

# Atividade Guiada — Associações em C#

Consolidar, em código e testes, os tópicos dos laboratórios guiados: **1:N unidirecional com composição** e **1:N bidirecional com sincronização**, além de **qualificadores** – aplicando **direção mínima, invariantes por design e testes xUnit (F.I.R.S.T. + AAA)**.

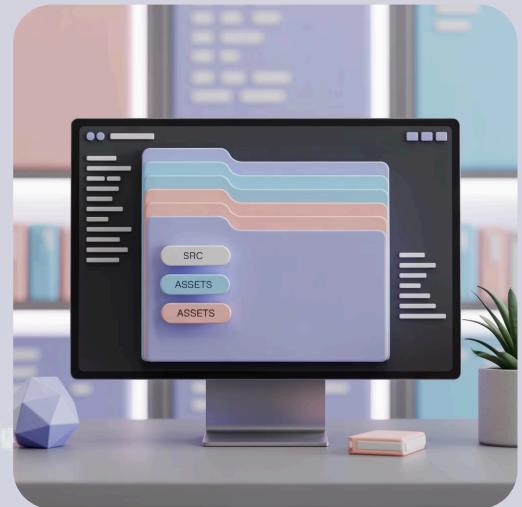
## Estrutura de Solução e Projetos

### Organização do Projeto

A estrutura de solução segue as melhores práticas de organização de código .NET, separando responsabilidades em camadas distintas:

- **src/** - Código fonte do domínio
- **tests/** - Testes unitários xUnit
- **apps/** - Aplicações de demonstração

Esta organização facilita a manutenção e permite escalabilidade do projeto.



01

#### Criar Diretórios

Estabelecer a estrutura base com pastas src, tests e apps

02

#### Configurar Solução

Criar solução principal e projetos de domínio, testes e aplicação

03

#### Definir Referências

Estabelecer dependências entre projetos e configurar propriedades

## Configurações do Projeto

### Nullable Enable

Habilita verificação de referências nulas em tempo de compilação, aumentando a segurança do código e reduzindo erros de runtime.

### Implicit Usings

Simplifica o código removendo a necessidade de declarar usings comuns, mantendo o foco na lógica de negócios.

### Target Framework

Define .NET 9.0 como versão alvo, garantindo acesso às funcionalidades mais recentes da plataforma.

```
<PropertyGroup>
<TargetFramework>net9.0</TargetFramework>
<Nullable>enable</Nullable>
<ImplicitUsings>enable</ImplicitUsings>
</PropertyGroup>
```

## Cenário Exemplo: Order e OrderItem

### Agregado 1:N com Composição

#### Order → OrderItem

O todo é dono do ciclo de vida das partes. Os itens não existem sem o pedido; manipulação apenas pelo todo com coleção encapsulada.

### Direção Unidirecional

Fluxo apenas do todo para as partes, simplificando a arquitetura e reduzindo acoplamento desnecessário.

### Invariante por Design

OrderItem sempre pertence a um Order. Totais derivados calculados sob demanda. Política clara para duplicatas.

*"O todo é dono do ciclo de vida das partes"* - Este princípio fundamental garante que os itens do pedido não podem existir independentemente, mantendo a integridade do agregado.

## Associação Bidirecional: Department e Employee

### Intuição da Bidirecionalidade

A leitura frequente nos **dois sentidos** justifica a bidirecionalidade. Precisamos navegar tanto de departamento para funcionários quanto de funcionário para departamento.

### Invariante Crítico

- Ao AddEmployee(emp), o emp.Department é sincronizado
- Ao RemoveEmployee(emp), o emp.Department vira null
- Sem duplicatas; coleção mantém unicidade



## Department

Gerencia coleção de funcionários com sincronização automática

## Employee

Mantém referência para departamento com controle interno

# Implementação da Classe Order

### Encapsulamento da Coleção

A lista privada `_items` garante que apenas a classe Order pode modificar os itens, mantendo a integridade do agregado.

### Interface ReadOnly

Exposição da coleção através de `IReadOnlyCollection<OrderItem>` permite leitura sem comprometer o encapsulamento.

### Validação na Fronteira

Todas as validações ocorrem no método `AddItem`, garantindo que apenas dados válidos entrem no sistema.

```
public sealed class Order
{
    private readonly List<OrderItem> _items = new();
    public IReadOnlyCollection<OrderItem> Items => _items.AsReadOnly();

    public void AddItem(string productId, int qty, decimal unitPrice,
        decimal discount = 0m)
    {
        // Validações na fronteira
        if (string.IsNullOrWhiteSpace(productId))
            throw new ArgumentException("productId required");
        if (qty <= 0)
            throw new ArgumentOutOfRangeException(nameof(qty));
        if (unitPrice < 0m || discount < 0m)
            throw new ArgumentOutOfRangeException();

        _items.Add(new OrderItem(productId, qty, unitPrice, discount));
    }
}
```

# Implementação da Classe OrderItem



## Imutabilidade por Design

Todas as propriedades são **somente leitura**, garantindo que uma vez criado, o item não pode ser modificado inadvertidamente.

## Cálculo Derivado

O Subtotal é calculado dinamicamente, eliminando a necessidade de sincronização manual e reduzindo possibilidades de inconsistência.

```
public sealed class OrderItem
{
    public string ProductId { get; }
    public int Quantity { get; }
    public decimal UnitPrice { get; }
    public decimal Discount { get; }
    public decimal Subtotal => (UnitPrice - Discount) * Quantity;

    public OrderItem(string productId, int qty, decimal unitPrice, decimal
        discount)
    {
        ProductId = string.IsNullOrWhiteSpace(productId)
            ? throw new ArgumentException("productId required")
            : productId;
        if (qty <= 0)
            throw new ArgumentOutOfRangeException(nameof(qty));
        if (unitPrice < 0m || discount < 0m)
            throw new ArgumentOutOfRangeException();
        UnitPrice = unitPrice;
        Discount = discount;
        Quantity = qty;
    }
}
```

# Implementação da Classe Department

HashSet para Unicidade  Uso de <b>HashSet&lt;Employee&gt;</b> garante automaticamente que não haverá funcionários duplicados na coleção.	Sincronização Bidirecional  Ao adicionar funcionário, automaticamente define o departamento no funcionário através de <b>e.SetDepartment(this)</b> .	Remoção Consistente  Ao remover funcionário, anula a referência do departamento no funcionário, mantendo consistência.
--	--	--

```
public sealed class Department
{
    private readonly HashSet<Employee> _employees = new();
    public IReadOnlyCollection<Employee> Employees => _employees;

    public bool AddEmployee(Employee e)
    {
        if (e is null)
            return false;
        if (_employees.Add(e)) {
            e.SetDepartment(this);
            return true;
        }
        return false; // ignora duplicata
    }

    public bool RemoveEmployee(Employee e)
    {
        if (e is null)
            return false;
        var removed = _employees.Remove(e);
        if (removed)
            e.SetDepartment(null);
        return removed;
    }
}
```

# Implementação da Classe Employee

## Controle de Acesso Interno

O método `SetDepartment` é marcado como `internal`, permitindo que apenas classes no mesmo assembly (como `Department`) possam modificar a referência.

## Propriedade Somente Leitura

A propriedade `Department` possui apenas getter público, impedindo modificação externa direta e mantendo a integridade da associação.



```
public sealed class Employee
{
    public string Name { get; }

    public Department? Department { get; private set; }

    public Employee(string name)
    {
        Name = string.IsNullOrWhiteSpace(name)
            ? throw new ArgumentException("name required")
            : name;
    }

    internal void SetDepartment(Department? d) => Department = d;
}
```

- ▢ **Sincronização Controlada:** O método interno garante que apenas o próprio `Department` pode modificar a associação, evitando inconsistências.

# Testes xUnit: Princípios F.I.R.S.T.



## Fast (Rápido)

Testes executam rapidamente, permitindo feedback imediato durante o desenvolvimento e não desencorajando sua execução frequente.



## Independent (Independente)

Cada teste pode ser executado isoladamente, sem depender do estado ou resultado de outros testes.



## Repeatable (Repetível)

Testes produzem os mesmos resultados independentemente do ambiente ou ordem de execução.



## Self-Validating (Auto-validação)

Testes têm resultado claro de sucesso ou falha, sem necessidade de interpretação manual.



## Timely (Oportuno)

Testes são escritos no momento apropriado, idealmente antes ou junto com o código de produção.

## Padrão AAA nos Testes

### Arrange (Preparar)

Configuração do estado inicial necessário para o teste. Criação de objetos, definição de valores e preparação do cenário.

### Act (Agir)

Execução da ação que está sendo testada. Geralmente uma única chamada de método ou operação específica.

### Assert (Verificar)

Verificação se o resultado obtido corresponde ao esperado. Validação do estado final e comportamento observado.

O padrão AAA torna os testes mais legíveis e organizados, facilitando a manutenção e compreensão da intenção de cada teste.

# Testes da Classe Department

```
// tests/Associations.Domain.Tests/DepartmentSpecs.cs
using Xunit;

public class DepartmentSpecs
{
    [Fact]
    public void AddEmployee_SincronizaLados()
    {
        // Arrange
        var d = new Department();
        var e = new Employee("Ana");

        // Act
        var ok = d.AddEmployee(e);

        // Assert
        Assert.True(ok);
        Assert.Contains(e, d.Employees);
        Assert.Equal(d, e.Department);
    }

    [Fact]
    public void RemoveEmployee_AnulaNoEmployee()
    {
        var d = new Department();
        var e = new Employee("Ana");
        d.AddEmployee(e);

        var ok = d.RemoveEmployee(e);

        Assert.True(ok);
        Assert.DoesNotContain(e, d.Employees);
        Assert.Null(e.Department);
    }
}
```

Os testes verificam a [sincronização bidirecional](#) e garantem que as invariantes são mantidas durante operações de adição e remoção.

# Testes da Classe Order

```
// tests/Associations.Domain.Tests/OrderSpecs.cs
using Xunit;

public class OrderSpecs
{
    [Fact]
    public void Total_SomaSubtotais()
    {
        var o = new Order();
        o.AddItem("P1", 2, 10m); // 20
        o.AddItem("P2", 1, 50m, 5m); // 45

        Assert.Equal(65m, o.Total());
    }
}
```

Este teste verifica se o cálculo do total está correto, somando os subtotais de todos os itens do pedido.

## Características Observadas

- **Rápidos:** Executam em milissegundos
- **Independentes:** Não dependem de outros testes
- **Determinísticos:** Sempre produzem o mesmo resultado
- **Comportamento único:** Cada teste verifica uma funcionalidade específica

## Sua Tarefa: Composição 1:N Unidirecional



### Recipe → Ingredient

Ingredientes pertencem à receita.  
Manipulação apenas através da receita, com coleção encapsulada e ciclo de vida controlado.



### Course → Lesson

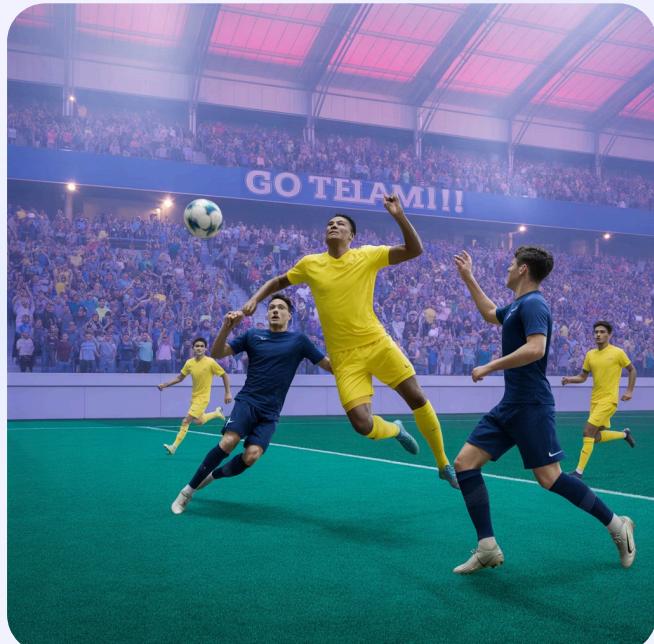
Aulas pertencem ao curso. Estrutura hierárquica onde as aulas não existem independentemente do curso que as contém.

## Características da Composição 1:N

- **Direção unidirecional:** Apenas o todo conhece as partes
- **Ciclo de vida:** Partes não existem sem o todo
- **Encapsulamento:** Coleção privada com interface somente leitura
- **Invariante:** Validação na fronteira e integridade garantida

**Importante:** Não use N..M. Se perceber essa tendência, reformule o domínio tratando as partes como pertencentes ao todo.

## Sua Tarefa: 1:N Bidirecional Sincronizado



Team  $\rightleftarrows$  Player

Entrada e saída sincronizada entre time e jogador. Navegação frequente nos dois sentidos justifica a bidirecionalidade.



Project  $\rightleftarrows$  Task

Tarefas conhecem o projeto ao qual pertencem. Sincronização automática ao adicionar ou remover tarefas.



### Sincronização Automática

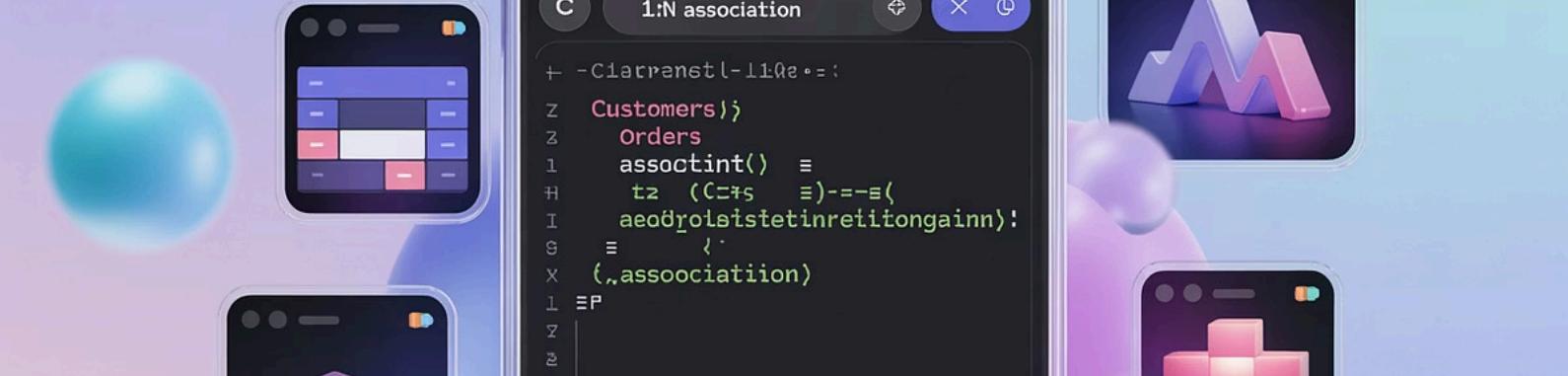
Ao adicionar/remover, ambos os lados da associação são atualizados automaticamente

### Controle de Acesso

Métodos internos garantem que apenas as classes apropriadas podem modificar associações

### Unicidade

HashSet ou validações garantem que não há duplicatas na coleção



# Entregáveis Obrigatórios

01

## README (1 página)

Cenário, decisões de direção mínima, invariantes e justificativas técnicas detalhadas

02

## Diagrama Mínimo

PNG simples com as relações e multiplicidades, mostrando a arquitetura das associações

03

## Código do Domínio

Classes com coleções encapsuladas, métodos de domínio, sem setters públicos em vínculos

04

## Testes xUnit

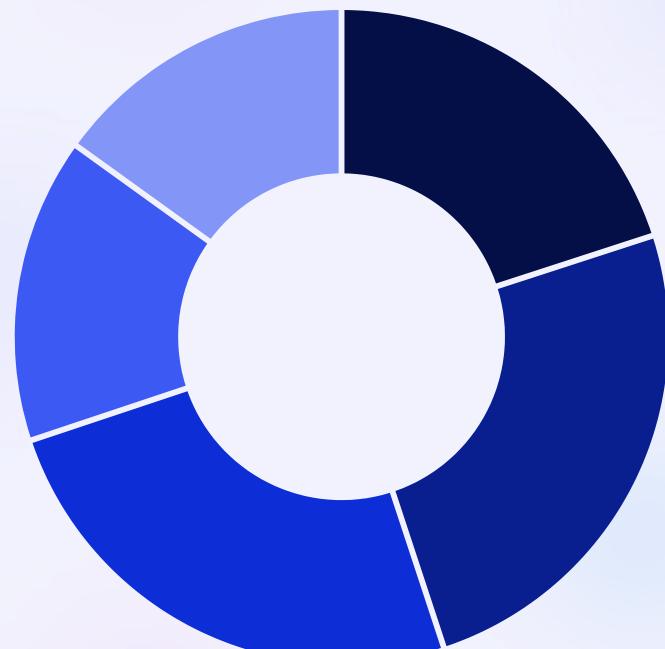
Pelo menos 4 testes (2 para composição; 2 para bidirecional), nomeados com convenção AAA

05

## Demonstração (Opcional)

App console com prints do estado após operações para validação visual

# Critérios de Aceite e Rubrica



■ Modelagem

■ Composição 1:N

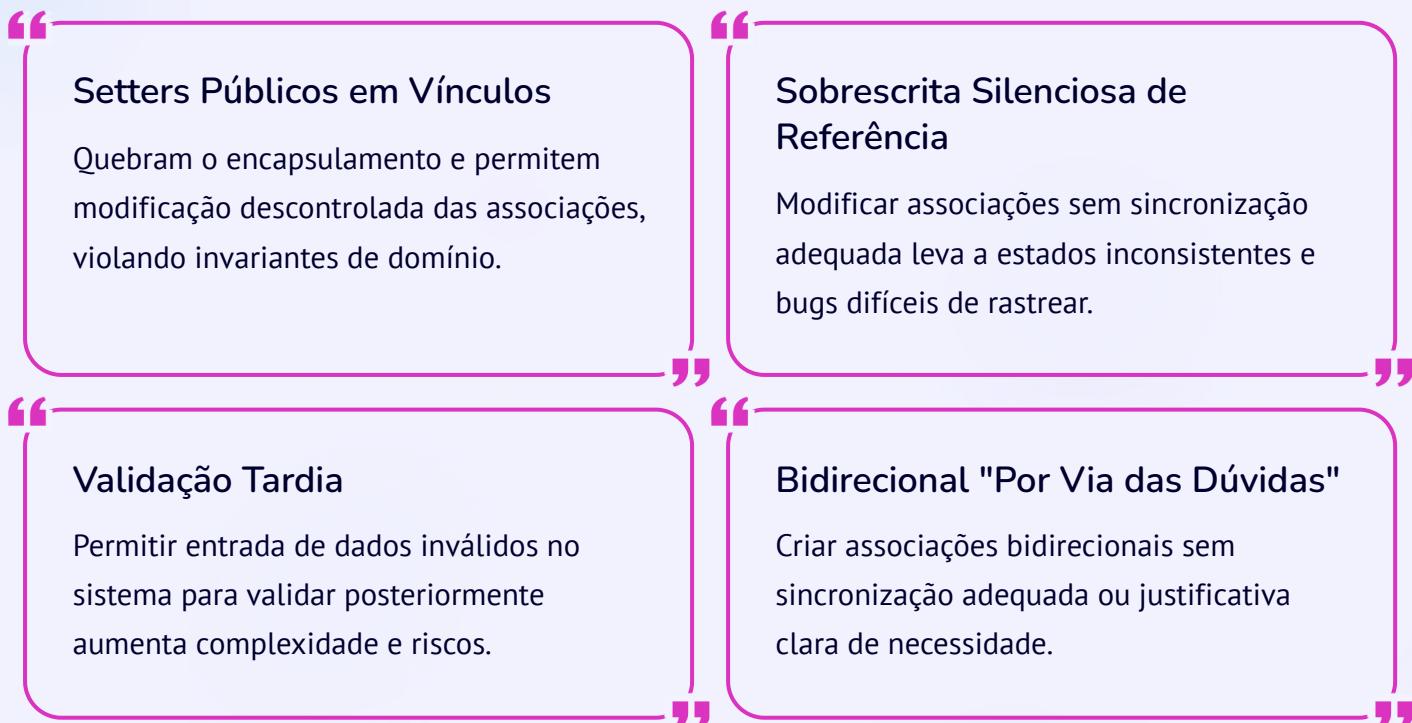
■ Bidirecional 1:N

■ Qualidade Código

■ Testes & Defesa



## Anti-Padrões Proibidos



- ☐ **Lembre-se:** Cada decisão arquitetural deve ter justificativa clara e implementação que garanta a integridade dos invariantes.

# Checklist Antes da Entrega

<input type="radio"/> README com Decisões	<input type="radio"/> Diagrama Atualizado	<input type="radio"/> Domínio Sem Setters Públicos
Documento completo explicando cenário, invariantes e justificativas técnicas para cada escolha arquitetural	Representação visual clara das associações, multiplicidades e direções implementadas	Verificação de que todas as associações mantêm encapsulamento adequado
<input type="radio"/> Testes AAA Passando	<input type="radio"/> Remoção Sincronizada	
Execução completa dos testes com prints do Test Explorer ou dotnet test	Verificação específica da sincronização bidirecional em casos de remoção	

## Defesa Técnica Obrigatória

Prepare um parágrafo explicando:

- *"O que garante que não há inconsistências?"*
- *"Onde proíbo nulos/duplicatas?"*
- *"Por que unidirecional ou bidirecional?"*

# Próximos Passos e Fechamento

## Implementação Atual

Reproduzir o roteiro com temas fixos, mantendo disciplina de direção mínima e invariantes por design

## Evolução Futura

Expandir conhecimentos para associações mais complexas e padrões avançados



## Próximas Aulas

Avançar para qualificadores e regras transversais, continuando sem N..M por enquanto

Você viu **exatamente** como esperamos que faça:  
estrutura, código mínimo, testes e justificativas. Agora,  
[repita o roteiro com os temas fixos](#), mantendo a  
disciplina de **direção mínima, invariantes por design e**  
**testes bem escritos.**

### Entrega Final

Pasta zip com solução completa, README.md, PNG do  
diagrama e capturas dos testes executados

### Qualidade Esperada

Código limpo, testes abrangentes e documentação  
clara demonstrando domínio dos conceitos

### Aprendizado Consolidado

Fundação sólida para associações mais complexas e  
padrões arquiteturais avançados



# Mini-apps: Implementando Associações 1:N

Nesta fase do laboratório, o objetivo é consolidar a compreensão das associações 1:N em C# através da criação de mini-aplicações interativas. Essas ferramentas práticas permitirão visualizar e manipular os conceitos de composição e sincronização bidirecional, aplicando os princípios de design discutidos anteriormente.

1

## Escopo Definido

A implementação deve focar exclusivamente em associações 1:N, abrangendo tanto a **composição unidirecional** (ex: Order → Item) quanto a **bidirecional sincronizada** (ex: Department ⇌ Employee), com navegabilidade mínima e invariantes por design. Associações N:M não são o foco desta etapa.

2

## Tecnologia Client-Side

Cada mini-aplicação será desenvolvida como uma solução **client-side**, utilizando uma pilha simples de HTML, JavaScript e CSS. Isso pode incluir interfaces geradas com o auxílio de ferramentas de IA para agilizar o processo de criação.

3

## Validações e Invariantes

É crucial que cada mini-app incorpore validações coerentes com o modelo 1:N, garantindo que as **invariantes** de cada associação sejam mantidas. A integridade dos dados e o comportamento esperado das relações devem ser visivelmente controlados pela aplicação.

4

## Evidências Exportáveis

Todas as mini-aplicações devem oferecer a capacidade de **exportar evidências** de seu funcionamento. Isso inclui a geração de arquivos PNG para capturas de tela da interface, e formatos como JSON ou CSV para representar o estado dos dados e das associações manipuladas, compondo a entrega final junto ao código C#.

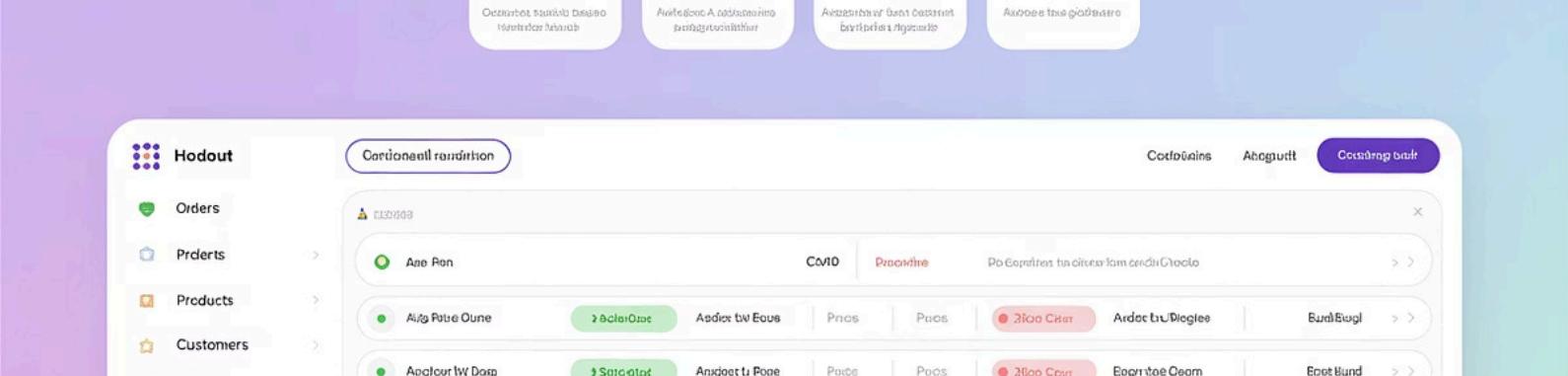
Os grupos utilizarão os mesmos cenários de domínio já explorados no código C#, como a relação Order → Item ou Department ⇌ Employee, para a construção e teste dessas mini-aplicações.

# M1 — Modelador de Composição 1:N (Order → Items)

Esta mini-aplicação tem como objetivo simular o comportamento de um agregado Order (Pedido) que gerencia uma coleção interna de OrderItem (Itens do Pedido). O foco é na composição unidirecional 1:N, onde os itens são integralmente dependentes do pedido, com o ciclo de vida das partes controlado pelo todo, e a manipulação ocorre exclusivamente através da entidade principal Order.

## Requisitos da Interface do Usuário (UI)

1	<h3>Formulário de Adição de Itens</h3> <p>Um formulário intuitivo para adicionar novos itens ao pedido, contendo os campos:</p> <ul style="list-style-type: none"><li>productId: Identificador único do produto (ex: string).</li><li>qty: Quantidade desejada do produto.</li><li>unitPrice: Preço unitário do item.</li><li>discount: Desconto opcional a ser aplicado (valor numérico).</li></ul> <p>Um botão "Add" deve estar presente para processar a adição do item à lista.</p>
2	<h3>Lista de Itens do Pedido</h3> <p>Uma tabela ou lista dinâmica que exiba todos os OrderItems adicionados, mostrando para cada um:</p> <ul style="list-style-type: none"><li>productId</li><li>qty</li><li>unitPrice</li><li>discount</li><li>O subtotal calculado para cada item.</li></ul> <p>A interface deve também exibir o <b>Total</b> geral do pedido, que é a soma dos subtotais de todos os itens.</p>
3	<h3>Funcionalidade de Remoção</h3> <p>Para cada linha de item na lista, deve haver um botão "Remove" que permita a exclusão de um OrderItem específico. A remoção deve ser baseada em um predicado simples, como o productId do item.</p>



## Validações e Invariantes

A aplicação deve garantir a integridade dos dados através das seguintes validações na entrada do formulário e nas operações:

- A quantidade (qty) de um item deve ser sempre maior que zero ( $qty > 0$ ).
- O desconto (discount) deve ser maior ou igual a zero ( $discount \geq 0$ ) e não pode exceder o preço unitário do item ( $discount \leq unitPrice$ ).
- O preço unitário (unitPrice) deve ser maior ou igual a zero ( $unitPrice \geq 0$ ).
- A coleção de OrderItems deve respeitar uma **capacidade máxima** configurável, impedindo a adição de mais itens do que o permitido.

## Evidências de Exportação

Para documentação e verificação, a mini-aplicação deve oferecer recursos de exportação:

- Um arquivo PNG da tela, capturando o estado atual da lista de itens e o total calculado.
- Um arquivo JSON representando o estado completo do objeto Order, incluindo seu identificador, capacidade máxima e todos os itens com suas respectivas propriedades:

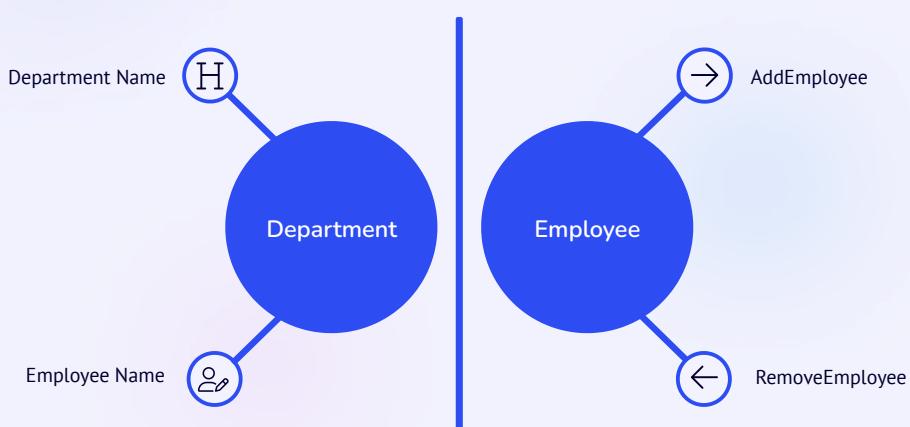
```
{
  "orderId": "string",
  "capacity": "number",
  "items": [
    {
      "productId": "string",
      "qty": "number",
      "unitPrice": "number",
      "discount": "number"
    }
  ]
}
```

# M2 — Editor 1:N Bidirecional (Department $\rightleftarrows$ Employee)

O objetivo principal desta mini-aplicação é proporcionar uma visualização clara e um reforço prático da sincronização bidirecional em associações 1:N. Através dela, será possível observar como as operações de adição e remoção de empregados afetam automaticamente o relacionamento com seus respectivos departamentos, mantendo a consistência dos dados em ambas as direções.

## Requisitos da Interface do Usuário (UI)

1	<h3>Cadastro de Entidades</h3> <p>A interface deve conter campos de entrada distintos para:</p> <ul style="list-style-type: none"><li><b>Nome do Departamento</b> ('Department{name}') : Para criar novos departamentos.</li><li><b>Nome do Empregado</b> ('Employee{name}') : Para criar novos empregados.</li></ul> <p>Cada entidade criada deve ser exibida visualmente na tela, permitindo uma interação fácil.</p>
2	<h3>Ações de Associação e Navegação</h3> <p>Funcionalidades para:</p> <ul style="list-style-type: none"><li><b>Adicionar Empregado</b> a um Departamento ('AddEmployee') : Conecta um empregado a um departamento, refletindo a associação bidirecional.</li><li><b>Remover Empregado</b> de um Departamento ('RemoveEmployee') : Desvincula um empregado de um departamento.</li></ul> <p>Essas ações devem ser representadas com setas ou conectores visuais no "canvas" da aplicação, mostrando a direção da associação (e.g., Dept <math>\rightarrow</math> Emp e Emp <math>\rightarrow</math> Dept), e a interface deve permitir a seleção de elementos para executar essas operações.</p>



# Validações e Invariantes

Para garantir a integridade da associação 1:N bidirecional, as seguintes validações devem ser implementadas:

- Um empregado **não pode pertencer a dois departamentos simultaneamente**. A tentativa de adicionar um empregado já associado a outro departamento deve resultar em um erro ou na remoção automática da associação anterior.
- Ao remover um empregado de um departamento, a associação no lado do empregado deve ser **anulada ou definida como nula**. Isso significa que o campo `department` do empregado deve ser resetado, mantendo a consistência.
- A criação de departamentos deve garantir a **unicidade dos nomes ou IDs de departamento**, impedindo duplicatas na coleção principal de departamentos.

# Evidências de Exportação

Para fins de documentação e teste, a mini-aplicação deve oferecer recursos de exportação do seu estado atual:

- Um **arquivo PNG** que capture o grafo visual do estado atual dos departamentos e empregados, incluindo suas associações.
- Um **arquivo JSON** que represente a estrutura de dados completa, detalhando os departamentos, seus empregados associados e quaisquer empregados que estejam sem departamento (órfãos):

```
{  
  "departments": [  
    {  
      "id": "string",  
      "name": "string",  
      "employees": [  
        {  
          "id": "string",  
          "name": "string"  
        }  
      ]  
    }  
  ],
```

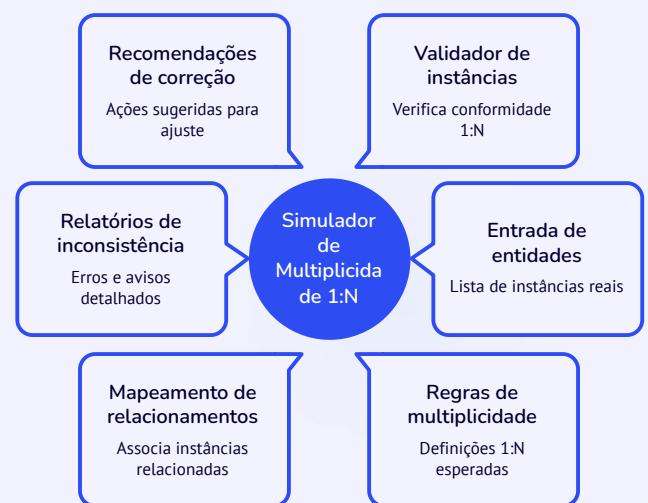
```

"orphans": [
    "employeeId1",
    "employeeId2"
]
}

```

## M3 — Simulador de Multiplicidade 1:N (validador de instâncias)

Este módulo tem como objetivo principal validar a robustez das associações 1:N implementadas, tanto em cenários de composição quanto bidirecionais. Através da geração de instâncias de dados aleatórias, o simulador irá identificar e reportar violações das regras de integridade e invariantes definidos para cada tipo de associação. Isso garante que os modelos de dados se comportem conforme o esperado sob diversas condições e cenários de uso, reforçando a integridade dos relacionamentos.



## Requisitos da Funcionalidade

1	<p><b>Seletor de Cenário</b></p> <p>A interface deve apresentar um mecanismo claro para que o usuário possa escolher qual tipo de associação 1:N será simulada, determinando o conjunto de regras de validação a serem aplicadas:</p> <ul style="list-style-type: none"> <li>• <b>Composição</b> (<code>Order → Item</code>): Focando nas regras de ciclo de vida e invariantes de itens de pedido.</li> <li>• <b>Bidirecional</b> (<code>Department ⇔ Employee</code>): Focando na consistência e sincronização de associações entre departamentos e empregados.</li> </ul> <p>A seleção do cenário ajustará dinamicamente os tipos de violações procuradas durante a simulação.</p>
---	---

2

## Botão "Simular 20"

Ao acionar este botão, o sistema deve executar as seguintes ações:

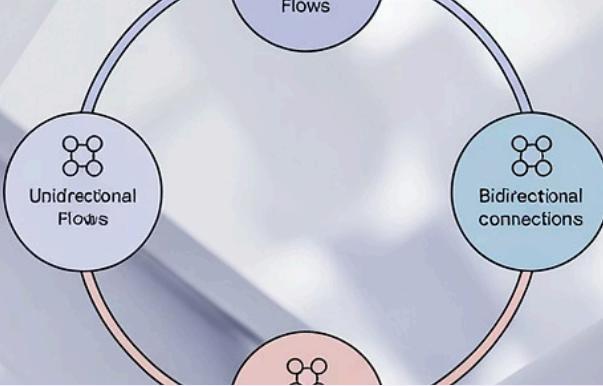
1. **Geração de Instâncias:** Criar 20 casos de teste aleatórios que envolvam os objetos e associações do cenário selecionado.
2. **Verificações de Conformidade:** Para cada instância gerada, aplicar as regras específicas do cenário:
  - **Para Composição (Order → Item):**
    - Detecção de um OrderItem "órfão" (sem um Order associado), violando o ciclo de vida da composição.
    - Validação de qty inválido (ex: qty ≤ 0) dentro de um OrderItem.
    - Verificação de capacidade máxima excedida para o número de itens em um Order.
  - **Para Bidirecional (Department ↔ Employee):**
    - Identificação de um Employee associado a dois ou mais Departments simultaneamente.
    - Detecção de um Employee desvinculado de um Department sem uma razão válida (ou seja, inconsistência bidirecional).
3. **Registro de Violações:** Todas as não conformidades encontradas devem ser registradas detalhadamente.

## Evidências de Exportação

Para facilitar a análise e a depuração das validações, o simulador deve fornecer as seguintes opções de exportação:

- Um arquivo JSON contendo todas as entradas geradas para a simulação, as saídas resultantes da validação e uma lista detalhada de violações (violations: `[{code,msg}])` encontradas, incluindo o tipo da violação e uma mensagem explicativa.
- Um arquivo CSV opcional que apresente um resumo conciso dos 20 casos simulados, indicando para cada um se foi um OK (passou nas validações) ou FAIL (falhou em uma ou mais validações), além de uma breve descrição do motivo do FAIL, se aplicável.





## M4 — Coach de Navegabilidade Mínima (1:N apenas)

O "Coach de Navegabilidade Mínima" é um módulo interativo projetado para auxiliar no design e na otimização de associações 1:N. Seu objetivo principal é guiar o usuário na identificação e remoção de redundâncias ou "setas" desnecessárias, promovendo um modelo de dados mais limpo, eficiente e fácil de manter. Ele reforça a compreensão de quando a bidirecionalidade é realmente justificável versus quando uma relação unidirecional é suficiente.

### Paleta de Relações e Cenários

Este módulo trabalha com uma paleta específica de relações 1:N já exploradas:

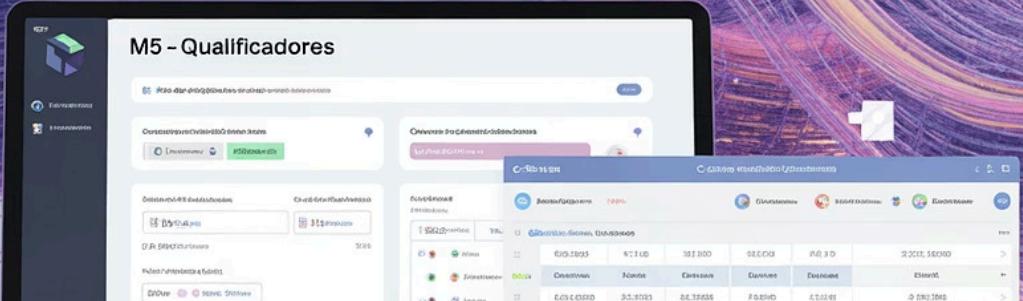
- **Composição (Order → Item):** Uma relação unidirecional, onde o OrderItem é intrinsecamente dependente do Order e não possui significado fora desse contexto. A seta vai apenas do Order para o Item.
- **Bidirecional (Department ⇌ Employee):** Uma relação onde tanto o Department precisa saber dos seus Employees quanto o Employee precisa saber do seu Department. Isso implica uma sincronização de estados e referências em ambos os lados.

### Controle de Densidade e Dicas

O sistema fornecerá um contador de arestas e uma métrica de densidade do grafo para visualização da complexidade. Uma dica essencial será exibida para provocar a reflexão sobre a necessidade real de cada "seta":

"Você realmente precisa da seta oposta? Pense nos custos de manutenção e nos invariantes distribuídos."

Essa provocação visa estimular o usuário a questionar a adição de referências bidirecionais que não são estritamente necessárias, evitando complexidade desnecessária.



## Checklist de Justificativa para Bidirecionalidade

Para auxiliar na decisão de manter uma relação bidirecional, o módulo apresentará um checklist de critérios. Uma relação bidirecional deve ser justificada por um ou mais dos seguintes pontos:

1

### Consulta Frequentes nos Dois Sentidos

A aplicação requer acesso rápido e frequente às informações em ambas as direções da associação (ex: buscar empregados de um departamento E buscar o departamento de um empregado).

2

### Invariante Distribuído

A manutenção da consistência de regras de negócio (invariante) exige que ambos os lados da associação estejam cientes um do outro e possam se comunicar para validar ou atualizar estados.

3

### Sincronização de Estados

Alterações em um lado da associação (ex: um empregado muda de departamento) precisam ser automaticamente refletidas e sincronizadas no outro lado.

4

### Propagação de Eventos

Eventos que ocorrem em um objeto (ex: um departamento é removido) precisam ser comunicados e processados pelos objetos associados (ex: empregados são desvinculados).

## Exportações

Para documentação e revisão das decisões de design, o módulo permitirá as seguintes exportações:

- Um **arquivo PNG** que visualize o grafo das relações, destacando as setas mantidas e, opcionalmente, as removidas.
- Um **arquivo JSON** contendo a descrição estrutural do grafo de relações.
- Uma **nota com a justificativa** marcada, onde o usuário pode selecionar os critérios do checklist que justificam cada relação bidirecional mantida.



# M5 — Qualificadores (Chaves para Evitar Colisão em 1:N)

O módulo **M5 – Qualificadores** aprofunda a compreensão das associações 1:N, introduzindo o conceito crucial de chaves qualificadoras. Ele tem como objetivo permitir a experimentação de como atributos específicos, quando combinados, podem atuar como chaves contextuais para garantir a unicidade de itens relacionados em uma associação 1:N, evitando colisões de dados e reforçando a integridade do modelo. Por exemplo, em um relacionamento de Order para OrderItem, um OrderItem pode ser qualificado pelo orderId combinado com um productId, assegurando que cada produto seja único dentro de um pedido específico.

Este módulo oferece uma interface prática para definir e testar esses qualificadores, simulando cenários reais de entrada de dados.

## Funcionalidades Essenciais



### Formulário de Configuração

Permite ao usuário selecionar o **escopo** (o "lado do todo", como Department ou Order) e definir os **atributos** que compõem a chave qualificadora para as entidades relacionadas.



### Importação de Dados CSV

Suporte a funcionalidade de drag-and-drop para importar um arquivo CSV fictício. Este CSV contém dados de entidades relacionadas (como Employees ou OrderItems) que serão processados usando os qualificadores definidos.



### Detecção de Colisões

Automaticamente detecta e reporta colisões (chaves qualificadoras repetidas) nos dados importados, aplicando a lógica de unicidade baseada nos atributos selecionados pelo usuário.



## Métricas e Exportações

Para uma análise aprofundada, o simulador exibirá métricas importantes e permitirá a exportação dos resultados:

12

88

12%

### Colisões Detectadas

Número total de instâncias onde a chave qualificadora resultou em duplicação.

### Itens Únicos

Quantidade de entidades relacionadas que foram devidamente qualificadas como únicas.

### Taxa de Problemas

Percentual de colisões em relação ao total de itens processados, indicando a robustez do qualificador.

Os resultados podem ser exportados nos seguintes formatos:

- **Arquivo PNG:** Uma visualização gráfica do relatório, destacando as colisões e a eficácia dos qualificadores.
- **Arquivo JSON:** Um documento estruturado contendo o scope (escopo), qualifierKeys (atributos qualificadores), stats (estatísticas das métricas) e uma lista detalhada de collisions[] (as colisões detectadas).

## M6 — Caçador de Cheiros 1:N (lint de modelagem)

O módulo **M6 — Caçador de Cheiros 1:N** atua como um "lint" de modelagem, projetado para identificar e reportar problemas comuns e "cheiros de código" (code smells) em associações 1:N. Seu objetivo é ajudar a garantir que as decisões de design, especialmente aquelas relacionadas à bidirecionalidade e encapsulamento, estejam alinhadas com as melhores práticas e com os princípios de um modelo de domínio robusto.

Ao automatizar a detecção dessas anomalias, o módulo visa auxiliar os desenvolvedores a construir modelos mais coesos, com menor acoplamento e maior integridade, evitando armadilhas comuns que podem levar a bugs difíceis de rastrear e a um código de difícil manutenção.

# Regras Sugeridas para Detecção

## Bidirecionalidade Desnecessária

Identifica associações bidirecionais que não possuem uma justificativa explícita e marcada no módulo M4, sugerindo que uma relação unidirecional poderia ser mais apropriada e menos custosa de manter.

## Coleções com Setter Público

Alerta sobre a exposição de coleções de entidades relacionadas através de setters públicos. Isso viola o encapsulamento, permitindo que o estado interno da coleção seja manipulado diretamente de fora da entidade principal, potencialmente comprometendo invariantes.

## Parte sem Todo em Composição

Deteta instâncias onde uma "parte" de uma relação de composição (como um OrderItem) existe sem estar corretamente associada a seu "todo" (como um Order), tornando-se um objeto órfão e sem significado contextual.

## Remoção sem Sincronizar em Bidirecionalidade

Aponta casos em associações bidirecionais onde a remoção de uma entidade de um lado da relação não é devidamente sincronizada com o outro lado, deixando referências inválidas ou desatualizadas.

## Falta de Validação na Fronteira

Sinaliza a ausência de validações essenciais na fronteira da entidade relacionada, como a verificação de valores como quantidade, preço ou capacidade, que são cruciais para a integridade dos dados.

# Exportações

Para documentação e depuração das questões de modelagem detectadas, o módulo oferecerá as seguintes opções de exportação:

- **Arquivo JSON:** Contendo um objeto estruturado com uma lista de issues. Cada issue detalhará a regra violada (rule), o nó ou entidade afetada (node) e uma mensagem explicativa (message) para o problema encontrado.



- **Arquivo PNG:** Uma visualização gráfica que apresentará um "diff" visual do modelo de domínio, destacando o estado "Antes" e "Depois" da aplicação das correções sugeridas, ou simplesmente marcando os "cheiros" detectados diretamente no grafo.

## Integração com Código C#: Usando as Evidências Geradas

Os módulos do nosso simulador não apenas identificam problemas e geram métricas, mas também produzem "evidências" que podem ser diretamente consumidas e verificadas em seu código C#. Esta seção detalha como integrar e validar essas evidências, garantindo a robustez e a correção das associações 1:N no seu modelo de domínio através de testes automatizados e revisões.

### Teste de Fumaça com M1

Utilize as exportações JSON do módulo M1 para um teste de fumaça. Carregue o JSON que representa a modelagem de Order → OrderItem e verifique a integridade básica. Isso inclui validar se o Total() calculado está correto e se a contagem de itens corresponde ao esperado, confirmando que a desserialização e o modelo básico funcionam.

```
[Fact]
```

```
public void SmokeTest_M1_OrderItemsTotalAndCount()
{
    var m1Json = File.ReadAllText("path/to/m1_order_data.json");
    var order = JsonConvert.DeserializeObject<Order>(m1Json);

    Assert.NotNull(order);
    Assert.Equal(ExpectedTotalValue, order.Total());
    Assert.Equal(ExpectedItemCount, order.Items.Count);
}
```

- Justifications for bidirectionality
- Don't enforce cascade
- Optimize eager loading
- Described associations
- Domain model improvements
- Don't implement bidirectional triggers
- Detection
- Detected code smells
- Domain model improvements checklist
- Domain model improvements checklist

## Teste de Consistência com M2/M3

A partir dos dados gerados ou manipulados nos módulos M2 (Editor 1:N Bidirecional) e M3 (Gerenciador de Órfãos), reconstrua as entidades Department. Implemente um teste que garanta a invariante crucial de que nenhum Employee esteja associado a dois Departments simultaneamente, prevenindo inconsistências em relações bidirecionais.

[Fact]

```
public void ConsistencyTest_M2M3_NoEmployeeInMultipleDepartments()
{
    // Simula carregamento
    var departmentsData = LoadDepartmentsFromM2M3Export();
    var allEmployees = new HashSet<Employee>();
    var employeesInMultipleDepartments = new HashSet<Employee>();

    foreach (var dept in departmentsData)
    {
        foreach (var employee in dept.Employees)
        {
            if (!allEmployees.Add(employee))
            {
                employeesInMultipleDepartments.Add(employee);
            }
        }
    }
    Assert.Empty(employeesInMultipleDepartments);
}
```

## Checklist de Revisão com M4/M6

Integre os relatórios do M4 (Coach de Navegabilidade Mínima) e M6 (Caçador de Cheiros 1:N) ao seu processo de revisão de código. O relatório de justificativas de bidirecionalidade do M4 e o relatório de "cheiros" do M6 devem ser anexados ao README do projeto ou ao sistema de controle de versão, servindo como base para discussões e melhorias no design do domínio.

```
// Exemplo de seção no README.md  
// ---  
// ## Relatório de Modelagem de Domínio (Gerado pelo M4 & M6)  
//  
// ### Justificativas para Bidirecionalidade (M4)  
// - `Department ⇄ Employee`: Acesso rápido a todos os funcionários de um departamento e ao departamento de um funcionário para fins de relatórios e validações de negócios.  
//  
// ### Cheiros de Código Detectados (M6)  
// - **Rule**: `Bidirectionalidade Desnecessária`  
// - **Node**: `Product ⇄ Category` (Justificativa ausente - Verificar necessidade.)  
// - **Rule**: `Coleções com Setter Público`  
// - **Node**: `Order.Items` (Coleção modificável externamente - Encapsulamento violado.)  
// ---
```

# Entregáveis Obrigatórios

Para a conclusão bem-sucedida desta etapa do projeto, espera-se a entrega de um conjunto específico de artefatos que demonstrem a compreensão e aplicação dos conceitos de associações 1:N, validações e exportações. Os seguintes itens são mandatórios para avaliação:

1

### Mini-aplicativos

A pasta `miniapps/` deve conter os 6 mini-aplicativos desenvolvidos. Cada mini-aplicativo deve ser acompanhado de um arquivo `README.md` (de no máximo 1 página) detalhando como cada um reforça os princípios das associações 1:N.

2

### Exportações de Dados

Para cada mini-aplicativo, deverão ser fornecidos os arquivos de exportação gerados nos formatos PNG, JSON e CSV, conforme a funcionalidade de cada módulo.



# 1

## Comprovação de Testes

Um print ou evidência do teste xUnit que realiza o carregamento do JSON gerado pelos módulos M1 (Modelador de Composição 1:N) ou M2 (Editor 1:N Bidirecional) e valida sua integridade.

A rúbrica abaixo detalha os critérios de avaliação para os mini-aplicativos, totalizando 100 pontos:

Aderência ao escopo 1:N	20
Validações e Invariantes	20
Exportações e Evidências	20
Integração com C#	20
Usabilidade/Clareza	20
<b>Total</b>	<b>100</b>

### Observações Finais Importantes:

- É **proibido** modelar ou simular associações N:M (muitos para muitos) nesta etapa. O foco deve ser estritamente em relações 1:N.
- Utilize os **mesmos cenários** empregados nos exercícios de código anteriores para garantir a consistência e o alinhamento das implementações.
- Mantenha o foco nos seguintes aspectos essenciais: composição 1:N, associações bidirecionais 1:N com sincronização adequada, e o princípio da navegabilidade mínima.