Trechos do Livro Jornada Ágil de Qualidade (Brasport, 2019)

Foi muito bom ter participado deste livro e ter trocado muitos exemplos práticos: vale muito a leitura 😊.

Pelo que entendi você realiza os testes unitários automatizados dos seus componentes, certo?

O foco dos testes de componentes são os unitários.

Veja se você encaixa seus testes nestes exemplos e me diga 😊

Abraços

Trechos do capítulo que eu escrevi com o Lucas neste livro sobre Testes Unitários com exemplos práticos de componentes 😊

Capítulo 13: Testes Unitários e TDD

Lucas Tagliani Aguiar, Analia Irigoyen

"Os testes são uma parte fundamental do desenvolvimento do software já que são eles que garantem e atestam a qualidade (funcionalidade, serviços, usabilidade, desempenho, segurança, carga etc.) do software desenvolvido antes mesmo de ser ele implantado (entrar em produção).

O foco deste capítulo são os testes unitários e a discussão sobre qual a melhor estratégia a seguir neste escopo e como iniciar esta jornada. Neste sentido, os testes unitários focam em cada componente do produto individualmente, garantindo que ele funciona como unidade, não sendo foco deste teste, rodar um servidor de banco de dados ou subir uma instância de alguma outra aplicação para que seus testes unitários sejam executados, já que eles são independentes. Isso resulta em execuções rápidas, geralmente menos de 1 segundo por teste unitário, tornando-os mais baratos, quando automatizados.

Por que automatizar?

A automação dos testes é a codificação, feita por um desenvolvedor, de um *script* de testes que tem o objetivo de garantir a rapidez e a precisão necessária para preservar a qualidade do software que está sendo produzido. Além disso, quanto maior o software ou a complexidade das regras, maior o esforço em executar estes testes manualmente antes de cada necessidade de implantação em produção. De forma contrária, quando temos os testes automatizados o esforço maior é no início do desenvolvimento, momento em que os testes são criados e mantidos (BECK, 2002).

A própria Pirâmide de Testes ideal do *DevOps* (Muniz et al, 2019) também destaca que o esforço maior de automação deve ser concentrado nos testes unitários, já que por não possuírem dependências nem complexidade, levam menos esforço para serem programados, se tornando bem mais baratos que os outros tipos de testes (sistema, serviço e manuais).

Como automatizar?

A automação de um teste consiste na utilização de um software para automatizar as tarefas presentes em um *script* de testes, como por exemplo (BECK, 2002): :

- Execução de casos testes de um determinado componente:
  - Entrada (dados possíveis de entrada) + Saída (resultados esperados a partir de um conjunto de dados de entrada).

- Os dados do teste devem ser definidos para todas as situações possíveis:
  - Válidas, inválidas e inoportunas, ou seja, não pensar somente no "caminho feliz"; as exceções também são importantes.

É possível criá-los em praticamente qualquer plataforma de desenvolvimento, seja ela *back-end* (.NET, Java, Python, Node.js…) ou *front-end* (javascript, Angular, React, Vue.js…). Falando especificamente em ambientes *back-end,* é bem comum ver estes testes sendo criados para cobrir cenários em classes das camadas de serviço, de lógica, de negócios, de modelagem (se seu modelo não é anêmico, ou seja, se ele tem regras de negócio próprias!) e até naquelas famosas classes "utils", "helper" e afins. Já no *front-end*, o mais comum é testarmos pequenos componentes individualmente como: cabeçalho, rodapé, menus, listagens ou, em funções de javascript puro, aqueles métodos que possuem parâmetros de entrada e um resultado na saída que possua regra de negócio.

Na Figura 13.1 abaixo, um exemplo de teste unitário que valida a soma de uma calculadora, passando como parâmetros os números 2 e 3, esperando 5 como resultado na asserção.

```
[TestMethod]
public void Sum_TwoAndThree_Five()
{
    int result = Calculator.Sum(2, 3);
    Assert.AreEqual(5, result);
}
```

Figura 13.1 - Exemplo de Teste unitário para validação da soma de uma calculadora
Fonte: Elaborado pelo autor

Neste exemplo implementado na linguagem C#, a palavra *Assert* é da classe que faz a asserção, enquanto o método *AreEqual* verifica se os resultados são iguais. Se a variável *result* não for igual ao valor 5, o teste falhará.

É bastante comum, principalmente em sistemas corporativos, que no meio do código haja integrações dos mais variados tipos: consulta/escrita à banco de dados, chamadas de APIs ou web servers. Estas integrações (ou dependências) são chamadas de código de infraestrutura, são códigos que não fazem parte do domínio do sistema e, por isso, não deveriam ser testados de forma real pelo sistema que os utiliza. Quando existem estes tipos de integrações a pessoa desenvolvedora utiliza os *mocks*, *stubs* e spies para auxiliar na escrita dos testes unitários. Codificar *mocks* ou simplesmente "mockar" significa dizer para o código "quando este método for chamado, não faça o que ele geralmente faz, ao invés disso, retorne isto que estou mandando agora".

Vamos supor que um sistema possua uma funcionalidade com a qual é possível buscar um produto pelo código e, na classe *ProductService*, há um método chamado *GetProductByCode* que recebe o código do produto como parâmetros de entrada. O retorno é o produto completo (com o código e o nome). Contudo, neste caso, para consultar um produto, é necessário buscá-lo do banco de dados do sistema, chamando uma classe *Repository*. Na Figura 13.2 abaixo é possível ver esta situação-exemplo.

```
public class ProductService
{
    private readonly IProductRepository repository;
    public ProductService(IProductRepository repository)
    {
        this.repository = repository;
    }

    public Product GetProductByCode(string productCode)
    {
        if (productCode == null)
        {
            throw new ArgumentException("O código do produto não pode ser vazio!");
        }

        List<Product> products = repository.GetAllProducts();

        return products.Find(product => product.Code == productCode);
    }
}
```

Figura 13.2 - Exemplo da classe Product Service
Fonte: Elaborado pelo autor

Neste caso, o objetivo é testar as regras de negócio da camada de serviço, especificamente do método GetProductByName, mas sem realmente consultar os produtos no banco de dados cada vez que o teste unitário for executado. Então, para que isso seja possível - e para que a execução do teste seja rápida, sem precisar acessar um banco de dados realmente -, pode-se “mockar” o retorno do método *GetAllProcuts* no repositório. A Figura 13.3 mostrar o código de um teste unitário com a utilização de mocks.

```
public class ProductService
{
    private readonly IProductRepository repository;
    public ProductService(IProductRepository repository)
    {
        this.repository = repository;
    }

    public Product GetProductByCode(string productCode)
    {
        if (productCode == null)
        {
            throw new ArgumentException("O código do produto não pode ser vazio!");
        }

        List<Product> products = repository.GetAllProducts();

        return products.Find(product => product.Code == productCode);
    }
}
```

Figura 13.2 - Exemplo da classe Product Service
Fonte: Elaborado pelo autor

```
[TestMethod]
public void GetProductByCode_02_Tomate()
{
    // Prepara os dados que serão mockados
    List<Product> mockedProducts = new List<Product>
    {
        new Product("01", "Alface"),
        new Product("02", "Tomate")
    };

    // Configura o mock
    var mockRepository = new Mock<IProductRepository>();
    mockRepository.Setup(p => p.GetAllProducts()).Returns(mockedProducts);
    ProductService productService = new ProductService(mockRepository.Object);

    // Chama o método que deve ser testado
    Product producut = productService.GetProductByCode("02");

    // Verifica se o produto retornado é o correto
    Assert.AreEqual("Tomate", producut.Name);
}
```

Figura 13.3 - Código de Teste Unitário *mockado*
Fonte: Elaborado pelo autor

A Figura 13.3 mostra um código, também na linguagem C#, nela o teste unitário é separado em quatro etapas: definir os dados que serão retornados pelo repositório "mockado" quando o método *GetAllProducts* for invocado; configurar o repositório "mockado" adicionando os dados que foram preparados anteriormente; fazer a chamada do método *GetProductByCode* passando um código de produto válido; e, por último, fazendo a asserção do nome do produto que deve ser retornado para aquele código.

É importante ressaltar que para que o repositório seja facilmente "mockado", deve-se separar o código do domínio de negócio (*ProductService*) e o código de infraestrutura (*ProductRepository*), fazendo assim, uso do padrão de design Injeção de Dependência.

TDD

Uma das maneiras mais eficazes para garantir que haja testes automatizados confiáveis é escrever estes testes como parte da metodologia de desenvolvimento. Kent Beck (BECK, 2002), um dos criadores da agilidade, definiu TDD da seguinte forma: "TDD = Test-First + Design Incremental". Ele nasceu do XP (*Extreme Programming*) em 1999, que defendia que os testes deveriam ser desenvolvidos antes do código.

Test-Driven-Development não é considerada uma ferramenta ou um tipo de teste automatizado, TDD é uma metodologia de desenvolvimento, ou seja, o programador codifica orientado a testes (BECK, 2002).

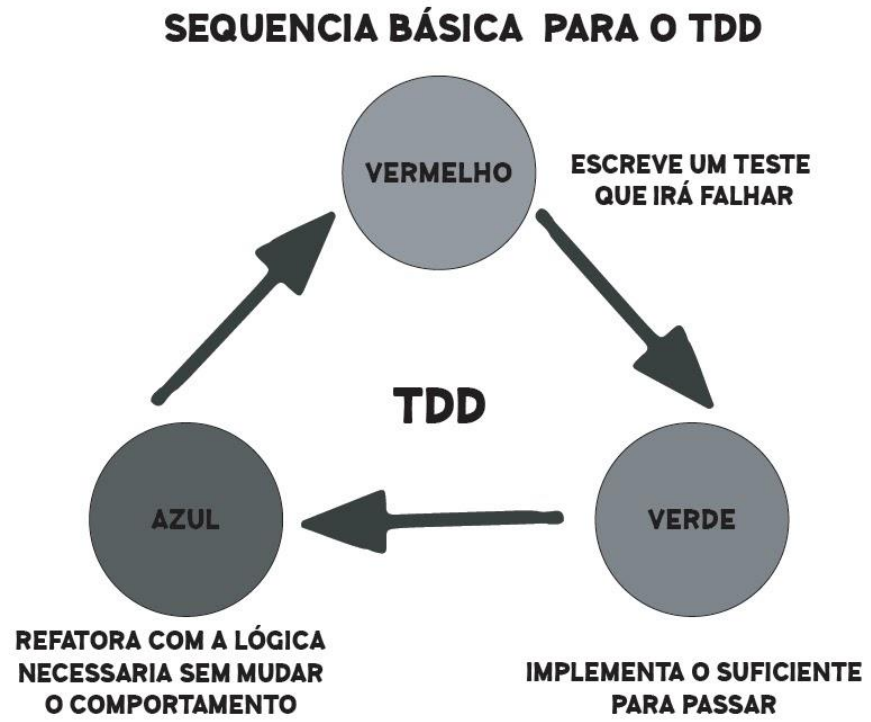

Figura 13.4 - Sequência básica do TDD
Fonte: Adaptado de BECK, Kent. TDD Desenvolvimento Guiado por Testes

Segundo o Livro Jornada *DevOps* (Muniz et al, 2019) a sequência básica para o TDD funciona em três passos como ilustra a Figura 13.4 acima:

- O primeiro passo em vermelho representando um resultado negativo: é escrever um teste que vai falhar, ou seja, quando a pessoa desenvolvedora começa a programar o código ela não programa o código imediatamente, primeiro ela vai escrever um teste considerando um cenário de teste e executar a compilação do código, que vai falhar. Esta falha acontece porque não tem código ainda só tem um teste em cima de nada.

- No segundo passo em verde representando um resultado positivo: a pessoa desenvolvedora escreve somente o código suficiente para passar no teste e, ao executar a compilação deste código o teste vai passar.

- No terceiro passo em amarelo representando a melhoria contínua do código: a pessoa desenvolvedora melhora o código sem alterar o seu comportamento. Este refatoramento deve considerar melhorar a qualidade da escrita do código, tais como: legibilidade, padrões e cobertura de testes unitários.

A abordagem TDD (*Test Driven Development* ou Desenvolvimento Guiado por testes) utilizada para pessoas que desenvolvem a automação dos testes unitários faz com que elas entendam que não só o código, mas que a qualidade do código é da sua responsabilidade. Quando existe uma outra área ou outra pessoa desenvolvedora realizando os testes manuais é normal que as primeiras “relaxem” em relação à qualidade do seu código, já que outra pessoa será responsabilidade pela qualidade.

Como começar?

Se você trabalha em um sistema que não possui nenhum tipo de teste automatizado, provavelmente o mais fácil seja começar com testes unitários com o objetivo principal de adquirir a cultura de testes. Como o TDD é uma orientação para o desenvolvimento, dependendo do time, isso pode ser uma grande mudança de paradigma, por isso, o ideal é estabelecer essas diretrizes junto com o time aos poucos. Comece em funcionalidades menos críticas como forma de aprendizado, use a técnica de programação por pares para que seja possível trocar ideias e promover maior aprendizado.

Quando a automação de teste é executada de forma controlada vários benefícios são relatados pelos times de desenvolvimento, são eles:

- Aumento da produtividade e consequentemente a redução de custos na fase de execução dos testes;
- Aumento sistemático da cobertura de testes;
- Repetibilidade na execução dos casos de teste;
- Precisão dos resultados.

Mostrando os avanços conseguidos com o teste unitário e o TDD, o time tem embasamento para evoluir para outros tipos de testes que falaremos nos próximos capítulos. Neste momento ainda podemos usar o conceito de se minha empresa não quer implementar automação, nada impede que você implemente nas suas atividades.”