Tests Automatisés avec DOT NET

Utilité

Un test automatisé permet de tester une fonctionnalité d'un programme informatique.
La plus-value d'un test automatisé est (surtout) de vérifier que la partie concernée reste opérationnelle après une modification sans avoir à repasser des tests manuels (tests de non-régression).
Bien sûr, en cas de modification d'une fonctionnalité déjà testée dans une version précédente, il y a de fortes chances que l'on doit alimenter la batterie de tests automatisés.

Quels sont les tests que l'on peut automatiser ?

On peut automatiser tous les tests que l'on peut faire subir à un programme (mais pas toujours avec les mêmes outils), c'est-à-dire :

- Les tests unitaires (la partie de programme que l'on ajoute ou modifie indépendamment du microcosme)
- Les tests d'intégration (la partie de programme que l'on ajoute ou modifie une fois intégré dans son microcosme)
- Les tests fonctionnels (garantie que chaque fonctionnalité fonctionne conformément aux exigences)
- Les tests de montée en charge (permet de valider une solution et de disposer d'une connaissance exacte de ses limites et des goulots d'étranglement)
- Les tests de performance (les temps de réponse sont-ils à la hauteur ? La recherche de l'optimisation)

Ce document portera son attention sur les tests unitaires.

Qui est concerné ?

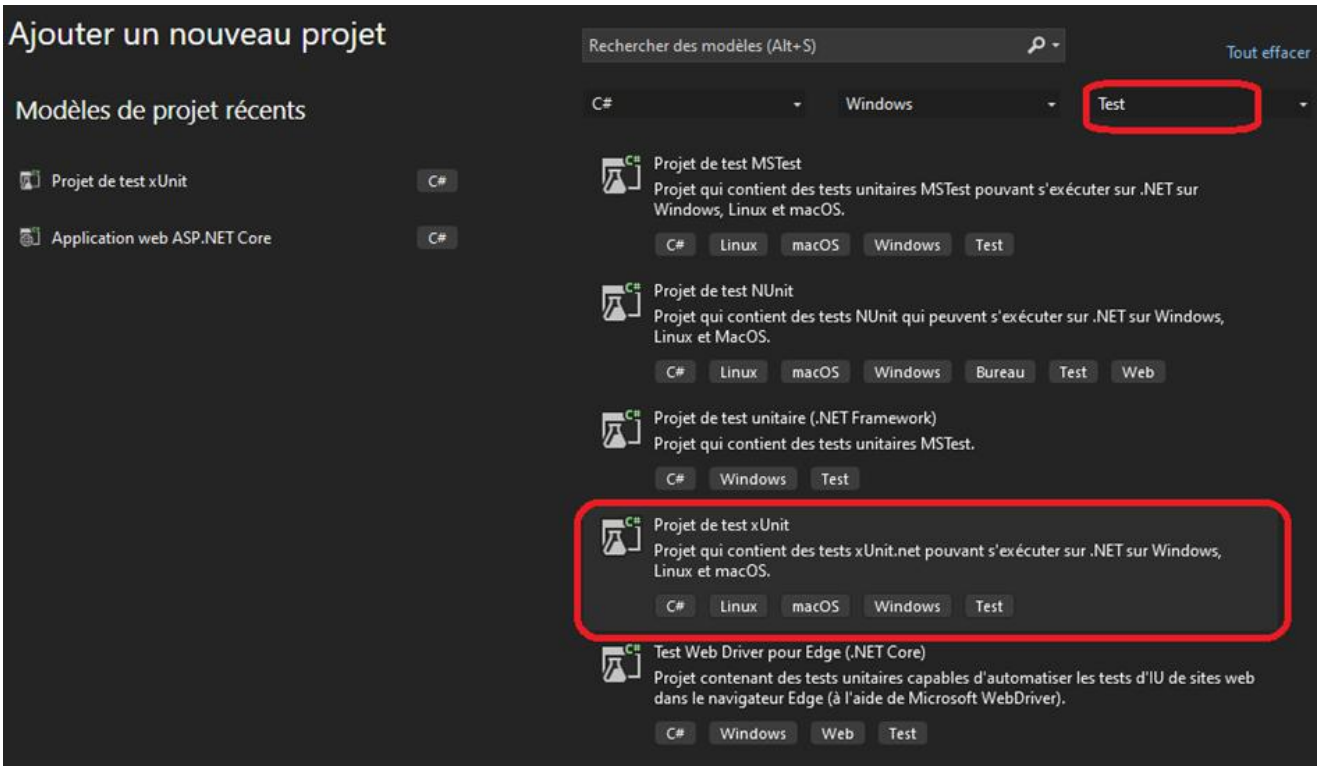
Chaque personne créant ou modifiant une fonctionnalité dans un programme se doit de mettre en place des tests automatisés pour garantir la stabilité du programme concerné.

Comment faire pour commencer ?

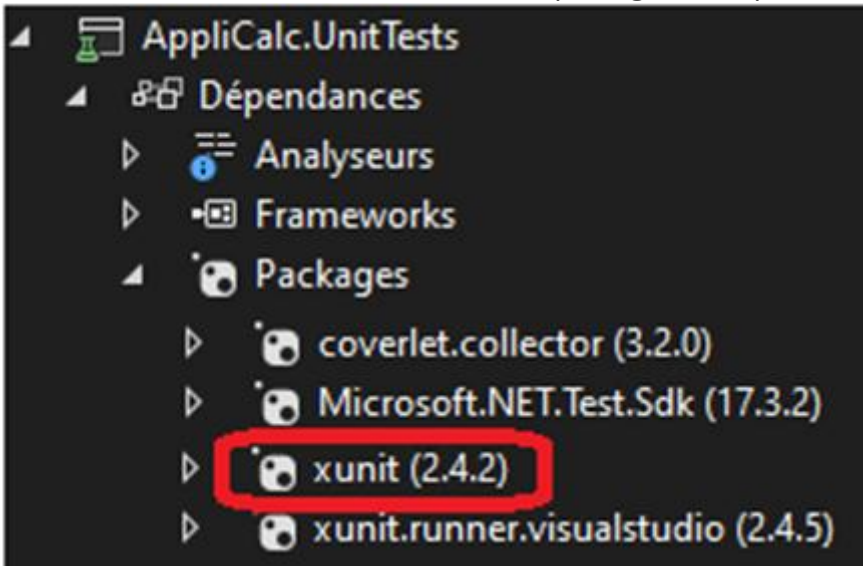
La démarche présentée sera effectuée avec Visual Studio 2022.

Quel outil de base pour créer des tests automatisés ?

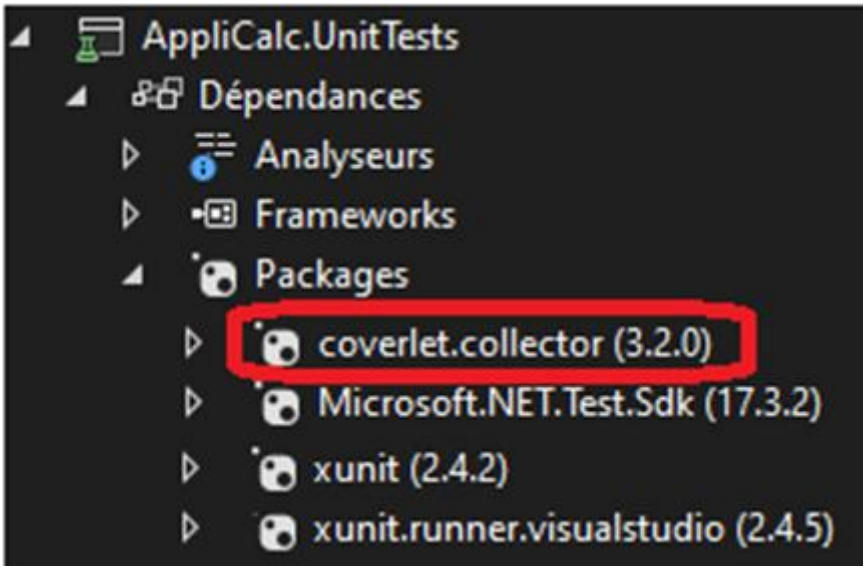
Visual Studio met à notre disposition des projets de tests automatisés : **xUnit**.
Il suffit d'ouvrir la solution Visual Studio contenant le projet à tester et de créer un nouveau projet et de sélectionner



Cet outil de base sera sans doute accompagné d'autres outils selon les cas pour nous permettre de faire des tests exhaustifs.
On constate sa bonne installation dans les packages de dépendance

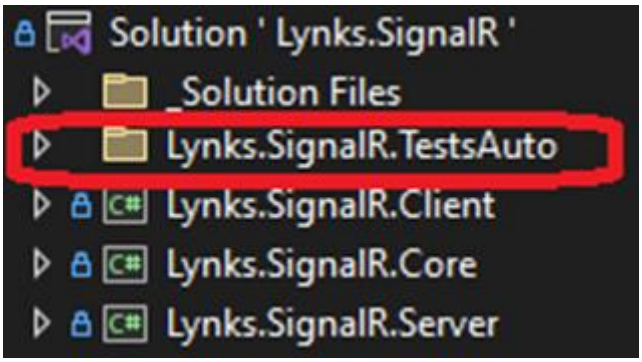


On vérifie également à cet endroit que le package « coverlet.collector » est également installé (sinon, nous vous suggérons de l'installer grâce au même principe que pour xUnit)

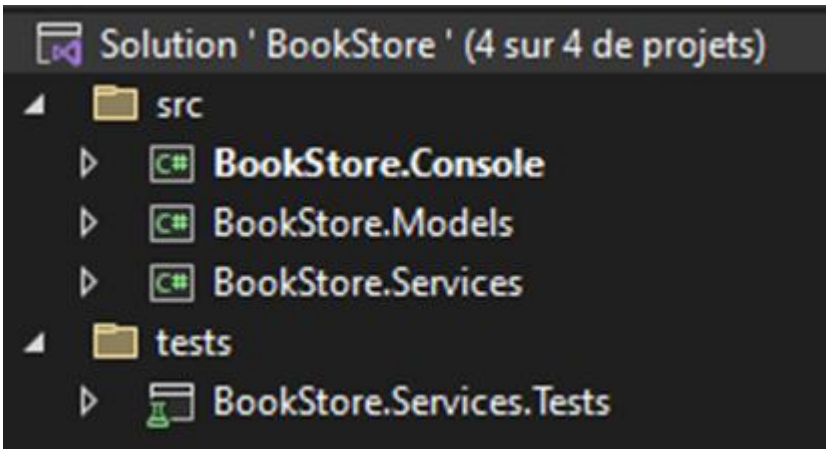


Architecture de la solution

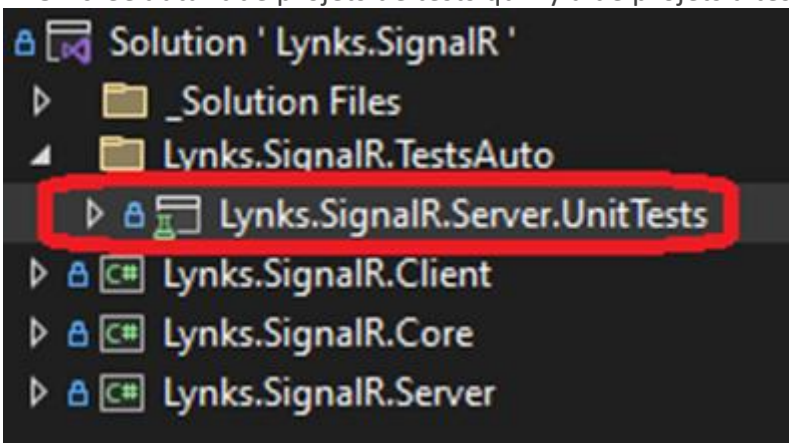
La finalité est d'obtenir une architecture de tests qui colle à l'architecture des sources.
On crée un dossier dans la solution qui porte le même nom que la solution suffixé de « .TestsAuto »



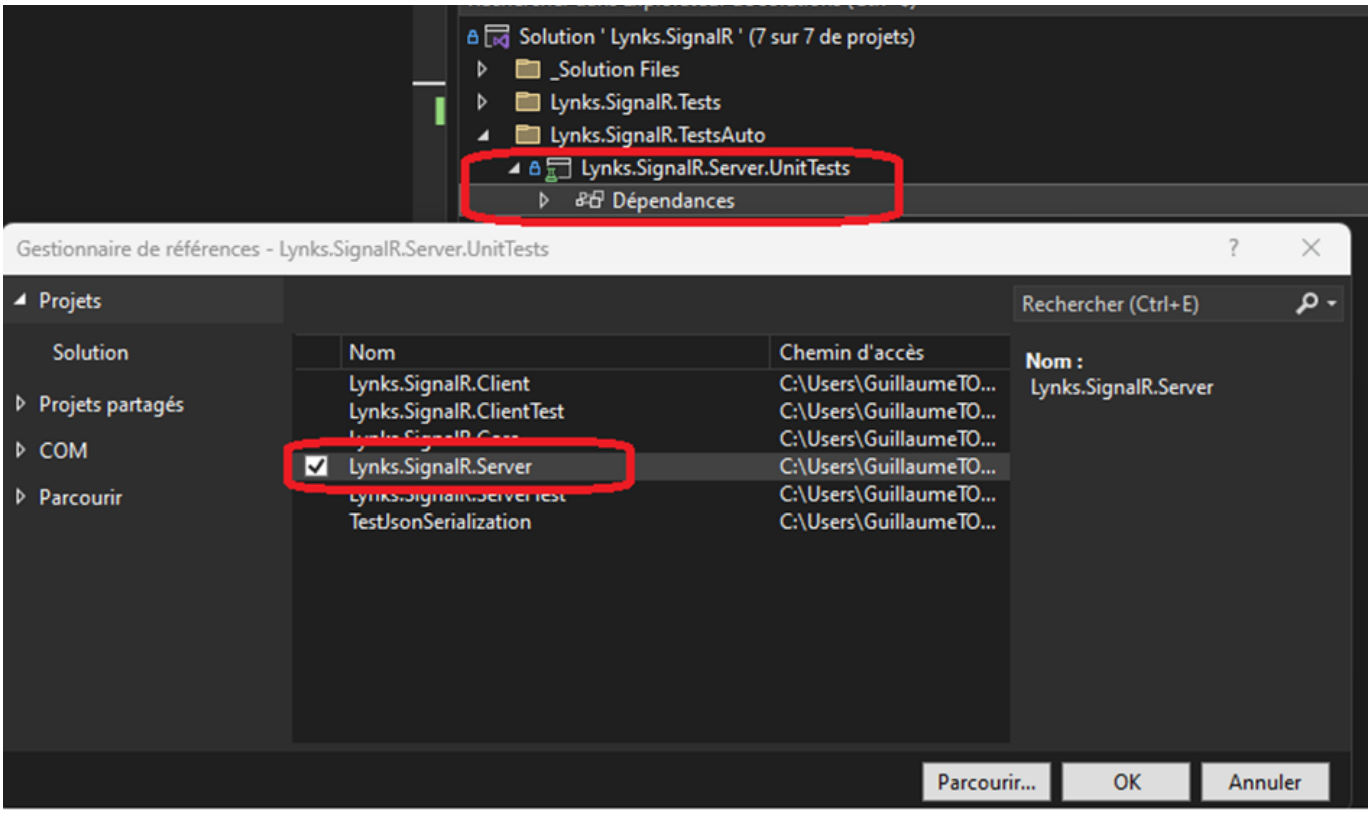
Ps : les bonnes pratiques poussées au maximum voudraient que la totalité des sources soient dans un dossier « src » et la totalité des projets tests dans un dossier « tests », mais comme nous ajoutons la notion de tests automatisés une fois les projets largement commencés, nous nous adaptons aux circonstances présentes. Mais peut-être que pour les futurs projets, nous pourrons nous en inspirer.



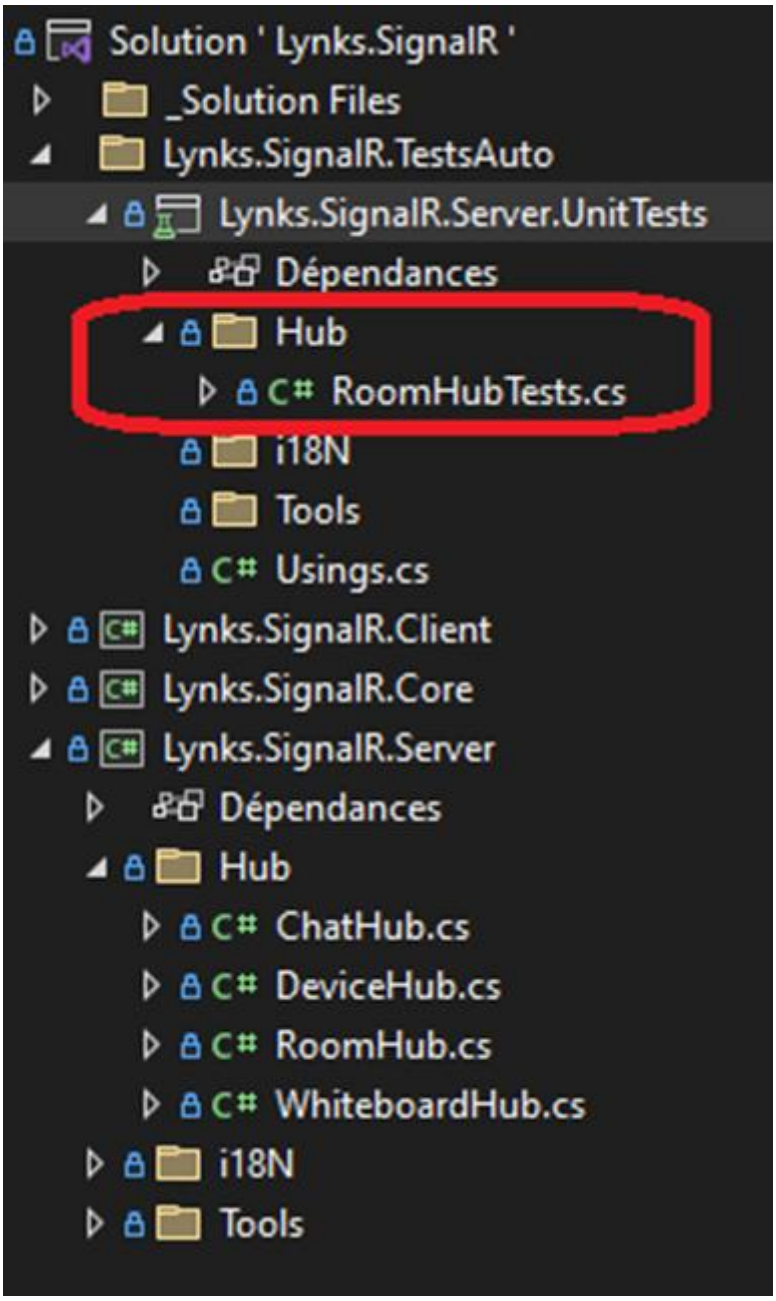
2. On crée autant de projets de tests qu'il y a de projets à tester



Le nom du projet de tests sera le nom du projet à tester suffixé de « *.type de tests* » (*UnitTests* pour les tests unitaires, et *IntegrationTests* pour les tests d'intégration).
Dans l'exemple ci-dessus, on crée des tests unitaires pour le projet « Lynks.SignalR.Server », aussi le projet de tests sera « Lynks.SignalR.Server.UnitTests »
IMPORTANT : Une fois le projet de tests créé, il faudra créer une référence projet pointant sur le projet à tester

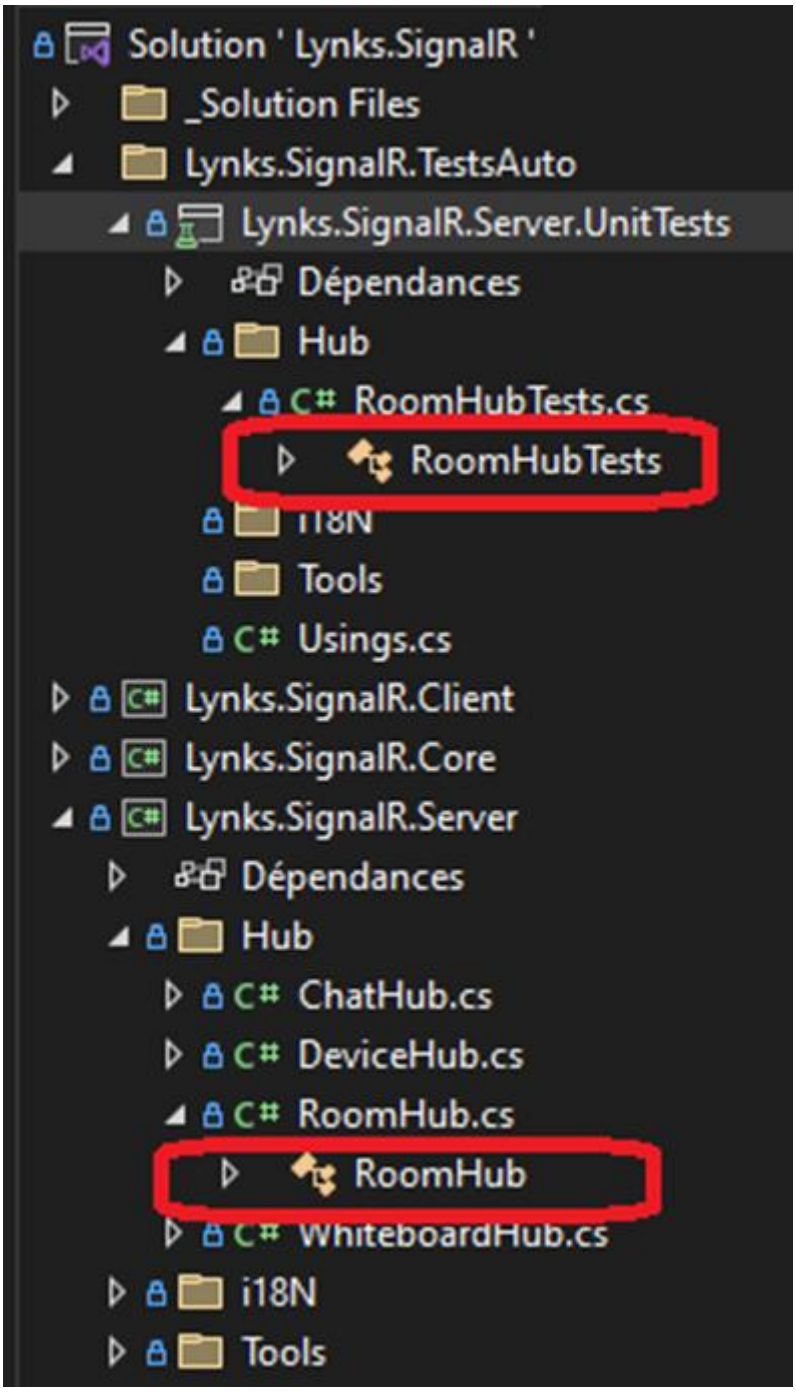


3. On crée autant de fichiers .CS de tests qu'il y a de fichiers .CS à tester en respectant l'arborescence



Le nom du fichier .CS de tests sera le nom du fichier .CS à tester suffixé de « Tests ».
Dans l'exemple ci-dessus, on crée des tests pour le fichier « RoomHub » qui est dans un dossier « Hub », aussi le fichier de tests sera « RoomHubTests » dans un dossier « Hub »

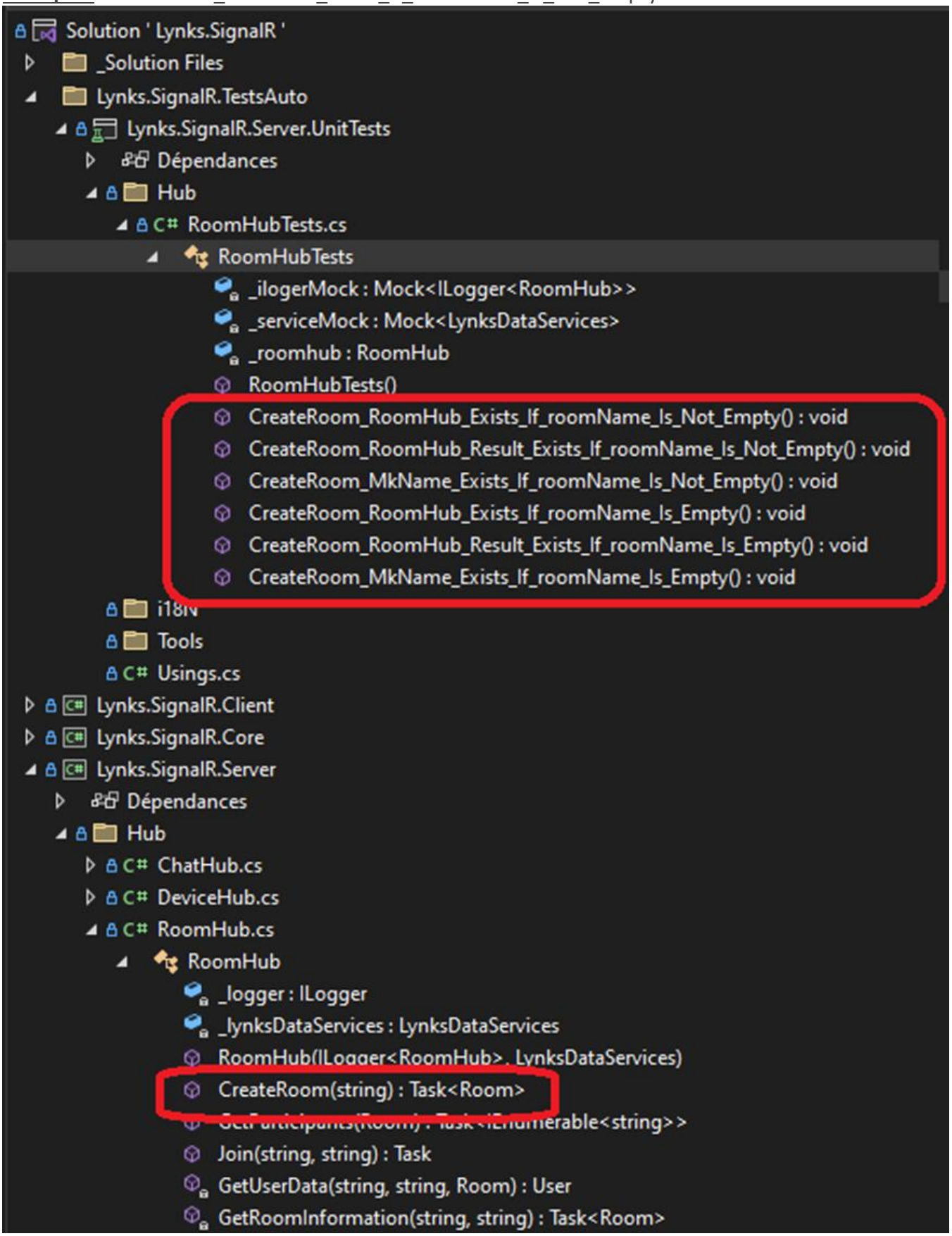
4. On crée autant de classes de tests qu'il y a de classes à tester en respectant l'arborescence



Le nom de la classe de tests sera le nom de la classe à tester suffixée de « Tests ».
Dans l'exemple ci-dessus, on crée des tests pour la classe « RoomHub », aussi la classe de tests sera « RoomHubTests ».

5. Enfin, on crée autant de méthode de tests que nécessaire pour tester une méthode.
Les règles de gestion, le bon sens et l'expérience vous permettront de créer autant de méthodes que nécessaire pour blinder votre développement.
Le nom de la méthode de tests sera celle de la méthode que vous testez suffixée d'un message explicit (on n'économise pas les caractères) : nom de méthode + scénario de test + comportement attendu

Exemple : CreateRoom_RoomHub_Exists_If_roomName_Is_Not_Empty



Dans l'exemple ci-dessus, on crée 6 tests unitaires pour tester la méthode « CreateRoom ».

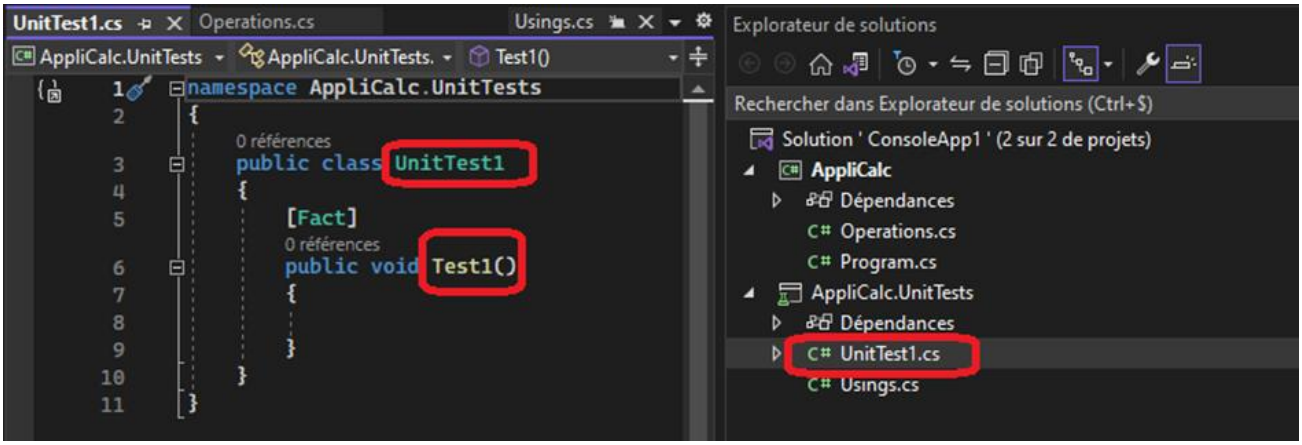
Quelles méthodes tester ?

Pour faire simple : toutes les méthodes *public* et toutes les méthodes *internal* (pour ces dernières, une petite manipulation sera à faire, la procédure est donnée plus loin dans ce document).
Ce faisant, toutes méthodes privées seront de facto testées.

Pour commencer

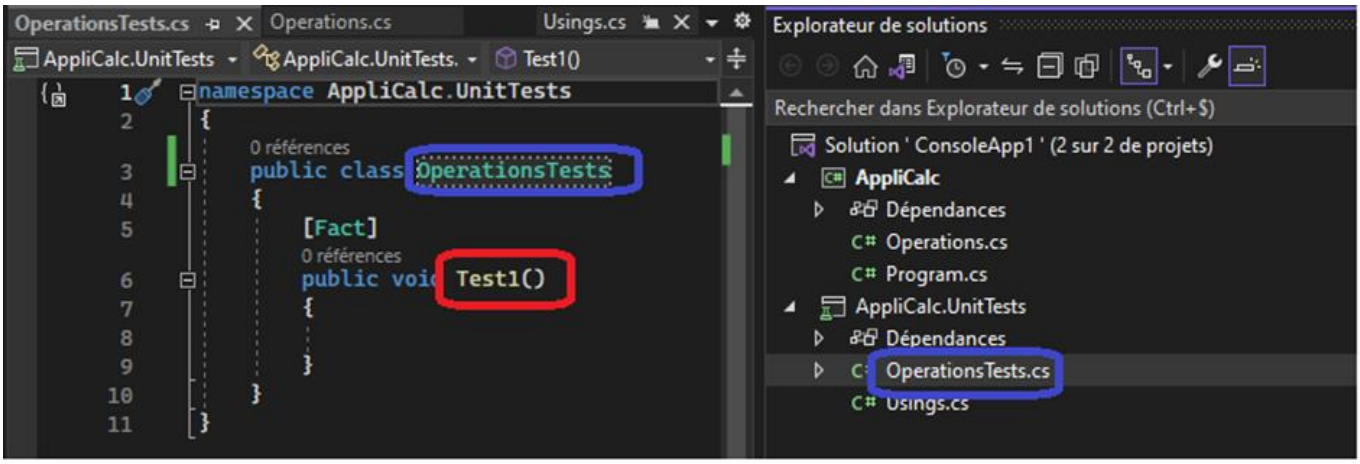
Par défaut, Visual Studio vous a créé un premier fichier .CS, une première classe et une première méthode. Vous pouvez les bouger / renommer ou les supprimer pour partir sur des bases saines.

AVANT



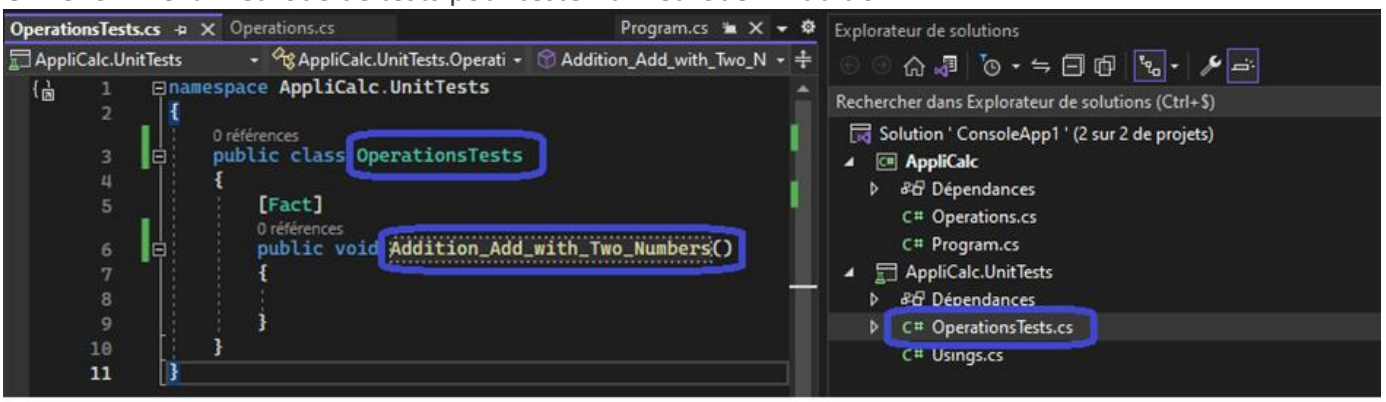
APRES (étape 1)

On renomme le fichier .CS et la classe



APRES (étape 2)

On renomme la méthode de tests pour tester la méthode « Addition »



Maintenant, nous pouvons alimenter notre méthode de tests pour atteindre notre objectif : garantir la qualité de notre code.

Mais avant d'aller plus loin, qu'est-ce que ce **[Fact]** que nous a insérer Visual Studio au-dessus de la méthode ?

L'attribut **[Fact]** déclare une méthode de test exécutée par l'exécuteur de test (quand on commandera l'exécution des tests, les méthodes seront exécutées automatiquement).

Ainsi, chaque méthode de tests doit porter cet attribut (ou d'autres que l'on verra plus tard).

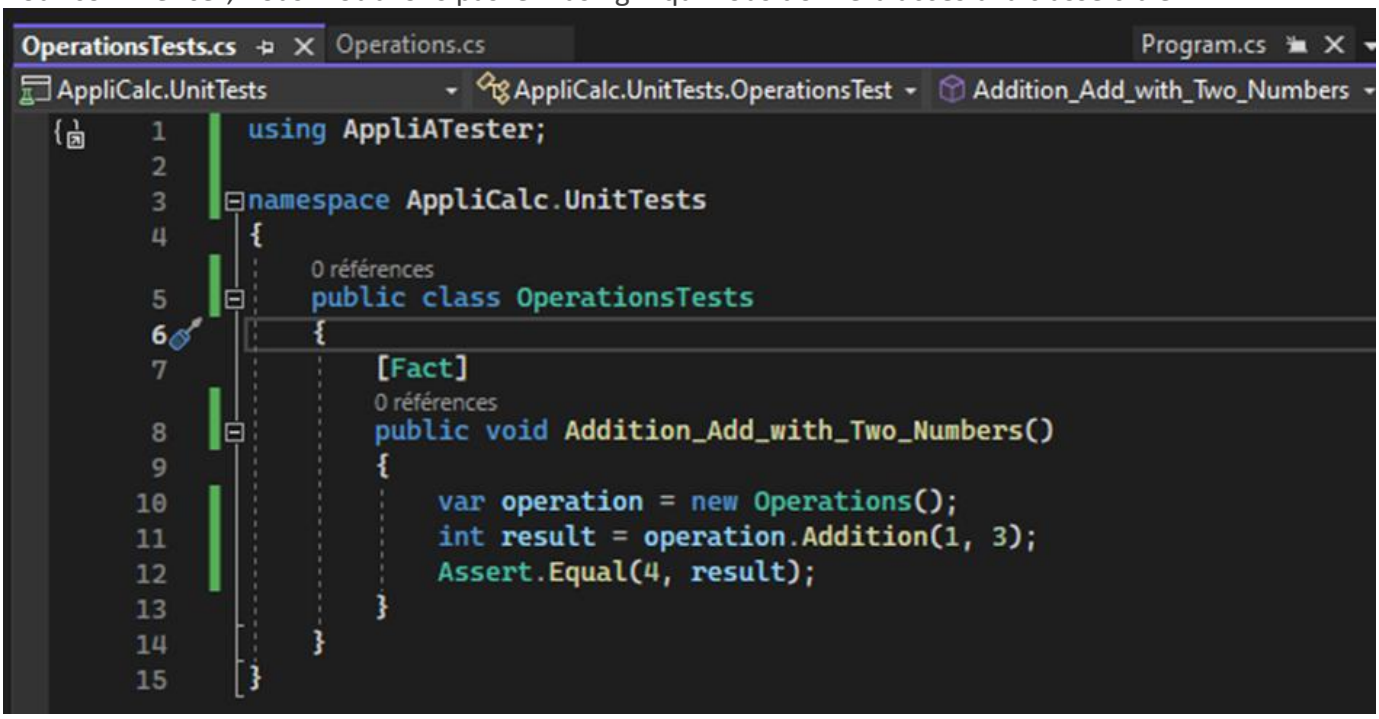
Comment faire pour tester une méthode ?

La méthode à tester

```
namespace AppliATester
{
    0 références
    public class Operations
    {
        /// <summary>
        /// Permet l'addition de 2 nombres donnés en paramètre
        /// </summary>
        /// <param name="a"></param>
        /// <param name="b"></param>
        /// <returns></returns>
        0 références
        public int Addition(int a , int b)
        {
            return a + b;
        }
```

La méthode de tests

Pour commencer, nous n'oublions pas le « using » qui nous donnera accès à la classe cible.

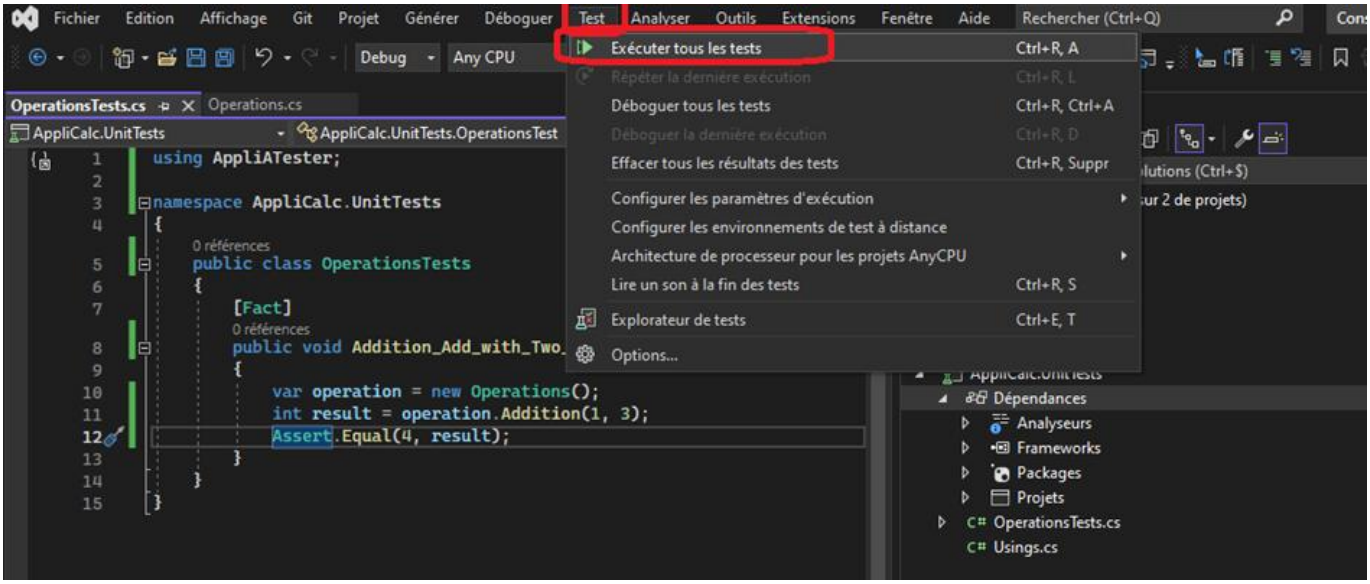


Pour tester cette méthode, nous choisissons arbitrairement de comparer le résultat de notre méthode « Addition » avec l'addition 1 + 3.
Grace à l'objet « Assert », nous pouvons comparer (ici, une égalité) le résultat attendu du résultat obtenu.
Ces 3 lignes de code représentent à elles 3 les 3 étapes que l'on retrouve dans un test automatisé : « Arrange / Act / Assert » (ou appelé aussi « AAA ») :
Arrange : prépare le terrain en initialisant les objets et en définissant la valeur des données transmises à la méthode testée.
Act : fait appel à la méthode à tester
Assert : vérifie que l'action de la méthode testée se comporte comme prévu
Donc notre code avec les parties AAA identifiées devient :

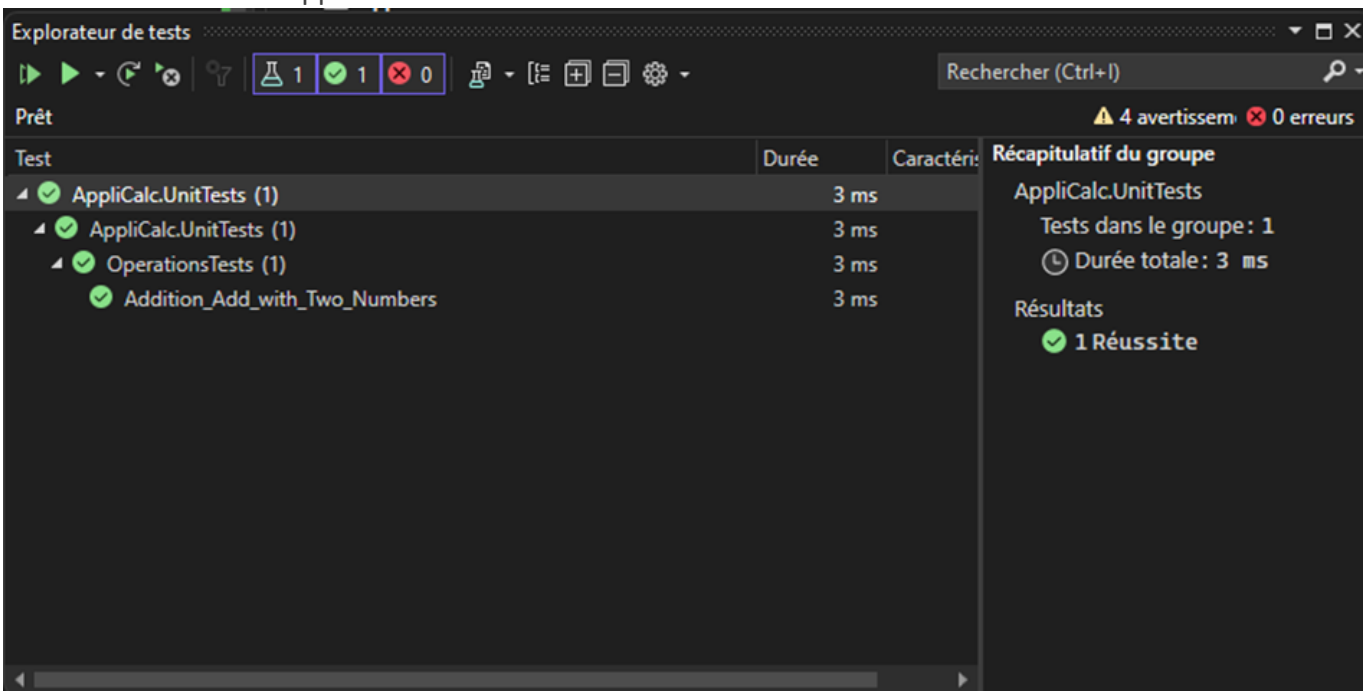
```
[Fact]
0 références
public void Addition_Add_with_Two_Numbers()
{
    // Arrange
    var operation = new Operations();

    // Act
    int result = operation.Addition(1, 3);

    // Assert
    Assert.Equal(4, result);
}
```

Nous aurons l'occasion de revenir plus en détail sur les bonnes pratiques du AAA dans un chapitre suivant.

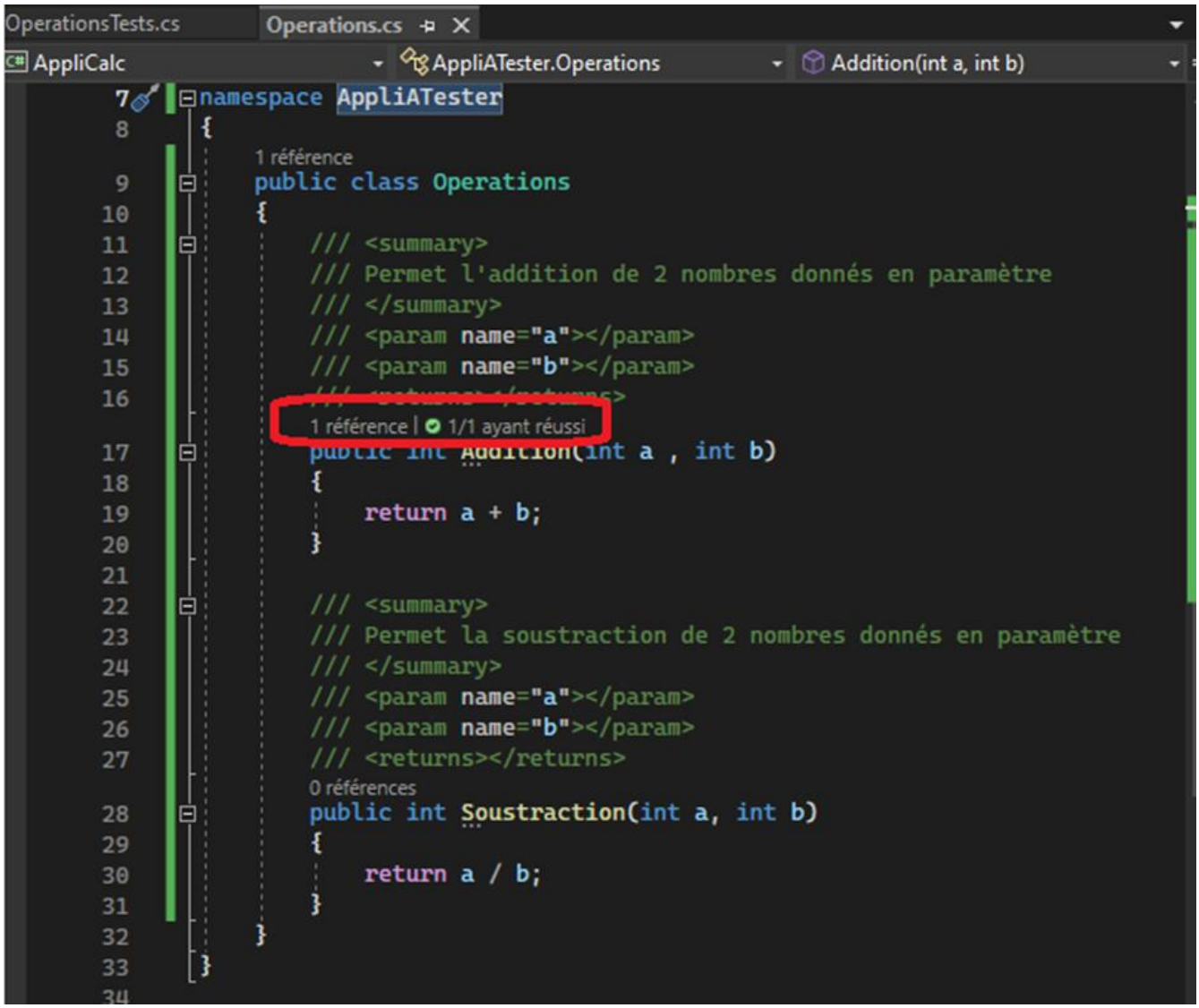
Lancer le test



Une nouvelle fenêtre apparait et donne le résultat des tests



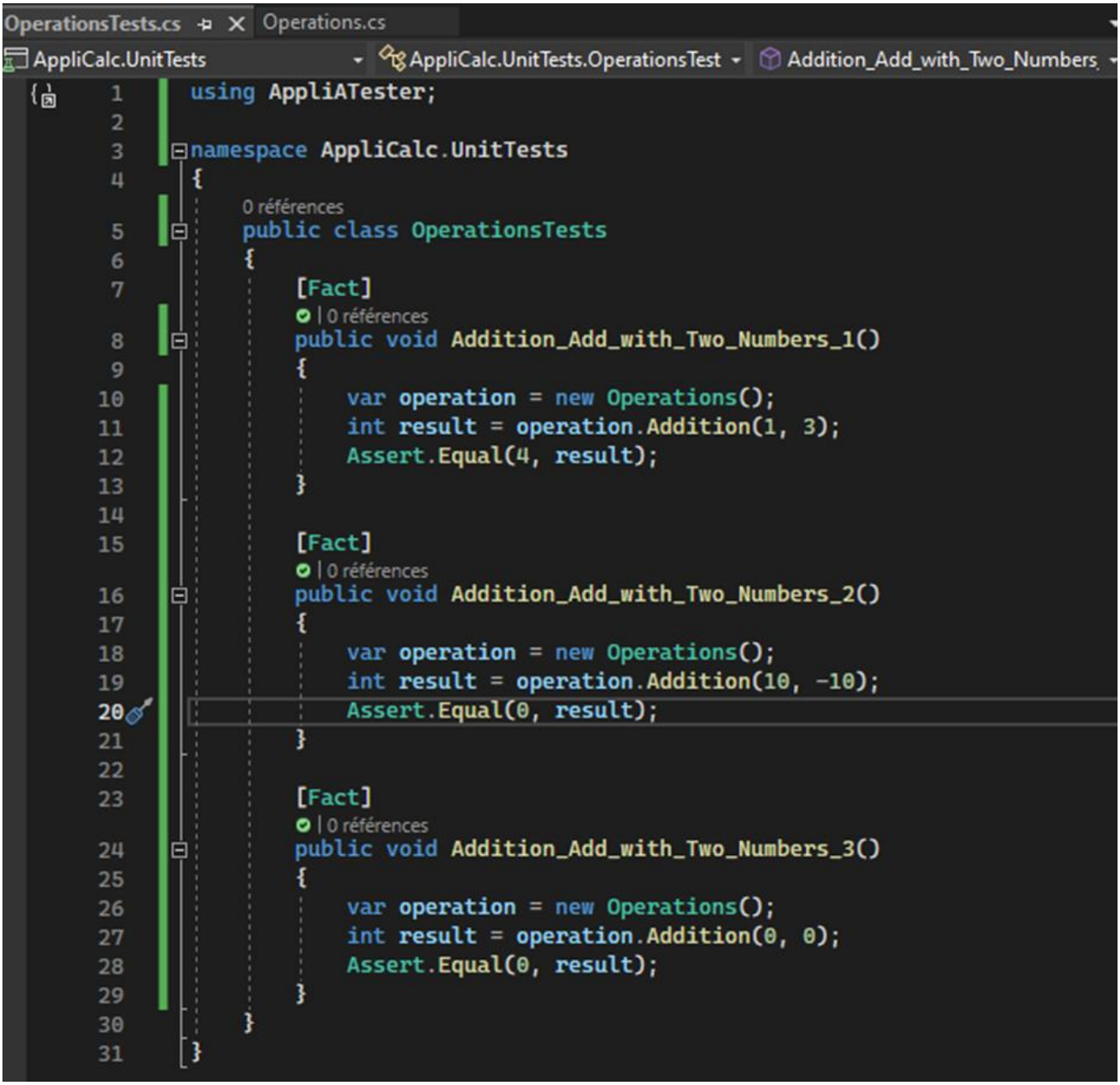
Et au passage, si on retourne sur notre méthode « Addition » qui vient d'être testée, on constate un petit ajout (elle a été testée).


OperationsTests.cs
Operations.cs
AppliCalc
AppliATester.Operations
Addition(int a, int b)
namespace AppliATester
{
1 référence
public class Operations
{
/// <summary>
/// Permet l'addition de 2 nombres donnés en paramètre
/// </summary>
/// <param name="a"></param>
/// <param name="b"></param>
1 référence | 1/1 ayant réussi
public int Addition(int a , int b)
{
return a + b;
}
/// <summary>
/// Permet la soustraction de 2 nombres donnés en paramètre
/// </summary>
/// <param name="a"></param>
/// <param name="b"></param>
/// <returns></returns>
0 références
public int Soustraction(int a, int b)
{
return a / b;
}
}
}

Faire plusieurs fois le même test avec des paramètres différents

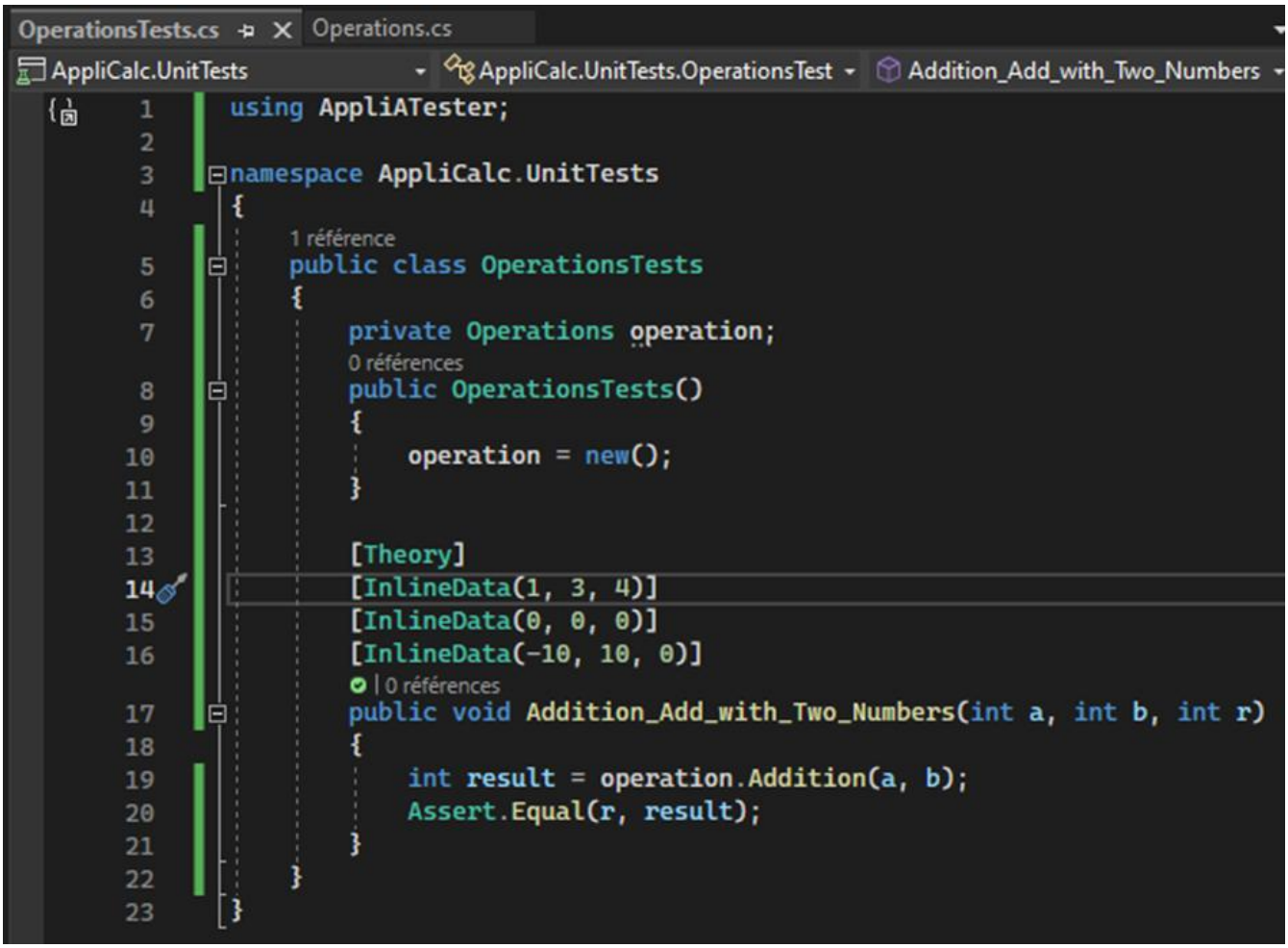
Admettons que nous voulions faire plusieurs fois le même test avec des paramètres différents (peut-être pour tester des valeurs extrêmes ou autres raisons) …

Méthode copier / coller



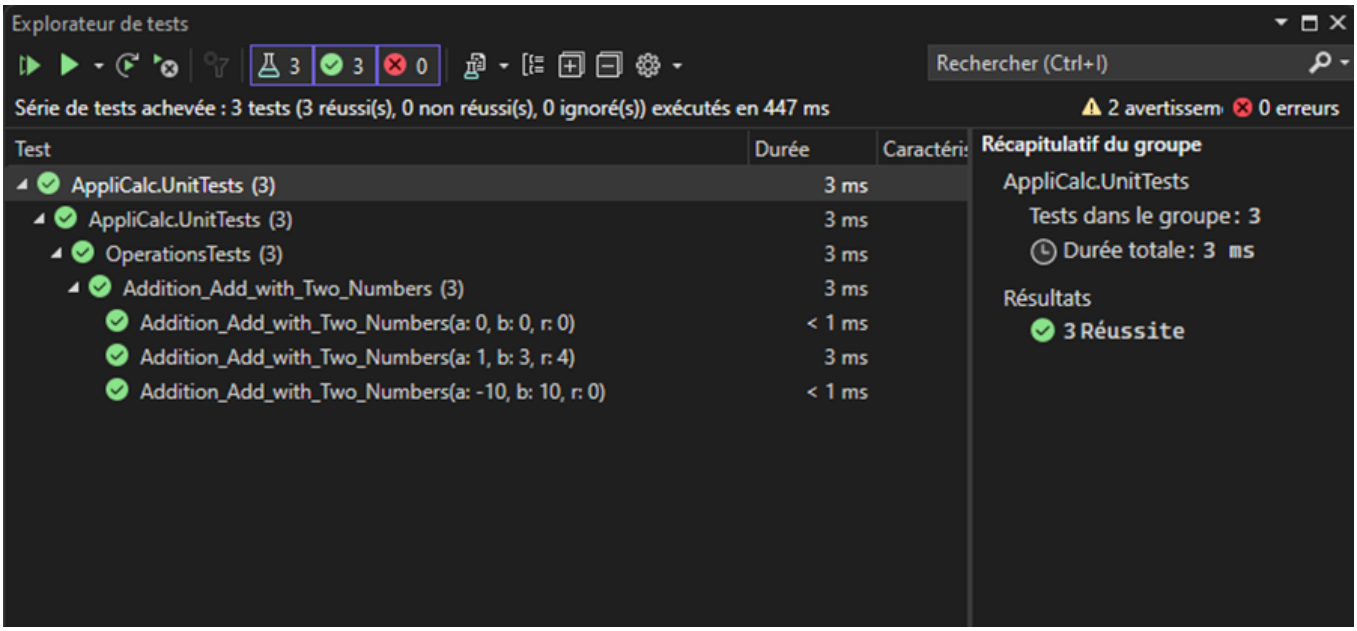
On duplique la méthode autant de fois que nécessaire :’(

Méthode « suite de tests »



[Fact] est remplacé par **[Theory]** è **[Theory]** représente une suite de tests qui exécutent le même code, mais qui ont des arguments d'entrée différents.
L'attribut **[InlineData]** spécifie des valeurs pour ces entrées (les 2 paramètres de la méthode à tester + le résultat attendu).
L'ajout des 3 paramètres dans la méthode de tests.
Comme cette méthode de tests est appelée 3 fois dans ce cas, je décale la création de l'objet «*operation*» dans le constructeur de la classe de tests pour éviter la création de 3 objets «*operation*».

Voici le résultat des tests : il y a bien eu 3 exécutions du même test avec des paramètres diférents



Tester des Exceptions

Tester des exceptions (prévues) n'est pas compliqué ! Voyons quelques exemples.

Division par zéro

Méthode à tester

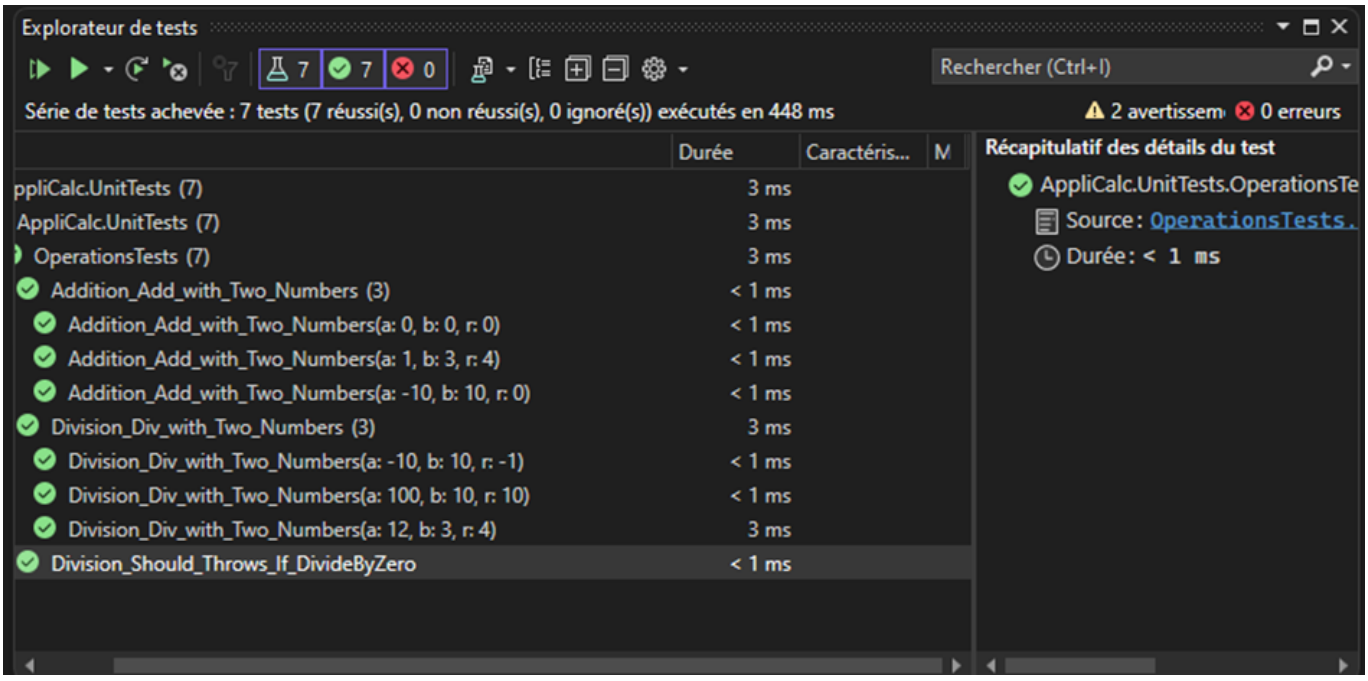
```
        /// <summary>
        /// Permet la division de 2 nombres donnés en paramètre
        /// </summary>
        /// <param name="a"></param>
        /// <param name="b"></param>
        /// <returns></returns>
        0 références
        public int Division(int a, int b)
        {
            return a / b;
        }
    }
```

Comme on ne peut pas comparer le résultat obtenu (il n'y en a pas) avec le résultat attendu (il n'y en a pas non plus), on ne pourra pas utiliser « Assert.Equal »

Méthode de tests (ici, on divise 120 par 0)

```
[Fact]
0 références
public void Division_Should_Throws_If_DivideByZero()
{
    // Arrange
    int a = 120;
    int b = 0;

    // Act
    Action act = () => operation.Division(a, b);

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

Résultat des tests



ArgumentNullException (par exemple)

NB : Ce qui décrit ci-après est valable pour tous les types d'exceptions et non uniquement pour « ArgumentNullException »

Considérons le code à tester suivant

```
/// <summary>
/// Permet l'addition du contenu d'un tableau d'entiers
/// </summary>
/// <param name="tableauEntiers"></param>
/// <returns></returns>
1 référence | 1/1 ayant réussi
public int Addition(int[] tableauEntiers)
{
    if (tableauEntiers == null)
    {
        throw new ArgumentNullException();
    }
    var value = 0;
    foreach (var item in tableauEntiers)
    {
        value += item;
    }
    return value;
}
```

Si le tableau donné en argument, nous arrêtons avec un exception le déroulement de la méthode.

Je fais donc le test suivant pour vérifier la bonne exécution si je donne un tableau d'entiers valide en paramètre

```
[Fact]
| 0 références
public void Addition_Array_Int_with_a_Valid_Array()
{
    // Arrange
    var operation = new Operations();
    int[] tab = new int[5] {1,2,3,4,5};

    // Act
    int result = operation.Addition(tab);

    // Assert
    Assert.Equal(15, result);
}
```

Maintenant je veux tester l'exception elle-même, aussi je crée le test suivant, qui ressemblera fortement au test de l'exemple précédent :

```
[Fact]
| 0 références
public void Addition_Array_Int_with_a_Invalid_Array()
{
    // Arrange
    var operation = new Operations();
    int[] tab = null;

    // Act
    Action act = () => operation.Addition(tab);

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

Le déroulement du test réussi…
Mais maintenant, imaginons que le cahier des charges impose un message d'erreur spécifique !

Notre code de production devient donc ceci

```
public int Addition(int[] tableauEntiers)
{
    if (tableauEntiers == null)
    {
        throw new ArgumentNullException(null, "Un tableau valide est nécessaire");
    }
    var value = 0;
    foreach (var item in tableauEntiers)
    {
        value += item;
    }
    return value;
}
```

NB : on est d'accord, une chaine de texte littéral ne devrait pas être déclarée à ce niveau, mais l'exemple voulait que nous visualisons tous le contenu du texte.

Et bien sûr, nous voulons tester également le contenu du message d'erreur.

Notre test devient donc ceci :

```
[Fact]
| 0 références
public void Addition_Array_Int_with_a_Invalid_Array()
{
    // Arrange
    var operation = new Operations();
    int[] tab = null;
    var txtMessage = "Un tableau valide est nécessaire";

    // Act
    Action act = () => operation.Addition(tab);

    // Assert
    ArgumentNullException exception = Assert.Throws<ArgumentNullException>(act);
    Assert.Equal(txtMessage, exception.Message);
}
```

Et là, nous constatons 2 Assert : un qui retourne un objet de type « ArgumentNullException », et le second qui n'est pas récupéré.

La validité du test inclus forcément que les 2 soient valides.

Comme vous le lirez plus tard dans la partie des bonnes pratiques, il est fortement conseillé d'avoir une seule partie « Act » mais rien ne vous empêche d'avoir plusieurs parties « Assert » (sous réserve qu'elles aient un lien direct et indissociable, comme dans le cas présent où nous avons besoin du premier pour récupérer le message que nous testons ensuite).

Tests en échec

Considérons la méthode suivante

```
4 références | 3/7 ayant réussi
public static string GetNewRandomString(int size)
{
    var chars = "abcdefghijklmnopqrstuvwxyz1234567890";
    var result = new string(Enumerable.Repeat(chars, size).Select(s => s[random.Next(s.Length)]).ToArray());
    return result;
}
```

A ce jour, elle n'est appelée qu'à un seul endroit et le paramètre « size » a une valeur égale à 20.
La méthode retourne donc une chaine de 20 caractères.
Nous créons donc la méthode de tests suivante :

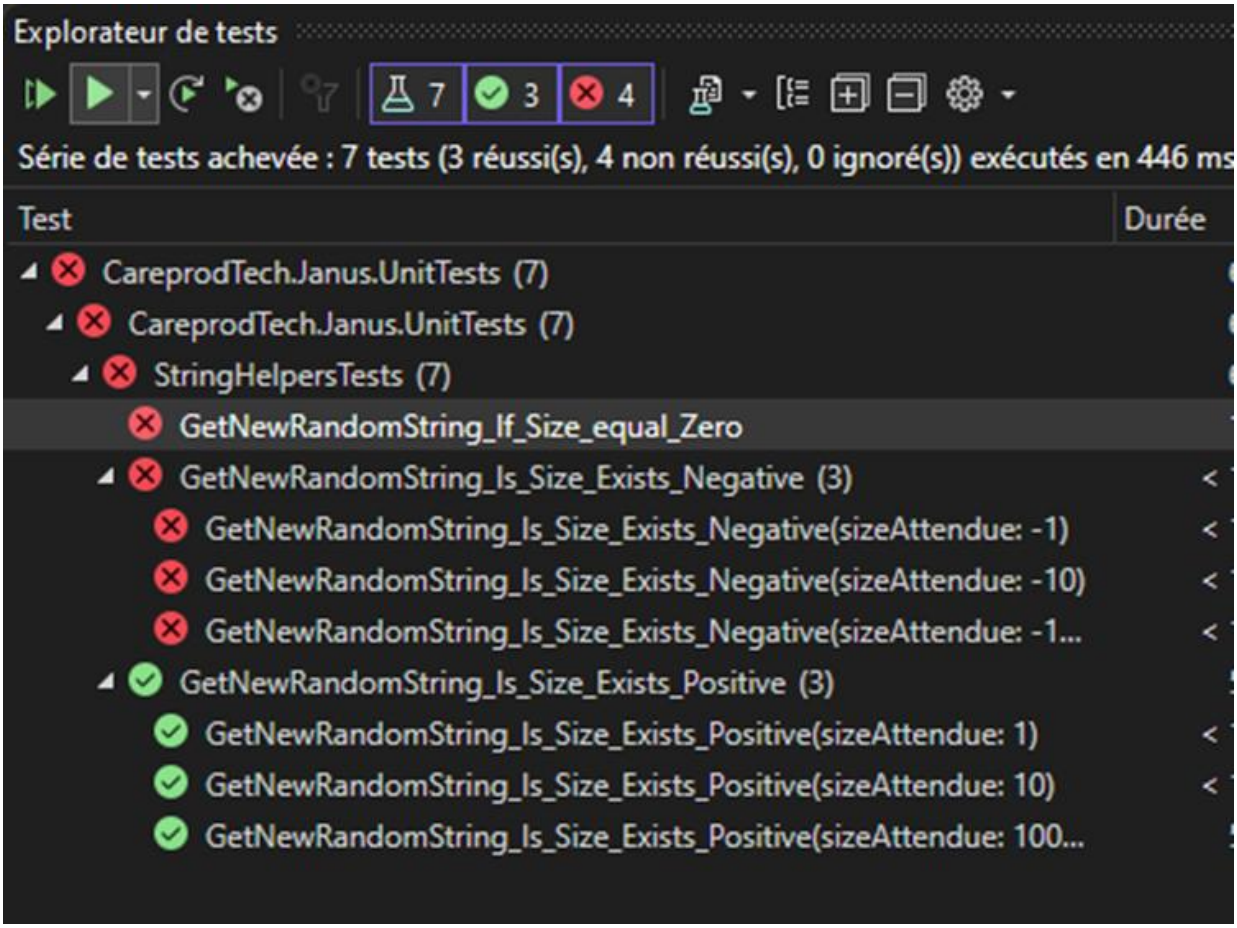
```
/// <summary>
/// Teste GetNewRandomString en injectant des tailles strictement positives
/// Résultat attendu : avoir une chaine d'une taille égale au paramètre
/// </summary>
/// <param name="sizeAttendue"></param>
[Theory]
[InlineData(1)]
[InlineData(10)]
[InlineData(10000)]
| 0 références
public void GetNewRandomString_Is_Size_Exists_Positive(int sizeAttendue)
{
    var sizeObtenue = StringHelpers.GetNewRandomString(sizeAttendue).Length;
    Assert.Equal(sizeAttendue, sizeObtenue);
}
```

Imaginons que demain, quelqu'un développe une nouvelle fonctionnalité qui fera appel à cette méthode mais, dans cette nouveauté, le paramètre ne sera pas fixe, il sera le fruit d'une opération.
Que se passera-t-il si le résultat de l'opération est inférieur ou égal à zéro ?
Dans l'état actuel des choses, la méthode « GetNewRandomString » plantera.
On va donc ajouter une règle de gestion qui stipule que si le paramètre « size » est inférieur ou égal à zéro, la méthode renvoie *null*.

On ajoute donc un test pour un paramètre « size » à zéro, et un test pour un paramètre « size » négatif

```
/// <summary>
/// Teste GetNewRandomString en injectant des tailles strictement négatives
/// Résultat attendu : ne pas avoir de chaine
/// </summary>
/// <param name="sizeAttendue"></param>
[Theory]
[InlineData(-1)]
[InlineData(-10)]
[InlineData(-10000)]
0 références
public void GetNewRandomString_Is_Size_Exists_Negative(int sizeAttendue)
{
    var result = StringHelpers.GetNewRandomString(sizeAttendue);
    Assert.Null(result);
}

/// <summary>
/// Teste GetNewRandomString en injectant une taille à zéro
/// Résultat attendu : ne pas avoir de chaine
/// </summary>
[Fact]
0 références
public void GetNewRandomString_If_Size_equal_Zero()
{
    var result = StringHelpers.GetNewRandomString(0);
    Assert.Null(result);
}
```

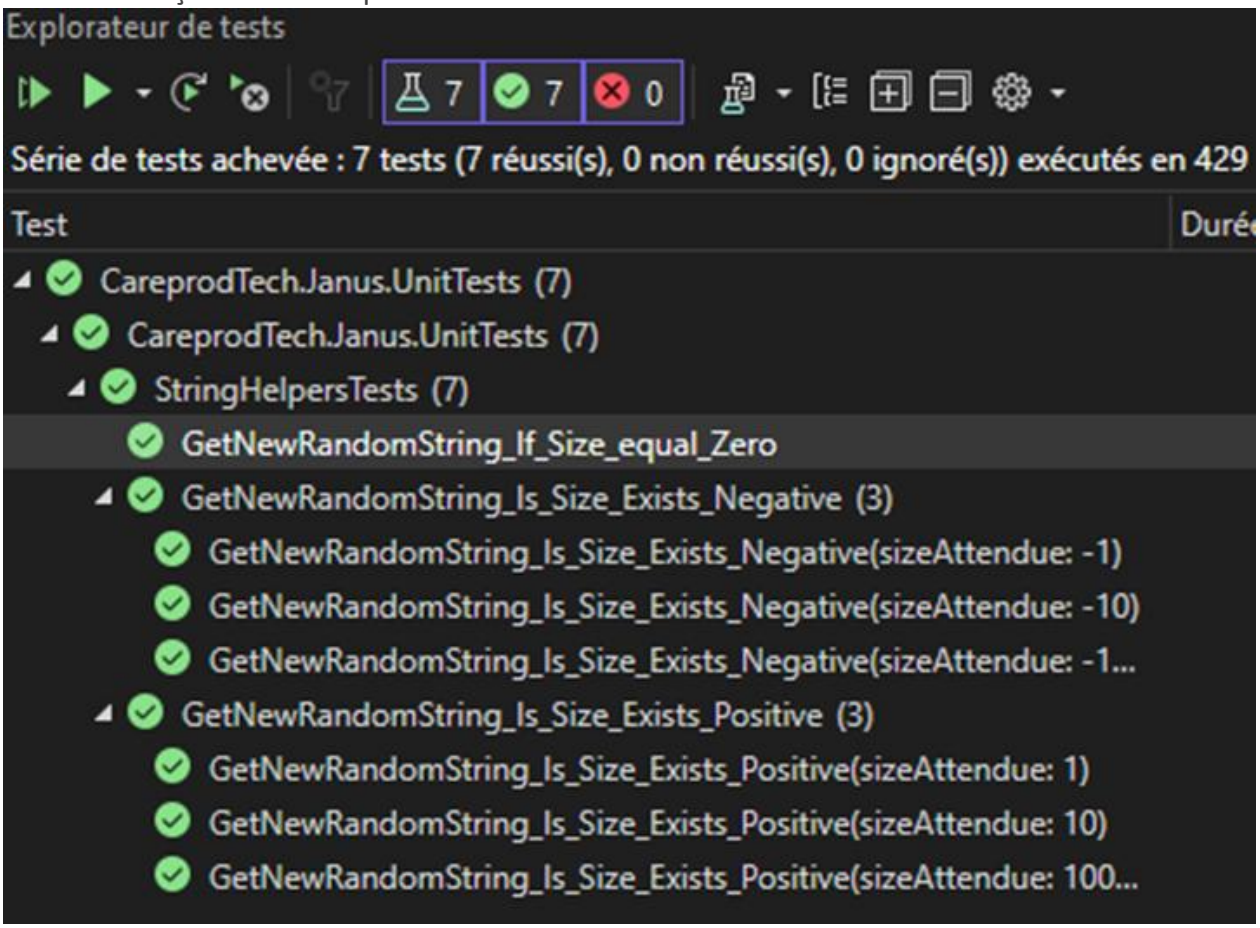
On lance les tests et …



Les nouveaux tests sont en échecs… Pourquoi ? Parce que notre méthode « GetNewRandomString » n'a pas encore été modifiée pour prendre en compte la nouvelle règle de gestion.

Nous la modifions donc

Et nous relançons les tests qui sont tous OK



Cette façon d'écrire les tests AVANT le contenu d'une nouvelle méthode ou d'une modification s'appelle le TDD (Test Driven Developpement).
Les tests tels qu'ils sont mis à profit dans TDD permettent d'explorer et de préciser le besoin, puis de spécifier le comportement souhaité du logiciel en fonction de son utilisation, avant chaque étape de codage. Le logiciel ainsi produit est tout à la fois pensé pour répondre avec justesse au besoin et conçu pour le faire avec une complexité minimale.
On obtient donc un logiciel mieux conçu, mieux testé et plus fiable, autrement dit de meilleure qualité.

Besoin d'un objet au préalable pour pouvoir tester

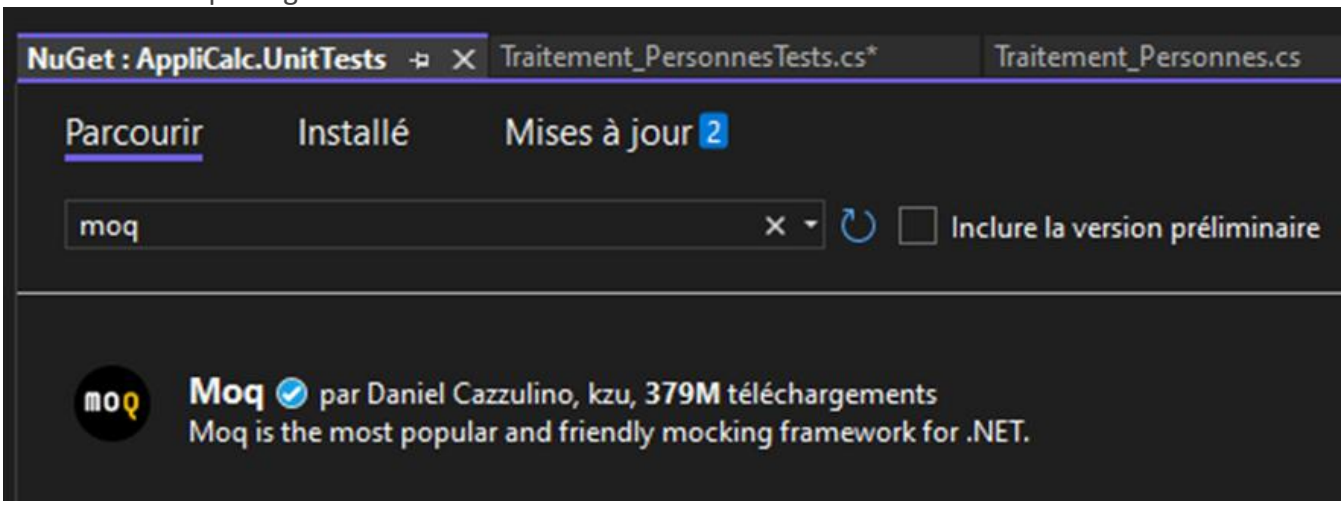
Un test unitaire doit valider de manière isolée une partie du code sans tenir compte de ses dépendances. Dans les exemples précédents, cela se fait de manière aisée puisque nos méthodes ne dépendent d'aucune autre classe.
Dans le cas où notre code à tester a une dépendance vers un objet inaccessible ou non implémenté, il nous faut utiliser un **Mock**. Le Mock est un objet qui permet de simuler le comportement attendu de l'objet dont on dépend afin de se concentrer sur l'objet que l'on souhaite tester.
Allant du constat que bien souvent le développeur dépense beaucoup de temps à initialiser les variables, à préparer un jeu de données ou encore à maintenir la partie Arrange dans une méthode de tests unitaires, la bibliothèque **AutoFixture** nous vient en aide, car elle offre des fonctionnalités intéressantes afin d'alléger le travail à fournir dans cette partie de la méthode de tests.

Bibliothèque « Moq »

Comme dit dans l'introduction de ce chapitre :
Dans le cas où notre code à tester a une dépendance vers un objet inaccessible ou non implémenté, il nous faut utiliser un Mock. Le Mock est un objet qui permet de simuler le comportement attendu de l'objet dont on dépend afin de se concentrer sur l'objet que l'on souhaite tester.

Installer le package « Moq »

Depuis le projet de tests, allons dans le gestionnaire de Nuggets, chercher « Moq » et sélectionner ce package



Utiliser « Moq »

Créons une classe « ArrayManagement » dont la finalité est de récupérer un tableau d'entiers contenu en base de données et la méthode « SumOfItems » retourne la somme de ce tableau d'entiers.

L'interface et la classe gérant la connexion à la base de données :

```
4 références
public interface IDatabaseManagement
{
    2 références
    bool IsConnected();
    1 référence
    void Connect();
    3 références | ✓ 1/1 ayant réussi
    int[] GetItemsFromDatabase();
}
0 références
public class DatabaseManagement : IDatabaseManagement
{
    3 références | ✓ 1/1 ayant réussi
    public int[] GetItemsFromDatabase()...

    2 références
    public bool IsConnected()...

    1 référence
    public void Connect()...

}
```

La méthode « GetItemsFromDatabase » retourne le tableau d'entiers présent en base de données.

La classe « ArrayManagement » et sa méthode « SumOfItems » que nous allons tester :

```
2 références
public class ArrayManagement
{
    IDatabaseManagement _databaseManagement;
    1 référence | 1/1 ayant réussi
    public ArrayManagement(IDatabaseManagement databaseManagement)
    {
        _databaseManagement = databaseManagement;
    }
    /// <summary>
    /// Retourne la somme d'un tableau d'entiers trouvé en BDD
    /// </summary>
    /// <returns></returns>
    /// <exception cref="ArgumentNullException"></exception>
    1 référence | 1/1 ayant réussi
    public int SumOfItems()
    {
        var items = _databaseManagement.GetItemsFromDatabase();
        if (items == null)
        {
            throw new ArgumentNullException();
        }
        var value = 0;
        foreach (var item in items)
        {
            value += item;
        }
        return value;
    }
}
```

Problématique :

La classe créée a besoin dans son constructeur d'un objet implémentant IDatabaseManagement.

Nous sommes dans le cadre de tests unitaires, aussi nous ne nous connecterons pas à une base de données (là, nous serions en tests d'intégration) mais simuler cette connexion grâce à un Mock.

Création du test :

```
using Moq;

[Fact]
0 références
public void SumOfItems_ThenReturnSumOfEachItems()
{
    // Arrange
    // -------

    // Création d'un objet MOCK se basant sur IDatabaseManagement
    var databaseManagement = new Mock<IDatabaseManagement>();

    // Simulation de l'utilisation de la méthode "GetItemsFromDatabase" qui retourne un tableau
    // d'entier que nous maitrisons
    databaseManagement.Setup(x => x.GetItemsFromDatabase()).Returns(new int[5] { 3, 4, 5, 7, 10 });

    // Création d'un objet qui implémente "ArrayManagement" en lui envoyant comme paramètre l'objet
    // de type IDatabaseManagement issu de l'objet MOCK
    var arrayManagement = new ArrayManagement(databaseManagement.Object);

    // Définition de la valeur attendue pour le test
    var expectedValue = 29;

    // Act
    // ---
    // Appel de la méthode que nous testons
    var actualValue = arrayManagement.SumOfItems();

    // Assert
    // ------
    // Vérification du résultat
    Assert.Equal(expectedValue, actualValue);
}
```

Donc nous venons de simuler la création d'un objet qui implémente *IDatabaseManagement* pour nos besoins de tests unitaires.

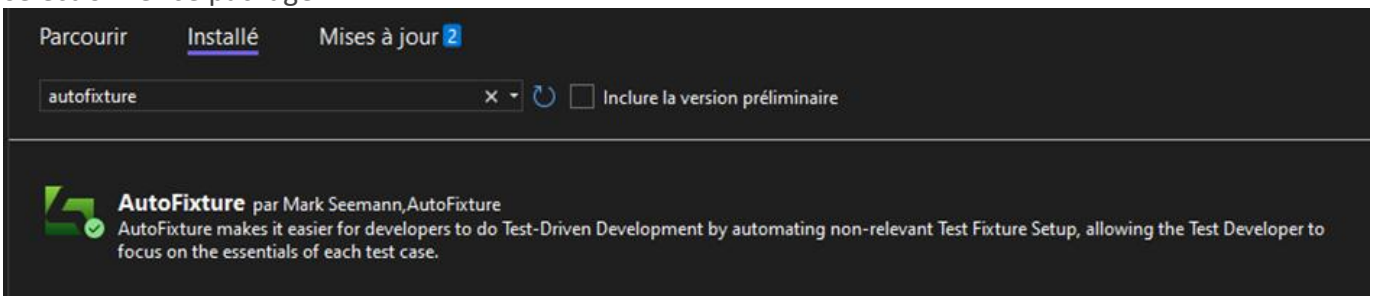
NB : Mock utilise les « override » pour simuler les méthodes aussi, il est impératif de lui donner en paramètre des interfaces ou des classes abstraites.

Bibliothèque AutoFixture

Comme dit dans l'introduction de ce chapitre :
Allant du constat que bien souvent le développeur dépense beaucoup de temps à initialiser les variables, à préparer un jeu de données ou encore à maintenir la partie Arrange dans une méthode de tests unitaires, la bibliothèque Autofixture nous vient en aide, car elle offre des fonctionnalités intéressantes afin d'alléger le travail à fournir dans cette partie de la méthode de tests.

Installer « AutoFixture »

Depuis le projet de tests, allons dans le gestionnaire de Nuggets, chercher « AutoFixture » et sélectionner ce package



Utiliser « AutoFixture »

Le but de cette bibliothèque est d'alimenter les propriétés d'un objet.

Classe « Employee » qui sera utilisée par la méthode à tester

```
4 références
public class Employee
{
    1 référence
    public string EmployeeId { get; set; }
    1 référence
    public string FirstName { get; set; }
    1 référence
    public string LastName { get; set; }
    5 références | 15/15 ayant réussi
    public char Grade { get; set; }

    [Range(0, 50)]
    6 références | 15/15 ayant réussi
    public int ServiceYears { get; set; }


    0 références
    public Employee(string employeeId, string firstName, string lastName, char grade, int serviceYears)
    {
        EmployeeId = employeeId;
        FirstName = firstName;
        LastName = lastName;
        Grade = grade;
        ServiceYears = serviceYears;
    }
}
```

La classe / méthode à tester : « GetPrime »

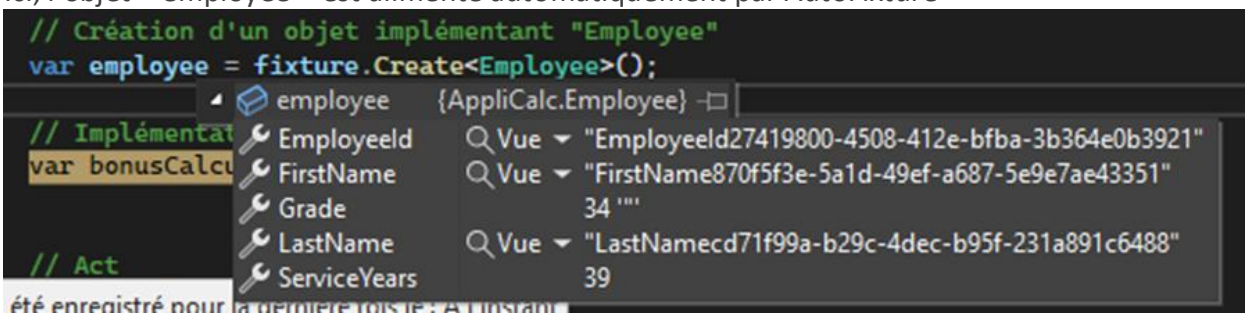
Elle détermine une prime en fonction du grade et de l'ancienneté

```
2 références
public class BonusCalculator
{
    1 référence
    public int MttBase { get; } = 1000;

    2 références | ● 16/16 ayant réussi
    public float GetPrime(Employee employee)
    {
        int mttBase = this.MttBase;
        int prime = 0;

        if (employee.ServiceYears > 0)
        {
            if (employee.Grade == 'A')
            {
                prime = employee.ServiceYears >= 5 ? (mttBase * 3) + mttBase : (mttBase * 3);
            }
            else
            {
                if (employee.Grade == 'B')
                {
                    prime = employee.ServiceYears >= 5 ? (mttBase * 2) + mttBase : (mttBase * 2);
                }
                else
                {
                    if (employee.Grade == 'C')
                    {
                        prime = employee.ServiceYears >= 5 ? (mttBase * 1) + mttBase : (mttBase * 1);
                    }
                    else
                    {
                        prime = 0;
                    }
                }
            }
        }
        return prime;
    }
}
```

La première version de notre test :

```
using AutoFixture;

[Fact]
0 références
public void GetPrime_Mtt_Positif()
{
    // Arrange
    // -------

    // Implémentation de AutoFixture
    var fixture = new Fixture();

    // Création d'un objet implémentant "Employee"
    var employee = fixture.Create<Employee>();

    // Implémentation de la classe qui contient la méthode à tester
    var bonusCalculator = new BonusCalculator();


    // Act
    // ---

    // Appel de la méthode à tester
    var actualValue = bonusCalculator.GetPrime(employee);

    // Assert
    // ------

    // Resultat du test
    Assert.True(actualValue >= 0);
}
```

Ici, l'objet « employee » est alimenté automatiquement par AutoFixture



Imposer des valeurs.

Dans notre méthode « GetPrime » que nous testons, nous voulons imposer des grades et des anciennetés puisque les primes sont en fonction de ces 2 critères.
Nous pourrions imposer la création d'un objet avec, par exemple, le grade à 'A' et l'ancienneté à 5 ans :

```
var employee = fixture.Build<Employee>().With(x => x.ServiceYears, 5).With(x => x.Grade, 'A').Create();
```

Mais cela nous obligerait à créer autant de méthodes de tests que de cas à tester. Nous allons donc utiliser la façon de faire « suite de tests » que nous avons vu plus tôt.

```
[Theory]
[InlineData('A', 0, 0)]
[InlineData('A', 1, 3000)]
[InlineData('A', 4, 3000)]
[InlineData('A', 5, 4000)]
[InlineData('A', 9, 4000)]

[InlineData('B', 0, 0)]
[InlineData('B', 1, 2000)]
[InlineData('B', 4, 2000)]
[InlineData('B', 5, 3000)]
[InlineData('B', 9, 3000)]

[InlineData('C', 0, 0)]
[InlineData('C', 1, 1000)]
[InlineData('C', 4, 1000)]
[InlineData('C', 5, 2000)]
[InlineData('C', 9, 2000)]

0 références
public void GetPrime_ReturnValueGoodValue(char grade, int anciennete, int resultatAttendu)
{
    // Arrange
    // -------

    // Implémentation de AutoFixture
    var fixture = new Fixture();

    // Création d'un objet implémentant "Employee"
    var employee = fixture.Build<Employee>().With(x => x.ServiceYears, anciennete).With(x => x.Grade, grade).Create();

    // Implémentation de la classe qui contient la méthode à tester
    var bonusCalculator = new BonusCalculator();


    // Act
    // ---

    // Appel de la méthode à tester
    var actualValue = bonusCalculator.GetPrime(employee);

    // Assert
    // ------

    // Resultat du test
    Assert.True(actualValue == resultatAttendu);
}
```

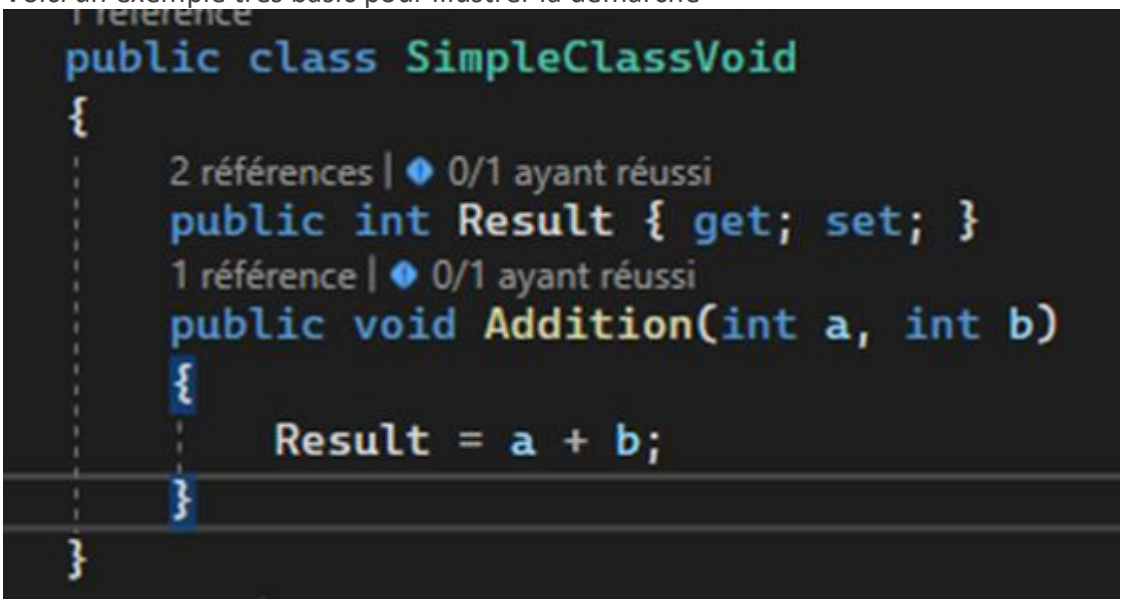
Ainsi nous venons de tester plusieurs cas avec autant d'objets « employee » générés que nécessaire, en imposant les valeurs qui nous intéressent.
NB : évidemment, le jour où le Montant de base changera (BonusCalculator.MttBase), les valeurs littérales de nos tests devront évoluer.

Comment tester des méthodes « void » ?

Jusqu'à présent, nous avons récupéré la valeur d'une méthode pour tester le résultat.
Que faire avec des méthodes « void » ?
Généralement, une méthode « void », même si elle ne retourne aucune méthode, change l'état d'une ou plusieurs propriétés, voir d'un ou plusieurs objets.
Il suffit donc de capter les valeurs qui ont été modifiées et de les comparer avec un résultat attendu.

Exemple très basic

Voici un exemple très basic pour illustrer la démarche



Et la méthode de tests qui en découle

```
[Fact]
| 0 références
public void Addition_with_two_valid_numbers()
{
    //Arrange
    var myClass = new SimpleClassVoid();
    int attendu = 8;

    //Act
    myClass.Addition(3, 5);

    //Assert
    Assert.Equal(myClass.Result, attendu);
}
```

Autre exemple de code de production

Voici un code plus réaliste

```
public class TransactionViewModel : ITransactionViewModel
{
    7 références | 1/1 ayant réussi
    public bool IsDeposit { get; set; }
    6 références | 1/1 ayant réussi
    public DateTime? TransactionDate { get; set; }

    private ITransactionRepository _transactionRepository;
    1 référence | 1/1 ayant réussi
    public TransactionViewModel(ITransactionRepository transactionRepo)
    {
        _transactionRepository = transactionRepo;
    }

    2 références
    public Transaction GetById(int id)
    {
        return _transactionRepository.GetById(id);
    }

    3 références | 1/1 ayant réussi
    public void Initialize(string id)
    {
        IsDeposit = true;

        if (string.IsNullOrEmpty(id) || !int.TryParse(id, out int transactionId))
        {
            IsDeposit = false;
            TransactionDate = null;
            return;
        }
        var selectedTransaction = GetById(transactionId);
        if(selectedTransaction == null)
        {
            IsDeposit = false;
            TransactionDate = null;
            return;
        }
        IsDeposit = selectedTransaction.Status == Constants.TRANSACTION_DEPOSITED;
        TransactionDate = selectedTransaction.TransactionDate;
    }
}
```

Nous allons créer la méthode de test pour la partie où l'Id n'est pas un Id valable

```
[Fact]
0 références
public void Initialize_sets_transaction_date_null_when_id_cant_be_parsed()
{
    // Arrange
    TransactionViewModel viewModel = new(new TransactionRepository());

    // Act
    viewModel.Initialize("id_qui_est_alphanumerique");

    // Assert
    Assert.False(viewModel.IsDeposit);
    Assert.Null(viewModel.TransactionDate);
}
```

Dernier exemple de code de production (avec Mock)

Voici le code de production en plusieurs parties

Une classe de type entité

```
public class Person
{
    4 références
    public int Id { get; set; }
    5 références | 1/1 ayant réussi
    public string Nom { get; set; } = "";
    5 références | 1/1 ayant réussi
    public string Prenom { get; set; } = "";
    2 références | 1/1 ayant réussi
    public DateTime DateNaissance { get; set; }
}
```

Une classe de connexion à une base de données

```
public class MyDbContext : DbContext
{
    3 références
    public DbSet<Person> Persons { get; set; } = default!;

    0 références
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)...

    0 références
    protected override void OnModelCreating(ModelBuilder modelBuilder)...
}
```

Une interface de type Repository et la classe qui l'implémente

```
1 référence
public class PersonRepository : IPersonRepository
{
    private readonly MyDbContext ctx;

    0 références
    public PersonRepository(MyDbContext ctx)
    {
        this.ctx = ctx;
    }

    4 références | 1/1 ayant réussi
    public void Add(Person unePerson)
    {
        ctx.Persons.Add(unePerson);
        ctx.SaveChanges();
    }

    2 références
    public IEnumerable<Person> GetAll()
        => ctx.Persons.ToList();

    2 références
    public Person? GetById(int id)
        => ctx.Persons.SingleOrDefault(p => p.Id == id);
}
```

Et enfin, une classe de type services qui va s'appuyer sur la repository

```
public class PersonService
{
    private readonly IPersonRepository repository;
    1 référence | 1/1 ayant réussi
    public PersonService(IPersonRepository repository)
    {
        this.repository = repository;
    }

    0 références
    public IEnumerable<Person> GetAll()
        => repository.GetAll();

    0 références
    public Person? GetById(int id)
        => repository.GetById(id);

    1 référence | 1/1 ayant réussi
    public void Add(Person person)
        => repository.Add(person);
}
```

Nous voulons maintenant tester la méthode « Add » de la classe « PersonService ».

Arrange

Le constructeur de la classe « PersonService » nécessite en paramètre un objet repository. Mais ce dernier faisant un appel à une connexion de la base de données, nous serons dépendant du microcosme et donc, nous ne serons plus dans le cadre d'un test unitaire (nous serions en test d'intégration).

Donc, comme vu précédemment, nous allons simuler l'objet repository grâce à Mock et nous initialiserons le service grâce à ce mock.

La méthode service.Add attendant un objet « Person » en tant que paramètre, nous allons également créer cet objet en laissant AutoFixture le remplir.

Puis, nous allons également déclarer un objet de type « Person », ce sera un objet témoin qui sera alimenté par la méthode repository.Add au moment où cette dernière sera appelée (appelée par service.Add).

Et enfin, nous allons préparer la simulation de l'appel de la méthode repository.Add grâce à l'objet Mock créé juste avant. Cette simulation renverra dans l'objet témoin le contenu de ce repository.Add a reçu en paramètre.

Act

Nous appellerons la méthode service.Add

Assert

Nous vérifierons que la méthode repository.Add a bien été appelé

Nous vérifierons que l'objet témoin n'est pas null

Nous vérifierons que les propriétés de l'objet témoin sont bien identiques à l'objet envoyé à service.Add

Ce qui donne ceci :

```
[Fact]
0 références
public async Task Add_People_To_Db_Should_Call_Add_Repo()
{
    // Nous allons tester la méthode Add de la classe PersonService qui s'appuie sur PersonRepository

    // Arrange
    // -------
    Mock<IPersonRepository> personRepoMock = new(); // On simule la création d'un objet repository
    var service = new PersonService(personRepoMock.Object); ; // on cré une instance du service
    Fixture fixture = new(); // on inialise une instance Fixture
    var randomPerson = fixture.Create<Person>(); // on crée un objet (de Person) qui sera rempli par
                                                 // l'outil Fixture qui sera envoyé à service.Add
    Person? p = null; // on crée un objet p et l'initialise à null
    // Préparation de la simulation
    personRepoMock
        .Setup(m => m.Add(It.IsAny<Person>())) // on prépare la simulation de l'appel de la méthode Add du repository
        .Callback((Person unePersonne) => p = unePersonne); // paramètre de la méthode Add (du repository).
                                                            // Le paramètre est "unePersonne"
                                                            // et "p" prendra sa valeur lors de l'appel de la méthode


    // Act
    // ---
    // Jusqu'ici, p est null car personRepoMock.Add n'a pas encore été appelée.
    // Elle le sera quand service.Add sera appelée.
    // Si tout est OK, en passant dans service.Add,            // p sera normalement égal à randomPerson
    service.Add(randomPerson); // on fait appel à la méthode Add du service
                               // (qui s'appuie sur un repo Mock)

    // Assert
    // ------
    // On vérifie bien que la méthode Add du repository (simulé) a été appelée une fois
    personRepoMock.Verify(m => m.Add(It.IsAny<Person>()), Times.Exactly(1));

    // On vérifie que l'objet "p" n'est pas null
    Assert.NotNull(p);

    // On vérifie que les propriété sont identiques
    Assert.Equal(p.Nom, randomPerson.Nom);
    Assert.Equal(p.Prenom, randomPerson.Prenom);
    Assert.Equal(p.DateNaissance, randomPerson.DateNaissance);
    // ou tout simplement
    Assert.Equal(p, randomPerson);

    /*
    Si tous ces tests sont OK alors on sait
        1. Que la méthode service.Add fait bien appel à repository.Add
        2. Que la méthode repository.Add a bien reçu un objet "Person" valide de la part de service.Add
        2. Que l'objet "Person" envoyé à service.Add est le même que celui qui a été transmis à repository.Add
    */
}
```

Cas des classes / méthodes *internal*

Pour que notre projet de tests voit les classes / méthodes internal, il nous faudra faire une petite manipulation.
Il faut modifier le fichier « .csproj » du projet qui abrite la classe / méthode *internal* à tester et intégrer ce bloc dans le bloc **<ItemGroup></ItemGroup>**

```
<AssemblyAttribute Include="System.Runtime.CompilerServices.InternalsVisibleTo">
    <_Parameter1>[redacted].Magewell.UnitTests</_Parameter1>
</AssemblyAttribute>
```

Ce qui donne ceci :

```
<ItemGroup>                                                                    ">
    <PackageReference Include="Newtonsoft.Json" Version="13.0.2" />
    <AssemblyAttribute Include="System.Runtime.CompilerServices.InternalsVisibleTo">
        <_Parameter1>[redacted].Magewell.UnitTests</_Parameter1>
    </AssemblyAttribute>
</ItemGroup>
```

Dans le bloc **<_Parameter1></_Parameter1>**, on indique le nom du projet de tests qui va tester nos classes / méthodes *internal.*

Les bonnes pratiques du AAA (Arrange / Act / Assert)

Nommage de vos tests

Comme déjà dit plus haut, le nom de votre test doit être composé de trois parties :

- Nom de la méthode testée
- Scénario de test utilisé
- Comportement attendu quand le scénario est appelé

Pas bien !

```
[Fact]
public void Test_Single()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

Bien !

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

Organisation du test

Séparons bien les 3 étapes
Pourquoi ?
Permet de séparer clairement ce qui est testé des étapes organisation et assertion.
Moins de risques de mélanger les assertions avec le code de l'étape « Action ».

Pas bien !

```
[Fact]
public void Add_EmptyString_ReturnsZero()
{
    // Arrange
    var stringCalculator = new StringCalculator();

    // Assert
    Assert.Equal(0, stringCalculator.Add(""));
}
```

Bien !

```
[Fact]
public void Add_EmptyString_ReturnsZero()
{
    // Arrange
    var stringCalculator = new StringCalculator();

    // Act
    var actual = stringCalculator.Add("");

    // Assert
    Assert.Equal(0, actual);
}
```

Écrire des tests concluants minimaux

L'entrée à utiliser dans un test unitaire doit être la plus simple possible afin de vérifier le comportement que vous testez actuellement.
Pourquoi ?
Les tests deviennent plus résilients face aux futurs changements du code base.
Plus proche du comportement de test que de l'implémentation.
Les tests qui contiennent plus d'informations que nécessaire pour être réussis ont plus de chances d'introduire des erreurs et peuvent rendre l'intention moins claire.
Lorsque vous écrivez des tests, vous souhaitez vous concentrer sur le comportement.
La définition de propriétés supplémentaires pour les modèles ou l'utilisation de valeurs différentes de zéro quand cela n'est pas nécessaire, ne fait que nuire à ce que vous essayez de prouver.

Pas bien !

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("42");

    Assert.Equal(42, actual);
}
```

Bien !

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

Éviter les chaînes magiques

Nommer des variables dans les tests unitaires est important, sinon plus important, que de nommer des variables dans le code de production. Les tests unitaires ne doivent pas contenir de chaînes magiques.
Pourquoi ?
Évite au lecteur du test d'inspecter le code de production pour déterminer ce qui rend la valeur spéciale.
Montre explicitement ce que vous essayez de prouver et non ce que vous essayez d'accomplir.
Les chaînes magiques peuvent être sources de confusion pour le lecteur de vos tests. Si une chaîne sort de l'ordinaire, ils peuvent se demander pourquoi une certaine valeur a été choisie pour un paramètre ou une valeur de retour.
Ce type de valeur de chaîne peut les amener à examiner de plus près les détails de l'implémentation, plutôt que de se concentrer sur le test.

Pas bien !

```
[Fact]
public void Add_BigNumber_ThrowsException()
{
    var stringCalculator = new StringCalculator();

    Action actual = () => stringCalculator.Add("1001");

    Assert.Throws<OverflowException>(actual);
}
```

Bien !

```
[Fact]
void Add_MaximumSumResult_ThrowsOverflowException()
{
    var stringCalculator = new StringCalculator();
    const string MAXIMUM_RESULT = "1001";

    Action actual = () => stringCalculator.Add(MAXIMUM_RESULT);

    Assert.Throws<OverflowException>(actual);
}
```

Éviter la logique dans les tests

Lors de l'écriture de vos tests unitaires, évitez la concaténation manuelle des chaînes, les conditions logiques, telles que *if*, *while*, *for*, et *switch*...

Pourquoi ?

Moins de risques d'introduire un bogue dans vos tests.

Concentrez-vous sur le résultat final plutôt que sur les détails de l'implémentation.

Quand vous introduisez une logique dans votre suite de tests, le risque d'introduction d'un bogue augmente considérablement.

Votre suite de tests est bien le dernier endroit où vous devez trouver un bogue.

Vous devez avoir un niveau élevé de confiance dans le fonctionnement de vos tests, sinon, vous ne leur faites pas confiance. Les tests auxquels vous n'avez pas confiance ne fournissent aucune valeur.

Lorsqu'un test échoue, vous souhaitez avoir l'impression que quelque chose ne va pas dans votre code et qu'il ne peut pas être ignoré.

Conseil : Si le recours à la logique dans votre test semble inévitable, scindez-le en deux ou plusieurs tests distincts.

Pas bien !

```
[Fact]
public void Add_MultipleNumbers_ReturnsCorrectResults()
{
    var stringCalculator = new StringCalculator();
    var expected = 0;
    var testCases = new[]
    {
        "0,0,0",
        "0,1,2",
        "1,2,3"
    };

    foreach (var test in testCases)
    {
        Assert.Equal(expected, stringCalculator.Add(test));
        expected += 3;
    }
}
```

Bien !

```
[Theory]
[InlineData("0,0,0", 0)]
[InlineData("0,1,2", 3)]
[InlineData("1,2,3", 6)]
public void Add_MultipleNumbers_ReturnsSumOfNumbers(string input, int expected)
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add(input);

    Assert.Equal(expected, actual);
}
```

Éviter plusieurs actes

Lors de l'écriture de vos tests, essayez d'inclure un seul acte par test.
Les approches courantes de l'utilisation d'un seul acte sont les suivantes :
Créez un test distinct pour chaque acte.
Utilisez des tests paramétrables.

Pourquoi ?

Lorsque le test échoue, il est clair quel acte échoue.
Garantit que le test est axé sur un seul cas.
Vous donne une idée complète des causes de l'échec des tests.
Plusieurs actes doivent être affirmés individuellement et il n'est pas garanti que tous les asserts seront exécutés. Dans la plupart des frameworks de test unitaire, une fois qu'une assertion échoue dans un test unitaire, les tests en cours sont automatiquement considérés comme ayant échoué. Ce type de processus peut prêter à confusion, car les fonctionnalités qui fonctionnent réellement, seront affichées comme défaillantes.

Pas bien !

```
[Fact]
public void Add_EmptyEntries_ShouldBeTreatedAsZero()
{
    // Act
    var actual1 = stringCalculator.Add("");
    var actual2 = stringCalculator.Add(",");

    // Assert
    Assert.Equal(0, actual1);
    Assert.Equal(0, actual2);
}
```

Bien !

```
[Theory]
[InlineData("", 0)]
[InlineData(",", 0)]
public void Add_EmptyEntries_ShouldBeTreatedAsZero(string input, int expected)
{
    // Arrange
    var stringCalculator = new StringCalculator();

    // Act
    var actual = stringCalculator.Add(input);

    // Assert
    Assert.Equal(expected, actual);
}
```

Valider les méthodes privées en effectuant un test unitaire des méthodes publiques

Dans la plupart des cas, il n'est pas nécessaire de tester une méthode privée. Les méthodes privées sont un détail d'implémentation et n'existent jamais isolément.
À un moment donné, il y aura une méthode publique qui appelle la méthode privée dans le cadre de son implémentation. Vous devez prendre en compte le résultat final de la méthode publique qui appelle la méthode privée.

Références statiques

L'un des principes d'un test unitaire est qu'il doit avoir le contrôle total du système testé. Ce principe peut être problématique lorsque le code de production inclut des appels à des références statiques (par exemple, *DateTime.Now*).
Considérez le code suivant :

```
public int GetDiscountedPrice(int price)
{
    if (DateTime.Now.DayOfWeek == DayOfWeek.Tuesday)
    {
        return price / 2;
    }
    else
    {
        return price;
    }
}
```

Comment ce code peut-il faire l'objet d'un test unitaire ? Vous pouvez essayer une approche telle que :

```
public void GetDiscountedPrice_NotTuesday_ReturnsFullPrice()
{
    var priceCalculator = new PriceCalculator();

    var actual = priceCalculator.GetDiscountedPrice(2);

    Assert.Equals(2, actual)
}

public void GetDiscountedPrice_OnTuesday_ReturnsHalfPrice()
{
    var priceCalculator = new PriceCalculator();

    var actual = priceCalculator.GetDiscountedPrice(2);

    Assert.Equals(1, actual);
}
```

Malheureusement, vous vous rendrez rapidement compte qu'il existe quelques problèmes avec vos tests.
Si la suite de tests est exécutée un mardi, le second test est une réussite, mais le premier test est un échec.
Si la suite de tests est exécutée un autre jour, le premier test est une réussite, mais le second test est un échec.
Pour résoudre ces problèmes, vous allez devoir modifier votre code de production.
Il existe une approche qui consiste à inclure dans un wrapper le code que vous devez contrôler dans une interface, et à faire en sorte que le code de production dépende de cette interface.

```
public interface IDateTimeProvider
{
    DayOfWeek DayOfWeek();
}

public int GetDiscountedPrice(int price, IDateTimeProvider dateTimeProvider)
{
    if (dateTimeProvider.DayOfWeek() == DayOfWeek.Tuesday)
    {
        return price / 2;
    }
    else
    {
        return price;
    }
}
```

Votre suite de tests se présente désormais comme suit :

```
public void GetDiscountedPrice_NotTuesday_ReturnsFullPrice()
{
    var priceCalculator = new PriceCalculator();
    var dateTimeProviderStub = new Mock<IDateTimeProvider>();
    dateTimeProviderStub.Setup(dtp => dtp.DayOfWeek()).Returns(DayOfWeek.Monday);

    var actual = priceCalculator.GetDiscountedPrice(2, dateTimeProviderStub);

    Assert.Equals(2, actual);
}

public void GetDiscountedPrice_OnTuesday_ReturnsHalfPrice()
{
    var priceCalculator = new PriceCalculator();
    var dateTimeProviderStub = new Mock<IDateTimeProvider>();
    dateTimeProviderStub.Setup(dtp => dtp.DayOfWeek()).Returns(DayOfWeek.Tuesday);

    var actual = priceCalculator.GetDiscountedPrice(2, dateTimeProviderStub);

    Assert.Equals(1, actual);
}
```

Désormais, la suite de tests a un contrôle total sur `DateTime.Now` et peut créer une simulation de n'importe quelle valeur au moment d'appeler la méthode.

Couverture du code

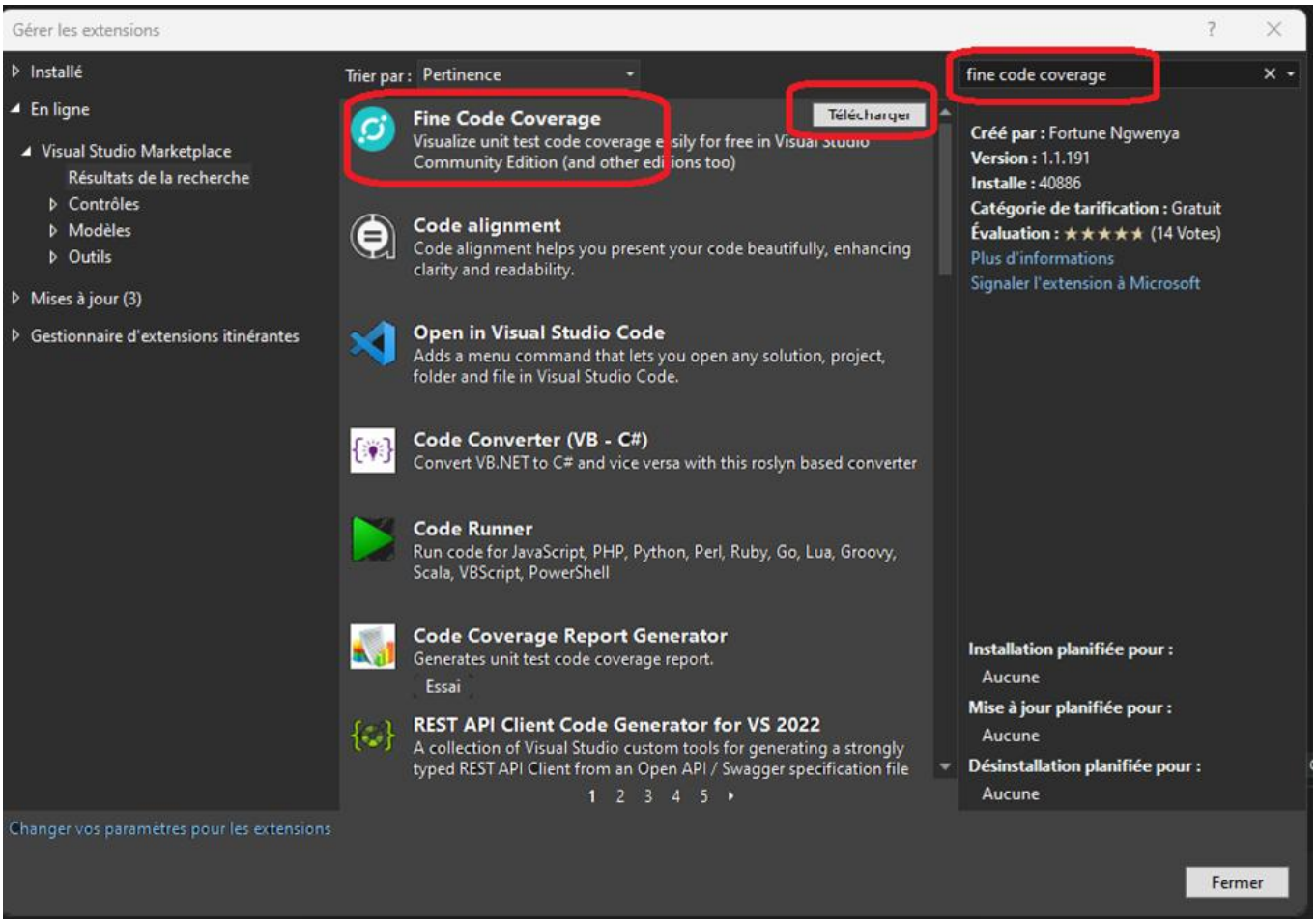
Un pourcentage élevé de couverture du code est souvent associé à une qualité supérieure du code. Toutefois, la mesure elle-même *ne peut pas* déterminer la qualité du code.
La définition d'un objectif de pourcentage de couverture du code trop ambitieux peut être contre-productive. Imaginez un projet complexe avec des milliers de branches conditionnelles, et imaginez que vous définissez un objectif de couverture de code de 95 %. Actuellement, le projet maintient une couverture du code de 90 %. Le temps nécessaire pour tenir compte de tous les cas de périphérie dans les 5 % restants pourrait être une entreprise massive, et la proposition de valeur diminue rapidement.
Un pourcentage élevé de couverture du code n'est pas un indicateur de réussite et n'implique pas non plus une qualité élevée du code. Il représente simplement la quantité de code couverte par les tests unitaires.

Outil pour mesurer la couverture du code

Dans la longue liste des outils disponibles, nous pourrions retenir « Fine Code Coverage ». Il s'agit d'une extension pour Visual Studio.

Installer l'extension

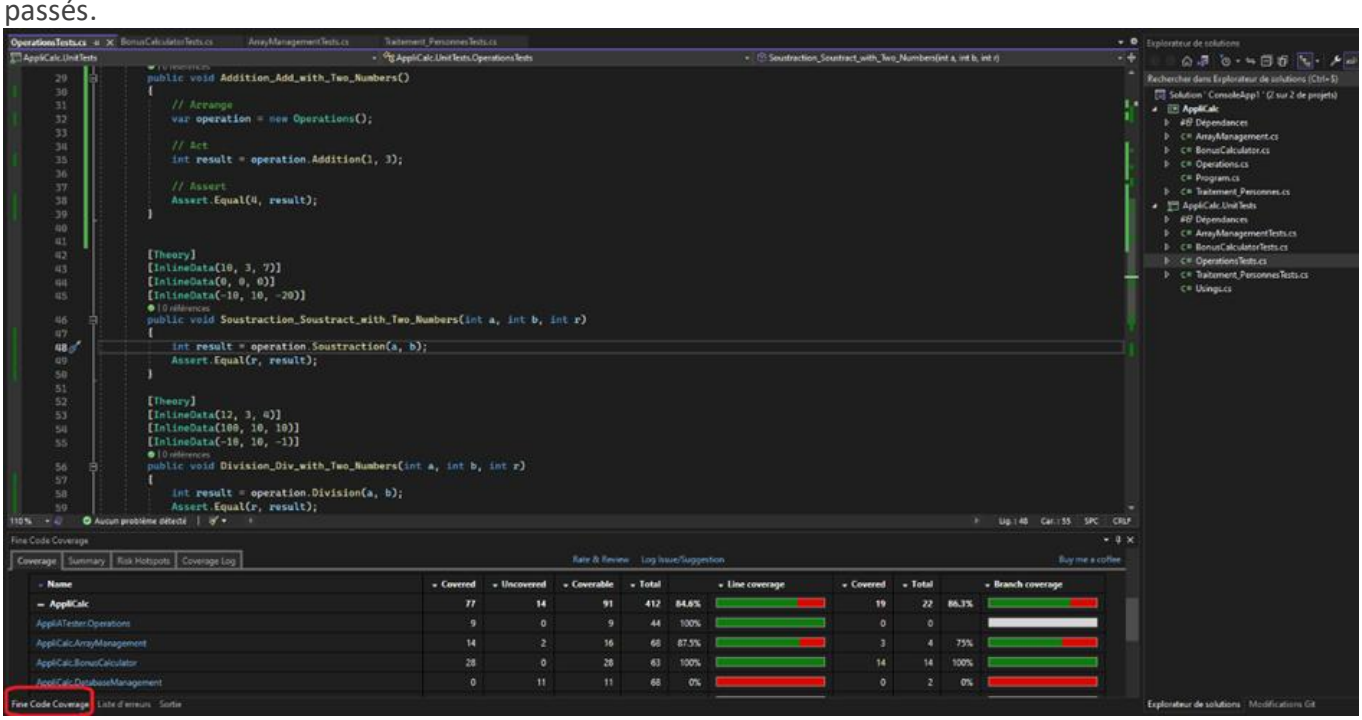
Visual Studio / Extension / Gérer les extensions
Nous saisissons « fine code coverage » dans la barre de recherche et sélectionnons l'extension et cliquons sur « Télécharger »



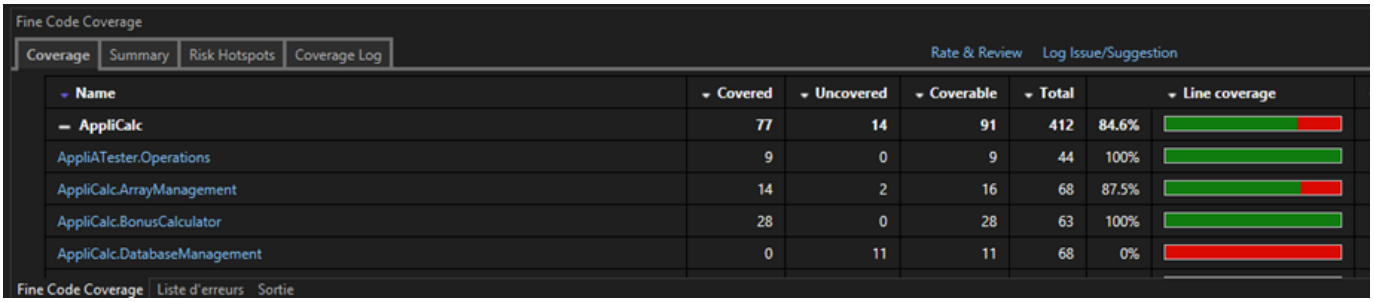
Maintenant, « Fine Code Coverage » est installé sur Visual Studio donc est actif pour tous les projets qui implémentent des tests unitaires automatisés.

Utiliser Fine Code Coverage

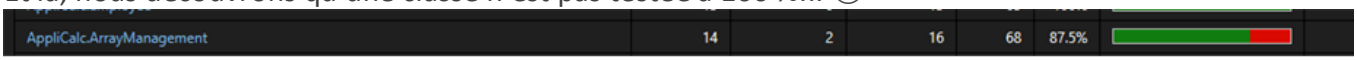
Nous lançons notre batterie de tests et une nouvelle fenêtre apparait une fois les tests passés.



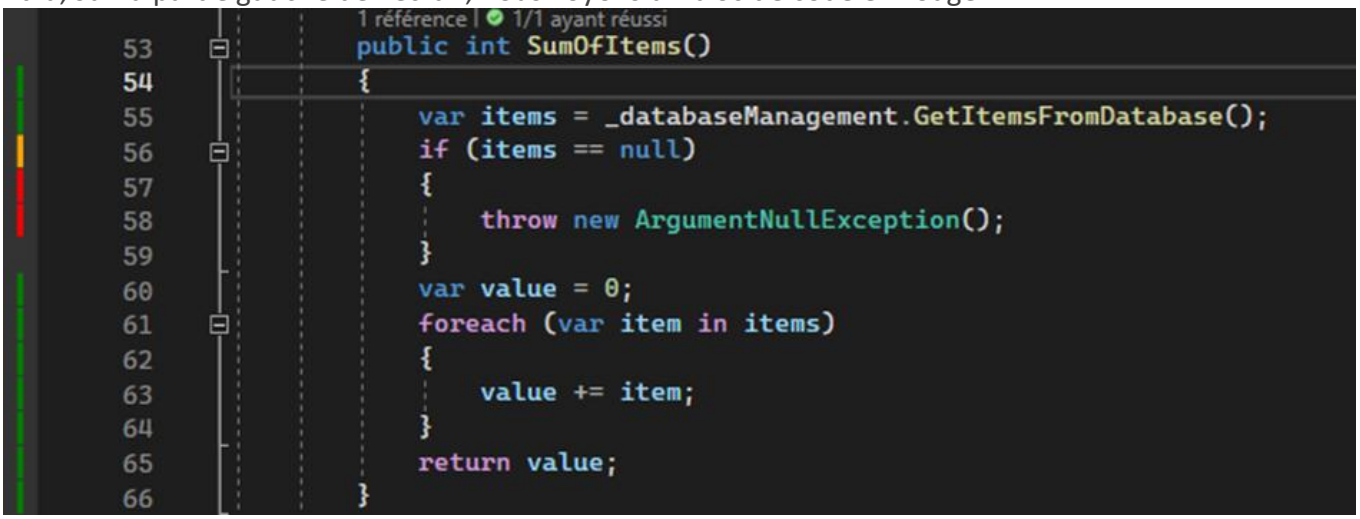
Petit zoom…



Et là, nous découvrons qu'une classe n'est pas testée à 100 %... ☹



Il s'agit de ArrayManagement… Allons voir cette classe !
Et là, sur la partie gauche de l'écran, nous voyons un bloc de code en rouge



Il s'agit de la partie qui gère la nullité des items… Nous ne testons pas ce qui se passe en cas de nullité.
Nous créons donc un nouveau test :

```
[Fact]
0 références
public void SumOfItems_ThenReturnNull()
{
    // Arrange
    // -------

    // Création d'un objet MOCK se basant sur IDatabaseManagement
    var databaseManagement = new Mock<IDatabaseManagement>();

    // Simulation de l'utilisation de la méthode "GetItemsFromDatabase" qui retourne un tableau
    // d'entier que nous maitrisons --> ici, ce sera un tableau NULL
    int[] tab = null;
    databaseManagement.Setup(x => x.GetItemsFromDatabase()).Returns(tab);

    // Création d'un objet qui împlémente "ArrayManagement" en lui envoyant comme paramètre l'objet
    // de type IDatabaseManagement issu de l'objet MOCK
    var arrayManagement = new ArrayManagement(databaseManagement.Object);

    // Act
    // ---
    // Appel de la méthode que nous testons
    Action act = () => arrayManagement.SumOfItems();

    // Assert
    // ------
    // Vérification du résultat
    Assert.Throws<ArgumentNullException>(act);
}
```

Nous lançons notre test…
100% de couverture pour la classe ArrayManagement

AppliCalc.ArrayManagement	16	0	16	68	100%

Et la méthode concernée apparait en vert dans sa totalité 😊

```
2 références | 2/2 ayant réussi
53      public int SumOfItems()
54      {
55          var items = _databaseManagement.GetItemsFromDatabase();
56          if (items == null)
57          {
58              throw new ArgumentNullException();
59          }
60          var value = 0;
61          foreach (var item in items)
62          {
63              value += item;
64          }
65          return value;
66      }
67  }
```

Attention, certaines classes ou méthodes étant intestable par nature (techniquement ou fonctionnellement parlant), il existe un attribut permettant de les exclure de l'analyse de couverture du code.

Le voici :

```
11
12      namespace Lynks.SignalR.Server.Tools;
13      +
14      +[ExcludeFromCodeCoverage]
            2 références
15      public static class ServerConfHelper
16      {
17
                1 référence
18          public static IServiceCollection AddLynksSignalRConfiguration(this IServ
19          {
20              services.AddSignalR(hubOptions =>
```

[ExcludeFromCodeCoverage]

Un exemple de TDD pas à pas

La théorie

Le cahier des charges :

Créer une méthode qui renvoie une chaine de caractères.
Cette méthode reçoit 2 arguments de type « string », le premier est une chaine initiale et le second donne une taille. La méthode renvoie les n premiers caractères de la chaine initiale, n correspondant à la taille donnée en paramètre.
En cas de soucis, une exception sera déclenchée, avec un message explicit.
Nous allons donc créer cette méthode en TDD.

Premier pas :

On imagine les premiers tests (après lecture des règles de gestion) :
La méthode doit retourner une valeur
le paramètre *tailleCible* doit être convertible en entier
le paramètre *tailleCible*, une fois convertit en entier, doit être strictement > 0
le paramètre *maChaineInitiale* doit être d'une taille >= *tailleCible*

Deuxième pas :

On crée une méthode minimaliste.

```
0 références
public string CreerChaine(string maChaineInitiale, string tailleCible)
{
    return "";
}
```

Troisième pas :

On écrit le premier test, on exécute, on adapte le code de production pour que le test soit OK, on reteste et quand c'est OK, on passe au test suivant… Et ainsi de suite jusqu'à ce que tous les tests soient OK.

La mise en pratique

Premier test

On vérifie que la méthode retourne une valeur quand les paramètres sont OK

```
[Fact]
0 références
public void CreerChaine_Etape1_Verifie_si_retourne_une_chaine_quand_tout_est_correct()
{
    // Arrange
    var o = new TDD_Exemple();
    string maChaine = "abcde";
    string size = "3";
    string resultatAttendu = "abc";

    // Act
    var resultObtenu = o.CreerChaine(maChaine, size);

    // Assert
    Assert.Equal(resultatAttendu, resultObtenu);
}
```

Evidemment, ce test va échouer donc nous allons modifier notre méthode de production pour que ce test soit OK.

```
1 référence | 0/1 ayant réussi
public string CreerChaine(string maChaineInitiale, string tailleCible)
{
    int size = int.Parse(tailleCible);
    string result = maChaineInitiale.Substring(0, size);
    return result;
}
```

En relançant notre test, on verra qu'il est OK.

Second test

On vérifie que la méthode retourne une exception si le paramètre « taille » n'est pas convertible en entier (on va envoyer des lettres et une chaine vide)

```
[Theory]
[InlineData("AAA")]
[InlineData("")]
0 références
public void CreerChaine_Etape2_Verifie_si_Exception_quand_tailleCible_pas_convertible(string size)
{
    // Arrange
    var o = new TDD_Exemple();
    string maChaine = "abcde";

    // Act
    Action act = () => o.CreerChaine(maChaine, size);

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

Evidemment, ce test va échouer donc nous allons modifier notre méthode de production pour que ce test soit OK.

```
1 référence | 0/1 ayant réussi
public string CreerChaine(string maChaineInitiale, string tailleCible)
{
    bool isNumeric = int.TryParse(tailleCible, out int size);
    if (!isNumeric)
    {
        throw new Exception();
    }
    string result = maChaineInitiale.Substring(0, size);
    return result;
}
```

En relançant notre test, on verra qu'il est OK.

Troisième test

On vérifie que la méthode retourne une exception si le paramètre « taille » n'est pas convertible en entier ET que le message d'erreur soit explicit.

```
[Theory]
[InlineData("AAA")]
[InlineData("")]
0 références
public void CreerChaine_Etape3_Verifie_si_Exception_et_Message_explicit_quand_tailleCible_pas_convertible(string size)
{
    // Arrange
    var o = new TDD_Exemple();
    string maChaine = "abcde";

    // Act
    Action act = () => o.CreerChaine(maChaine, size);

    // Assert
    Exception exception = Assert.Throws<Exception>(act);
    Assert.Equal(MessageUtilisateur.Taille_Cible_NON_NUMERIQUE, exception.Message);
}
```

Evidemment, ce test va échouer donc nous allons modifier notre méthode de production pour que ce test soit OK.

```
public string CreerChaine(string maChaineInitiale, string tailleCible)
{
    bool isNumeric = int.TryParse(tailleCible, out int size);
    if (!isNumeric)
    {
        throw new Exception(MessageUtilisateur.Taille_Cible_NON_NUMERIQUE);
    }
    string result = maChaineInitiale.Substring(0, size);
    return result;
}
```

Et on ajoute des messages explicits

```
12 références
public static class MessageUtilisateur
{
    public static string Taille_Cible_NON_NUMERIQUE = "Le paramètre tailleCible n'est pas un numérique";
    public static string Taille_Cible_PAS_POSITIF = "Le paramètre tailleCible doit être strictement positif";
    public static string Taille_Cible_TROP_GRAND = "Le paramètre tailleCible ne doit pas dépasser la taille du paramètre maChaineInitiale";
}
```

En relançant notre test, on verra qu'il est OK.

Quatrième test

On vérifie que la méthode retourne une exception si le paramètre « taille », une fois convertit en entier, est égal ou inférieur à zéro (et on vérifie le message d'erreur).

```
[Theory]
[InlineData("0")]
[InlineData("-1")]
0 références
public void CreerChaine_Etape4_Verifie_si_Exception_et_Message_explicit_quand_tailleCible_negatif_ou_zero(string size)
{
    // Arrange
    var o = new TDD_Exemple();
    string maChaine = "abcde";

    // Act
    Action act = () => o.CreerChaine(maChaine, size);

    // Assert
    Exception exception = Assert.Throws<Exception>(act);
    Assert.Equal(MessageUtilisateur.Taille_Cible_PAS_POSITIF, exception.Message);
}
```

Evidemment, ce test va échouer donc nous allons modifier notre méthode de production pour que ce test soit OK.

```
5 références | 1/5 ayant réussi
public string CreerChaine(string maChaineInitiale, string tailleCible)
{
    bool isNumeric = int.TryParse(tailleCible, out int size);
    if (!isNumeric)
    {
        throw new Exception(MessageUtilisateur.Taille_Cible_NON_NUMERIQUE);
    }
    if (size <= 0)
    {
        throw new Exception(MessageUtilisateur.Taille_Cible_PAS_POSITIF);
    }
    string result = maChaineInitiale.Substring(0, size);
    return result;
}
```

En relançant notre test, on verra qu'il est OK.

Cinquième test

On vérifie que la méthode retourne une exception si le paramètre « taille », une fois convertit en entier, est supérieur à la taille du paramètre « chaine initiale » (et on vérifie le message d'erreur).

```
[Fact]
0 références
public void CreerChaine_Etape5_Verifie_si_Exception_et_Message_explicit_quand_tailleCible_plus_grand_que_chaine_initiale()
{
    // Arrange
    var o = new TDD_Exemple();
    string maChaine = "abcde";
    string size = "10";

    // Act
    Action act = () => o.CreerChaine(maChaine, size);

    // Assert
    Exception exception = Assert.Throws<Exception>(act);
    Assert.Equal(MessageUtilisateur.Taille_Cible_TROP_GRAND, exception.Message);
}
```

Evidemment, ce test va échouer donc nous allons modifier notre méthode de production pour que ce test soit OK.

```
7 références | 1/7 ayant réussi
public string CreerChaine(string maChaineInitiale, string tailleCible)
{
    bool isNumeric = int.TryParse(tailleCible, out int size);
    if (!isNumeric)
    {
        throw new Exception(MessageUtilisateur.Taille_Cible_NON_NUMERIQUE);
    }
    if (size <= 0)
    {
        throw new Exception(MessageUtilisateur.Taille_Cible_PAS_POSITIF);
    }
    if (size > maChaineInitiale.Length)
    {
        throw new Exception(MessageUtilisateur.Taille_Cible_TROP_GRAND);
    }
    string result = maChaineInitiale.Substring(0, size);
    return result;
}
```

En relançant notre test, on verra qu'il est OK.

Sixième test

J'ai un doute !!!

Je n'ai pas testé le cas où la chaine initiale est vide… ou alors je ne crois pas l'avoir testé !

Normalement, le cinquième test devrait suffire, mais pour en avoir le cœur net, je modifie mon test.

```
[Theory]
[InlineData("abcde", "10")]
[InlineData("","1")]
0 références
public void CreerChaine_Etape5_Verifie_si_Exception_et_Message_explicit_quand_tailleCible_plus_grand_que_chaine_initiale(string maChaine, string size)
{
    // Arrange
    var o = new TDD_Exemple();

    // Act
    Action act = () => o.CreerChaine(maChaine, size);

    // Assert
    Exception exception = Assert.Throws<Exception>(act);
    Assert.Equal(MessageUtilisateur.Taille_Cible_TROP_GRAND, exception.Message);
}
```

Et effectivement, le test est OK.

Cela suffit pour garantir la solidité de notre code de production.

Toutefois, le message d'erreur peut ne pas convenir (bien que le test soit mathématiquement vrai) --> à creuser !

- Si cela convient à tout le monde, on en reste là
- Si cela ne convient pas, on modifie notre code pour tenir compte de ce cas précis avec un message explicit et on crée un nouveau test.

Comme cela ne convient pas de laisser en l'état, on modifie le code de production

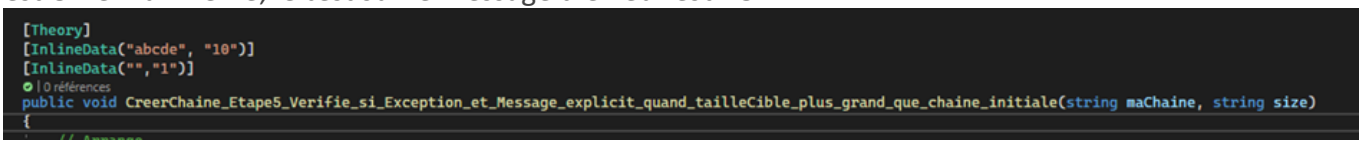
```
6 références | 9/10 ayant réussi
public string CreerChaine(string maChaineInitiale, string tailleCible)
{
    bool isNumeric = int.TryParse(tailleCible, out int size);
    if (!isNumeric)
    {
        throw new Exception(MessageUtilisateur.Taille_Cible_NON_NUMERIQUE);
    }
    if (size <= 0)
    {
        throw new Exception(MessageUtilisateur.Taille_Cible_PAS_POSITIF);
    }
    if (size > maChaineInitiale.Length)
    {
        if (maChaineInitiale.Length == 0)
        {
            throw new Exception(MessageUtilisateur.Chaine_Initiale_VIDE);
        }
        else
        {
            throw new Exception(MessageUtilisateur.Taille_Cible_TROP_GRAND);
        }
    }
    string result = maChaineInitiale.Substring(0, size);
    return result;
}
```

Là, nous ajoutons une nouvelle condition de façon imbriquée à celle ajoutée lors du test précédent
Et le test correspondant

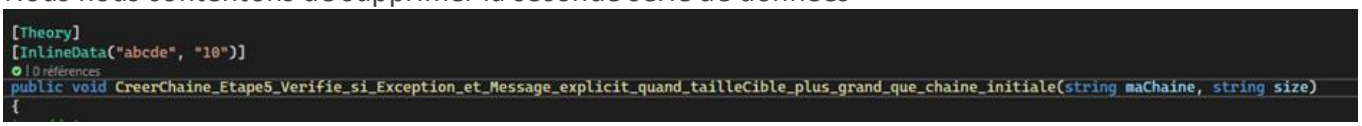
```
[Fact]
0 références
public void CreerChaine_Etape6_Verifie_si_Exception_et_Message_explicit_quand_chaineInitiale_est_vide()
{
    // Arrange
    var o = new TDD_Exemple();
    string maChaine = "";
    string size = "10";

    // Act
    Action act = () => o.CreerChaine(maChaine, size);

    // Assert
    Exception exception = Assert.Throws<Exception>(act);
    Assert.Equal(MessageUtilisateur.Chaine_Initiale_VIDE, exception.Message);
}
```

Quand nous lançons les tests, le test est OK... Mais…

Un des cas du test précédent est KO !!!
En effet, en lui injectant une série de données, nous lui injectons une chaine vide et si le test est OK en lui-même, le test sur le message d'erreur est KO.

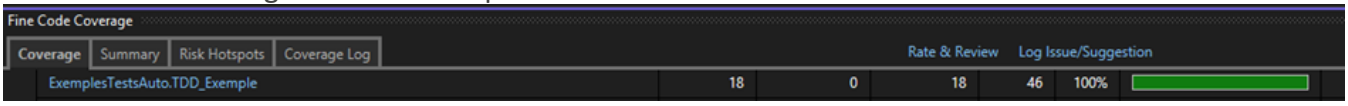


Nous nous contentons de supprimer la seconde série de données



Et là, tous les tests sont OK !

Et Fine Code Coverage me confirme que tout est couvert !



Happy end !

En savoir plus

Tests unitaires avec xUnit

https://learn.microsoft.com/fr-fr/dotnet/core/testing/unit-testing-best-practices
https://learn.microsoft.com/fr-fr/dotnet/core/testing/unit-testing-with-dotnet-test
https://hamidmosalla.com/category/xunit/
https://nacoumblesoro.developpez.com/tutoriel/xunit/apprendre-faire-tests-unitaires/#LII-1

Mock

https://github.com/Moq/moq4/wiki/Quickstart
http://www.blackwasp.co.uk/SearchResults.aspx?page=1&q=mock

AutoFixture

https://nacoumblesoro.developpez.com/tutoriel/xunit/apprendre-faire-tests-unitaires-xunit-partie2-autofixture/#LIII-A

Fine Code Coverage

Visualize unit test code coverage easily in Visual Studio Community Edition