Multiplicidade 0..1 e 1..1: Modelagem e Validação

Neste capítulo, transformaremos os princípios de associações estudados anteriormente em prática de código, concentrando nas multiplicidades 0..1 (opcional) e 1..1 (obrigatória e única). Nosso objetivo é projetar e validar relações com navegabilidade mínima, invariantes claros e encapsulamento robusto, prevenindo os problemas comuns de dados inconsistentes e regras de negócio quebradas.

A falha em modelar corretamente essas relações em código pode levar a cenários problemáticos. Por exemplo, permitir que um objeto principal seja associado a múltiplos objetos opcionais quando apenas um é permitido, ou deixar um objeto "mandatório" nulo, contrariando a lógica de negócio. Isso compromete a integridade do sistema e dificulta a manutenção.

Multiplicidade 0..1 (Opcional)

- **Significado:** "zero ou um". Um objeto pode ou não estar associado a outro. Ex: Uma pessoa pode ter um passaporte (ou não).
- **Representação em Código:** Uma referência anulável (nullable reference) ao objeto associado. Ex: `public Passport? Passport { get; private set; }`.
- **Validação:** A lógica para atribuir ou remover essa associação deve ser encapsulada, garantindo que o objeto opcional seja definido ou removido corretamente. Nenhuma validação é necessária para a ausência, apenas para a presença (se houver regras específicas).
- **Encapsulamento:** O método que define o objeto opcional deve verificar se já existe um, e o método de remoção deve invalidar a referência atual, mantendo a integridade.

Multiplicidade 1..1 (Obrigatória e Única)

- **Significado:** "exatamente um". Um objeto é sempre associado a um, e apenas um, outro objeto. Ex: Uma pessoa sempre tem um CPF.
- **Representação em Código:** Uma referência não anulável ao objeto associado. Idealmente, definida no construtor para garantir que a associação seja estabelecida na criação do objeto. Ex: `public CPF CPF { get; }`.
- **Validação:** Essencial garantir que a associação nunca seja nula e que, se houver, o objeto associado seja único e válido. Isso geralmente ocorre no construtor ou em um método de fábrica.
- **Imutabilidade:** Preferencialmente, torne a propriedade associada imutável após a criação para evitar alterações acidentais e garantir o invariante "sempre um".

Princípio chave: Encapsule a lógica de associação dentro das entidades, utilizando métodos controlados para adicionar ou remover as relações, validando-as de acordo com a multiplicidade definida. Isso garante que os objetos estejam sempre em um estado válido e consistente.

Boas práticas e os problemas que elas evitam

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.

Em 0..1/1..1, prefira Person → Passport; só crie o inverso com caso de uso explícito.

Encapsulando Vínculos 1:1 e 0..1: Protegendo a Integridade do Domínio

A modelagem adequada de relacionamentos de multiplicidade 1:1 (um para um) e 0..1 (zero ou um) é crucial para garantir a consistência e a integridade do estado dos objetos em um sistema. Falhas na encapsulamento e validação desses vínculos podem levar a dados inconsistentes, violação de regras de negócio e comportamentos imprevisíveis da aplicação.

Anti-padrão Comum

Utilizar propriedades mutáveis externamente para representar associações, como `public Passport? Passport { get; set; }`, permite que qualquer parte do código altere ou substitua a referência de forma silenciosa e sem validação.

Problemas Gerados

- **Quebra de Invariantes:** Permite a "emissão" de um segundo passaporte quando o domínio dita que uma pessoa só pode ter um.
- **Estados Inválidos:** Possibilita que um objeto passe para um estado logicamente inválido (ex: emitir um passaporte com dados vazios ou uma data de expiração passada).
- **Inconsistência:** Cria um cenário onde o estado do objeto pode não refletir a realidade do negócio, dificultando a depuração e manutenção.

Para combater esses problemas, a solução envolve o encapsulamento rigoroso da lógica de associação e validação. O objetivo é garantir que as regras de negócio sejam aplicadas no ponto de entrada da alteração do estado, mantendo o objeto sempre em um estado válido.

01

1. Encapsular a Lógica de Associação

Utilize um `private set` para a propriedade associada (ex: `Passport { get; private set; }`) e exponha um método público (ex: `IssuePassport`) para controlar a criação ou substituição do vínculo.

02

2. Validar na Fronteira

Dentro do método de associação, realize todas as validações necessárias nos dados de entrada (ex: número do passaporte não vazio, data de expiração futura) e na regra de negócio (ex: impedir a emissão de um segundo passaporte).

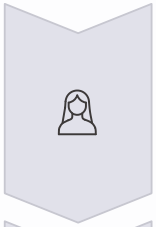
03

3. Tornar o Dependente Imutável

Idealmente, o objeto associado (ex: `Passport`) deve ser imutável, garantindo que, uma vez criado e validado, suas propriedades internas não possam ser alteradas, evitando inconsistências.

Validar Multiplicidade (0..1 e 1..1)

A correta validação da multiplicidade é fundamental para assegurar a integridade dos dados e o cumprimento das regras de negócio. Ignorar esses limites é um dos anti-padrões mais comuns, resultando em sistemas inconsistentes e difíceis de manter. A validação deve ocorrer no ponto de entrada da alteração do estado do objeto, garantindo que ele permaneça sempre em um estado válido.



Anti-Padrão Comum

Ignorar os limites da multiplicidade (ex.: permitir que uma `Pessoa` tenha mais de um `Passaporte` quando a regra é 0..1).

Efeitos Consequentes

Gera estados inválidos, violação de regras de negócio e a propagação de erros que só são detectados tardiamente em camadas posteriores da aplicação.

Solução Essencial

Realizar a validação rigorosa na "fronteira" – no método que cria, vincula ou troca o objeto dependente. Os princípios são:

- **1..1:** Exigir a existência e unicidade do dependente.
- **0..1:** Permitir a ausência, mas impedir a criação de valores "default" silenciosos.
- **Ambos:** Impedir duplicatas, garantindo a unicidade quando presente.

Passos Práticos para uma Validação Eficaz

01

Exigir Existência (1..1)

Para relações 1..1, o objeto dependente (ex: `CPF` de uma `Pessoa`) deve ser criado ou vinculado no construtor da entidade principal ou através de um método de fábrica. Isso assegura que o objeto principal nunca esteja em um estado "meio válido" sem seu dependente obrigatório.

02

Ausência Consciente (0..1)

Para relações 0..1, permita que a referência seja `null` ou ausente. O objeto dependente (ex: `Passaporte` de uma `Pessoa`) só deve ser vinculado explicitamente através de um método dedicado. Evite lógica que crie dependentes "silenciosamente" ou com valores padrão, pois isso pode mascarar a intenção do domínio.

03

Impedir Duplicatas (Ambos)

Tanto para 1..1 quanto para 0..1, o sistema deve garantir que, se um dependente já existe, qualquer tentativa de adicionar outro resulte em falha (ex: retornando `false` ou lançando uma exceção), sem alterar o estado atual. Isso assegura a unicidade e evita inconsistências.

Dica de Implementação: Utilize `private set` na propriedade do lado "dono" do vínculo. Crie métodos públicos (ex: `EmitirPassaporte()`) que encapsulam a lógica de validação: checagem de dados de entrada (número válido, data futura), e a regra de que o dependente "já existe?". Idealmente, o objeto dependente deve ser imutável após a criação para reforçar o 1:1. Para sistemas persistentes, adicione índices únicos no banco de dados para complementar e reforçar a regra de unicidade.

Distinguir composição × agregação

Anti-padrão comum	Efeitos	Solução
Usar composição quando as partes têm vida própria (ou vice-versa).	Exclusões indevidas (apagar Player ao remover de um Team), ou acúmulo de "órfãos" quando a parte deveria morrer com o todo.	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

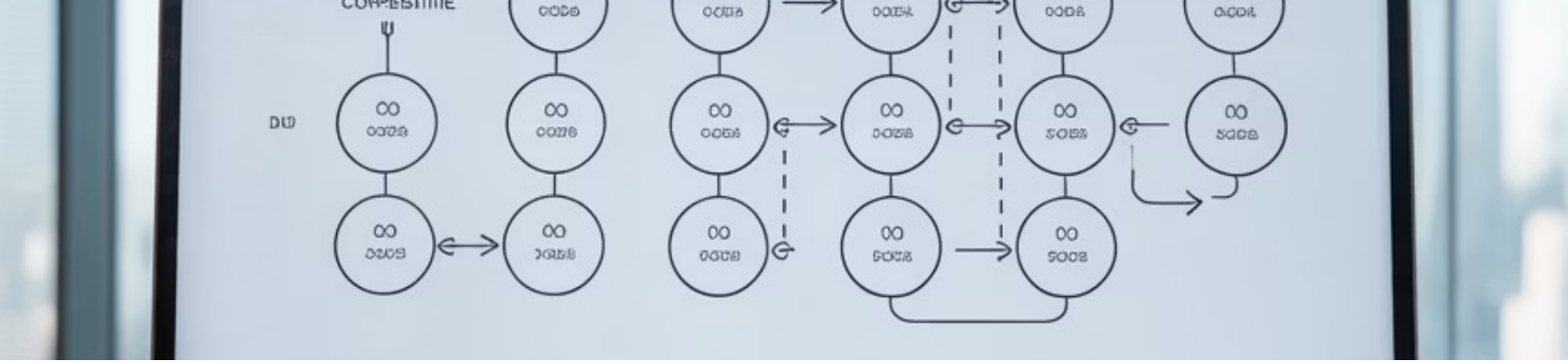
Navegabilidade mínima?

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 "Amanhã":** 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

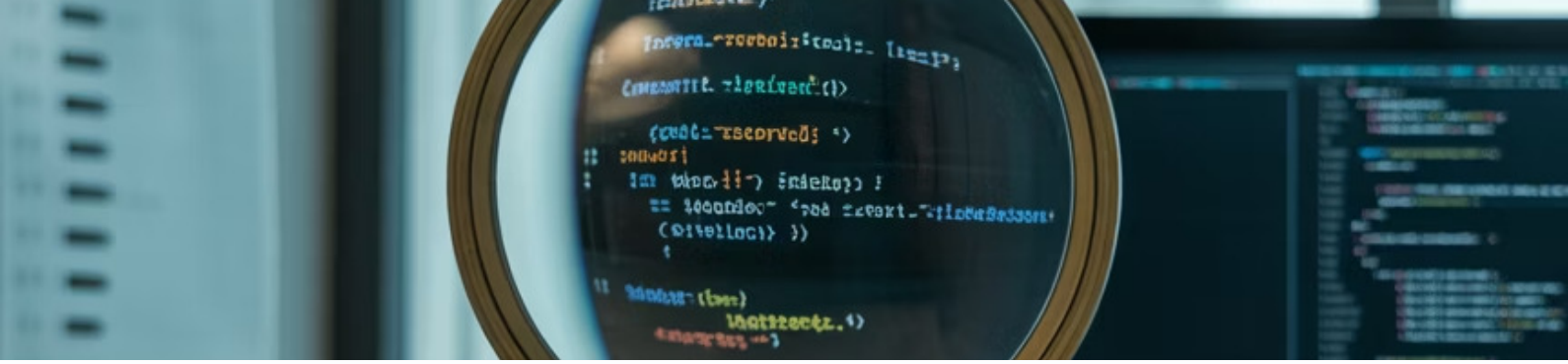
Uma implementação é considerada correta e robusta se atender aos seguintes critérios:

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.

Fechamento — Associações entre Classes: O Essencial para Levar

Ao final desta jornada sobre associações entre classes, é crucial consolidar os princípios que garantem designs robustos e sustentáveis. Este resumo condensa as lições chave e os antídotos para os problemas mais comuns.

Definição e Dimensões

Associações descrevem 'quem conhece quem' e 'em que quantidade'. A **Multiplicidade** (ex: 0..1, 1, 0..*, 1..*) e a **Navegabilidade** (unidirecional ou bidirecional) são as dimensões fundamentais que guiam as validações no código.

Composição vs. Agregação

Decida com base no ciclo de vida: **Composição** implica que a 'parte' depende integralmente do 'todo' (ex: um `OrderItem` não existe sem uma `Order`). **Agregação** permite que a 'parte' exista independentemente do 'todo' (ex: um `Player` pode existir sem um `Team`).

Encapsulamento e Validação na Fronteira

Evite `setters` públicos em associações. Centralize as mudanças de vínculo em um **método de domínio** (ex: `IssuePassport`) e mantenha **dados sensíveis imutáveis** (`get-only`) para garantir a integridade.

Navegabilidade Unidirecional como Padrão

Comece com associações unidirecionais. Torne-as bidirecionais apenas quando houver uma **necessidade clara e comprovada**, sempre sincronizando ambos os lados em um único ponto de controle.

Erros Frequentes e Seus Antídotos

A seguir, uma compilação dos erros mais comuns na modelagem de associações 0..1 e 1..1, juntamente com as estratégias para evitá-los. Estas práticas visam fortalecer a integridade dos dados e a robustez do sistema.

Expor o Vínculo com `setter` Público

Problema: Usar uma propriedade com `setter` público (ex: `public Passport? Passport { get; set; }`) permite que o vínculo seja alterado ou sobrescrito silenciosamente por qualquer parte do código, sem validação.

Antídoto: Utilize um `setter` privado e crie um método de domínio específico (ex: `IssuePassport`) para encapsular toda a lógica de vinculação e validação. Garanta que o objeto dependente (`Passport`) seja imutável.

Tornar Tudo Bidirecional "Por Via das Dúvidas"

Problema: Criar associações bidirecionais sem uma necessidade clara aumenta a complexidade, o acoplamento e o risco de inconsistências na sincronização entre as duas pontas do relacionamento.

Antídoto: Adote a navegabilidade mínima, começando com associações unidirecionais. Torne-as bidirecionais somente quando houver um caso de uso explícito e comprovado, e, nesse caso, centralize a sincronização do vínculo em um único ponto de controle.

Ignorar Limites da Multiplicidade

Problema: Não impor as restrições de multiplicidade no código (ex: permitir que uma pessoa tenha dois passaportes ou nenhum, quando o modelo exige um).

Antídoto: Implemente validações rigorosas na fronteira do domínio: para 1..1, exija a presença do dependente; para 0..1, permita a ausência. Em ambos os casos, impeça duplicatas para garantir a unicidade do vínculo conforme a regra de negócio.

Checklist Rápido de Validação (Associação Pessoa–Passaporte)

Use este checklist para revisar rapidamente a implementação de associações de multiplicidade 0..1/1..1 e garantir que as boas práticas foram aplicadas.

1

A propriedade `Person.Passport` possui um `setter` privado e só é modificada através de um método de domínio?

2

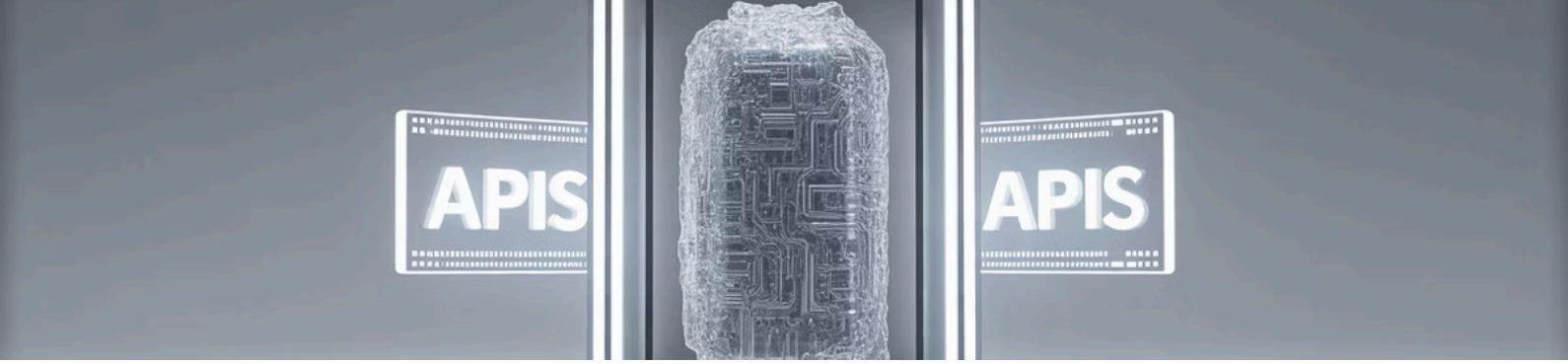
O método de domínio para vincular (ex: `IssuePassport`) impede a duplicação do passaporte e recusa dados inválidos (como número vazio ou data de expiração anterior à data atual)?

3

O objeto `Passport` é imutável (com propriedades `get-only`) e é instanciado em um estado sempre válido (validando no construtor)?

4

A navegabilidade é mínima (unidirecional por padrão) e, se bidirecional, a sincronização é realizada em um único ponto de controle?



5

Para associações 1..1, há uma garantia de que o dependente estará presente (seja no construtor da entidade principal, por um factory, ou imediatamente após a criação)?

6

Existe (opcionalmente, no banco de dados) um índice único para reforçar a unicidade da associação e prevenir duplicação em um nível persistente?

A Regra dos Três Cs para a Leitura Final

Para garantir que sua implementação está à prova de falhas, aplique a "Regra dos Três Cs" na sua revisão de código:

Clareza

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

Coesão

A lógica de emissão ou ligação da associação reside no lugar certo, preferencialmente dentro da entidade responsável (ex: `Person.IssuePassport`)?

Consistência

Após a criação do vínculo, o estado do objeto e de suas associações permanece válido e íntegro, sem 'atalhos' que possam violar as regras de domínio?

Checklist Rápido de Validação (1:1 "Pessoa-Passaporte")

- O atributo `Person.Passport` possui um `setter` privado e só é modificado via um método de domínio?
- O método de domínio impede a duplicação de passaportes e recusa dados inválidos (número vazio, data não-futura)?
- O objeto dependente (`Passport`) é imutável (propriedades `get-only`) e é instanciado sempre em um estado válido?
- A navegabilidade é mínima (evita-se dependência inversa se não houver um uso claro e justificado)?

Critérios de "Pronto e Robusto"

Considere uma associação correta por design e robusta quando:

1

Os **invariantes de negócio** são sempre mantidos (ex: 0 ou 1 passaporte válido por pessoa, com dados consistentes).

2

A **API é segura e clara**, oferecendo um único caminho para criar o vínculo, que nunca permite que o objeto entre em um estado inválido.

3

Dados sensíveis são imutáveis após a criação e a **navegabilidade é mínima** e intencional.

Mini-Desafios para Consolidar: Modelagem 0..1 e 1..1

Para solidificar a compreensão e a aplicação das boas práticas em associações 0..1 e 1..1, propomos uma série de mini-desafios práticos. O objetivo é reforçar a importância da validação rigorosa e do encapsulamento para garantir a integridade dos objetos e a robustez do sistema.

1

Emissão Controlada (Associação 0..1)

Implemente o método `Person.IssuePassport(string number, DateTime expiration)` que retorna um `bool`. Este método deve:

- Validar o `number` (não vazio) e a `expiration` (data futura em relação a `UtcNow.Date`).
- Impedir a duplicação: se a pessoa já possui um passaporte, a operação deve falhar (retornar `false`) sem alterar o estado existente.
- Garantir que a classe `Passport` tenha propriedades `get-only`, tornando-a imutável após a criação.

Este desafio foca em garantir que a emissão de um passaporte seja um processo controlado e seguro, mantendo a pessoa em um estado válido.

2

Obrigatoriedade (Associação 1..1)

Crie uma variação da classe `Person` (ou um novo construtor/factory method como `Person.CreateWithPassport(...)`) onde a pessoa **já nasce com um passaporte**. O objetivo é eliminar qualquer estado "meio válido" durante a criação, assegurando que o invariante de "uma pessoa sempre tem um passaporte" seja mantido desde o início.

3

Substituição Segura do Dependente

Adicione um método como `ReplaceLostPassport(string newNumber, DateTime newExpiration)` (ou `RenewPassport(...)`). Este método deve:

- Permitir a substituição apenas se a pessoa **já possui** um passaporte.
- Validar os novos dados (`newNumber` e `newExpiration`) seguindo as mesmas regras de emissão.
- Ser uma operação **atômica** (o passaporte só é substituído se todas as validações passarem) e retornar `bool`.

Isso demonstra controle explícito sobre a mutação do vínculo.

4

Navegabilidade Mínima e Sincronização

Para a associação `Person-Passport`:

- **Versão A:** Mantenha a navegabilidade unidirecional (`Person → Passport` apenas).
- **Versão B (Opcional):** Torne a associação bidirecional (`Person ↔️ Passport`) **somente se conseguir identificar um caso de uso explícito e justificável**. Se o fizer, certifique-se de que a sincronização do vínculo (quando `Person.Passport` é setado, `Passport.Person` também é atualizado, e vice-versa) seja centralizada em um **único ponto de controle**.

Este desafio enfatiza a necessidade de design consciente e a evitação de acoplamento desnecessário.

5

Validação na Fronteira (Checklist)

Revise as implementações, garantindo que as seguintes regras sejam estritamente aplicadas na "fronteira" do domínio (métodos que alteram o estado):

- Rejeite números de passaporte vazios.
- Exija data de expiração futura.
- Impeça a duplicação de passaportes (uma pessoa, um passaporte).
- Mantenha o `setter` da propriedade `Passport` na classe `Person` como `private`.

A validação na fronteira é a primeira linha de defesa contra estados inválidos.

6

Extra (Opcional/Persistência)

Se aplicável ao seu contexto, crie um índice único no banco de dados para a coluna que representa o vínculo `Person-Passport`. Isso reforça a unicidade da associação em nível persistente, prevenindo inconsistências mesmo fora da lógica da aplicação.

Mensagem Final: A Essência da Modelagem Robusta

> **Associe com propósito.** Em relacionamentos 0..1 e 1..1, valide onde o estado muda, mantenha a navegabilidade mínima e encapsule rigorosamente a mutação em métodos de domínio. Assim, seu modelo permanece confiável, fácil de testar e de evoluir — livre de acoplamento desnecessário e de surpresas indesejadas.