



TESTES UNITÁRIOS

Treinamento

OBJETIVOS

- Compreender quais as motivações para se escrever testes unitários, e como as alterações que foram feitas podem facilitar esse processo.



01

O que é um teste unitário?

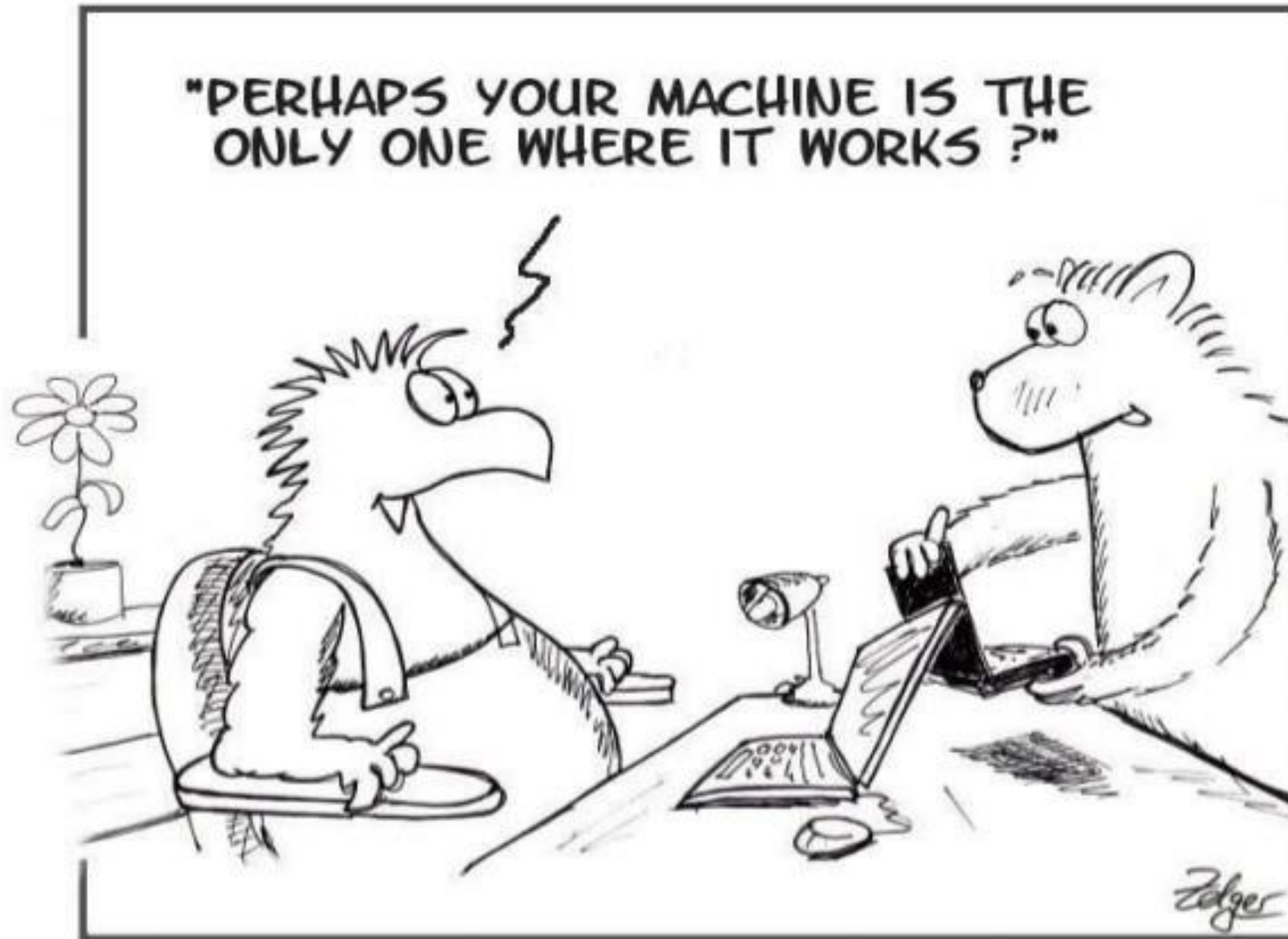
— O que é um teste unitário?

- É o ato de se testar **unidades individuais** de código com o objetivo de se determinar se essas unidades se comportam de acordo com o esperado / projetado;
- “Verificar uma única prerrogativa a respeito do funcionamento de um sistema”

— Por que escrever testes unitários?

- Verificar o comportamento apropriado do sistema;
- Trazer mais segurança quando algo for alterado;
- Ter uma documentação “viva” do que se espera das funcionalidades;
- Servir como orientação para escrita de novos códigos (TDD);
- Identificar quebras de compatibilidade com mais agilidade;

— Por que escrever testes unitários?



It works on my machine

02

Como escrever código testável?

SOLID PRINCIPLES

Os princípios SOLID tem a pretensão de garantir códigos e arquiteturas mais flexíveis, manuteníveis e com maior legibilidade. Frequentemente aplicados em linguagens O.O, esses princípios estão intimamente associados à metodologia Ágil

SOLID PRINCIPLES

1. Single Responsibility Principle (Princípio de Responsabilidade Única)

2. Open Close Principle (Princípio do Aberto Fechado)

3. Liskov Substitution Principle (Princípio da Substituição de Liskov)

4. Interface Segregation Principle (Princípio da Segregação de Interface)

5. Dependency Inversion Principle (Princípio da Inversão de Dependência)

SOLID PRINCIPLES

1 - Single Responsibility Principle (Princípio de Responsabilidade Única)

“A classe e método devem ter uma única responsabilidade”. Ou seja, devem ter apenas uma única razão para ser alterada, devem ter um único papel a se desempenhar.

— Como escrever código testável?



SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

SOLID PRINCIPLES

2 – Open Close Principle (Princípio do Aberto Fechado)

“Objetos ou entidades devem ser abertas para serem estendidas, mas fechadas para serem modificadas (Extensibilidade, Abstração). Código maduro e confiável.”

— Como escrever código testável?



OPEN CLOSED PRINCIPLE

Brain surgery is not necessary when putting on a hat.

SOLID PRINCIPLES

3 – Liskov Substitution Principle

- “-Objetos em um programa devem ser substituíveis por seus subtipos sem que isso altere o funcionamento correto do sistema (Barbara Liskov).
- Classes filhas nunca deveriam infringir as definições de tipo da classe pai.
- Polimorfismo”

— Como escrever código testável?



LISKOV SUBSTITUTION

If it looks like a duck, quacks like a duck, but needs batteries —
you probably have the wrong abstraction.

SOLID PRINCIPLES

4 – Interface Segregation Principle (Princípio da Segregação de Interface)

“Um grande número de pequenas interfaces específicas são melhores do que uma interface de propósito geral”

— Como escrever código testável?



SOLID PRINCIPLES

5 – Dependency Inversion Principle (Princípio da Inversão de Dependência)

“Uma classe deve possuir dependências apenas de interfaces. Nunca deve depender de tipos concretos” (IOC)



1. Classe deve depender apenas de abstrações (Interfaces)
2. Dependenda de uma abstração e não de uma implementação.
3. A classe não deve ser responsável pela construção de suas dependências.

Baixo Acoplamento & Alta coesão

— Como escrever código testável?



DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

— Como escrever código testável?

- É difícil escrever código testável?
- Vale a pena escrever código testável?
- Quem escreve código ruim também não irá escrever testes ruins?

— Como escrever código testável?

- É difícil escrever código testável?
É uma questão de prática. Com o tempo se torna algo natural.
- Vale a pena escrever código testável?
Definitivamente sim. E é uma prática cada vez mais difundida no mundo todo
- Quem escreve código ruim também não irá escrever testes ruins?
Desenvolvedores distintos podem escrever código e testes caso necessário.

02.1

TDD: Test Driven Development

TDD (Test-Driven Development)

Test-Driven Development (ou Desenvolvimento Orientado a Testes) é uma técnica utilizada geralmente em projetos que seguem metodologias ágeis, e prega que, antes de iniciar a escrita de código, devemos construir testes que provem que o código que será escrito funciona como esperado.

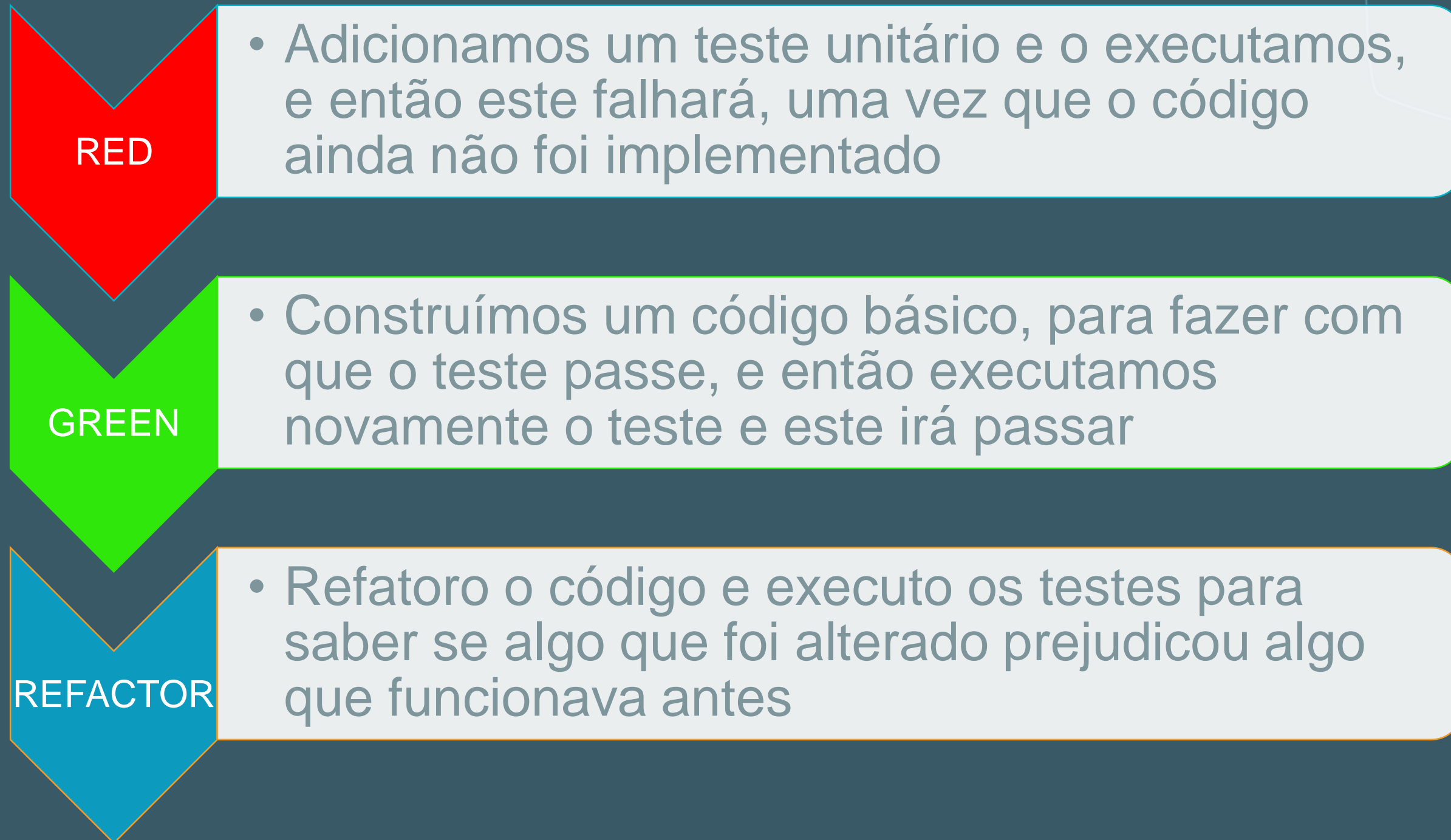
Baseada em uma das técnicas utilizada no XP (extreme programming), conhecida como Test-First (Test primeiro), começou a ser utilizada com o nome Test-Driven Development por Kent Back, em meados do ano 2003, e hoje é muito utilizada pelo crescimento de projetos e equipes ágeis

— Organizações e Convenções

TDD e o ciclo Red/Green/Refactor

A construção dos scripts seguem um fluxo muito conhecido, chamado Red/Green/Refactor, que em português significa "Vermelho/Verde/Refatoração", fazendo referência a Falha/Sucesso/Manutenção do código.

— Organizações e Convenções



Estrutura AAA

Os testes unitários possuem uma estrutura padrão de escrita, que envolve uma preparação, a execução do teste e uma verificação, para provar que o mesmo funcionou. Estes passos são conhecidos como Estrutura AAA (Arrange, Act e Assert).

Arrange (Preparação)

Área do testes onde colocamos todas as pré-condições necessárias para que o teste ocorra conforme esperado.

— Organizações e Convenções

- ***Act (Ações)***

Nesta área serão adicionadas ações que farão a iteração com a unidade que foi ou será construída, de modo que ela gere uma resposta que tornará possível dizer se a unidade funciona como esperado.

- ***Assert (Verificação)***

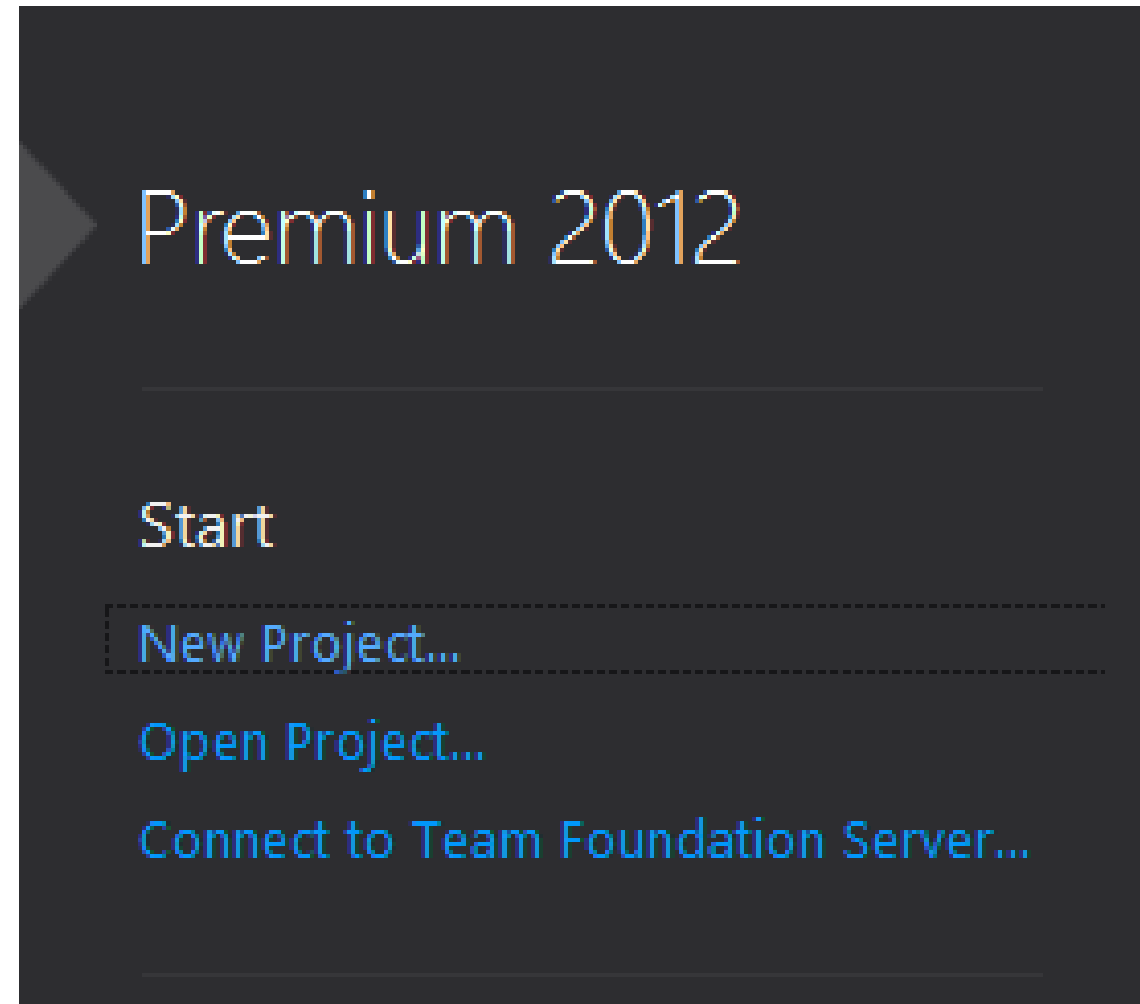
Aqui é onde iremos adicionar a verificação da resposta às ações realizadas.

Não existe um teste sem que exista a asserção que comprove que ele passou ou não. Geralmente asserções verificam respostas numéricas, alfanuméricas ou booleanas.




Criando um teste unitário utilizando TDD em C# com MSTest


- 1) Clico sobre o link "New Project" para criar um novo projeto



GET STARTED HO

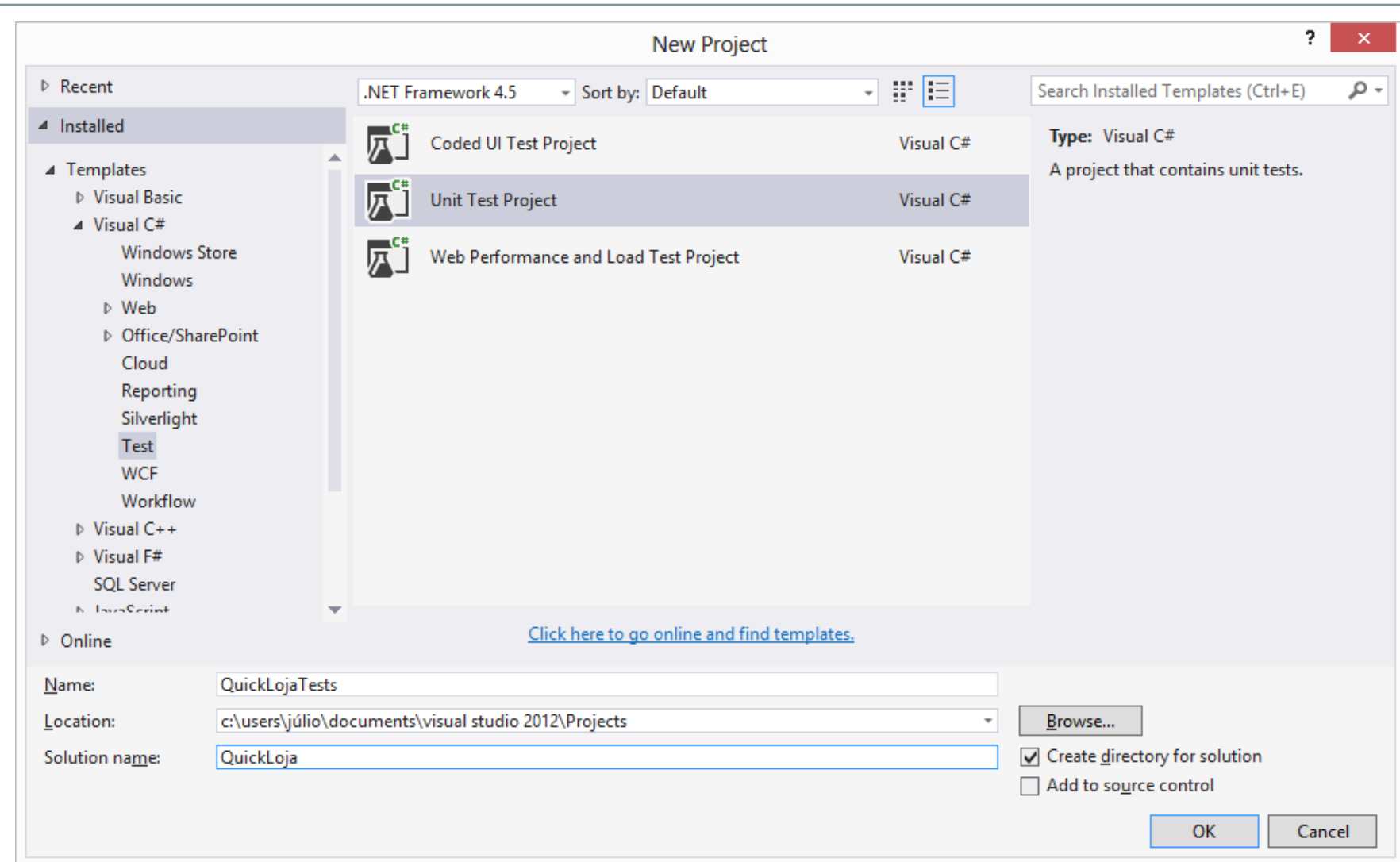
Learn more v

 Improve unit t
workflow with
Test Explorer
improvements

 Using Code Re
to improve qu

Criando um teste unitário utilizando TDD em C# com MSTest

- 2) Clico sobre o a categoria "Templates", depois em "Visual C#" e então em "Test"
- 3) Seleciono "Unit Test Project" e informo o nome do projeto de testes, que será "QuickLojaTests" e o nome da solução que será criada, e na qual este projeto será alocado, neste caso "QuickLoja"



Criando um teste unitário utilizando TDD em C# com MSTest

4) Uma classe com nome padrão será criada, chamada de UnitTest1.cs, e o conteúdo desta classe será:

[TestClass]

Define que esta é uma classe que possui métodos de teste, a classe que possui este atributo precisa ser pública e não pode ser herdada

[TestMethod]

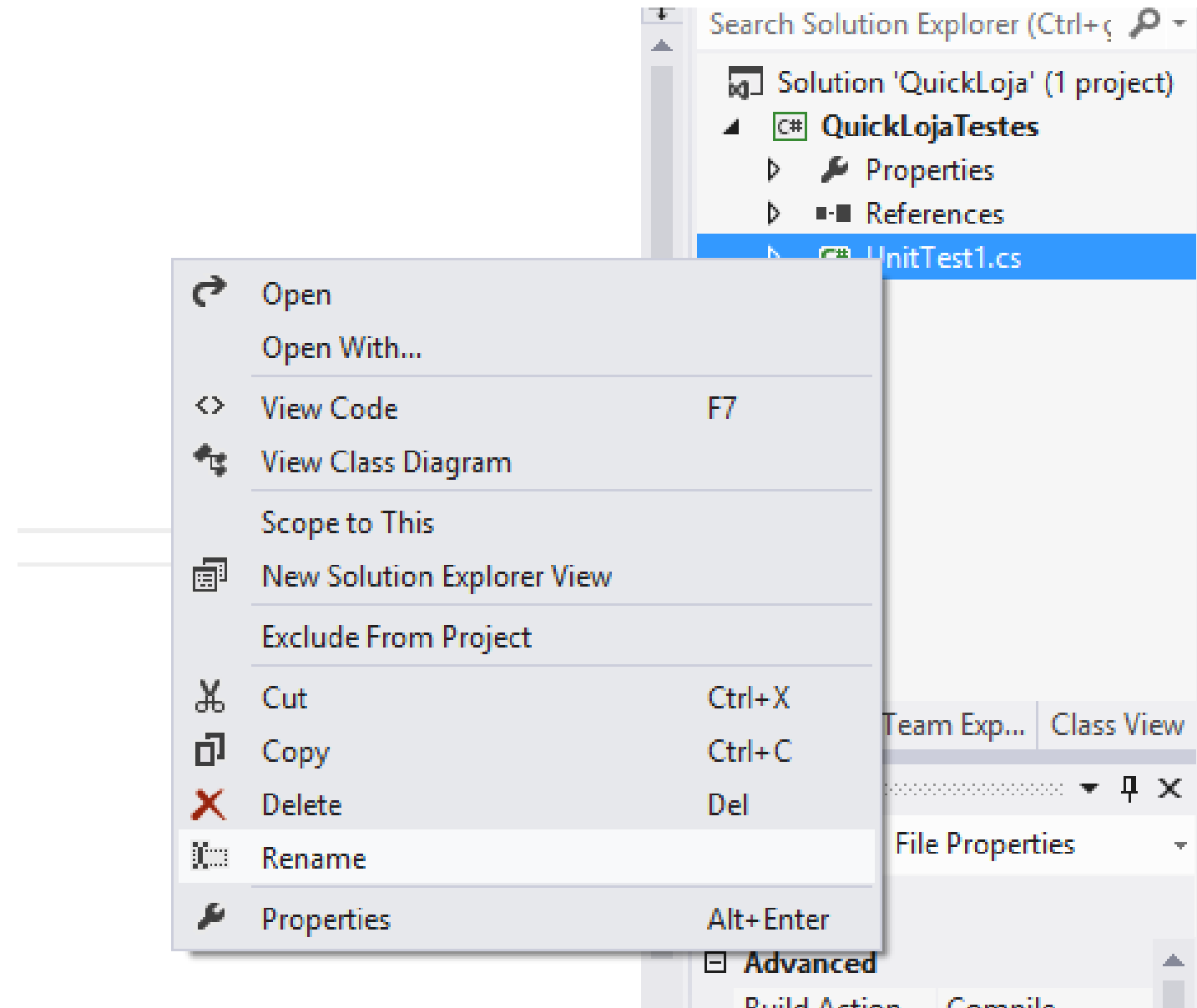
Define que este é um método de teste. Este atributo espera que o método seja um método público e a classe não pode ser herdada

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace QuickLojaTestes
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

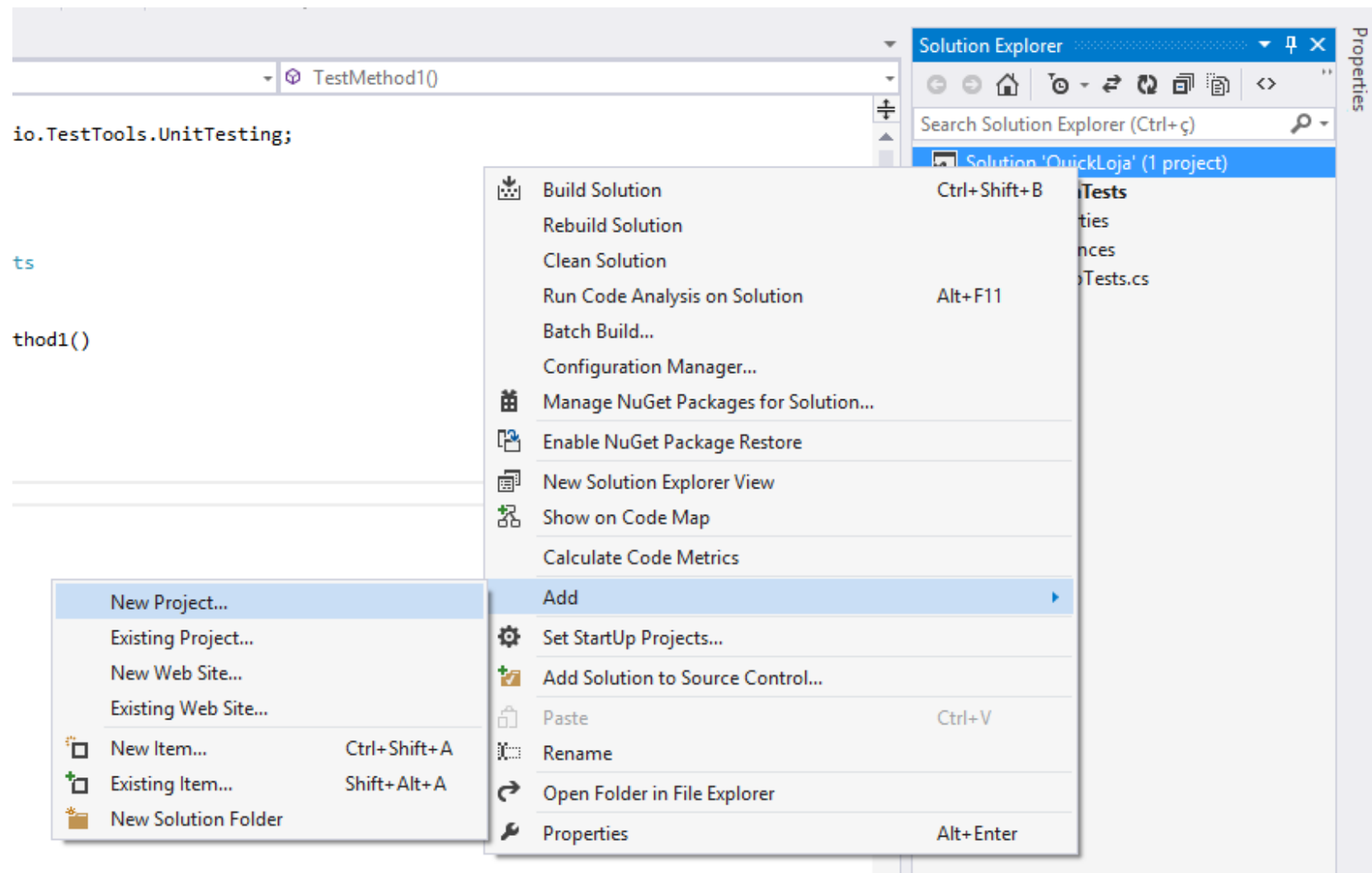

Criando um teste unitário utilizando TDD em C# com MSTest

5) Vamos renomear esta classe para o nome "PedidoTests"



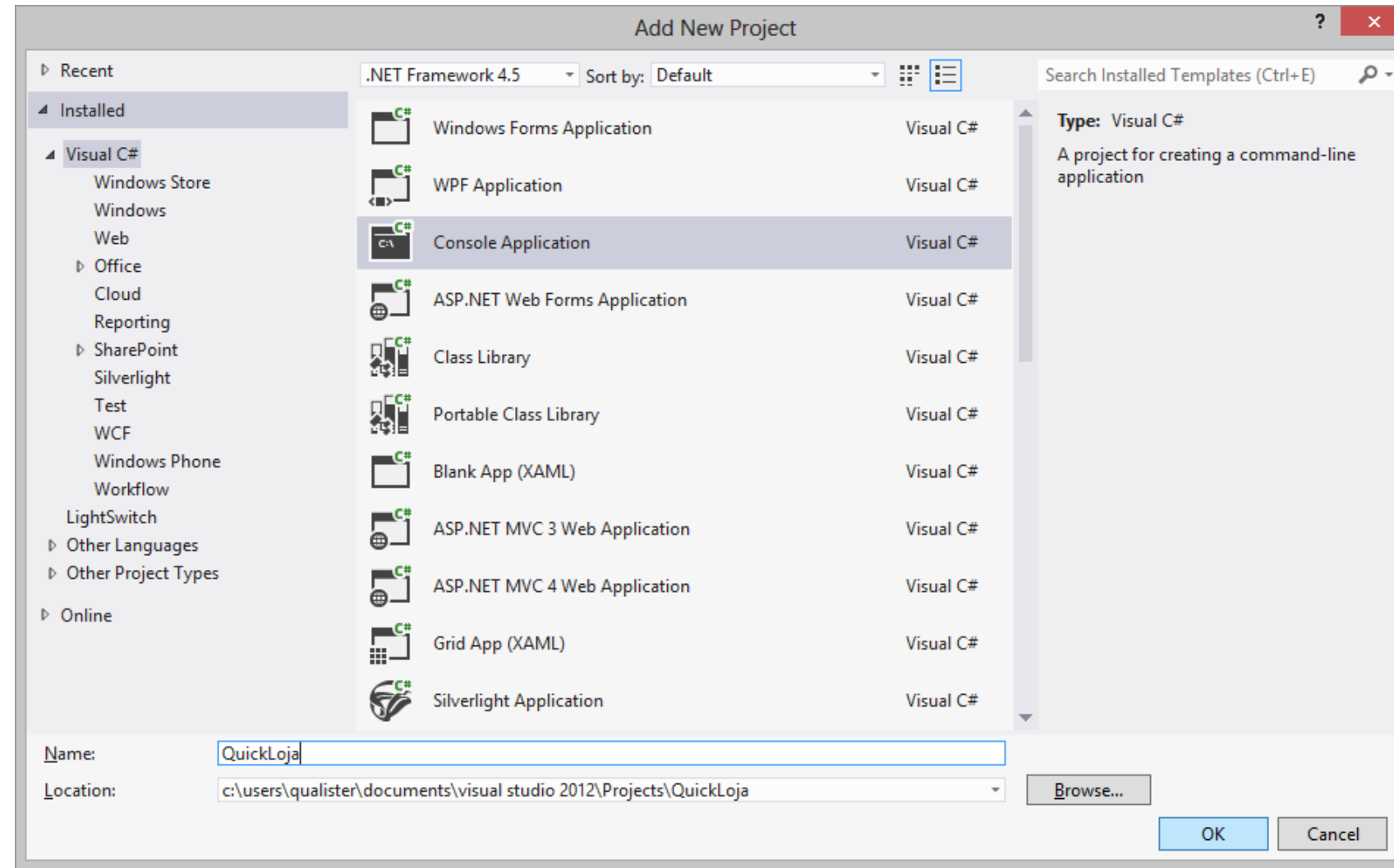
Criando um teste unitário utilizando TDD em C# com MSTest

- 6) Vamos criar um novo projeto dentro da solução QuickLoja, o nome deste projeto será QuickLoja.Tests, este armazenará nossas classes da aplicação. Para isso, clique com o botão direito do mouse sobre a solução, contida na janela "Solution Explorer", e selecione "Add" e depois em "New Project"



Criando um teste unitário utilizando TDD em C# com MSTest

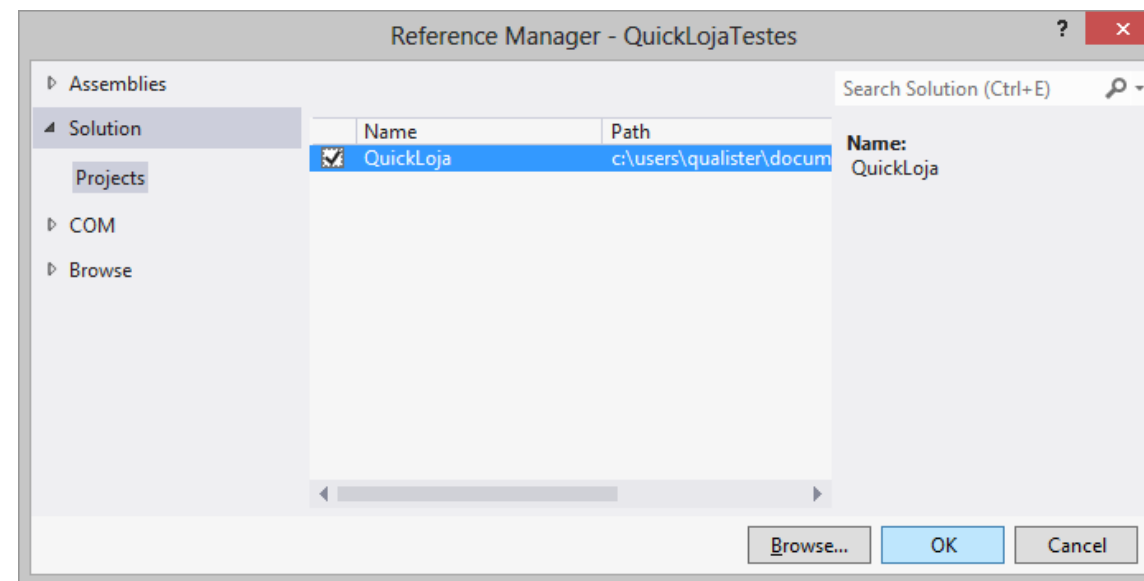
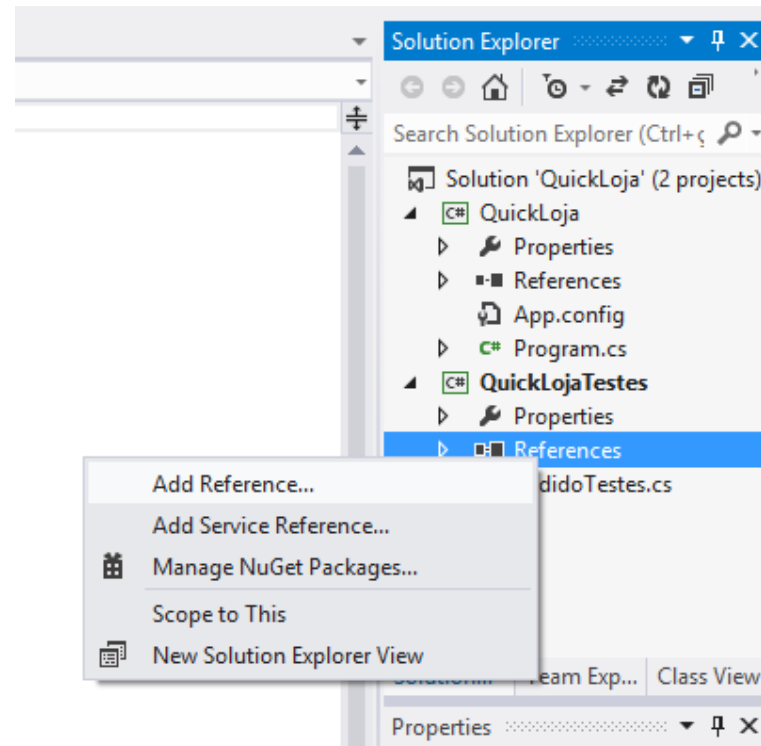
7) Nosso projeto será baseado no template de "Console Application", e como dito antes, será chamado "QuickLoja"



Criando um teste unitário utilizando TDD em C# com MSTest

8) Faremos agora a referência do projeto "QuickLoja" no projeto "QuickLojaTests", isso para que possamos utilizar as classes da aplicação nos testes. Clique sobre o item "References" no projeto "QuickLojaTests" e selecione a opção "Add Reference"

9) Em "Solution" marque a opção "QuickLoja" e pressione "Ok"



Criando um teste unitário utilizando TDD em C# com MSTest

10) Retorne à classe "PedidoTests" e declare a utilização do projeto "QuickLoja"

11) Vamos iniciar a criação do primeiro método de teste, será chamado **Pedido_ListarItens_RetornaQuantidade()**. Este método servirá para testar se o método de listagem de itens, pertencente à classe Pedido, após ser executado retorna o número de itens esperado. Este método será organizado usando o método AAA (Arrange, Act e Assert)

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using QuickLoja;

namespace QuickLojaTestes
{
    [TestMethod]
    public void Pedido_ListarItens_RetornaQuantidade()
    {
        // Arrange

        // Act

        // Assert
    }
}
```

Criando um teste unitário utilizando TDD em C# com MSTest

12) Na área Arrange, iremos instanciar a classe Pedido. O Visual Studio identificará que a classe ainda não existe e então marcá-la-á em vermelho. Se clicarmos no menu de refatoração apresentado abaixo do sublinhado do nome da classe, poderemos selecionar a opção "Generate class for 'Pedido'", que iria gerar uma classe no projeto "QuickLojaTests" e a opção "Generate new type", que gera um novo tipo (Class, Struct, Interface ou Enum) em qualquer projeto contido na Solução atual

```
[TestMethod]
public void Pedido_ListarItens_RetornaQuantidade()
{
    // Arrange
    var pedido = new Pedido();

    // Act

    // Assert
}
```



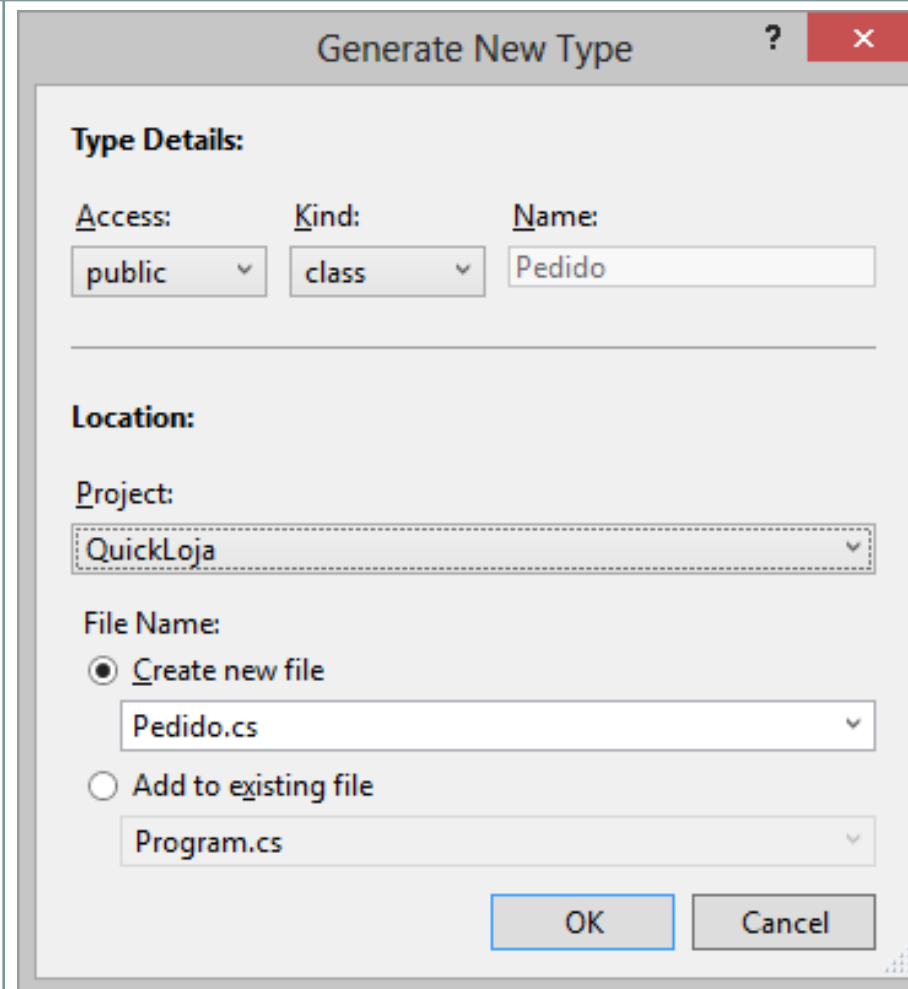
Generate class for 'Pedido'

Generate new type...

Criando um teste unitário utilizando TDD em C# com MSTest

13) Selecionaremos a opção "Generate new type..." e informaremos que queremos criar uma classe pública dentro do projeto QuickLoja

14) Vemos que a classe Pedido foi criada no projeto "QuickLoja"



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace QuickLoja
{
    public class Pedido
    {
    }
}
```

Criando um teste unitário utilizando TDD em C# com MSTest

15) Vemos também que agora a classe Pedido() já não está mais sublinhada de vermelho na classe "PedidoTests"

16) Na área "Act" vamos executar a ação da adição do item ao pedido, espera-se que este método retorne a quantidade de itens já adicionados a este pedido, o Visual Studio grifará o método em vermelho para dizer que o mesmo ainda não foi declarado antes, mas poderemos criar o método a partir do menu de refatoração contido abaixo do sublinhado do método

```
[TestMethod]
public void Pedido_ListarItens_RetornaQuantidade()
{
    // Arrange
    var pedido = new Pedido();

    // Act

    // Assert
}
```

```
// Act
int itens = pedido.ListarItens();
```

```
// Assert
```



Generate method stub for 'ListarItens' in 'QuickLoja.Pedido'

Criando um teste unitário utilizando TDD em C# com MSTest

17) Isso fará com que o método "ListarItens" seja gerado na classe pedido de maneira automática

18) Vemos que uma exception "NotImplementedException" foi adicionada ao método, demonstrando que este método ainda não foi implementado. A classe de teste agora deixa de grifar o método, pois o mesmo agora já está disponível

19) Vamos agora escrever, na área "Assert" do método de teste, o código responsável por validar se o teste teve sucesso, para isso iremos utilizar a classe Assert para fazer esta verificação

```
public int ListarItens()
{
    throw new NotImplementedException();
}
```

```
[TestMethod]
public void Pedido_ListarItens_RetornaQuantidade()
{
    // Arrange
    var pedido = new Pedido();

    // Act
    int itens = pedido.ListarItens();

    // Assert
}
```

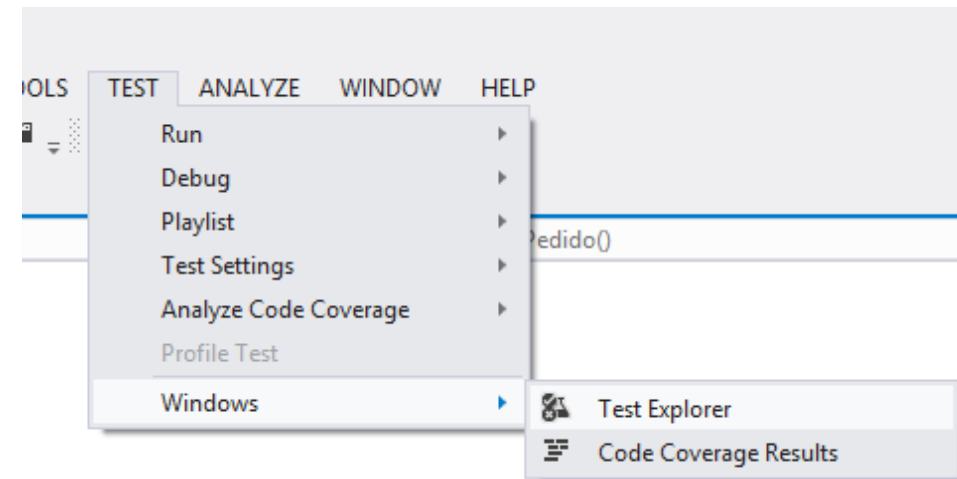
```
[TestMethod]
public void Pedido_ListarItens_RetornaQuantidade()
{
    // Arrange
    var pedido = new Pedido();

    // Act
    int itens = pedido.ListarItens();

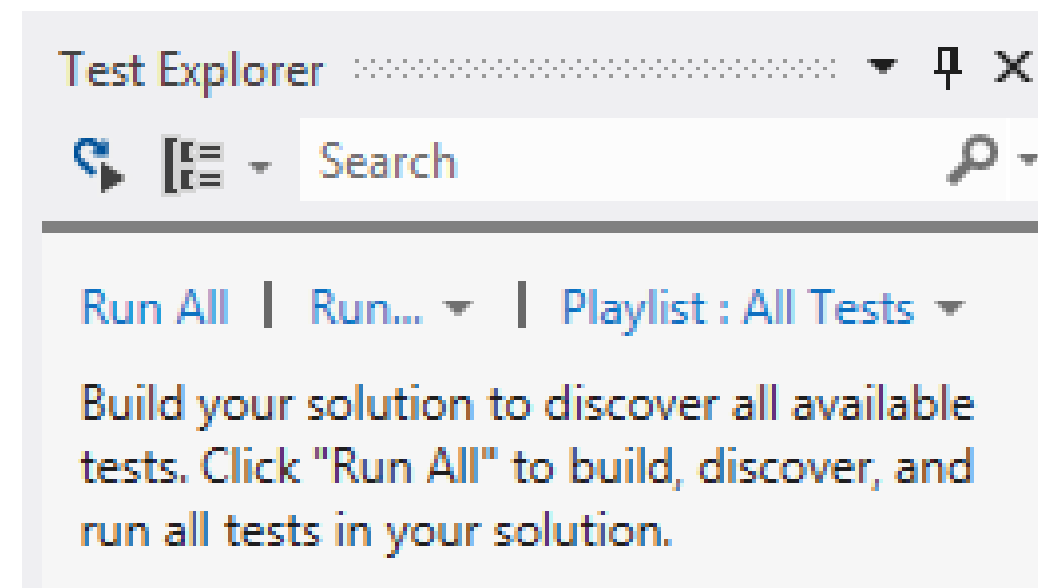
    // Assert
    Assert.AreEqual(0, itens);
}
```

Criando um teste unitário utilizando TDD em C# com MSTest

20) Nosso teste já está pronto para ser executado, e para isso, iremos ativar a janela do Test Explorer. Clique sobre o menu Test > Windows > Test Explorer

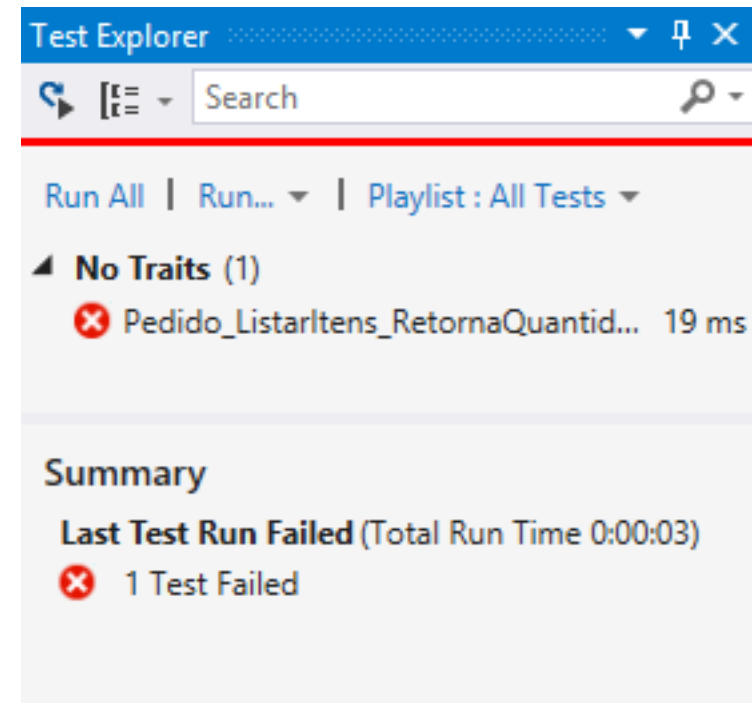


21) Vamos executar os testes, para isso, clicaremos no botão "Run All". Após isso, uma barra de progresso será exibida, demonstrando que os testes estão sendo executados

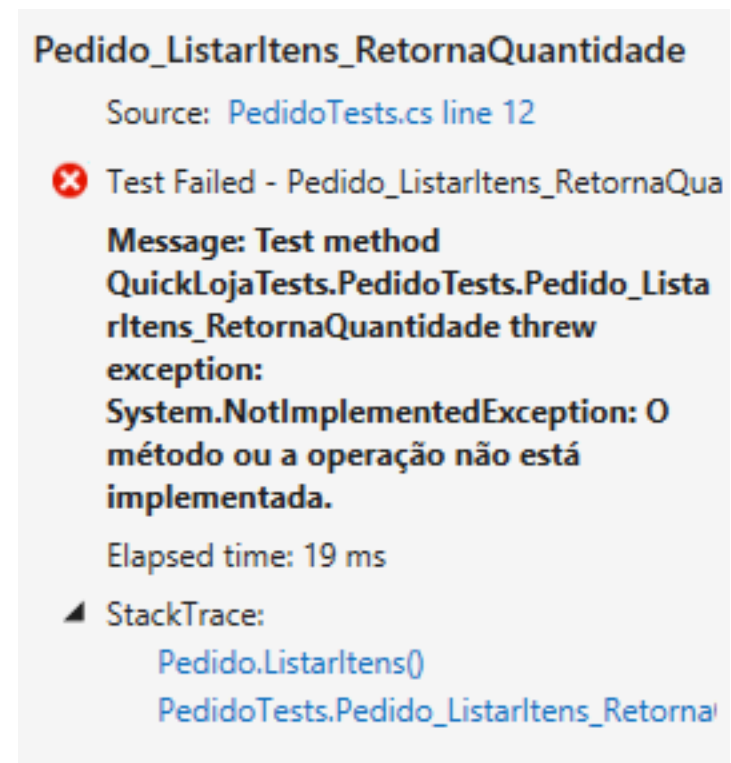


Criando um teste unitário utilizando TDD em C# com MSTest

22)O resultado da execução é apresentado logo após a execução dos testes



23)Como esperado, a primeira execução do nosso teste falhou, e clicamos sobre ele para conhecermos o motivo da falha. Vemos então que o teste falhou porque o método lançou a exceção "NotImplementedException", que quer dizer que nosso método não foi implementado

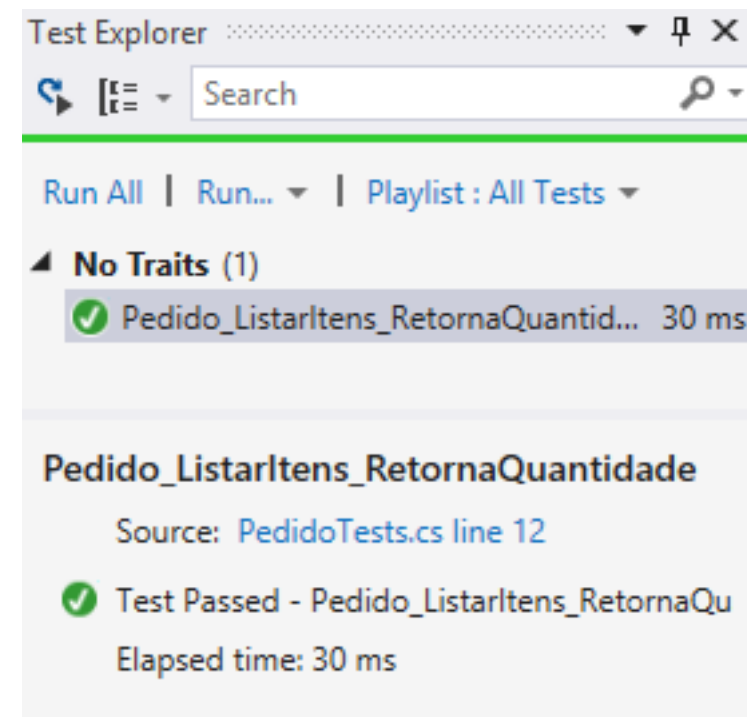


Criando um teste unitário utilizando TDD em C# com MSTest

24) Voltamos ao método e faremos com que o nosso teste falhe, para isso, iremos fazer com que ele retorne 0, que é o valor que o nosso Assert está aguardando

25) Execute novamente o teste, clicando em "Run All", e o resultado desta vez será positivo

```
public int ListarItens()
{
    return 0;
}
```



Criando um teste unitário utilizando TDD em C# com MSTest

26) Agora que o teste passou, a ideia é não mais fazer alterações no teste. O próximo passo será refatorar o código, pois hoje o valor retornado é fixo e não representa a quantidade real de itens contida no pedido. Criaremos uma propriedade, chamada "PedidoItens", na classe Pedido, que possa armazenar os itens que serão adicionados ao mesmo

```
private List<string> pedidoItens = new List<string>();

public int ListarItens()
{
    return pedidoItens.Count;
}
```

Criando um teste unitário utilizando TDD em C# com MSTest

27) Novamente, iremos executar todos os testes clicando em "Run All", e o teste deve passar. É muito importante que o teste, após passar pela primeira vez, não deve mais ser alterado. Isto nos ajuda a evitar a geração de falhas em códigos que antes funcionavam corretamente. Vamos adicionar um novo teste, para uma nova implementação, a de adicionar itens ao pedido. Crie um método de teste novo, chamado "Pedido_AdicionarProdutoAoPedido_QuantidadeIncrementada" que adiciona um item da classe Produto ao Pedido

```
[TestMethod]
public void Pedido_AdicionarProdutoAoPedido_QuantidadeIncrementada()
{
    // Arrange

    // Act

    // Assert
}
```

Criando um teste unitário utilizando TDD em C# com MSTest

28) Como ponto inicial iremos adicionar, na área "Arrange", a instanciação da classe Pedido, pois utilizaremos ela no nosso teste

29) Neste momento precisamos imaginar como este método será implementado, e a melhor forma neste caso é imagina que uma classe chamada Produto será implementada, e teremos nela, inicialmente, as seguintes propriedades: ProdutoId, ProdutoNome, ProdutoValor e ProdutoEstoque. Ao declarar isso, o Visual Studio percebe a intenção de criação da classe e nos fornece o menu para refatoração, e nele, selecionaremos a opção "Generate new type..." para criar a classe Produto

```
[TestMethod]
public void Pedido_AgregarProdutoAoPedido_QuantidadeIncrementada()
{
    // Arrange
    var pedido = new Pedido();

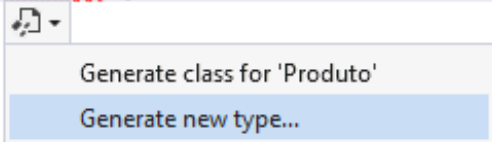
    // Act

    // Assert
}
```

```
[TestMethod]
public void Pedido_AgregarProdutoAoPedido_QuantidadeIncrementada()
{
    // Arrange
    var pedido = new Pedido();
    var produto = new Produto { ProdutoId = 1, ProdutoNome = "Boné", ProdutoValor = 39.90, ProdutoEstoque = 10 };

    // Act

    // Assert
}
```



Criando um teste unitário utilizando TDD em C# com MSTest

30)Na janela selecionaremos o projeto "QuickLoja"

Generate New Type ? X

Type Details:

Access: Kind: Name:

public class Produto

Location:

Project:

QuickLoja

File Name:

☒ Create new file

Produto.cs

☐ Add to existing file

Pedido.cs

OK Cancel

Criando um teste unitário utilizando TDD em C# com MSTest

31)A classe Produto foi criada conforme esperado, mas agora as propriedades estão sendo grifadas pelo Visual Studio, isto porque elas não foram criadas dentro da classe Produto. Para que estas propriedades sejam criadas iremos clicar sobre o menu de refatoração, em cada uma das propriedades

32)A classe Produto agora possui as propriedades que foram declaradas

```
// Arrange
var pedido = new Pedido();
var produto = new Produto { ProdutoId = 1, ProdutoNome = "Camiseta", ProdutoValor = 35.90, ProdutoEstoque = 10 };
```

```
// Act
```

```
// Assert
```



Generate property stub for 'ProdutoId' in 'QuickLoja.Produto'

Generate field stub for 'ProdutoId' in 'QuickLoja.Produto'

```
public class Produto
{
    public int ProdutoId { get; set; }

    public string ProdutoNome { get; set; }

    public double ProdutoValor { get; set; }

    public int ProdutoEstoque { get; set; }
}
```

Criando um teste unitário utilizando TDD em C# com MSTest

33)Iremos criar agora o método que adicionar no teste o método AdicionarItem da classe Pedido, que passará por parâmetro o objeto Produto e retornará a lista de itens contida no pedido

34)Vamos adicionar o NameSpacing System.Collections.Generic para podermos usar o recurso List

35)Fazendo isso o Visual Studio conseguirá identificar a classe List<Produto>

36)O Visual Studio verifica que o método AdicionarItem ainda não existe na classe Pedido, por isso iremos acionar o menu de refatoração e criar o método

```
// Act
List<Produto> produtos = pedido.AdicionarItem(produto);
```

```
using System;
using System.Collections.Generic;
using Microsoft.VisualStudio.TestTools.UnitTesting;
```

```
// Act
List<Produto> produtos = pedido.AdicionarItem(produto);
```

```
// Act
List<Produto> produtos = pedido.AdicionarItem(produto);
```

```
// Assert
```



Generate method stub for 'AdicionarItem' in 'QuickLoja.Pedido'

Criando um teste unitário utilizando TDD em C# com MSTest

37)Vemos que o método foi criado na classe Pedido, e uma exception foi adicionada a ele para mostrar que o mesmo ainda não foi implementado

```
public List<Produto> AdicionarItem(Produto produto)
{
    throw new NotImplementedException();
}
```

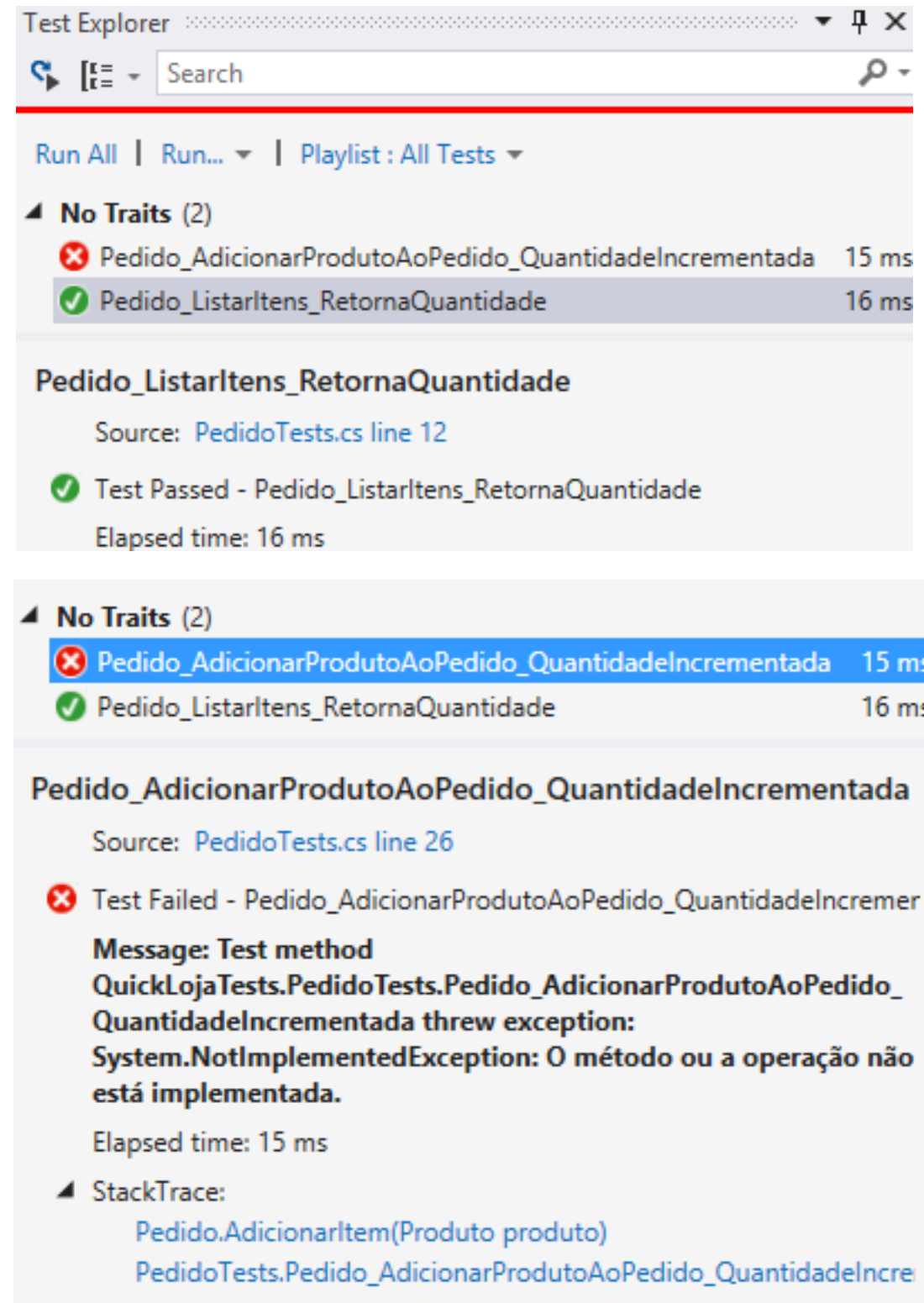
38)Vamos adicionar agora o CollectionAssert, classe que possui métodos usados para fazer verificações em coleções, e este irá verificar se o Produto foi adicionado à lista, faremos isso verificando se o Produto está presente dentro da lista de produtos que foi retornada do método AdicionarItem

```
// Assert
CollectionAssert.Contains(produtos, produto);
```

Criando um teste unitário utilizando TDD em C# com MSTest

39)Iremos executar todos os teste, para isso, clique no botão "Run All" e o resultado dos testes será

40)Vemos que o teste de inserção de produtos ao pedido falhou, e para ver o motivo da falha iremos clicar sobre o teste que falhou



Criando um teste unitário utilizando TDD em C# com MSTest

41) A falha foi causada pelo método `AdicionarItem`, que ainda não foi implementado, por isso iremos implementá-lo, adicionando o produto à lista `pedidoItens` contida na classe `Pedido` e retornando a mesma. Vemos que ao fazer isso o Visual Studio apresenta uma falha, dizendo que a lista não é uma lista de `Produtos`

42) Vamos alterar a lista `pedidoItens` para que se torne uma lista de `Produtos` ao invés de uma lista de strings

```
private List<string> pedidoItens = new List<string>();

public int ListarItens()
{
    return pedidoItens.Count;
}
```

```
public List<Produto> AdicionarItem(Produto produto)
{
    pedidoItens.Add(produto);
}
```

`void List<string>.Add(string item)`
Adds an object to the end of the `System.Collections.Generic.List<T>`.

Error:

The best overloaded method match for '`System.Collections.Generic.List<string>.Add(string)`' has some invalid arguments

```
public class Pedido
{
    private List<Produto> pedidoItens = new List<Produto>();
}
```


Criando um teste unitário utilizando TDD em C# com MSTest

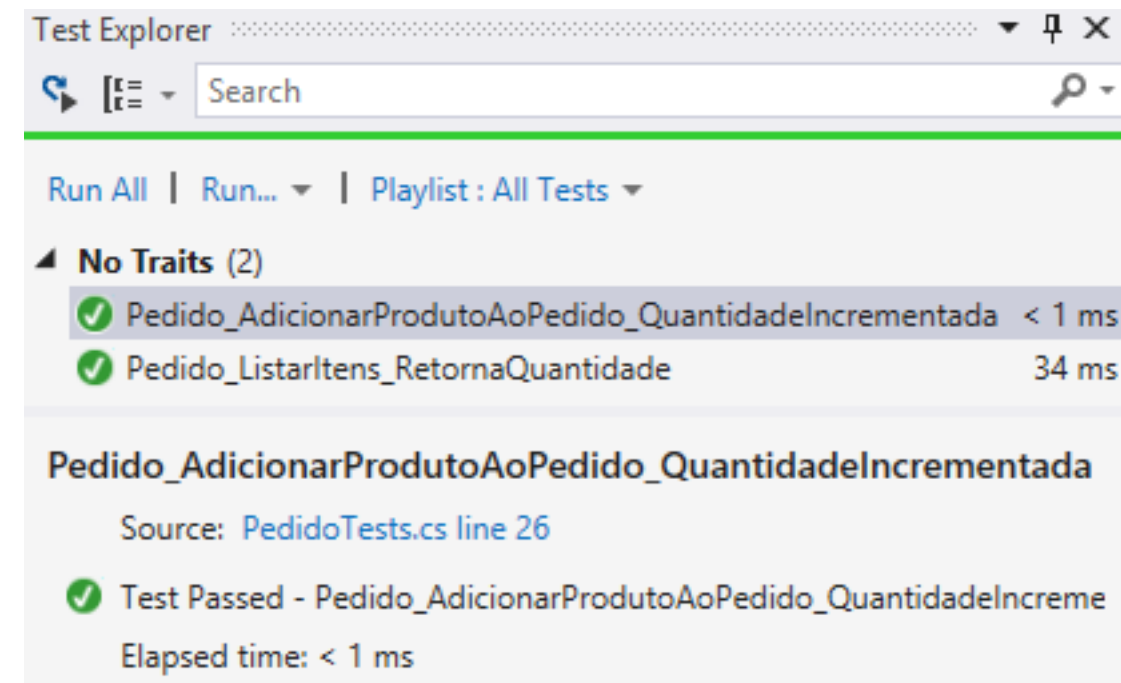
43) Agora vemos que a classe Pedido não apresenta mais nenhuma inconsistência

```
public class Pedido
{
    private List<Produto> pedidoItens = new List<Produto>();

    public int ListarItens()
    {
        return pedidoItens.Count;
    }

    public List<Produto> AdicionarItem(Produto produto)
    {
        pedidoItens.Add(produto);
        return pedidoItens;
    }
}
```

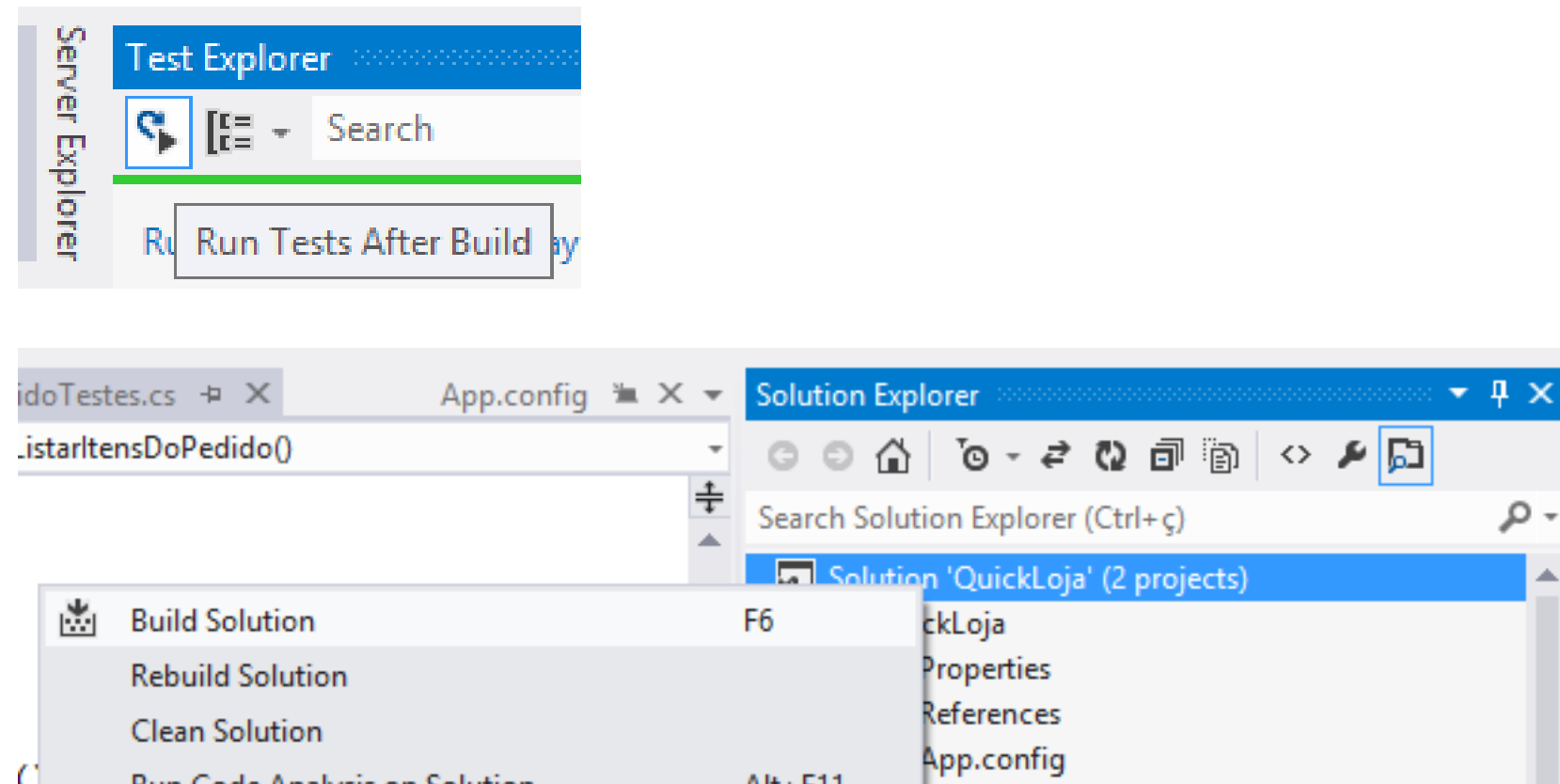
44) Iremos rodar todos os testes novamente, clicando em "Run All" e então nossos testes irão passar e nosso suíte de testes então estará pronta para ser executada a qualquer momento



Criando um teste unitário utilizando TDD em C# com MSTest

45)Clicando sobre "Run Tests After Build" ativaremos o recurso de executar os testes sempre após a execução de Builds

46)Clique com o botão direito sobre a solução, na janela "Solution Explorer", e selecione "Build Solution", ou então pressione "F6" para fazer o Build da solução "QuickLoja" e executar os testes automaticamente





03

Organizações e Convenções

— Organizações e Convenções

Por Método

Nesta abordagem, criamos uma classe de teste para cada um dos métodos da aplicação que serão testados. Os nomes destas classes são finalizados por "Should", ou, em português "Deve".

Cada método de teste prova que uma funcionalidade, do método que está sendo testado, faz o que deveria fazer. O nome dos métodos faz referência às ações e resultados esperados pelo teste, por exemplo:

Classe MétodoParaSomarDoisValoresDeve

```
{  
    Teste Fornecer2E2ERetornar4()  
    { // Conteúdo do Teste }  
}
```

— Organizações e Convenções

Por Classe

Cada classe da aplicação possui uma classe de teste, e dentro desta classe temos os métodos de teste que testam todos os métodos desta classe.

A classe de testes, neste caso recebe o nome da classe da aplicação, seguida da palavra Testes ou Tests, por exemplo:

Classe MóduloDaAplicaçãoTestes

```
{  
    Teste CalcularAlgoFornecendo2E2ERetorne4()  
    {  
        // Conteúdo do Teste  
    }  
}
```


— Organizações e Convenções

LEMBRE-SE:

Não engesse seus testes em função das convenções. O objetivo primordial dos testes é garantir que o código funciona conforme sua especificação. Logo, os nomes dos testes também podem identificar qual comportamento está sendo testado!

ATENÇÃO:

- Ao criar novos projetos é obrigatório seguir o padrão `RM.SiglaSistema.XXX.TesteUnitario`, caso contrário a IC não irá executar os testes.
- Coloque `LayerSideKind.TestesUnitarios` no `AssemblyInfo.cs`, caso contrário a IC não irá executar os testes.

04

Mocks e Stubs: preechendo
as dependências das
classes testadas

— Mocks e Stubs

Nos testes, haverá a necessidade de se instanciar as dependências da classe testada, mas sem dispor de infraestrutura para tal (com instâncias **fake**)

Para isso, podemos utilizar em princípio **Mocks** e **Stubs**

Stubs basicamente são fakes que sempre respondem da mesma forma, enquanto os **Mocks** devem seguir uma ordem específica de respostas

— Mocks e Stubs

Na biblioteca RM, já estão disponíveis 2 assemblies de Mocking Frameworks que podem ser utilizados de acordo com a preferência do Desenvolvedor:

- MOQ
- Rhino Mocks

Moq tem a API mais simples e intuitiva, mas tem limitações para parâmetros por referência e parâmetros OUT. **Rhino Mocks** não tem essas limitações, é mais poderosa mas menos amigável.



05

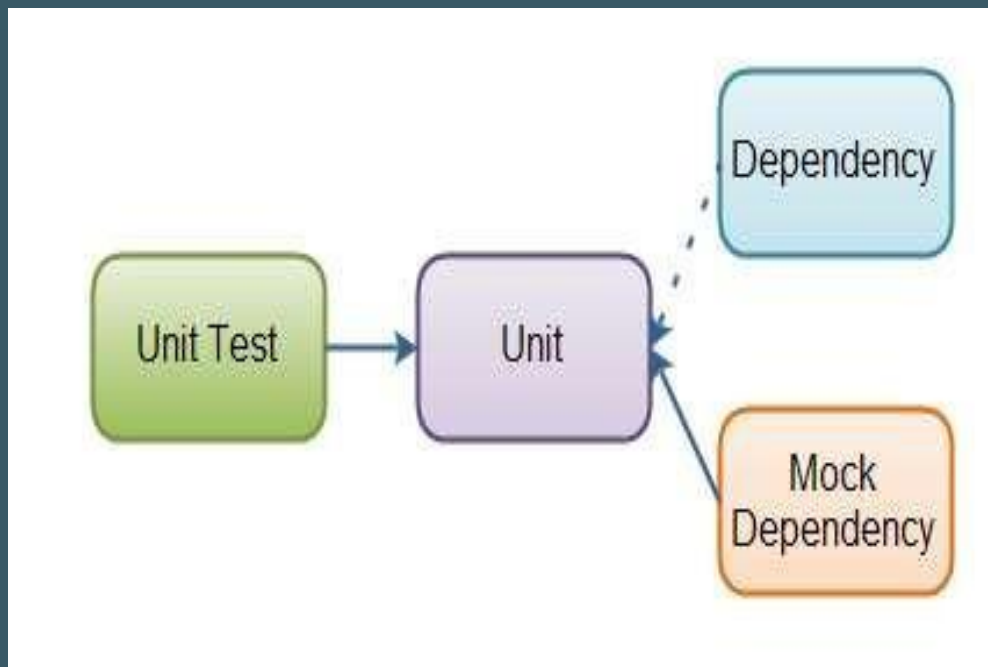
Exercício Prático: Refatorando código



06

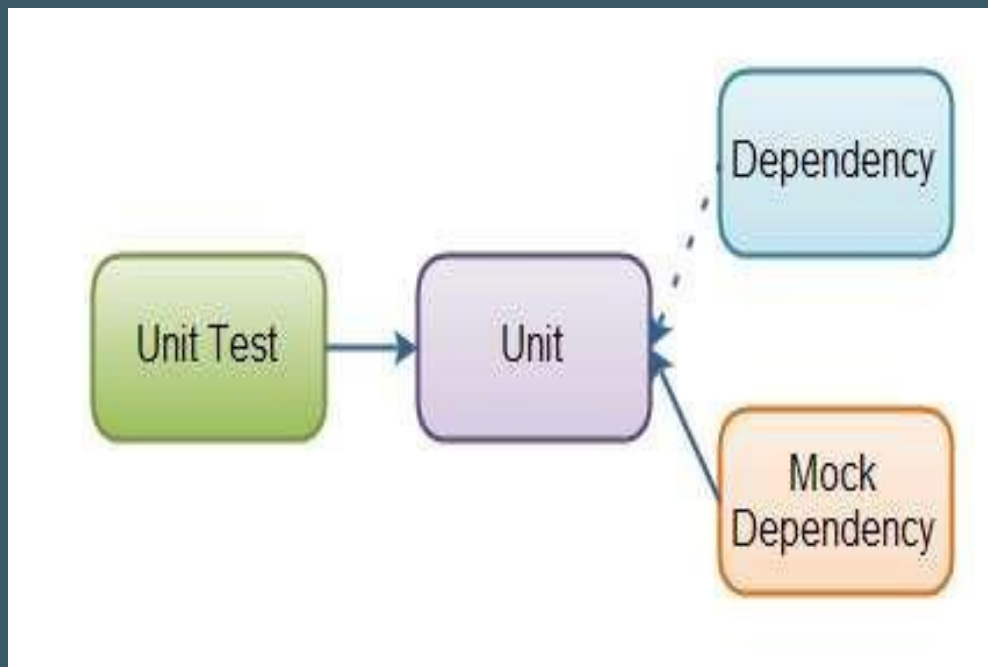
Injeção de Dependência com Castle Windsor

— Injeção de Dependência com Castle Windsor



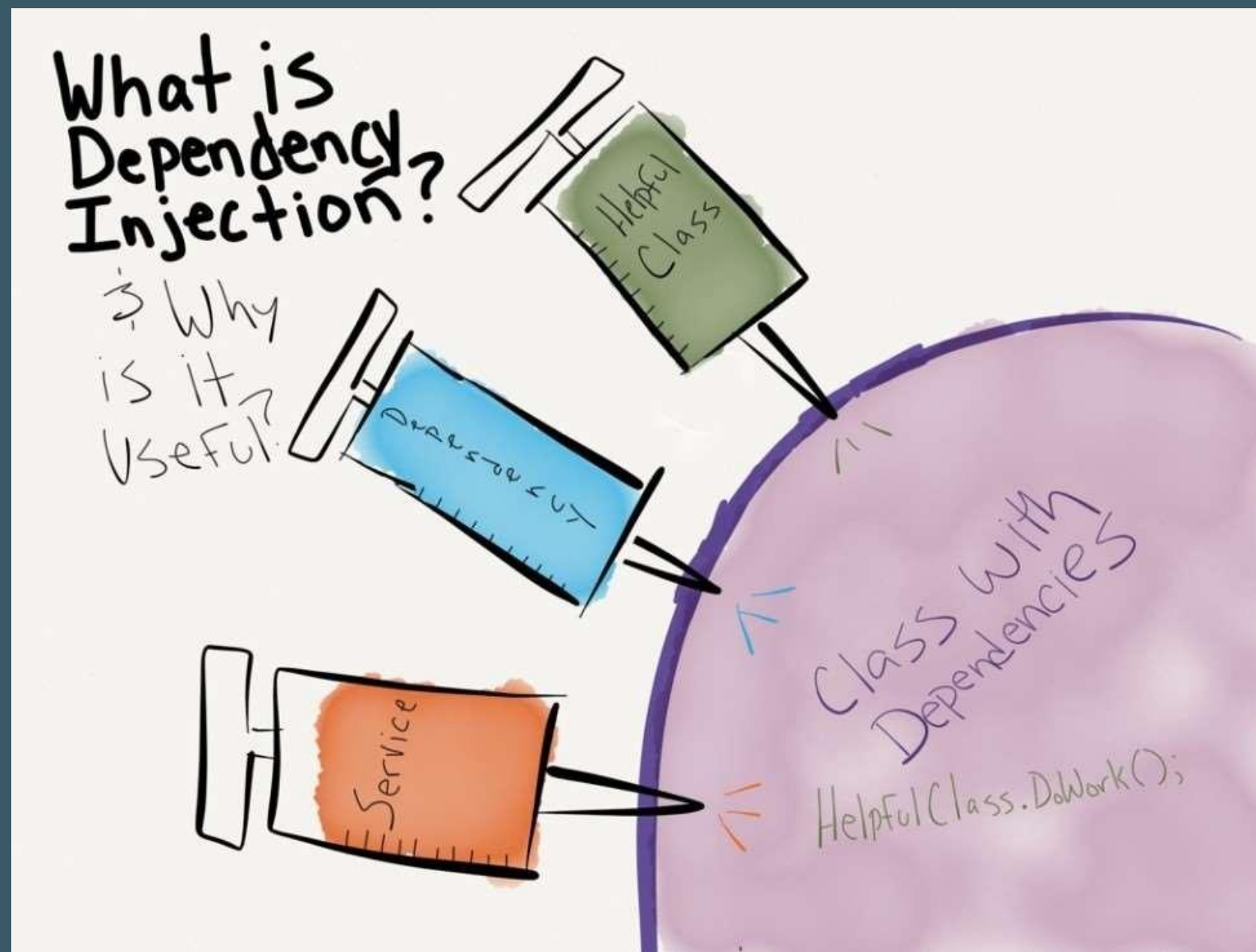
- D.I. e I.O.C estão intimamente relacionados;
- Existem diversos containers opensource amplamente utilizados;
- Castle Windsor é um dos mais populares e será o utilizado no treinamento;

— Injeção de Dependência com Castle Windsor

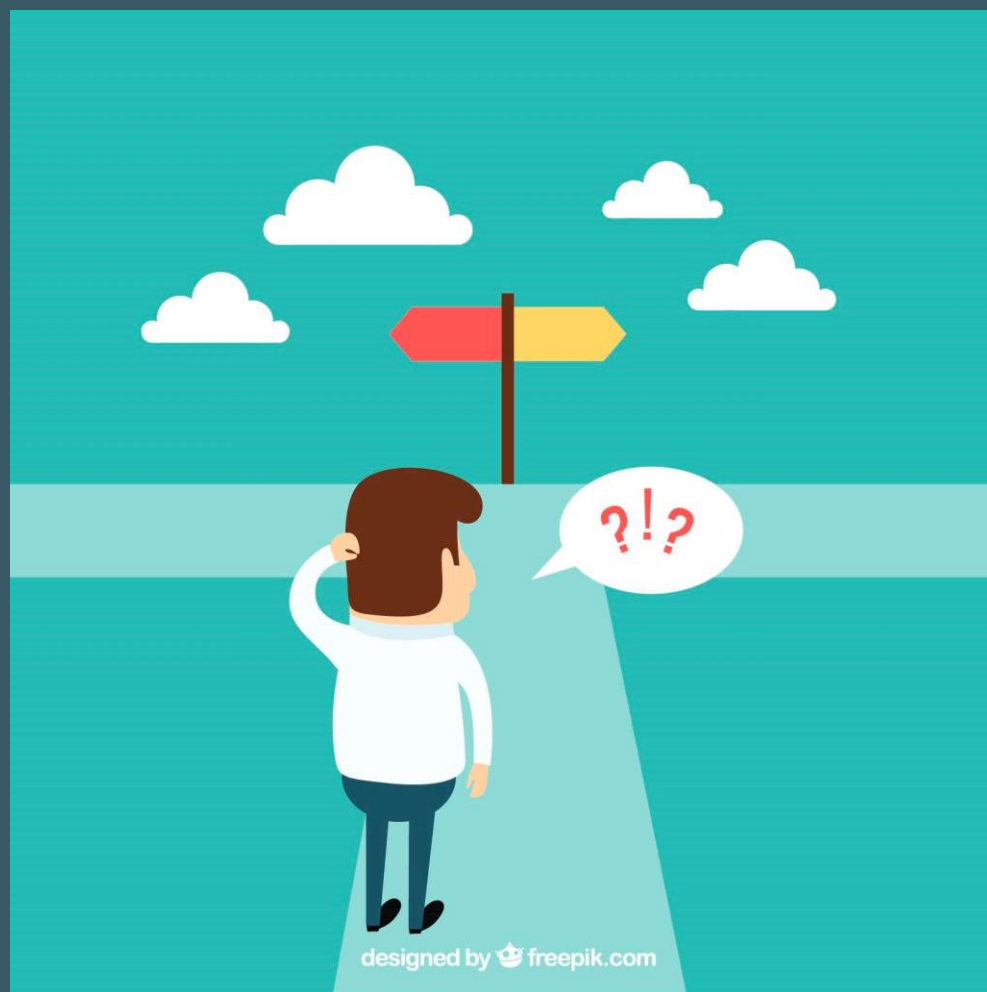


- Reduz significativamente o acoplamento entre componentes e suas dependências;
- Incentiva o desenvolvedor a utilizar Composição e Herança, resultando em implementações mais flexíveis;
- Permite a alteração de implementações via config ou em runtime;
- Suporta arquiteturas Ncamadas;

— Injeção de Dependência com Castle Windsor



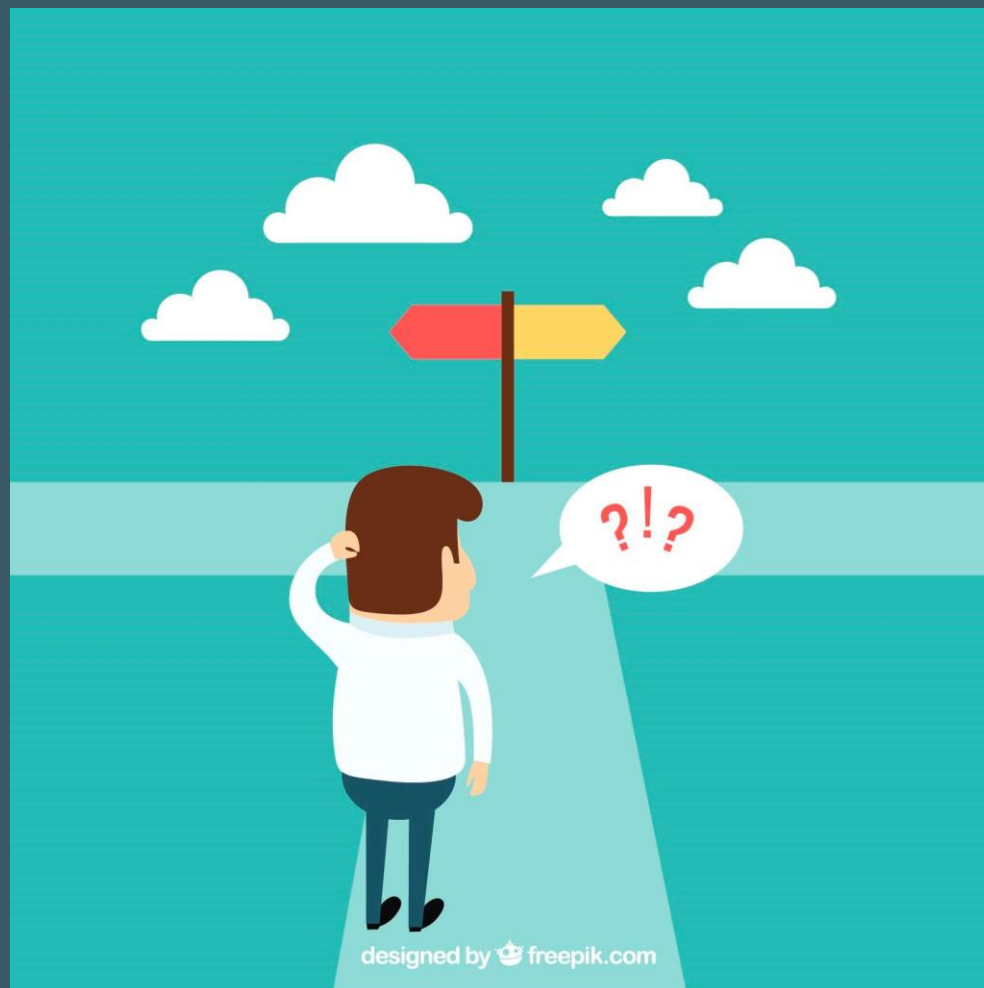
— Injeção de Dependência com Castle Windsor



Fazendo uma analogia ao `RMSBroker.CreateServer<T>`:

- Quais as semelhanças?
- Quais as diferenças?
- Em que aspecto um pode ser superior ao outro?
- O que dizer em relação aos métodos `CreateModule`, `CreateObject` e `CreateFacade`?

— Injeção de Dependência com Castle Windsor



Após ajustar seu exercício para utilizar o Container de D.I., reflita sobre a seguinte questão:

Por que no exemplo do instrutor foi criada uma “Casca” para o container de Injeção de Dependência?

— Material para estudo gratuito sobre TDD, Mocking e Padrões de Projeto

<https://www.alura.com.br/curso-online-test-driven-development-tdd-dotnet>

<https://www.alura.com.br/curso-online-mocks-em-net>

* <https://www.pluralsight.com/courses/csharp-unit-testing-enterprise-applications>

* <https://www.pluralsight.com/courses/full-stack-dot-net-developer-architecture-testing>

[https://en.wikipedia.org/wiki/GRASP_\(object-oriented_design\)](https://en.wikipedia.org/wiki/GRASP_(object-oriented_design))



OBRIGADO



Renato Rocha Silva

Engenharia de Software

Renato Rocha

Christiano Coutinho

Autor: christiano.coutinho@totvs.com.br

Revisado: renato.silva@totvs.com.br



Tecnologia + Conhecimento são nosso DNA
O sucesso do cliente é o nosso sucesso
— Valorizamos gente boa que é boa gente

 totvs.com

 company/totvs

 @totvs

 fluig.com

#SOMOSTOTVERS