Formation aux tests automatisés

TESTS UNITAIRES

Séance 2

Quelques notions liées aux tests

LES TESTS UNITAIRES

La notion de boîte blanche

Dans la théorie des systèmes, une boîte blanche, ou boîte transparente, est un module d'un système dont on peut prévoir le fonctionnement interne car on connaît les caractéristiques de fonctionnement de l'ensemble des éléments qui le composent.

Le test en boîte blanche (white box testing, en anglais) est une méthode de test logiciel qui utilise le code source d'un programme comme base pour concevoir des tests et des scénarios de test pour l’assurance qualité (QA).

La notion de boîte noire

A l'opposé des tests de boîte blanche, on effectuera des tests logiciels appelés de type « boîte noire » sur des systèmes dont on n'a pas nécessairement accès au code source.

Nota Bene : Il peut être intéressant de considérer tous les tests de non-régression comme des tests de boîte noire, pour découpler le comportement souhaité de leur implémentation technique.

Les tests unitaires

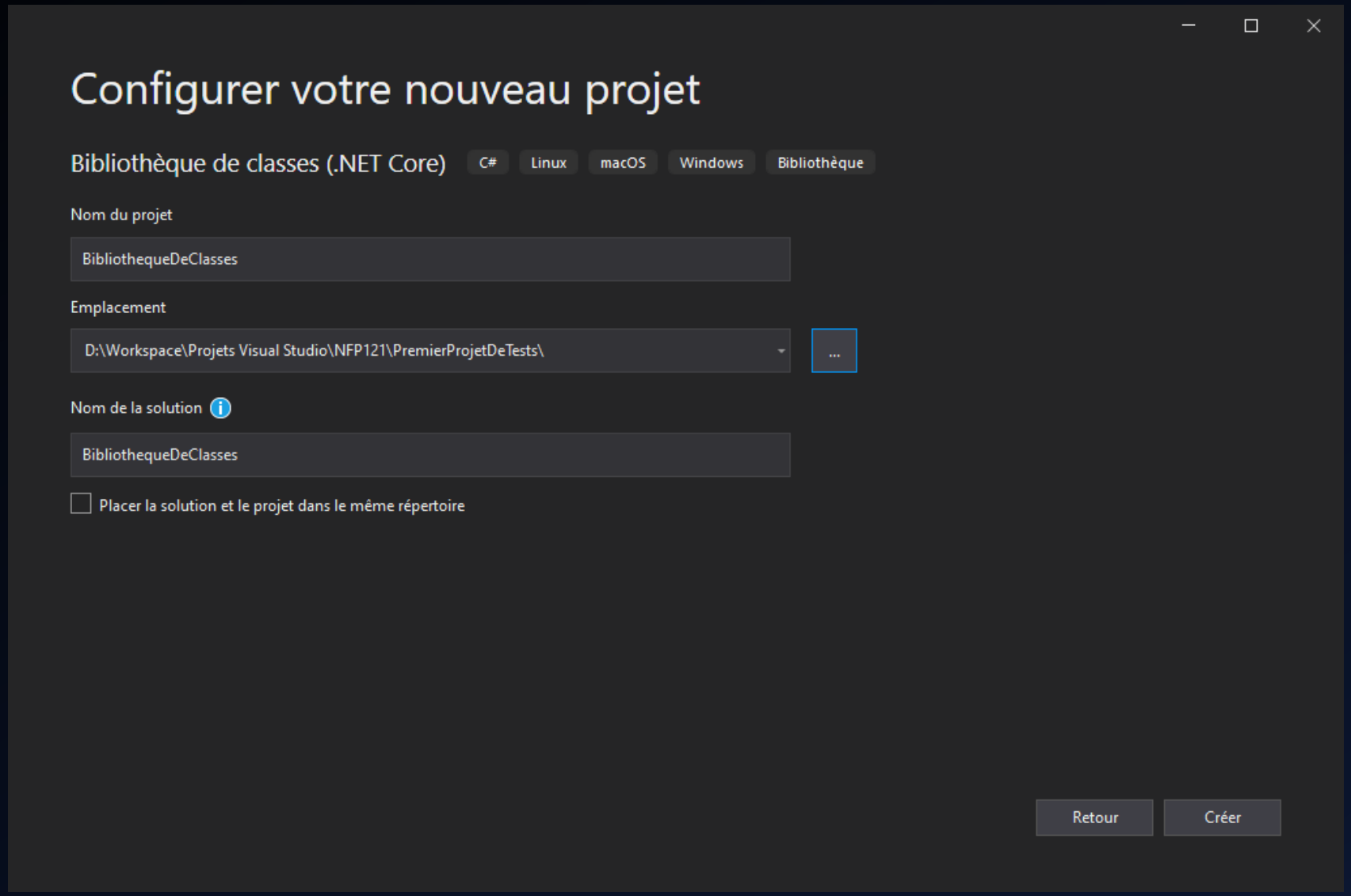
PREMIER EXEMPLE

Notre premier test unitaire

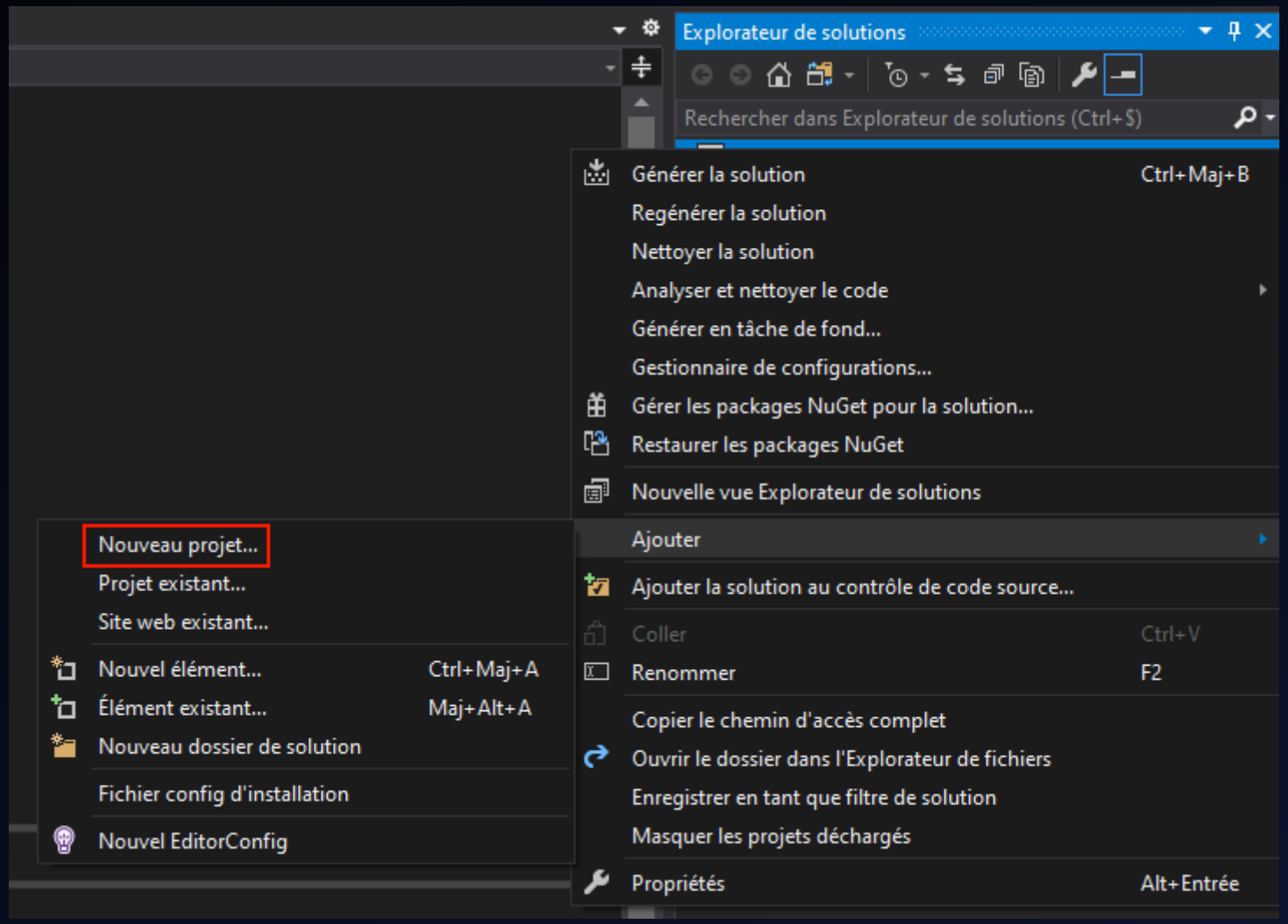
Nous allons prendre pour premier exemple la fonction la plus simple qu’on puisse trouver: l’addition.

Static int plus(int a, int b)

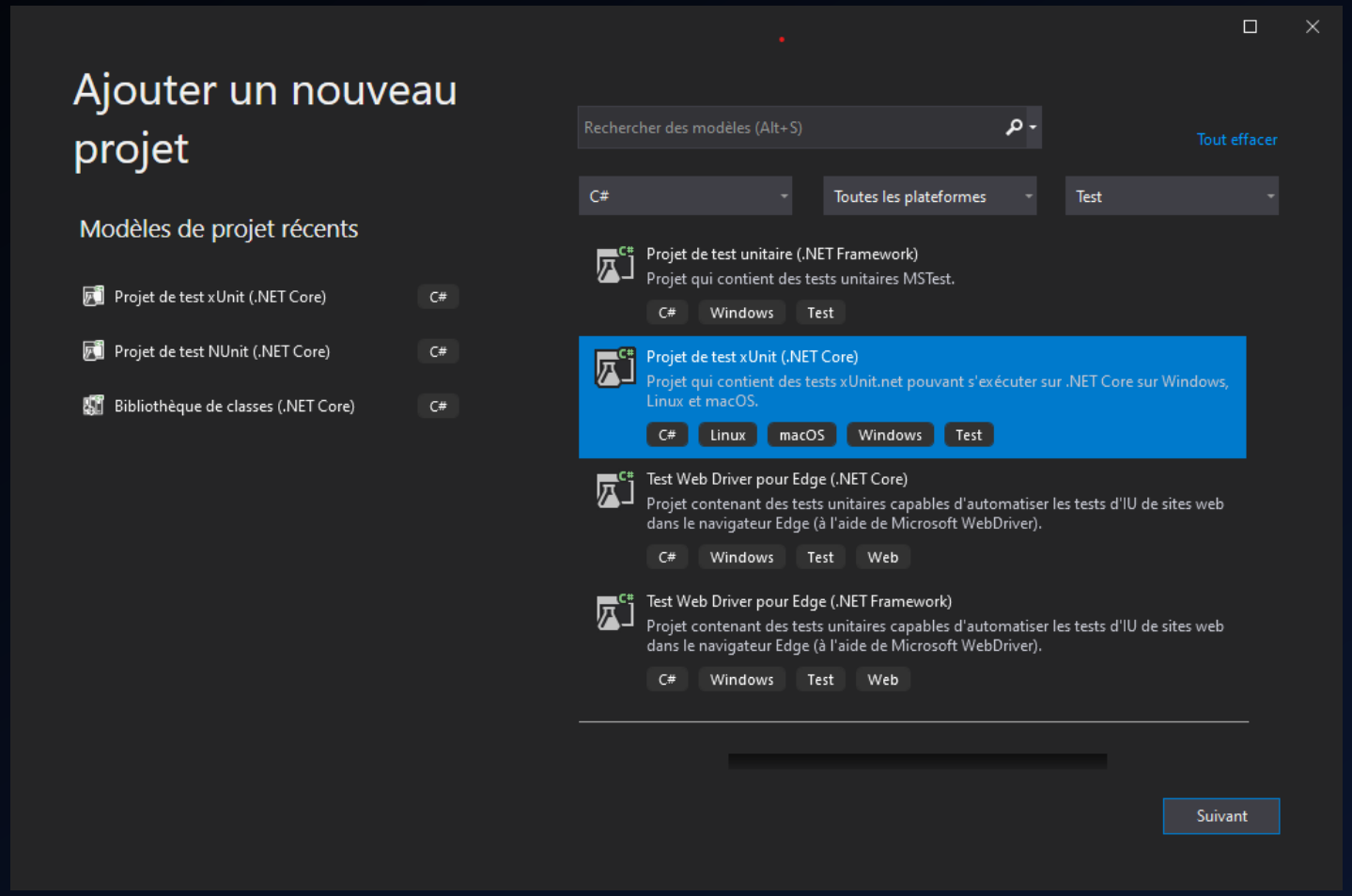
Création du projet à tester



Création du projet de tests



Création du projet de tests



Création du projet de tests

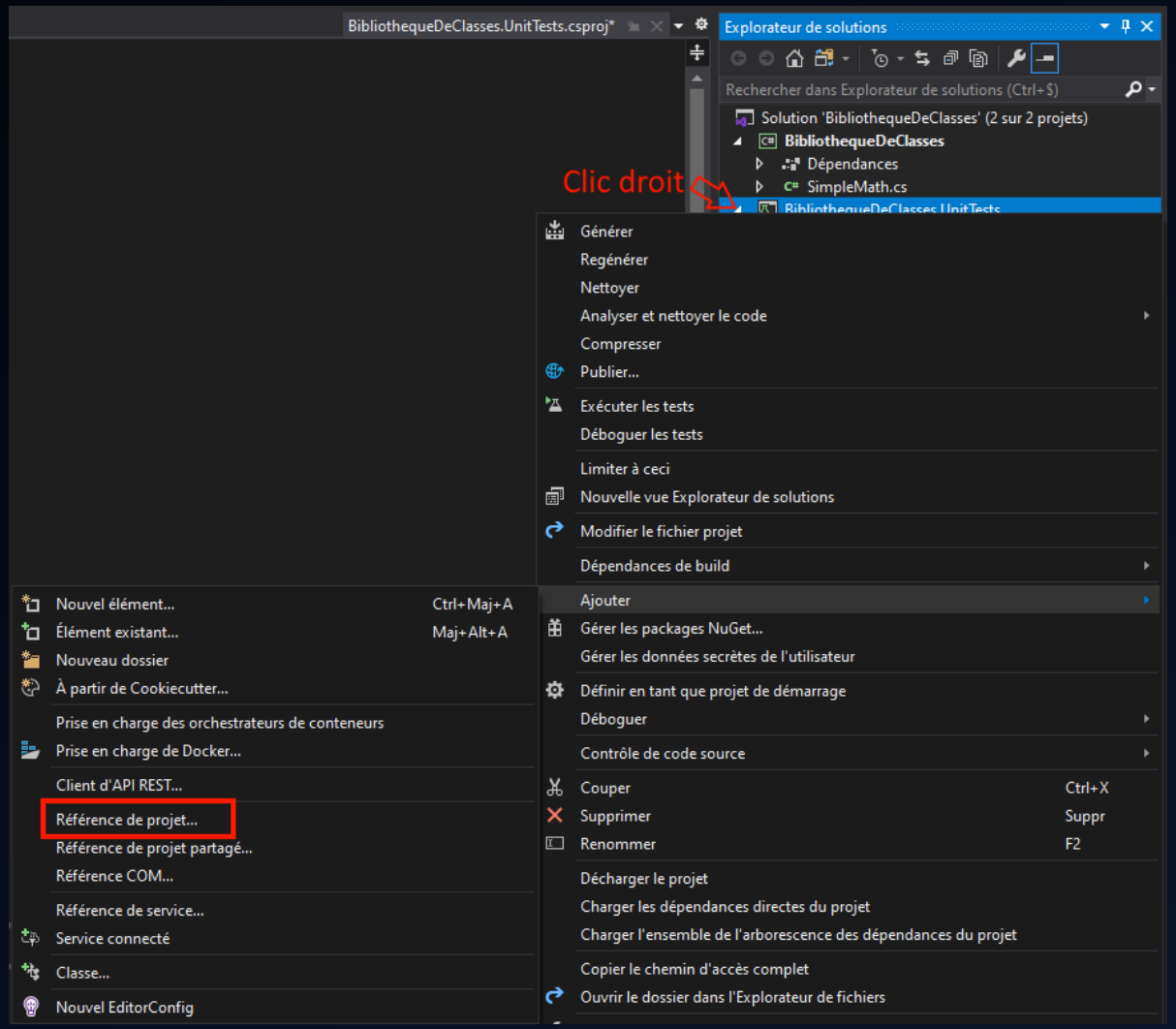
Code auto-généré

```
using System;
using Xunit;

namespace Bibliotheque.UnitTests
{
    0 références
    public class UnitTest1
    {
        [Fact]
        0 références
        public void Test1()
        {

        }
    }
}
```

Ajout de référence de projet au projet de tests



Ecriture de notre premier test

```
[Fact]
0 références
public void Test1()
{
    Assert.Equal(0, SimpleMath.Plus(0,0));
    Assert.Equal(1, SimpleMath.Plus(1,0));
    Assert.Equal(1, SimpleMath.Plus(0,1));
    Assert.Equal(5, SimpleMath.Plus(3,2));
}
```

Laquelle de ces assertions est inutile, ou peut nous induire en erreur?

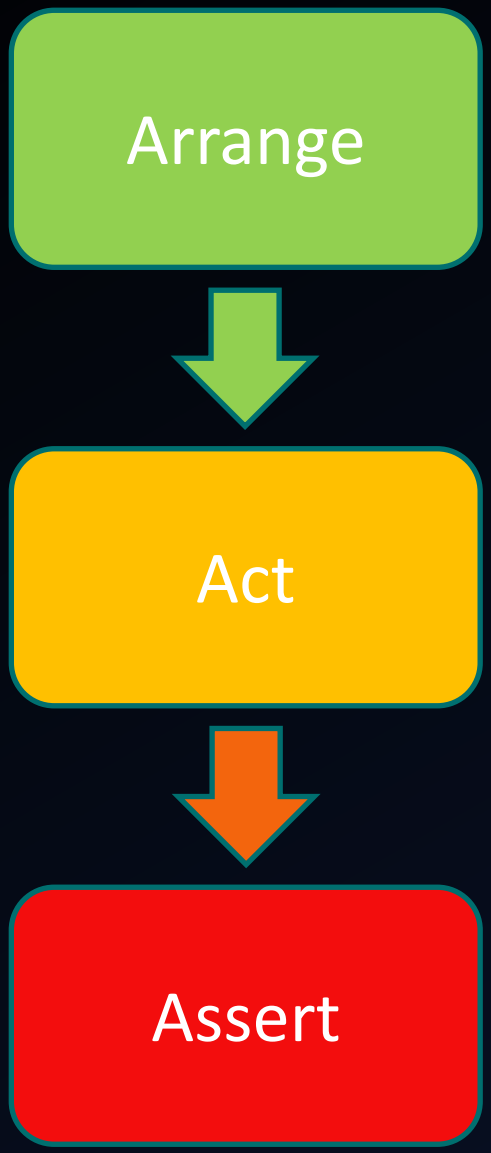
Ecriture de notre premier test

```
[Test]
✓ | 0 références
public void TestPlus()
{
    Assert.AreEqual(0, SimpleMath.Plus(0, 0));
    Assert.AreEqual(1, SimpleMath.Plus(1, 0));
    Assert.AreEqual(1, SimpleMath.Plus(0, 1));
    Assert.AreEqual(2, SimpleMath.Plus(1, 1));
}
```

Les tests unitaires

QUELQUES BONNES PRATIQUES

Comment écrire un test ?



- Création du contexte,
- Création des objets,
- Création des jeux de données d’entrée

- Exécution du code de production,
- Appel des méthodes,
- Assignation des propriétés,

- Vérification du résultat,
- Le test passe ou échoue.

Les Assertions

LES TESTS UNITAIRES

Les assertions

Définition :

- En linguistique et en philosophie, une **assertion** représente un énoncé considéré ou présenté comme vrai.
- En programmation informatique, une assertion est une expression qui doit être évaluée à vrai. Si cette évaluation échoue elle peut mettre fin à l'exécution du programme, ou bien lancer une exception.

Combien de Assert par test ?

Comme toujours en programmation, on essaie de faire une seule chose à la fois.

On peut parfois lire qu'un seul Assert doit être présent dans un test. Mais, certaines écoles plus « souples » permettent plusieurs Assertions par test **à la condition qu'elles testent toutes le même comportement**.

Les types d’assertions de xUnit

LES TESTS UNITAIRES

Différentes utilisations d'Assert

Booleans :

- True / False

Valeurs numériques :

- Égalité / Inégalité
- Dans des bornes
- Précision des floats

Méthodes :

Assert.True(boolean)
Assert.False(boolean)

Différentes utilisations d'Assert

Strings :

- Égalité / Inégalité
- Vide
- Commence / Finit par …
- Contient une substring
- Valide une expression régulière

Méthodes :

```
Assert.Equal("expected", "SUT");
Assert.Empty("SUT");
Assert.NotEmpty("SUT");
Assert.StartsWith("Recherché", "SUT");
Assert.EndsWith("Recherché", "SUT");
Assert.Contains("Recherché", "SUT");
Assert.DoesNotContain("Recherché", "SUT");
Assert.Matches("Regex", "SUT");
Assert.DoesNotMatch("Regex", "SUT");
```

Différentes utilisations d'Assert

Les collections :

- Égalité / Inégalité
- Contient ou non un élément
- Tous les éléments valident une vérification

```
Assert.Equal(ListeFilms , ListeFilmsNuls);
Assert.Contains("The Prestige", ListeFilms.Select(m => m.Title));
Assert.DoesNotContain("Transformers", ListeFilms.Select(m => m.Title));
Assert.All(ListeFilms, film => Assert.NotEmpty(film.DirectorName));
```

Différentes utilisations d'Assert

Les Types d’objets :

Est ou non d’un certain type concret

```
Assert.IsNotType<Enfant>(new Parent());
Assert.IsNotType<Parent>(new Enfant());
Assert.IsType<Enfant>(new Enfant());
```

Différentes utilisations d'Assert

Les Exceptions :

C’est une bonne pratique de tester si un code renvoie bien ou ne renvoie pas une exception.

```
int Zero = 0;
Assert.Throws<DivideByZeroException>(() => { int i = 5 / Zero; });

Enfant enfant = new Enfant();
ArgumentNullException ex = Assert.Throws<ArgumentNullException>(() => enfant.Appeler(null));
// on peut faire d'autrs asserts sur l'exception
Assert.Equal(_messageExceptionParametreNull, ex.ParamName);
```

Organiser ses tests dans un projet

LES TESTS UNITAIRES

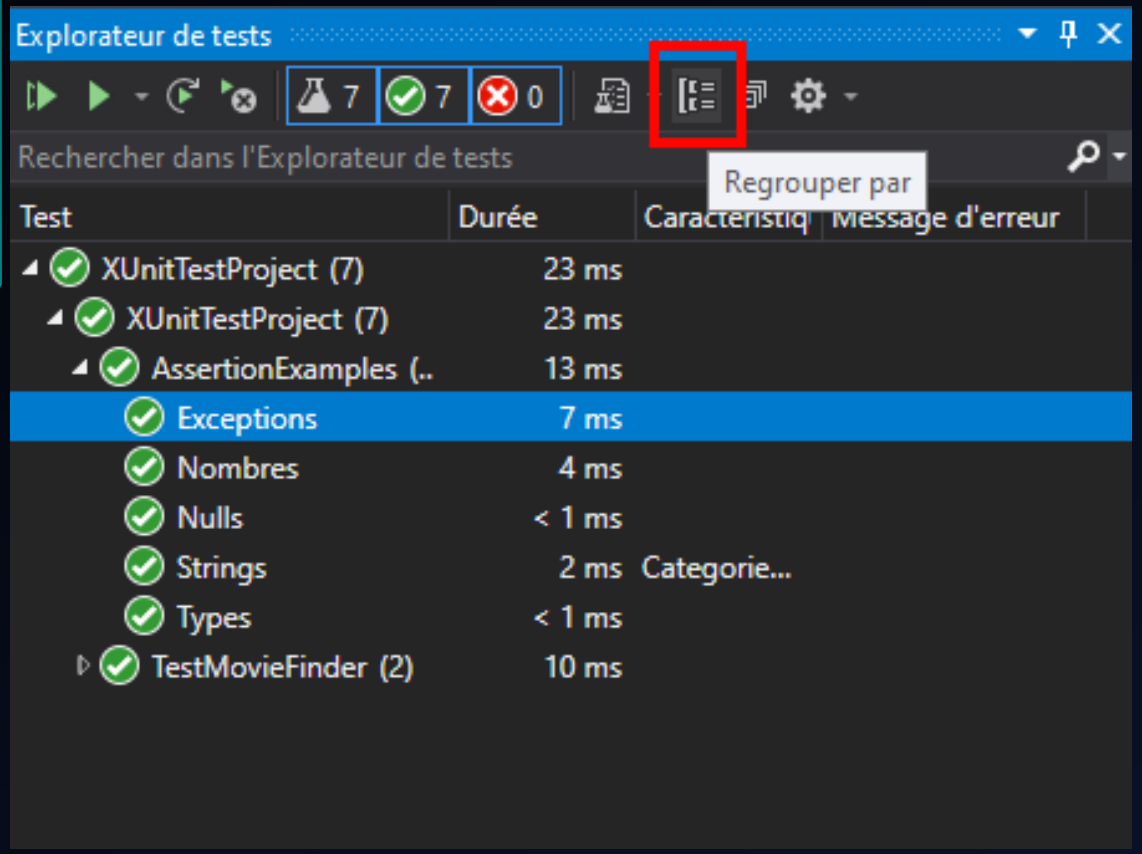
Créer des sous-groupes de tests

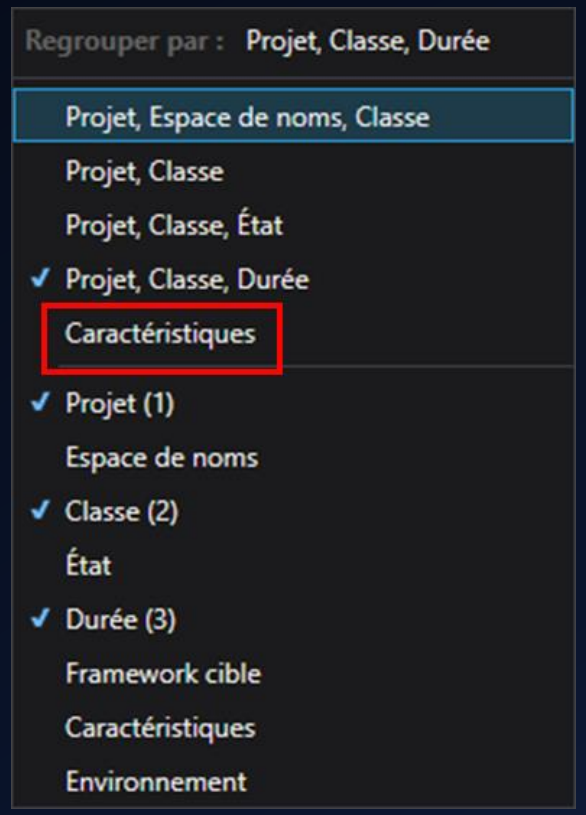
Pour classifier les tests, on peut les grouper en sous-groupes. Pour cela, le framework xUnit nous propose un décorateur nommé « Trait ».

```
[Fact]
[Trait("Categorie", "Strings")]
0 références
public void Strings()
{
```

Créer des sous-groupes de tests

Une fois la catégorie appliquée, on peut trier par caractéristiques.





Créer des sous-groupes de tests

Astuce, on peut appliquer ceci à une classe plutôt qu'à chaque test.

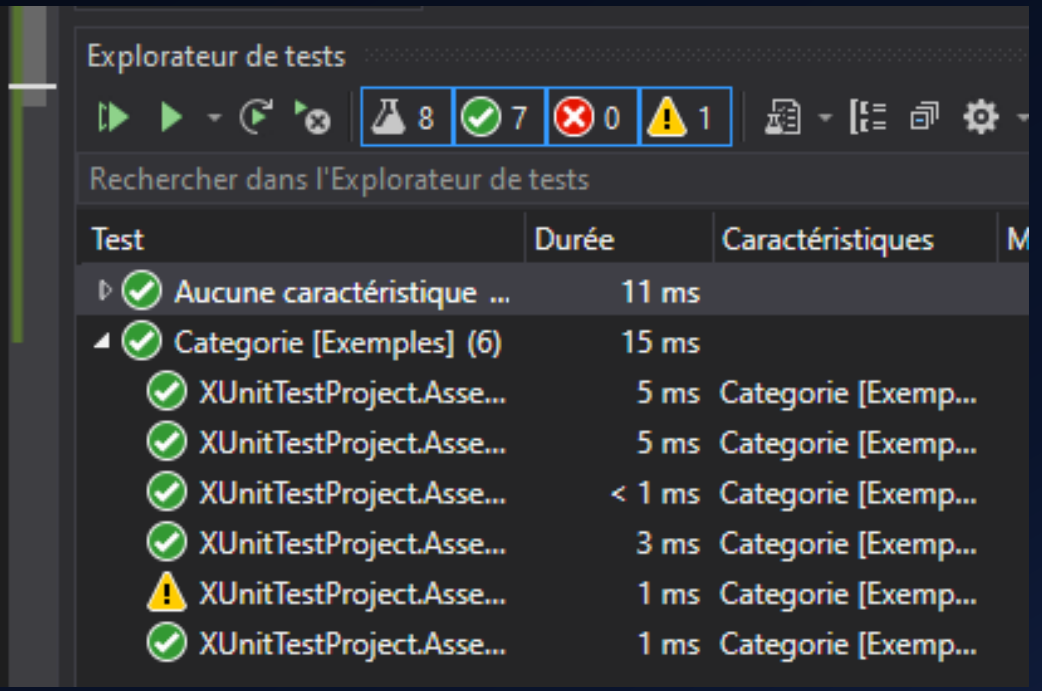
```
[Trait("Categorie", "Exemples")]
0 références
public class AssertionExamples
{
```

Ignorer des tests

Pour diverses raisons (pas le fait qu'ils échouent sans qu'on sache pourquoi) on peut avoir besoin d'ignorer un ou plusieurs tests.

```
[Fact(Skip = "Ignoré")]
⚠ | 0 références
public void TestAIgnorer()
{
    Assert.True(false);
}

3 références
```



Afficher du texte en sortie de test

LES TESTS UNITAIRES

Afficher du texte en sortie des tests

Les tests n'étant pas des applications console, on ne peut pas utiliser Console.Writeline(string) pour afficher du texte dans un test. Pour cela, on doit utiliser un attribut dans la classe de tests.

Cet attribut doit être une implémentation de l'interface ITestOutputHelper

```
[Trait("Categorie", "Exemples")]
1 référence
public class AssertionExamples
{
    private const string _messageExceptionParametreNull = "Message d'exception";
    private static ITestOutputHelper _output;
```

Afficher du texte en sortie des tests

Une fois l'attribut de l'interface ITestOutputHelper déclaré dans la classe, il faut lui donner une valeur par défaut. C'est le framework xUnit qui va s'en charger. On doit juste créer un constructeur qui prend en paramètre un objet de l'interface ITestOutputHelper et qui met à jour l'instance de classe.

```
0 références
public AssertionExamples(ITestOutputHelper testOutputHelper)
{
    _output = testOutputHelper;
}
```

Afficher du texte en sortie des tests

Insérer un objet de type d'interface dans le constructeur utilise un mécanisme appelé « Injection de dépendances ».

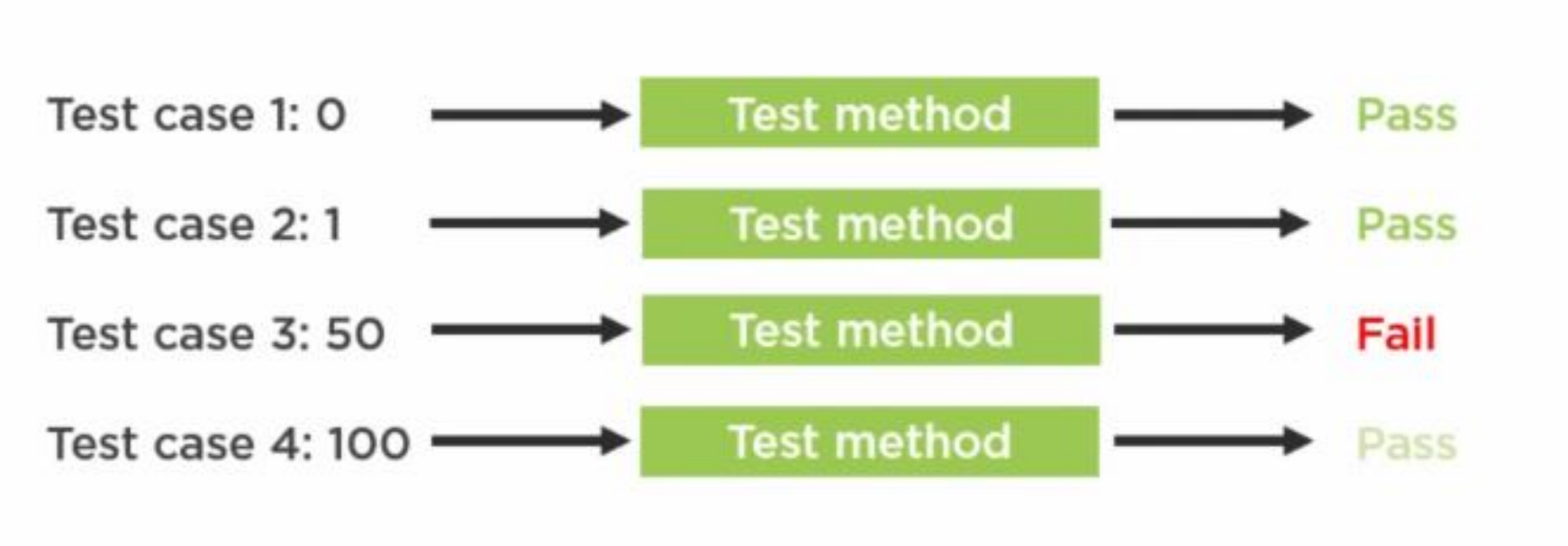
```
0 références
public AssertionExamples(ITestOutputHelper testOutputHelper)
{
    _output = testOutputHelper;
}
```

Quelles données utiliser pour les tests

LES TESTS UNITAIRES

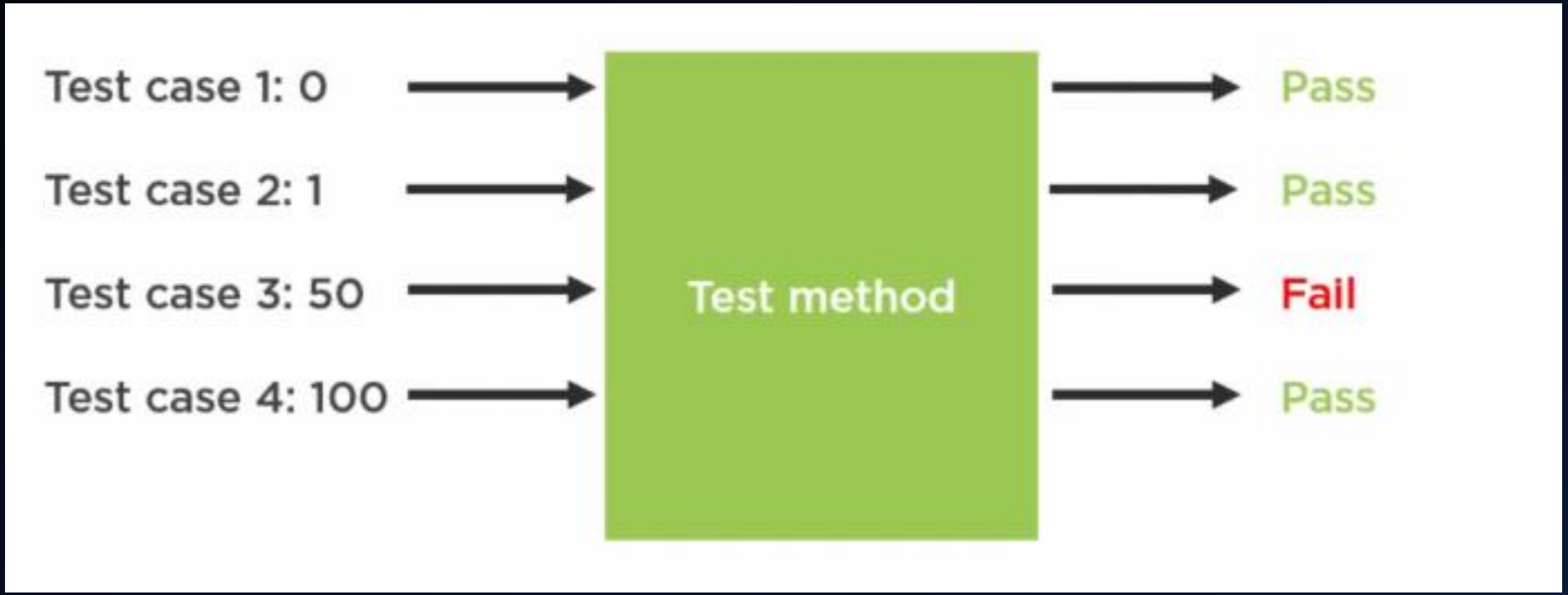
Utiliser des jeux de données pour une fonction de test

Avec les techniques vues jusqu'ici, il faut une méthode de test, ou un assert par jeux de données. C'est assez rébarbatif, et coûteux en termes de temps d'écriture, mais surtout de maintenabilité.



Utiliser des jeux de données pour une fonction de test

On cherchera donc à utiliser des jeux de données pour pouvoir appeler les méthodes de tests avec seulement les données qui varient.



Utiliser des jeux de données pour une fonction de test

On cherchera donc à utiliser des jeux de données pour pouvoir appeler les méthodes de tests avec seulement les données qui varient.

Pour cela, on utilisera les décorateurs suivants:

```
[Theory]
[InlineData(1 , 1, 2)]
[InlineData(0 , 1, 1)]
[InlineData(1 , 0, 1)]
[InlineData(1 , 4, 5)]
 | 0 références
public void TestAjout(int a, int b, int somme)
{
    Assert.Equal(somme, Ajouter(a, b));
}
```

Quelles données utiliser pour les tests

LES TESTS UNITAIRES

Partager des jeux de données

Malheureusement l’attribut InlineData ne permet pas de partager de données entre différentes méthodes. Pour cela, nous allons créer des classes spécifiques.

Partager des jeux de données

On crée une classe dans le projet de tests, dans notre exemple on la nomme InternalHealthDamageTestData. On implémente ensuite une propriété qui nous permettra de récupérer les données dans les tests.

```
public class InternalHealthDamageTestData
{
    0 références
    public static IEnumerable<object[]> TestData
    {
        get
        {
            yield return new object[] { 0, 100 };
            yield return new object[] { 1, 99 };
            yield return new object[] { 50, 50 };
            yield return new object[] { 75, 25 };
            yield return new object[] { 101, 1 };
        }
    }
}
```

Partager des jeux de données

Une fois la classe InternalHealthDamageTestData crée, on indique au test qu'il doit utiliser cette source de données pour choisir les données à passer au SUT.

```
[Theory]
[MemberData(nameof(InternalHealthDamageTestData.TestData),
    MemberType = typeof(InternalHealthDamageTestData))]
| 0 références
public void TakeDamage(int damage, int expectedHealth)
{
    _sut.TakeDamage(damage);

    Assert.Equal(expectedHealth, _sut.Health);
}
```