**Introdução**

**Boas vindas**! Sou o instrutor **Daniel Portugal** e 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.

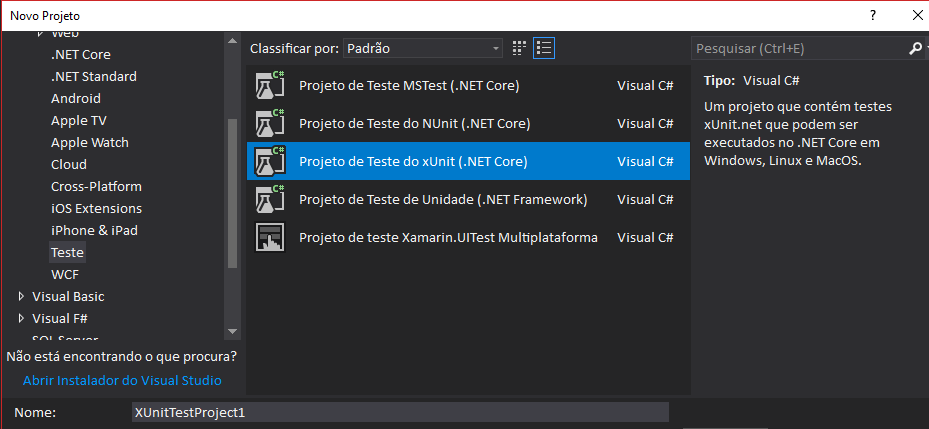
A seguir, escreveremos o sistema de leilão. Então, **vamos lá**!

**O sistema de leilões**

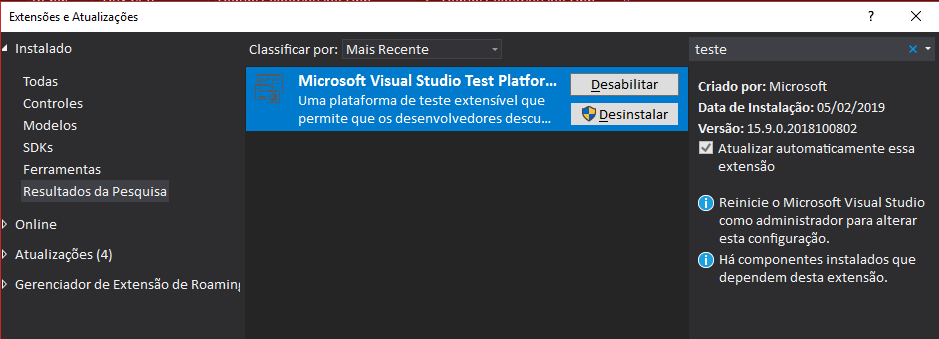
## Ferramentas

Você só vai precisar do Visual Studio 2017 versão Community. Se não tiver instalado baixe-o [aqui](https://visualstudio.microsoft.com/pt-br/vs/community/).

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.



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.



Depois disso podemos começar a implementar a solução!

## 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.

## 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 Leilao, Lance e Interessada. Veja o código dessas classes abaixo.

using System.Collections.Generic;

using System.Linq;

namespace Alura.LeilaoOnline.Core

{

public class Leilao

{

private IList<Lance> \_lances;

public IEnumerable<Lance> Lances => \_lances;

public string Peca { get; }

public Leilao(string peca)

{

Peca = peca;

\_lances = new List<Lance>();

}

public void RecebeLance(Interessada cliente, double valor)

{

\_lances.Add(new Lance(cliente, valor));

}

public void IniciaPregao()

{

}

public void TerminaPregao()

{

}

}

}

namespace Alura.LeilaoOnline.Core

{

public class Lance

{

public Interessada Cliente { get; }

public double Valor { get; }

public Lance(Interessada cliente, double valor)

{

Cliente = cliente;

Valor = valor;

}

}

}

namespace Alura.LeilaoOnline.Core

{

public class Interessada

{

public string Nome { get; }

public Leilao Leilao { get; }

public Interessada(string nome, Leilao leilao)

{

Nome = nome;

Leilao = leilao;

}

}

}

Se preferir pode baixar a solução inicial [aqui](https://caelum-online-public.s3.amazonaws.com/1120-testes-csharp/01/Alura.LeilaoOnline.ProjetoInicial.zip).

# 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 leilao, tem como parâmetro de Leilao() 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 leilao. Depois, adicionamos mais uma pessoa interessada sob o nome maria com a mesma metodologia.

Ainda no mesmo bloco, dizemos que leilao 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);

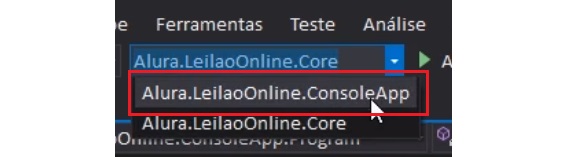
leilao.RecebeLance(fulano, 1000);

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.



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.

# 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.

# 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](http://wiki.c2.com/?ArrangeActAssert), 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](https://martinfowler.com/bliki/GivenWhenThen.html) 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](https://dev.to/hatsrumandcode/net-core-2-why-xunit-and-not-nunit-or-mstest--aei).

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](https://xunit.github.io/docs/comparisons).

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!

# Consolidando seus conhecimentos

Chegou a hora de você seguir todos os passos realizados por mim durante esta aula:

* criar a solução inicial com a biblioteca de classes que representa o sistema de leilões conforme [essa atividade](https://cursos.alura.com.br/course/tdd-csharp-xunit/task/52226)
* criar um projeto do tipo console para testar um cenário inicial onde uma peça de Van Gogh é disputada por dois interessados
* criar o projeto de testes do xUnit para colocar os testes automatizados usando esse framework
* copiar os dois testes do projeto console para o xUnit

Caso já tenha feito, ótimo! Se ainda não, é importante que você execute o que foi visto nos vídeos para poder continuar com a próxima aula.

# Referências Utilizadas

# Referências mencionadas nessa aula

### 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>

# O que aprendemos?

Nessa aula você:

* conheceu o padrão Arrange, Act e Assert
* usou testes automatizados para agilizar o feedback do seu código
* conheceu o framework xUnit, com facilidades para verificar suas expectativas e integração com o Visual Studio
* criou métodos de teste do xUnit através da anotação [Fact]

# Escrevendo mais testes

Continuaremos nosso desenvolvimento com os testes já passados para o xUnit.

Por enquanto, os lances estão de forma desordenada. Criaremos uma nova verificação para o sistema de leilão que recebe ofertas ordenadas por valor sob o método LeilaoComLancesOrdenadosPorValor() no topo de LeilaoTestes, sem esquecer de adicionar a anotação [Fact] da mesma maneira que já vimos.

Na declaração, copiamos e colamos o mesmo código do corpo do teste com vários lances. Nas linhas de leilao.RecebeLance(), ordene-as por valor dado. Repare que o Gerenciador de Testes já reconheceu o novo método.

Criamos uma verificação antes da anterior chamada LeilaoComTresClientes(), recebendo três ofertas diferentes. Copie e cole novamente o código no corpo do novo método, inserindo mais uma variável beltrano para representar o terceiro cliente após fulano e maria.

Em seguida, adicione mais um lance feito por Beltrano no valor de 1400 e depois altere para o mesmo valor em valorEsperado. Ainda, faremos um teste para identificar o nome da pessoa com a maior oferta, neste caso, beltrano através de uma nova linha Assert.Equal().

public class LeilaoTestes

{

[Fact]

`LeilaoComTresClientes()`

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

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

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

var beltrano = new Interessada ("Beltrano", leilao);

leilao.RecebeLance(fulano, 800);

leilao.RecebeLance(maria, 900);

leilao.RecebeLance(fulano, 1000);

leilao.RecebeLance(maria, 990);

leilao.RecebeLance(beltrano, 1400);

leilao.TerminaPregao();

var valorEsperado = 1400;

var valorObtido = leilao.Ganhador.Valor;

Assert.Equal(valorEsperado, valorObtido);

Assert.Equal(beltrano, leilao.Ganhador.Cliente);

}

Podemos ter mais de um Assert.Equal() na verificação relacionada ao ganhador, pois este método possui sobrecarga para vários tipos de dados diferentes como double e a referência Interessada nestes casos.

Executamos os testes para ver que todos estão aprovados no Gerenciador.

Existe ainda algumas questões; quantos testes seriam necessários para assegurar o funcionamento de um programa? Veremos adiante. Mas por enquanto, nosso código tem muitas duplicações, deixando-o extenso e repetitivo.

No próximo passo, veremos como deixá-lo mais limpo e organizado.

# Objetivos da aula

Nessa aula nosso objetivo é discutir as seguintes perguntas:

### Quantos testes são o bastante?

### Será que estou criando os testes certos? Afinal, que cenários devo testar?

### Como organizo meus testes? Que nome devo dar às classes e métodos de testes?

Durante essa reflexão vamos conhecer uma missão importante do testador e como o xUnit pode nos ajudar a cumprí-la. Além disso iremos organizar nossos testes segundo um padrão específico.

Vamos lá?

# O que devo testar

Neste passo, vamos responder à pergunta anterior descobrindo quando terminar os testes, ou seja, confirmar se a funcionalidade é suficientemente confiável e pronta para produção.

Para isso, temos duas dicas: a primeira é procurar orientação de profissionais mais experientes e a segunda é usar o padrão de teste sob três sessões como já vimos. Além do **AAA**, temos o ***Given***, ***When*** e ***Then***.

Começando por LeilaoComTresClientes(), aplicamos o conceito do segundo padrão descrevendo as etapas de verificação em comentários no próprio código com //, para esclarecer mais cada situação do programa e o que esperamos como resultado do teste em cada um, podendo ser transcrito para qualquer forma de registro de fluxo de trabalho que preferir para não poluir nosso código, como o **[Trello](https://trello.com/" \t "_blank)** por exemplo.

Faça a mesma descrição de cada cenário diferente para os quatro testes que implementamos. Analisando-os, há situações quase idênticas acontecendo, como as expectativas de maior valor esperado.

Então podemos reduzir os quatro cenários em apenas um; nesta análise, observando os valores de entrada e saída esperada, trabalhamos com uma técnica de teste chamada [**classes de equivalência**](https://en.wikipedia.org/wiki/Equivalence_partitioning) ou *equivalence partitioning* ou ainda *equivalence class partitioning* (ECP), a qual consiste em dividir os dados em várias partições para tornar os testes mais eficientes.

Da forma como está, os quatro testes lidam com uma mesma partição, ou seja, está bem ineficiente. Não importa o cenário, a expectativa final sempre é a de maior valor com a exibição do cliente ganhador dos lances e por isso o LeilaoComTresClientes() é o mais completo dentre todos. Neste, precisamos que haja pelo menos um lance com qualquer número de clientes.

Esta análise só pode ser feita com a experiência de fazê-la, contando idealmente com uma mentoria, sempre focando nos valores. Vale a pena pesquisar uma outra técnica chamada [**análise dos valores de fronteira**](https://en.wikipedia.org/wiki/Boundary-value_analysis) ou ***boundary-value analysis***.

De volta às anotações, é preciso pensar em um outro cenário possível onde nenhum lance é dado.

# Leilão sem lances

Resolveremos sobre o cenário onde nenhum lance é dado no leilão.

Neste caso, quem seria o ganhador? É uma decisão a ser tomada pelo [responsável](https://www.scrum.org/resources/what-is-a-product-owner) pelo produto final de forma arbitrária. Neste caso, imaginemos que a oferta vencedora tem valor 0 para podermos fazer um teste baseado neste cenário.

Então, no Visual Studio, adicionamos mais um método público de teste dentro de LeilaoTestes com anotação [Fact] chamado LeilaoSemLances(). Dentre deste, copiamos e colamos o corpo de código das outras verificações para podermos realizar as devidas alterações.

Começamos retirando as linhas dos clientes interessados e das ofertas. Depois, altere o valor de valorEsperado para 0.

public void LeilaoSemLances()

{

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

leilao.terminaPregao();

var valorEsperado = 0;

var valorObtido = leilao.Ganhador.Valor;

Assert.Equal(valorEsperado, valorObtido);

}

Rodamos os testes no Gerenciador para observarmos o resultado. Vemos que esta última verificação apresenta falha, e devemos ver suas especificações exibidas.

Há uma exceção nativa do .Net detectada chamada System.InvalidOperationException, apontando que a sequência não contém elementos, ou seja, o enumerado de lances não possui nenhum item dentro.

Pela pilha de execução vemos que isso foi feito justamente quando chamamos o método Last() dentro de TerminaPregao(). Na documentação deste método problemático, o programa indica que há exceções que podem ser lançadas justamente por não haver elementos nesta sequência.

Para resolver, alteramos o uso de Last() para LastOrDefault(), o qual se diferencia do primeiro por verificar se está vazio, e se estiver, retorna um objeto *default* qualquer. Para um enumerado de lances, definimos o retorno com o método DefaultIfEmpty() criando um objeto a ser usado neste caso com cliente null e valor 0.

public void TerminaPregao()

{

Ganhador = Lances

.DefaultIfEmpty(new Lance(null, 0))

.OrderBy(l => l.Valor)

.LastOrDefault();

}

Com essas alterações, executamos novamente no console para analisar os resultados. Dessa vez, tudo funciona como esperado.

Vale lembrar que é importante conhecer, explorar e estudar a **API** do .NET, pois são nesses momentos que este tipo de conhecimento é necessário para resolver questões apresentadas no código de maneira eficiente.

Nossa funcionalidade de avaliação do leilão possui duas classes de equivalências, sendo que ainda temos cinco testes implementados. A seguir, veremos como reduzir e limpar nosso código.

# Teorizando com xUnit

Temos duas classes de equivalência com testes criados: uma para o leilão com pelo menos um lance e outra para uma situação sem nenhum. Para o primeiro, fizemos quatro testes distintos com códigos duplicados, o que é pouco eficiente.

Portanto, sabemos que uma boa prática é reduzir e organizar o programa ao máximo, tornando-o mais fácil de lidar. Para isso, o próprio xUnit possui um recurso chamado de **teoria**.

Começando por LeilaoComVariosLances(), substitua [Fact] pela anotação **[Theory]** que transforma o método a partir de algumas alterações. Assim, podemos ver um erro de compilação neste método apontando que os que estão anotados desta forma devem ter dados de teste, que é justamente o que queremos.

Para isso, usamos outra anotação abaixo da anterior chamada **[InlineData()]**; em seu construtor, passe os dados de entrada com valores dos lances, ignorando os interessados neste momento.

Insira outra anotação como esta para informar apenas o lance de R$800. Precisamos agrupar esta coleção em apenas uma variável através de uma lista ou ***array***. Por fim, adicione mais um [InlineData()] para receber os lances na ordem numérica dos valores.

Então temos três testes com dados de entrada, os quais devem ser injetados como argumentos de entrada do método LeilaoComVariosLances() junto com ofertas. Desta forma, eliminamos o erro de compilação apontado anteriormente.

Como por enquanto ofertas não está sendo utilizado, usamos foreach() para cada valor dentro das ofertas. Em seu corpo, fazemos com que receba o cliente e o valor da array. Assim estamos parametrizando os valores de entrada para o teste em questão verificações por valor ordenado, desordenado e único.

O valorEsperado deve ser recebido como valor de entrada e dado de teste de entrada.

[Theory]

[InlineData] (1000, new double[] { 800, 900, 1000, 1200 })]

[InlineData] (1000, new double[] { 800, 900, 1000, 990 })]

[InlineData] (800, new double[] { 800 })]

public void LeilaoComVariosLances(

double valorEsperado,

double[] ofertas)

{

//código omitido

foreach(var valor in ofertas)

{

leilao.RecebeLance(fulano, valor);

}

leilao.TerminaPregao();

var valorObtido = leilao.Ganhador.Valor;

Assert.Equal(valorEsperado, valorObtido);

}

Observe que o Gerenciador de Teste apenas reconhece o método o qual fizemos as modificações contendo três testes representados pelas três anotações [InlineData] que inserimos. Conseguimos então uma classe de equivalência onde variamos os dados de entrada com a mesma expectativa.

Sendo assim, podemos apagar os métodos LeilaoComApenasUmLance(), LeilaoComLancesOrdenadosPorValor() e LeilaoComTresClientes(), ficando com um código bem mais enxuto com a ajuda dos recursos do próprio xUnit.

Portanto, se queremos passar dados de entrada para orientar os testes através de [InlineData], usamos a anotação [Theory]. Se não importar quais dados são passados e a expectativa permanecer a mesma, usamos [Fact]. Caso queira se aprofundar nessa diferença, leia a documentação do xUnit neste [link](https://xunit.github.io/docs/getting-started/netfx/visual-studio#write-first-theory).

Entenderemos melhor sobre como organizar e evoluir nossos testes em termos de nomenclatura no passo seguinte.

**Nomenclatura**

Uma importante questão relacionada à organização dos nossos testes é a **nomenclatura** destes.

Seguiremos duas recomendações da Microsoft, ainda que existam outras discussões em outros lugares. Em um dos artigos disponíveis neste [link](https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices#best-practices), são apresentadas as melhores práticas para testes unitários e seus nomes que devem conter três partes, sendo justamente as mesmas que abordamos até agora: *Given*, *When* e *Then*.

Ou seja, a primeira parte diz qual o método a ser testado, a segunda revela as condições de entrada ou cenário e a terceira possui a expectativa de resultado para balizar o teste.

Agora temos que ver como organizar em nome especificamente. Usamos a segunda sugestão encontrada [aqui](https://docs.microsoft.com/pt-br/dotnet/standard/modern-web-apps-azure-architecture/test-asp-net-core-mvc-apps#test-naming). O nome da classe de teste é o nome do método que está sendo testado, e os métodos de teste terão as outras duas partes de cenário e comportamento esperado.

Como temos apenas dois, volte ao Visual Studio para aplicar as recomendações. Na janela "Gerenciador de Soluções", altere o nome da classe de LeilaoTestes.cs para LeilaoTerminaPregao.cs. Com isso, o sistema alerta um erro ao renomear, sendo necessário alterar manualmente em sua declaração também.

public class LeilaoTerminaPregao

Em seguida, o comportamento esperado para LeilaoComVariosLances() é a devolução do maior valor ofertado. Portanto, seu nome deve ser alterado para RetornaMaiorValorDadoLeilaoComPeloMenosUmLance() e, ainda que fique bem extenso, sua função é bastante clara. No caso de LeilaoSemLances(), renomeie-o para RetornaZeroDadoLeilaoSemLances().

Repare que a janela "Gerenciador de Testes" é atualizada e já fornecem a descrição dos testes com o comportamento esperado.

Para validar as nomenclaturas, introduza um terceiro cenário para ser testado onde o leilão é finalizado com um número qualquer de lances e depois recebe nova oferta, a qual deve ser ignorada. Este novo contexto não é uma classe de equivalência já existente, pois não estamos mais testando o TerminaPregao() e sim o RecebeOferta().

Agora precisamos pensar na nova nomenclatura. Para isso, crie uma nova classe pública clicando com o botão direito sobre AluraLeilaoOnlineTests para selecionar "Adicionar > Classe..." e nomear como LeilaoRecebeOferta. Na nova aba, mude para a biblioteca xUnit substituindo System por Xunit na primeira importação. Em seguida, importe também as nossas classes de negócio em Alura.LeilaoOnline.Core e delete a última namespace, declarando a biblioteca LINQ.

using Xunit;

using Alura.LeilaoOnline.Core;

using System.Linq;

Crie um teste dentro de LeilaoRecebeOferta, anotando como [Fact] a princípio e declare um método com o nome NaoPermiteNovosLancesDadoLeilaoFinalizado(). Copie o código dos testes anteriores e cole no corpo deste novo para termos um ponto de partida.

Como o método testado não é mais o TerminaPregao(), este passa a ser parte do cenário. Inserimos a variável de fulano interessado na pintura leiloada, o qual faz duas ofertas no valor de R$800,00 e R$900,00.

Agora, adicionamos o método leilao.RecebeLance() que está sob teste, cujos parâmetros são o fulano e sua nova oferta de R$1000,00. O valorEsperado passa a ter valor 2 e o valorObtido recebe a contagem dos lances do leilão para testarmos.

public class LeilaoRecebeOferta

{

[Fact]

public void NaoPermiteNovosLancesDadoLeilaoFinalizado();

{

//Arranje - cenário

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

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

leilao.RecebeLance(fulano, 800);

leilao.RecebeLance(fulano, 900);

leilao.TerminaPregao();

//Act - método sob teste

leilao.RecebeLance(fulano, 1000);

//Assert

var valorEsperado = 2;

var valorObtido = leilao.Lances.Count();

Assert.Equal(valorEsperado, valorObtido);

}

}

Neste ponto, o Gerenciador de Testes já reconheceu nossa implementação e podemos executar no console.

São apontadas várias falhas. Primeiro, alteramos o valor da primeira linha do [InlineData] para R$1200,00.

[Theory]

[InlineData] (1200, new double[] { 800, 900, 1000, 1200 })]

[InlineData] (1000, new double[] { 800, 900, 1000, 990 })]

[InlineData] (800, new double[] { 800 })]

Rodando novamente, resolvemos esta questão. Falta-nos verificar novamente a falha em NaoPermiteNovosLancesDadoLeilaoFinalizado(), o qual permite apenas dois lances, e são apresentados três. Para tentar corrigir, inserimos um estado antes da declaração de Leilao em Leilao.cs.

Crie um enumerado chamado EstadoLeilao, e em seu corpo insira os dois estados de LeilaoEmAndamento e LeilaoFinalizado. Na declaração da classe Leilao, adicione a propriedade pública EstadoLeilao do tipo Estado que será *setada* internamente.

Em seguida, dentro do método Leilao logo na sequencia, escreva Estado = EstadoLeilao.LeilaoEmAndamento;, para que já comece em andamento. Quando o pregão terminar em TerminaPregao(), seu Estado passa a ser igual a EstadoLeilao.LeilaoFinalizado.

Desta forma, na hora de receber lances na declaração do método RecebeLance(), aplicamos um laço if() que verifica se o estado está em andamento para poder receber novas ofertas.

Assim, temos o código para executar tudo novamente no Gerenciador de Testes:

namespace Alura.LeilaoOnline.Core

{

public enum EstadoLeilao

{

LeilaoEmAndamento,

LeilaoFinalizado

}

public class Leilao

{

private IList<Lance> \_lances;

public IEnumerable<Lance> Lances => \_lances;

public string Peca { get; }

public Lance Ganhador { get; private set; }

public EstadoLeilao Estado { get; private set; }

public Leilao (string peca)

{

Peca = peca;

\_lances = new List<Lance>();

Estado = EstadoLeilao.LeilaoEmAndamento;

}

public void RecebeLance (Interessada cliente, double valor)

{

if (Estado == EstadoLeilao.LeilaoEmAndamento)

{

\_lances.Add(new Lance(cliente, valor));

}

//código omitido

public void TerminaPregao()

{

Ganhador = Lances

.DefaultIfEmpty (new Lance(null, 0))

.orderBy(l => l.Valor)

.LastOfDefault();

Estado = EstadoLeilao.LeilaoFinalizado;

}

}

//código omitido

}

Validamos as classes de equivalência e nomenclatura de código, faltando apenas as teorias e fatos. Em NaoPermiteNovosLancesDadoLeilaoFinalizado(), substituímos a anotação [Fact] por [Theory], adicionando a quantidade esperada recebendo double[] com as ofertas.

Nesta teoria, há vários dados de entrada. Por exemplo, para as duas ofertas, recebemos new double[] com um array de 800 e 900 em [InlineData()]. Ainda, use o foreach() na sequencia do cenário onde obtemos o valor que está dentro do array de ofertas e recebemos o lance para Fulano.

Então passaremos duas ofertas, o que nos permite apagar a linha que contém a variável var valorEsperado = 2;. Por fim, substitua o nome da variável valorObtido por qtdeObtida.

Passamos ainda mais alguns dados para validarmos a questão da teoria e do fato através de um teste parametrizado. Por isso, adicione mais uma entrada [InlineData()] que recebe quatro ofertas de 1000, 1200, 1400, 1300. Inseridas as quatro ofertas, ainda haverá uma outra que não será aceita

[Theory]

[InlineData(4, new double[] {100, 1200, 1400, 1300})]

[InlineData(2, new double[] { 800,900 })]

public void NaoPermiteNovosLancesDadoLeilaoFinalizado()

int qtdeEsperada, double[]ofertas)

{

//Arranje - cenário

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

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

foreach (var valor in ofertas)

{

leilao.RecebeLance(fulano, valor);

}

leilao.TerminaPregao();

//Act - método sob teste

leilao.RecebeLance(fulano, 1000);

//Assert

var qtdeObtida = leilao.Lances.Count();

Assert.Equal(qtdeEsperada, valorObtido);

}

Rodamos novamente para avaliar os resultados. Assim, também validamos os fatos e teorias, fazendo com que os testes sejam aprovados. [Fact] é quando testamos algo sem depender de valores de entrada, enquanto [Theory] é quando criamos o mesmo código com o mesmo teste para várias condições.

# Consolidando seus conhecimentos

Chegou a hora de você seguir todos os passos realizados por mim durante esta aula:

* reduza os métodos de teste para 1 teste por classe de equivalência usando as anotações [Theory] e [InlineData]
* renomeie seus testes e classe de teste para usar o padrão discutido na aula
* crie nova classe e método de teste para validar quando um leilão já finalizado recebe nova oferta

Caso já tenha feito, ótimo! Se ainda não, é importante que você execute o que foi visto nos vídeos para poder continuar com a próxima aula.

# Referências Utilizadas

## Referências mencionadas nessa aula

### 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>

# Para saber mais: outras formas de enviar dados

No vídeo **Teorizando com xUnit** você deve ter ficado se perguntando o seguinte: e se eu tiver dados de entrada mais complexos do que um simples **double**? E se quiser parametrizar objetos de entrada? O que fazer?

Para isso o xUnit possui outras duas anotações, [MemberData] e [ClassData], onde você pode construir os dados de entrada usando membros da própria classe de teste ou então uma classe específica para isso.

Veja mais detalhes nesse artigo (em inglês): <https://andrewlock.net/creating-parameterised-tests-in-xunit-with-inlinedata-classdata-and-memberdata/>

# O que aprendemos?

Nessa aula você:

* conheceu as **classes de equivalência** e descobriu que é missão do testador descobrí-las
* aprendeu sobre as anotações do xUnit [Theory] e [InlineData], que são usadas para injetar cenários diferentes no mesmo teste
* descobriu um padrão para nomear seus métodos e classes de teste

# Manutenção dos testes

Quando o leilão é criado, já não entra no pregão; há um período de divulgação da peça e marketing para atrair interessados.

Precisamos criar um novo estado LeilaoAntesDoPregao no enumerado EstadoLeilao e estanciá-lo em Leilao(). Em seguida, o estado é em andamento no método IniciaPregao().

public Leilao(string peca)

{

Peca = peca;

\_lances new List<Lance>();

Estado = EstadoLeilao.LeilaoAntesDoPregao;

}

//código omitido

public void IniciaPregao()

{

Estado = EstadoLeilao.LeilaoEmAndamento;

}

Para esta situação, já finalizamos a implementação e podemos executar todos. Como ainda não criamos nenhum teste para a nova funcionalidade, e quase todos os já existentes apresentam falhas. Ou seja, quando inserimos esta nova, todo o resto do código é quebrado.

Os testes executados nos fazem receber um feedback rápido de que nossa implementação regrediu ao invés de evoluir, sendo então chamados de **testes de regressão** por justamente nos passarem esta clareza antes de finalizar o produto.

Para corrigir esses problemas, realizamos **manutenções** nos próprios cenários criados. Em RetornaMaiorValorDadoLeilaoComPeloMenosUmLance(), quando recebemos ofertas, o pregão ainda não foi iniciado. Logo, o comportamento foi alterado mas precisamos que este seja válido. Então, inserimos o comando leilao.IniciaPregao() logo antes do laço foreach() deste teste para executar novamente.

Assim, vemos no Gerenciador que o teste é aprovado, demonstrando que nem sempre é necessário alterar o código de produção, mas sim realizar mudanças no cenário elaborado.

A mesma coisa acontece no caso de NaoPermiteNovosLancesDadoLeilaoFinalizado() que recebe ofertas ainda sem ter iniciado o pregão. Então, inserimos o mesmo comando leilao.IniciaPregao() antes de foreach() da mesma maneira que fizemos anteriormente.

Executando tudo, os testes voltam a funcionar da forma esperada. Então fica essa dica: muitas vezes mudamos o comportamento do sistema e alguns testes deixam de ser válidos. Alguns inclusive devem ser excluídos por não fazerem mais sentido, enquanto outros precisam ser mantidos e modificados, daí a importância de um código de qualidade dos testes.

Mas fica a questão: Como garantimos a disciplina de sempre ter testes para nas novas funcionalidades criadas?

# Objetivos da aula

### Como evitar que a implementação de novas funcionalidades quebre o sistema?

Para responder essa pergunta vamos conhecer uma prática que vai permitir a evolução do nosso conjunto de testes a cada funcionalidade. Esse conjunto de testes servirá como **testes de regressão** para a próxima funcionalidade e espera-se que aumente a confiança no código que vai subir para produção.

**[Errata] Nome da classe de teste**

O professor se confundiu e escreveu como nome da classe de teste para validar o método RecebeLance() **LeilaoRecebeOferta**.

Porém, conforme ele mesmo mencionou, a convenção adotada é utilizar o nome da classe concatenado com o nome do método sob teste. Portanto o nome correto da classe de teste seria **LeilaoRecebeLance**.

# Novos testes a cada funcionalidade

No passo anterior, ficamos com a questão: como garantimos teste para cada nova funcionalidade?

Uma prática comum nas empresas é ter um testador ou uma equipe de teste (QA) responsável justamente por testar os incrementos dos softwares, deixando o desenvolvedor livre para trabalhar eu sua área com o respaldo dos demais.

Isso pode acarretar problemas, pois o foco somente no desenvolvimento da funcionalidade e a dependência de outros para teste torna o trabalho mais demorado e menos dinâmico, tendo em vista que os testes automatizados fornecem um feedkback imediato visando otimizar o tempo de trabalho.

Assim, podemos ter total controle sobre o programa se implementarmos essas verificações automáticas desde que haja uma disciplina suficiente para tal. A proposta aqui é alterar a orientação do fluxo de trabalho; ao invés de desenvolver para depois testar, criamos primeiro o teste e depois escrevemos o código do negócio.

Começamos indo para as classes de teste e elaborando pelo menos um cenário inicial para a nova implementação que ainda não foi desenvolvida, fazendo com que falhe num primeiro momento. Em seguida, fazemos com que funcione escrevendo o código da funcionalidade.

Esta prática de caráter aparentemente contraintuitiva é bastante usada por times ágeis e chama-se **Test Driven Development** ou **TDD**, a qual orienta o desenvolvimento pelos testes.

Veremos seu funcionamento no passo seguinte.

# Praticando TDD

Veremos na prática o funcionamento do **TDD**.

Implementaremos a funcionalidade que impede o cliente de fazer lances consecutivos no leilão já iniciado, e com a nova prática, desenvolvemos o teste primeiro para este caso.

O método sob verificação é o LeilaoRecebeOferta(), então abrimos a aba de sua classe. Na declaração, usamos a anotação [Fact] e geramos um novo método com nome que contenha o cenário e a expectativa: NaoAceitaProximoLanceDadoMesmoClienteRealizouUltimoLance().

Em seu corpo, copiamos e colamos o teste que está na sequência para nos basearmos. Retiramos a parte do laço foreach() e o substituímos por leilao.RecebeLance() com o Fulano ofertando R$800,00.

A quantidade obtida é o valor da contagem do enumerado de lances que está no leilão e a quantidade esperada é de 1.

public class LeilaoRecebeOferta

{

[Fact]

public void NaoAceitaProximoLanceDadoMesmoClienteRealizouUltimoLance()

{

//Arrange - cenário

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

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

leilao.IniciaPregao();

leilao.RecebeLance(fulano, 800);

//Act - método sob teste

leilao.RecebeLance(fulano, 1000);

//Assert

var qtdeEsperada = 1;

var qtdeObtida = leilao.Lances.Count();

Assert.Equal(qtdeEsperada, qtdeObtida);

}

//código omitido

}

Executamos tudo no Gerenciador de Teste para ver a falha neste novo implementado. Na aba Leilao.cs, vamos à declaração de RecebeLance() para escrever o código de produção. Precisamos verificar se o cliente passado como argumento deste método é diferente do último cliente a fazer um lance.

Logo, inserimos um laço if() dentro do if() já existente para verificação. Guardamos o ultimo usuário a fazer um lance em uma variável da classe de instância \_ultimoCliente que é o campo. Se for diferente, guardamos este dado para os futuros lances.

public void RecebeLance(Interessada cliente, double valor)

{

if (Estado == EstadoLeilao.LeilaoEmAndamento)

{

if (cliente != \_ultimoCliente)

{

\_lances.Add(new Lance(cliente, valor));

\_ultimoCliente = cliente;

}

}

}

Assim, falta-nos declarar este novo campo na classe Leilao, começando com o cliente diferente de null, entrando com um novo lance para guardar a referência no campo \_ultimoCliente para depois verificarmos se são diversos.

public class Leilao

{

private Interessada \_ultimoCliente;

private IList<Lance> \_lances;

public IEnumerable<Lance> Lances => \_lances;

public string Peca { get; }

public Lance Ganhador { get; private set; }

public EstadoLeilao Estado {get; private set; }

//código omitido

}

Com os blocos implementados, podemos executar para ver que agora o teste passa, mas outros voltam a falhar, pois o cenário mudou.

Em NaoPermiteNovosLancesDadoLeilaoFinalizado(), o leilão termina e são emitidos novos lances para o mesmo interessado. Com a nova regra que não permite ofertas consecutivas, a expectativa de quantidade mudou causando falha. Podemos alterar a expectativa nas linhas de [InlineData] deste método ou podemos mudar o cenário. Aqui, optamos pela segunda alternativa adicionando dois interessados.

Ao invés do array foreach(), verificamos com um laço comum for() o tamanho do array, inserindo um if() para testar se se o índice é par. Se for, recebe um lance de Fulano, e se for ímpar, recebe de Maria.

Então, estamos simulando um leilão com dois clientes alternando as ofertas entre si. Por enquanto, não há lances consecutivos neste cenário fazendo com que o teste seja aprovado novamente.

public void NaoPermiteNovosLancesDadoLeilaoFinalizado(

int qtdeEsperada, double[] ofertas)

{

//Arrange - cenário

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

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

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

leilao.IniciaPregao();

for (int i = 0; i < ofertas.Lenght; i++)

{

var valor = ofertas [i];

if ((i%2)==0)

{

leilao.RecebeLance(fulano, valor);

}

else

{

leilao.RecebeLance(maria, valor);

}

}

//código omitido

}

Mesmo executando tudo no Gerenciador, o teste continua fazendo lances consecutivos para o mesmo interessado provocando erros.

Para resolver, copiamos a linha da variável maria e o laço for() e os colamos no cenário de RetornaMaiorValorDadoleilaoComPeloMenosUmLance(), substituindo seu foreach() também.

public void RetornaMaiorValorDadoleilaoComPeloMenosUmLance(

double valorEsperado,

double[] ofertas)

{

//Arrange - cenário

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

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

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

leilao.IniciaPregao();

for (int i = 0; i < ofertas.Lenght; i++)

{

var valor = ofertas [i];

if ((i%2)==0)

{

leilao.RecebeLance(fulano, valor);

}

else

{

leilao.RecebeLance(maria, valor);

}

}

//código omitido

}

Executamos tudo para ver que todos os testes voltam a funcionar com a inclusão do novo. Por isso, usar a prática de TDD e criar uma disciplina para acompanhar a proposta contraintuitiva pode nos facilitar bastante o trabalho.

Outra vantagem é a possibilidade de reduzir as condicionais e melhorar o código. Veremos a seguir.

# Melhorando nosso código

Veremos como melhorar nosso código com duas condicionais if() nesta etapa.

Caso apareça uma outra condição, é interessante extrair para um método os critérios para o leilão aceitar os lances. Desta forma, podemos isolar e evoluir a partir deste.

Para isso, acesse a aba Leilao.cs para criar um método privado que retorna um booleano acima de RecebeLance() chamado NovoLanceEhAceito(). Como parâmetro, recebe o cliente e o valor do lance como argumento de entrada. O retorno são os dois primeiros critérios com o leilão em andamento e o cliente diferente do último para não aceitar ofertas consecutivas.

Com isso, substituímos o argumento de if() pelo novo método isolando os critério de aceitação de um novo lance.

private bool NovoLanceEhAceito(Interessada cliente, double valor)

{

return (Estado == EstadoLeilao.LeilaoEmAndamento)

&& (cliente != \_ultimoCliente);

}

public void RecebeLance(Interessada cliente, double valor)

{

if (NovoLanceEhAceito(cliente, valor))

{

\_lance.Add(new Lance(cliente, valor));

\_ultimoCliente = cliente;

}

}

Agora, como podemos testar métodos privados uma vez que não são visíveis nas classes de teste? Para isso, consultamos o artigo da Microsoft que apresenta as melhores práticas para testes unitários acessível [aqui](https://docs.microsoft.com/pt-br/dotnet/core/testing/unit-testing-best-practices#validate-private-methods-by-unit-testing-public-methods).

Lemos a parte que indica que na maioria dos casos de métodos privados não há necessidade de verificação, dizendo que em algum momento do código este deve ser sensibilizado por algum método publico. Logo, se este último é testado, o outro já é submetido automaticamente.

Executamos novamente do Gerenciador de teste para confirmar o funcionamento. Assim, fizemos melhorias no programa interno sem alterar o comportamento do sistema externo, ou seja, realizamos uma **refatoração**.

Com isso, encerramos o ciclo do TDD: começamos escrevendo um novo teste sem nenhum escrito de negócio que, ao ser executado, apresenta falha. Depois, corrigimos fazendo com que a verificação passe, seja no código de produção ou cenário. Com isso, a execução é realizada com sucesso e podemos refatorar o código para torná-lo mais eficiente e limpo.

Esta prática permite acompanhar todas as etapas até o resultado desejado, sendo que uma se conecta à outra de forma mais clara e possibilita a criação de novos testes sempre que necessário.

# Consolidando os conhecimentos

Chegou a hora de você seguir todos os passos realizados por mim durante a aula...

* criar testes para a funcionalidade **Leilão tem período inicial antes do pregão**, conforme esse exercício e esse também
* criar testes para a funcionalidade **Leilão não permite lances consecutivos do mesmo cliente** usando a prática TDD
* escrever o código que vai implementar a funcionalidade acima
* refatorar o código do método RecebeLance() extraindo as condições de aceitação de um novo lance para um método privado

# Referências Utilizadas

## Referências usadas na aula

#### 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>

# O que aprendemos?

Nessa aula você:

* aprendeu o que são testes de regressão
* percebeu que testes também precisam ser mantidos e até mesmo descartados
* conheceu a prática **TDD**, Test Driven Development, e a aplicou em uma nova funcionalidade
* descobriu que o TDD permite que seu código seja melhorado sem regredir o sistema, termo conhecido como **refatoração**
* conheceu o ciclo TDD: escrever testes que falhem >> corrigir os testes ajustando código de produção >> refatorar o código

# Terminando leilão sem pregão

Começamos abrindo o Gerenciador de Testes para acessar o método RetornaZeroDadoLeilaoSemLances(), o qual anuncia a criação de um leilão seguido do término do pregão sem nem mesmo ter começado.

Para o produto final, é indicado fazer uma comunicação através de uma exceção indicando que o pregão não pode ser terminado. Crie um novo teste chamado LancaInvalidOperationExceptionDadopregaoNaoIniciado() para representar este cenário declarando com uma anotação [Fact] antes do método anterior.

O TerminaPregao() lança um InvalidOperationException se o pregão não for iniciado. Copie e cole o conteúdo de RetornaZeroDadoLeilaoSemLances() para dentro da nova declaração. Este último falhará com a nova funcionalidade implementada, mas será corrigido depois.

No novo cenário, não precisamos de clientes nem ofertas. Agora, a verificação não é mais para o valor esperado, é para o lançamento da exceção. O recurso da linguagem C# para capturá-la é o try catch. Ainda, usamos um método de Assert chamado IsType<>, onde perguntamos se o tipo da exceção é InvalidOperationException.

[Fact]

public void LancaInvalidOperationExceptionDadopregaoNaoIniciado()

{

//Arrange - cenário

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

try

{

//Act - método sob teste

leilao.TerminaPregao();

}

catch (System.Exception e)

{

//Assert

Assert.IsType<System.InvalidOperationException>(e);

}

}

Executamos apenas este teste para ver que passa com sucesso, pois o TerminaPregao() ainda não implementou a nova funcionalidade e, com isso, o catch não captura uma exceção lançada. Se tentarmos finalizar o pregão e seu método não gerar nenhuma exceção, forçamos uma falha na linha seguinte com Assert.True(false); e rodamos novamente.

try

{

//Act - método sob teste

leilao.TerminaPregao();

Assert.True(false);

}

Assim, podemos implementar a funcionalidade, já que este nosso teste está falhando. Veremos a seguir um modo mais eficiente de testar exceções usando o **Unit**.

# Objetivos da aula

Exceções fazem parte do comportamento de nosso sistema e precisam ser testadas também! Mas...

#### Como testá-las de modo elegante e prático com o xUnit?

Nessa pequena aula veremos os recursos do xUnit para testar exceções e não deixar esse tipo de comportamento de fora de nosso conjunto de testes.

# Exceções com xUnit

O xUnit possui uma maneira de testar exceções usando um método da classe Assert chamado **Throws<>()**.

Começamos excluindo o bloco try catch. No argumento de entrada, insira o método sob teste leilao.TerminaPregao() como um delegate.

[Fact]

public void LancaInvalidOperationExceptionDadopregaoNaoIniciado()

{

//Arrange - cenário

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

//Assert

Assert.Throws<System.InvalidOperationException>(

//Act - método sob teste

() => leilao.TerminaPregao()

);

}

Desta forma, nosso código fica sucinto e limpo. Executamos apenas este teste no Gerenciador, verificando se a exceção está sendo lançada para o método correto. A expectativa é de que o teste continue falhando, sendo a maneira com a qual o xUnit testa exceções. Comparando esta maneira com outros frameworks neste [link](https://xunit.github.io/docs/comparisons), as diferenças são bastante claras.

Implementamos o comportamento indo na declaração de TerminaPregao() em Leilao.cs e inserindo uma condicional if() para o estado.

public void TerminaPregao()

{

if (Estado != Estadoleilao.LeilaoEmAndamento)

{

throw new System.invalidOperationException();

}

//código omitido

}

Rodamos a execução deste teste para ver seu funcionamento com sucesso. Para garantir que o sistema não está regredindo, execute todos os testes novamente.

O RetornaZeroDadoLeilaoSemLances() apresenta erro no Gerenciador pois também encerra o pregão sem tê-lo iniciado, sendo necessário que adicionemos o comando que o começa.

[Fact]

public void RetornaZeroDadoLeilaoSemLances()

{

//Arranje - cenário

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

leilao.IniciaPregao();

//código omitido

}

Desta forma, todos os testes voltam a funcionar com sucesso após execução. Se quisermos testar inclusive valores ou propriedades dentro da exceção, precisamos capturá-la em uma variável e assim fazer a verificação.

Dentro de LancaInvalidOperationExceptionDadopregaoNaoIniciado(), crie uma variável excecaoObtida antes do comando que finaliza o pregão e adicione Assert.Equal() para mensagem esperada e a exceção obtida sendo propriedade Message.

Esta msgEsperada declarada logo antes de Assert.Equal() imprime "Não é possível terminar o pregão sem que ele tenha começado. Para isso, utilize o método IniciaPregao()".

[Fact]

public void LancaInvalidOperationExceptionDadopregaoNaoIniciado()

{

//Arrange - cenário

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

//Assert

var excecaoObtida = Assert.Throws<System.InvalidOperationException>(

//Act - método sob teste

() => leilao.TerminaPregao()

);

var msgEsperada = "Não é possível terminar o pregão sem que ele tenha começado. Para isso, utilize o método IniciaPregao().";

Assert.Equal(msgEsperada, excecaoObtida.Message);

}

Execute apenas esse teste selecionado para avaliar o erro apontado de que a mensagem esperada não foi exibida. Esta deve ser copiada e colada como parâmetro no momento do lançamento da exceção dentro de System.invalidOperationException() em TerminaPregao().

Execute todos os testes novamente para ver todos funcionando com sucesso.

Neste passo, verificamos como testar exceções usando xUnit através do método Throws(), inclusive testando as propriedades deste objeto capturando a exceção em uma variável.

# Testando outra exceção

Fixaremos a técnica de usar a exceção usando um novo comportamento.

Imagine que podemos criar um leilão com lances negativos que devem ser impedidos gerando uma exceção dependente de um teste. Para isso, temos uma nova classe de equivalência para o fato de valores de ponto flutuante negativo.

Neste caso, o método sob teste é o construtor da classe Lance. Clique com o botão direto sobre Alura.LeilaoOnline.Tests no Gerenciador de Soluções, acesse "Adicionar > Classe..." para nomear como LanceCtor e finalizar em "Adicionar".

Declare-a como pública para que o xUnit possa reconhecê-la e importe as classes escrevendo using Alura.LeilaoOnline.Core; e using Xunit; no topo do código.

De volta ao corpo de LanceCtor, implemente o método de teste para lançar uma exceção quando o construtor do Lance for criado com valor negativo. Com a anotação [Fact], insira LancaArgumentExceptionDadoValorNegativo().

O cenário é a própria construção de uma oferta negativa -100. Verifique se a exceção System.ArgumentException é gerada para quando o método Lance recebe de um cliente null o valorNegativo.

public class LanceCtor

{

[Fact]

public void LancaArgumentExceptionDadoValorNegativo()

{

//Arrange - cenário

var valorNegativo = -100

//Assert

Assert.Throws<System.ArgumentException>(

//Act

() => new Lance(null, valorNegativo)

);

}

}

No Gerenciador de Testes, execute apenas este clicando com o botão direito para escolher "Executar Testes Selecionados" e receber um retorno indicando falha, o que significa podemos implementar essa funcionalidade segundo o fluxo **TDD**.

No construtor em Lance.cs, verifique com if() se caso o valor do lance for menor que zero, deve ser lançada a exceção ArgumentException. Em seguida, insira a mensagem "valor do lance deve ser igual ou maior que zero".

public class Lance

{

public Interessada Cliente { get; }

public double Valor { get; }

public Lance(Interessada cliente, double valor)

{

if (valor<0)

{

throw new System.ArgumentException("Valor do lance deve ser igual ou maior que zero.");

}

Cliente = cliente;

Valor = valor;

}

}

Testamos novamente executando apenas a nova verificação. Depois, executamos tudo no Gerenciador de Testes para confirmar que todos passaram sem problemas.

Quanto mais implementamos cenários e testes, cobrimos nosso código e conferimos mais segurança ao projeto e à evolução da aplicação.

# Consolidando os conhecimentos

Chegou a hora de fazer os passos realizados por mim durante a aula, sempre dentro do ciclo TDD:

* crie um teste para verificar se a exceção InvalidOperationException será lançada quando um leilão é encerrado sem ter um pregão
* corrija o teste anterior ajustando o código do método TerminaPregao()
* crie um teste para verificar se a exceção ArgumentException será lançada quando um lance é criado com valor negativo
* corrija o teste anterior incluindo uma validação adequada no construtor da classe Lance

# Referências Utilizadas

## Referências usadas na aula

**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>

# O que aprendemos?

Nessa aula você:

* descobriu que exceções também são comportamentos do sistema e também devem ser testadas
* conheceu o método Assert.Throws<>() que permite verificar a expectativa de lançamento de determinada exceção

# Outra forma de avaliar o leilão

Nesta etapa, implementaremos uma forma de avaliar a melhor oferta por valor superior mais próximo do destino.

O cenário consiste no leilão iniciado com a nova modalidade de avaliação cujo objetivo é de **R$1.200,00** com vários lances: R$800, R$1150, R$1.400,00 e R$1.250,00, sendo este último o vencedor mais aproximado numericamente.

Com a orientação TDD, começamos pela implementação do teste. Como se trata do final do leilão, a classe de teste é a LeilaoTerminaPregao. Dentro desta, crie o método público RetornaValorSuperiorMaisProximoDadoLeilaoNessaModalidade() com a anotação [Theory] para receber valores.

Para o [InlineData], temos 1250 e um array de oferta com os valores definidos anteriormente. Estes dados de entrada são injetados como argumentos do método.

public class LeilaoTerminaPregao

{

[Theory]

[InlineData(1200, 1250, new double [] { 800, 1150, 1400, 1250 })]

public void RetornaValorSuperiorMaisProximoDadoLeilaoNessaModalidade(

double valorDestino,

double valorEsperado,

double[] ofertas)

{

//Arrange - Cenário

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

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

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

for (int i = 0; i < ofertas.Lenght; i++)

{

if ((i % 2 == 0))

{

leilao.RecebeLance(fulano, ofertas[i]);

}

else

{

leilao.RecebeLance(maria, ofertas[i]);

}

}

//Act

leilao.TerminaPregao();

//Assert

Assert.Equal(valorEsperado, leilao.Ganhador.Valor);

}

//código omitido

}

O compilador aponta que valorDestino não foi usado. No passo seguinte, veremos onde deve ser passado este valor de destino.

# Objetivos da aula

Repare que o valor de destino do leilão não foi utilizado na criação do cenário apresentado. Mas como vou saber qual é o lance ganhador nessa nova modalidade sem comparar o valor do lance com o de destino?

Essa dificuldade nos leva a refletir que algumas novas implementações vão exigir que alteremos a **interface** de uma classe, ou seja, como ela interage com outras classes. Você verá que nesse exemplo teremos que mudar a classe Leilao para aceitar um valor de destino para essa modalidade.

O processo de planejar a interface de uma classe é chamado de **Design OO**. Nosso objetivo nessa aula é mostrar que o TDD também pode ajudar o design!

### Como assim, Daniel?

# Testes ajudam o design

Neste novo teste, destacamos uma novidade: até o momento, não tínhamos precisado realizar alterações na interface da classe Leilao ou qualquer outra do negócio.

A definição de interface de uma classe ou objeto pode ser vista [aqui](https://en.wikipedia.org/wiki/Object-oriented_design), e fala sobre a interação com esta através de métodos, propriedades e construtor.

Então, ainda não havíamos necessitado repensar a forma como interagimos com o leilão. Desde o primeiro exercício deste curso, definimos o sistema de leilão na biblioteca de classes e continuamos usando toda a interface sem nenhuma modificação.

O processo de elaborar a forma como os objetos interagem entre si é chamado **design**. Todos os testes escritos até agora simulam a interação com os objetos das classes de negócio, ou seja, as verificações estão ajudando a validar este processo.

Pensaremos este design usando o teste implementado anteriormente, sendo o valorDestino indicativo de que a modalidade de avaliação do leilão mudou. Quando este é criado, já deve indicar sua forma avaliativa.

No construtor, passamos o valor de destino que, quando for maior do que zero, significa que modalidade é por oferta superior mais próxima ao destino. Desta forma:

public class LeilaoTerminaPregao

{

[Theory]

[InlineData(1200, 1250, new double [] { 800, 1150, 1400, 1250 })]

public void RetornaValorSuperiorMaisProximoDadoLeilaoNessaModalidade(

double valorDestino,

double valorEsperado,

double[] ofertas)

{

//Arrange - Cenário

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

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

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

//código omitido

}

}

No construtor da classe Leilao, aplicamos as modificações. Crie uma nova propriedade somente de leitura para representar o valorDestino, pois vai receber este valor justamente em seu construtor. Além disso, dizemos que este parâmetro assume o padrão valor zero caso nenhum outro seja passado para a modalidade cujo ganhador dá a maior oferta.

Em TerminaPregao(), defina o ganhador segundo as duas modalidades em um laço if(). Quando o valor de destino for maior que zero, o vencedor é definido pela forma avaliativa de maior lance próximo ao objetivo. Caso contrário, igual ou menor que zero, é o valor padrão.

Calculamos a modalidade onde o ganhador cujos lances possuem um valor maior que o de destino dentre todas as ofertas. Depois, ordene pelo valor para podermos pegar o primeiro item do subconjunto DefaultIfEmpty().

public class Leilao

{

private Interessada \_ultimoCliente;

private IList<Lance> \_ lances;

public IEnumerable<Lance> Lances => \_lances;

public string Peca { get; }

public Lance Ganhador { get; private set; }

public EstadoLeilao Estado { get; private set; }

public double ValorDestino { get; }

public Leilao(string peca, double valorDestino = 0)

{

Peca = peca;

\_lances = new List<Lance>();

Estado = EstadoLeilao.LeilaoAntesDoPregao;

ValorDestino = valorDestino;

}

//código omitido

public void TerminaPregao()

{

if (Estado! = EstadoLeilao.LeilaoEmAndamento)

{

throw new System.InvalidOperationException("Valor do lance deve ser igual ou maior que zero.");

}

if (ValorDestino >0)

{

//modalidade oferta superior mais próxima

Ganhador = Lances

.DefaultIfEmpty(new Lance(null, 0))

.Where(1 => 1.Valor > ValorDestino)

.OrderBy(1 => 1.Valor)

.FirstOrDefault();

}

else

{

//modalidade maior valor

Ganhador = Lances

.DefaultIfEmpty(new Lance(null, 0))

.OrderBy(1 => 1.Valor)

.LastOfDefault();

}

Estado = EstadoLeilao.LeilaoFinalizado;

}

}

Com este código, podemos testar apenas o RetornaValorSuperiorMaisProximoDadoLeilaoNessaModalidade() no Gerenciador para que seja aprovado com sucesso. Para evitar a regressão, Execute todos.

Desta forma, usamos um teste para repensar todo o design da classe Leilao. Veremos outras melhorias a seguir.

# Estratégia de avaliação

Podemos melhorar ainda mais o código de TerminaPregao(); segundo os princípios do TDD, a refatoração deve ser feita após a aprovação de todos os testes.

Execute todos no Gerenciador e confira se estão passando. Se sim, já podemos pensar em como aprimorar tendo em mente o design das classes e as referências, lidando com alguns princípios como o **SOLID**, cujo objetivo é transformar o design em sistemas mais **flexíveis**, **fáceis de manter** e de **entender**:

* **S**ingle responsibility principle
* **O**pen/close principle
* **L**iskov substitution principle
* **I**nterface segregation principle
* **D**ependency inversion principle

Para um sistema **fácil de entender**, vemos a questão da modalidade em nosso código. Temos que garantir a clareza da interface para que o teste consiga compreender diferentes modalidades, sendo que o único registro escrito destas está nos dois comentários feitos em TerminaPregao(). Logo, este quesito ainda deve ser melhor estruturado.

O conceito seguinte de **flexibilidade** diz respeito ao acolhimento de novas modalidades de avaliação do leilão pelo código, e o nosso ainda está pouco flexível e engessado.

A terceira característica de um bom design é a **facilidade** com a qual é **mantido**. Para avaliar essa questão, vemos o primeiro princípio do acrônimo SOLID, o *single responsibility principle* ou princípio da responsabilidade única: este diz que uma classe só deve ter uma razão para ser modificada.

Se a classe Leilao tiver mudanças nas modalidades, irá ser alterada e se tiver novas propriedades, também mudará. Então, pode ser que seja modificada por várias razões, e uma delas é a possibilidade de ter categorias de avaliação diferentes. Logo, vemos que ambas estão intimamente ligadas, e chamamos isso de **acoplamento**.

Devemos criar um código que combata esse aspecto para que o sistema todo seja mais eficiente, desacoplando o conceito de modalidade do conceito de leilão. Fazemos este planejamento de design usando os testes.

O método RetornaValorSuperiorMaisProximoDadoLeilaoNessaModalidade() considera a modalidade de oferta com valor superior mais próxima ao destino. Nesta, crie um objeto dessa categoria de avaliação chamado OfertaSuperiorMaisProxima() em uma variável modalidade.

Esta classe ainda não existe, pois estamos usando a abordagem prática do design escrevendo o teste primeiro. A modalidade precisa receber valorDestino como argumento do construtor.

Assim, quando criamos o leilão, já dizemos qual é a categoria de avaliação através de seu conceito como parâmetro do construtor Leilao(). Desta forma, o código fica bem mais legível e eficiente.

[Theory]

[InlineData(1200, 1250, new double [] { 800, 1150, 1400, 1250 })]

public void RetornaValorSuperiorMaisProximoDadoLeilaoNessaModalidade(

double valorDestino,

double valorEsperado,

double[] ofertas)

{

//Arrange - Cenário

var modalidade =

new OfertaSuperiorMaisProxima(valorDestino);

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

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

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

//código omitido

}

Pensando nos demais testes, vá até a declaração de RetornaMaiorValorDadoLeilaoComPeloMenosUmLance() e aplique a modalidade padrão, ou seja, apenas o maior lance. Seguimos o mesmo procedimento com o construtor criando uma variável modalidade com uma nova classe que a representa chamada MaiorValor().

Em seguida, passe essa modalidade como segundo argumento do construtor de leilao. Assim, temos dois tipos que dizem respeito ao mesmo conceito de modalidade de avaliação.

[Theory]

[InlineData(1200, new double [] { 800, 900, 1000, 1200 })]

[InlineData(1000, new double [] { 800, 900, 1000, 990 })]

[InlineData(800, new double [] { 800 })]

public void RetornaMaiorValorDadoLeilaoComPeloMenosUmLance(

double valorEsperado,

double[] ofertas)

{

//Arrange - Cenário

var modalidade = new MaiorValor();

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

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

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

//código omitido

}

Crie um tipo que una as duas categorias substituindo var pela classe ancestral ou abstrata IModalidadeAvaliacao no bloco de RetornaValorSuperiorMaisProximoDadoLeilaoNessaModalidade que represente a avaliação. Dando preferência à interface, fazemos com que as classes a implementem.

public void RetornaValorSuperiorMaisProximoDadoLeilaoNessaModalidade(

double valorDestino,

double valorEsperado,

double[] ofertas)

{

//Arrange - Cenário

IModalidadeAvaliacao modalidade =

new OfertaSuperiorMaisProxima(valorDestino);

//código omitido

}

Assim, finalizamos nosso design usando a abordagem prática através do teste para elaborar melhor e cumprir os quesitos de **facilidade de manutenção**, **legibilidade** e **flexibilidade**, com uma interface que passa a ser o tipo do segundo argumento do construtor e desacoplando os conceitos.

# Inversão de dependência

Implementaremos essas classes e interfaces que ficaram com alertas do compilador. Comece clicando com o botão direito sobre Alura.LeilaoOnline.Core, selecione "Adicionar > classe...", escolha a opção "Interface" e nomeie como IModalidadeAvaliacao.

Exclua as três linhas de importação com using e declare a nova interface como pública. Em seu corpo, há apenas um método que retorna um Lance e se chama Avalia, recebendo como argumentos Leilao e leilao.

namespace Alura.LeilaoOnline.Core

{

public interface IModalidadeAvaliacao

{

Lance Avalia(Leilao leilao)

}

}

Terminada a interface, o teste já a reconhece. Em seguida, crie outra classe para implementar OfertaSuperiorMaisProxima() clicando com o botão direito sobre Alura.LeilaoOnline.Core, selecione "Adicionar > classe...", escolha a opção "Classe" e nomeie como OfertaSuperiorMaisProxima.

Declare-a como pública e implemente a modalidade de avaliação. Importe o namespace do LINQ para usarmos posteriormente e delete as demais linhas de importação com using.

No corpo de Avalia(), retorne os lances do leilão, copie e cole o bloco do comentário //modalidade oferta superior. Falta-nos inserir o valor de destino passado pelo design do construtor da classe.

Voltando em RetornaValorSuperiorMaisProximoDadoLeilaoNessaModalidade, podemos tentar mudar selecionando o nome OfertaSuperiorMaisProxima() e clicando "Ctrl + ." para gerar o construtor. Queremos que seja uma propriedade pública somente de leitura na classe OfertaSuperiorMaisProxima.

using System.LINQ;

namespace Alura.LeilaoOnline.Core

{

public class OfertaSuperiorMaisProxima : IModalidadeAvaliacao

{

public double valorDestino { get; }

public OfertaSuperiorMaisProxima(double valorDestino)

{

ValorDestino = valorDestino;

public Lance Avalia(Leilao leilao)

{

return leilao.Lances

.DefaultIfEmpty(new Lance(null, 0))

.Where(1 => 1.Valor > ValorDestino)

.OrderBy(1 => 1.Valor)

.FirstOrDefault();

}

}

}

Desta forma, a classe OfertaSuperiorMaisProxima foi definida com a implementação bem mais rápida por já termos uma série de outros elementos já prontos na declaração do teste, usando inclusive o próprio Visual Studio para gerar o código, aumentando a produtividade.

Siga a mesma metodologia adotada anteriormente para gerar a nova classe pública MaiorValor implementando a interface IModalidadeAvaliacao e usando Linq.

Da mesma maneira que fizemos antes, o método Avalia() retorna lances do leilão e copiamos o bloco de código do outro comentário //modalidade maior valor.

using System.LINQ;

namespace Alura.LeilaoOnline.Core

{

public class MaiorValor : IModalidadeAvaliacao

{

public Lance Avalia(Leilao leilao)

{

return leilao.Lances

.DefaultIfEmpty(new Lance(null, 0))

.OrderBy(1 => 1.Valor)

.LastOrDefault();

}

}

Assim, podemos fechar as abas MaiorValor.cs, OfertaSuperiorMaisProxima.cs,Leilao.cs e IModalidadeAvaliacao.cs, ficando somente com LeilaoTerminaPregao.cs aberto no Visual Studio.

Com quase todos os testes bem definidos, ainda precisamos mudar o construtor do Leilao para receber a modalidade. Abra o construtor para realizarmos melhorias manualmente.

Não recebemos mais o valorDestino, pois este já está na avaliação de oferta superior mais próxima, sendo substituído por IModalidadedeAvaliacao com um objeto avaliador que implementa a interface como segundo argumento.

Este é o avaliador, e precisamos criar um campo chamado \_avaliador do tipo privado IModalidadedeAvaliacao. Também não temos mais ValorDestino.

Aqui, estamos declarando a dependência da classe Leilao com uma modalidade de avaliação. Logo, para ser instanciado, depende de uma string que representa a peça em questão, dependente também de uma maneira de avaliar o leilão.

Seguimos o princípio da **inversão de dependência** ou dependency inversion principle da sigla SOLID; ao invés do Leilao resolver todas as dependências que uma classe possui, deixamos para os elementos que o usam criarem instâncias de Leilao, injetando a dependência via construtor.

public class Leilao

{

private IModalidadeAvaliacao \_avaliador;

private Interessada \_ultimoCliente;

private IList<Lance> \_lances;

public IEnumerable<Lance> Lances => \_lances;

public string Peca { get; }

public Lance Ganhador { get; private set; }

public EstadoLeilao Estado {get; private set; }

public Leilao(string peca, IModalidadeAvaliacao avaliador)

{

peca = peca;

\_lances = new List<Lance>();

Estado = EstadoLeilao.LeilaoAntesDoPregao;

\_avaliador = avaliador;

}

//código omitido

}

Resta-nos apenas refatorar o código no momento de avaliação do leilão no segundo if() e else de TerminaPregao(). Delete todo o laço para delegar a responsabilidade de gerar o Ganhador para o avaliador, passando o próprio Leilao como argumento de entrada.

Assim, o código fica bastante simples e sucinto com a refatoração. Os testes passaram, melhoramos o código para torná-lo mais flexível, legível e manutível, mantendo o sistema sem regressão.

public void Termina Pregao()

{

if(Estado != EstadoLeilao.LeilaoEmAndamento)

{

throw new System.InvalidOperationException("Não é possível terminar o pregão sem que ele tenha começado. Para isso, utilize o método IniciaPregao().");

}

Ganhador = \_avaliador.Avalia(this);

Estado = EstadoLeilao.LeilaoFinalizado;

}

Execute todos os testes novamente no Gerenciador para observar os resultados. O sistema aponta erros de build, impedindo a execução. Clique em "Não". Isso significa que todos os construtores e códigos que constroem a classe Leilao precisam de uma modalidade.

Insira a linha var modalidade = new MaiorValor(); e adicione modalidade como segundo argumento do construtor Leilao nos cenários de todos os momentos os quais são criados o leilão. Ou seja, a cada criação de novo leilão, é preciso inserir uma modalidade, principalmente a mais simples de MaiorValor(). Use os erros apontados na "Lista de Erros" para identificar onde deve incluir os elementos citados.

Feito isso, execute mais uma vez todos os testes para verificar se todos estão funcionando com sucesso.

Neste passo, usamos os testes para fazer um redesign nas classes, melhorando-o com a técnica TDD e a segurança das verificações no desenvolvimento, impedindo a regressão.

# Consolidando os conhecimentos

Chegou a hora de você fazer os passos realizados por mim durante a aula.

* crie um teste para a nova modalidade **Leilão por valor superior mais próximo** e perceba que precisará redesenhar a interface de sua classe Leilao, incluindo no construtor mais um parâmetro (valor de destino)
* corrija o teste ajustando o construtor e implementando a nova modalidade de avaliação
* extraia o conceito de modalidade para uma interface IModalidadeAvaliacao e duas classes concretas OfertaSuperiorMaisProxima e MaiorValor
* transfira a avaliação do leilão para as duas classes criadas acima
* ajuste a classe Leilao para declarar sua dependência de um objeto que implementa IModalidadeAvaliacao, delegando a tarefa de avaliar o ganhador para esse objeto

# Referências Utilizadas

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

**Conclusão**

**Parabéns!** Chegamos ao final do curso **C#: Testes de unidade e TDD com xUnit**.

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.

Portanto, esperamos que tenha sido de grande utilidade para a nova competência de testador.

**Agradecemos a participação, e até a próxima!**