Capitulo 1

1. Promovendo rápido feedback

Este é o curso C#: testes de unidade e TDD com xUnit.

Primeiramente, fazemos a seguinte pergunta: como saber se o código está funcionando?

Podemos depender de uma pessoa ou de uma equipe de teste para receber um feedback, porém é possível que demore demais. Então, temos que pensar em uma maneira de diminuir esse tempo e nos tornarmos mais independentes.

Neste curso, conheceremos e aprenderemos algumas técnicas que aumentam nossa segurança e confiança no código, como o uso de TDD e xUnit que veremos ao longo das aulas.

No Visual Studio, teremos uma solução contendo uma biblioteca de classes com o sistema de leilões que iremos trabalhar, evoluiremos esse sistema introduzindo testes automatizados para nos proporcionar essa eficácia.

Começaremos com a aplicação console, observando as questões trazidas para depois trabalharmos com xUnit.

Ao longo do processo, conheceremos mais a fundo a janela Gerenciador de Testes ou Test Explorer do Visual Studio.

Essas automatizações nos retornarão um feedback mais rápido, sua organização nos permitirá comunicar o comportamento do sistema com eficiência, poderemos evitar quebra do código a cada nova funcionalidade implementada e por fim, tornaremos nosso programa melhor com este apoio.

Veremos que o teste é um método de uma classe, o qual terá algumas anotações no xUnit. Escreveremos este código dentro de um padrão específico **Arrange**, **Act** e **Assert**.

Esperamos que este curso seja de grande utilidade e que tenha despertado a curiosidade sobre os testes automatizados.

A seguir, escreveremos o sistema de leilão. Então, vamos lá!

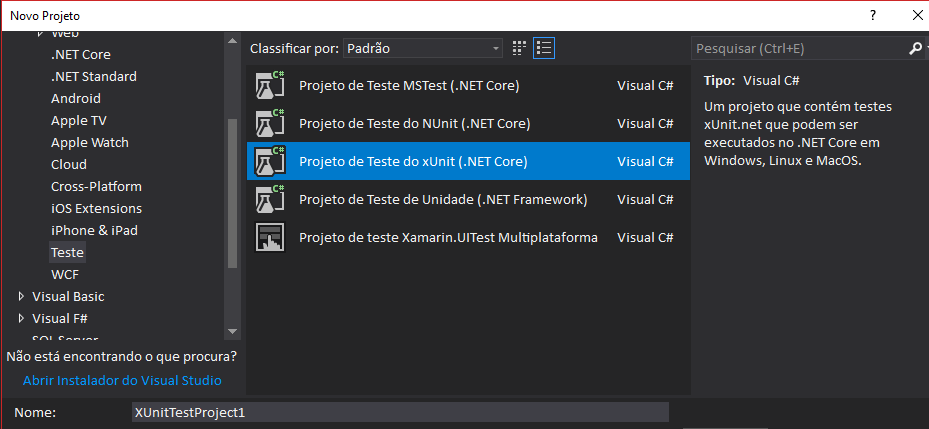
1. O sistema de leilões

Você só vai precisar do Visual Studio 2019 versão Community.

Se não tiver baixe instale.

Além disso precisa confirmar se o Visual Studio tem disponível um template de projeto chamado Projeto de Teste do xUnit (.NET Core). Você pode verificar isso quando seleciona o comando Novo Projeto. Nessa janela deverá constar um item chamado Teste com os projetos de teste específicos para cada framework, conforme a imagem abaixo.

template-projeto.png

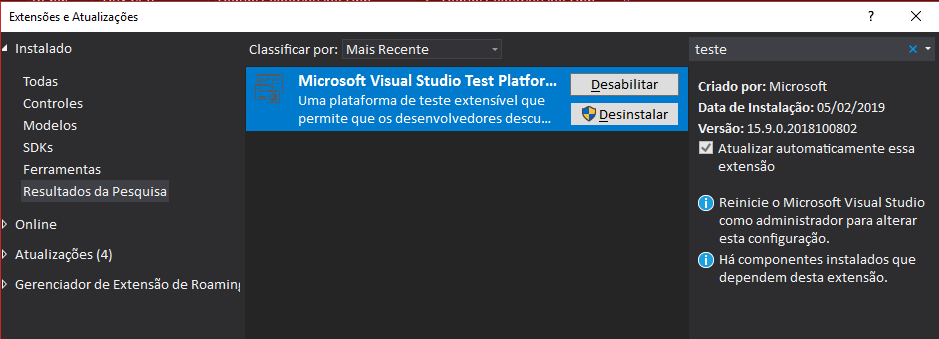


Se não tiver esse template de projeto instalado abra o menu Ferramentas\Extensões e Atualizações e procure pela extensão **Microsoft Visual Studio Test Platform**.

Em seguida basta instalá-la ou habilitá-la.

A imagem abaixo mostra a extensão já habilitada no meu Visual Studio.

extensao-test-platform.png



Depois disso podemos começar a implementar a solução!

* 1. Entendendo o negócio

O negócio que queremos implementar é um sistema de leilões.

* + - Alguns conceitos iniciais aos quais vamos nos basear.
    - Um Leilão é uma promessa de venda de uma peça específica para o cliente que der o melhor lance.
    - Um cliente é uma pessoa física interessada em fazer ofertas no leilão.
    - Os lances são ofertados dentro de em um período pré-determinado chamado pregão.
    - Os lances devem ser aceitos segundo regras específicas determinadas no leilão.
    - Quando o pregão termina é determinado um ganhador (do leilão) baseado nos critérios estabelecidos para aquele leilão. O critério mais comum é o maior valor ofertado.
  1. Criando a solução.

Crie uma solução vazia chamada Alura.LeilaoOnline.

Em seguida crie duas pastas de solução chamadas respectivamente.

Src e tests.

Dentro da pasta src crie um projeto do tipo biblioteca de classes.

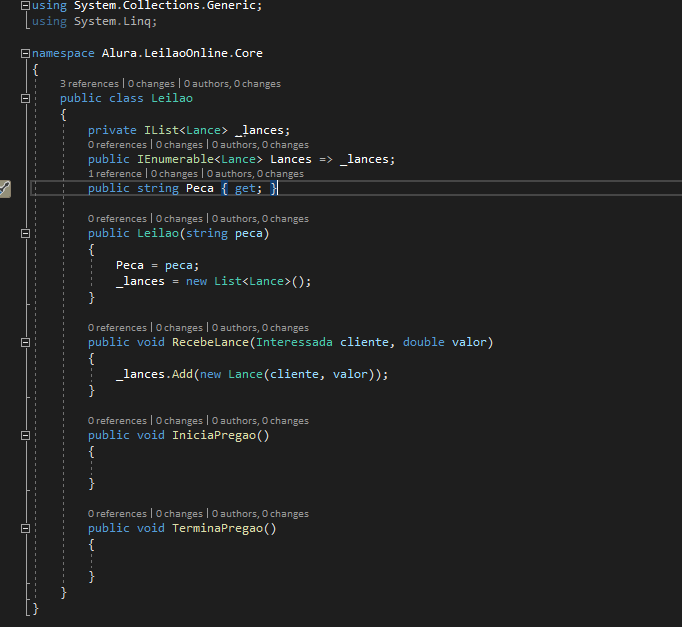
Net Standard 2.0 chamado Alura.LeilaoOnline.Core.

Em seguida crie dentro desse projeto as classes Leilão,

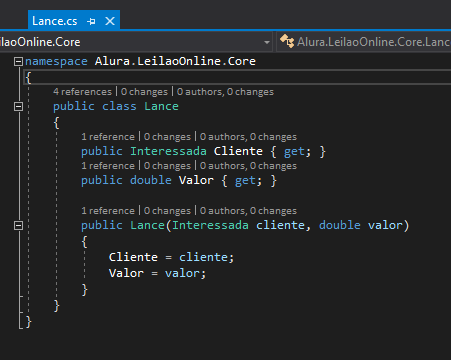
Lance e Interessada.

Veja o código dessas classes abaixo.

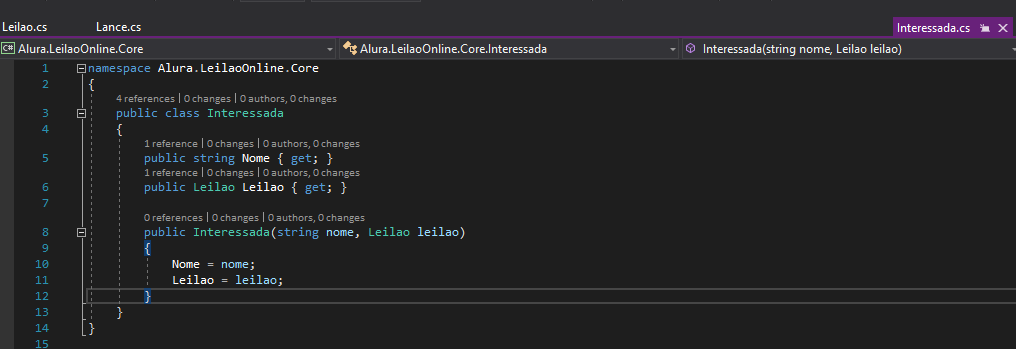
Exemplo Código Leilão.



Exemplo Código Lance



Exemplo Código Lance



* 1. Avaliando um leilão.

No exercício anterior, criamos a solução e o projeto contendo as classes de negócios do sistema de leilão.

Com o Visual Studio, abrimos a classe Leilao.cs e começaremos a primeira implementação que avalia o melhor lance, ou seja, o maior valor dentre as ofertas.

No código, temos a criação das propriedades Lances e Peca, sendo esta última a peça leiloada. Quando o pregão terminar, há um ganhador representado pela nova propriedade Ganhador somente de leitura e escrita apenas internamente cujo tipo é Lance.

Public Lance Ganhador {get; private set; }

Na declaração de TerminaPregao(), pegamos o maior lance e colocamos em Ganhador, o qual começa como NULL e termina com um valor do objeto de tipo Lances.

Isso é feito a partir do método Last(), que resgata o último lance vencedor.

public void TerminaPregao()

{

Ganhador = Lances.Last();

}

Em seguida, no topo do código importamos o namespace do Linq para usarmos este método de extensão.

Assim, finalizamos a funcionalidade de avaliar a melhor oferta de um leilão.

Para ter certeza de sua correta execução, acesse a janela "Gerenciador de Soluções" e clique com o botão direito sobre o diretório "tests", selecionando "Adicionar > Novo Projeto..." para criar um projeto do tipo "Aplicativo do Console" ou "Console Application" que guarda todos os testes. Este será nomeado como Alura.LeilaoOnline.ConsoleApp.

Automaticamente é criada um ponto de entrada Main (), e neste crie os objetos que representam as classes. Começando por leilão, tem como parâmetro de Leilão() o nome do pintor "Van Gogh".

Porém, este último método não está sendo reconhecido no projeto porque precisamos fazer a referência do projeto que o contém. Então, clique com o botão direito do mouse sobre Dependências na janela "Gerenciador de Soluções", escolha "Adicionar Referência..." e selecione o item Alura.LeilaoOnline.Core para finalizar em "OK".

Depois, importe escrevendo using Alura.LeilaoOnline.Core no topo do código.

Insira outra variável em Main() para representar uma pessoa interessada no leilão chamada fulano, a qual é uma nova instância de Interessada() e chama a variável leilão. Depois, adicionamos mais uma pessoa interessada sob o nome maria com a mesma metodologia.

Ainda no mesmo bloco, dizemos que leilão recebe um lance com RecebeLance() de fulano valendo R$800,00 como parâmetro. Na sequência, escrevemos a mesma sentença para a oferta de R$900,00 de maria. Agora, imaginando que Fulano quer cobrir o último valor, inserimos mais uma linha com lance de R$1.000,00.

Depois, finalizamos o pregão com TerminaPregao(), verificando o ganhador e o valor imprimindo-os no console.

static void Main()

{

var leilao = new Leilao("Van Gogh");

var fulano = new Interessada("Fulano", leilao);

var maria = new Interessada ("Maria", leilao);

leilao.RecebeLance(fulano, 800);

leilao.RecebeLance(maria, 900);

leilao.RecebeLance(fulano, 1000);

leilao.TerminaPregao();

Console.WriteLine(leilao.Ganhador.Valor);

}

Executamos a aplicação com "Ctrl + F5" alterando o projeto de inicialização da solução como sendo o Console Application Alura.LeilaoOnline.ConsoleApp na barra superior.

Opção Alura.LeilaoOnline.ConsoleApp em destaque no campo da barra superior no console, vemos a janela apresentando apenas o valor ganhador "1000" de Fulano e assim, temos um código que verifica a funcionalidade. Podemos pensar em outras situações, como um outro lance de R$990,00 feito por Maria. Nesse cenário, a oferta vencedora continuaria sendo mil reais.

Ao adicionar esta nova situação, o terminal imprime este último valor, significando um funcionamento incorreto do programa. Logo, o código que contém Lances.Last() ainda não está funcionando.

Este exemplo de processo de teste com os diversos lances através de um código nos permite automatizar a verificação. Mas este ainda não está completo; no próximo passo veremos o que resta aplicar.

* 1. Objetivos da aula

Nessa aula vamos refletir sobre as perguntas:

Como garantir que meu sistema está pronto?

Como garantir que meu sistema não tem problemas ou defeitos?

Essa discussão vai nos fazer conhecer a necessidade de um conjunto de testes abrangente e automatizado.

Para atender essa necessidade, vamos aprender como escrever testes usando o C# com xUnit dentro do Visual Studio.

* 1. Estrutura de um teste automatizado

No passo anterior, vimos que o código da verificação não está completo, e resolveremos essa questão aqui demonstrando com a definição de algumas referências.

A primeira é encontrada aqui, e é uma proposta de padrão para arranjar e formatar o código em métodos que são testes unitários.

Devemos organizar os testes automatizados em três sessões funcionais chamadas AAA: Arrange, Act e Assert. A primeira propõe arranjar todas as pré-condições de entrada ou o cenário. Na segunda, deixamos claro o método testado e por último, verificamos os resultados esperados.

Agora, veremos se nosso código possui as três partes citadas; o cenário é constituído pelo objeto do leilão e seus interessados, bem como seus respectivos lances. Em seguida, o método a ser testado é o TerminaPregao(). Por fim, falta-nos verificar a última sessão.

Este Assert não está de forma automatizada por enquanto, pois ainda precisamos verificar por conta própria o resultado no console. Para fazer isso, adicione a variável valorEsperado com valor 1000 logo antes de Console.WriteLine().

Logo após esta linha, insira outra variável valorObtido que é igual a leilao.Ganhador.Valor, parâmetro retirado do comando de impressão.

Para finalizar a verificação, adicione uma comparação com if() para imprimir a mensagem que indica se o valor esperado é igual ao obtido.

Depois, usamos else para apontar o caso contrário.

static void Main()

{

//código omitido

//Act - método sob teste

leilao.TerminaPregao();

//Assert

var valorEsperado= 1000;

var valorObtido = leilao.Ganhador.Valor;

if (valorEsperado == valorObtido)

{

Console.WriteLine("TESTE OK");

}

else

{

Console.WriteLine("TESTE FALHOU");

}

}

Assim, nossa sessão de Assert está automatizada. Logo, não precisamos mais nos preocupar em validar a expectativa observando o terminal diretamente, apenas vendo seu retorno.

Recebemos a mensagem de falha ao rodar a aplicação, afinal o último lance dado por Maria de R$990,00 não é o esperado, o que indica que o processo está completo e podemos arrumar o código.

Portanto, usaremos estas três sessões para formatar nosso programa e garantir uma verificação rápida e segura.

Outra referência encontrada aqui apresenta um artigo de Martin Fowler que propõe outra maneira baseada em Given, When e Then. Também é um padrão associado com o anterior.

De volta ao Visual Studio, corrigimos o problema ordenando os lances pelo valor com OrderBy() na declaração de TerminaPregao().

public void TerminaPregao()

{

Ganhador = Lances

.OrderBy(l = l.Valor)

.Last();

}

Ao executarmos novamente, o teste é aprovado após a correção no código.

Outro ponto levantado nas referências é o Unit Test Methods, considerado a parte mais interna para teste do software onde verificamos uma unidade de trabalho. Em geral, é representado por um método em uma classe, mas não necessariamente todos são testados unitariamente.

Com o primeiro teste finalizado, arrumamos o código criando um método estático void em Program chamado LeilaoComVariosLances(), onde extraímos o bloco das três sessões anteriores de Main() e o inserimos neste novo método declarado.

Assim, chamamos este novo método dentro de Main(). Antes deste escopo, declaramos outro método chamado LeilaoComApenasUmLance() e o inserimos em Main() com uma cópia do mesmo conteúdo extraído anteriormente, excluindo os lances de Maria para deixar apenas os de Fulano cujo valor esperado passa a ser de R$800,00. Desta forma, temos dois cenários de teste.

static void Main()

{

LeilaoComVariosLances();

LeilaoComApenasUmLance();

}

Executamos a aplicação para ver a informação dos dois testes. Ainda, nosso código está bastante grande, pois copiamos o bloco de verificação com as três sessões em cada um dos novos métodos. Podemos extrair o corpo de if() e else para um novo método Verifica() antes de LeilaoComVariosLances(). Interpolamos a última string para verificar quais eram os valores esperados e obtidos. Assim, podemos chamar Verifica() recebendo o valorEsperado e valorObtido dentro dos métodos LeilaoComVariosLances() e LeilaoComApenasUmLance().

Ainda, podemos melhorar o programa ainda mais adicionando cores ao console para cada resultado com ForegroundColor e ConsoleColor, capturando a cor original e retornando-a ao final.

private static void Verifica(double esperado, double obtido)

{

var cor = Console.ForegroundColor;

if (esperado == obtido)

{

Console.ForegroundColor = ConsoleColor.Green;

Console.WriteLine("TESTE OK");

}

else

{

Console.ForegroundColor = ConsoleColor.Red;

Console.WriteLine(

$"TESTE FALHOU! Esperado: {esperado}, obtido: {obtido}.");

}

Console.ForegroundColor = cor;

}

Teste o resultado alterando temporariamente o valor do maior lance e rodando no console. Vimos que alguns recursos eram preexistentes, e no próximo passo veremos algumas bibliotecas que já realizam estes trabalhos.

Na etapa anterior, fizemos um código de verificação automatizada no padrão AAA e começamos a nos afastar um pouco do sistema de leilão.

Vimos que já existem bibliotecas que realizam este trabalho, permitindo que continuemos focados na implementação do negócio. Veremos sobre os chamados frameworks de teste neste passo.

Atualmente, os maiores concorrentes desta área na plataforma .NET são o xUnit, MSTest e NUnit, e neste curso usaremos o primeiro. Para se aprofundar mais no porquê de usar este recurso, leia o artigo de Chris Mathurin neste link.

Ainda, a própria Microsoft está movendo seus testes de códigos para a biblioteca xUnit e os conceitos aprendidos nesta framework também são aplicáveis nos outros dois. Também é possível fazer uma comparação entre estes acessando este endereço.

Para ver como usar os dois testes que fizemos anteriormente no xUnit, crie um novo projeto dentro da pasta "tests" para agrupar todos os testes. No Visual Studio, clique com o botão direito sobre este diretório na caixa "Gerenciador de Soluções" e escolha "Adicionar > Novo Projeto..." para selecionar "Teste > Projeto de Teste do xUnit (.NET Core)".

Nomeie-o como Alura.LeilaoOnline.Tests e conclua com OK.

Este novo projeto depende de alguns pacotes do NuGet; a própria biblioteca xUnit, um pacote de integração desta com o Visual Studio e as interfaces que o .NET precisa para reconhecer os testes dos frameworks.

Agora, começamos a colocar os métodos LeilaoComApenasUmLance() e LeilaoComVariosLances() no xUnit.

Na janela lateral, clique com o botão direito sobre UnitTest1.cs e o exclua.

Em seguida, clique novamente sobre Alura.LeilaoOnline.Tests e selecione "Adicionar > Classe...".

Na caixa de dálogo, nomeie como LeilaoTestes e aperte "Adicionar".

Na nova aba, remova as três primeiras linhas, copie os códigos dos métodos citados da aba Program.cs e cole no corpo de LeilaoTestes.

Percebemos alguns erros como em Leilao() e Interessada(); para resolver, clique com o botão direito sobre Dependências na janela lateral, escolha "Adicionar Referência..." e selecione Alura.Leilao.Online.Core para relacionar à biblioteca que possui as classes do núcleo.

Além disso, temos que importar este namespace escrevendo using Alura.Leilao.Online.Core no topo do código de LeilaoTestes. Falta-nos ainda realizar algumas outras alterações. Primeiro, os métodos de teste não são mais estáticos, pois o xUnit não trabalha com este tipo e devemos excluir esta parte. Depois, altere-os para públicos.

O nome do código Verifica() deve ser substituído por Assert.Equal(), relacionando diretamente com a sessão do mesmo nome. Para o programa reconhecer esta classe, importamos using Xunit;.

Então, o método Verifica() que pega dois valores e demonstra o erro ou acerto do teste, passa a ser representado por Assert.Equal(). O mesmo procedimento deve ser adotado na ocorrência do segundo teste para que ambos sejam passados para o xUnit.

De agora em diante, os testes não são mais apresentados no terminal e sim em uma janela integrada ao Visual Studio, e por isso os pacotes do NuGet são necessários. Para executar, vá na barra superior de opções e acesse "Teste > Executar > Todos os testes" pela primeira vez, sendo possível acionar outras execuções.

Ao fazer isso, uma janela "Saída" é exibida com a compilação da solução como um todo e as tentativas de execução dos testes. Observe que o novo console aponta a ausência de testes disponíveis, ainda que tenhamos elaborado dois no programa. O que precisamos fazer é informar o xUnit de que os dois métodos são testes através de uma anotação [Fact].

Ao inserir a anotação antes de cada teste, um alerta é apresentando orientando que as classes de testes devem ser públicas.

public class LeilaoTestes

{

[Fact]

public void LeilaoComVariosLances()

//código omitido

[Fact]

public void LeilaoComApenasUmLance()

//código omitido

}

Executamos todos os testes para observar a resolução no console. Repare que, desta vez, o programa detectou dois testes. Porém, apresenta uma falha na verificação do método Assert.Equal() e indica que o valor esperado e o atual são diferentes.

Há outra visualização disponível para os resultados dos testes em uma janela chamada "Gerenciador de Testes" ou "Test Explorer" encontrada em "Teste > Janelas". Nesta, vemos uma lista que apresenta os métodos de teste com falha e com sucesso. Ao clicar no LeilaoComVariosLances() que falhou, o sistema mostra mais informações sobre o ocorrido.

Arrumamos o valor da variável valorEsperado deste teste para 1000 e, desta forma, podemos executar novamente apenas clicando em "Executar Tudo" nesta nova janela de gerenciamento ou clicando com o botão direito sobre o teste que se quer executar separadamente, além de outras opções como "Executar > Executar Testes com Falha".

Então, ao realizar a execução novamente, percebemos que os testes foram aprovados com sucesso após as mudanças. A partir de agora, começaremos a colocar todos os testes automatizados neste projeto do xUnit.

Faça os exercícios a seguir e até a próxima aula!