ROBUSTEZ SISTEMÁTICA E ASSOCIAÇÕES DE DOMÍNIO

Avançando dos fundamentos da validação pontual, o foco agora se volta para a construção de um sistema intrinsecamente robusto, onde a integridade dos dados é garantida por um conjunto de testes abrangente e pelo design cuidadoso das associações de domínio. Este é um salto do "código que apenas compila" para modelos de domínio resilientes e autoconsistentes.

1

VALIDAÇÃO DE ENTRADAS CRÍTICAS

Rejeitar quantidades (Quantity) menores ou iguais a zero e impedir SKUs nulos ou vazios são validações essenciais nas fronteiras do domínio, aplicando uma estratégia fail-fast para evitar dados inconsistentes.

2

PRECISÃO MONETÁRIA

Garantir a precisão das somas monetárias, especialmente com valores decimais, é crucial para a integridade financeira do sistema, eliminando potenciais erros de arredondamento.

3

GESTÃO SEGURA DE COLEÇÕES

Evitar a exposição direta de `List<T>` ou coleções com `set` público para manter o encapsulamento e controlar a mutação dos itens dentro do domínio.

4

REMOÇÃO DE ITENS INEXISTENTES

Testar a remoção de itens que não existem sem causar efeitos colaterais indesejados no estado total do objeto ou gerar exceções inesperadas.

Este checklist de testes forma a espinha dorsal de um processo de desenvolvimento que não apenas valida entradas, mas também assegura a consistência interna do modelo de domínio, acelerando a depuração e facilitando a manutenção através de falhas claras e informativas.

ASSOCIAÇÕES ROBUSTAS E SINCRONIZADAS

Além dos testes unitários detalhados, o próximo passo é construir associações de domínio que mantenham a integridade de forma proativa. O exemplo clássico é a relação 1:N bidirecional, como entre um `Department` e seus `Employees`. A robustez aqui reside na sincronização automática: a adição de um funcionário a um departamento deve, automaticamente, associar esse departamento ao funcionário, e a remoção deve anular essa associação.

Isso é alcançado através de um ponto único de mutação, onde as alterações de relacionamento são orquestradas para garantir que ambos os lados da associação permaneçam consistentes, eliminando a chance de estados inconsistentes. Os esqueletos de especificação fornecidos permitem estender a suíte de testes para validar estas regras de consistência em todo o ciclo de vida da aplicação.

TESTES DE FRONTEIRA E ARMADILHAS

TESTES DE FRONTEIRA

- Quantity menor ou igual a zero
- SKU vazio ou nulo
- Somas com valores decimais precisos
- Remoção de SKU inexistente

ARMADILHAS COMUNS

- Expor List<T> diretamente
- Permitir set público em coleções
- Total persistido divergente do cálculo
- Não validar parâmetros de entrada

Sempre mantenha o encapsulamento das coleções e valide entradas para preservar a integridade dos dados.

TESTE UNITÁRIO: ADIÇÃO DE ITENS AO PEDIDO

`ADDITEM_SKUINEXISTENTE_DEVEADICIONARNOVOITEM`

Verifica se um item com um SKU não existente é adicionado corretamente à coleção de itens do pedido.

O teste `AddItem_SkuInexistente_DeveAdicionarNovoItem` é um exemplo fundamental de teste unitário, projetado para validar o comportamento da classe `Order` ao adicionar um novo item que ainda não está presente no pedido. Ele garante a inserção correta e a integridade dos dados no domínio.

OBJETIVO PRINCIPAL

O propósito deste teste é assegurar que, quando um item com um SKU (Stock Keeping Unit) inexistente é adicionado a um pedido:

- Ele seja corretamente inserido na coleção de itens do pedido.
- Os dados associados (SKU, quantidade, preço unitário) sejam armazenados com precisão.

ESTRUTURA DO TESTE: PADRÃO AAA

Seguindo o padrão **Arrange-Act-Assert (AAA)**, o teste é dividido em três fases claras:

```
// Arrange (Preparação)
var order = new OrderEntity();
var sku = "PROD-001";
var quantity = 2.0;
var unitPrice = new Money(50.00m);

// Act (Ação)
order.AddItem(sku, quantity, unitPrice);

// Assert (Verificação)
Assert.Single(order.Items);
Assert.Contains(order.Items, item =>
  item.Sku == "PROD-001" && item.Quantity == 2.0);
```

- **Arrange:** Prepara o ambiente, instanciando um novo `Order` e definindo os parâmetros do item a ser adicionado.
- **Act:** Executa a ação que está sendo testada, neste caso, a chamada ao método `AddItem`.
- **Assert:** Verifica o resultado, garantindo que a coleção de itens do pedido contém exatamente um item e que este item possui o SKU e a quantidade esperados.

CONCEITOS FUNDAMENTAIS ABORDADOS

NOMENCLATURA (NAMING CONVENTION)

O nome do teste segue `MetodoTestado_Cenario_ComportamentoEsperado`, tornando-o auto-documentado e facilitando a compreensão de sua finalidade.

VALUE OBJECTS (MONEY)

A utilização de um `Value Object` como `Money` em vez de um tipo primitivo (`decimal`) confere semântica, encapsula regras de negócio e garante validação automática para valores monetários.

ALIAS DE NAMESPACE

O uso de `using OrderEntity = Associations.Domain.Order.Order;` resolve conflitos de nomes, melhorando a clareza do código quando a classe e o namespace compartilham o mesmo nome.

IMPORTÂNCIA DESTE TESTE

Este teste é crucial não apenas para validar a funcionalidade de adição de itens, mas também para:

- **Documentar o Comportamento:** Serve como uma especificação viva do comportamento esperado da classe `Order`.
- **Detectar Regressões:** Garante que futuras alterações no código não quebrem a funcionalidade existente.
- **Facilitar Refatoração:** Permite modificações na implementação com maior confiança.
- **Servir como Exemplo:** Demonstra o uso correto da classe `Order` e suas associações.

PARA REFLEXÃO

Considere como o sistema deveria reagir ao tentar adicionar o mesmo SKU múltiplas vezes. Isso levaria a uma atualização da quantidade ou a uma exceção? Que outros cenários de teste seriam relevantes para cobrir completamente a funcionalidade de adição de itens?

TESTE UNITÁRIO: DECREMENTA QUANTIDADE MAIOR QUE ATUAL E LANÇA EXCEÇÃO

Continuando nossa exploração sobre a robustez de domínios, este teste foca em um cenário crítico: o que acontece quando se tenta diminuir a quantidade de um item de pedido em um valor maior do que o realmente disponível? A integridade dos dados de um sistema financeiro ou de estoque depende da correta validação dessas operações, evitando saldos negativos ou inconsistências lógicas.

`DECREASE_QUANTIDADEMAIORQUEATUAL_LANCAEXCECAO`

Assegura que uma exceção é lançada se a tentativa de diminuir a quantidade de um item resultar em um valor negativo.

O teste `Decrease_QuantidadeMaiorQueAtual_LancaExcecao` é um exemplo fundamental de teste unitário de validação de regras de negócio, projetado para assegurar que a classe **OrderItem** proteja a integridade dos dados ao impedir operações inválidas. Ele garante que tentativas de decrementar uma quantidade superior à disponível em estoque resultem em uma exceção apropriada, preservando a consistência do domínio e prevenindo estados inválidos no sistema. Este tipo de teste é crucial para validar comportamentos defensivos e garantir que as invariantes do modelo de domínio sejam sempre respeitadas, demonstrando um princípio essencial da programação orientada a objetos: o encapsulamento de regras de negócio dentro das entidades.

O teste `Decrease_QuantidadeMaiorQueAtual_LancaExcecao` é vital para garantir que a entidade `OrderItem` preserve suas invariantes. Ele simula uma tentativa de decremento que violaria a lógica de negócio (por exemplo, remover 10 unidades de um item que só possui 5 em estoque), e valida que o sistema responde com uma exceção apropriada, protegendo a consistência do modelo de domínio.

ESTRUTURA DO TESTE: GARANTINDO LIMITES (PADRÃO AAA)

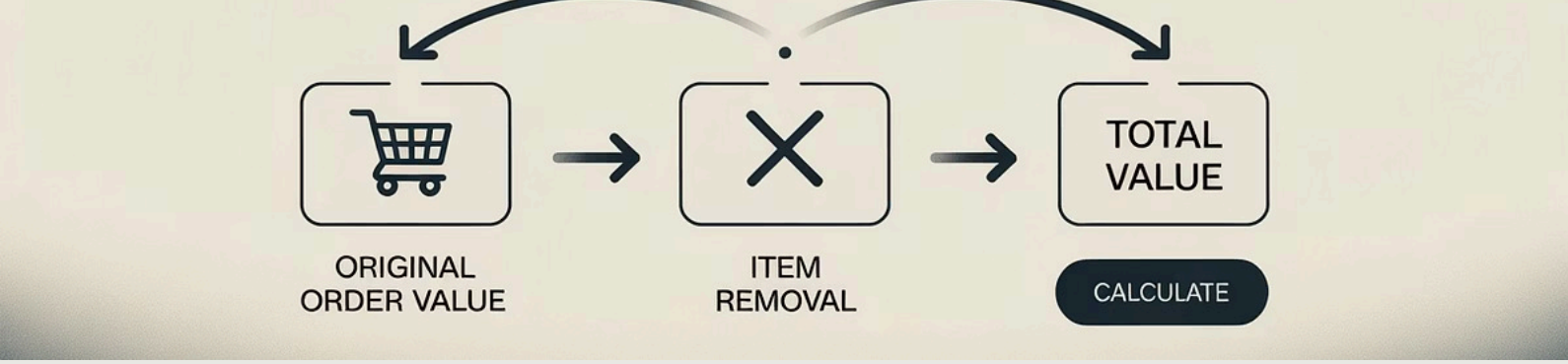
A aplicação do padrão Arrange-Act-Assert (AAA) neste teste é clara e direta, focando na simulação de uma condição de erro e na verificação da exceção esperada.

```
[Fact]
public void Decrease_QuantidadeMaiorQueAtual_LancaExcecao()
{
    // Arrange
    var sku = "PROD-002";
    var quantidadeInicial = 5.0;
    var unitPrice = new Money(100.00m);
    var orderItem = new OrderItem(sku, quantidadeInicial, unitPrice);
    var quantidadeParaDecrementar = 10.0; // Maior que a quantidade atual

    // Act & Assert
    var exception = Assert.Throws<InvalidOperationException>(() =>
    orderItem.Decrease(quantidadeParaDecrementar));

    Assert.Equal("10 é superior ao que existe em estoque",
       exception.Message);
}
```

ARRANGE (PREPARAR)

Primeiro, criamos um `OrderItem` com um SKU específico ("PROD-002"), uma quantidade inicial de 5.0 unidades e um preço unitário de R$ 100,00. Em seguida, definimos a quantidade que tentaremos decrementar: 10.0 unidades, um valor propositalmente maior do que o disponível.

```
var item = new OrderItem("PROD-002", 5, new Money(100));
var amountToDecrease = 10;
```



ACT & ASSERT (AGIR E VERIFICAR)

A fase de "Act" é incorporada diretamente no "Assert". Usamos `Assert.Throws<InvalidOperationException>` para envolver a chamada ao método `Decrease`. Isso instrui o teste a esperar uma exceção do tipo `InvalidOperationException`. Capturamos essa exceção para então verificar se sua mensagem corresponde exatamente à esperada: "10 é superior ao que existe em estoque".

```
var exception = Assert.Throws<InvalidOperationException>(() =>
item.Decrease(amountToDecrease));
Assert.Equal($"{amountToDecrease} é superior ao que existe em estoque",
exception.Message);
```

A IMPORTÂNCIA DA VALIDAÇÃO DE LIMITES

Este teste transcende a mera verificação de uma funcionalidade; ele é um guardião da integridade do sistema:

- **Impede Quantidade Negativa:** Garante que a quantidade de um item nunca se torne negativa, o que seria um estado ilógico para um estoque ou pedido.
- **Protege o Estoque:** Assegura que o estoque não seja decrementado além do disponível, evitando fraudes ou erros de contabilidade.
- **Evita Operações Silenciosas:** Impede que operações inválidas sejam ignoradas ou causem efeitos colaterais indesejados sem um aviso adequado.
- **Documenta Regras de Negócio:** Serve como documentação executável das regras de negócio relacionadas à gestão de quantidades.

CONCEITO: ASSERT.THROWS

Este método do xUnit é a ferramenta padrão para verificar o lançamento de exceções. Ele executa o código fornecido em uma expressão lambda e passa no teste se uma exceção do tipo especificado for lançada. Caso contrário, o teste falha. A capacidade de capturar a exceção permite verificações adicionais, como a mensagem da exceção, que é crucial para garantir que a exceção seja lançada pelo motivo correto.

TESTE UNITÁRIO: CÁLCULO DO VALOR CORRETO APÓS REMOÇÃO

`TOTAL_APOSREMOCAO_CALCULAVALORCORRETO`

Confirma que o valor total do pedido é atualizado precisamente após a remoção de um item.

Essa estrutura permite que, mesmo ao falhar, o nome do teste indique exatamente qual regra de negócio ou invariante foi violado, acelerando o processo de depuração e manutenção.

O teste `Total_AposRemocao_CalculaValorCorreto` é um exemplo fundamental de teste unitário de cálculos agregados, projetado para validar que a propriedade **Total** da classe **Order** reflete com precisão o valor total do pedido após operações de remoção de itens. Ele garante que a lógica de cálculo do total seja recalculada dinamicamente e mantenha a consistência financeira do domínio, mesmo após modificações na composição do pedido. Este tipo de teste é essencial para assegurar a integridade de valores monetários e a correção de regras de negócio relacionadas a agregações e cálculos derivados.

ESTRUTURA DETALHADA DO TESTE DE CÁLCULO DO TOTAL APÓS REMOÇÃO

Este teste segue rigorosamente o padrão **Arrange-Act-Assert (AAA)** para validar a precisão do cálculo do valor total de um pedido após a remoção de um item. Ele garante que a lógica de negócio para a atualização de preços seja robusta e livre de erros.

```
[Fact]
public void Total_AposRemocao_CalculaValorCorreto()
{
    // Arrange
    var order = new OrderEntity();
    order.AddItem("PROD-001", 3.0, new Money(50.00m));  // 3 x 50 = 150
    order.AddItem("PROD-002", 2.0, new Money(100.00m)); // 2 x 100 = 200
    // Total inicial esperado: 350

    // Act
    order.RemoveItem("PROD-001", 1.0); // Remove 1 unidade do PROD-001
    // Novo total esperado: (3-1) x 50 + 2 x 100 = 100 + 200 = 300

    // Assert
    Assert.Equal(300.00m, order.Total.Value);
}
```

ARRANGE (PREPARAR)

Nesta fase, simulamos um cenário de pedido inicial. Criamos um novo objeto `Order` e adicionamos dois itens distintos para estabelecer um estado inicial conhecido e calculável:

- **PROD-001:** 3 unidades ao preço de R$ 50,00 cada, totalizando R$ 150,00.
- **PROD-002:** 2 unidades ao preço de R$ 100,00 cada, totalizando R$ 200,00.

O total inicial esperado para este pedido é de R$ 350,00.

ACT (AGIR)

Aqui, executamos a ação que desejamos testar. Para este cenário, removemos uma unidade do "PROD-001". A expectativa é que a quantidade deste item seja reduzida para 2 unidades, mantendo o preço unitário de R$ 50,00.

ASSERT (VERIFICAR)

Finalmente, verificamos se o sistema se comportou conforme o esperado. Confirmamos que o valor total do pedido foi recalculado corretamente após a remoção. Com 2 unidades de "PROD-001" (R$ 100,00) e 2 unidades de "PROD-002" (R$ 200,00), o total esperado é R$ 300,00. Este teste garante que a propriedade `Total` reflete o estado atualizado do pedido.

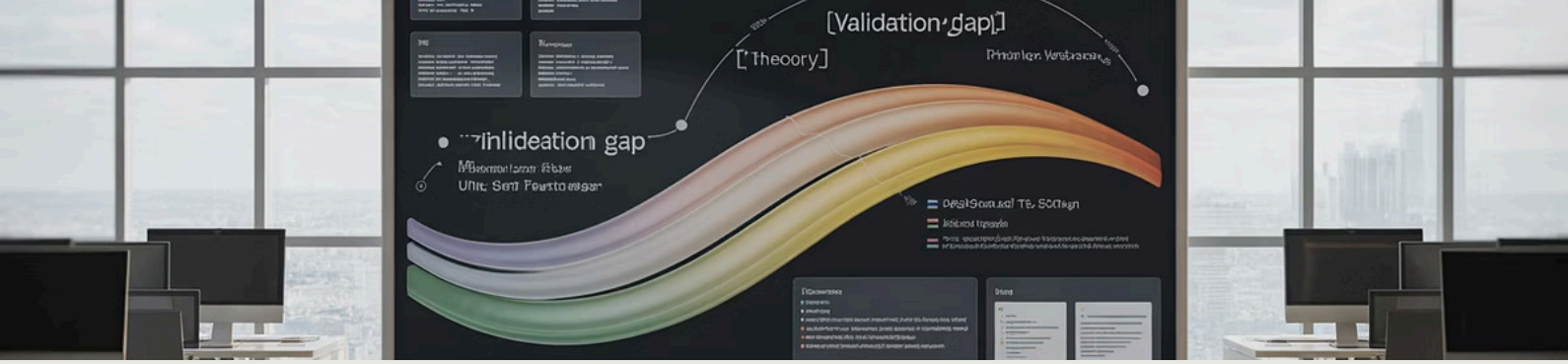
CONCEITOS FUNDAMENTAIS EVIDENCIADOS

Este teste não apenas valida a funcionalidade, mas também sublinha conceitos cruciais no design de software:

PROPRIEDADE CALCULADA (COMPUTED PROPERTY)

A propriedade `Total` não é um valor armazenado diretamente, mas sim calculada dinamicamente sempre que é acessada. Este teste assegura que:

- O cálculo sempre considera todos os itens presentes no pedido.
- A remoção de itens afeta o total de forma precisa.
- Não existem valores "cacheados" ou desatualizados, garantindo a consistência.



TESTE DE INTEGRAÇÃO ENTRE MÉTODOS

Este teste vai além de validar um único método isoladamente. Ele confirma a integração e o comportamento esperado entre múltiplos métodos e propriedades:

- AddItem(): Sua interação na adição de itens.
- RemoveItem(): Sua função na remoção ou decremento de itens.
- Total: Sua responsabilidade em calcular o valor total agregado.

A IMPORTÂNCIA ESTRATÉGICA DESTE TESTE

Testes como este são pilares para a construção de sistemas robustos e confiáveis, oferecendo múltiplos benefícios:

VALIDAÇÃO DE CÁLCULOS FINANCEIROS

Garanti a precisão em valores monetários, evitando discrepâncias financeiras que poderiam ter sérias implicações comerciais e legais.

CONSISTÊNCIA DE ESTADO

Confirma que o estado interno do pedido está sempre correto e reflete as operações realizadas, prevenindo dados inconsistentes.

DETECÇÃO DE REGRESSÃO

Atua como uma rede de segurança, detectando se futuras alterações no código quebram o cálculo do total, mesmo em outras partes do sistema.

DOCUMENTAÇÃO VIVA

Serve como uma especificação executável, demonstrando claramente como o total deve se comportar sob diferentes cenários, sendo útil para desenvolvedores.

Considere um cenário de negócio real, como um carrinho de compras: um cliente adiciona 3 camisetas (R$ 50 cada) e 2 calças (R$ 100 cada), resultando em um subtotal de R$ 350. Se o cliente remove 1 camiseta, o novo total deve ser R$ 300. Este teste garante que tal cenário funcione perfeitamente, refletindo a lógica de negócio esperada.

A PODEROSA COMBINAÇÃO DE [THEORY] E [INLINEDATA] NO XUNIT

O problema pedagógico e prático que buscamos resolver é o clássico "buraco de validação". Muitos métodos inadvertidamente aceitam strings inválidas (nulas, vazias ou contendo apenas espaços), o que frequentemente passa despercebido durante testes de cenários "felizes". As consequências aparecem mais tarde: dados corrompidos no domínio, erros difíceis de rastrear e regras de negócio que "quebram" silenciosamente, resultando em estados inconsistentes.

Para mitigar isso, é essencial executar o mesmo teste contra **variações do mesmo cenário-limite (boundary cases)**. Isso garante que as **guard clauses** rejeitem entradas inválidas de forma consistente e antecipada, implementando uma estratégia `fail-fast` que impede a persistência de dados problemáticos.

POR QUE USAR?

Para cobrir, de forma **DRY (Don't Repeat Yourself) e sistemática**, os casos-limite mais críticos (`null`, `""` e `" "`) e evitar regressões na validação de entradas.

O QUE FAZ?

O xUnit **invoca o mesmo método de teste repetidamente**, injetando cada valor de `[InlineData]` no parâmetro do método e verificando o comportamento esperado, como o lançamento de uma `ArgumentException`.

BENEFÍCIO DIRETO:

Aumenta a cobertura de fronteira, reduz o código repetido, documenta claramente a intenção de validação e torna os erros visíveis desde as fases iniciais do desenvolvimento.

Com `[Theory]` e `[InlineData]`, garantimos, sem duplicar código, que nossa validação falha-cedo funcione para todas as variações relevantes de entradas inválidas, como demonstrado no exemplo esquemático a seguir:

```
[Theory]
[InlineData(null)]
[InlineData("")]
[InlineData("   ")]
public void CtorDeveRejeitarNomeInvalido(string invalidName)
{
   // Act + Assert
   Assert.Throws<ArgumentException>(() => new Customer(invalidName));
}
```

Se algum desses casos de teste "passar" indevidamente, o sistema de testes sinaliza imediatamente que nossa borda de validação está vulnerável. Essa abordagem é crucial para identificar e corrigir proativamente falhas que poderiam comprometer a integridade do domínio.

COBERTURA ABRANGENTE DO CHECKLIST DE TESTES

Para garantir a robustez e a integridade de nossas associações, é crucial que os testes unitários cubram não apenas os cenários de uso feliz, mas também as condições de fronteira, entradas inválidas e "armadilhas" comuns no desenvolvimento. O checklist a seguir detalha os pontos essenciais a serem verificados, assegurando que o comportamento de classes como `OrderItem` e `Order` esteja em conformidade com as invariantes de negócio e princípios de design.

∅

`ADDITEM_QUANDOQUANTITYMENOROUIGUALZERO_DEVELANCAREXCECAO`

Garante que a adição de um item com quantidade menor ou igual a zero falhe, mantendo a consistência do estoque.

```
[Fact]
public void AddItem_DeveFalhar_QuandoQuantityMenorOuIgualZero()
{
    var order = new Order();
    Assert.Throws<ArgumentOutOfRangeException>(() =>
        order.AddItem("ABC", 0, new Money(10)));

    Assert.Throws<ArgumentOutOfRangeException>(() =>
        order.AddItem("ABC", -1, new Money(10)));
}
```

∅

`ADDITEM_QUANDOSKUVAZIOOUNULO_DEVELANCAREXCECAO`

Verifica que SKUs vazios ou nulos são rejeitados, prevenindo itens sem identificação válida no pedido.

```
[Theory]
[InlineData(null)]
[InlineData("")]
[InlineData("   ")]
public void AddItem_DeveFalhar_QuandoSkuVazioOuNulo(string? sku)
{
   var order = new Order();
   Assert.Throws<ArgumentException>(() => order.AddItem(sku!, 1,
      new Money(10)));
}
```

ADDITEM_QUANDOSKUVAZIOOUNULO_DEVELANCAREXCECAO

Verifica que SKUs vazios ou nulos são rejeitados, prevenindo itens sem identificação válida no pedido.

```
Fact]
public void Total_DeveSerPreciso_ComDecimais()
{
    var order = new Order();
    order.AddItem("A", 1, new Money(10.10m)); // 10,10
    order.AddItem("B", 3, new Money(0.99m));  // 2,97
    Assert.Equal(13.07m, order.Total.Value);  // 10,10 + 2,97
}
```

TOTAL_COMDECIMAIS_DEVESERPRECISO

Confirma que o cálculo do valor total do pedido mantém a precisão necessária para valores monetários, utilizando tipos adequados como `decimal`.

```
[Fact]
public void RemoveItem_DeveRetornarFalse_ParaSkuInexistente_EManterTotal()
{
   var order = new Order();
   order.AddItem("X", 2, new Money(5m)); // 10
   var totalAntes = order.Total.Value;
   var removed = order.RemoveItem("NAO-EXISTE");
   Assert.False(removed);
   Assert.Equal(totalAntes, order.Total.Value);
}
```

REMOVEITEM_PARASKUINEXISTENTE_RETORNAFALSE_EMANTERTOTAL

Assegura que a tentativa de remover um item com um SKU não existente retorne `false` e não altere o total do pedido nem cause efeitos colaterais indesejados.

```
[Fact]
public void ColecaoNaoPodeSerExposta_ComoListMutavel()
{
   var order = new Order();
   order.AddItem("X", 1, new Money(1));
   var items = order.Items;

   Assert.IsAssignableFrom<System.Collections.Generic.
```

```
  Assert.False(items is System.Collections.Generic.List<OrderItem>);
}
```

COLECAOITEMS_NAODEVESEREXPOSTAOUTERSETPUBLICO

Verifica que a coleção interna de itens do pedido não pode ser modificada diretamente por fora, protegendo as invariantes do agregado.

```
[Fact]
public void PropriedadeItems_NaoPossuiSetPublico()
{
  var prop = typeof(Order).GetProperty("Items");
  Assert.NotNull(prop);
  Assert.False(prop!.CanWrite);
}
```

TOTALDERIVADO_REFLETEMUDANCASSEMCACHE

Confirma que a propriedade `Total` sempre reflete o estado atual dos itens, sem depender de caches desatualizados.

```
[Fact]
public void TotalDerivado_DeveRefletirMudancasSemCache()
{
  var order = new Order();
  order.AddItem("A", 2, new Money(10)); // 20
  order.AddItem("B", 1, new Money(5));  // 5
  Assert.Equal(25m, order.Total.Value);

  order.RemoveItem("B");
  Assert.Equal(20m, order.Total.Value);
}
```

DECREASE_VALIDACAOPARAMETROS_LANCAEXCECAO

Testa se o método `Decrease` valida corretamente seus parâmetros, lançando exceção para valores inválidos de redução de quantidade.

```
[Fact]
public void Decrease_DeveValidarParametros_ERespeitarLimites()
{
```

```
    var order = new Order();
    order.AddItem("A", 3, new Money(10));
    var item = order.Items.First(i => i.Sku == "A");
    Assert.Throws<ArgumentOutOfRangeException>(() => item.Decrease(0));
     // igual à quantidade
    Assert.Throws<InvalidOperationException>(() => item.Decrease(3));
    // maior que a quantidade
    Assert.Throws<InvalidOperationException>(() => item.Decrease(4));
}
```

ENCERRAMENTO — ASSOCIAÇÕES SÓLIDAS, TESTES COMO CONTRATO E PRÓXIMOS PASSOS

A aderência a este checklist é fundamental para a criação de um sistema robusto e manutenível, onde cada componente se comporta de forma previsível e segura, especialmente em cenários que envolvem as complexidades das associações e transações financeiras.

Encerramos este estudo elevando o padrão do nosso modelo de domínio: saímos do código que “apenas compila” para um desenho que **protege invariantes**, **sincroniza associações** e **documenta comportamento via testes**. Consolidamos um **checklist de fronteiras** (quantidades ≤ 0, SKU nulo/vazio, precisão monetária, remoção de inexistentes) que previne estados inválidos e guia refatorações seguras, sempre com foco em encapsular coleções e manter o total como **valor derivado e consistente**.

No plano das **associações**, destacamos a robustez de relacionamentos 1:N (como Departamento→Funcionários) com **sincronização automática** em um único ponto de mutação — adicionar/remoção de um lado atualiza o outro — eliminando inconsistências de navegação. Essa disciplina estrutural é sustentada por testes claros no padrão **AAA**, cobrindo desde a **adição de itens novos** e a **remoção de inexistentes sem efeitos colaterais**, até a verificação de **totais recalculados** após mudanças. Além disso, adotamos **[Theory]/[InlineData]** para fechar o “buraco de validação” de strings nulas, vazias ou em branco com mínima duplicação e máxima cobertura de fronteira.

MANTRA DE PROJETO

VALIDAR CEDO

Valide cedo e com mensagens claras, garantindo que os dados inseridos no domínio sejam sempre consistentes.

ENCAPSULAR COLEÇÕES

Não exponha coleções mutáveis nem utilize `set` público em propriedades que representam associações, protegendo a integridade interna.

DERIVAR TOTAIS

Derive totais e outros valores a partir do estado atual dos objetos, sem caches desatualizados, e teste cada regra que protege o domínio.

PRÓXIMOS PASSOS SUGERIDOS

01

TESTES PARA ASSOCIAÇÕES BIDIRECIONAIS E N:M

Estenda a bateria de testes para cobrir associações bidirecionais (garantindo consistência em ambos os lados) e cenários N:M com classes de associação, incluindo regras de negócio específicas no vínculo.

02

CENÁRIOS DE DECREMENTO INVÁLIDO

Exercite cenários de decremento inválido (ex.: tentar reduzir uma quantidade além do disponível), garantindo que o sistema lance exceções adequadas com mensagens didáticas e úteis.

03

AMPLIAR O USO DE THEORY/INLINEDATA

Amplie o uso de `[Theory]`/`[InlineData]` para outros limites de validação (espaços em branco, formatação, faixas numéricas), mantendo a suíte de testes como documentação viva do seu domínio.

Com esse arsenal conceitual e prático, você tem as peças certas para evoluir o modelo com confiança: **associações sincronizadas, invariantes explícitas e testes que convertem as regras do negócio em contrato executável**.