C#: Testes de unidade e TDD com xUnit

Referências Aula 1

Definição de Teste

http://softwaretestingfundamentals.com/definition-of-test/

Padrão AAA (Arrange, Act, Assert)

http://wiki.c2.com/?ArrangeActAssert

Padrão Given/When/Then

https://martinfowler.com/bliki/GivenWhenThen.html

xUnit

https://xunit.github.io/

MSTests

https://github.com/Microsoft/testfx-docs

NUnit

https://nunit.org/

Comparativo entre os frameworks de Teste

https://xunit.github.io/docs/comparisons

Porque xUnit?

https://www.martin-brennan.com/why-xunit/

Microsoft está usando o xUnit

https://dev.to/hatsrumandcode/net-core-2-why-xunit-and-not-nunit-or-mstest--aei

Referências Aula 2

Classes de Equivalência - técnica para identificação de testes relevantes:

https://en.wikipedia.org/wiki/Equivalence_partitioning

Análise de Fronteira - outra técnica:

https://en.wikipedia.org/wiki/Boundary-value_analysis

Definição de Product Owner

https://www.scrum.org/resources/what-is-a-product-owner

Diferença entre [Fact] e [Theory]

https://xunit.github.io/docs/getting-started/netfx/visual-studio#write-first-theory

Nomenclatura de testes

https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices#best-practices
https://docs.microsoft.com/pt-br/dotnet/standard/modern-web-apps-azure-architecture/test-asp-net-core-mvc-apps#test-naming

Dados de entrada complexos: [MemberData] e [ClassData]

https://andrewlock.net/creating-parameterised-tests-in-xunit-with-inlinedata-classdata-and-memberdata/

Referências Aula 3

Testes de regressão

http://softwaretestingfundamentals.com/regression-testing/

Intro a Métodos Ágeis na Alura

https://cursos.alura.com.br/course/introducao-aos-metodos-ageis

Livro TDD By Example, de Kent Beck

https://www.amazon.com.br/Test-Driven-Development-Kent-Beck/dp/0321146530/

Livro sobre TDD na Casa do Código

https://www.casadocodigo.com.br/products/livro-tdd-dotnet

Testes de métodos privados

https://docs.microsoft.com/pt-br/dotnet/core/testing/unit-testing-best-practices#validate-private-methods-by-unit-testing-public-methods

Referências Aula 4

Diferenças entre os frameworks ao testar exceções

https://xunit.github.io/docs/comparisons

Visual Studio tem uma ferramenta de cobertura de código, mas infelizmente apenas nas suas versões pagas.

https://docs.microsoft.com/en-us/visualstudio/test/using-code-coverage-to-determine-how-much-code-is-being-tested

Esse artigo do Martin Fowler debate o real propósito da cobertura de código, que em sua opinião (na minha também!) deveria ser para encontrar partes não testadas do seu sistema ao invés de ser uma métrica utilizada em contratos e objetivos do time.

https://www.martinfowler.com/bliki/TestCoverage.html

Referências Aula 5

Consulte as referências usadas nessa aula.

Design OO, Interfaces e Implementação

https://en.wikipedia.org/wiki/Object-oriented_design

Princípios de Design: SOLID

https://en.wikipedia.org/wiki/SOLID

Talk de Michael Feathers sobre como testes melhoram o design
https://www.youtube.com/watch?v=4cVZvoFGJTU&feature=emb_title

Conclusão

Na primeira aula, introduzimos **testes automatizados**, começamos fazendo um teste usando uma aplicação do tipo console, e vimos que ainda não é um teste automatizado.

Descobrimos isso a partir do padrão ***Arrange***, ***Act*** e ***Assert***. Esta última é a parte de verificação da expectativa que ainda estava por conta do desenvolvedor.

Conhecemos o *framework* **xUnit** onde criamos um novo projeto, e neste usamos suas classes para criar os testes. Identificamos cada método de teste com a anotação `[Fact]` e a parte de verificação da expectativa com `Assert`.

Para ver a execução dos testes, usamos uma nova janela do Visual Studio chamada "**Gerenciador de Testes**", onde encontramos a visualização destes, recebendo

alertas do sistema de sucesso ou falha. Vimos que essa técnica nos retorna um rápido *feedback* sobre o funcionamento do código.

Já na segunda aula, começamos a pensar na criação de uma série de testes para confirmar a segurança do código. Porém, precisávamos saber uma quantidade ideal de testes para termos certeza dessa segurança.

Conhecemos as **classes de equivalência** que nos ajudaram a entender a prioridade de um teste por classe, ou seja, quando temos a mesma expectativa para condições de entrada diferentes. Vimos os padrões ***Given***, ***When*** e ***Then***, parecidos com as três partes do padrão anterior para resumir o cenário a um teste apenas.

O objetivo do testador é justamente achar essas classes de equivalência para o sistema. Para ajudar a reproduzir cenários com várias condições de entrada para a mesma expectativa de saída, temos as anotações **[Theory]** e **[InlineData]**.

Continuando na questão de organização dos testes, falamos sobre **nomenclatura** e como nomeá-los de forma a tornar claro sua função. O primeiro critério é a consistência, usando as mesmas regras para todo o trabalho. Outra questão é a comunicação clara do uso do padrão **AAA**, contendo o cenário, o que está sendo testado e quais são os objetivos.

Depois, passamos para a terceira aula onde implementamos uma nova funcionalidade e nos deparamos com o fato desta quebrar o sistema, ou seja, **regrediu**. Para evitar isso, executamos uma suíte de testes que servem como **testes de regressão**.

Porém, uma dúvida surgiu acerca da disciplina sobre a garantia da criação de verificações a cada nova funcionalidade implementada. Nisso, conhecemos a prática do ***Test Driven Development*** ou **TDD**; um fluxo de desenvolvimento começando com o teste para a funcionalidade, para somente depois implementá-la efetivamente, sendo contraintuitivo em um primeiro momento.

O **Ciclo TDD** é composto por teste***, *correção** e **refatoração**, ou seja, propõe escrever o teste primeiro, para depois corrigir sua falha devida à ausência do código de produção alertado pelo Gerenciador, fazendo com que sua execução seja

bem sucedida. Por fim, ficamos livres para refatorar o código, deixando-o mais limpo e eficiente.

Na quarta aula, vimos que nosso sistema e os sistemas em geral comunicam **exceções**. Testamos com o método `Throws<>` pertencente à classe `Assert`, onde verificamos se determinado método testado lança a exceção a partir de uma condição de entrada.

Qualquer comportamento deve ser traduzido em verificações, inclusive as exceções. Depois, quando olhamos para a suíte de testes, vimos que ajuda a documentar o comportamento do sistema.

Por fim, a última aula abordou uma situação inédita; determinada funcionalidade exigia um ***redesign* das classes**, repensando a interação entre elementos e a interface das classes.

Vimos que poderíamos usar o próprio teste para repensar o design. No cenário, injetamos um código que nem mesmo compilava em um primeiro momento, mas auxiliava nessa reelaboração. Neste sentido, o teste também ajuda a melhorar o código.

Alguns autores inclusive dizem que o TDD é sigla para ***Test Driven Design***, pois o design é orientado pelos testes escritos.