.

**Principais Funcionalidades:**

- **Construtores:** Criam instâncias de clientes com atributos como nome, telemóvel, morada, contribuinte e idCliente.
- **CriaCliente:** Método estático que permite criar e retornar instâncias de cliente de forma padronizada.

## Classe Compra

A classe Compra regista as transações feitas pelos clientes, permitindo armazenar os produtos adquiridos e associá-los ao cliente que realizou a compra.

**Principais Funcionalidades:**

- **Construtores**: Criam compras, associando-as a um cliente através do idCliente e registando os produtos adquiridos.
- **CriaCompra**: Cria a compra e permite adicionar produtos a uma compra específica.
- **ValorPagar**: Calcula o valor a pagar pelo carrinho do cliente

## Classe Campanha

A classe Campanha representa campanhas promocionais que podem incluir descontos ou produtos gratuitos, com critérios baseados em datas, valores de compras, e condições específicas.

**Principais Funcionalidades:**

- **Construtores**: Permitem criar uma campanha com nome, data inicial e final, e descrição, gerando automaticamente um identificador único.
- **Criação de Campanhas**: Método estático CriaCampanha que inicializa uma nova campanha e valida os dados fornecidos.
- **Redefinir Datas**: O método RedefenirDataCampanha permite atualizar as datas de início e fim de uma campanha, verificando se a campanha existe e se os valores são válidos.
- **Descontos para Compras Superiores**: O método CompraSuperior aplica um desconto para compras com valor superior a 1500, respeitando o período de validade da campanha.
- **Produto Gratuito em Compras Qualificadas**: O método OfereceProduto verifica se uma compra cumpre as condições para receber um produto gratuito. As condições incluem comprar três ou mais produtos, sendo pelo menos dois deles com preço superior a 600.

## Dados

### Classe Produtos

A classe **Produtos** gere uma lista estática de produtos, permitindo adicionar, remover e consultar produtos com base no seu ID.

**Principais Funcionalidades:**

- **Gestão de Produtos:**
  - Métodos para adicionar um produto à lista (AdicionarProduto).
  - Remover um produto pelo ID (RemoverProduto).
- **Consultas:**
  - Verifica se um produto existe através do ID (ExisteProduto).
  - Pesquisa um produto na lista através do ID (EncontrarProduto).
- **Acesso:**
  - Propriedade estática TodosProduto que expõe a lista de produtos.

## Classe Categorias

A classe Categorias organiza uma lista estática de categorias, permitindo adicionar, remover e consultar categorias com base no nome.

**Principais Funcionalidades:**

- **Gestão de Categorias:**
  - Adicionar uma Categoria à lista (AdicionarCategoria).
  - Remover uma Categoria pelo ID (RemoverCategoria).
- **Consultas:**
  - Verifica se uma categoria existe através do nome (ExisteCategoria).
  - Pesquisa uma categoria na lista através do nome (EncontrarCategoria).
- **Gestão de Stock:**
  - Método Stock que calcula a soma das quantidades de todas as categorias.
- **Acesso:**
  - Propriedade estática TodasCategoria que expõe a lista de categorias.

## Classe Clientes

A classe Clientes gere uma lista estática de clientes, permitindo adicionar, remover, atualizar e consultar clientes com base no seu ID.

**Principais Funcionalidades:**

- **Gestão de Clientes:**
  - Adicionar um cliente à lista (AdicionarCliente).
  - Reemover um cliente pelo ID (RemoverCliente).
  - Atualizar um Cliente pelo ID (AtualizarCliente).
- **Consultas:**
  - Verifica se um cliente existe através do ID (ExisteCliente).
  - Pesquisa um cliente na lista através do ID (EncontrarCliente).
- **Acesso:**
  - Propriedade estática TodosCliente que expõe a lista de clientes.

## Classe Compras

A classe Compras é responsável por gerir uma lista estática de compras, permitindo adicionar, remover, verificar a existência e localizar compras com base no ID.

**Principais Funcionalidades:**

- **Gestão de Compras:**
  - Adicionar uma compra à lista (AdicionarCompra).
  - Remover uma compra pelo ID (RemoverCompra).
- **Consultas:**
  - Verificar se uma compra existe pelo ID (ExisteCompra).
  - Localizar uma compra pelo ID (EncontrarCompra).
- **Acesso:**
  - Propriedade TodasCompra que permite aceder à lista de todas as compras.

## Classe Carrinhos

A classe **Carrinhos** é responsável por gerir os carrinhos de compras de clientes, utilizando um dicionário estático que associa IDs de clientes a listas de produtos.

**Principais Funcionalidades:**

- **Gestão de Carrinhos:**
  - Criar um novo carrinho para um cliente (CriaCarrinho).
  - Adicionar produtos a um carrinho existente (InsereDadosNoCarrinho).
  - Modificar os produtos de um carrinho (ModificaCarrinho).
- **Manipulação de Produtos:**
  - Adicionar uma unidade de um produto a um carrinho (AdicionaUnidade).
  - Remover uma unidade de um produto de um carrinho (RemoveUnidade).
- **Consultas:**
  - Verificar se um carrinho existe (ExisteCarrinho).
  - Encontrar os produtos associados a um carrinho (EncontraProdutosCompra).
- **Acesso:**
  - Propriedade CompraProdutos para aceder ao dicionário de carrinhos.

## Classe Campanhas

A classe Campanhas é responsável por gerir uma lista estática de campanhas, permitindo adicionar, remover, verificar a existência e localizar campanhas com base no ID.

**Principais Funcionalidades:**

- **Gestão de Campanhas:**
  - Adicionar uma campanha à lista (AdicionarCampanha).
  - Remover uma campanha pelo ID (RemoverCampanha).
- **Consultas:**
  - Verificar se uma campanha existe pelo ID (ExisteCampanha).
  - Localizar uma campanha pelo ID (EncontrarCampanha).
- **Acesso:**
  - Propriedade TodasCampanha que permite aceder à lista de todas as campanhas.

## Regras de Negócios

### Classe RegrasProduto

A classe RegrasProduto define métodos para gerir a criação, adição e remoção de produtos.

**Funcionalidades principais:**

- **Criação de Produtos:**
  O método TentaCriarProduto valida os dados fornecidos e tenta criar uma instância de um produto. Devolve o produto criado ou null caso os dados sejam inválidos.
- **Gestão da Lista de Produtos:**
  Métodos como TentaInserirLista e TentaRemoverProduto permitem adicionar e remover produtos da lista, verificando a validade dos dados e tratando exceções em caso de erros.

### Classe RegrasCategoria

A classe RegrasCategoria gere operações relacionadas às categorias, incluindo criação, remoção e controlo de stock.

**Funcionalidades principais:**

- **Criação de Categorias:**
  O método TentarCriarCategoria verifica se o nome da categoria começa com uma letra maiúscula e tenta criar uma nova categoria.
- **Gestão da Lista de Categorias:**
  Métodos como TentaInserirLista e TentaRemoverCategoria permitem adicionar ou remover categorias da lista, garantindo a integridade dos dados.
- **Controlo de Stock:**
  O método PedeStock verifica o stock total das categorias e devolve um alerta (0) se o stock for inferior a 30 unidades.

## Classe RegrasProdutoCategoria

A classe RegrasProdutoCategoria trata da integração entre produtos e categorias, facilitando a associação automática durante a criação ou remoção de produtos.

**Funcionalidades principais:**

- **Criação e Associação Automática:**
  O método (TentarCriarProdutoCategoria) cria um produto e associa-o a uma categoria existente ou cria uma nova categoria, caso esta não exista.
- **Gestão das Listas:**
  O método (TentarRemoverProdutoCategoria) remove um produto e ajusta automaticamente a quantidade na categoria correspondente, garantindo consistência no stock.

## Classe RegrasCliente

A classe RegrasCliente define métodos para a gestão de clientes, como criação, atualização e remoção**.**

**Funcionalidades principais:**

- **Criação de Clientes:**
  O método TentaCriarInserirCliente valida os dados do cliente (por exemplo, número de telemóvel e contribuinte) e tenta criar um cliente, adicionando-o à lista estática.
- **Remoção de Clientes:**
  O método TentaRemoverLista verifica se o cliente existe na lista antes de removê-lo, garantindo a integridade dos dados.
- **Atualização de Dados:**
  O método TentaAtualizarCliente permite modificar informações específicas de um cliente existente.

## Classe RegrasCompra

A classe **RegrasCompra** lida com a execução de compras e a gestão de carrinhos associados.

**Funcionalidades principais:**

- **Execução de Compras:**
  O método TentarExecutarCompra permite criar uma compra associada a um cliente, gerar um carrinho e adicionar produtos.
- **Modificação de Carrinhos:**
  Métodos como TentarModificarTodoCarrinho, TentarAdicionarProduto, e TentarRemoverProduto gerem o conteúdo dos carrinhos de compras, garantindo operações seguras e consistentes.

## Classe RegrasCampanha

A classe **RegrasCampanha** foca na criação e gestão de campanhas promocionais.

**Funcionalidades principais:**

- **Gestão de Campanhas:**
  O método TentaCriarInserirCampanha permite criar uma nova campanha e adicioná-la à lista de campanhas existentes. Também inclui métodos para remover campanhas ou alterar datas.
- **Aplicação de Campanha:**
  Métodos como TentaAplicarOferta e TentaAplicarDesconto permitem aplicar ofertas ou descontos em compras associadas a campanhas específicas, influenciando diretamente os valores finais da compra.

# Exemplo de Métodos

## Classe Produto, Produtos, Regras ProdutoCategoria

```
public static Produto CriaProduto(string n, string marca, string mod, string cat, int quant, double preco)
{
    try
    {
        Produto produto = new Produto(n, marca, mod, cat, quant, preco);
        return produto;
    }
    catch (ArgumentException)
    {
        throw new ArgumentException();
    }
    catch (Exception)
    {
        throw new Exception();
    }
}
```

*Ilustração 1- Criar Produto, Classe Produto*

```
public static bool AdicionarProduto(Produto produto)
{
    if (produto == null) return false;
    try
    {
        todosProduto.Add(produto);
        return true;
    }
    catch (Exception)
    {
        throw new Exception();
    }
}
```

*Ilustração 2 - Adicionar Produto, Classe Produtos*

```
public static bool TentarCriarProdutoCategoria(string nome, string marca, string modelo, string cat, int quant, double preco)
{
    try
    {
        Produto p = RegrasProduto.TentaCriarProduto(nome, marca, modelo, cat, quant, preco);
        RegrasProduto.TentaInserirLista(p);
    }
    catch(ArgumentException e)
    {
        throw e;
    }
    catch (Exception)
    {
        throw new Exception();
    }

    if (Categorias.ExisteCategoria(cat))
    {
        Categoria categoria = Categorias.EncontrarCategoria(cat);
        categoria.Quantidade = +quant;
    }
    else if (char.IsUpper(cat[0]))
    {
        try
        {
            Categoria categoria = RegrasCategoria.TentarCriarCategoria(cat, quant);
            RegrasCategoria.TentaInserirLista(categoria);
            return true;
        }
        catch (ArgumentException )
        {
            throw new ArgumentException();
        }
        catch(Exception)
        {
            throw new Exception();
        }
    }
```

*Ilustração 3 - Tenta Criar ProdutoCategoria, Classe ProdutoCategoria*

## Classe Categoria, Categorias, Regras ProdutoCategoria

```
public static Categoria CriaCategoria(string n, int quant)
{
    try
    {
        Categoria categoria = new Categoria(n, quant);
        return categoria;
    }
    catch (ArgumentException e)
    {
        throw e;
    }
    catch(Exception)
    {
        throw new Exception();
    }
}
```

*Ilustração 4 - Cria Categoria, Classe Categoria*

```
public static bool RemoverCategoria(string nome)
{
    Categoria categoria = EncontrarCategoria(nome);

    if (categoria != null && nome != null)
    {
        try
        {
            todasCategoria.Remove(categoria);
            return true;
        }
        catch(Exception)
        {
            throw new Exception();
        }
    }
    return false;
}
```

*Ilustração 5 - RemoverCategoria, Classe Categorias*

```
/// <summary>
/// Tenta remover um produto e atualizar a quantidade da categoria associada.
/// </summary>
/// <param name="id">ID do produto a ser removido.</param>
/// <returns>True se o produto foi removido com sucesso; caso contrario, false.</returns>
/// <exception cref="Exception">Lança excecao para erros durante a remocao do produto ou atualizacao da categoria.</exception>
0 referências
public static bool TentarRemoverProdutoCategoria(int id)
{
    if (!Produtos.ExisteProduto(id)) return false;

    // Identifica qual o Produto e a Categoria associada
    Produto produto = Produtos.EncontrarProduto(id);
    Categoria categoriaExistente = Categorias.EncontrarCategoria(produto.CategoriaP);

    if (produto == null || categoriaExistente == null) return false;

    // Atualiza a quantidade em categoria e remove o Produto em questao
    categoriaExistente.Quantidade = -(produto.Quantidade);

    try
    {
        Produtos.RemoverProduto(produto.IdProduto);
        return true;
    }
    catch(Exception)
    {
        throw new Exception();
    }
}
```

*Ilustração 6 - TentaRemoverProdutoCategoria, Classe ProdutoCategoria*

**Classe Cliente, Clientes, Regras Cliente**

```
public static bool AtualizarCliente(int id, int op, string alteracao)
{
    Cliente cliente = EncontrarCliente(id);

    switch (op)
    {
        case 0:
            cliente.Nome = alteracao;
            break;

        case 1:
            cliente.Telemovel = alteracao;
            break;

        case 2:
            cliente.Morada = alteracao;
            break;

        case 3:
            if (int.TryParse(alteracao, out int contribuinte))
                cliente.Contribuinte = contribuinte;
            else
                return false; // Falha na conversão para Contribuinte
            break;

        default:
            return false;
    }
    return true;
}
```

*Ilustração 8 - AtualizarCliente, Classe Cliente*

```
public static Cliente CriaCliente(string nome, int telemovel, string morada, int contrib)
{
    try
    {
        Cliente novoCliente = new Cliente(nome, telemovel, morada, contrib);
        return novoCliente;
    }
    catch (ArgumentException)
    {
        throw new ArgumentException();
    }
    catch(Exception)
    {
        throw new Exception();
    }
}
```

*Ilustração 7 - CriaCliente, Classe Cliente*

## Classe Compra, Compras, Carrinhos, Regras Compra

```
public static Compra CriaCompra(int idCliente, bool contrib)
{
    try
    {
        if (contrib)
        {
            Compra novaCompraCli = new Compra(idCliente);
            if (novaCompraCli != null && novaCompraCli.IdCompra == idCliente)
                return novaCompraCli;
        }
        Compra novaCompra = new Compra();
        return novaCompra;
    }
    catch (ArgumentException)
    {
        throw new ArgumentException();
    }
    catch(Exception )
    {
        throw new Exception();
    }
}
```

*Ilustração 9 - CriaCompra, Classe Compra*

```
public static bool InsereDadosNoCarrinho(int id, List<int> idProdutos)
{
    if (!ExisteCarrinho(id))
        return false;
    try
    {
        compraProdutos[id].AddRange(idProdutos);

        if (compraProdutos[id].Count == 0)
            return false;

        return true;
    }
    catch (ArgumentException)
    {
        throw new ArgumentException();
    }
    catch (Exception)
    {
        throw new Exception();
    }
}
```

*Ilustração 10 - InsereDadosNoCarrinho, Classe Carrinhos*

## Classe Campanha, Campanhas, Regras Campanha

```
public static double TentaAplicarDesconto(int idCampanha, int Desconto, int idCompra)
{
    if (Campanhas.ExisteCampanha(idCampanha))
    {
        Campanha campanha = Campanhas.EncontrarCampanha(idCampanha);

        if (DateTime.Now < campanha.DataInicial || DateTime.Now > campanha.DataFinal)
            return RegrasCompra.ValorPagar(idCompra);

        double total = RegrasCompra.ValorPagar(idCompra);
        if (total < 1500) return RegrasCompra.ValorPagar(idCompra);

        total = RegrasCompra.ValorPagar(idCompra) - RegrasCompra.ValorPagar(idCompra) * Desconto / 100;
        return total;
    }
    return RegrasCompra.ValorPagar(idCompra);
}
```

*Ilustração 12 - TentaAplicarDesconto*

```
/// <summary>
/// Verifica se uma campanha existe na lista pelo ID
/// </summary>
/// <param name="id">ID da campanha</param>
/// <returns>True se a campanha existir, caso contrario False</returns>
3 referências
public static bool ExisteCampanha(int id)
{
    foreach (Campanha campanha in todasCampanha)
    {
        if (campanha.IdCampanha == id)
        {
            return true;
        }
    }
    return false;
}
```

*Ilustração 11 - ExisteCampanha, Classe Campanhas*

## Tratamento de Ficheiros

### Guardar

```
public static Registo GuardarDados()
{
    return new Registo
    {
        TodasCategoria = Categorias.TodasCategoria,
        TodosCliente = Clientes.TodosCliente,
        TodasCompra = Compras.TodasCompra,
        TodosProduto = Produtos.TodosProduto,
        TodasCampanha = Campanhas.TodasCampanha,
        Carrinho = Carrinhos.CompraProdutos //Dictionary
    };
}
```

*Ilustração 13 - GuardarDados, Classe Registo*

```
public static bool GuardarDados(string caminho, Registo dados)
{
    try
    {
        Stream stream = File.Open(caminho, FileMode.Create);
        BinaryFormatter bin = new BinaryFormatter();
        bin.Serialize(stream, dados);
        stream.Close();
        return true;
    }
    catch (ExcecaoFicheiroGuardar)
    {
        throw new ExcecaoFicheiroGuardar();
    }
    catch (Exception)
    {
        throw new Exception();
    }
}
```

*Ilustração 14 - GuardarDados, Classe TratamentoFicheiro*

## Carregar

```
public static bool AdicionarDados(Registo dados)
{
    if (dados != null)
    {
        Categorias.TodasCategoria.Clear();
        Categorias.TodasCategoria.AddRange(dados.TodasCategoria);

        Clientes.TodosCliente.Clear();
        Clientes.TodosCliente.AddRange(dados.TodosCliente);

        Compras.TodasCompra.Clear();
        Compras.TodasCompra.AddRange(dados.TodasCompra);

        Produtos.TodosProduto.Clear();
        Produtos.TodosProduto.AddRange(dados.TodosProduto);

        Campanhas.TodasCampanha.Clear();
        Campanhas.TodasCampanha = dados.TodasCampanha;

        //Dictionary
        Carrinhos.CompraProdutos.Clear();
        foreach (var item in dados.Carrinho)
        {
            Carrinhos.CompraProdutos[item.Key] = item.Value;
        }
        return true;
    }
    return false;
}
```

*Ilustração 15 - Adicionar Dados, Classe Registo*

```
public static Registo CarregarDados(string caminho)
{
    try
    {
        if (File.Exists(caminho))
        {
            Stream stream = File.Open(caminho, FileMode.Open);
            BinaryFormatter bin = new BinaryFormatter();
            return (Registo)bin.Deserialize(stream);
        }
        return null;
    }
    catch (ExcecaoFicheiroCarregar)
    {
        throw new ExcecaoFicheiroGuardar();
    }
    catch (Exception)
    {
        throw new Exception();
    }
}
```

*Ilustração 16 - CarregarDados, Classe TratamentoFicheiro*

## Implementações Futuras

Uma das implementações que ficou pendente neste projeto foi a criação de interfaces, que iria posteriormente fazê-lo com o método de comparação 'CompareTo'

No futuro, planejo adicionar essa funcionalidade, implementando a interface 'ICompara' nas classes necessárias. Isso permitirá uma maior flexibilidade no meu código, bem como um melhor controle sobre a comparação de objetos em diferentes cenários.

# Conclusão

Este projeto foi uma mais-valia, pois sinto que aprendi muito sobre Programação Orientada a Objetos (POO) e também sobre a linguagem C#. Ao longo do desenvolver do projeto, pude aplicar conceitos importantes, como a criação e manipulação de classes, o uso de propriedades e métodos, além da organização e estruturação do código de maneira eficiente e modular. Foi um projeto muito útil para aprimorar o meu método de trabalho, permitindo-me melhorar a forma como abordo problemas e soluções de programação.

Este projeto contribuiu significativamente para o meu crescimento técnico e para o meu entendimento prático da POO e do desenvolvimento em C#.