GCT088 — Aula 4.1
Camada de Domínio e Testes do Domínio

Projeto de Software

UFLA

Visual Studio Community + .NET 8

Objetivo da aula

- Modelar o Domínio com entidades e value objects
- Definir invariantes e comportamentos no núcleo
- Escrever testes de unidade do Domínio (xUnit)
- Critério: Domínio sem dependências de Web/Infra

Por que começar pelo Domínio?

- O Domínio captura as regras do negócio (fonte da verdade)
- Reduz acoplamento: Application e Infra orbitam o Domínio
- Testabilidade: código puro, feedback rápido, menos flakiness

Entidades

- Identidade + ciclo de vida; devem manter invariantes
- Ex.: Product (Id, Name, Price, StockQuantity)
- Comportamentos encapsulados (ex.: AddStock/RemoveStock)

Exemplo — Entidade Product

```
namespace LojaOnLine.Domain.Entities;

public sealed class Product
{
    public Guid Id { get; }
    public string Name { get; }
    public decimal Price { get; private set; }
    public int StockQuantity { get; private set;
        }

    public Product(string name, decimal price,
        int stockQuantity)
    {
        if (string.IsNullOrWhiteSpace(name))
            throw new ArgumentException("Name is
                required", nameof(name));
        if (price < 0)
            throw new ArgumentException("Price
                must be >= 0", nameof(price));
        if (stockQuantity < 0)
```

Value Objects (VO)

- Imutáveis; igualdade por valor (não por identidade)
- Encapsulam conceitos: Money, Sku, Email
- Eliminam duplicação de validações e regras

Exemplo — Value Object Money (opcional)

```
namespace LojaOnLine.Domain.ValueObjects;

public sealed record Money(string Currency,
    decimal Amount)
{
    public Money : this(Currency.
        ToUpperInvariant(), Amount)
    {
        if (string.IsNullOrWhiteSpace(Currency))
            throw new ArgumentException("
                Currency required", nameof(
                Currency));
        if (Amount < 0) throw new
            ArgumentException("Amount >= 0",
            nameof(Amount));
    }
}
```

Value Object — visão intuitiva

- Pense em "um valor com regras": sempre válido e comparável por conteúdo
- Não tem identidade própria (sem Id); dois VOs iguais em conteúdo são o mesmo
- Bom para: Email, CPF/CNPJ, Money, Sku, FaixaEtaria, etc.
- Benefícios práticos: validação centralizada, menos ifs repetidos, código mais legível

Exemplo (conceitual): Email só existe se for válido; duas instâncias com o mesmo texto são iguais.

Classe sealed — visão intuitiva

- "Não deixa herdar": evita extensões que quebrem regras do tipo
- Use quando: a classe já expressa tudo o que precisa; herança tentaria burlar invariantes
- Ajuda a manter o contrato estável e facilita raciocinar sobre o tipo
- Ex.: Domínio sensível (Product com regras de estoque/preço) — bloqueie herança

Dica: prefira composição a herança; libere herança só quando houver um motivo arquitetural claro.

Invariantes e comportamentos

- Invariantes: Name obrigatório; Price ≥ 0; Stock ≥ 0
- Comportamentos: AddStock, RemoveStock, (ChangePrice)
- Padrão: lançar exceções ao violar invariantes

Serviços de Domínio

- Regras que não pertencem a uma única entidade
- Funções puras sobre entidades/VOs
- Ex.: política de desconto (tabela de regras) aplicada a Products

Testes do Domínio (xUnit)

- Sem banco/HTTP; apenas Domínio
- Testes de invariantes: criação inválida deve falhar
- Testes de comportamento: estoque, preço
- Test Data Builders para reduzir repetição

Exemplo — Testes xUnit (invariantes)

```
using LojaOnLine.Domain.Entities;
using Xunit;

public class ProductTests
{
    [Theory]
    [InlineData(null)]
    [InlineData("")]
    [InlineData("   ")]
    public void Create_InvalidName_Throws(string
        ? name)
    {
        Assert.Throws<ArgumentException>(() =>
            new Product(name!, 10m, 1));
    }

    [Fact]
    public void Create_NegativePrice_Throws()
    {
        Assert.Throws<ArgumentException>(() =>
```

Exemplo — Testes xUnit (comportamentos)

```
using LojaOnLine.Domain.Entities;
using Xunit;

public class ProductBehaviorTests
{
    [Fact]
    public void
        AddStock_WithPositiveAmount_IncreasesStock
        ()
    {
        var p = new Product("A", 10m, 1);
        p.AddStock(3);
        Assert.Equal(4, p.StockQuantity);
    }

    [Fact]
    public void RemoveStock_Insufficient_Throws
        ()
    {
        var p = new Product("A", 10m, 1);
```

Critérios de sucesso

- Domínio expressa regras com clareza e consistência
- Testes cobrem cenários positivos e negativos
- Domínio permanece independente de tecnologia

Próxima aula

- Camada de Application: casos de uso e orquestração usando o Domínio
- Ainda sem Infra: foco em DTOs e serviços de aplicação