Laboratório :: Desenvolvimento em Camadas
Contas Pagar/Receber

Eduardo Cardoso[1]

[1]Sistemas Distribuidos – Instituto Federal Catarinense (IFC)

{edu.cardoso516@gmail.com

Abstract. *This article describes the development of a layered financial control system, using the C# language and Visual Studio Code. Classes are organized in layers to ensure a functional structure, facilitating code comprehension and maintenance. The system allows for efficient creation, manipulation, and persistence of accounts payable and receivable data, providing a comprehensive solution for personal or business financial management. Some difficulties were encountered in formatting the document using LaTeX, especially in paragraph structuring and image sizing.*

Resumo. *Este artigo descreve o desenvolvimento de um sistema de controle financeiro em camadas, utilizando a linguagem C# e o Visual Studio Code. As classes são organizadas em camadas para garantir uma estrutura funcional, facilitando a compreensão e manutenção do código. O sistema permite criar, manipular e persistir dados de contas a pagar e receber de forma eficiente, proporcionando uma solução abrangente para gestão financeira pessoal ou empresarial. Algumas dificuldades foram encontradas na formatação do documento utilizando LaTeX, especialmente na estruturação de parágrafos e no dimensionamento de imagens.*

1. Introdução

Escolhido tema contas a pagar e receber, para realizar o desenvolvimento foi escolhida a linguagem C# para programar foi utilizada a o Visual Studio Code como ambiente de desenvolvimento sobre o sistema Operacional POP-OS 22.04.

Durante o desenvolvimento foi utilizado as técnicas de programação orientada objetos. Essa abordagem de desenvolvimento em camadas e orientada a objetos não apenas automatiza processos com o tema definido, fornece uma base sólida para desenvolvimento como futuras inovações, ao decorrer desse relatório poderemos ver a explicação das classes e suas estruturas.

2. Estrutura e Diagrama de Classes

No sistema de contas a pagar e receber, as classes foram organizadas em camadas para garantir uma estrutura clara e funcional. Cada classe, como Conta, Parcela e Resumo-Financeiro, desempenha um papel específico para a regra de negócio. A disposição das classes foi pensada para agrupar funcionalidades relacionadas e manter uma organização lógica do código. Essa abordagem facilita a manutenção e compreensão do sistema.

O diagrama mostra como o sistema de controle financeiro esta organizado. No centro, temos a ideia de ”contas”, que representam transações de dinheiro. Cada conta

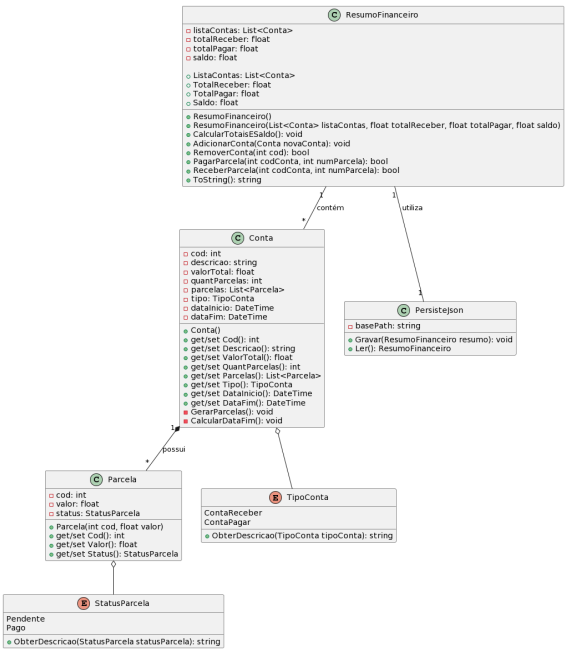


Figura 1. Diagrama de Classes

pode ser dividida em várias ”parcelas”, que são pedaços menores do dinheiro total. Por exemplo, se você tem uma conta de 1000 reais para pagar, pode dividi-la em 10 parcelas de 100 reais.

As cores e setas mostram como as diferentes partes se relacionam. Por exemplo, uma ”Conta”pode ter muitas ”Parcelas”, e cada ”Parcela”pode ter um ”Status”, como ”Pendente”(ainda precisa ser paga) ou ”Pago”(já foi paga). Essas relações nos ajudam a entender como o dinheiro está sendo usado e quais contas precisam de nossa atenção. Ao compreender o diagrama, podemos criar um sistema mais eficiente para lidar com nossas Contas.

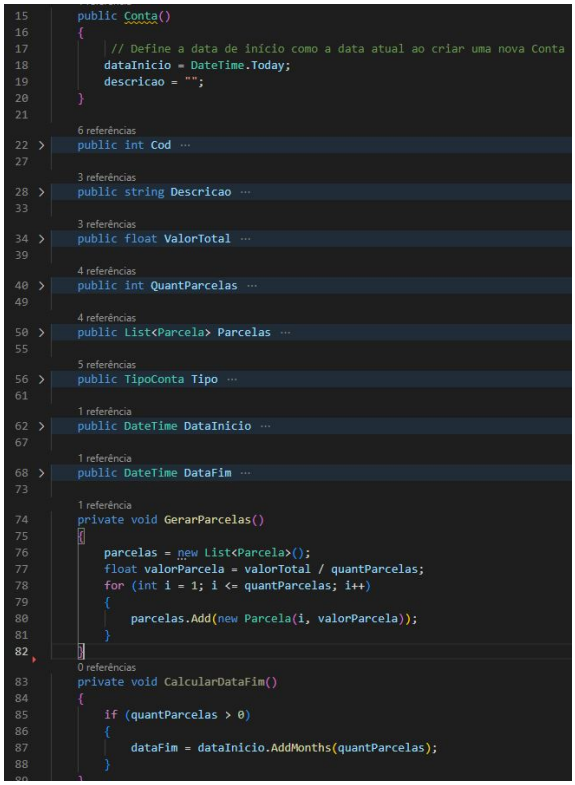
3. Código

Dentro do nosso projeto, foi criado classes para regra e funcionamento. A classe **Conta** é o ponto central, representando cada transação financeira e guardando informações como código, descrição, valor total e datas importantes. A **Parcela** entra em cena detalhando cada transação, com seu valor e se já foi paga ou não. As classes enum **StatusParcela** e **TipoConta**, que definem se a parcela está **pendente** ou **paga**, e se a conta é para **receber** ou **pagar**. O **ResumoFinanceiro** faz o papel de calculadora, somando o que a gente tem para receber, o que tem para pagar e o saldo total. E, para guardar esses dados, a classe **PersisteJson** cuida de gravar e ler tudo em arquivos no formato JSON. A seguir vamos comentar sobre as classes principais Conta, ResumoFinanceiro e PersisteJson.

3.1. Conta

A baixo podemos ver a classe conta com seus principais métodos em aberto.

Construtor Conta(): Este método é responsável por inicializar uma nova instância da classe Conta. Ele define a data de início como a data atual e a descrição como uma string vazia.



```
15      public Conta()
16      {
17          // Define a data de início como a data atual ao criar uma nova Conta
18          dataInicio = DateTime.Today;
19          descricao = "";
20      }
21
        6 referências
22  >   public int Cod ···
27
        3 referências
28  >   public string Descricao ···
33
        3 referências
34  >   public float ValorTotal ···
39
        4 referências
40  >   public int QuantParcelas ···
49
        4 referências
50  >   public List<Parcela> Parcelas ···
55
        5 referências
56  >   public TipoConta Tipo ···
61
        1 referência
62  >   public DateTime DataInicio ···
67
        1 referência
68  >   public DateTime DataFim ···
73
        1 referência
74      private void GerarParcelas()
75      {
76          parcelas = new List<Parcela>();
77          float valorParcela = valorTotal / quantParcelas;
78          for (int i = 1; i <= quantParcelas; i++)
79          {
80              parcelas.Add(new Parcela(i, valorParcela));
81          }
82      }
        0 referências
83      private void CalcularDataFim()
84      {
85          if (quantParcelas > 0)
86          {
87              dataFim = dataInicio.AddMonths(quantParcelas);
88          }
```

Figura 2. Classe Conta

Propriedade QuantParcelas: Essa propriedade representa a quantidade de parcelas que uma conta terá. Quando atribuído um valor, o método GerarParcelas() é chamado para criar automaticamente as parcelas com base nesse número.

Método GerarParcelas(): Este método é chamado automaticamente quando a quantidade de parcelas é definida. Ele cria as parcelas para a conta, dividindo o valor total igualmente entre elas.

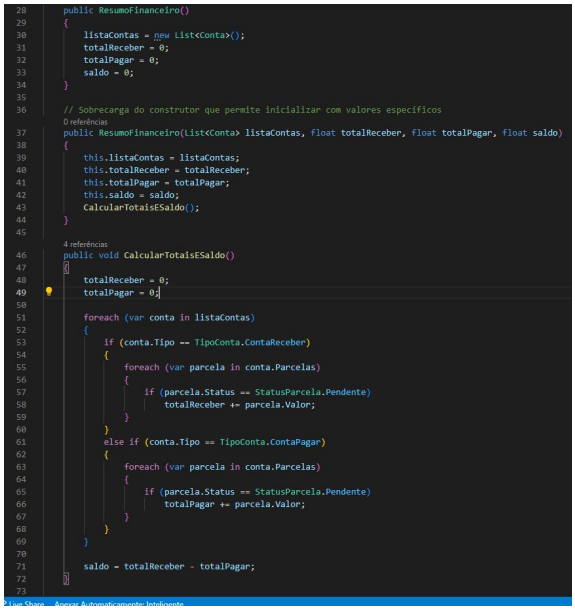
Método CalcularDataFim(): Este método é responsável por calcular a data de término da conta com base na data de início e na quantidade de parcelas. Se a quantidade de parcelas for maior que zero, ele adiciona o número de meses correspondente à quantidade de parcelas à data de início para obter a data de término.

3.2. Classe ResumoFinanceiro

A baixo podemos ver a classe ResumoFinanceiro com seus principais métodos em aberto.

Sobrecarga de construtor: A classe ResumoFinanceiro possui dois construtores. O primeiro é um construtor padrão que inicializa o resumo com valores padrão, enquanto o segundo permite a inicialização com valores específicos, como uma lista de contas, o total a receber, o total a pagar e o saldo. Ambos os construtores chamam o método CalcularTotaisESaldo() para garantir que os totais e o saldo sejam calculados corretamente.

Método CalcularTotaisESaldo(): Este método é responsável por calcular os totais a receber e a pagar, bem como o saldo total do resumo financeiro. Ele percorre todas as contas na lista e, para cada conta, verifica se é uma conta a receber ou a pagar. Em

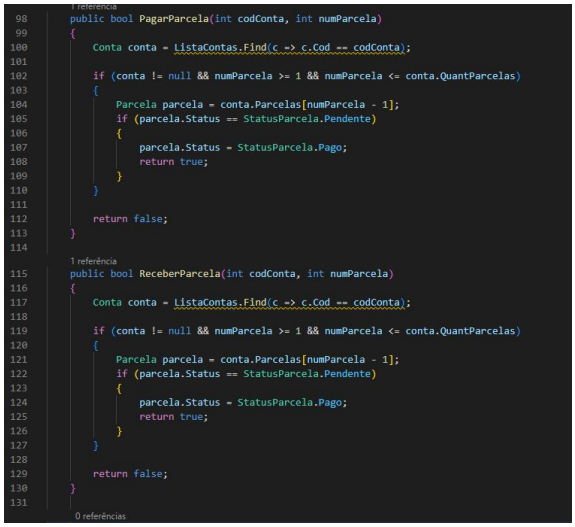


```
public ResumoFinanceiro()
{
    listaContas = new List<Conta>();
    totalReceber = 0;
    totalPagar = 0;
    saldo = 0;
}

// Sobrecarga do construtor que permite inicializar com valores específicos
0 referências
public ResumoFinanceiro(List<Conta> listaContas, float totalReceber, float totalPagar, float saldo)
{
    this.listaContas = listaContas;
    this.totalReceber = totalReceber;
    this.totalPagar = totalPagar;
    this.saldo = saldo;
    CalcularTotaisESaldo();
}

4 referências
public void CalcularTotaisESaldo()
{
    totalReceber = 0;
    totalPagar = 0;

    foreach (var conta in listaContas)
    {
        if (conta.Tipo == TipoConta.ContaReceber)
        {
            foreach (var parcela in conta.Parcelas)
            {
                if (parcela.Status == StatusParcela.Pendente)
                    totalReceber += parcela.Valor;
            }
        }
        else if (conta.Tipo == TipoConta.ContaPagar)
        {
            foreach (var parcela in conta.Parcelas)
            {
                if (parcela.Status == StatusParcela.Pendente)
                    totalPagar += parcela.Valor;
            }
        }
    }

    saldo = totalReceber - totalPagar;
}
```

Figura 3. Classe Resumo Financeiro Sobrecarga de Construtor e Cálculo

seguida, soma o valor das parcelas pendentes para calcular os totais a receber e a pagar. Por fim, subtrai o total a pagar do total a receber para obter o saldo final.

Método GerarParcelas(): Este método é chamado automaticamente quando a quantidade de parcelas é definida. Ele cria as parcelas para a conta, dividindo o valor total igualmente entre elas.



```
1 referência
public bool PagarParcela(int codConta, int numParcela)
{
    Conta conta = ListaContas.Find(c => c.Cod == codConta);

    if (conta != null && numParcela >= 1 && numParcela <= conta.QuantParcelas)
    {
        Parcela parcela = conta.Parcelas[numParcela - 1];
        if (parcela.Status == StatusParcela.Pendente)
        {
            parcela.Status = StatusParcela.Pago;
            return true;
        }
    }

    return false;
}

1 referência
public bool ReceberParcela(int codConta, int numParcela)
{
    Conta conta = ListaContas.Find(c => c.Cod == codConta);

    if (conta != null && numParcela >= 1 && numParcela <= conta.QuantParcelas)
    {
        Parcela parcela = conta.Parcelas[numParcela - 1];
        if (parcela.Status == StatusParcela.Pendente)
        {
            parcela.Status = StatusParcela.Pago;
            return true;
        }
    }

    return false;
}

0 referências
```

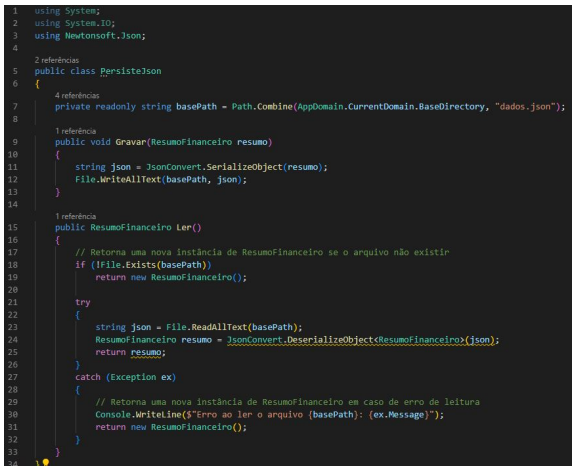
Figura 4. Classe Resumo Financeiro Pagar e Receber Parcela

Métodos PagarParcela e ReceberParcela: Esses métodos são responsáveis por marcar uma parcela como paga ou recebida. Eles recebem como parâmetros o código da conta e o número da parcela a ser marcada. Primeiro, eles procuram a conta correspondente na lista de contas. Se a conta existir e a parcela estiver dentro do intervalo válido, o status da parcela é atualizado para ”Pago”ou ”Recebido”, respectivamente. Esses métodos retornam true se a operação for bem-sucedida e false caso contrário.

Esses elementos da classe ResumoFinanceiro trabalham em conjunto para fornecer funcionalidades importantes de cálculo e gerenciamento das finanças, tornando o sistema mais completo e útil para o usuário.

3.3. PersisteJson

A baixo podemos ver a classe PersisteJson com seus principais métodos em aberto.



```
using System;
using System.IO;
using Newtonsoft.Json;

2 referências
public class PersisteJson
{
    4 referências
    private readonly string basePath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "dados.json");

    1 referência
    public void Gravar(ResumoFinanceiro resumo)
    {
        string json = JsonConvert.SerializeObject(resumo);
        File.WriteAllText(basePath, json);
    }

    1 referência
    public ResumoFinanceiro Ler()
    {
        // Retorna uma nova instância de ResumoFinanceiro se o arquivo não existir
        if (!File.Exists(basePath))
            return new ResumoFinanceiro();

        try
        {
            string json = File.ReadAllText(basePath);
            ResumoFinanceiro resumo = JsonConvert.DeserializeObject<ResumoFinanceiro>(json);
            return resumo;
        }
        catch (Exception ex)
        {
            // Retorna uma nova instância de ResumoFinanceiro em caso de erro de leitura
            Console.WriteLine($"Erro ao ler o arquivo {basePath}: {ex.Message}");
            return new ResumoFinanceiro();
        }
    }
}
```

Figura 5. Classe PersisteJson

private readonly string basePath: Esta é uma variável privada que armazena o caminho completo para o arquivo JSON onde os dados serão salvos. O caminho é formado pela combinação do diretório base do domínio de aplicativo (AppDomain.CurrentDomain.BaseDirectory) com o nome do arquivo, que é ”dados.json”.

Gravar(ResumoFinanceiro resumo): Este método é responsável por gravar um objeto ResumoFinanceiro em formato JSON no arquivo especificado. Ele primeiro converte o objeto para uma string JSON usando o método JsonConvert.SerializeObject(). Em seguida, escreve essa string no arquivo usando o método File.WriteAllText().

Ler(): Este método é responsável por ler o conteúdo do arquivo JSON e desserializá-lo de volta para um objeto ResumoFinanceiro. Primeiro, verifica se o arquivo existe usando File.Exists(). Se não existir, retorna uma nova instância de ResumoFinanceiro. Se o arquivo existir, lê o conteúdo do arquivo usando File.ReadAllText() e, em seguida, usa JsonConvert.DeserializeObject¡ResumoFinanceiro¿() para desserializar a string JSON de volta para um objeto ResumoFinanceiro.

Manuseio de exceções: Ambos os métodos Gravar e Ler incluem blocos try-catch para lidar com possíveis erros durante a escrita ou leitura do arquivo. Se ocorrer um erro, uma mensagem de erro é exibida no console e é retornado um novo ResumoFinanceiro, garantindo que o programa não falhe completamente em caso de problema de leitura ou gravação do arquivo.

Essa classe PersisteJson fornece uma maneira conveniente de salvar e carregar objetos ResumoFinanceiro em formato JSON, facilitando o armazenamento persistente dos dados do resumo financeiro em um arquivo.

4. Conclusão

Em suma, as classes desenvolvidas proporcionam uma estrutura sólida para o controle financeiro, permitindo a criação, manipulação e persistência de dados de forma eficiente. Com métodos intuitivos e funcionalidades bem definidas, o sistema oferece uma solução abrangente para a gestão financeira pessoal ou empresarial.

Eu tive algumas dificuldades com a linguagem de marcação LaTeX, especialmente ao lidar com a estruturação de parágrafos, correção de erros e dimensionamento de imagens. Foi necessário praticar a inserção de linhas em branco entre os parágrafos, revisar os logs de erros para fazer correções precisas.