Tests Unitaires avec XUnit

Pour créer un nouveau projet de test avec le framework de tests XUnit :

```
dotnet new xunit -o "MonProjet.Test"
```

Test unitaire

Cas de test

Pour créer un test unitaire, créez une classe dans ce projet de test. Une classe correspond à une collection de test, chaque méthode est un cas de test. Pour créer un cas de test, ajoutez une méthode avec l'attribut `[Fact]` :

```
public class MaCollectionDeTests {

  [Fact]
  public void MonCasDeTest(){
  
  }

}
```

Organisation et nommage

Les tests se déroulent en 3 étapes :

- "Etant donné" (*Given* en anglais) : mette en place la situation (souvent les données) requises par le cas de test
- "Lorsque" (*When* en anglais) : appel au code que l'on veut tester dans ce cas de test
- "Ainsi" (*Then* en anglais) : vérifications (assertions) sur les résultats du tests

Il convient donc, pour avoir des tests lisible des les diviser entre ces trois étapes, ainsi que de nommer les méthode correspondant au cas de tests de façon à expliciter le contenu de ces étapes. Le pattern suivant est utilisé de manière général pour la nommage : `Action_Given_Then`. "Action"

étant souvent le nom de méthode testée.

Exemple :

```
public class CalculatorTests {

  [Fact]
  public void Multiply_ValidOperands_ReturnsCorrectResult(){
  // Given
    int leftOperand = 3;
    int rightOperand = 4;

    var calculator = new Calculator();

    // When
    int result = calculator.Multiply(leftOperand, rightOperand);

    // Then
    Assert.Equal(12, result);
  }
}
```

Et avec un cas négatif, avec une vérification d'exception :

```
  [Fact]
  public void Divide_By0_Throws(){
  // Given
    int leftOperand = 3;
    int rightOperand = 0;

    var calculator = new Calculator();

    // When
    Action act = () => calculator.Divide(leftOperand, rightOperand);



    // Then
    Assert.Throws<DivideBy0Exception>(act);
  }
```

Setup

Pour faire un setup commun à tous les cas de tests d'une collection, il suffit d'utilise le constructeur de la classe qui correspond à cette collection, il est appelé pour chaque cas de test.

Tests paramétrisés

Il est possible de faire des tests paramétrisés, avec l'attribut `[Theory]`. On peut passer les paramètres de plusieurs manières.

Avec l'attribut `[InlineData]`

L'attribut `[InlineData]` permet de passer des données à un test paramètrisé mais ne supporte que les constantes à la compilation (littéraux de types primitif) :

```
public class CalculatorTests {

  [Theory]
  [InlineData(3,4, 12)]
  [InlineData(2,3, 6)]
  [InlineData(3,5, 15)]
  [InlineData(4,4, 16)]
  public void Multiply_ValidOperands_ReturnsCorrectResult(int leftOperand, int rightOperand,
int expectedResult){
  // Given

    var calculator = new Calculator();

    // When
    int result = calculator.Multiply(leftOperand, rightOperand);

    // Then
    Assert.Equal(expectedResult, result);
  }
}
```

Cela exécutera autant d'instant de cas de tests que de `[InlineData]`, ce cas de test en est en fait quatres !

Avec `TheoryData`

`TheoryData` Permet de passer des données à un test paramètrisé qui ne sont pas des constantes de compilation, et ce de façon fortement typée.

```
public class CalculatorTests {

  public static TheoryData<int, int, int> Data =>
    new TheoryData<int, int, int>
        {
            { 1, 2, 3 },
            { -4, -6, -10 },
            { -2, 2, 0 },
            { int.MinValue, -1, int.MaxValue }
        };


  [Theory]
  [MemberData(nameof(Data))]
  public void Add_ValidOperands_ReturnsCorrectResult(int leftOperand, int rightOperand, int
expectedResult){
  // Given

    var calculator = new Calculator();

    // When
    int result = calculator.Add(leftOperand, rightOperand);

    // Then
    Assert.Equal(expectedResult, result);
  }
}
```

Assertions

XUnit fourni un bon nombre de fonctions d'assertion, comme `Assert.Equal` ou `Assert.Throws`, mais certains librairies comme FluentAssertions existent pour faire des assertions un petit peu plus expressives :

```
string actual = "ABCDEFGHI";
actual.Should().StartWith("AB").And.EndWith("HI").And.Contain("EF").And.HaveLength(9);
```

```
IEnumerable<int> numbers = new[] { 1, 2, 3 };
numbers.Should().OnlyContain(n => n > 0);
numbers.Should().HaveCount(4, "because we thought we put four items in the collection");
```

Created 2024-11-21 14:58:51 UTC by Nicolas
Updated 2024-11-21 14:59:12 UTC by Nicolas