

# TRANSIÇÃO PARA ORDERITEM E ORDER: EXPANDINDO INVARIANTES

Concluimos a análise da classe `Money` com três pilares fundamentais: a aplicação de validação fail-fast para valores monetários negativos, a priorização da imutabilidade em todas as operações, e a escolha estratégica do tipo `decimal` para garantir precisão em cálculos financeiros. Esses princípios, testados e validados, estabeleceram um contrato claro sobre o comportamento esperado de valores monetários no domínio.

Com essa base sólida, a transição para as classes `OrderItem` e `Order` é um passo natural, permitindo-nos explorar como esses conceitos de invariantes e validação se aplicam em um contexto de associações 1:N. Saímos do "valor isolado" para o "valor em contexto", onde cada componente da associação tem suas próprias regras a zelar.

## ORDERITEM: INVARIANTES LOCAIS

`OrderItem` representa uma linha individual de um pedido e é responsável por proteger seu próprio estado. Isso inclui:

1

- Normalização de SKU: Eliminar variações de formatação (espaços, maiúsculas/minúsculas) para garantir consistência.
- Quantidade Positiva: Assegurar que a quantidade de itens seja estritamente maior que zero.
- Subtotal Derivado: Calcular o subtotal com base nos valores internos, evitando duplicação ou cache inconsistente.

## ORDER: ORQUESTRAÇÃO E AGREGAÇÃO

A classe `Order` atua como o orquestrador do agregado, encapsulando a coleção de `OrderItems` e garantindo a consistência global:

2

- Encapsulamento: Gerencia a coleção de itens, protegendo-a de acesso direto e mutações externas.
- Normalização Centralizada: Aplica a normalização de SKU em operações de busca, adição e remoção para manter a integridade.
- Acúmulo Inteligente: Consolida quantidades quando o mesmo SKU é adicionado repetidamente.
- Total Derivado: Expõe o valor total do pedido como um cálculo em tempo real, refletindo sempre o estado atual dos itens.

Esse arranjo reforça o princípio de que o agregado concentra a mutação e a coordenação da sua coleção, enquanto as partes (os `OrderItems`) garantem a validade local dos seus próprios dados. A navegabilidade, por enquanto unidirecional do pedido para seus itens, reduz o acoplamento e facilita a testabilidade, criando um efeito em cadeia onde parâmetros inválidos são rejeitados e estados inconsistentes não persistem.



Antes de avançarmos para a implementação, é crucial verificar alguns pontos: a classe `Money` está completa e seus dois testes mínimos passam; a decisão de normalizar o SKU é uma regra sistemática para `OrderItem` e `Order`; o total do pedido será sempre derivado e não armazenado em cache; e a estrutura do repositório já contém os projetos de domínio e testes. Com esses pré-requisitos atendidos, prosseguiremos com a implementação detalhada de `OrderItem` e, em seguida, de `Order`, consolidando a compreensão de como multiplicidade e invariantes se traduzem em código legível, testável e coerente.

## CLASSE ORDERITEM

```
using System;

namespace Associations.Domain
{
    /// <summary>
    /// Item de pedido (parte do agregado Order).
    /// Invariantes locais:
    /// - SKU é sempre armazenado normalizado (Trim + UpperInvariant).
    /// - Quantity > 0.
    /// - Subtotal é derivado (evita divergência).
    /// </summary>
    public sealed class OrderItem
    {
        /// <summary>SKU sempre normalizado (Trim + UpperInvariant).
        /// </summary>
        public string Sku { get; }

        public int Quantity { get; private set; }

        public Money UnitPrice { get; }
```

```

/// <summary>Cálculo derivado para evitar inconsistências.
/// </summary>
public Money Subtotal => new(UnitPrice.Value * Quantity);

public OrderItem(string sku, int quantity, Money unitPrice)
{
    if (string.IsNullOrEmpty(sku))
        throw new ArgumentException("SKU vazio", nameof(sku));

    if (quantity <= 0)
        throw new ArgumentOutOfRangeException(nameof(quantity),
            "Quantidade deve ser maior que zero.");

    UnitPrice = unitPrice ?? throw new
        ArgumentNullException(nameof(unitPrice));

    // Invariante local: armazenar SKU normalizado
    Sku = NormalizeSku(sku);
    Quantity = quantity;
}

internal void Increase(int delta)
{
    if (delta <= 0)
        throw new ArgumentOutOfRangeException(nameof(delta),
            "Variação de quantidade deve ser maior que zero.");
    Quantity += delta;
}

/// <summary>
/// Reduz a quantidade sem permitir zerar/ultrapassar.
/// Caso deseje tratar "zerou" como remoção do item no agregado,
/// ver a variante sugerida no material (Decrease que retorna
/// bool).
/// </summary>
internal void Decrease(int delta)
{
    if (delta <= 0)
        throw new ArgumentOutOfRangeException(nameof(delta),
            "Delta deve ser > 0.");

    if (delta >= Quantity)
        throw new InvalidOperationException(
            "Redução inválida: zeraria ou tornaria negativa a
            quantidade. Use a remoção do item no agregado.");
}

```

```

        Quantity -= delta;
    }

    /// <summary>Normalização usada pela própria classe para garantir o
    /// invariante.</summary>
    private static string NormalizeSku(string raw)
        => raw.Trim().ToUpperInvariant();
    }
}

```

# IMPLEMENTAÇÃO DA CLASSE ORDER

```

using System;
using System.Collections.Generic;
using System.Linq;

namespace Associations.Domain
{
    /// <summary>
    /// Agregado Order (orquestra as partes).
    /// Regras:
    /// - Coleção encapsulada (ICollection).
    /// - Total é derivado (soma dos subtotais), sem cache/persistência.
    /// - SKU é normalizado ANTES de buscar/adicionar/remover (consistência
    /// com OrderItem).
    /// </summary>
    public sealed class Order
    {
        private readonly List<OrderItem> _items = new();

        /// <summary>Exposição somente-leitura da coleção (evita mutação
        /// externa).</summary>
        public ICollection<OrderItem> Items => _items.AsReadOnly();

        /// <summary>Total derivado = soma dos subtotais.</summary>
        public Money Total => _items.Aggregate(new Money(0), (acc, it) =>
            acc + it.Subtotal);
    }
}

```

```

/// <summary>
/// Adiciona item. Se o SKU já existir, acumula quantidade.
/// SKU é normalizado para garantir comparações consistentes.
/// </summary>
public void AddItem(string sku, int quantity, Money price)
{
    var key = NormSku(sku);
    if (string.IsNullOrEmpty(key))
        throw new ArgumentException("SSKU não pode ser nulo ou
            vazio.", nameof(sku));
    if (price is null)
        throw new ArgumentNullException(nameof(price));

    // Busca por SKU já normalizado
    var existing = _items.FirstOrDefault(i => i.Sku == key);

    if (existing is null)
    {
        // OrderItem também normaliza internamente; aqui já
        // passamos normalizado
        _items.Add(new OrderItem(key, quantity, price));
    }

    else
    {
        existing.Increase(quantity);
    }
}

/// <summary>
/// Remove todos os itens com o SKU informado (após normalização).
/// Retorna true se removeu algo.
/// </summary>
public bool RemoveItem(string sku)
    => _items.RemoveAll(i => i.Sku == NormSku(sku)) > 0;

/// <summary>
/// Remoção parcial por quantidade (opcional).
/// Se a redução zerar ou ultrapassar, converte para remoção total
/// do SKU.
/// </summary>

```

```


public bool RemoveItem(string sku, int quantity)
{
    var key = NormSku(sku);
    var it = _items.FirstOrDefault(i => i.Sku == key);
    if (it is null) return false;
    if (quantity <= 0)
        throw new ArgumentOutOfRangeException(nameof(quantity),
            "Quantity deve ser > 0.");

    if (quantity >= it.Quantity)
        return _items.Remove(it); // remoção total do SKU

    it.Decrease(quantity);    // remoção parcial
    return true;
}

/// <summary>
/// Normalização no agregado garante buscas/remoções consistentes
/// mesmo que o usuário digite com espaços/casing diferentes.
/// </summary>
private static string NormSku(string? raw)
    => string.IsNullOrEmpty(raw) ? string.Empty :
        raw.Trim().ToUpperInvariant();
}
}

```

 Note o uso de coleção privada com expositor IReadOnlyCollection e métodos controlados para mutação.

Como o `Total` é uma **propriedade derivada** – `public Money Total => _items.Aggregate(...)` – ele **recalcula automaticamente** após qualquer mudança em `_items`; não precisa “forçar” recálculo.

# NORMALIZAÇÃO DE SKU: INTEGRIDADE EM ORDERITEM E ORDER

A normalização do SKU é um pilar fundamental para garantir a integridade e consistência dos dados em nosso sistema de pedidos. Ela é aplicada tanto na entidade `OrderItem` quanto no agregado `Order`, cada qual com um papel distinto, mas complementar, no ciclo de vida de um item.



## NORMALIZAÇÃO COMO INVARIANTE LOCAL EM ORDERITEM

No `OrderItem`, a normalização (realizada via `trim()` e `ToUpperInvariant()`) é um invariante local. Isso significa que:

- Qualquer instância de `OrderItem` é construída com seu SKU já normalizado, protegendo o estado interno contra variações de formatação.
- Garante que a comparação ou uso do SKU dentro do próprio `OrderItem` seja sempre consistente, independentemente da entrada original.
- Reforça a integridade da entidade em seu nível mais granular.



## NORMALIZAÇÃO NA ORQUESTRAÇÃO DO AGREGADO ORDER

Na classe `Order`, a normalização complementa a do `OrderItem`, atuando como parte essencial da orquestração do agregado. Aqui, ela garante a consistência das operações externas:

- Toda busca, adição ou remoção de itens normaliza o SKU de entrada antes de qualquer comparação com os `OrderItems` contidos.
- Assegura que SKUs como "abc-123" e "ABC-123" sejam tratados como o mesmo item no contexto do pedido.
- Evita a criação de "itens fantasmas" ou inconsistências na coleção do pedido, mesmo que o usuário digite com casing ou espaços diferentes.

Essa abordagem de dupla normalização é crucial para prevenir bugs sutis, como a criação de itens duplicados devido a diferenças de maiúsculas/minúsculas ou espaços. Ela demonstra na prática como a combinação de multiplicidade e invariantes se traduz em regras de código explícitas e robustas, garantindo que o sistema se comporte de forma previsível e confiável.

# LAB GUIADO - COMPOSIÇÃO 1:N

## 1 CRIAR ORDER E INSERIR ITENS


Crie uma nova `Order` e adicione 3 itens, incluindo um SKU repetido para testar a lógica de consolidação

## 2 VERIFICAR CONTAGEM E TOTAL

Valide que `Items.Count` reflete a quantidade correta e que o `Total` corresponde à soma esperada dos subtotais

## 3 REMOVER ITEM E REVALIDAR

Execute `Removeltem()` para um SKU existente e confirme que o total foi recalculado automaticamente



isolated value

Value in context,

```
using System;
using System.Diagnostics;

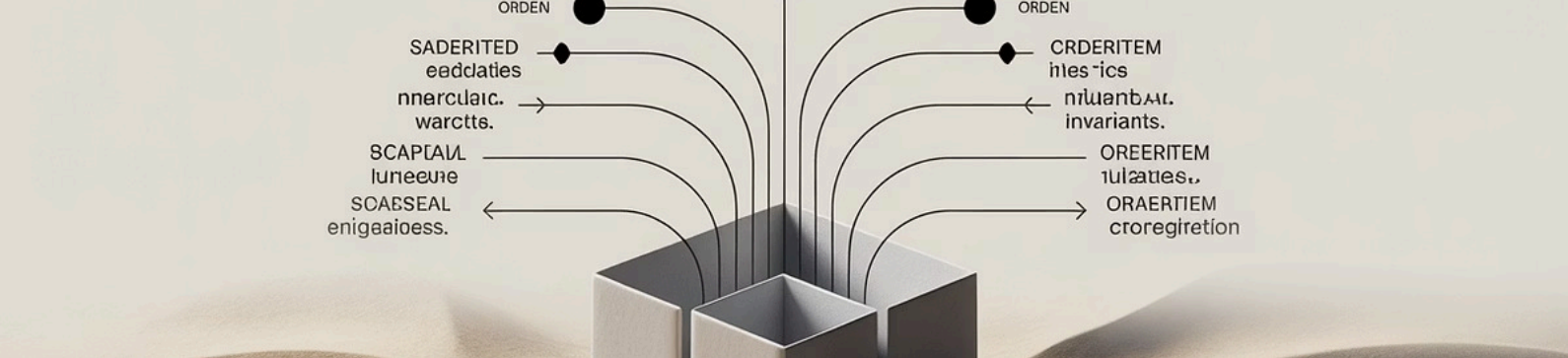
class Program
{
    static void Main()
    {
        var order = new Order();

        // Arrange — adiciona 3 itens (um SKU repetido para somar
        // quantidades)
        order.AddItem("ABC-001", 2, new Money(10.00m)); // subtotal = 20.00
        order.AddItem("XYZ-9", 1, new Money(5.50m)); // subtotal = 5.50
        order.AddItem("ABC-001", 3, new Money(10.00m)); // acumula ABC-001:
        // 5 * 10 = 50.00

        // Verifica total antes da remoção
        var totalAntes = order.Total.Value; // 50.00 + 5.50 = 55.50
        Debug.Assert(totalAntes == 55.50m, $"Esperado 55.50, veio
            {totalAntes}");

        // Act — remove SKU existente
        var removed = order.RemoveItem("XYZ-9");
        Debug.Assert(removed, "Item XYZ-9 deveria existir e ser removido");

        // Assert — total depois da remoção (fica só ABC-001 = 50.00)
        var totalDepois = order.Total.Value;
        Debug.Assert(totalDepois == 50.00m, $"Esperado 50.00, veio
            {totalDepois}");
        Console.WriteLine($"OK — Total antes: {totalAntes}, depois:
            {totalDepois}");
    }
}
```



# ENCERRAMENTO – DO “VALOR ISOLADO” AO “VALOR EM CONTEXTO”

Neste capítulo, saímos do **valor isolado** (tipos que se bastam, como `Money`) para o **valor em contexto**, modelando a associação **1:N por composição** entre `Order` e `OrderItem`. O foco foi transformar regras espalhadas em **invariantes explícitos** e **encapsulados** no próprio modelo.

## O QUE VOCÊ CONSOLIDOU

### INVARIANTES QUE SE REFORÇAM

`OrderItem` não aceita quantidade  $\leq 0$  nem SKU “torto”; `Order` garante que operações mantenham o agregado coeso (somar o mesmo SKU acumula, remover ajusta o total). O **Total** é derivado do estado – nada de cache divergente.

### COMPOSIÇÃO COM ENCAPSULAMENTO

A coleção de itens não é “terra de ninguém”; é exposta somente para leitura e toda mudança passa por métodos intencionais do agregado. Isso reduz acoplamento e torna a regra “visível” no código.

### NORMALIZAÇÃO E PREVISIBILIDADE

Nomes padronizados, comparações consistentes e validações nas bordas do tipo evitam duplicidades silenciosas e “itens fantasmas”.

### EXCEÇÕES COMO VOCABULÁRIO DO DOMÍNIO

Parâmetros inválidos → `Argument*`; operações que quebram o estado do agregado → `InvalidOperationException`. Mensagens curtas e diretas explicam a regra violada.

## ARMADILHAS QUE VOCÊ APRENDEU A EVITAR

- Expor listas mutáveis (quem mexe “por fora” burla as regras).
- Calcular total uma vez e “confiar na memória” (diverge do estado real).
- Tratar nulo e espaços de forma inconsistente (duplica itens, quebra igualdade).



# CHECKLIST RÁPIDO PARA LEVAR ADIANTE

01	02	03
<b>ESTADO VÁLIDO</b> O construtor e os métodos garantem <b>estado válido</b> após qualquer operação?	<b>COLEÇÕES PRIVADAS</b> Coleções são <b>privadas</b> e expostas como somente leitura?	<b>VALORES DERIVADOS</b> Valores derivados ( <b>Total</b> ) são <b>calculados a partir da fonte da verdade</b> ?
04	05	
<b>MENSAGENS DE EXCEÇÃO</b> Mensagens de exceção <b>dizem qual regra foi quebrada</b> em uma frase?	<b>NORMALIZAÇÃO</b> Há <b>normalização</b> (trim/case/format) no ponto certo para evitar duplicidade?	

## PONTE PARA OS PRÓXIMOS TÓPICOS DA DISCIPLINA

- **Exceções (93):** lapidar mensagens, escolher o tipo de exceção certo e padronizar pontos de validação (construtores e métodos que mudam estado).
- **Métodos estáticos (103):** separar **operações puras e utilitárias** (ex.: normalização, validação, conversão) para manter o agregado limpo e coeso.
- **Nulidade com intenção:** declarar quando algo pode faltar (0..1) vs. quando é obrigatório (1), alinhando design da associação com o uso de nulos no código.

### Fechamento

Você agora tem um **agregado coerente**, que **falha cedo, normaliza sempre e deriva o que deve ser derivado**. O próximo passo é polir a “superfície de contato” do seu modelo — mensagens de erro, utilitários estáticos e contratos de nulidade — mantendo o mesmo norte: **clareza, previsibilidade e código que ensina o domínio**.