UNIVERSIDADE DE VILA VELHA – UVV

Curso de Ciência da Computação

Disciplina: Programação Orientada a Objetos II

Entity Framework e os princípios SOLID

Giseli Rosa Lourenço
Guilherme Sousa Fagundes
João Pedro Valladares Araujo
João Cunha

Vila Velha – ES
Maio de 2025

30 de maio de 2025

Resumo

Este trabalho tem como objetivo explorar a aplicação prática dos frameworks Entity Framework e os princípios SOLID no desenvolvimento de aplicações em C#, utilizando o ambiente de desenvolvimento Visual Studio. A proposta é demonstrar como essas tecnologias e boas práticas permitem a construção de sistemas mais organizados, flexíveis e fáceis de manter.

Palavras-chave: *Entity Framework*, *ORM*, *migration*, *framework*

Conteúdo

1 Entity Framework

1.1 Introdução

O Entity Framework é uma ferramenta que ajuda os desenvolvedores .NET a trabalharem com bancos de dados de forma mais simples. Em vez de escrever muitos comandos SQL para acessar os dados, o desenvolvedor pode usar classes e objetos do próprio código para isso. Ou seja, ele permite usar o banco de dados como se fosse parte do programa, economizando tempo e reduzindo a chance de erros.

"O mapeamento objeto-relacional (ORM) do Entity Framework é um conjunto de tecnologias que permitem o desenvolvimento de aplicações de software orientadas a dados. Para isso, o framework permite modelar entidades, relacionamentos e lógica de negócios a partir do código que, quando migrado usando os métodos próprios da ferramenta, reflete-se no modelo de dados do repositório de dados configurado."(L. MARTÍNEZ, 2015)

"Com o Entity Framework, você primeiro usa dos modelos de dados e suas relações e, a partir disto, é mapeado, por exemplo, o banco de dados ou o repositório de dados a ser configurado."(L. MARTÍNEZ, 2015)

"O Entity Framework permite que você conecte um aplicativo escrito em .NET a qualquer repositório de dados sem a necessidade de alterar conexões e conectores dependendo do mecanismo de banco de dados. Se necessário, pode ser configurado para ter diferentes bancos conectados ao aplicativo e usá-los conforme especificado."(MICROSOFT, Visão Geral do Entity Framework, 2019)

1.2 O que é um framework?

Um framework é uma estrutura de software abstrata que fornece um conjunto de funcionalidades genéricas que podem ser modificadas pelo programador para atender necessidades específicas. Ele oferece uma arquitetura padrão para o desenvolvimento de aplicações, reduzindo a quantidade de código que os desenvolvedores precisam escrever do zero.

Um framework determina a estrutura do software, com extensões ou módulos personalizáveis para tarefas específicas. Essencialmente, ele dirige a lógica de fluxo de uma aplicação, definindo suas principais funções e permitindo que os desenvolvedores se concentrem nas partes únicas de seu projeto.

Pense no framework como um esqueleto ou uma estrutura básica em programação. Imagine um conjunto de ferramentas e códigos pré-fabricados que

você pode usar para desenvolver aplicativos e softwares de maneira mais eficiente.

Ao invés de começar do zero, os desenvolvedores utilizam frameworks para construir sobre uma base sólida, economizando tempo e evitando a repetição de códigos comuns. Essas estruturas oferecem um roteiro para a construção de programas, garantindo padrões de qualidade e eficiência.
[1]

[1]Desenvolver com o Entity Framework Core.`https://learn.microsoft.com/pt-br/ef/`

2 Mapeamento objeto-relacional

O Mapeamento objeto-relacional (ORM) é uma técnica de programação que permite ao programador trabalhar com dados na forma de objetos sem ter que se preocupar em como esses dados são armazenados no banco de dados. A melhor maneira de entender o ORM é vê-lo como uma camada de abstração entre o código-fonte e o banco de dados, permitindo que os desenvolvedores trabalhem com objetos de dados enquanto ocultam os detalhes de manipulação e consulta desses objetos em seus aplicativos (XIA; YU; TANG, 2009). O ORM foi útil no que se tange a modificação/manutenção de código, para criar, alterar, deletar ou atualizar uma tabela, basta acessar a classe que faz o mapeamento do domínio desejado, por exemplo: AtividadeMap, e configurar via código, ou seja, por meio de uma linguagem de programação é possível parametrizar qual será a estrutura a ser replicada no banco de dados, como acrescentar uma coluna na tabela. Outra vantagem é a facilidade de replicar as modificações, que se executa rodando um comando via package manager console do .NET conhecido como migration.

[2]

[2]Migrations são comandos que transformam alterações nas classes em comandos que modificam a estrutura do banco de dados. Elas permitem versionar e aplicar mudanças incrementais de forma controlada e rastreável.

3 SOLID

"Princípios de Desenvolvimento SOLID é uma sigla inventada por Robert C. Martin para estabelecer os cinco princípios básicos da design e programação orientados a objetos. Esta sigla está intimamente relacionada com padrões de design, especialmente com alta coesão e baixo acoplamento. O objetivo de ter um bom design de programação é cobrir a fase de manutenção de uma forma mais legível e simples, além de poder criar novas funcionalidades sem precisar modificar bastante o código antigo. Os custos de manutenção podem cobrir 80% de um projeto de software, portanto, um bom design deve ser valorizado."(RUBIRA, 2019)

SOLID, é um acrônimo de 5 princípios de programação que visam a boa prática no desenvolvimento de software orientado a objetos, idealizado por Robert C. Martin, são eles: (PAIXAO, 2019)

- S. Single responsability principle. Uma classe pode ter um, e somente um, motivo para mudar.
- O. Open/Close principle. Objetos ou entidades devem estar abertos para extensão, mas fechados para modificação
- L. Liskov Substituition Principle. Uma classe derivada deve ser substituível por sua classe base
- I. Interface Segregation Principle. Uma classe não deve ser forçada a implementar interfaces e métodos que não irão utilizar
- D. Dependency Inversion Principle. Dependa de abstrações e não de implementações

4 APLICAÇÃO DOS PRINCÍPIOS SOLID

Com o uso adequado dos princípios de design, o impacto para acomodar novos requisitos no sistema será mínimo.

4.1 Aplicação do SRP (Princípio da Responsabilidade Única)

Cada classe no sistema tem responsabilidades separadas. A classe SalaryCalculator é responsável apenas pelo cálculo do salário, sem se preocupar com o tipo de funcionário.

4.2 Aplicação do OCP (Princípio Aberto/Fechado)

A adição de um novo tipo de funcionário não requer alterações na classe SalaryCalculator, exceto pela criação de uma nova classe do tipo IEmployee.

4.3 Aplicação do LSP (Princípio da Substituição de Liskov)

Se quisermos usar a mesma lógica de SalaryCalculator para outros tipos de funcionários (além de HeadOfDepartment, Professor ou AssistantProfessor), a classe base atual pode ser usada como referência para as classes derivadas, pois não depende do tipo específico de Employee. O funcionário precisa apenas ser do tipo da interface IEmployee.

4.4 Aplicação do ISP (Princípio da Segregação de Interfaces)

As interfaces foram segregadas conforme suas responsabilidades. A interface ICalculateSalary foi definida para SalaryCalculator, e a interface ITaxCalculator foi criada para TaxCalculator, proporcionando contratos com propósitos claros.

4.5 Aplicação do DIP (Princípio da Inversão de Dependência)

As classes que lidam com Employee não precisam gerenciar o cálculo de impostos. O SalaryCalculator remove essa dependência e utiliza uma única instância de TaxCalculator.

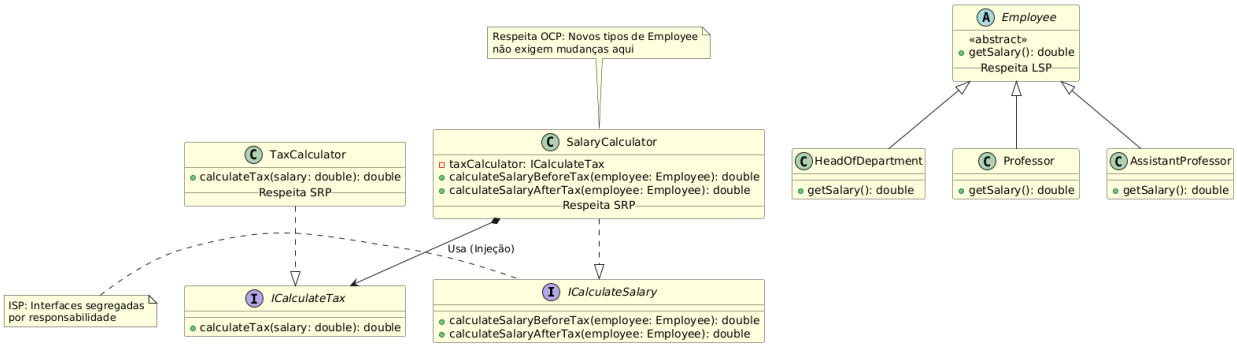

Figura 1: Design com os princípios do SOLID

5 VIOLAÇÃO DOS PRINCÍPIOS SOLID

A implementação de alterações no código para acomodar novos requisitos indica que pode haver um problema no design.

5.1 Violação do SRP (Princípio da Responsabilidade Única)

A classe SalaryCalculator não deve se preocupar com o tipo de funcionário. Em vez disso, ela deve ser responsável apenas pelo cálculo do salário, e não por lidar com detalhes do funcionário. Além disso, os funcionários não devem ser responsáveis por calcular seus próprios impostos sobre o salário.

5.2 Violação do OCP (Princípio Aberto/Fechado)

Tanto a classe quanto a interface SalaryCalculator precisam ser modificadas quando um novo tipo de funcionário é adicionado ao sistema.

5.3 Violação do LSP (Princípio da Substituição de Liskov)

Como a classe base SalaryCalculator está fortemente acoplada aos tipos de funcionários, não é possível reutilizar essa lógica para outros tipos de funcionários. Esse comportamento da classe base não faz sentido para as classes derivadas, impossibilitando o uso de um objeto da classe derivada em uma referência da classe base no sistema atual.

5.4 Violação do ISP (Princípio da Segregação de Interfaces)

A interface ICalculate contém diferentes APIs contratuais em uma única interface. O cálculo do salário e do imposto deveria ser segregado em contratos separados.

5.5 Violação do DIP (Princípio da Inversão de Dependência)

As classes de Employee estão fortemente acopladas à dependência TaxCalculator, pois todas criam uma nova instância da classe TaxCalculator. Qualquer alteração nas propriedades de TaxCalculator exigirá modificações em todas as classes onde a dependência foi acoplada rigidamente.

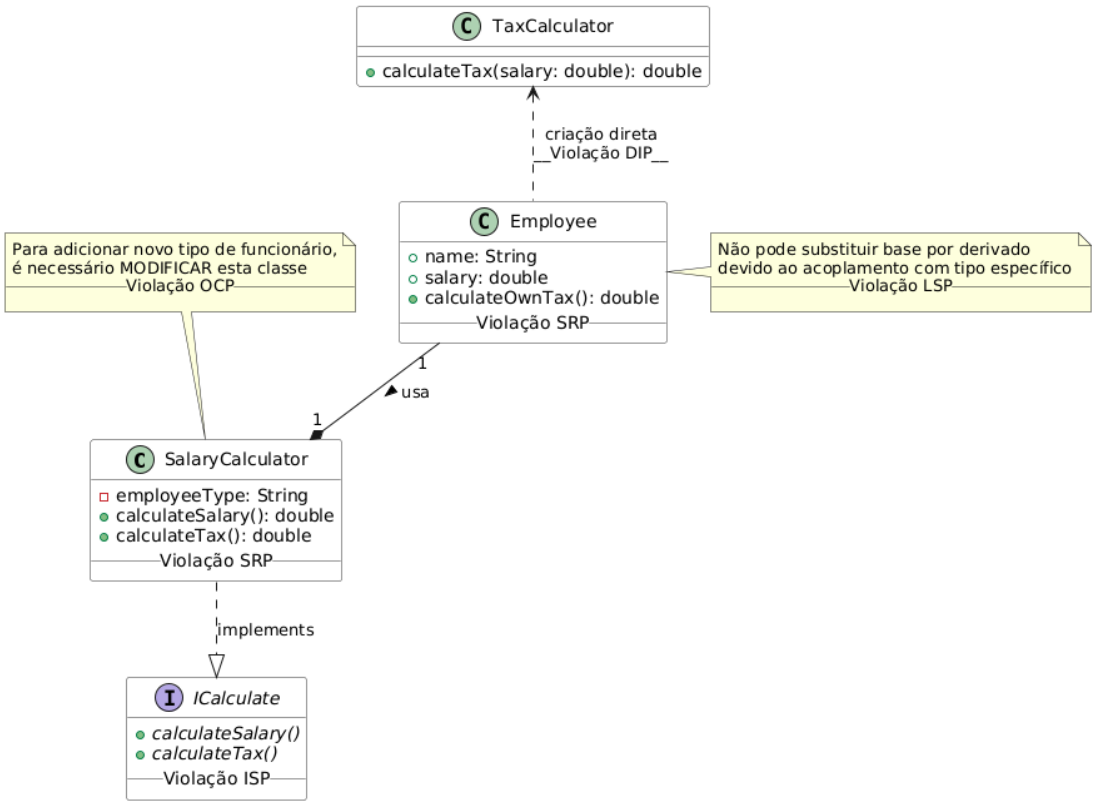

Figura 2: Design sem princípios SOLID

6 Aplicação Prática com exemplo de código

```
using Microsoft.EntityFrameworkCore;

namespace JogoDaOncinha.Data
{
    public class AppDbContext : DbContext
    {
        public DbSet<Usuario> Usuarios { get; set; }
        public DbSet<Jogo> Jogos { get; set; }
        public DbSet<Aposta> Apostas { get; set; }

         protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
        {
            optionsBuilder.UseSqlite("Data Source=CasaDeApostas.db");
        }
    }
}
```

Figura 3: Configurando o DbContext no Entity Framework

6.1 Definição das Entidades:

As propriedades `DbSet<T>` representam as tabelas no banco de dados. Neste caso, temos três entidades mapeadas:

- Usuarios: Tabela de usuários do sistema.
- Jogos: Tabela de jogos disponíveis para apostas.
- Apostas: Tabela que registra as apostas dos usuários.

6.2 Configuração do Banco de Dados:

O método *OnConfiguring* define a conexão com o banco de dados. Aqui, usamos o SQLite com um arquivo local chamado *CasaDeApostas.db*. Essa abordagem é comum para aplicações pequenas ou em desenvolvimento.

6.3 Padrão Repository com Implementação Genérica: Análise Profunda

```
public class GenericRepository<T> : IRepository<T> where T : class
 {
     private readonly AppDbContext _context;
     private readonly DbSet<T> _dbSet;

     public GenericRepository(AppDbContext context)
     {
         _context = context;
         _dbSet = context.Set<T>();
     }

     public void Add(T entity)
     {
         _dbSet.Add(entity);
         _context.SaveChanges();
     }
```

Figura 4: Padrão Repository com Implementação Genérica

6.4 Declaração da Classe Genérica:

- `GenericRepository<T>`: Define uma classe genérica que trabalha com qualquer tipo T;
- `IRepository<T>`: Indica que implementa uma interface que define operações básicas de repositório;
- `where T : class`: Restrição que exige que T seja um tipo de referência (classe), essencial para o Entity Framework.

6.5 Campos Privados:

- `_context`: Armazena a instância do banco de dados (DbContext);
- `_dbSet`: Representa a coleção específica de entidades no banco (equivalente a uma tabela);

- `readonly`: Garante que estas variáveis só podem ser atribuídas no construtor.

6.6 Resumo

O trecho de código implementa um Repositório Genérico que segue o padrão Repository, amplamente utilizado para isolar a lógica de acesso a dados da lógica de negócios. Isso promove maior coesão, reutilização e testabilidade. Por meio da injeção de dependência do DbContext, a classe interage com o banco de dados usando o Entity Framework, mantendo um alto grau de abstração e reduzindo o acoplamento entre camadas. O uso da restrição where T : class é fundamental para garantir a compatibilidade com o EF, que requer entidades como tipos de referência. A classe é facilmente reutilizável para múltiplos tipos de entidade, o que contribui para a escalabilidade e manutenção do sistema. Além de aplicar o padrão Repository, o código também respeita diversos princípios da arquitetura SOLID. O princípio da responsabilidade única (SRP) é evidente, pois a classe se limita à lógica de persistência de dados. Através da utilização de genéricos e interfaces, ela também está aberta para extensão e fechada para modificação (OCP), além de permitir substituições seguras conforme o princípio de Liskov (LSP). O fato de depender de uma abstração (IRepository¡T¿) e de receber o DbContext por injeção de dependência mostra a aplicação direta do princípio da inversão de dependência (DIP), tornando o código mais flexível e testável.

6.7 Diagrama de Relacionamento

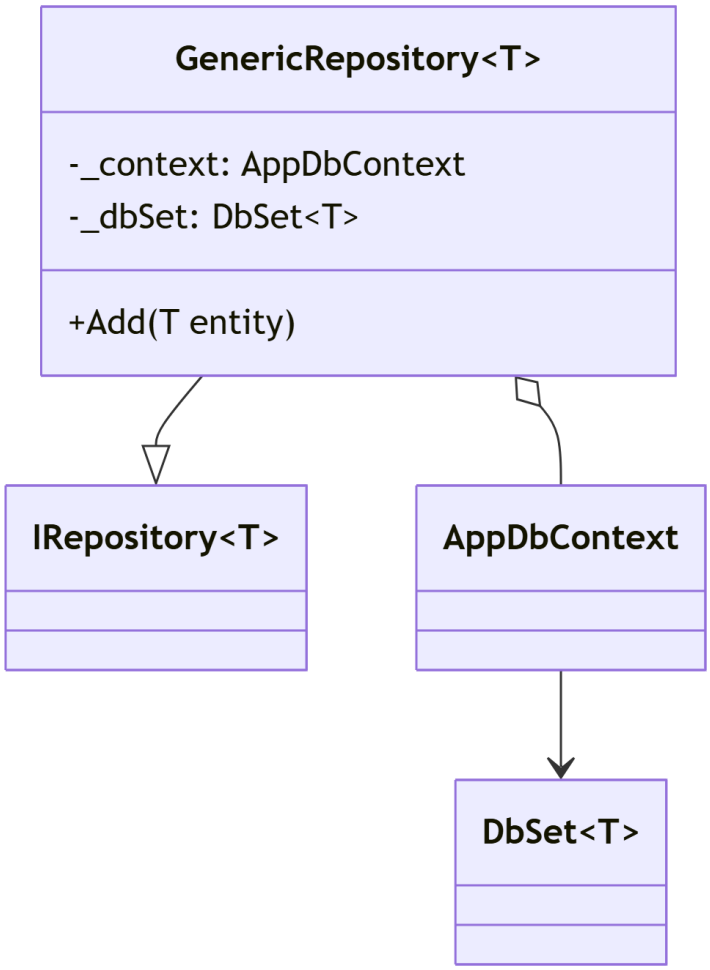

Figura 5:

7 Conclusão

Para explorar os exemplos em detalhes e experimentar a implementação, o código-fonte deste trabalho está disponível no repositório: `https://github.com/GuiFagun/JogoDaOncinhaPOO2`

Além disso, para complementar o estudo, gravamos um vídeo explicativo demonstrando a implementação dos conceitos abordados, incluindo configurações

do Entity Framework e aplicação dos princípios SOLID em um projeto real. Confira: `https://youtu.be/uZFvqtD_RAU?si=zoatI3CMaaDevhYP`

8 Referências

Referências

[1] SHARMA, Tushar; SPINELLIS, Diomidis. *Effect of SOLID Design Principles on Quality of Software: An Empirical Assessment*. Journal of Systems and Software, v. 123, p. 1–15, 2016.

[2] PRICE, Mark J. *C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development*. 4. ed. Packt Publishing, 2019.

[3] MICROSOFT. *Entity Framework Core - Introdução*. Documentação oficial, 2024. Disponível em: `https://learn.microsoft.com/pt-br/ef/core/get-started/overview/first-app`.