Associações entre Classes (Navegabilidade e Multiplicidade)

Guia Autônomo – com anti-padrões e correções

Este material foi revisado para destacar **o que costuma dar errado** e **como as boas práticas resolvem**.

Aviso: os **comentários nos trechos de código estão em português**, mas **nas próximas unidades migraremos os comentários para inglês**. Os identificadores de código (nomes de classes, variáveis e métodos) já estão em **inglês**.

Objetivos de aprendizagem

1

Entender associação × agregação × composição

Compreender as diferenças entre estes tipos de relacionamentos e quando usar cada um deles.

2

Aplicar multiplicidade e navegabilidade

Saber aplicar multiplicidade (0..1, 1, 0..*, 1..*, a..b) e definir a navegabilidade mínima necessária.

3

Implementar relacionamentos em C#

Implementar 1–1, 1–N (composição e agregação) e N–N em C#, **evitando anti-padrões** (erros de modelagem e de encapsulamento).

Vocabulário essencial

Associação

Vínculo conceitual entre classes.

Agregação (todo–parte fraca)

Parte **vive sem** o todo (Time ▯— Player).

Composição (todo–parte forte)

Parte **não faz sentido** sem o todo (Order ◆— OrderItem).

Multiplicidade

0..1 , 1 , 0..* , 1..* , a..b

Navegabilidade

Unidirecional (A → B) ou bidirecional (A ↔️ B).
Regra prática: comece unidirecional; torne bidirecional apenas com necessidade clara.

Boas práticas e os problemas que elas evitam

Encapsular coleções

Anti-padrão comum

Expor List publicamente (`public List Items { get; set; }`).

Problemas gerados

- **Quebra de invariantes**: qualquer parte do código pode inserir/remover sem validar
- **Acoplamento acidental**: partes externas dependem da estrutura interna
- **Vazamento de referência**: quem recebe a lista pode modificá-la depois
- **Concorrência/estado inesperado**: difícil rastrear alterações

Como a boa prática resolve

- Exposição **somente leitura** (ex.: IReadOnlyCollection) evita alterações externas
- Métodos Add/Remove centralizam validações
- Permite **trocar a estrutura interna** sem quebrar consumidores

> **Solução:** Expor IReadOnlyCollection e modificar via métodos controlados

Navegabilidade mínima

Anti-padrão comum

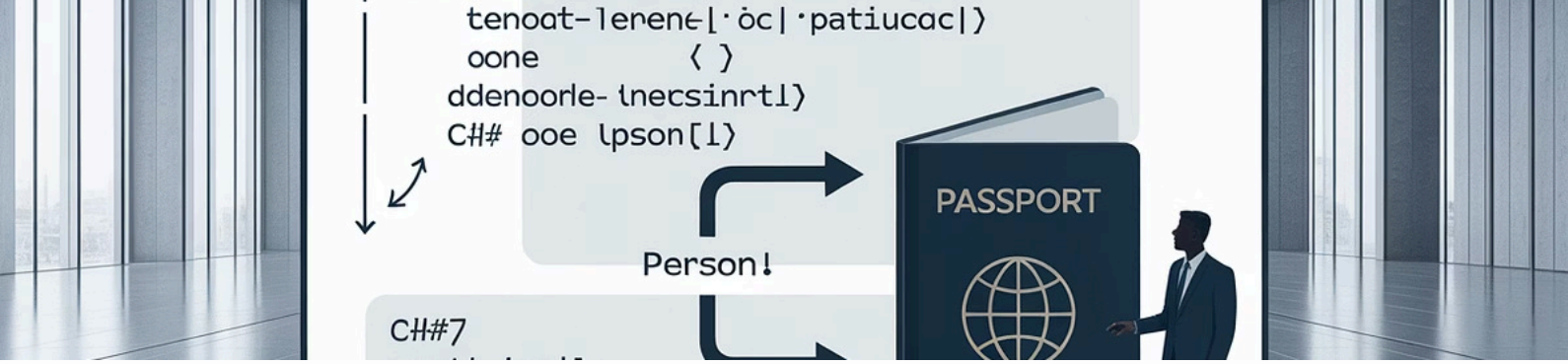
Tornar **todas** as relações bidirecionais "por via das dúvidas".

Problemas

Maior acoplamento, risco de **inconsistência** (um lado atualizado e o outro não), ciclo de referências desnecessário e testes mais difíceis.

Solução

Começar unidirecional; só elevar a bidirecional quando a leitura/manipulação direta **nos dois lados** é uma necessidade do domínio. Se bidirecional, **sincronizar ambos** os lados em um único ponto.



Validar multiplicidade

Anti-padrão comum

Ignorar limites (ex.: permitir Order sem itens; permitir Person com 2 passaportes).

Efeitos

Estados inválidos, regras de negócio quebradas, erros tardios em camadas posteriores.

Solução

Validar na **fronteira** (nos métodos que alteram o estado) – checar mínimo/máximo, impedir duplicatas, conferir unicidade.

1

Verificar limites mínimos

Garantir que coleções tenham pelo menos o número mínimo de elementos exigidos pela multiplicidade.

2

Verificar limites máximos

Impedir que coleções ultrapassem o número máximo de elementos permitidos pela multiplicidade.

3

Impedir duplicatas

Garantir que elementos únicos não sejam duplicados em coleções.

Do cotidiano ao conceito

Associações descrevem **quem conhece quem** (navegabilidade) e **em que quantidade** (multiplicidade).

Pessoa → Passaporte (1–1)

Uma pessoa possui no máximo um passaporte.

Distinguir composição × agregação

Anti-padrão comum

Usar composição quando as partes têm vida própria (ou vice-versa).

Efeitos

Exclusões indevidas (apagar Player ao remover de um Team), ou acúmulo de "órfãos" quando a parte deveria morrer com o todo.

Solução

Decidir pelo **ciclo de vida**: se a parte **depende** do todo, composição; se **vive sem** o todo, agregação.

```
// OneToOne_BadExample.cs
// Anti-padrão 1–1 (Person  Passport)
// Objetivo: mostrar um modelo que COMPILA, mas viola boas práticas e
//   invariantes.
// Comentários em PT indicam o problema e a correção esperada (que você
//   apresentará depois no código bom).

using System;

public class PersonBad
{
   // ERRO: setter público → identidade pode ser alterada por qualquer
   //   parte do sistema.
   // EFEITO: difícil garantir consistência e auditabilidade; regras que
   //   dependem do nome podem quebrar.
   // CORREÇÃO: tornar somente-leitura com validação no construtor.
   public string Name { get; set; }

   // ERRO: setter público → permite trocar/atribuir passaporte a qualquer
   //   momento, sem regra.
   // EFEITO: quebra da multiplicidade 1:1 (pode “substituir” o
   //   passaporte), aceitar expirado etc.
   // CORREÇÃO: setter privado + método de domínio que valide unicidade e
   //   data futura.
   public PassportBad Passport { get; set; }

   public PersonBad(string name)
   {
      // ERRO: nenhuma validação/normalização.
      // EFEITO: Name vazio/nulo, dificultando identificação e depuração.
      // CORREÇÃO: validar e normalizar aqui.
      Name = name;
   }
```

```
    public void IssuePassport(string number, DateTime expiration)
    {
        // ERRO: sem checar se já possui passaporte.
        // ERRO: aceita número vazio e data expirada.
        // EFEITO: pode haver 2 “estados” conflitantes (antigo e novo);
        //   regras de negócio quebradas.
        // CORREÇÃO: recusar se já houver passaporte; validar dados;
        //   retornar Result/bool ou lançar exceção (quando estudarem
        //   exceções).
        Passport = new PassportBad(number, expiration);
    }
}

public class PassportBad
{
    // ERRO: propriedades mutáveis externamente.
    // EFEITO: após a emissão, qualquer código pode alterar número/validade
    //   → viola integridade.
    // CORREÇÃO: tornar imutáveis (get-only) e validar no construtor.
    public string Number { get; set; }
    public DateTime Expiration { get; set; }

    public PassportBad(string number, DateTime expiration)
    {
        // ERRO: sem validação de número/data.
        // EFEITO: instâncias inválidas circulando (ex.: passaporte
        //   vencido).
        // CORREÇÃO: validar aqui e rejeitar estados inválidos.
        Number = number;
        Expiration = expiration;
    }
}

// Pequena demonstração do que pode dar errado (ilustrativo):
public static class OneToOneBadDemo
{
    public static void Run()
    {
        var p = new PersonBad(""); // ERRO: aceita nome vazio

        // ERRO: aceita número vazio e data expirada
        p.IssuePassport("", DateTime.UtcNow.AddDays(-1));
```

```
        // ERRO: qualquer parte do código pode sobrescrever o passaporte
        //   “oficial”
        p.Passport = new PassportBad("OVERRIDE-123",
          DateTime.UtcNow.AddYears(5));

        // ERRO: identidade mutável
        p.Name = "Anonymous";

        Console.WriteLine($"Name={p.Name}, PassportNumber=
          {p.Passport?.Number}, Exp={p.Passport?.Expiration:d}");
    }
}
```

Implementações em C# - Associação 1–1 corrigindo os anti-patterns

Person → Passport (unidirecional)

```
public class Person
{
    public string Name { get; }
    public Passport? Passport { get; private set; } // 0..1

    public Person(string name)
    {
        // Comentário (PT): normalização simples para exemplo
        Name = string.IsNullOrWhiteSpace(name) ? "Unnamed" : name;
    }

    // Emite passaporte apenas se ainda não existir e a data for futura
    public bool IssuePassport(string number, DateTime expiration)
    {
        if (Passport != null)
          return false; // evita 2 passaportes para a mesma pessoa
        if (string.IsNullOrWhiteSpace(number))
          return false;
        if (expiration <= DateTime.UtcNow.Date)
          return false; // precisa ser data futura

        Passport = new Passport(number, expiration);
        return true;
    }
}
```

```
public class Passport
{
    public string Number { get; }
    public DateTime Expiration { get; }

    public Passport(string number, DateTime expiration)
    {
        // Comentário (PT): objeto imutável para este exemplo
        Number = number;
        Expiration = expiration;
    }
}
```

Anti-padrões evitados aqui: duplicidade de passaporte (quebra 1–1), aceitar passaporte vencido, permitir alteração externa do vínculo (setter público).

Guia de Reflexão e Validação: Associações 1:1

Este guia oferece um roteiro para refletir sobre o design e validar a implementação de associações um-para-um (1:1), como **Pessoa → Passaporte**. Ele visa garantir que o modelo de domínio reflita corretamente a realidade e que o código previna estados inconsistentes e problemas de integridade de dados.

Modelo Mental e Invariantes do Problema

Ao projetar uma relação 1:1, é fundamental definir os "invariantes" – as condições que devem ser sempre verdadeiras. Esses pontos se tornarão a base para a validação e a construção de um código robusto:

- **Multiplicidade:** Uma Pessoa pode ter no máximo um Passaporte. O sistema deve impedir a duplicação.
- **Navegabilidade:** A necessidade de acesso é tipicamente unidirecional (consultar o passaporte a partir da pessoa).
- **Integridade do Vínculo:** Após a emissão, o passaporte não deve ser trocado ou removido livremente, sem regras de negócio.
- **Qualidade dos Dados:** O passaporte deve ter um número válido e uma data de expiração futura no momento da emissão.
- **Imutabilidade Útil:** Os dados do Passaporte (número, data de expiração) não devem ser alterados após a criação, para manter a integridade.

Perguntas que Evitam Erros (Checklist de Design)

Use estas perguntas como um "checklist mental" antes de iniciar a implementação, para antecipar problemas:

1	**Quem muda o quê?** Defina claramente as permissões: o nome da pessoa pode mudar? O passaporte pode ser substituído? Quem tem autoridade para isso?
2	**Onde validar?** A validação (ex: número/expiração do passaporte) deve acontecer na fronteira, no método que o emite, evitando lógica espalhada.
3	**Garantia 1:1?** O código impede uma segunda emissão se já existir um Passaporte? Há algum *setter* público que permita a reatribuição indevida?
4	**Imutabilidade útil?** O objeto Passaporte será somente leitura (get-only) após criado? Isso reduz significativamente a chance de corrupção acidental dos dados.
5	**Navegabilidade mínima?** É realmente necessário que o Passaporte aponte para a Pessoa? Evite acoplamento desnecessário se não houver uso claro.

Decisões de Design que Antecipam Problemas

A implementação ideal deve incorporar decisões de design que assegurem os invariantes desde o início:

- **Encapsulamento do Vínculo:** O atributo `Person.Passport` deve ter um *setter* privado, e a emissão deve ocorrer via um método específico (ex: `IssuePassport`) que controle o processo.
- **Validação na Fronteira:** O método `IssuePassport` deve ser o ponto central para validar se o número não é vazio e se a data de expiração é futura, recusando dados inválidos.
- **Imutabilidade de Dados Sensíveis:** As propriedades do objeto `Passport` (Número, Expiração) devem ser somente leitura (`get-only`), prevenindo alterações após sua criação.
- **Sem Trocas Silenciosas:** Se já houver um passaporte associado à pessoa, a tentativa de emissão de um novo deve retornar uma falha clara (ex: `false` ou lançar uma exceção, conforme a estratégia de tratamento de erros).
- **APIs Claras:** Deve haver um único ponto de entrada para criar ou vincular o passaporte, evitando a criação de estados inconsistentes.N–N sem classe de ligação

Sinais de Alerta no Code Review

Ao revisar o código de uma associação 1:1, observe atentamente estes pontos para identificar fragilidades que podem levar a inconsistências de dados:

Setter Público no Vínculo

Um atributo como `public Passport Passport { get; set; }` indica que o vínculo pode ser reatribuído a qualquer momento, sem respeitar regras de negócio ou a multiplicidade 1:1.

Propriedades Mutáveis em Objetos Dependentes

Propriedades como `public string Number { get; set; }` dentro de `Passport` permitem a alteração externa do estado do passaporte após sua criação, violando a integridade.

Construtor sem Validação

A ausência de validação (ex: número vazio, data expirada) no construtor de `Passport` permite a criação de objetos em estados inválidos, que podem circular pelo sistema.

Método de Emissão Incompleto

Um método `IssuePassport(...)` que não checa a existência de um passaporte anterior permite a duplicação, quebrando o invariante 1:1.

Comentários de "Validar Depois"

Comentários ou `TODOs` que adiam validações críticas de domínio sugerem um design incompleto e propenso a falhas futuras.

Falta de Testes Negativos

A ausência de testes para cenários inválidos (data expirada, número vazio, segunda emissão) é um forte indicativo de que essas regras podem não estar sendo aplicadas corretamente.

Validação Orientada por Invariantes: Conferência Pós-Implementação

Após a implementação, a validação não deve ser uma "tarefa a cumprir", mas uma leitura crítica e uma conferência do comportamento, focando nos invariantes estabelecidos.

Conferência de Design (Leitura do Código)

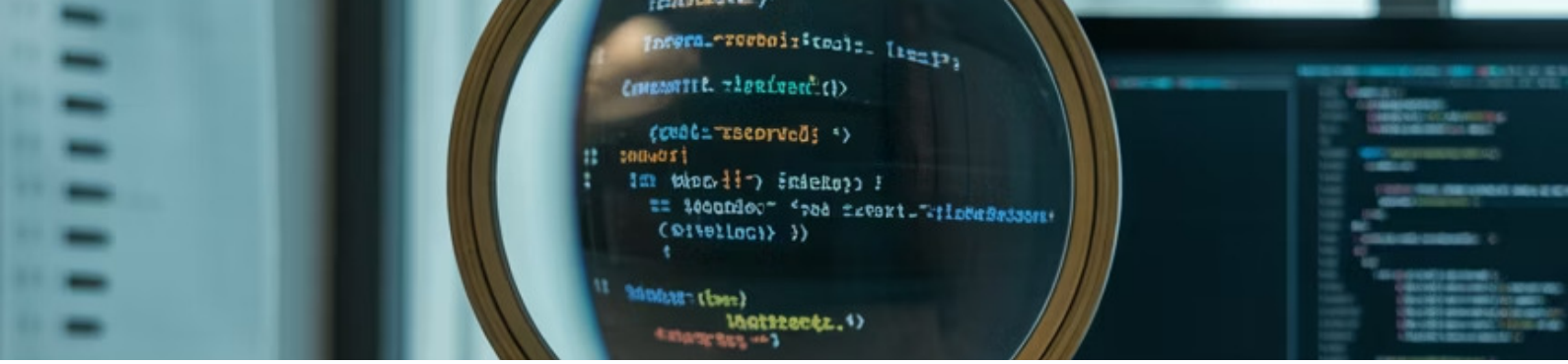
- **Encapsulamento:** O atributo `Person.Passport` possui um `setter` privado?
- **Ponto Único de Emissão:** Existe um método único (`IssuePassport`) que valida a inexistência de passaporte, número não vazio e expiração futura?
- **Imutabilidade de Dados:** O objeto `Passport` possui propriedades somente leitura (`get-only`) e é construído já em um estado válido?
- **Navegabilidade Mínima:** A relação é apenas `Person → Passport`, sem dependência inversa desnecessária?

Conferência de Comportamento (Execução/Testes)

- **Multiplicidade 1:1:** Ao tentar emitir um segundo passaporte, a operação falha claramente (retorna `false` ou lança exceção)?
- **Validação de Dados:** A emissão é recusada ao usar dados inválidos (data expirada, número vazio)?
- **Imutabilidade Efetiva:** Após a emissão, não é possível alterar o número/expiração do passaporte por "atalhos" externos?
- **Estado Coerente:** O estado final da pessoa é sempre coerente (0 ou 1 passaporte válido)?

Casos de Borda para Observar

- **Data "Hoje":** A regra de expiração considera estritamente o futuro (`>= DateTime.UtcNow.Date`), evitando passaportes "válidos no limite" que expiram imediatamente.
- **Normalização de Entradas:** Nomes vazios ou com apenas espaços em branco não passam silenciosamente (são normalizados para "Unnamed" ou similar).
- **Concorrência (Conceitual):** Conceitualmente, duas chamadas quase simultâneas de `IssuePassport` não deveriam causar um estado inconsistente, produzindo o mesmo resultado que uma única chamada (será aprofundado em tópicos futuros).



Critérios de Aceite para Implementações Robustas

Uma implementação é considerada correta e robusta se atender aos seguintes critérios:

Invariantes Garantidos

As regras de negócio são sempre mantidas: 1 pessoa → 0 ou 1 passaporte válido, com dados consistentes.

API Segura

A emissão é o único caminho de criação do vínculo, e nunca permite que o objeto entre em um estado inválido.

Imutabilidade Onde Importa

Dados sensíveis do passaporte não podem ser corrompidos ou alterados após sua criação.

Baixo Acoplamento

A navegação é mínima (`Person → Passport`), e as responsabilidades são claras e bem definidas.

Erros Clássicos e Por Que o Modelo Correto os Evita

- **Setter público no vínculo:** Permite a troca silenciosa ou sobrescrita do passaporte.
  Evita-se com setter privado e um método de emissão que encapsula a lógica e validação.
- **Construtor permissivo em objetos dependentes:** Facilita a circulação de instâncias inválidas.
  Evita-se validando rigorosamente no construtor e tornando as propriedades `get-only`.
- **Falta de ponto único de mutação:** Gera regras espalhadas e estados inconsistentes.
  Evita-se centralizando toda a lógica de criação e vínculo em um único método de domínio, como `IssuePassport`.

A Leitura Final: A Regra dos Três Cs

Ao fazer a leitura final do código e do comportamento, avalie a implementação pelos "Três Cs":

Clareza

O código deixa evidente quem pode criar ou alterar o vínculo e quais são as regras de negócio?

Coesão

A lógica de emissão e as validações vivem no lugar certo, dentro da entidade `Person` (ou entidade responsável)?

Consistência

Após a criação, o estado do objeto e de suas associações permanece válido, sem "atalhos" que possam quebrá-lo?

Se, ao refletir antes e checar depois, essas condições forem verdadeiras, a implementação da associação 1:1 está correta por design, e os problemas comuns associados a "código errado" são efetivamente prevenidos.

Anti-padrão comum

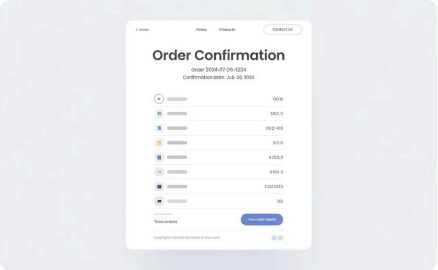
Modelar N–N apenas com duas coleções (Aluno.Courses e Course.Students), quando o vínculo **tem dados** (data, status, nota).

Efeitos

Perda de informação do relacionamento, dificuldade de evoluir para persistência relacional.

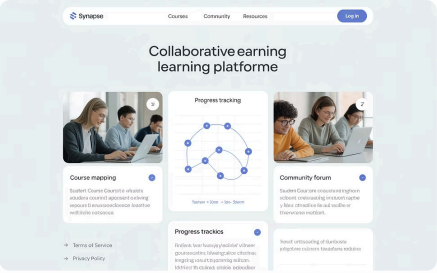
Solução

Introduzir **classe de ligação** (ex.: Enrollment) quando o vínculo tem atributos.



Pedido → Item do Pedido (1–N, composição)

Um pedido contém vários itens que não existem sem o pedido.

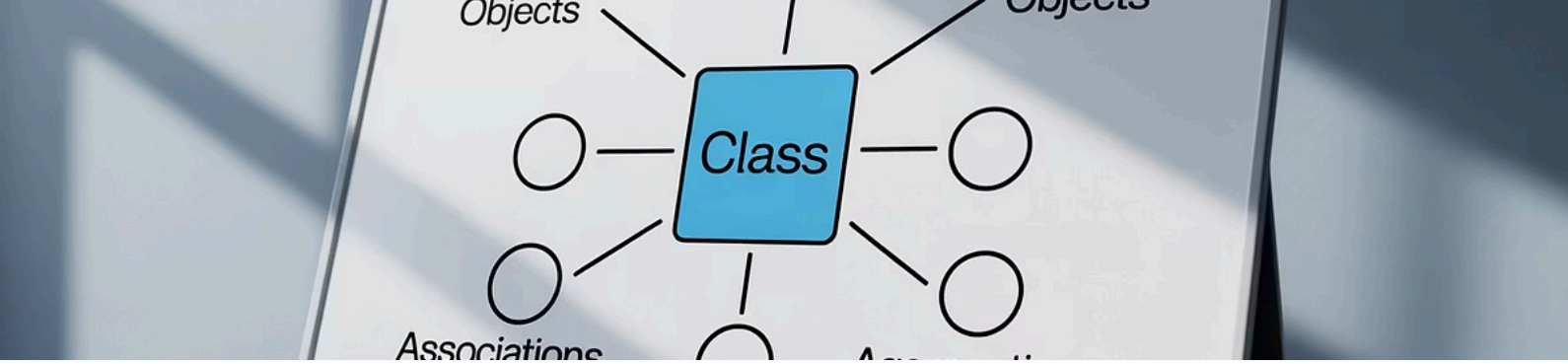


Aluno ↔️ Turma/Curso (N–N via matrícula)

Um aluno pode estar em várias turmas e uma turma tem vários alunos.

Time → Jogadores (1–N, agregação)

Um time possui vários jogadores, mas estes existem independentemente.


Objects
Objects
Class
Associations

Implementações em C# - Composição 1–N (Parte 1)

Order ◆— OrderItem (coleção encapsulada)

```
public class Product
{
 public string Name { get; }
 public decimal UnitPrice { get; }
 public Product(string name, decimal unitPrice)
 {
 // Comentário (PT): tolerância simples para preço não negativo
 Name = string.IsNullOrWhiteSpace(name) ? "Product" : name;
 UnitPrice = unitPrice < 0 ? 0 : unitPrice;
 }
}
public class OrderItem
{
 public Product Product { get; }
 public int Quantity { get; private set; }
 internal OrderItem(Product product, int quantity)
 {
 // Comentário (PT): item só existe dentro de Order (ctor interno)
 Product = product;
 Quantity = quantity < 1 ? 1 : quantity; // quantidade mínima
 }
 public void Increase(int value)
 {
 // Comentário (PT): mutação controlada mantém invariantes
 if (value > 0) Quantity += value;
 }
 public decimal Subtotal() => Product.UnitPrice * Quantity;
}
```

Implementações em C# - Composição 1–N (Parte 2)

```
public class Order
{
    private readonly List _items = new();
    public IReadOnlyCollection Items => _items.AsReadOnly(); // Comentário (PT): evita alteração
externa
    public bool AddItem(Product product, int quantity)
    {
        // Comentário (PT): única porta de entrada para validar regras
        if (product == null || quantity <= 0) return false;
        _items.Add(new OrderItem(product, quantity));
        return true;
    }
    public bool RemoveItemAt(int index)
    {
        if (index < 0 || index >= _items.Count) return false;
        _items.RemoveAt(index);
        return true;
    }
    public bool CanFinish() => _items.Count > 0; // Comentário (PT): não finalizar pedido vazio
    public decimal Total() => _items.Sum(i => i.Subtotal());
}
```

Anti-padrões evitados aqui: List pública (quebra invariantes), criação de OrderItem fora do Order (quebra composição), remoções/adições sem validação.

Implementações em C# - N–N (Parte 1)

Student ↔️ Course (bidirecional com consistência)

```
public class Student
{
 private readonly HashSet _courses = new();
 public IReadOnlyCollection Courses => _courses; // Comentário (PT): leitura apenas
 public string Name { get; }
 public Student(string name)
 {
 Name = string.IsNullOrWhiteSpace(name) ? "Student" : name;
 }
 // Comentário (PT): mantém consistência dos dois lados
 public bool Enroll(Course course)
 {
 if (course == null) return false;
 var added = _courses.Add(course);
 if (added) course.InternalAddStudent(this);
 return added;
 }
 internal void InternalAddCourse(Course course) => _courses.Add(course);
}
```

Implementações em C# - N–N (Parte 2)

```
public class Course
{
    private readonly HashSet _students = new();
    public IReadOnlyCollection Students => _students; // Comentário (PT): leitura apenas
    public string Title { get; }
    public Course(string title)
    {
        Title = string.IsNullOrWhiteSpace(title) ? "Course" : title;
    }
    // Comentário (PT): ponto único para manter bidirecionalidade
    public bool Register(Student student)
    {
        if (student == null) return false;
        var added = _students.Add(student);
        if (added) student.InternalAddCourse(this);
        return added;
    }
    internal void InternalAddStudent(Student student) => _students.Add(student);
}
```

Anti-padrões evitados aqui: duplicidade em N–N (uso de HashSet), quebra de consistência (um lado atualizado, outro não), exposição de coleção mutável.

Implementações em C# - Agregação 1–N

Team 1— Player (partes com vida própria)

```
public class Player
{
    public string FullName { get; }
    public Player(string fullName)
    {
        // Comentário (PT): entidade que existe fora do Team
        FullName = string.IsNullOrWhiteSpace(fullName) ? "Player" : fullName;
    }
}
public class Team
{
    private readonly List _players = new();
    public IReadOnlyCollection Players => _players.AsReadOnly();
    public string Name { get; }
    public Team(string name)
    {
        Name = string.IsNullOrWhiteSpace(name) ? "Team" : name;
    }
    public bool AddPlayer(Player player)
    {
        if (player == null) return false;
        if (_players.Contains(player)) return false; // Comentário (PT): evita duplicidade
        _players.Add(player);
        return true;
    }
    public bool RemovePlayer(Player player)
    {
        if (player == null) return false;
        return _players.Remove(player);
    }
}
```

Anti-padrões evitados aqui: apagar Player ao removê-lo do Team (confundir com composição), permitir duplicidades na mesma equipe.

Execução ilustrativa

```
// Program.cs (top-level statements) – demonstração de uso
var alice = new Person("Alice");
var issued = alice.IssuePassport("BR123456", DateTime.UtcNow.Date.AddYears(5));
Console.WriteLine($"Has passport now? { (alice.Passport == null ? "no" : "yes") } | Issued:
{issued}");

var order = new Order();
order.AddItem(new Product("Coffee", 18.90m), 2);
order.AddItem(new Product("Tea", 12.50m), 1);
Console.WriteLine($"Order items: {order.Items.Count} | Total: {order.Total():C}");
Console.WriteLine($"Can finish? {order.CanFinish()}");

var bob = new Student("Bob");
var csharp = new Course("C# Fundamentals");
bob.Enroll(csharp);
Console.WriteLine($"{bob.Name} courses: {string.Join(", ", bob.Courses.Select(c => c.Title))}");
Console.WriteLine($"{csharp.Title} students: {string.Join(", ", csharp.Students.Select(s =>
s.Name))}");

var team = new Team("Tigers");
var player = new Player("Carol Lee");
team.AddPlayer(player);
Console.WriteLine($"Team {team.Name} — players: {team.Players.Count}");
```

Leitura da saída: confirma emissão de passaporte, total do pedido, matrícula e contagem de jogadores.

Tabela de referência rápida

Anti-padrão → Efeito → Correção

Anti-padrão	Efeito colateral	Correção apresentada
public List exposta	Invariantes quebradas; acoplamento; vazamento de referência	Expor IReadOnlyCollection + métodos Add/Remove
Bidirecional "por padrão"	Acoplamento; inconsistência entre lados	Começar **unidirecional**; se bidirecional, sincronizar em um ponto
Ignorar multiplicidade	Estados inválidos (pedido vazio; duplicidades)	Validar em métodos de alteração (mínimos/máximos/únicos)
Confundir composição × agregação	Exclusão indevida ou órfãos	Decidir por **ciclo de vida** da parte
N–N sem classe de ligação	Perda de dados do vínculo	Introduzir classe de ligação quando houver atributos do relacionamento

FAQ (problema → resposta)

Posso usar List no getter e ainda assim proteger as regras?

Tecnicamente é possível, mas **arriscado**: qualquer código que receba a lista pode modificá-la. Prefira expor apenas leitura.

Quando realmente preciso de bidirecional?

Quando **ambos os lados** são consultados/manipulados diretamente pelo domínio (ex.: mostrar cursos de um aluno **e** alunos de um curso, com atualização coordenada).

Como garantir que Order não finalize vazio?

Mantendo método de verificação (CanFinish) e validando antes de finalizar; alternativamente, impondo a regra já na criação (ex.: primeiro item obrigatório).

Por que HashSet em N-N?

Para bloquear duplicatas de forma natural, simplificando a manutenção da consistência.

Olhando adiante (opcional)

LINQ para consultas

Consultas sobre coleções relacionadas usando LINQ para manipular dados de forma eficiente.

EF Core

Mapeamento de 1–1, 1–N e N–N (tabela de junção) usando Entity Framework Core.

Exceções e testes

Exceções e testes de unidade para reforçar invariantes e garantir a integridade dos dados.

Glossário e Encerramento

Associação

Vínculo conceitual entre classes.

Agregação

Todo–parte fraca, onde a parte vive sem o todo.

Composição

Todo–parte forte, onde a parte não faz sentido sem o todo.

Multiplicidade

Quantidade de instâncias em um relacionamento (0..1, 1, 0..*, 1..*, a..b).

Navegabilidade

Define quem conhece quem em um relacionamento (unidirecional ou bidirecional).

Invariante

Condição que deve ser sempre verdadeira durante a vida de um objeto.

Você agora tem, além das boas práticas, o **mapa dos riscos** e como mitigá-los diretamente no design do código. Na sequência do curso, os comentários de código passarão a ser em **inglês**, mantendo os identificadores já em inglês como neste guia.