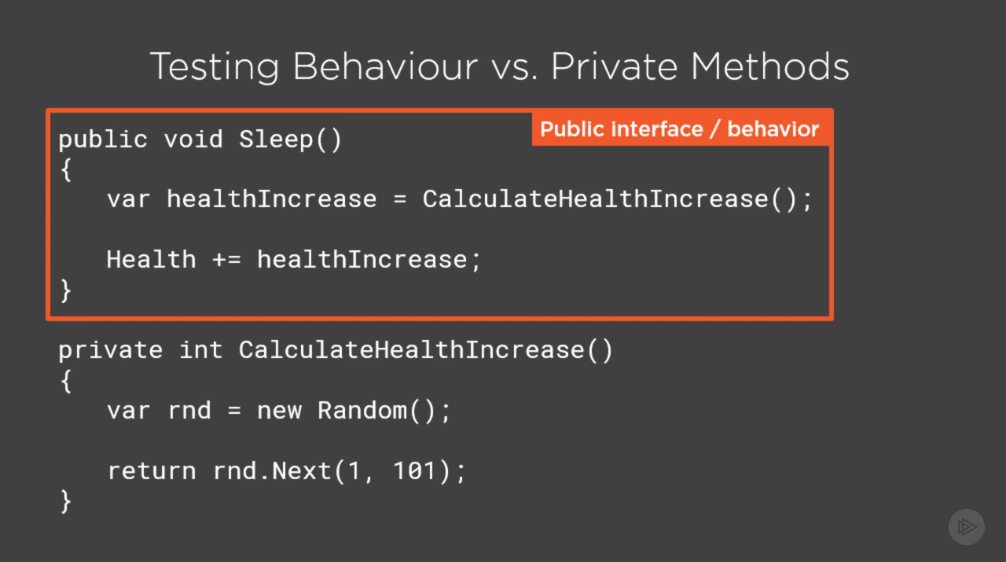
**[Testing .NET Code with xUnit.net: Getting Started](https://app.pluralsight.com/courses/2acc5d5d-1030-4373-8bb0-4ddf1721ec8d/table-of-contents" \t "_blank)**

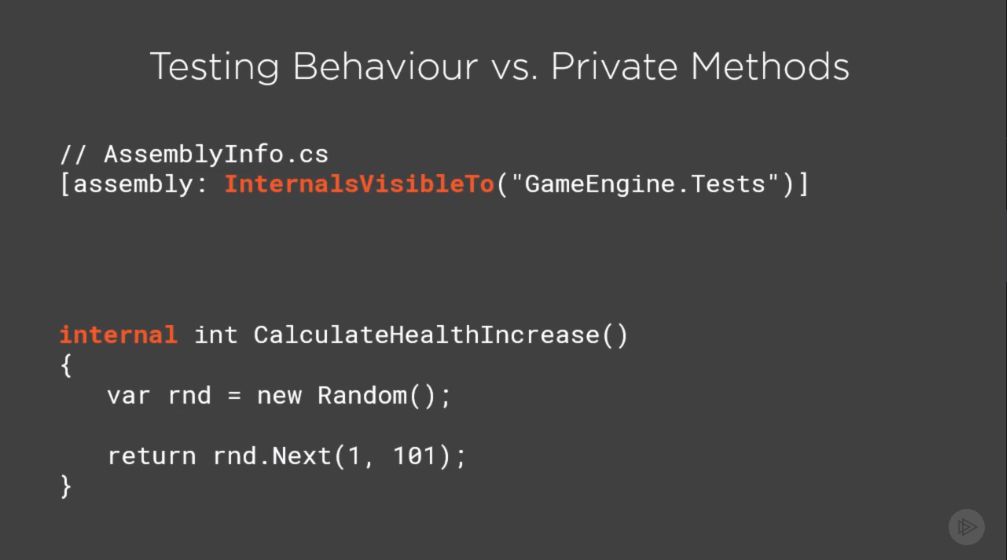
**Modulo II – Getting Started**

**Testing Behaviour vs. Private Methods**

O chapter aborda a importância de testar o comportamento geral da classe em detrimento da implementação interna, ou seja, os métodos privados.

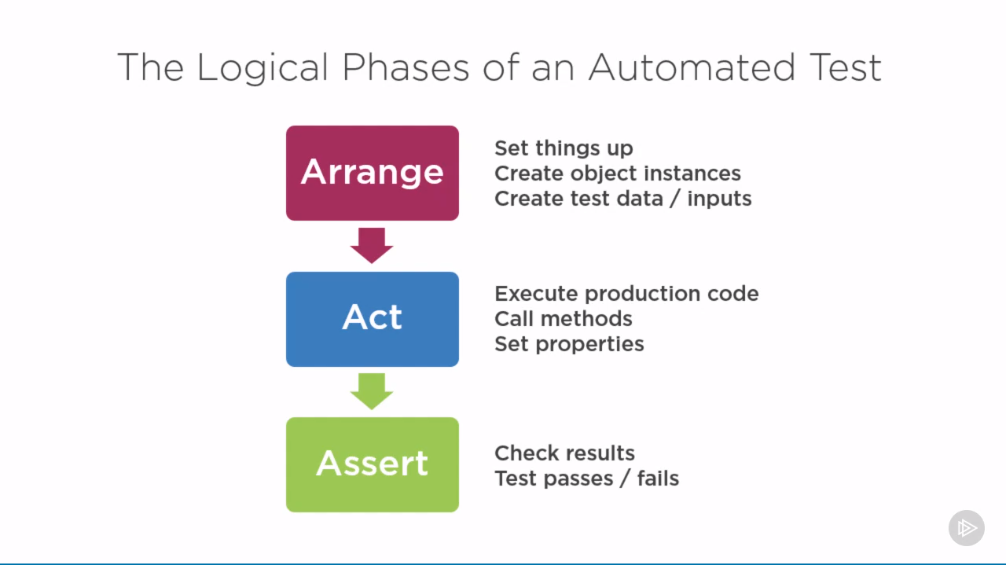


No exemplo acima, o método público *Sleep()* pertence a uma classe de personagem de um jogo RPG. O mesmo aplica um método de suspensão que regenera a saúde do personagem enquanto este dorme, invocando o método privado *CalculateHealthIncrease()*, que incrementa saúde de maneira randômica. Por este último método ser privado, não pode ser acessado pelos testes. Nesse caso, o teste se concentrará no comportamento do método público ou da classe como um todo, e deve retornar se houve ou não incremento de saúde.



Caso exista a necessidade de testar componentes privados da classe em detrimento de seu comportamento, realizar uma mudança no modificador de acesso para público irá de encontro ao conceito de encapsulamento da orientação a objetos. Como alternativa, podemos mudar o modificador de acesso para *internal* e configurar o arquivo *AssemblyInfo.cs* para que as classes e métodos de teste dentro do mesmo ­*Assemby* possam acessar estes componentes. *Em resumo, é uma boa ideia testar a interface pública no comportamento da classe em detrimento de acoplar firmemente os testes aos detalhes (classes encapsuladas) da implementação, mantendo assim o conceito de desacoplamento.*

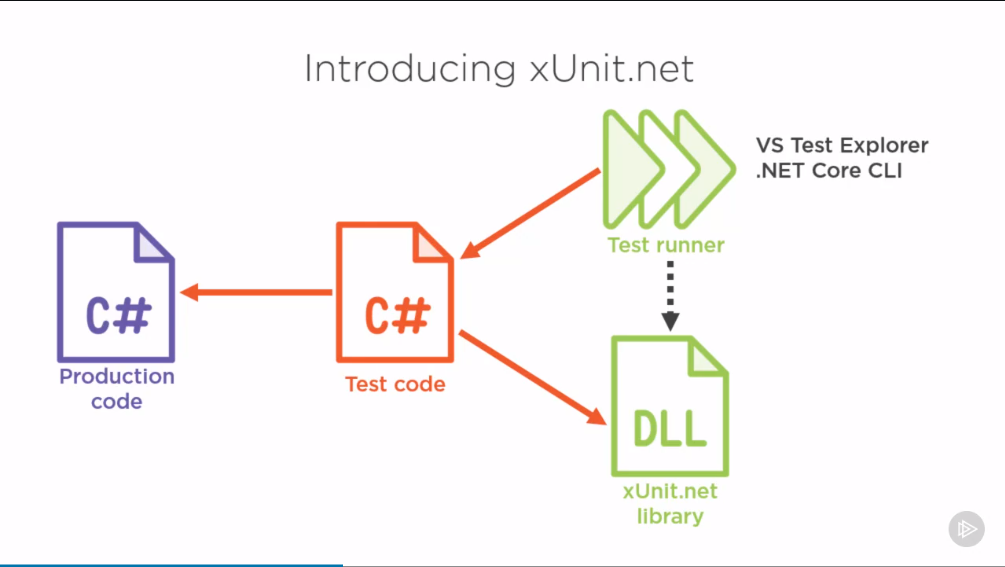
**The Logical Phases of na Automated Test**



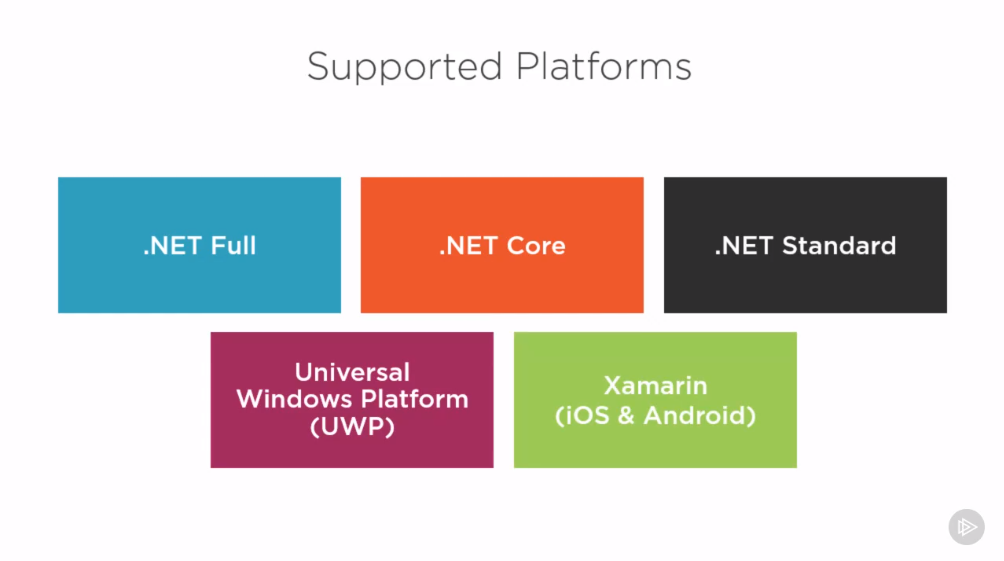
Este capítulo estabelece 3 fases inerentes aos testes automatizados:

1. Arrange – Set de parâmetros para a criação de instâncias de objetos, dados e inputs;
2. Act – Execução de código de produção, chamada de métodos e configuração de propriedades;
3. Assert – Verificação dos resultados dos testes após os dois estágios anteriores.

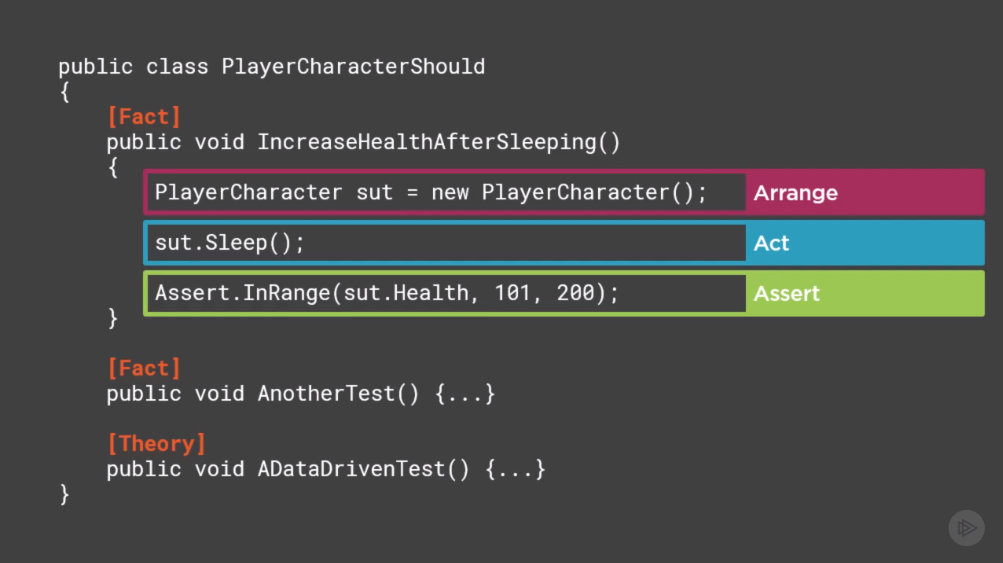
**Introducing xUnit.net**



Este capítulo introduz ao framework xUnit.net. Expõe as ferramentas disponíveis nesse frame seguindo o esquema acima: Classe de Teste fazendo referência às classes em produção. A classe teste utilizam as bibliotecas do xUnit.net. Essas bibliotecas, por sua vez são acessadas pelas ferramentas de implementação de testes das IDE’s, a exemplo de VS Test Explorer do Visual Studio ou da .NET Core CLI (Ferramenta de Terminal).



Acima a lista das plataformas suportadas pelo xUnit.net. (https://xinut.github.io).



O exemplo acima mostra uma classe de testes criada para testar o método de incremento de saúde do personagem de RPG anteriormente mencionado. Por convenção, as classes de testes devem possuir o prefixo *­Should* para que possam ser melhor identificadas pelas ferramentas de testes, melhorando sua legibilidade. O método que testará o incremento de saúde será o *IncreaseHealthAfterSleeping()*, que como o nome sugere, testa se o incremento de saúde do personagem foi devidamente executado. O atributo **[Fact]** indica ao xUnit que esse teste deve ser executado.

Seguindo as três fases dos métodos automatizados temos a seguinte lógica de execução do teste:

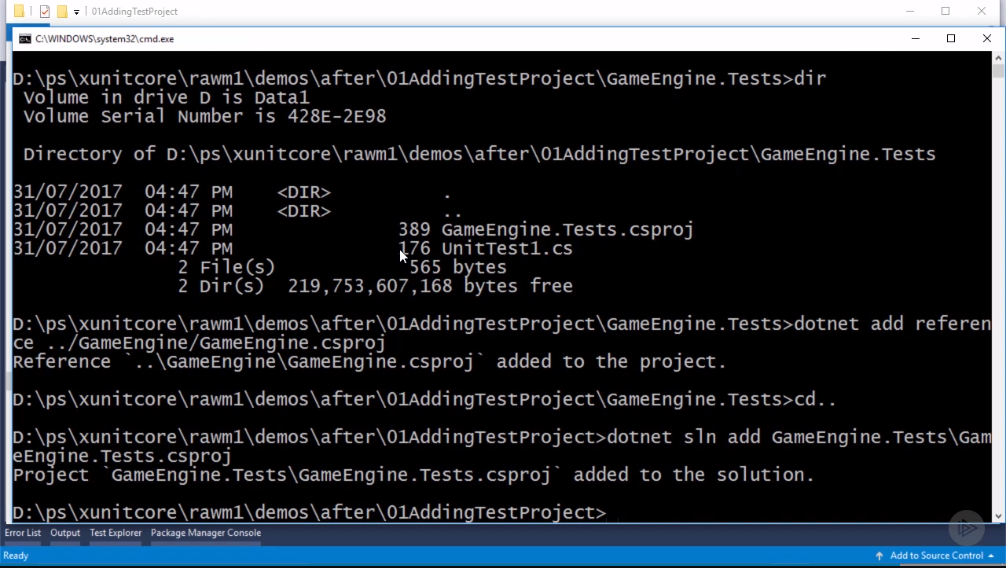
**Arrange** → Cria a instância do código em produção que desejamos testar – objeto da classe do personagem (**sut**);

**Act** → Executa a chamada do método de produção (**Sleep()**);

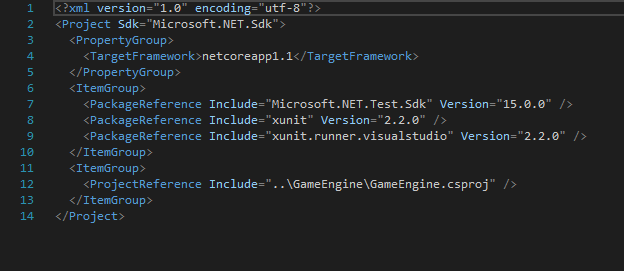
**Assert** → Verifica o retorno dos dados gerados pela fase anterior e finaliza o teste.

O atributo [**Theory]** indica ao xUnit a criação de testes orientados a dados (módulo a frente).

**Create a Testing Project**



Neste capítulo é criado uma *sln* (solution) de teste com referência a solution para a criação do personagem *GameEngine*. O nome da solução é ­GameEngine.Test. É feita a criação pelo VS e via CLI, com ambas as técnicas fazendo a devida referenciação entre o projeto de produção e o projeto de testes.



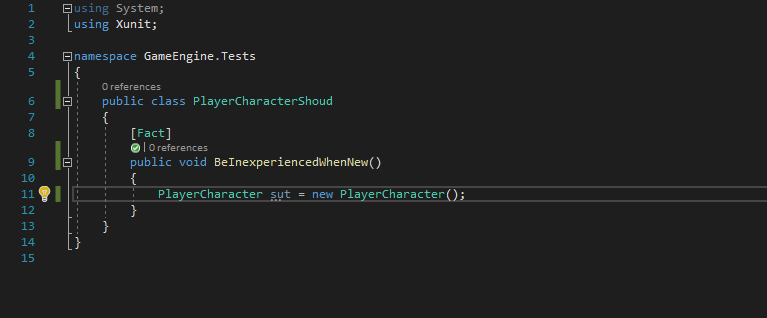
A imagem acima mostra a referenciação entre as soluções para o projeto de testes (GameEngine.Tests.csproj).

**Create a Testing Project**

Neste capítulo é criado um novo projeto de testes utilizando o .NET Framework, com o nome de *GameEngineFF.Tests*. Dessa vez, as bibliotecas do xUnit são instaladas pelo gerenciador de pacotes *Nugget*. De resto, o processo é muito similar ao capítulo anterior.

**Start to Create the First Test**

Nesse capitulo é criada a classe de teste com o sufixo *Should* (PlayerCharacterShould). É explicado que o objeto instanciado da classe *PlayerCharacter*, do projeto de produção deverá, por convenção, receber a nomenclatura ***sut***que é um acrônimo para *sytem under test*. Pra conseguir realizar os testes, precisei atualizar os pacotes do xUnit no gerenciador de pacotes *NuGet*.



No exemplo acima, apenas a fase **Arrange** foi executada no teste, com a instanciação da classe do projeto de produção *PlayerCharacter()*.

**Modulo III - Determining Passing and Failing Tests with Asserts**

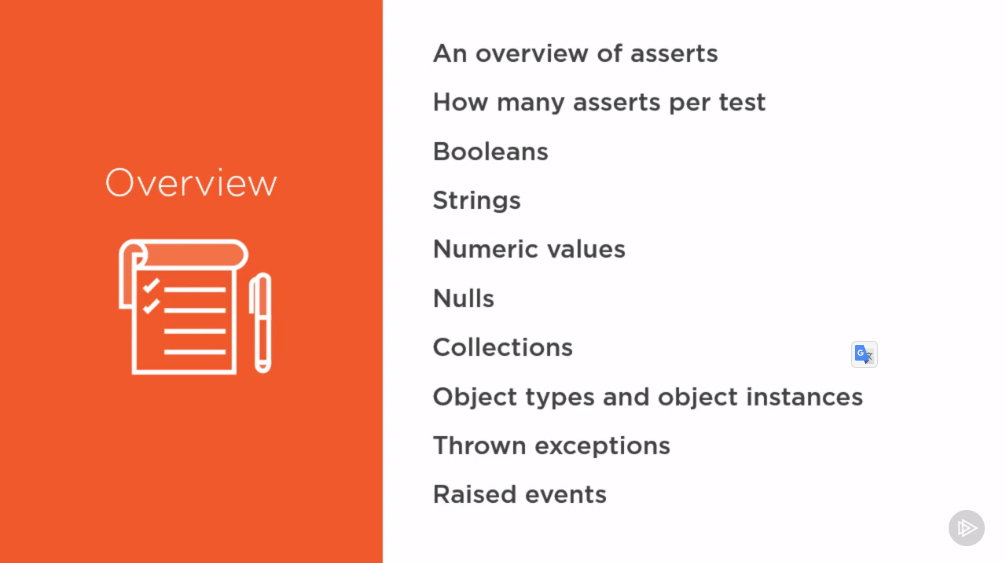
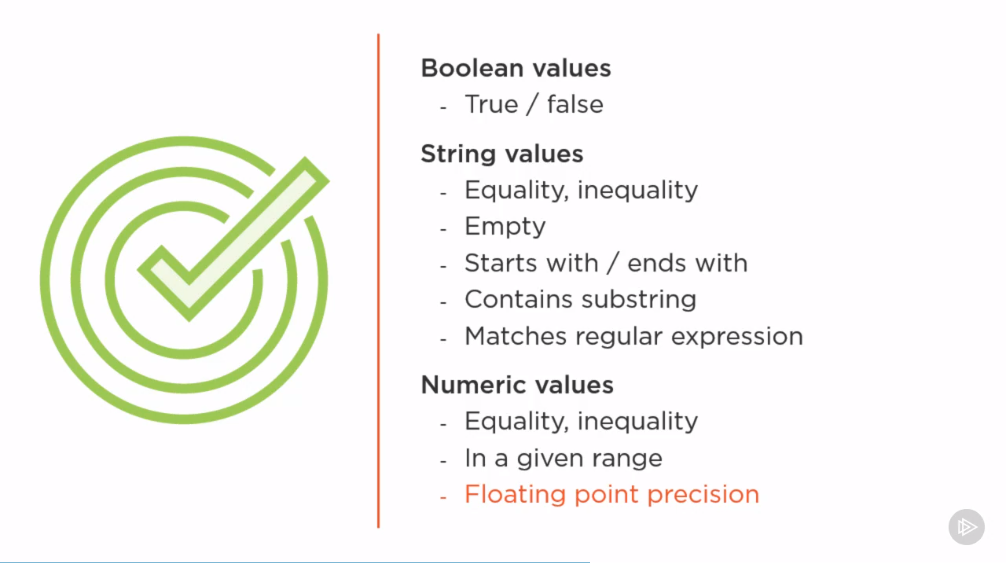


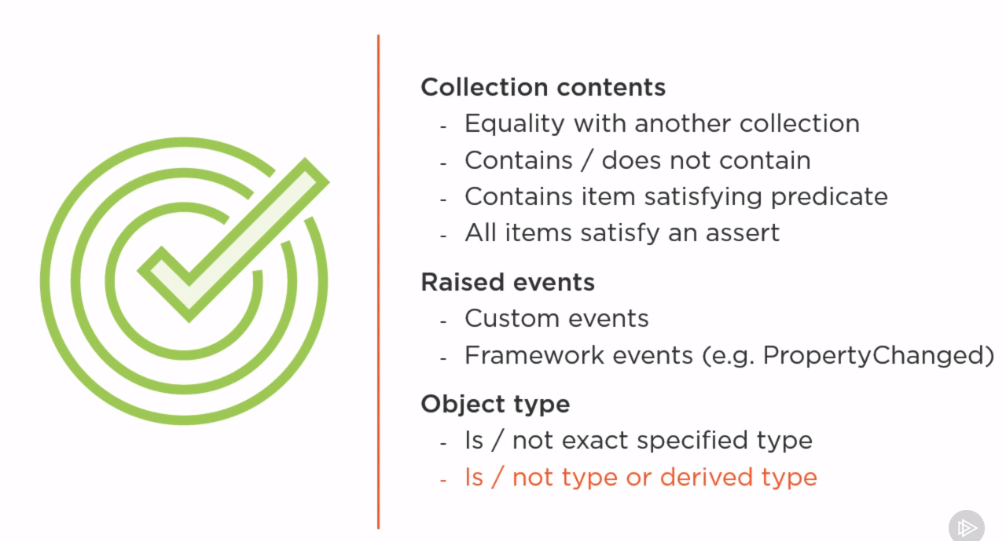
Figura 1 – Overview

**An Overview off Asserts**

Nesse capitulo é feita uma visão geral acerca da classe *Assert*, pertencente ao framework *xUnit*. Em resumo, *Asserts* verificam e avaliam o resultado dos testes com base no resultado retornado, estado do objeto final que foi criado e os eventos observados durante a execução. Um *Assert* pode falhar ou passar durante a execução. Se todas passarem, o teste será aprovado (e não aprovado caso ao menos uma delas não sejam satisfeitas.

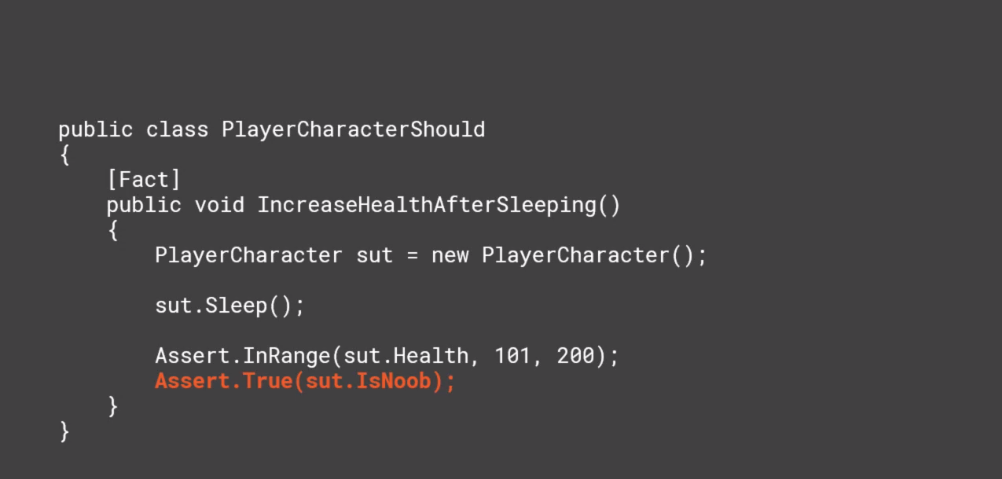
Abaixo, algumas verificações que os *Asserts* possuem suporte em sua biblioteca (Dentro do .NET):





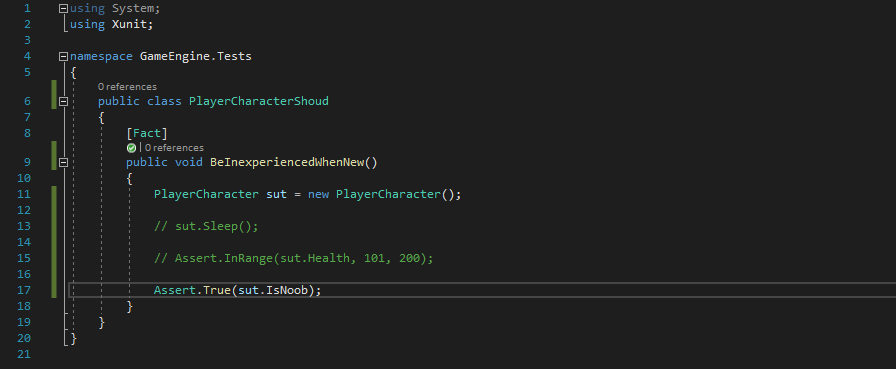
**How Mnay Asserts per Tests?**

Nesse capítulo é abordado que a quantidade de *Asserts* existentes devem, por convenção, testar um comportamento com base nas duas fases que o antecede (*Arrange* e *Act*). Não é uma boa prática utilizar ­*Asserts* que não testem o comportamento com base em dados e eventos. No exemplo abaixo, a chamada *Assert.True(sut.IsNoob)* não está testando nenhum comportamento do método *IsNoob*. É preferível, também que cada método de teste possua apenas um ­*Assert*.



**Adding na Assert to the First Test**

Neste capítulo, realizamos um teste para verificar um valor booleano, nesse caso *IsNoob* presente no construtor da classe de produção *PlayerCharacter.* *Assert.True* e *Assert.False* executam essa verificação.

**

**Making Assertions Against String Values**

Nesse episódio, foram abordados tipos de *Asserts* para verificação de *strings*. O testes de comportamento continuam a referir-se a classe *PlayerCharacter()*.

1. *Assert.Equal –* Compara duas strings e verifica se são equivalentes. No exemplo, é verificado se o valor atribuído à propriedade FullName (FirstName + LastName) corresponde ao valor passado. Obs: Os espaços no texto contam como valor.
2. *Assert.StartsWith* – Compara duas strings, verificando se a encontrada começa com os valores da string esperada.
3. *Assert.EndsWith* - Compara duas strings, verificando se a encontrada termina com os valores da string esperada.
4. *ignoreCase*: true (false) – Associado ao ­*Assert.Equal*, verifica se duas strings são equivalentes e ignora letras maiúsculas e minúsculas. Exemplo: Assert.Equal("Sarah Smith", sut.FullName, ignoreCase: true);
5. *Assert.Contains* – Verifica se a string encontrada contém uma substring esperada.
6. *Assert*.*Matches* – Verificar se a string encontrada segue um padrão de formatação da string esperada utilizando ­*Expressões Regulares.* No exemplo do capítulo, o teste verificou se a string encontrada correspondia ao *case* atribuído na expressão esperada.

**Asserting on Numeric Values**

Nesse episódio, foram abordados tipos de *Asserts* para verificação de valores numéricos. O testes de comportamento continuam a referir-se a classe *PlayerCharacter()*.

1. *Assert.NotEqual – Equal* – Verifica se dois valores numéricos são (ou não) equivalentes. No exemplo é comparado um inteiro esperado com o inteiro encontrado do retorno do método público *Health()* em *PlayerCharacter()*.
2. *Assert.InRange* – Verfica se retorno ou variável está entre um *range*.

**Asserting on Floating Point Value**

Nesse episódio, realizamos a verificação de um valor com ponto flutuante (*double*). Foi adicionado na declaração do *­Assert.Equal*, mais um atributo para conferir precisão ao retorno:

Assert.Equal(166.667, sut.TotalSpecialAttackPower, **3**);

**Asserting Null Values**

*Assert.Null – NotNull* – Verifica se o valor é (ou não) nulo.

**Asserting with Collectios**

Nesse episódio foram utilizados os métodos ­*Assert.Contains* e *Assert.DoNotContain* para verificar um item específico contido dentro da *collection* *Weapons()* em *PlayerCharacter()*. Também foi construído um teste pra verificar uma substring dentro dos itens da coleção com o *Assert.Contains*. Para comparar com a string encontrada, foi utilizada uma ­*Lambda Expression* para explicitar a substring esperada. O método ­*Contains* da expressão pertence a classe ­*Object()*(????)

[Fact]

public void HaveAtLeastOneKindOfSword()

{

PlayerCharacter sut = new PlayerCharacter();

Assert.Contains(sut.Weapons, **weapon => weapon.Contains("Sword")**);

}

O código abaixo verifica se a coleção encontrada corresponde a coleção esperada, passada no método de teste.

[Fact]

public void HaveAllExpectedWeapons()

{

PlayerCharacter sut = new PlayerCharacter();

var expectedWeapons = new[]

{

"Long Bow",

"Short Bow",

"Short Sword"

};

Assert.Equal(expectedWeapons, sut.Weapons);

}

O código em seguida verifica se a coleção encontrada contém itens nulos. O método booleano *IsNullOrWhiteSpace* (string) verifica se o objeto encontrado é nulo ou possue apenas espações em branco.

[Fact]

public void HaveNoEmptyDefaultWeapons()

{

PlayerCharacter sut = new PlayerCharacter();

Assert.All(sut.Weapons, weapon => Assert.False(string.IsNullOrWhiteSpace(weapon)));

}