Implementando Associações 0..1 e 1:1 em C#

Uma atividade prática focada em consolidar o entendimento de multiplicidade e navegabilidade mínima através de código C# bem estruturado, garantindo invariantes de domínio por design.

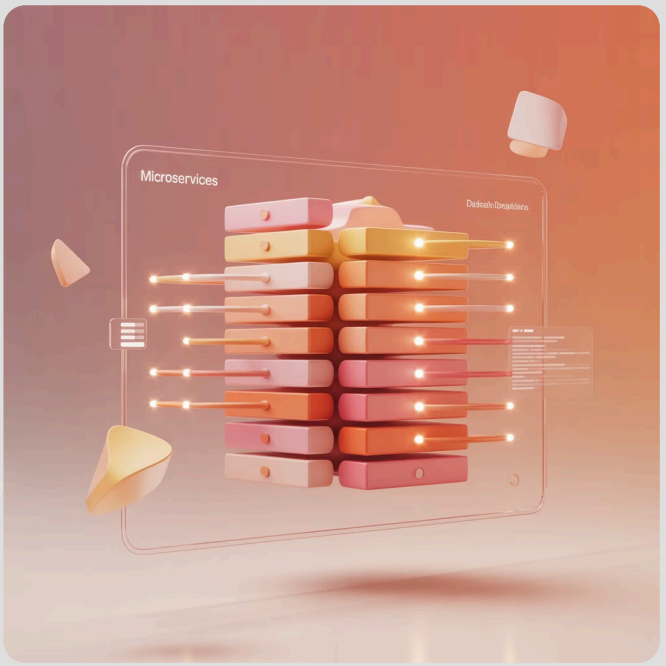
Propósito da Atividade

Objetivo Principal

Consolidar em **código C#** o entendimento de multiplicidade e navegabilidade mínima para os casos **0..1** (opcional) e **1:1** (obrigatório).

O foco é projetar **invariantes de domínio** e garantir estes invariantes por design através de encapsulamento, validação na fronteira e imutabilidade útil.

Evitamos anti-padrões através de práticas sólidas de programação orientada a objetos, mantendo o código limpo e as regras de negócio bem definidas.



Regras Norteadoras Fundamentais

Direção Mínima

Modele apenas o sentido necessário para o caso de uso. Só torne bidirecional quando houver justificativa explícita.

Multiplicidade 0..1

No máximo um dependente; ausência é válida; atribuição **controlada**; **sem setter público**.

Multiplicidade 1:1

Dependente **sempre presente** e único; vínculo garantido **na criação**; **imutável** após criado.

Validação na Fronteira

Entradas inválidas não entram no estado do objeto. Validação acontece nos pontos de entrada.

Dinâmica de Trabalho em Equipe

01

Formação dos Grupos

Grupos de 3-4 pessoas com papéis definidos: Arquiteto(a) de Domínio, Implementador(a), Testador(a) e Porta-voz.

02

Ritmo de Trabalho

3 rodadas curtas: Modelar → Codar 0..1 → Codar 1:1, cada uma fechando com defesa de 3-5 minutos.

03

Defesa Técnica

Explicar por que o código garante a multiplicidade e a direção mínima; apontar invariantes e onde são verificados.

Cenários de Implementação

Escolha um dos cenários abaixo para desenvolver. Todos oferecem um vínculo 0..1 e um vínculo 1:1.

Paciente - Leito / Prontuário

- **0..1:** Patient → Bed (pode não ter)
- **1:1:** Patient → MedicalRecord (sempre tem)

Veículo - Rastreador / Placa

- **0..1:** Vehicle → Tracker
- **1:1:** Vehicle → LicensePlate

Conta - Cartão Primário / Documento Fiscal

- **0..1:** Account → PrimaryCard
- **1:1:** Account → TaxId

Dica: Justifique se uma composição faria sentido (parte não vive sem o todo) ou se é apenas associação (vida própria).

Missão A: Modelagem e Invariantes

1 ### Diagrama Mínimo

Esboce um diagrama com apenas o necessário, focando nas relações essenciais entre as classes.

2 ### Invariantes por Vínculo

0..1: 0 ou 1, nunca >1
1:1: existe e único, nunca nulo

3 ### Validações de Fronteira

Códigos não vazios, datas futuras, formatos válidos - tudo verificado na entrada.

4 ### Navegabilidade

Registre a navegabilidade adotada e quando tornaria bidirecional, se necessário.

Missão B: Implementação do 0..1

Objetivo: Vincular um dependente opcional de forma encapsulada

Diretrizes Mínimas

- Propriedade **anulável** com `private set`
- Método de domínio para atribuir com validação
- Proibir duplicação (não sobrescrever silenciosamente)
- Dependente preferencialmente **imutável útil**

```
public Bed? Bed { get; private set; }

public bool AssignBed(Bed bed)
{
    if (bed is null) return false;
    if (Bed != null) return false;
    if (!bed.IsAvailable) return false;

    Bed = bed;
    return true;
}
```

Tempo estimado: 25-35 minutos

Assinaturas de Referência para 0..1

Account - PrimaryCard

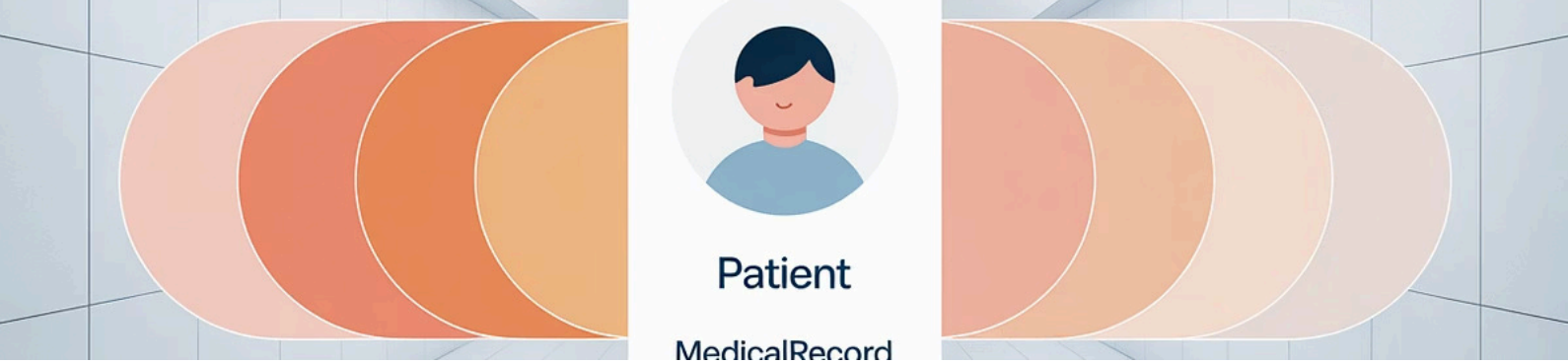
```
Account.SetPrimaryCard(Card c)
```

Patient - Bed

```
public bool AssignBed(Bed bed);
public void UnassignBed();
```

Vehicle - Tracker

```
Vehicle.AttachTracker(Tracker t)
```



⚠ **Importante:** Se o domínio permitir remoção, implemente método específico para isso.

Missão C: Implementação do 1:1

Objetivo: Garantir dependente obrigatório desde a criação

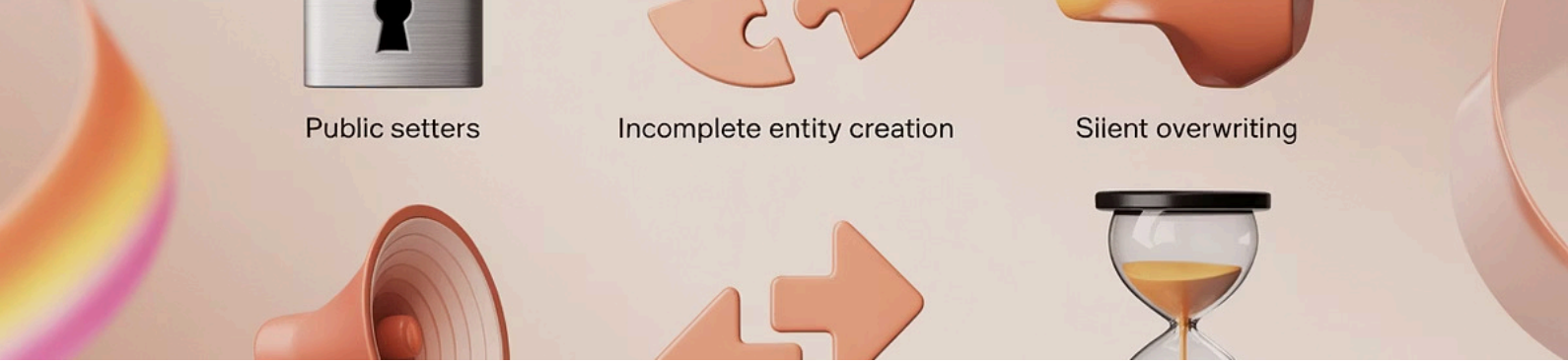
Diretrizes Mínimas

- Receber o dependente **no construtor** (ou usar factory)
- Propriedade **get-only** (imutável pós-criação)
- Validar **não-nulo** e regras de domínio na criação

Lembre-se: Mantenha o exemplo enxuto nos slides. A equipe pode ter versões mais completas no repositório.

```
public sealed class Patient
{
   public string Name { get; }
   public MedicalRecord Record { get; }

   public Patient(string name, MedicalRecord record)
   {
      Name = string.IsNullOrWhiteSpace(name)
         ? throw new ArgumentException("Name required")
         : name;
      Record = record ?? throw new ArgumentNullException(nameof(record));
   }
}
```

Tempo estimado: 25-35 minutos



Estratégia de Testes

Testes Manuais (Mínimo)

1. Criar entidade raiz com dependente **1:1** válido → deve instanciar
2. Tentar criar sem dependente **1:1** → deve falhar
3. Atribuir dependente **0..1** uma vez → deve funcionar
4. Atribuir **0..1** segunda vez → deve falhar
5. Remover **0..1** (se permitido) → estado volta a null

Testes Unitários (Sugestão)

Use xUnit/NUnit com casos de borda: nulos, strings vazias, datas inválidas, formatos incorretos.

Teste especialmente as validações de fronteira e os invariantes de multiplicidade.

Critérios de Aceite

Multiplicidade 0..1

- Propriedade anulável com **private set**
- Método de domínio **impede duplicação**
- Validação na borda (nulos, formatos, regras)
- Estado final: **0 ou 1**, nunca >1

Multiplicidade 1:1

- Dependente exigido no **construtor/factory**
- Propriedade **get-only** (imutável)
- Nenhum caminho alternativo para troca

Critérios Gerais

- **Direção mínima** preservada
- **Invariantes documentados** no código
- Sem setters públicos em vínculos

Anti-Padrões Proibidos

❌ Setters Públicos

`public set` em propriedades de associação (0..1 ou 1:1) quebra o encapsulamento.

❌ Criação Incompleta

Criar entidade **1:1** com dependente nulo e "preencher depois" viola invariantes.

❌ Sobrescrita Silenciosa

Sobrescrever dependente **0..1** silenciosamente causa perda de rastreabilidade.

❌ Bidirecional Desnecessário

Tornar modelo bidirecional "por via das dúvidas" sem sincronização clara.

❌ Validação Tardia

Validar "no fim" em vez de **na fronteira** permite estados inválidos.

Entregáveis por Equipe

01

README Curto

≤ 1 página: cenário escolhido, invariantes, decisões de navegabilidade e justificativas.

02

Diagrama Mínimo

Imagem simples mostrando as relações essenciais entre as classes.

03

Código C#

Duas classes principais com os respectivos dependentes (0..1 e 1:1).

04

Roteiro de Teste

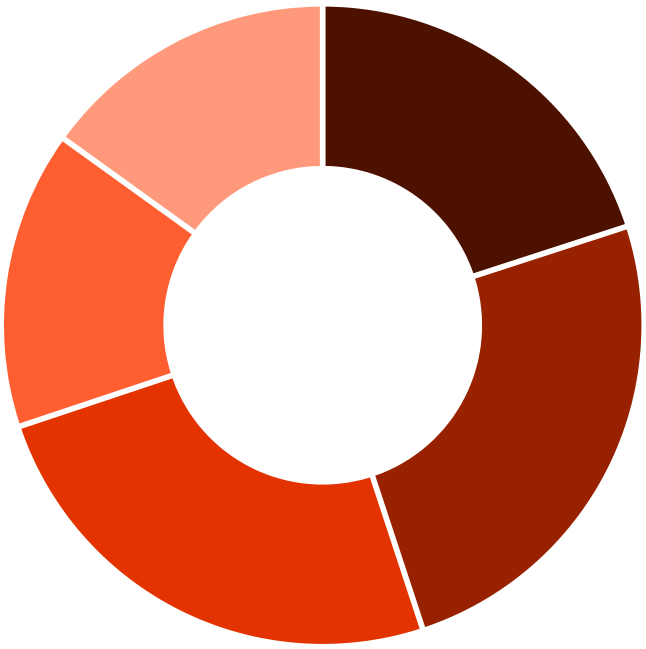
5 passos do teste manual com saída/prints demonstrando funcionamento.

05

Defesa Técnica

3-5 minutos: invariantes, fronteiras, direção mínima, trade-offs composição × associação.

Sistema de Avaliação



Modelagem | Implementação 0..1 | Implementação 1:1 | Qualidade Código | Teste & Defesa

Total: **100 pontos** distribuídos entre modelagem e invariantes, implementações das multiplicidades, qualidade do código, e capacidade de teste e defesa técnica.

Detalhamento dos Critérios

1

Modelagem e Invariantes (20 pts)

Completude do modelo, clareza na definição dos invariantes, e aplicação correta do princípio de direção mínima.

2

Implementação 0..1 (25 pts)

Encapsulamento adequado, impedimento efetivo de duplicação, e validações robustas na fronteira.

3

Implementação 1:1 (25 pts)

Criação obrigatória garantida, imutabilidade pós-criação, e ausência de atalhos que quebrem invariantes.

4

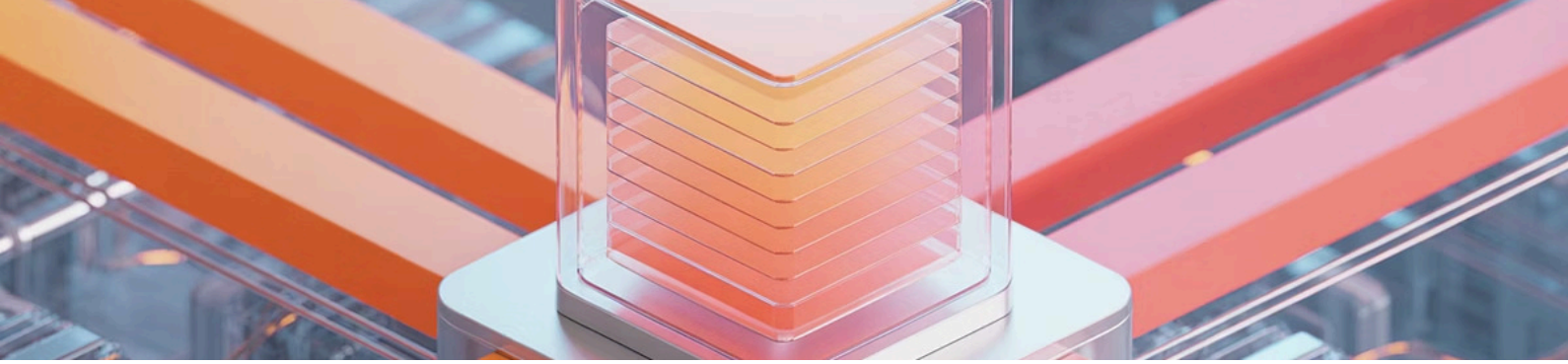
Qualidade do Código (15 pts)

Nomes expressivos, simplicidade na implementação, comentários úteis sem verborragia desnecessária.

5

Teste & Defesa (15 pts)

Cobertura de casos de borda, coerência na apresentação, e justificativas técnicas bem fundamentadas.



Extensões Opcionais

Se sobrar tempo, considere implementar estas funcionalidades avançadas:

Bidirecional Justificado

Implemente o inverso da relação com um único ponto de sincronização, garantindo consistência em ambas as direções.

Factory Avançado

Crie factories com validações avançadas, incluindo normalização e parsing de dados de entrada.

Persistência Fake

Implemente persistência in-memory para demonstrar unicidade fora do objeto (ex.: verificar placa duplicada).

Guia de Apresentação

1

Cenário Escolhido

Explique o cenário e por que as multiplicidades 0..1 e 1:1 fazem sentido no contexto.

2

Invariantes e Garantias

Demonstre onde os invariantes são garantidos no código e como são verificados.

3

Trechos-Chave

Foque em 2-3 métodos principais, destacando fronteira e encapsulamento.

4

Casos de Borda

Apresente um caso de borda específico que a equipe capturou e tratou.

5

Evolução Bidirecional

Explique até onde iriam se o vínculo precisasse ser bidirecional e a justificativa.

Tempo de apresentação: 3-5 minutos por equipe

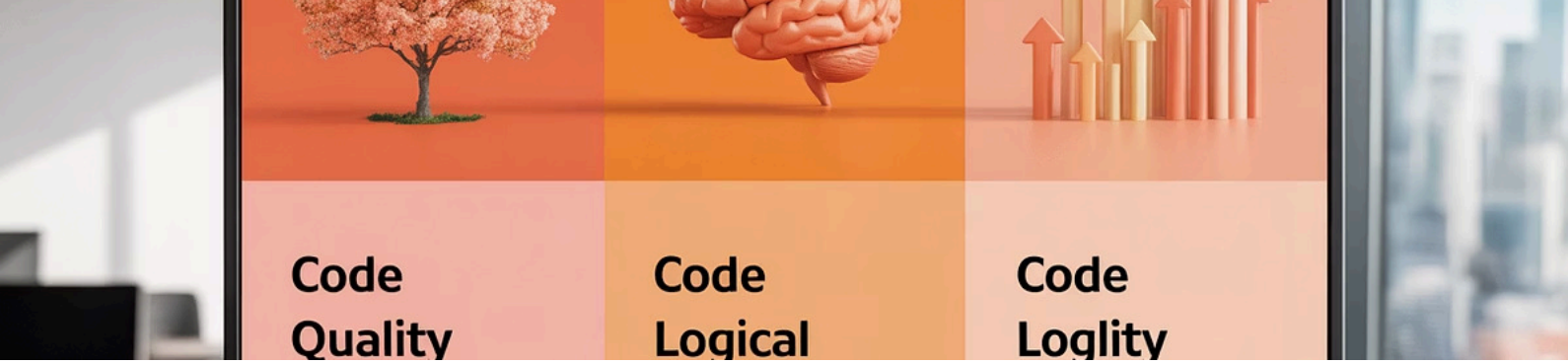
Exemplo Prático Completo

Classe Patient Completa

```
public sealed class Patient
{
    public string Name { get; }
    public MedicalRecord Record { get; } // 1:1
    public Bed? Bed { get; private set; } // 0..1

    public Patient(string name, MedicalRecord record)
    {
        Name = string.IsNullOrWhiteSpace(name)
            ? throw new ArgumentException("Name required")
            : name;
        Record = record ??
            throw new ArgumentNullException(nameof(record));
    }

    public bool AssignBed(Bed bed)
    {
        if (bed is null) return false;
        if (Bed != null) return false;
        if (!bed.IsAvailable) return false;

        Bed = bed;
        return true;
    }

    public void UnassignBed() => Bed = null;
}
```

Pontos-Chave do Exemplo

- MedicalRecord obrigatório no construtor
- Bed opcional com controle de atribuição
- Validações na fronteira de entrada
- Propriedades imutáveis onde apropriado
- Métodos de domínio para operações controladas



Observações Finais

Código Mínimo e Legível

Mantenha o código **mínimo e legível**, sem poluir os slides com implementações desnecessariamente complexas.

Demonstre o Raciocínio

Esforce-se para demonstrar o raciocínio: "o que garante que nunca teremos 2 dependentes?", "onde eu proíbo nulos?"

Objetivo Final

Sair com **duas associações corretas por design** e um repertório de defesa técnica para o restante da disciplina.

Sucesso na Implementação

Resultados Esperados

Ao final desta atividade, você terá:

- Domínio prático das multiplicidades 0..1 e 1:1
- Código C# que garante invariantes por **design**
- Experiência em validação na fronteira
- Capacidade de defender decisões técnicas
- Repertório para aplicar em projetos futuros

Esta base sólida será fundamental para implementações mais complexas ao longo da disciplina, sempre mantendo os princípios de **encapsulamento**, **imutabilidade útil** e **direção mínima**.

100	3	5
Pontos Totais	Cenários	Entregáveis
Distribuídos entre todos os critérios	Opções para escolher	Por equipe

Desafios de Implementação

A seguir, uma série de desafios práticos para aplicar os conceitos de associações 0..1 e 1:1, garantindo o uso correto de tipos anuláveis e o respeito aos invariantes do domínio. Cada cenário exige uma modelagem cuidadosa e uma implementação robusta, seguindo as diretrizes apresentadas anteriormente.

01

Carro e Motor (1:1)

Modele a relação entre um `Carro` e seu `Motor`. Um carro sempre possui um motor, e um motor pertence a exatamente um carro. Garanta que um `Carro` não possa ser criado sem um `Motor` e que o motor não possa ser alterado após a criação do carro.

02

Pessoa e Carteira de Motorista (0..1)

Crie classes para `Pessoa` e `CarteiraMotorista`. Uma pessoa pode ou não ter uma carteira de motorista. A carteira, se existir, pertence a uma única pessoa. Implemente métodos para atribuir e remover a carteira, garantindo que os estados sejam sempre consistentes e que a remoção seja permitida pelo domínio.

03

Livro e Capa (1:1)

Defina a associação entre um `Livro` e sua `Capa`. Todo livro deve ter uma capa, e uma capa é exclusiva de um livro. Assegure que a criação de um `Livro` exija a especificação de uma `Capa`, e que esta associação seja imutável para a instância do livro.

04

Produto e Descrição Detalhada (0..1)

Implemente a relação onde um `Produto` pode ter uma `DescricaoDetalhada` opcional. Se presente, a descrição pertence apenas àquele produto. Permita que a descrição seja adicionada ou removida de um produto existente, utilizando tipos anuláveis de forma adequada.

05

Funcionário e Crachá (1:1)

Modele a associação entre um `Funcionario` e seu `Cracha`. Um funcionário recém-contratado sempre recebe um crachá, e um crachá é emitido para um único funcionário. Preveja a criação conjunta e assegure a imutabilidade da associação do crachá ao funcionário.

06

Cliente e Endereço de Entrega Preferencial (0..1)

Crie uma classe `Cliente` que pode ter um `Endereco` de entrega preferencial. Este endereço é opcional, mas se definido, é exclusivo daquele cliente. Implemente a lógica para definir e alterar o endereço preferencial, incluindo a possibilidade de removê-lo.

7

Cliente e Endereço de Entrega Preferencial (0..1)

Crie uma classe `Cliente` que pode ter um `Endereco` de entrega preferencial. Este endereço é opcional, mas se definido, é exclusivo daquele cliente. Implemente a lógica para definir e alterar o endereço preferencial, incluindo a possibilidade de removê-lo.

8

Pedido e Cupom de Desconto (0..1)

Modele um `Pedido` que pode, opcionalmente, ter um `CupomDesconto` aplicado. Um cupom específico, se usado, é associado a um único pedido. Permita a aplicação e remoção de cupons, com as devidas validações na fronteira de entrada.

9

Apartamento e Inquilino Principal (0..1)

Desenvolva classes `Apartamento` e `Inquilino`. Um apartamento pode estar vago ou ter um `Inquilino` principal. Um inquilino só pode ser principal em um único apartamento por vez. Garanta a correta atribuição e liberação do inquilino no apartamento.

100

Computador e Placa de Vídeo Dedicada (0..1)

Defina a relação onde um `Computador` pode ou não ter uma `PlacaVideoDedicada`. Uma placa, se instalada, é exclusiva daquele computador. Implemente a funcionalidade para instalar e desinstalar a placa, considerando os cenários de hardware disponível e compatibilidade.

1

Usuário e Dados Biométricos (1:1)

Crie classes para `Usuario` e `DadosBiometricos`. Todo usuário deve possuir dados biométricos para autenticação, e estes dados são exclusivos de um usuário. Garanta que a criação de um `Usuario` inclua a configuração de seus `DadosBiometricos`, e que essa ligação seja forte e imutável.