Trabalho Prático: Desenvolvimento de Aplicação Console ou WebAPI com Testes Unitários

Valor: 3 pontos

Em dupla ou até três pessoas

✨ Objetivo

Desenvolver uma aplicação console app ou API em .NET utilizando boas práticas de programação orientada a objetos e implementação de testes automatizados com xUnit, Moq, FluentAssertions, Faker e Fixture.

🔢 Requisitos Mínimos da Aplicação

- Projeto Console App ou Web API
- Pelo menos 4 classes de domínio (ex: Cliente, Produto, Item)
  - *Classes como Base.cs não serão consideradas para essa contagem*
- Pelo menos 3 serviços com lógica de negócio significativa
- Pelo menos 2 repositórios com interface (ex: IClienteRepository, IProductRepository)
  - *Criações adicionais de serviços e repositórios com testes serão bem vistas*
- Utilizar banco de dados InMemory para simular a persistência dos dados

🧠 Requisitos dos Testes Automatizados

- Utilizar o framework xUnit
- Para cada método, criar no mínimo:
  - 1 teste do caminho feliz (valores válidos)
  - 1 teste com espera de erro ou exceção
  - *Criação de cenários adicionais será bem vista*
- Utilizar InlineData em pelo menos 2 testes de cada classe
- Utilizar:
  - Moq ou NSubstitute para simular dependências
  - FluentAssertions com pelo menos 3 validações por teste
  - Faker para gerar dados aleatórios
  - Fixture para reutilização de configurações de teste

📝 Sugestões de Cenários

▶ Sistema de Clientes e Pedidos

- Classes: Cliente, Pedido, Produto

- Serviços: ClienteService, PedidoService, ProdutoService

- Repositórios: IClienteRepository, IPedidoRepository

- Regras Possíveis:

- • Adicionar cliente com nome e e-mail válidos
- • Impedir e-mail duplicado ou inválido
- • Criar pedido com valor positivo e cliente existente

▶ Sistema de Livros e Empréstimos

- Classes: Livro, Emprestimo

- Serviços: LivroService, EmprestimoService

- Repositórios: ILivroRepository

- Regras Possíveis:

- • Validar disponibilidade de livro
- • Calcular dias de atraso na devolução

▶ Sistema de Reservas de Sala

- Classes: Sala, Reserva, Pessoa

- Serviços: SalaService, ReservaService

- Repositórios: ISalaRepository, IReservaRepository

- Regras Possíveis:

- • Criar reserva se não houver conflito de horários
- • Validar capacidade da sala

▶ Sistema de Garagem de Veículos

- Classes: Veiculo, Vaga, Estacionamento

- Serviços: EstacionamentoService, VeiculoService

- Repositórios: IVeiculoRepository, IVagaRepository

- Regras Possíveis:

- • Estacionar veículo em vaga livre
- • Impedir entrada se o veículo já estiver estacionado
- • Calcular tempo de permanência e valor a pagar

▶ Sistema de Controle de Vendas

- Classes: Produto, Venda, ItemVenda

- Serviços: VendaService, ProdutoService

- Repositórios: IVendaRepository, IProdutoRepository

- Regras Possíveis:

- • Criar venda com múltiplos itens
- • Calcular total da venda com desconto aplicado
- • Impedir venda com estoque insuficiente

◇ Orientações Finais

- Nomear os projetos separadamente: Ex: MinhaApp, MinhaAppService, MinhaApp.Tests
- Organizar o código em camadas: Domain, Services, Repositories, Program
- Utilizar boas práticas de codificação:
  - Responsabilidade única (SRP)
  - Injeção de dependência (DI)
  - Separação clara de responsabilidades
- Garantir cobertura mínima com testes significativos (80% — explicado em sala)
- Utilizar Clean Code
- Criar um documento README explicando o projeto
- Criar um repositório no GitHub e enviar o link para: ricardoalves@uniaraxa.edu.br
  - (caso o AVA ainda não esteja ativo)
- Fique à vontade para adicionar funcionalidades e expandir a aplicação com outras regras e serviços úteis!