A close-up photograph of a blue plastic tray containing numerous small, clear plastic test tubes. A pink plastic pipette is positioned vertically in the center of the tray, with its tip pointing downwards towards the tubes. The background is blurred, suggesting a laboratory setting.

# Tests unitaires et C#

# Objectifs

- Retour sur les types de données en C#
- Approche des tests unitaires :
  - Description
  - Réalisation en C#

# Types de données

- Deux types :
  - Valeur : une variable de type valeur stocke directement sa valeur
  - Référence : une variable de type référence stocke sur référence vers les données

# Types de données - Valeurs

- Types simples :
  - *sbyte, short, int, long* : Entiers signés
  - *byte, ushort, uint, ulong* : Entiers non signés
  - *char* : Caractères unicode
  - *float, double* : Nombres à virgule flottante (IEEE)
  - *decimal* : Nombres à virgule flottante à grande précision
  - *bool* : Valeur booléenne, soit *true* soit *false*

```
int unEntier = 1;  
double unReel = 1.0;  
decimal unDecimal = 1.0m;
```

: déclaration  
: initialisation

# Types de données - Valeurs

- Autres types :
  - Valeur *nullable* : extension des types simples en ajoutant la valeur nulle
  - Structure : regroupement de « variables »
  - Énumération : type de données créé par l'utilisateur qui énumère les valeurs possibles

# Types de données - Référence

- Classe :
  - *object* : classe de base de toutes les classes
  - *string* : classe représentant les chaînes de caractères unicode
  - *Classe utilisateur* : classe programmée par des programmeurs, **vous**
  - **Dans ce cours nous allons nous limiter à des structures implantées à partir de classes amputées de la partie orientée objet**
- Tableau

# Tests unitaires automatisés

- Tests unitaires automatisés aussi souvent appelés plus simplement tests unitaires sont :
  - Des tests qui permettent de valider un ensemble de cas de tests sur une fonction ou une méthode
- Cas de tests = description d'un cas
  - Données d'entrée
  - Données attendues
- Exemple : test d'une fonction CalculerMinimum

Données d'entrée	Données attendues
12, 3	3
-12,-4	-12
...	...

# Tests unitaires automatisés

- Est-ce utilisé en entreprise ?
  - Les plus grosses entreprises le font depuis des années
  - Les autres s'y mettent de plus en plus :
    - Agilité
    - Clean code

⇒Responsabilité du professionnel que vous voulez devenir ! \*
  - Il y a une approche de développement basée sur les tests, le TDD :
    1. Écriture d'un test qui échoue
    2. Écriture du programme minimum pour faire passer le test
    3. Réusiner le code
    4. Retour en 1
- Cas Volkswagen sur la responsabilité des développeurs :  
<https://www.nytimes.com/2017/08/25/business/volkswagen-engineer-prison-diesel-cheating.html>
- Cas Boing <https://www.washingtonpost.com/technology/2020/02/07/boeing-starliner-software-problems/>



# Tests unitaires automatisés - AAA

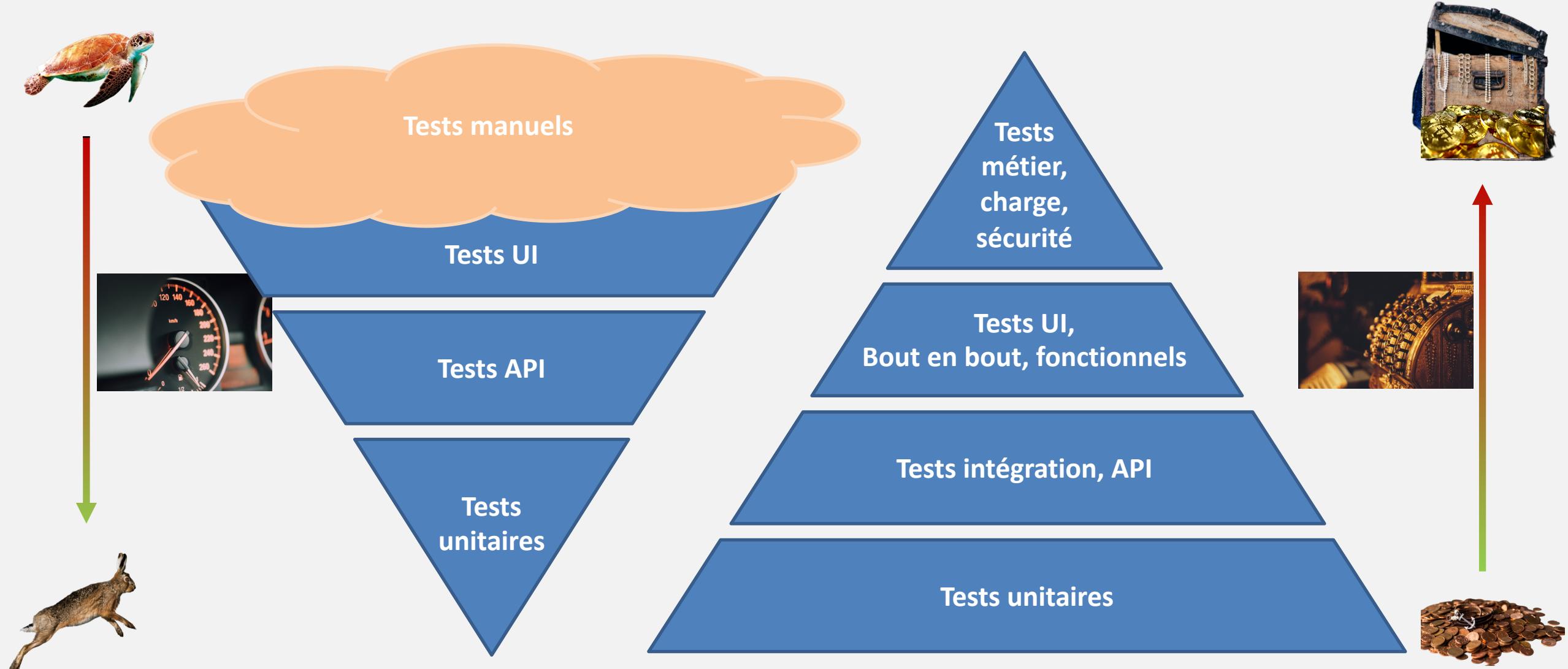
- Nous allons décrire nos tests en utilisant le découpage AAA :
  - **Arranger (Arrange)** : zone permettant de **décrire les valeurs initiales et les valeurs attendues**
  - **Agir (Act)** : zone permettant **d'effectuer l'appel à la fonction ou à la méthode à tester**
  - **Auditer (Assert)** : zone permettant de **valider que les valeurs attendues sont bien celles que nous avons obtenues** dans la zone « Agir » ; cette zone permet aussi de tester des **post-conditions** (exemple : le tableau est différent, etc.)

# Tests unitaires automatisés – Bonnes propriétés

Pour écrire des tests, Robert C. Martin indique qu'il faut respecter l'acronyme FIRST :

- **Fast (Rapide)** : Les tests doivent être rapides pour qu'ils soient lancés régulièrement
- **Independant (Indépendant)** : Les tests doivent pouvoir être lancés dans n'importe quel ordre
- **Repeatable (Reproductible)** : Les tests doivent pouvoir être reproduits dans n'importe quel environnement (votre poste de dev, la recette ou même la production si nécessaire)
- **Self-Validating (Auto validant)** : Les tests doivent avoir un résultat binaire (succès ou échec)
- **Timely (Au moment opportun)** : Les tests doivent être écrits juste avant le code de production. Si vous écrivez les tests après, vous remarquerez qu'il sera assez difficile de tester le code de production

# Tests – Manuels vs automatisation



# Tests unitaires – Création d'une fonction à tester

- Soit la fonction « *CalculerMinimum* » :

```
public static int CalculerMinimum(int p_nombre1, int p_nombre2)
{
    int minimum = p_nombre1;

    return minimum;
}
```

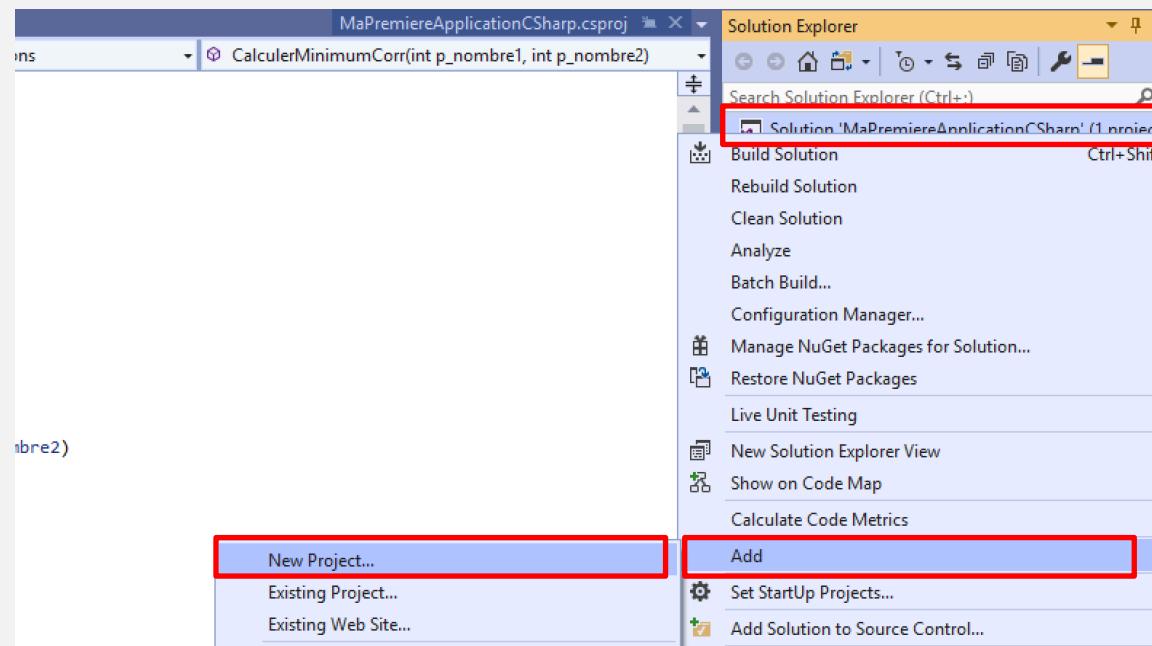
# Tests unitaires – Comment le faire dans Visual Studio ?

- Afin de tester un ensemble de fonctions qui appartiennent à un projet, nous allons devoir créer un autre projet de tests de type « **xUnit** »
- Il faut ensuite **lier ce projet au projet à tester**
- Une fois le tout effectuer, nous allons pouvoir décrire nos cas de tests en suivant le découpage **AAA**

Quelques bonnes pratiques : <https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices>

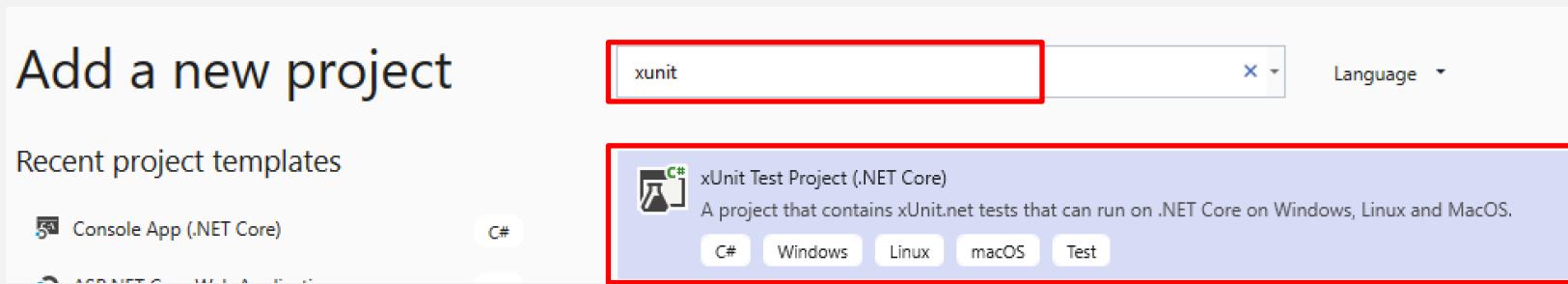
# Tests unitaires – Créer un projet de tests

- À partir de la solution, créez un projet de tests



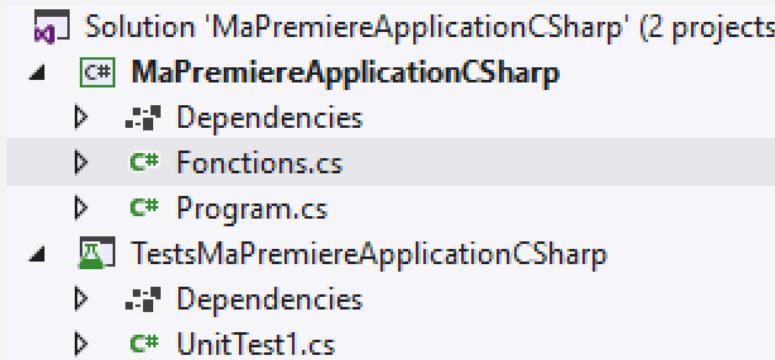
# Tests unitaires – Créer un projet de tests

- Cherchez et choisissez « xUnit »



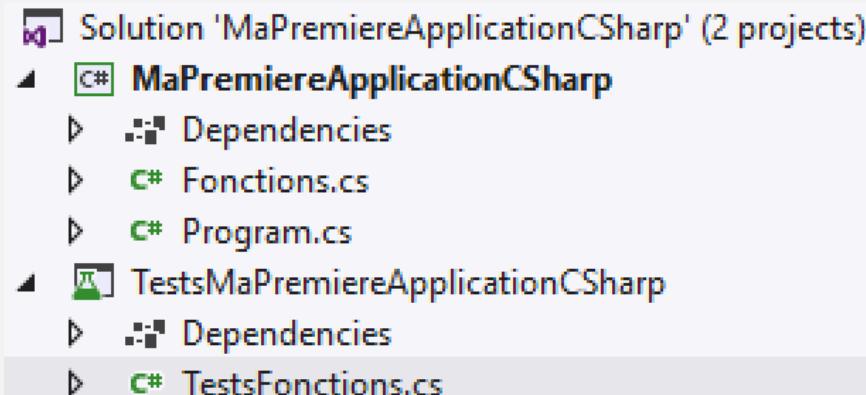
# Tests unitaires – Créer un projet de tests

- Entrez le nom « *TestsMaPremiereApplicationCSharp* »
- Vous devriez avoir maintenant un nouveau projet « *TestsMaPremiereApplicationCSharp* » dans la solution « *MaPremiereApplicationCSharp* »



# Tests unitaires – Renommer le fichier et la classe

- Renommez le fichier « UnitTest1.cs » en lui attribuant le nom « *TestsFonctions.cs* »
- Ouvre ce fichier et renommez la class « *UnitTest1* » en « *TestsFonctions* »



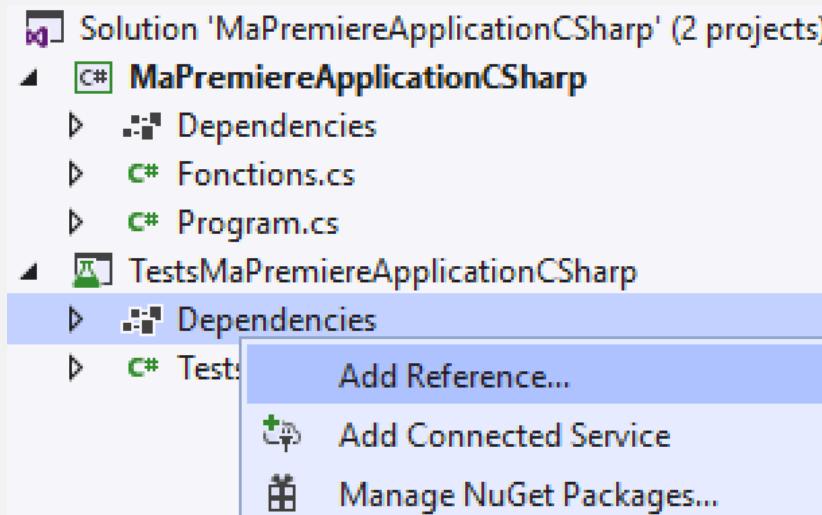
```
using System;
using Xunit;

namespace TestsMaPremiereApplicationCSharp
{
    public class TestsFonctions
    {
        [Fact]
        public void Test1()
        {
        }
    }
}
```

# Tests unitaires -

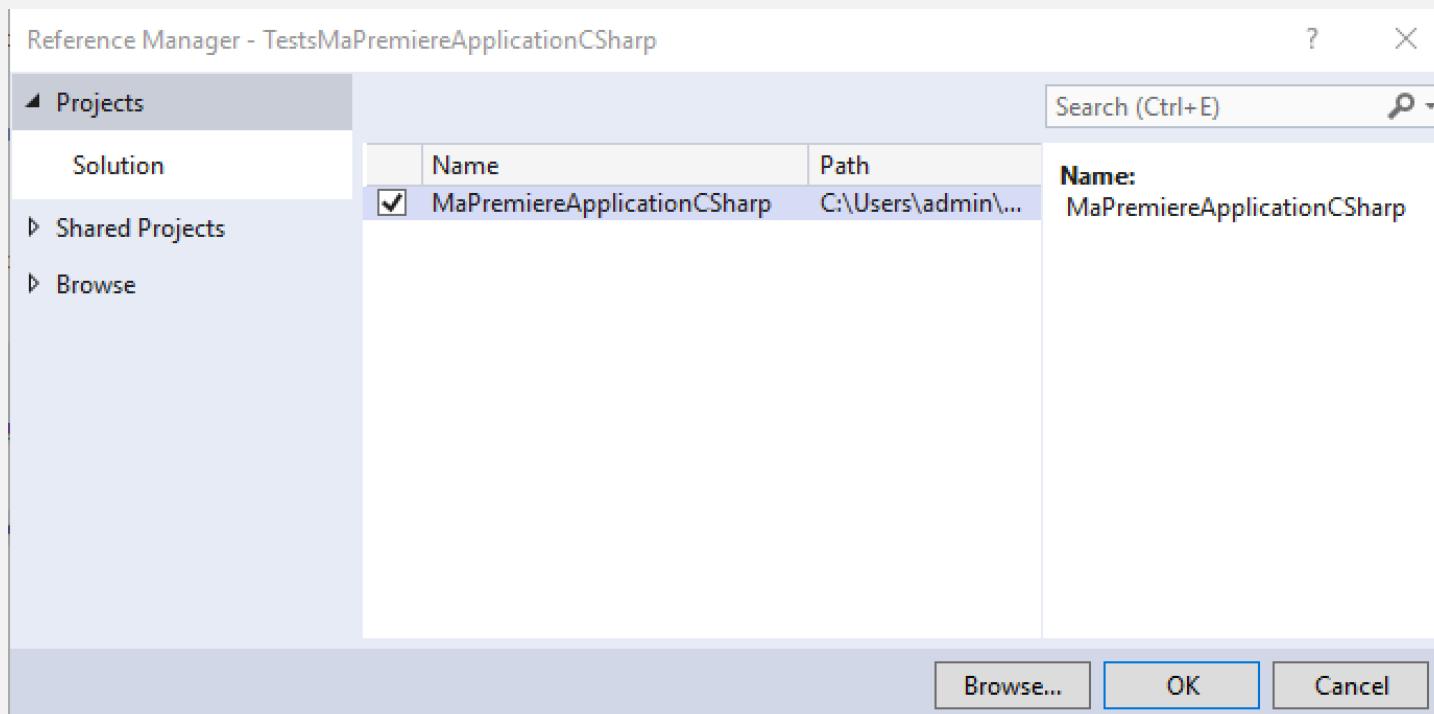
## Ajouter la liaison de projet

- Faites un clic droit sur « *Dépendances* » (*Dependencies*) et faites « *ajouter une référence* » (« *Add Reference* »)



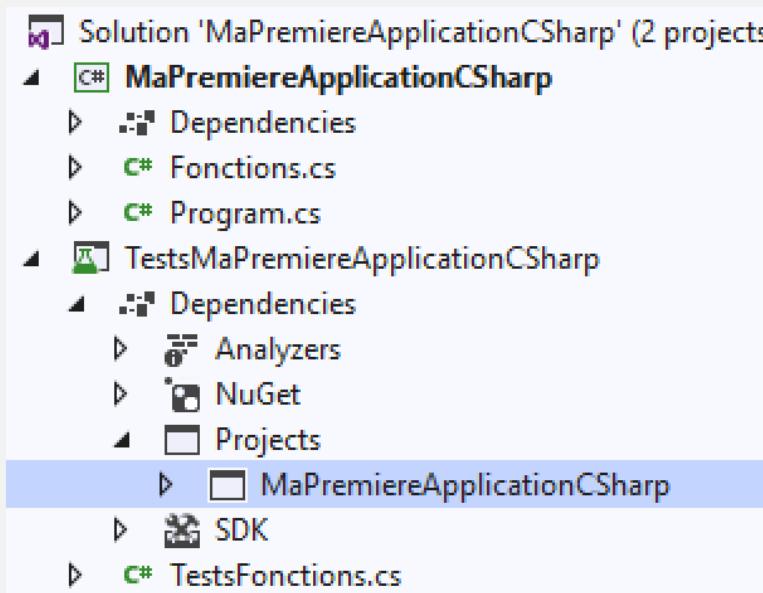
# Tests unitaires – Ajouter la liaison de projet

- Cochez le projet à tester (« *MaPremiereApplicationCSharp* ») et validez



# Tests unitaires – Ajouter la liaison de projet

- Vous pouvez valider la liaison en dépliant l'option « *Dependancies* » puis « *Projects* »



# Tests unitaires – Création d'une fonction à tester

- Nous allons ajouter de deux tests :
  - Un où le premier est le minimum
  - Un où le second est le minimum
- Pour cela, nous allons ajouter deux méthodes dans la classe « *TestsFonctions* » qui suivent la nomenclature suivante :

<Nom de la fonction / méthode à tester> <Description du cas de test> <Résultat attendu>

- Dans notre cas :
  - `CalculerMiminum_PremierElementEstMin_LePremierElement`
  - `CalculerMiminum_SecondElementEstMin_LeSecondElement`

# Tests unitaires – Créer des cas de tests

- Tapez le cas de tests suivant :

```
[Fact]
public void CalculerMiminum_PremierElementEstMin_LePremierElement()
{
    // Arranger
    int nombre1 = 23;
    int nombre2 = 42;
    int minimumAttendue = 23;

    // Agir
    int minimumCalcule = Fonctions.CalculerMinimum(nombre1, nombre2);

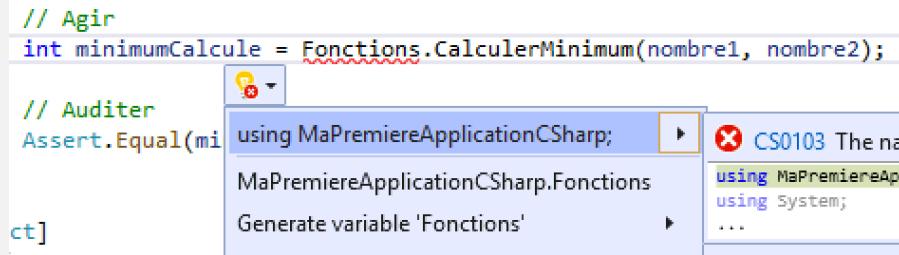
    // Auditer
    Assert.Equal(minimumAttendue, minimumCalcule);
}
```

# Tests unitaires – Créer des cas de tests

- Le compilateur va afficher une erreur et surligner « *Fonctions* »

```
// Agir
int minimumCalcule = Fonctions.CalculerMinimum(nombre1, nombre2);
```

- Affichez les suggestions du compilateur et prenez la première option



- Il suggère d'utiliser l'espace de noms « *MaPremiereApplicationCSharp* »

# Tests unitaires – Créer des cas de tests

- Tapez le cas de tests suivant :

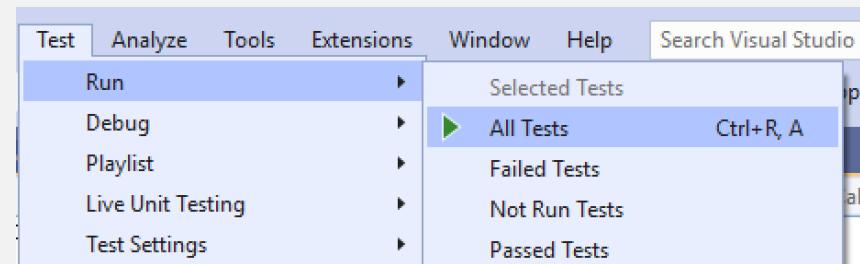
```
[Fact]
public void CalculerMiminum_SecondElementEstMin_LeSecondElement()
{
    // Arranger
    int nombre1 = 42;
    int nombre2 = 23;
    int minimumAttendue = 23;

    // Agir
    int minimumCalcule = Fonctions.CalculerMinimum(nombre1, nombre2);

    // Auditer
    Assert.Equal(minimumAttendue, minimumCalcule);
}
```

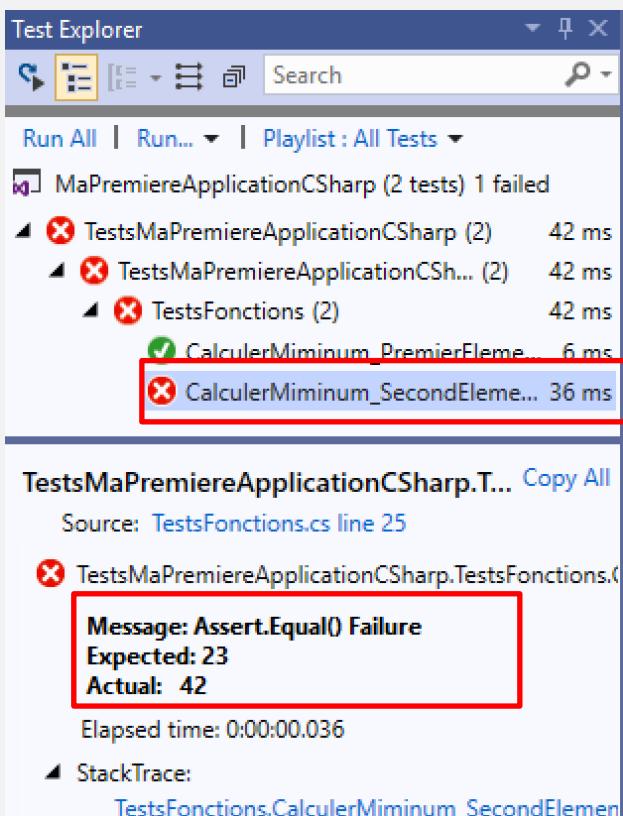
# Tests unitaires – Exécuter les cas de tests

- Exédez les tests en allant dans le menu « *Test* », « *Run* » et « *All Tests* »



# Tests unitaires – Exécuter les cas de tests

- L'explorateur de tests devrait s'afficher et vous donner le résultat des tests



Comme nous nous attendions, nous avons un échec. La valeur attendue est 23, nous avons eu 42

# Tests unitaires – Autres méthodes

Méthode	Remarque
Assert.Equal(<valeur attendue>, <valeur réel>[, <précision>]) Assert.NotEqual(<valeur attendue>, <valeur réel>[, <précision>])	égale / différent
Assert.True(<valeur>) Assert.False(<valeur>)	vraie, fausse
Assert.Null(<valeur>) Assert.NotNull(<valeur>)	nulle, non nulle
Assert.Same(<variable référence 1>, <variable référence 2>) Assert.NotSame(<variable référence 1>, <variable référence 2>)	validation d'égalité de références
Assert.Throws<Type exception>(<un appel de méthode>)	validation de la levée d'une exception
Assert.Throws<ArgumentException>(() => { Fonctions.CalculerMinimum(null); });	

# Tests unitaires – Tester vos pré-conditions

- Prenons l'exemple de la fonction « *CalculerMinimum* »

```
public static int CalculerMinimum(int[] p_tableauEntiers)
{
    if (p_tableauEntiers == null
        || p_tableauEntiers.Length == 0)
    {
        throw new ArgumentException("Le tableau ne doit pas être nul ou
vide", "p_tableauEntiers");
    }

    int minimum = p_tableauEntiers[0];
    // Code...

    return minimum;
}
```

# Tests unitaires – Tester vos pré-conditions

- Pour le tester nous allons utiliser la méthode « Throws » :

```
[Fact]
public void CalculerMinimum_TableauNull_Exception()
{
    // Arranger
    int[] tableauDEntiers = null;

    // Agir && Auditer
    Assert.Throws<ArgumentException>(
        () => { int minimumCalcule = Fonctions.CalculerMinimum(tableauDEntiers);
    });
}
```

# Tests unitaires – Comment choisir les tests

- Plus variés possibles
- Couvrir le plus de cas différents
- Couvrir les cas limites
- Pour les trouver :
  - Lire la description de la fonction
  - Identifier la plage de valeurs possibles
  - Cas normaux :
    - Avec des valeurs « au milieu » de la plage de valeurs
  - Cas limites :
    - Avec les extrêmes (minimum et maximum)
- Plus il y a de tests, plus le code sera robuste !

# Tests unitaires – À vous...

- Essayez les cas de tests précédent en suivant toutes les étapes
- Finissez le code de la code précédente
- Testez ce que vous pouvez tester