CONSOLIDANDO A SINCRONIZAÇÃO BIDIRECIONAL E INVARIANTES

Este estudo aprofunda a implementação de associações 1:N bidirecionais entre `Department` e `Employee` em C#, destacando a importância da sincronização controlada e de um ponto único de mutação. Priorizamos a navegação bidirecional apenas quando os casos de uso justificam a complexidade, garantindo que a sincronização seja robusta e minuciosamente testada.

SINCRONIZAÇÃO CONTROLADA

O método `AddEmployee` é o ponto único de mutação, garantindo que a adição de um funcionário ao departamento também atualize a referência de departamento no objeto `Employee`. Isso mantém a coerência entre os dois lados da associação.

TESTES XUNIT COMO CONTRATOS

Uma suíte de testes xUnit valida rigorosamente as regras do modelo: verifica a coerência das referências após a adição, assegura que a remoção de um funcionário anula corretamente sua associação com o departamento, e confirma que tentativas duplicadas de adicionar o mesmo funcionário são ignoradas para preservar a integridade da coleção.

ENCAPSULAMENTO E INVARIANTES

A abordagem adotada reforça práticas de encapsulamento de coleções e a gestão de valores derivados sem cache, elementos cruciais para estabelecer e manter invariantes de domínio. Isso forma uma base sólida para cenários mais complexos, como associações N:M e regras transversais, promovendo um código seguro e alinhado às regras de negócio.

Dominar estas práticas assegura que o modelo de domínio seja não apenas funcional, mas também resiliente a mudanças e pronto para evoluir com segurança.

1:N BIDIRECIONAL COM SINCRONIZAÇÃO

Objetivo: Manter dois lados coerentes (Department ↔️ Employee) com ponto único de mutação e sincronização automática.

Essência: Navegação bidirecional apenas quando há casos de uso fortes que justifiquem a complexidade adicional. A sincronização deve ser controlada e testada.

IMPLEMENTAÇÃO DEPARTMENT E EMPLOYEE

CLASSE DEPARTMENT

```
using System.Collections.ObjectModel;
using Associations.Domain.DepartmentAggregate;

public class Department
{
    private readonly List<Employee> _employees = [];
    public ReadOnlyCollection<Employee> Employees =>
        _employees.AsReadOnly();

    public void AddEmployee(Employee e)
    {
        if (_employees.Contains(e))
        {
            return;
        }

        _employees.Add(e);
        e.AssignDepartment(this);
    }
}
```

CLASSE EMPLOYEE

```
public sealed class Employee
{
 public Department? Department { get; private set; }

 internal void AssignDepartment(Department d)
 {
    if (Department == d)
       return;
    Department = d;
 }

 internal void ClearDepartment(Department expected)
 {
    if (Department == expected)
       Department = null;
 }
}
```

DOBRANDO A COBERTURA DE TESTES: ESQUELETOS PARA OUTROS LABS

Para garantir a robustez e a manutenibilidade do nosso sistema, é crucial estender a cobertura de testes para outros modelos de domínio. A seguir, apresentamos esqueletos para arquivos de especificação de teste, com exemplos de métodos a serem implementados, focando nas invariantes e comportamentos esperados de cada associação.

`DepartmentSpecs.cs` (1:N BIDIRECIONAL)

Estes testes verificam a sincronização e consistência da associação bidirecional entre `Department` e `Employee`, assegurando que as operações de adição e remoção mantenham ambos os lados da relação atualizados.

`AddEmployee_SincronizaLados`

Verifica se, ao adicionar um funcionário a um departamento, a propriedade `Department` do objeto `Employee` é automaticamente atualizada para apontar para o departamento correto.

`RemoveEmployee_AnulaDepartamentoNoEmployee`

Assegura que, ao remover um funcionário de um departamento, a propriedade `Department` do objeto `Employee` é definida como nula ou o departamento anterior, refletindo a remoção da associação.

`AddEmployee_SincronizaLados`

Verifica se, ao adicionar um funcionário a um departamento, a propriedade `Department` do objeto `Employee` é automaticamente atualizada para apontar para o departamento correto.

`RemoveEmployee_AnulaDepartamentoNoEmployee`

Assegura que, ao remover um funcionário de um departamento, a propriedade `Department` do objeto `Employee` é definida como nula ou o departamento anterior, refletindo a remoção da associação.

`AddEmployee_IgnoraDuplicado`

Confirma que a adição de um funcionário que já pertence ao departamento não resulta em duplicação na coleção de funcionários do departamento.

IMPLEMENTAÇÃO DETALHADA DOS TESTES DE `DEPARTMENTSPECS`

Para consolidar nossa compreensão sobre a associação 1:N bidirecional entre `Department` e `Employee`, apresentamos a implementação completa dos testes de especificação para a classe `Department`. Estes testes validam a lógica de sincronização dos lados da associação, garantindo que as regras de negócio sejam mantidas em todas as operações.

```
using Xunit;

namespace Associations.Domain.Tests
{
    public class DepartmentSpecs
    {
        [Fact]
        public void AddEmployee_SincronizaLados()
        {
            var d = new new Department("Engenharia");
            var e = new Employee("Ana Silva");

            d.AddEmployee(e);

            Assert.Contains(e, d.Employees);
            Assert.Same(d, e.Department);
        }

        [Fact]
        public void RemoveEmployee_AnulaDepartamentoNoEmployee()
        {
            var d = new Department("Dados");
            var e = new Employee("João Souza");
            d.AddEmployee(e);
            var removed = d.RemoveEmployee(e);

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

```
        [Fact]
        public void AddEmployee_IgnoraDuplicado()
        {
            var d = new Department("Produto");
            var e = new Employee("Rita Lima");

            d.AddEmployee(e);
            d.AddEmployee(e); // tentativa duplicada

            Assert.Single(d.Employees);
            Assert.Same(d, e.Department);
        }
    }
}
```

Analisando os testes acima, podemos observar como as invariantes da nossa associação bidirecional são verificadas, garantindo a coerência entre os objetos `Department` e `Employee`.

SINCRONIZAÇÃO BIDIRECIONAL

O teste `AddEmployee_SincronizaLados` valida que, ao adicionar um `Employee` a um `Department`, a referência para o `Department` no objeto `Employee` é atualizada corretamente. Isso é crucial para manter a consistência em ambos os lados da associação.

REMOÇÃO COERENTE

O teste `RemoveEmployee_AnulaDepartamentoNoEmployee` verifica que a remoção de um funcionário de um departamento não apenas o remove da coleção do departamento, mas também garante que a propriedade `Department` do `Employee` seja definida como `null`, refletindo a desassociação completa.

TRATAMENTO DE DUPLICATAS

O método `AddEmployee_IgnoraDuplicado` assegura que a tentativa de adicionar um funcionário que já faz parte do departamento não resulte em duplicação na coleção de funcionários. A coleção `Employees` deve manter apenas uma instância de cada funcionário, conforme esperado para associações bem modeladas.

CONSOLIDANDO AS ASSOCIAÇÕES E INVARIANTES DO DOMÍNIO

O percurso que cobrimos até agora representa um marco crucial na construção de software robusto e orientado a objetos. Exploramos como as **invariantes de domínio**, quando codificadas diretamente nas classes, protegem a integridade dos dados, e como as **coleções encapsuladas** evitam manipulações externas indevidas. Discutimos também a importância de **valores derivados sem cache**, garantindo que o estado do objeto seja sempre consistente e refletindo a realidade do negócio em tempo real.

A base para essa solidez foi firmada com a introdução do `Money` como um Value Object imutável, a modelagem da composição 1:N entre `Order` e `OrderItem`, e o estabelecimento de uma associação 1:N bidirecional entre `Department` e `Employee`. Estes exemplos serviram como **contratos vivos**, onde os testes unitários validam continuamente o comportamento esperado e as regras de negócio, assegurando que qualquer refatoração futura seja segura e previsível.

Com estes fundamentos bem estabelecidos, estamos agora preparados para avançar para associações mais complexas, como as relações N:M, a inclusão de qualificadores e a implementação de regras de domínio transversais. As próximas etapas focarão em aprimorar a sincronização bidirecional, finalizar os testes de remoção/movimentação e na padronização de mensagens e exceções, solidificando ainda mais as práticas de design e garantindo um "trilho completo" para a prática e evolução contínua da disciplina.