ASSOCIAÇÕES EM C# — LABORATÓRIOS GUIADOS

Sequência didática e prática para consolidar associações após conceitos básicos de 0..1 e 1..1. Esta continuação dos laboratórios anteriores oferece uma abordagem estruturada com objetivos claros, padrões de código minimalistas e verificações práticas para dominar as associações em C#.

ESTRUTURA DOS LABORATÓRIOS

01

OBJETIVO CLARO

Cada laboratório define metas específicas de aprendizagem

02

ESSÊNCIA CONCEITUAL

Fundamentos teóricos explicados de forma concisa

03

PADRÕES DE CÓDIGO

Implementações C# minimalistas e práticas

04

LAB GUIADO

Passos detalhados com verificações constantes

05

TESTES DE FRONTEIRA

Validação de casos extremos e situações limite

06

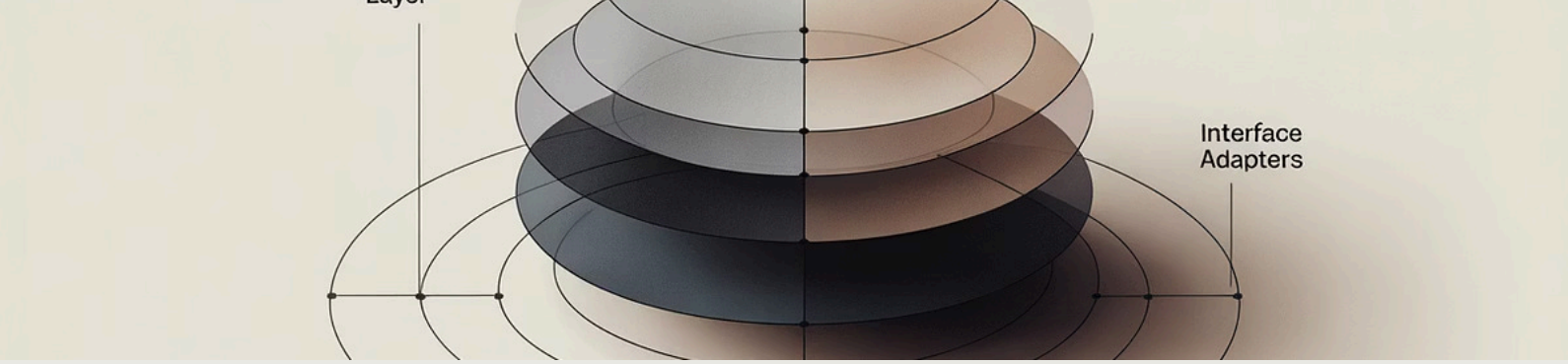
ARMADILHAS COMUNS

Erros frequentes e como evitá-los

1:N UNIDIRECIONAL - COMPOSIÇÃO

Objetivo: Implementar um agregado simples (Order → OrderItem) com controle total da coleção, cálculo automático de totais e regras rigorosas de integridade.

Essência: "O todo é dono do ciclo de vida das partes". A multiplicidade Order 1 para * OrderItem permite navegação apenas do todo para as partes, garantindo controle total sobre o relacionamento.



ESTRUTURA DE PROJETOS: DIVIDINDO RESPONSABILIDADES

A organização de um projeto em múltiplas camadas é uma prática fundamental no desenvolvimento de software moderno, especialmente em sistemas complexos. Essa abordagem não apenas melhora a manutenibilidade e a testabilidade, mas também otimiza a colaboração em equipe e a escalabilidade do sistema.

Compreender o porquê dessa divisão e a estrutura lógica mais comum é crucial para construir aplicações robustas e de fácil evolução. A seguir, exploramos os principais benefícios e a arquitetura recomendada.

SEPARAÇÃO DE RESPONSABILIDADES

Cada projeto gerencia uma parte específica do sistema (domínio, aplicação, UI/API), minimizando o acoplamento e o ruído entre as camadas.

REUSO E EVOLUÇÃO

Facilita o compartilhamento do domínio entre diferentes aplicações (API, console, testes) sem duplicação de código, e suporta a evolução independente das camadas.

ESCALABILIDADE ORGANIZACIONAL

Permite que equipes trabalhem em camadas distintas com menos conflitos, otimizando o desenvolvimento em ambientes maiores.

BUILD E DEPLOY SELETIVOS

Otimiza processos de compilação e deploy, permitindo que apenas os componentes alterados sejam testados e publicados, agilizando o ciclo de entrega.

QUALIDADE E SEGURANÇA

Isola os testes em projetos separados, prevenindo modificações indevidas no código produtivo e garantindo a integridade das regras de negócio.

ESTRUTURA LÓGICA COMUM DE PROJETOS

A arquitetura de camadas é um padrão consolidado que estabelece uma hierarquia clara de dependências, garantindo que o coração da aplicação – o domínio – permaneça intocado por detalhes de infraestrutura ou interface.

.API / .CONSOLE / .WORKER

Camadas de entrada (controladores REST, CLI, jobs assíncronos) que interagem com o usuário ou outros sistemas. Fazem a orquestração final, referenciando a camada de aplicação.

.APPLICATION

Contém a lógica de negócio específica para casos de uso (use cases), orquestrando o domínio. Define interfaces (ports) para gateways externos e depende do domínio.

.INFRASTRUCTURE

Responsável pelas implementações das interfaces definidas em `.Application`, como acesso a banco de dados (EF Core), serviços de e-mail ou clientes HTTP. Depende da camada de aplicação.

.DOMAIN

O núcleo do sistema: entidades, Value Objects, regras de negócio e invariantes. Não deve possuir dependências externas "pesadas", sendo a base que todas as outras camadas referenciam.

.TESTS

Projetos de testes (unitários, de integração) que validam o comportamento das outras camadas. Referenciam diretamente os projetos que estão sendo testados.

A "Regra de Ouro" dessa arquitetura é clara: as dependências sempre apontam para dentro. Ou seja, as camadas externas (UI, Infraestrutura) dependem das camadas internas (Aplicação, Domínio), mas o Domínio nunca deve ter conhecimento das camadas que o utilizam.

ORGANIZAÇÃO FÍSICA: PADRÃO DE PASTAS

A organização física do código dentro do sistema de arquivos é crucial para a clareza, manutenibilidade e escalabilidade do projeto. Um padrão bem definido garante que novos desenvolvedores possam entender rapidamente a estrutura e que as responsabilidades de cada módulo sejam distintas. Geralmente, seguimos uma estrutura de pastas que espelha as camadas lógicas discutidas anteriormente, separando explicitamente o código produtivo dos testes.

RAIZ DA SOLUÇÃO

Contém as pastas /src (para o código de produção) e /tests (para os projetos de teste).

/SRC

Agrupa os projetos principais, como Project.Domain, Project.Application, Project.Infrastructure e as interfaces de usuário/serviço (Project.Api, Project.Console ou Project.Worker).

/TESTS

Contém os projetos de teste correspondentes, como Project.Domain.Tests e Project.Application.Tests. Isso mantém o código de teste e de produção rigorosamente separados.

Essa organização não apenas mantém o código produtivo e os testes separados, mas também escala excepcionalmente bem. Adicionar um novo *front-end* (como um `Project.Web`) ou um novo serviço (como `Project.BackgroundService`) é tão simples quanto criar outro projeto dentro da pasta `/src`, mantendo a consistência e a modularidade.

FLUXO DE REFERÊNCIAS ENTRE PROJETOS

As referências entre os projetos devem seguir um fluxo unidirecional estrito, apontando sempre para as camadas mais internas, garantindo que o domínio permaneça agnóstico em relação à infraestrutura e à apresentação.

API/CONSOLE/WORKER

Referencia `Project.Application` para orquestrar os casos de uso.

APPLICATION

Referencia `Project.Domain` para acessar entidades e regras de negócio.

INFRASTRUCTURE

Referencia `Project.Application` (para implementar interfaces de repositório, por exemplo) e bibliotecas externas (EF Core, etc.).

TESTS

Cada projeto de teste referencia diretamente o projeto alvo que está sendo testado (ex.: `Project.Domain.Tests` referencia `Project.Domain`).

Evite referências “para trás” (ex.: Domain apontando para Infrastructure). Se precisar que o domínio interaja com o "mundo externo", exponha interfaces em Application e implemente-as em Infrastructure, seguindo o Princípio da Inversão de Dependência.

QUANDO CRIAR UM NOVO PROJETO NA SOLUÇÃO?

A decisão de criar um novo projeto em uma solução C# é fundamental para a arquitetura e a manutenibilidade do sistema. Não se trata apenas de organização de pastas, mas de uma separação lógica e física que reflete diferentes responsabilidades e ciclos de vida. Projetos distintos impõem limites claros sobre o que pode ser referenciado e por quem, garantindo que as camadas internas permaneçam independentes de detalhes de infraestrutura ou apresentação.

BARREIRA DE MUDANÇA DISTINTA

Crie um novo projeto quando um conjunto de funcionalidades muda em um ritmo diferente ou por razões distintas do restante do sistema, minimizando o impacto em outras partes do código.

DEPENDÊNCIAS ESPECÍFICAS

Se uma parte do código requer dependências pesadas ou específicas (como Entity Framework, Dapper, ou clientes HTTP), isolá-la em um projeto separado evita que essas dependências "vazem" para o domínio ou outras camadas, mantendo o controle sobre o acoplamento.

REUSO REAL E TESTABILIDADE

Quando a mesma lógica de negócio ou conjunto de entidades precisa ser reutilizada por múltiplas interfaces de usuário (API REST, aplicação de console, worker service) ou para facilitar testes unitários puros, um projeto dedicado para o domínio se torna essencial.

Contudo, é crucial balancear essa modularização. Um excesso de projetos pequenos e sem propósito claro pode adicionar complexidade desnecessária à solução. O ideal é começar de forma enxuta (com `.Domain` e `.Tests`) e introduzir `.Application` e `.Infrastructure` à medida que a necessidade de orquestração e abstração se torna evidente.

BENEFÍCIOS DA MODULARIZAÇÃO COM PROJETOS

- **Didática Aprimorada:** Facilita a compreensão de onde cada responsabilidade (regra de negócio vs. endpoint) deve residir.
- **Pull Requests Menores:** Mudanças focadas em um único projeto resultam em revisões de código mais rápidas e seguras.
- **Pipelines Simplificados:** Permite compilação e teste granular (`dotnet build/test` por projeto/camada), otimizando o CI/CD.
- **Governança Eficaz:** Aplicação consistente de analisadores de código, cobertura de testes e políticas de desenvolvimento por projeto.

ARMADILHAS COMUNS E COMO EVITÁ-LAS

"GOD PROJECT" (PROJETO ÚNICO)

Quando todo o código reside em um único projeto, testes e reuso se tornam desafiadores. **Solução:** Quebre a solução em, no mínimo, `.Domain` e `.Tests` para isolar a lógica de negócio.

DEPENDÊNCIAS INVERTIDAS

O projeto de domínio importa bibliotecas de infraestrutura (ex: EF, HTTP). **Solução:** Mova integrações para `.Infrastructure` e exponha contratos (interfaces) em `.Application`, aderindo ao Princípio da Inversão de Dependência.

ORGANIZAÇÃO APENAS POR PASTA

Separar por pastas sem projetos distintos não impede as mesmas dependências problemáticas. **Solução:** Use projetos separados quando a barreira de mudança e as dependências justificam a separação.

ACESSO A "INTERNALS" PELOS TESTES

Se os testes precisam acessar membros internos do código de produção. **Solução:** Revise o encapsulamento ou use `InternalsVisibleTo` com muita parcimônia, priorizando testar o comportamento público.

ESTRATÉGIA E INDICADORES DE ARQUITETURA SAUDÁVEL

A adoção de uma abordagem incremental na evolução da arquitetura de software é crucial para manter a agilidade e a manutenibilidade. Começar com uma estrutura enxuta e expandir apenas quando a complexidade justifica a adição de novos projetos ou camadas evita a superengenharia e permite que a solução cresça organicamente com as necessidades do negócio. Essa estratégia garante que cada componente adicionado tenha um propósito claro e uma responsabilidade bem definida, otimizando o desenvolvimento e a gestão do projeto.

ESTRATÉGIA DE EVOLUÇÃO INCREMENTAL

INÍCIO ENXUTO

Comece com os projetos essenciais: `Project.Domain` para a lógica de negócio e `Project.Domain.Tests` para sua validação. Opcionalmente, um `Project.Console` pode ser incluído para demonstrações rápidas.

CRESCIMENTO DA COMPLEXIDADE

Quando a aplicação exige orquestração de casos de uso e separação de responsabilidades da UI, adicione `Project.Application`. A interface do usuário passará a chamar serviços definidos aqui.

INTEGRAÇÕES EXTERNAS

Surgindo a necessidade de interagir com bancos de dados, APIs externas ou outros serviços, crie `Project.Infrastructure`. Este projeto implementará as interfaces (ports) definidas nas camadas mais internas.

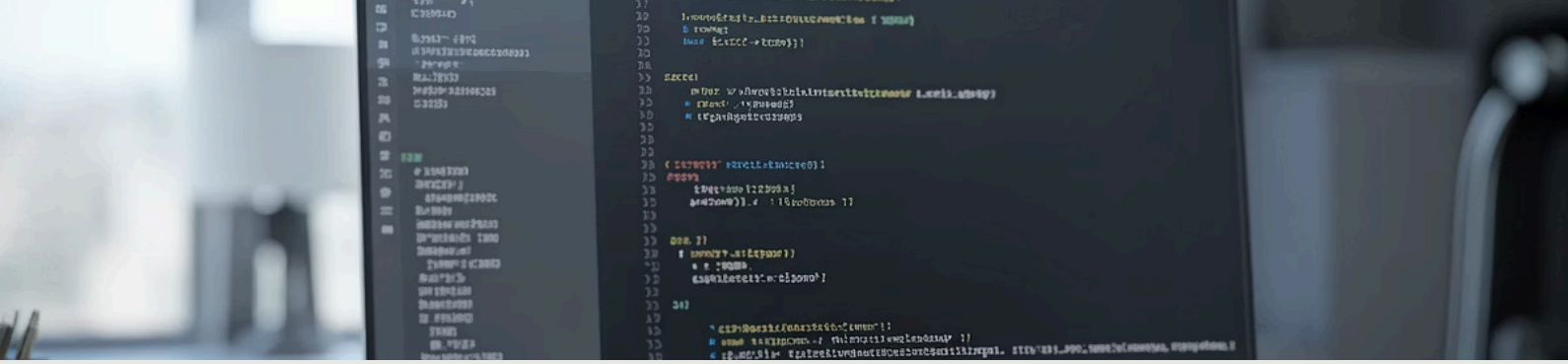
EXPOSIÇÃO HTTP

Para expor funcionalidades via HTTP, como uma API REST, inclua `Project.Api`. Ele se conectará diretamente ao `Project.Application` para executar os casos de uso.

CI/CD E GOVERNANÇA EM POUCAS LINHAS

A automação de processos de Integração Contínua (CI) e Entrega Contínua (CD) é um pilar fundamental para garantir a qualidade e a segurança do código em arquiteturas modulares. A governança eficaz é construída sobre a execução automatizada de testes, a verificação de cobertura e a aplicação de padrões de código rigorosos, protegendo a integridade da base de código.

- **Testes Granulares:** Rodar `dotnet test` especificamente para cada projeto de testes (`Domain.Tests`, `Application.Tests`, etc.) acelera o feedback e isola falhas.
- **Cobertura de Código:** Utilizar ferramentas como Coverlet/ReportGenerator para definir e monitorar metas de cobertura, com alvos específicos por camada (ex: `Domain` ≥ 90%).
- **Analisadores de Código:** Aplicar analisadores (Roslyn/StyleCop) nos projetos de produção para garantir consistência de estilo e boas práticas desde o início.
- **Proteção de Branch:** Implementar proteção de branch que exija um build bem-sucedido e a aprovação de todos os testes antes de permitir o merge de código.



SINAIS DE UMA ARQUITETURA SAUDÁVEL

Uma arquitetura bem-sucedida se manifesta através de indicadores claros que refletem sua flexibilidade, testabilidade e separação de preocupações. Esses sinais são cruciais para avaliar a robustez e a adaptabilidade do sistema.

DOMÍNIO ISOLADO

Você consegue rodar e testar o domínio (`Project.Domain`) sem a necessidade de um banco de dados, servidor web ou qualquer infraestrutura externa.

UI PLUGÁVEL

Trocar a interface do usuário (ex: de uma API para um aplicativo de console) não impacta ou exige mudanças na lógica de domínio.

DEPENDÊNCIAS PROTEGIDAS

Todas as dependências externas e detalhes de implementação (como ORMs ou APIs de terceiros) estão confinados e isolados no `Project.Infrastructure`.

TESTES COMO DOCUMENTAÇÃO

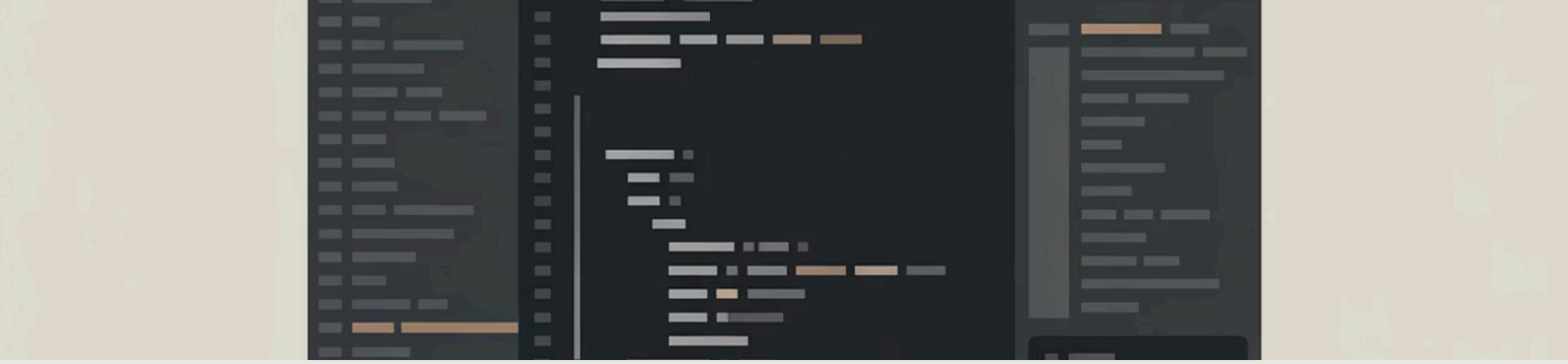
Seus testes unitários servem como uma documentação viva e atualizada dos contratos e comportamentos esperados do domínio.

CONFIGURAÇÃO INICIAL DO AMBIENTE

Para iniciar nossos laboratórios práticos, o primeiro passo é configurar a estrutura básica do projeto. Isso garante uma organização padronizada e facilita o gerenciamento dos diferentes componentes (código-fonte, testes e aplicativos) em uma única solução .NET.

```
mkdir associations && cd associations
mkdir src tests apps
dotnet new sln -n Associations
```

Os comandos acima criam uma pasta raiz chamada `associations`, navega para ela, e em seguida estabelece três subdiretórios principais: `src` para o código-fonte principal das bibliotecas e domínios, `tests` para os projetos de testes unitários e de integração, e `apps` para qualquer aplicação executável (console, web, etc.) que possa ser construída a partir do domínio. Finalmente, `dotnet new sln -n Associations` cria um arquivo de solução .NET, que agrupa todos esses projetos em um único contexto de desenvolvimento.



CRIAÇÃO E CONFIGURAÇÃO DO PROJETO DE DOMÍNIO

O primeiro passo prático é estabelecer o coração da nossa aplicação: o projeto de domínio. Este projeto, denominado `Associations.Domain`, será uma **Class Library**. Uma Class Library é um tipo de projeto .NET que compila seu código em um arquivo `.dll` (Dynamic Link Library). Isso permite que o código seja reutilizado facilmente por outras aplicações (como testes, APIs web ou aplicativos de console) sem duplicidade, mantendo a lógica de negócios centralizada e independente de qualquer camada de apresentação ou persistência específica.

```
dotnet new classlib -n Associations.Domain -o src/Associations.Domain --framework net8.0
```

Após a criação, é fundamental ajustar o arquivo `.csproj` do novo projeto. Essas configurações são boas práticas no desenvolvimento C# moderno, projetadas para aumentar a robustez do código e a produtividade do desenvolvedor.

```
<PropertyGroup>
  <TargetFramework>net8.0</TargetFramework>
  <Nullable>enable</Nullable>
  <ImplicitUsings>enable</ImplicitUsings>
</PropertyGroup>
```

1

NULLABLE (ENABLE)

Ativar a verificação de nulidade (`Nullable`) é uma medida crucial para prevenir `NullReferenceException` em tempo de execução. Com ela habilitada, o compilador C# passa a avisar sobre potenciais valores nulos não tratados, incentivando os desenvolvedores a fazerem verificações explícitas ou a projetarem seu código de forma a evitar nulos inesperados, resultando em software mais seguro e com menos bugs.

2

IMPLICIT USINGS (ENABLE)

A funcionalidade `ImplicitUsings` simplifica os arquivos de código, automaticamente adicionando diretivas `using` comuns (`System.Collections.Generic`, `System.Linq`, etc.) para o tipo de projeto. Isso reduz a "poluição" de boilerplate no topo de cada arquivo, tornando o código mais limpo e focado na lógica de negócio essencial, sem sacrificar a clareza.

PADRÕES DE CÓDIGO

CLASSE MONEY

```
namespace Associations.Domain
{
    /// <summary>
    /// Value Object de dinheiro — invariantes do domínio aplicados aqui
    /// para falhar cedo (fail-fast) em valores inválidos e manter
    /// operações previsíveis.
    /// </summary>
    public sealed class Money
    {
        /// <summary>
        /// Valor monetário imutável (usar sempre decimal para evitar erros
        /// de ponto flutuante).
        /// Imutável por desenho: uma vez criado, não muda.
        /// </summary>
        public decimal Value { get; }

        /// <summary>
        /// Regra de negócio: dinheiro negativo não é válido neste domínio.
        /// Lançamos exceção para denunciar o bug na origem (fail-fast) em
        /// vez de “mascarar” o erro.
        /// </summary>
        public Money(decimal value)
        {
            if (value < 0)
                throw new ArgumentOutOfRangeException(nameof(value),
                  "Money não pode ser negativo.");
            Value = value;
        }

        /// <summary>
        /// Operação pura: retorna um novo Money com a soma dos valores.
        /// Os operandos originais permanecem inalterados (sem efeitos
        /// colaterais).
        /// </summary>
        public static Money operator +(Money a, Money b) =>
          new(a.Value + b.Value);
```

```
        /// <summary>
        /// Opcional: multiplicação por quantidade inteira para facilitar
        /// subtotais (ex.: preço * quantidade).
        /// Mantém a imutabilidade e evita lógica repetida fora do VO.
        /// </summary>
        public static Money operator *(Money a, int qty)
        {
            if (qty < 0)
                throw new ArgumentOutOfRangeException(nameof(qty),
                    "Quantidade não pode ser negativa.");
            return new Money(a.Value * qty);
        }

        /// <summary>
        /// Igualdade semântica baseada no valor monetário.
        /// Útil para asserts em testes e uso seguro em coleções.
        /// </summary>
        public override bool Equals(object? obj)
            => obj is Money other && Value == other.Value;

        public override int GetHashCode() => Value.GetHashCode();

        /// <summary>
        /// Representação amigável para logs/depuração (formatação simples;
        /// ajuste conforme cultura/moeda).
        /// </summary>
        public override string ToString() => Value.ToString("0.##");
    }
}
```

CONTEXTO DIDÁTICO NA CLASSE MONEY

A classe `Money`, inclui uma validação crucial que impede a criação de instâncias com valores negativos. Este ajuste não é meramente uma verificação de erro; ele serve como uma poderosa ferramenta pedagógica para demonstrar a importância dos invariantes de domínio e a técnica de "fail-fast" na arquitetura de software.

POR QUE ESSE AJUSTE EXISTE?

Em sistemas financeiros, um valor monetário negativo geralmente indica um erro lógico ou uma manipulação indevida (como um desconto aplicado incorretamente ou uma inversão de sinal). Se a classe `Money` permitisse silenciosamente esses valores, o problema se propagaria pelo sistema, contaminando cálculos de totais, relatórios financeiros e, eventualmente, decisões de negócio. Ao lançar uma exceção clara e imediata, o erro é detectado próximo à sua origem, tornando a depuração e a correção muito mais simples. Isso reforça a ideia de que regras de negócio fundamentais devem ser solidificadas como invariantes no código, protegendo a integridade do modelo desde o princípio.

O QUE O AJUSTE GARANTE?

INVARIÂNCIA RIGOROSA

Assegura que nenhuma instância de `Money` seja criada com um valor inferior a zero, mantendo a integridade dos dados financeiros em todo o sistema.

MENSAGENS PEDAGÓGICAS

As exceções são formuladas com clareza ("Money não pode ser negativo"), indicando o parâmetro violado. Isso transforma o erro em uma oportunidade de aprendizado para desenvolvedores.

IMUTABILIDADE PREVISÍVEL

Operações como a soma (`operator +`) resultam em uma nova instância de `Money`, preservando os operandos originais. Isso promove boas práticas e evita "efeitos colaterais invisíveis".

PRECISÃO MONETÁRIA

O uso de `decimal` em vez de `double` ou `float` protege contra imprecisões de ponto flutuante, fundamentais para cálculos de preços e totais.

EFEITOS POSITIVOS NO RESTANTE DO MATERIAL

- **OrderItem / Order:** Estas classes se beneficiam diretamente. O `Subtotal` e o `Total` permanecem derivados e confiáveis, pois são blindados contra a possibilidade de preços unitários inválidos.
- **Testes:** O ajuste fortalece didaticamente dois cenários de teste essenciais: a verificação de que a classe `Money` não aceita valores negativos e a validação de que as operações de soma são imutáveis. Isso proporciona exemplos práticos para os alunos exercitarem tanto a regra de negócio quanto o comportamento esperado do código.

> "Dinheiro negativo não existe no nosso domínio. Se aparecer, é bug anterior. O nosso Money acusa isso na hora, com mensagem clara, para que a correção seja feita na origem. Assim, os cálculos derivados (Subtotal/Total) continuam confiáveis."

APROFUNDANDO EM MONEY: VALUE OBJECTS, EXCEÇÕES E BOAS PRÁTICAS

Para consolidar a compreensão da classe `Money` e seus princípios, vamos explorar alguns conceitos fundamentais que ela exemplifica. Entender o papel de um Value Object, a distinção entre preocupações de domínio e de apresentação, e a taxonomia de exceções ajudará a construir um modelo de domínio robusto e expressivo.

POR QUE `MONEY` É UM VALUE OBJECT (E POR QUE ISSO IMPORTA)

`Money` é um clássico Value Object (VO). Em VOs, a **identidade é definida pelo seu estado (valor)**, e não por um identificador único. Dois objetos `Money` com o mesmo valor (ex: R$ 10,00) são considerados iguais, independentemente de onde foram criados ou manipulados. Isso simplifica comparações, facilita testes e evita a necessidade de "IDs artificiais" para dados que representam um valor conceitual.

Os Value Objects são pilares para garantir a integridade do domínio, encapsulando regras de negócio e mantendo a imutabilidade.

IMUTABILIDADE E COMPARAÇÃO POR VALOR

VOs são inerentemente imutáveis e comparáveis pelo seu conteúdo. Isso garante que, uma vez criados, seus valores não mudam e que a comparação de dois VOs se baseia em seus dados internos.

INVARIANTES EXPLÍCITOS

Eles tornam as regras de negócio intrínsecas ao objeto. No caso de `Money`, o invariante "não pode ser negativo" é uma regra fundamental protegida no construtor.

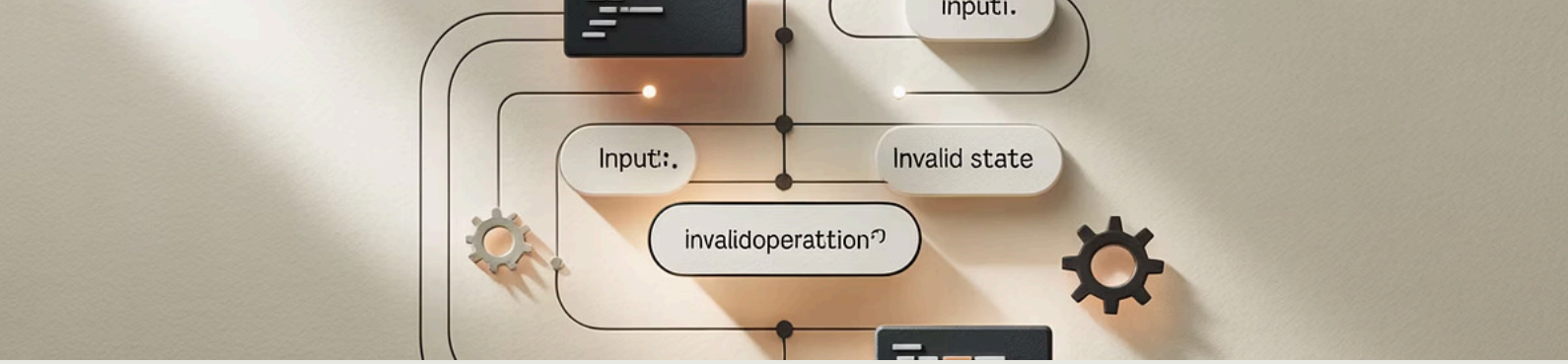
TESTABILIDADE FACILITADA

A comparação direta de igualdade por valor simplifica a escrita de testes, pois não é necessário orquestrar identidades ou mockar comportamentos complexos para validar o estado.

ARREDONDAMENTO, FORMATAÇÃO E CULTURA (O QUE *NÃO* ESTAMOS COBRINDO AQUI)

É crucial diferenciar o que pertence ao domínio do que é preocupação de apresentação ou infraestrutura. No contexto desta classe `Money` simplificada para aprendizado, focamos na representação pura do valor monetário.

- **Uso de `decimal` puro:** Implementamos `decimal` sem arredondamentos automáticos, mantendo a precisão exata do valor.



- **Formatação é Apresentação:** Símbolos de moeda, separadores de milhar/decimal, e configurações de cultura são responsabilidades da camada de UI ou de formatação, não do objeto de domínio `Money`. Ele se preocupa com o **valor**, não com a sua **exibição**.
- **Regras Fiscais e Arredondamento Complexo:** Regras de arredondamento específicas (bancário, casas decimais por moeda) e considerações fiscais variam por contexto de negócio. Elas serão introduzidas apenas quando a modelagem do sistema exigir, em estágios mais avançados do projeto.

Essa abordagem evita poluir o conceito de Value Object com decisões que pertencem a outras camadas do sistema, mantendo o foco nas regras de negócio essenciais.

EXCEÇÕES: QUANDO USAR QUAL (TAXONOMIA ENXUTA)

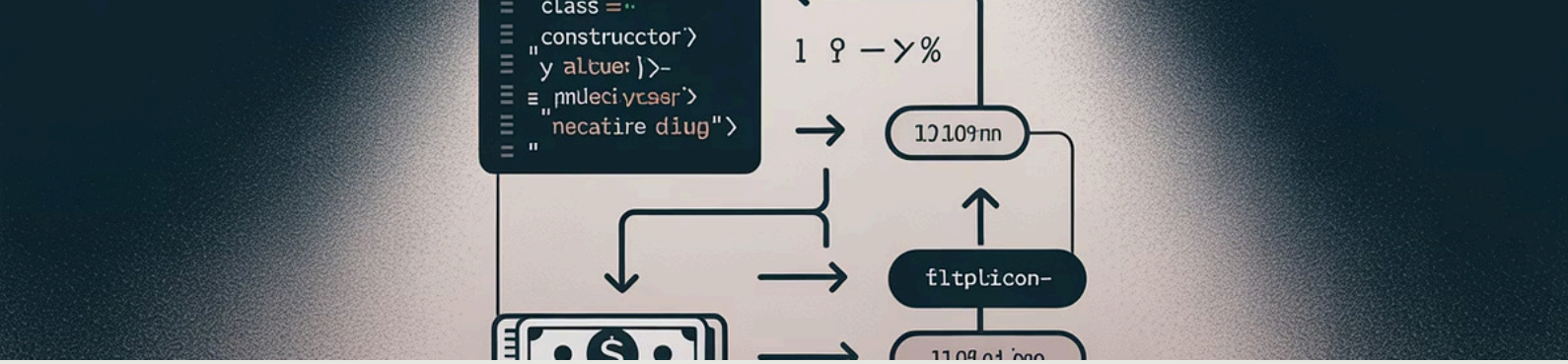
A escolha correta da exceção é crucial para comunicar de forma clara a natureza do problema, facilitando a depuração e garantindo que as regras de negócio sejam compreendidas e respeitadas. Adotaremos uma taxonomia mínima e consistente para o uso de exceções em nosso domínio.

Cada tipo de exceção tem um propósito específico, ensinando ao desenvolvedor se o problema está na **entrada de dados** ou no **estado do sistema**.

ARGUMENTEXCEPTION E VARIAÇÕES

Utilize quando o problema reside nos argumentos recebidos por um construtor ou método, indicando que a entrada fornecida é inválida.

- **Quando usar:** Valores numéricos fora do domínio permitido (ex: negativo, zero onde não pode), texto malformado ou em branco (ex: SKU vazio), ou referência nula passada para uma dependência obrigatória.
- **Qual escolher:**
  - `ArgumentOutOfRangeException` para faixas/intervalos inválidos (ex: quantidade ≤ 0).
  - `ArgumentException` para formato ou conteúdo inválido (ex: SKU com caracteres proibidos).
  - `ArgumentNullException` quando o erro for "nulo onde não pode".



INVALIDOPERATIONEXCEPTION

Use quando o problema não é a validade dos argumentos em si, mas o estado atual do objeto, indicando que a operação solicitada violaria um invariante de estado do objeto ou agregado.

- **Quando usar:** A operação viola um invariante de estado do agregado/entidade. Exemplos incluem tentar remover um item que não existe, diminuir um estoque abaixo de zero, ou mover um funcionário para o mesmo departamento.
- **Mensagem:** Estruture como "Operação inválida | Invariante afetada | Estado atual (opcional)". Exemplo: Não é possível remover item inexistente. Coleção deve permanecer consistente.

> **Regras rápidas:**
>
> Entrada inválida → `Argument*` (ensine a regra do parâmetro).
>
> Estado inválido → `InvalidOperationException` (explique a invariante quebrada).
>
> Mensagens devem ser curtas, afirmativas e pedagógicas, evitando jargões ou stack-traces falantes.

Essencialmente, se o problema pode ser validado antes mesmo de o objeto ser criado ou a operação iniciada (baseado nos parâmetros), é uma `Argument*`. Se só faz sentido validar depois de o objeto estar em um determinado estado, é uma `InvalidOperationException`.

`ARGUMENTOUTOFRANGEEXCEPTION`: VALIDANDO LIMITES NUMÉRICOS

Dando continuidade à nossa taxonomia de exceções, a `ArgumentOutOfRangeException` é uma variação específica da `ArgumentException`, utilizada quando um argumento numérico está fora de uma faixa ou limite permitido. No contexto do nosso `Value Object Money`, ela desempenha um papel crucial na proteção de invariantes numéricas, garantindo que operações financeiras e a criação de valores monetários sigam as regras de negócio.

Esta exceção é fundamental para comunicar claramente que um valor fornecido está além do que o sistema considera logicamente aceitável para aquele parâmetro, evitando estados inconsistentes e cálculos incorretos.

CONSTRUTOR DE `MONEY`: EVITANDO VALORES NEGATIVOS

O construtor da classe `Money` é o primeiro ponto de defesa contra a criação de instâncias com valores monetários inválidos. Conforme nossa regra de negócio, um valor monetário não pode ser negativo.

```
if (amount < 0) throw new ArgumentOutOfRangeException(nameof(amount), "Valor monetário não pode ser negativo.");
```

Ao tentar criar um `Money` com um valor como `-10.00m`, esta exceção é lançada, protegendo a integridade do dado desde sua origem.

OPERAÇÕES DE MULTIPLICAÇÃO: QUANTIDADES VÁLIDAS

Similarmente, o operador de multiplicação sobrecarregado para `Money` exige que a quantidade seja um número positivo. Multiplicar um valor monetário por uma quantidade negativa não faz sentido no domínio de negócio e levaria a resultados ambíguos.

```
if (qty < 0) throw new ArgumentOutOfRangeException(nameof(qty), "Quantidade não pode ser negativa.");
```

Assim, tentar calcular um subtotal como `money * -5` resultará nesta exceção, reforçando a regra de que quantidades devem ser maiores ou iguais a zero (fator/quantidade negativos).

A clareza das mensagens de erro, como `"Valor monetário não pode ser negativo. (param: amount)"` ou `"Quantidade não pode ser negativa. (param: qty)"`, orienta o desenvolvedor sobre qual parâmetro causou o problema e qual regra foi violada, tornando a depuração mais eficiente e o código mais robusto.

CONSTRUINDO MENSAGENS DE EXCEÇÃO CLARAS E EFICAZES

A qualidade das mensagens de exceção é um aspecto crucial para a manutenibilidade e debugabilidade de qualquer sistema. Mensagens bem formuladas não apenas indicam que algo deu errado, mas também orientam o desenvolvedor sobre a causa raiz e, idealmente, como corrigi-la. Elas servem como uma ponte entre o comportamento inesperado e a compreensão do invariante ou da regra de negócio violada.

Para garantir a clareza e utilidade de nossas exceções, seguimos um conjunto de diretrizes que transformam cada mensagem em uma pequena instrução sobre o uso correto ou o estado esperado do sistema. Isso minimiza o tempo gasto na depuração e fortalece a aderência aos princípios do design do domínio.

FALE A REGRA DE FORMA AFIRMATIVA

Comunique a regra de negócio violada de maneira direta e inequívoca. Use frases como "deve ser maior que" ou "não pode ser nulo", expressando claramente a expectativa. Isso evita ambiguidades e deixa evidente qual invariante foi desrespeitada.

Ex: "Valor monetário não pode ser negativo."

DIGA O PORQUÊ (QUANDO ÚTIL)

Sempre que a explicação adicionar valor, inclua o motivo pelo qual a regra existe. Mencionar que a operação "violaria um invariante" ou um "valor mínimo/derivado" ajuda a contextualizar a falha, educando o desenvolvedor sobre a lógica de domínio.

Ex: "Quantidade deve ser positiva para evitar subtotais inconsistentes."

APONTE O PARÂMETRO EM DESTAQUE

É fundamental que a mensagem de exceção indique qual parâmetro específico causou o problema. Adicionar " (param: nomeDoParametro)" no final da mensagem torna a depuração muito mais eficiente, direcionando rapidamente para a origem do erro.

Ex: "SKU vazio ou nulo. (param: sku)"

EVITE "NÚMEROS MÁGICOS" SEM CONTEXTO

Em vez de usar valores numéricos arbitrários, priorize expressões que contextualizem o limite. Prefira "maior que zero (> 0)" ou "não pode ser menor que um (>= 1)" a simplesmente "valor inválido", pois isso descreve a restrição com mais clareza.

Ex: "Quantidade não pode ser zero. (quantidade > 0)"

USE A CLASSE DE EXCEÇÃO CORRETA

Evite o uso de exceções genéricas (como `Exception`) e opte sempre pela classe de exceção mais específica (ex: `ArgumentOutOfRangeException`, `InvalidOperationException`). Isso fornece metadados valiosos sobre o tipo de problema, facilitando o tratamento programático e a leitura do código.

Ex: Em vez de `throw new Exception(...)` , use `throw new ArgumentException(...)`

Ao seguir estas diretrizes, transformamos as exceções de meros sinais de erro em ferramentas poderosas para comunicação e garantia da integridade do domínio, contribuindo para um código mais robusto e fácil de manter.

VERIFICAÇÃO RÁPIDA (2 MINUTOS)

Para fixar o contrato do Value Object `Money`, considere mentalmente (ou rapidamente em um console) os seguintes cenários:

TESTE 1 — INVARIANTE DO VALOR

Tentar criar uma instância de `Money` com um valor negativo (ex: `new Money(-10m)`) **deve falhar** imediatamente. A exceção lançada deve ser clara, indicando que "Money não pode ser negativo".

TESTE 2 — IMUTABILIDADE DA SOMA

Ao somar dois objetos `Money` (ex: `moneyA + moneyB`), os objetos `moneyA` e `moneyB` originais **não devem ser alterados**. A operação deve retornar uma **nova** instância de `Money` com o resultado da soma.

ERROS COMUNS (E COMO EVITÁ-LOS)

Ao trabalhar com valores monetários e Value Objects, alguns erros são recorrentes. Evitá-los desde cedo reforça a robustez do seu domínio:

- **"Clampar" Negativo para Zero:** Forçar um valor negativo para zero (`if (value < 0) value = 0;`) máscara o bug original. É sempre melhor **falhar cedo** com uma exceção, direcionando a correção para a fonte do erro.
- **Usar `double` ou `float`:** Tipos de ponto flutuante introduzem imprecisões que são inaceitáveis em cálculos financeiros. **Sempre use `decimal`** para valores monetários.
- **Misturar Formatação com Domínio:** Adicionar lógica de símbolos de moeda, casas decimais de exibição ou regras de cultura diretamente no Value Object. Esses detalhes pertencem à **camada de UI** ou de apresentação, não à lógica central do domínio.

TESTES UNITÁRIOS EM C# PARA ASSOCIAÇÕES

Este módulo é uma continuação direta dos laboratórios práticos sobre associações em C#, abordando temas como composição 1:N, bidirecionalidade, associações N:M com classes auxiliares, qualificadores e restrições de cardinalidade. Nosso objetivo é fornecer a fundamentação didática para a escrita de testes unitários eficazes, complementado por um projeto xUnit pronto que valida todos os cenários discutidos.

POR QUE TESTES UNITÁRIOS?

O objetivo pedagógico principal é transformar regras de modelagem de domínio — como multiplicidade, navegabilidade e invariantes de objeto — em comportamentos verificáveis através de testes. Os testes unitários atuam como a "tradução executável" do que o modelo de domínio promete e espera, garantindo a integridade e o funcionamento correto de nossas associações.

BENEFÍCIOS CHAVE DOS TESTES UNITÁRIOS

FEEDBACK RÁPIDO

Erros e violações de invariantes são identificados precocemente no ciclo de desenvolvimento, como coleções expostas, totais divergentes ou sincronização de associações quebrada.

CONFIANÇA PARA EVOLUIR

Permitem refatorar o código sem medo de introduzir regressões, facilitando a aplicação de padrões de composição/agregação e a modificação da navegabilidade das associações.

DOCUMENTAÇÃO VIVA

Os nomes descritivos dos testes funcionam como uma documentação executável, detalhando o contrato e o comportamento esperado das classes e associações.

PRINCÍPIOS ESSENCIAIS (F.I.R.S.T.)

Para que os testes unitários sejam realmente eficazes e sustentáveis, eles devem aderir a um conjunto de princípios fundamentais, frequentemente resumidos no acrônimo F.I.R.S.T.

FAST (RÁPIDOS)

Os testes devem ser executados rapidamente, permitindo feedback imediato aos desenvolvedores.

INDEPENDENT (INDEPENDENTES)

Cada teste deve ser autônomo, não dependendo da ordem de execução ou do estado de outros testes.

REPEATABLE (REPETÍVEIS)

Devem produzir o mesmo resultado cada vez que são executados, sendo determinísticos e sem dependências externas imprevisíveis.

SELF-VALIDATING (AUTO-VALIDÁVEIS)

Devem indicar claramente sucesso ou falha por meio de asserções, sem a necessidade de verificação manual de logs.

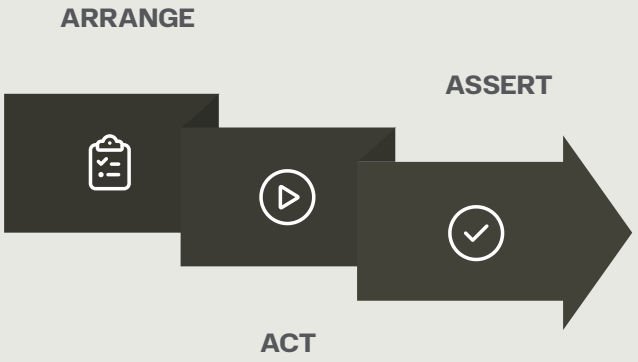
TIMELY (OPORTUNOS)

Idealmente, os testes devem ser escritos junto com a implementação do código, ou até mesmo antes (TDD).

PADRÃO ARRANGE-ACT-ASSERT (AAA)

Um padrão amplamente adotado para estruturar testes unitários é o Arrange-Act-Assert (AAA), que divide cada teste em três fases claras:

- **Arrange:** Configura o estado inicial e os objetos necessários para o teste.
- **Act:** Executa a ação ou o método que está sendo testado.
- **Assert:** Verifica se o resultado da ação corresponde ao esperado.



BOAS PRÁTICAS PARA TESTES EFICAZES

Ao escrever testes unitários para associações e outros componentes, é importante seguir algumas diretrizes para maximizar sua eficácia:

- **Nomes Descritivos:** Use nomes de métodos de teste que expliquem claramente o que está sendo testado, a condição e o resultado esperado (ex: `Metodo_Condicao_ResultadoEsperado`).
- **Foco Único:** Cada teste deve focar em verificar uma única unidade de comportamento ou um único aspecto da funcionalidade.
- **Testes de Fronteira:** Inclua cenários que cobrem valores limites ou extremos (ex: 0, 1, valores máximos/mínimos, SKUs vazios).
- **Precisão Monetária:** Utilize tipos de dados apropriados, como `decimal`, para lidar com valores monetários e evitar erros de precisão de ponto flutuante.

CONFIGURAÇÃO DO PROJETO DE TESTES E NOMENCLATURA

Para dar continuidade aos nossos laboratórios práticos, é fundamental estabelecer uma base sólida para nossos testes unitários. Utilizamos o xUnit como framework de testes, reconhecido por sua simplicidade e extensibilidade. A configuração de um projeto de testes em C# com xUnit envolve a adição de algumas dependências essenciais e a estruturação adequada do projeto para garantir a detecção e execução correta dos testes.

DEPENDÊNCIAS ESSENCIAIS

As seguintes bibliotecas NuGet são cruciais para a execução dos testes:

- **xunit:** O framework principal para escrever os testes.
- **xunit.runner.visualstudio:** Permite a integração do xUnit com o Test Explorer do Visual Studio, facilitando a execução e depuração dos testes na IDE.

- **coverlet.collector:** Utilizado para coletar métricas de cobertura de código, essencial para garantir que uma boa parte do seu código esteja sendo testada.

CONFIGURAÇÃO DO PROJETO DE TESTES (XUNIT)

Primeiro, utilizaremos o comando `dotnet new xunit` para gerar um novo projeto de testes baseado no framework xUnit. O parâmetro `-n` define o nome do projeto, `-o` especifica o diretório de saída dentro da estrutura da solução, e `--framework` garante a compatibilidade com a versão alvo do .NET.

Para concretizar os conceitos discutidos sobre testes unitários e o padrão Arrange-Act-Assert, o primeiro passo prático é a criação e configuração do projeto de testes que abrigará todas as verificações. Este projeto será dedicado a validar o comportamento do seu projeto de domínio, garantindo sua integridade e as associações modeladas.

```
dotnet new xunit -n Associations.Domain.Tests -o tests/Associations.Domain.Tests --framework net8.0
```

Em seguida, é crucial adicionar as dependências necessárias para que o xUnit funcione corretamente e para que possamos coletar informações importantes como a cobertura de código. Os pacotes `xunit.runner.visualstudio` (para integração com o Visual Studio Test Explorer) e `coverlet.collector` (para análise de cobertura) são indispensáveis.

```
dotnet add tests/Associations.Domain.Tests package xunit.runner.visualstudio --version 2.5.7
dotnet add tests/Associations.Domain.Tests package coverlet.collector --version 6.0.2
```

Por fim, o projeto de testes precisa ter acesso ao código do projeto de domínio que será testado. Isso é feito adicionando uma referência ao arquivo `.csproj` do domínio. Esta etapa é fundamental, pois permite que os testes instanciem e interajam com as classes do domínio, aplicando os princípios de validação de invariantes e comportamento esperado.

```
dotnet add tests/Associations.Domain.Tests reference src/Associations.Domain/Associations.Domain.csproj
```

Ao seguir estes passos, teremos uma base robusta para desenvolver testes unitários eficazes, permitindo a validação contínua da lógica de negócio e das associações do nosso modelo de domínio. Isso não apenas ajuda a identificar erros precocemente, mas também fornece uma rede de segurança para futuras refatorações e evoluções do sistema.

CONFIGURAÇÃO ADICIONAL: PROJETO CONSOLE E SOLUÇÃO

Para facilitar demonstrações e testes rápidos do modelo de domínio em um ambiente de execução simples, é útil criar um projeto de aplicação de console. Este projeto servirá como um ponto de entrada para instanciar e interagir com as classes e associações que estamos desenvolvendo no projeto de domínio.

Primeiramente, criamos o novo projeto de console com o comando:

```
dotnet new console -n Associations.Console -o apps/Associations.Console --framework net8.0
```

Em seguida, para que o projeto de console possa utilizar as classes definidas em `Associations.Domain`, precisamos adicionar uma referência a ele. Isso permite que o console chame métodos e crie objetos do nosso domínio.

```
dotnet add apps/Associations.Console reference
src/Associations.Domain/Associations.Domain.csproj
```

Com todos os projetos criados (domínio, testes e console), o próximo passo é integrá-los à solução principal (`.sln`). Adicionar todos os projetos à solução garante que o Visual Studio ou outras ferramentas de desenvolvimento .NET reconheçam a estrutura completa do nosso trabalho, facilitando a compilação, o gerenciamento de dependências e a execução dos testes e da aplicação de console. Isso também organiza o explorador de soluções de forma coesa, refletindo a estrutura que definimos.

```
dotnet sln add src/Associations.Domain/Associations.Domain.csproj
dotnet sln add tests/Associations.Domain.Tests/Associations.Domain.Tests.csproj
dotnet sln add apps/Associations.Console/Associations.Console.csproj
```

Estes comandos garantem que todos os componentes estejam devidamente linkados e acessíveis dentro do ambiente de desenvolvimento, promovendo uma organização clara e eficiente.

ESTRUTURA DO PROJETO DE TESTES

O arquivo `.csproj` do seu projeto de testes, como o exemplo abaixo, define as dependências necessárias e a estrutura básica. Ele referencia o projeto principal (`Associations.Domain` neste caso) para que os testes possam acessar as classes a serem testadas.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <IsPackable>false</IsPackable>
    <Nullable>enable</Nullable>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="xunit" Version="2.7.0" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.5.7" />
    <PackageReference Include="coverlet.collector" Version="6.0.2" />
  </ItemGroup>
  <ItemGroup>
    <ProjectReference Include="..\..\src\Associations.Domain\Associations.Domain.csproj" />
  </ItemGroup>
</Project>
```

NOMES DESCRITIVOS PARA MÉTODOS DE TESTE

A clareza na nomenclatura dos testes é tão importante quanto a lógica do teste em si. Nomes bem escolhidos funcionam como micro-documentação, explicando instantaneamente o propósito do teste para qualquer desenvolvedor que o leia. Adotamos a convenção `{MétodoTestado}_{CenárioDeTeste}_{ResultadoEsperado}`.

```
using System;
using Associations.Domain;
using Xunit;

namespace Associations.Domain.Tests
{
    public class MoneySpecs
    {
        // AAA: Arrange-Act-Assert
        // Nome no padrão:
        //    {MétodoTestado}_{CenárioDeTeste}_{ResultadoEsperado}
        [Fact]
        public void Ctor_ValorNegativo_DeveLancarArgumentOutOfRange()
        {
            // Arrange: (não há preparação — vamos direto ao ato)

            // Act + Assert: criar Money com valor negativo deve falhar
```

```
            Assert.Throws<ArgumentOutOfRangeException>(() =>
                new Money(-0.01m));
        }

        [Fact]
        public void
            OperatorPlus_SomaValores_NaoAlteraOperandosERetornaNovoMoney()
        {
            // Arrange: dois valores válidos
            var a = new Money(10m);
            var b = new Money(2.50m);

            // Act: somar (deve criar um NOVO Money, sem modificar a e b)
            var c = a + b;

            // Assert: valor correto no resultado e imutabilidade dos
            // operandos
            Assert.Equal(12.50m, c.Value);
            Assert.Equal(10m, a.Value);     // a permanece igual
            Assert.Equal(2.50m, b.Value);   // b permanece igual
        }
    }
}
```

CONCLUSÃO: FUNDAMENTOS PARA ASSOCIAÇÕES SÓLIDAS

Este ciclo de laboratórios guiados culmina com a consolidação de conhecimentos essenciais para o desenvolvimento de software robusto e de domínio. Abordamos a criação de Value Objects, a importância dos testes unitários e a organização de uma estrutura de projeto eficiente, preparando o terreno para a próxima fase.

VALUE OBJECT MONEY SÓLIDO

Implementação de um Money confiável com invariantes de domínio, imutabilidade em operações (+ e *) e comparação por valor (Equals/GetHashCode). Isso assegura subtotais e totais consistentes em qualquer contexto.

TESTES UNITÁRIOS ABRANGENTES

Desenvolvimento de testes unitários com xUnit, seguindo os princípios F.I.R.S.T. e o padrão AAA. Incluímos testes de fronteira e nomenclatura descritiva para garantir a qualidade do VO, cobrindo cenários como criação com valor negativo e imutabilidade de operações.

ESTRUTURA DE SOLUÇÃO ORGANIZADA

Criação de uma estrutura de projetos .NET coesa, com projetos de Domínio, Testes e Console devidamente referenciados e integrados à solução (.sln). Isso otimiza o ciclo de desenvolvimento, permitindo depuração e medição de cobertura de forma eficiente.

PRÓXIMOS PASSOS: CONSTRUINDO ASSOCIAÇÕES NO CÓDIGO

Com uma base sólida estabelecida, o foco agora se volta para a implementação prática das associações de domínio. O próximo estágio explorará as complexidades e as melhores práticas para representar relacionamentos entre entidades, garantindo que o comportamento do sistema reflita fielmente as regras de negócio.

- **Associações Unidirecionais e Bidirecionais:** Entenda como modelar e implementar relacionamentos como 1:N, focando no encapsulamento e na integridade de dados, utilizando o Value Object `Money` para cálculos de totais.
- **Associações N:M e Qualificadores:** Aprofunde-se em cenários mais complexos, incluindo tabelas de associação explícitas e a aplicação de qualificadores e restrições de cardinalidade, sempre com o suporte de testes unitários que validam cada regra do modelo.

Este material encerra com a certeza de que você possui um **Value Object confiável**, **testes exemplares** e uma **solução preparada para o crescimento**. No próximo módulo, transformaremos essas regras de negócio em código, solidificando as associações de verdade.