1 Tests unitaires

Les tests unitaires sont des classes déroulant une séquence d'appels de méthodes pour les composants d'une application. Les résultats retournés par ces méthodes sont comparés à leurs valeurs théoriques.

Il est nécessaire de s'assurer que tous les scénarios d'un développement sont couverts par un test. Lorsque les tests couvrent tous les scénarios d'un code, nous pouvons assurer que ce code fonctionne. De plus, cela permet par la suite de faire des opérations de maintenance sur le code tout en étant certain que ce code n'aura pas subi de régressions.

1.1 Tests unitaires manuels

Créez un projet de type bibliothèque de classe `Calculatrice` afin d'y coder la classe statique `Calcul` :
Créez une méthode Addition(Double,Double) qui retourne l'addition de 2 nombres. Compilez cette classe pour créer une dll. Cette bibliothèque de classe sera donc réutilisable via la dll.
Créez un projet *Application console (.NET Framework)* nommée `TestCalcul` qui va permettre de tester la bibliothèque de classes.
Ajouter une référence au projet `Calculatrice` (ajout référence → projet)

Vous remarquerez que l'application a été développée en respectant le principe de réutilisation puisque la bibliothèque de classes est réutilisable pour développer une calculatrice sous la forme d'un client lourd, d'une application console ou autre.

Dans la méthode `Main()`, ajouter le code suivant :

```
// Arranger
Double a = 1.0;
Double b = 2.0;
// Agir
Double resultat = Calcul.Addition(a, b);
// Auditer
if (resultat != 3.0)
    Console.WriteLine("Test Addition : échec");
else
    Console.WriteLine("Test Addition : réussi");
Console.ReadKey();
```

Un test est toujours défini dans un projet externe et est basé sur 3 parties (AAA) : *Arrange – Act – Assert* ou en français *Arranger – Agir – Auditer*.

Définir `TestCalcul` comme projet de démarrage et exécuter l'application.

En principe, le code suivant serait suffisant car on ne souhaite afficher que les messages des tests en échec, surtout quand le nombre de méthodes à tester est important.

```
// Arranger
Double a = 1.0;
Double b = 2.0;
// Agir
Double resultat = Calcul.Addition(a, b);
// Auditer
if (resultat != 3.0)
    Console.WriteLine("Test Addition : échec");
```

Pour être complet, le test doit couvrir un maximum de situations. Il faut donc tester notre code avec d'autres valeurs, et ne pas oublier les valeurs limites :
NB : Vous pouvez modifier les messages affichés pour les rendre plus explicites.

```
// Arranger
Double a = 1.0;
Double b = 2.0;
```

```
// Agir
Double resultat = Calcul.Addition(a, b);
// Auditer
if (resultat != 3.0)
    Console.WriteLine("Test Addition 1 : échec");

// Arranger
a = 0;
b = 0;
// Agir
resultat = Calcul.Addition(a, b);
// Auditer
if (resultat != 0)
    Console.WriteLine("Test Addition 2 : échec");

// Arranger
a = 1.0;
b = -2.0;
// Agir
resultat = Calcul.Addition(a, b);
// Auditer
if (resultat != -1.0)
    Console.WriteLine("Test Addition 3 : échec");

// Arranger
a = -1.0;
b = -2.0;
// Agir
resultat = Calcul.Addition(a, b);
// Auditer
if (resultat != -3.0)
    Console.WriteLine("Test Addition 4 : échec");

Console.ReadKey();
```

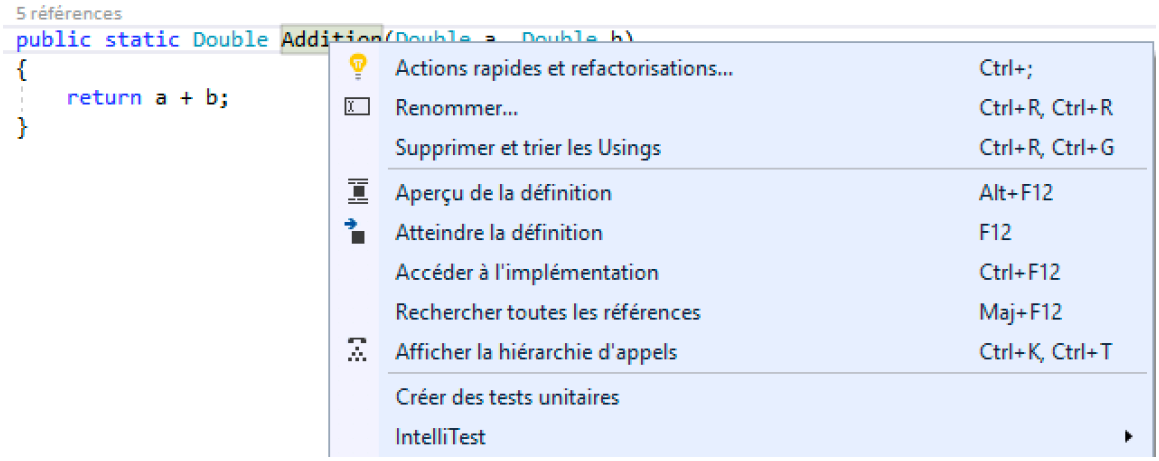
Ici, nous considérons avoir écrit suffisamment de tests pour nous assurer que cette méthode est bien fonctionnelle.

1.2 Tests unitaires automatisés

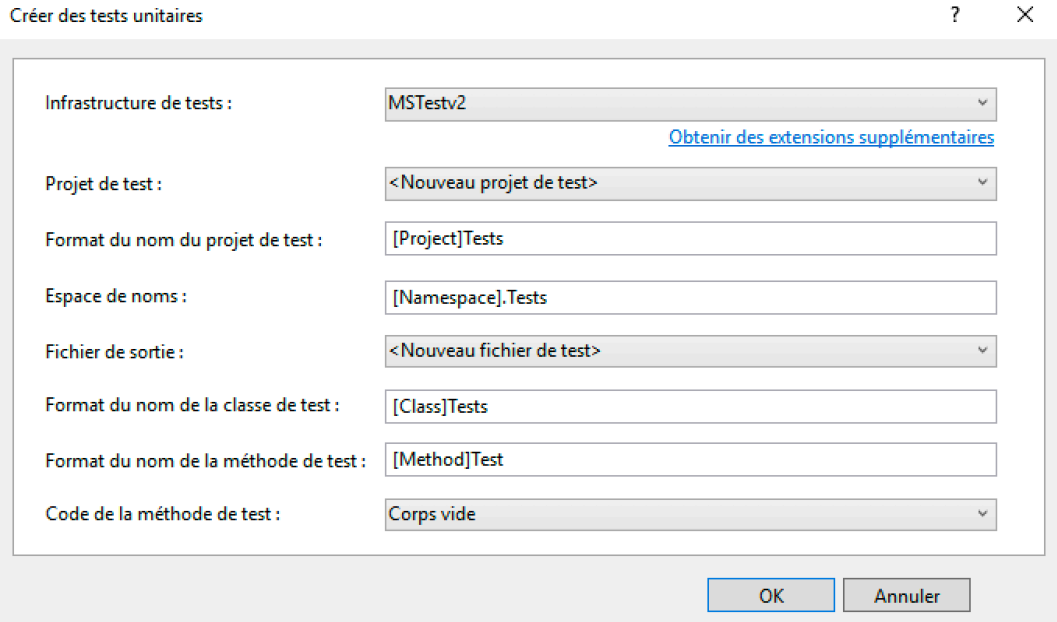
Plutôt que d'écrire ces tests à la main, Visual Studio fournit un framework de tests automatisés (depuis Visual Studio 2013, il est intégré gratuitement dans toutes les versions). Il fournit au développeur un environnement structuré permettant l'exécution de tests et des méthodes pour aider au développement de ceux-ci.

Il existe plusieurs frameworks de test. Microsoft dispose de son propre framework, *MSTest Version 2*.
Remarque : La version 2 est disponible depuis Visual Studio 2017. Il est cependant toujours possible d'utiliser la version précédente MSTest.
D'autres framework de tests existent, comme *NUnit*. NUnit est la version pour .NET du framework XUnit, qui se décline pour plusieurs environnements, avec, par exemple, PHPUnit pour le langage PHP, JUnit pour java, etc. Mais dans ce cas, il est nécessaire d'installer un package NuGet.

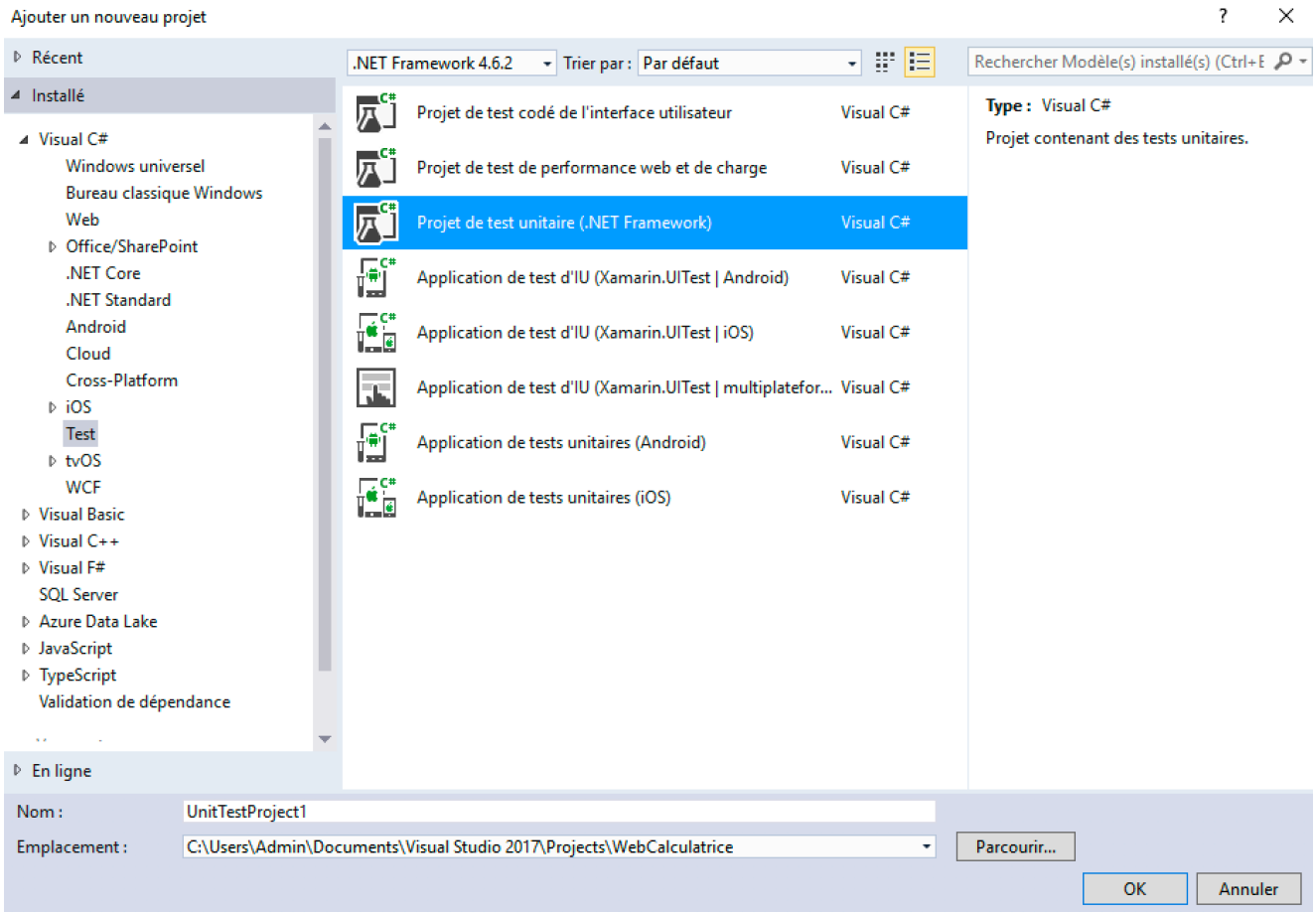
Cliquer avec le bouton droit de la souris sur la méthode `Addition` puis « Créer des tests unitaires ».



Par défaut, quand vous créez un projet de test unitaire, c'est le framework MSTest Version 2 qui est utilisé. Choisir « Corps vide » comme code de la méthode de test et valider.



Un nouveau projet `CalculatriceTests` est créé.
Remarque : On peut aussi ajouter le projet manuellement à la solution :



L'écriture de tests unitaires automatisés nécessite la création d'un second projet car les tests sont toujours déroulés dans un processus distinct de l'application.

La classe de test consiste en une série de méthodes vérifiant le fonctionnement de chaque méthode ciblée de la classe « fonctionnelle » (i.e. à tester). Le programmeur doit renseigner pour chacune d'elles les valeurs des paramètres, les valeurs attendues, le type de comparaison et éventuellement le message d'erreur.

Dans notre cas, nous allons utiliser la méthode de MSTestv2 `Assert.AreEqual()`. Elle permet de vérifier qu'une valeur est égale à une autre attendue. Elle contient en général 2 ou 3 paramètres :

- 1er paramètre : valeur attendue
- 2ème : valeur retournée
- 3ème : message d'erreur

Renommer la méthode de test ainsi et saisir le code suivant. Si cela n'est pas fait, ne pas oublier de rajouter la référence au projet à tester depuis le projet de test.

```
[TestMethod()]
public void AdditionTest_AvecValeur_1_2_Retourne3()
{
    //Arrange
    Double a = 1.0;
    Double b = 2.0;
    //Act
    Double resultat = Calcul.Addition(a, b);
    //Assert
    Assert.AreEqual(3.0, resultat, "Test non OK. La valeur doit être égale à 3");
}
```

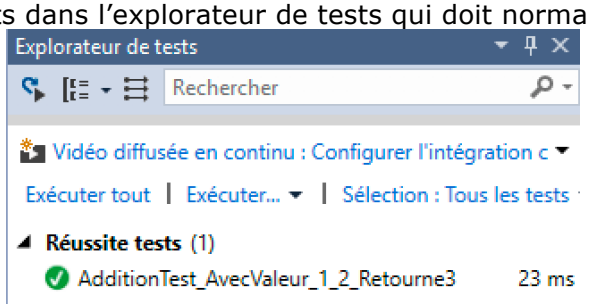
Il est préférable d'ajouter un message d'erreur en 3ème paramètre :

```
Assert.AreEqual(3.0, resultat, "Test non OK. La valeur doit être égale à 3");
```

Une fois les méthodes de test renseignées, le projet de test est démarré comme tous les autres. Il va instancier chaque composant cible et lui appliquer la séquence de tests. Un rapport est ensuite élaboré à l'attention du programmeur. Pour cela, aller dans le menu *Test > Exécuter > Tous les tests.*

Remarque : Dans le menu Test > Déboguer, on peut déboguer les tests, utiliser des points d'arrêt, etc., car il s'agit toujours de code !

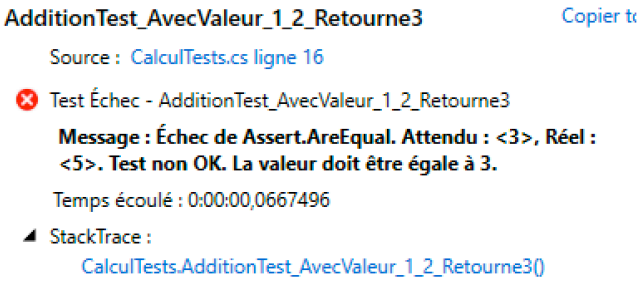
On peut voir le résultat des tests dans l'explorateur de tests qui doit normalement s'afficher :



Avec une méthode si simple, il y a peu de chance que l'on voit l'échec du test. Pour cela, modifier la méthode `Addition` (normalement, on ne retouche plus au test une fois celui-ci créé), par exemple :

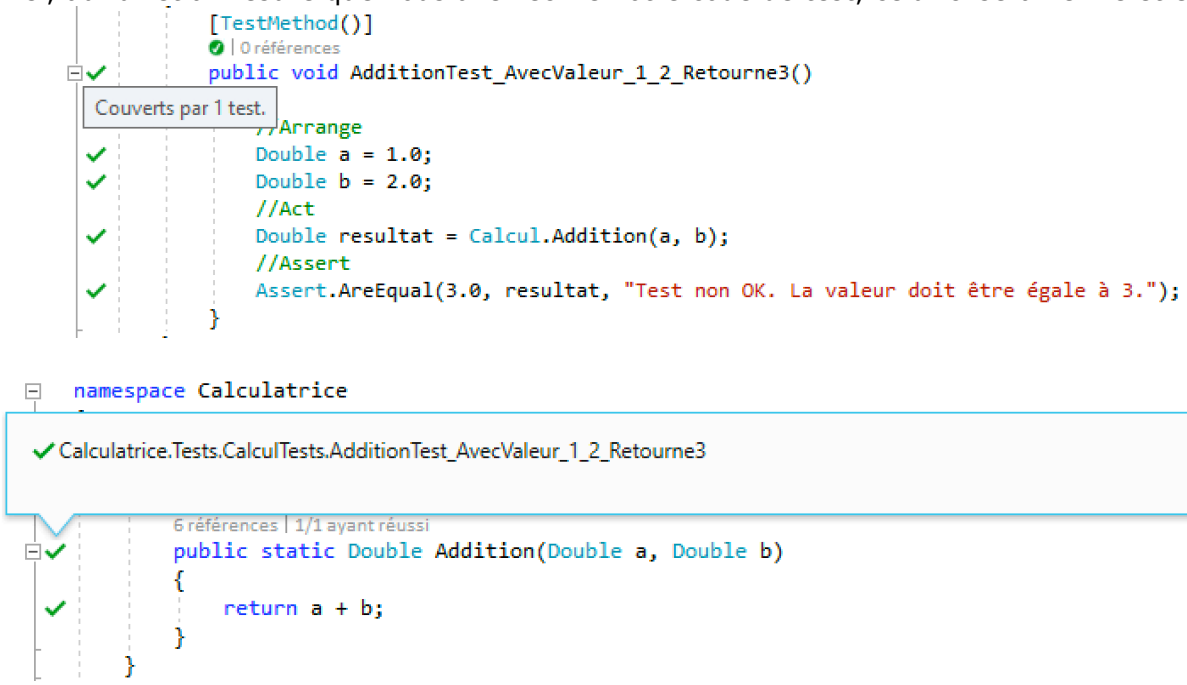
```
return a + b + 2;
```

Maintenant, vous devriez voir l'échec en réexécutant les tests. Ici, nous avons - volontairement – créer une regression du code.



Un test, même après modification de la méthode testée, doit toujours fonctionner (sauf si on l'a bien sûr mal codé !). Il est important de lancer systématiquement les tests après des modifications réalisées sur la méthode testée.

Dans le menu *Test > Live Unit Testing*, cliquer sur « Démarrer ». Le live unit testing, nouvelle fonctionnalité disponible en version 2017, permet d'avoir des indicateurs en temps réel sur la couverture de code ainsi que son statut. Ces indicateurs sont représentés par des petits sigles sur le côté gauche du code.Ainsi, au fûr et à mesure que vous allez écrire votre code de test, celui-ci sera vérifié et exécuté.

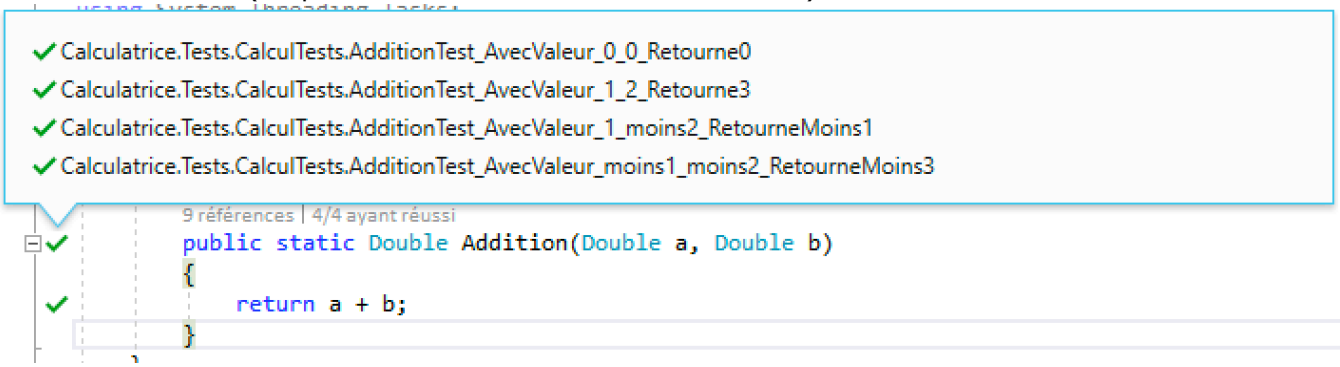


Si vous modifiez votre méthode de test, on peut voir en temps réel si elle réussit ou est en échec.

```
[TestMethod()]
0 références
public void AdditionTest_AvecValeur_1_2_Retourne3()
{
    //Arrange
    Double a = 1.0;
    Double b = 2.0;
    //Act
    Double resultat = Calcul.Addition(a, b);
    //Assert
    Assert.AreEqual(3.0, resultat, "Test non OK. La valeur doit être égale à 3.");
}
```

```
6 références | 0/1 ayant réussi
public static Double Addition(Double a, Double b)
{
    return a + b;
}
```

Il est important qu'une méthode de test ne s'occupe de tester qu'un seul cas d'une unique fonctionnalité (méthode), comme nous venons de le faire. La première méthode teste la méthode `Addition` pour le cas où les opérandes sont 1 et 2. Rajouter les 3 autres méthodes nécessaires au test de la méthode `Addition` (ne pas oublier l'attribut `[TestMethod()]`).



Remarque : on peut suspendre, arrêter ou redémarrer le live testing à tout moment.

On peut également calculer la couverture du code (menu *Test* > *Analyser la couverture du code* > *Tous les tests*).

Résultats de la couverture du code

Admin_DESKTOP-P14QAEA 2017-09-18 20_5

Hiérarchie	Non couverts (blocs)	Non couverts (% blocs)	Couverts (blocs)	Couverts (% blocs)
Admin_DESKTOP-P14QAEA 2017-0...	0	0,00 %	14	100,00 %
calculatrice.dll	0	0,00 %	2	100,00 %
calculatricetests.dll	0	0,00 %	12	100,00 %

Il faut bien sûr essayer de tendre vers 100% de couverture (ce n'est pas toujours très simple !) pour les classes fonctionnelles (`calculatrice.dll` ici).

Bilan : La méthode `Addition()` est par définition fonctionnelle, mais il est important de prendre le réflexe de tester des fonctionnalités qui sont déterminantes pour l'application. Il faut écrire au moins 1 test qui réussit et un test par exception (si une ou plusieurs exceptions sont gérées dans le code). Mais comme dans le cas de l'addition, on souhaite également additionner des nombres nuls, positifs et/ou négatifs, nous avons créé plusieurs tests qui "réussissent".

2 Test Driven Development (TDD)

Lors d'une approche TDD, le but est de pouvoir faire un développement à partir des cas de tests préalablement établis par la personne qui exprime le besoin ou suivant les spécifications fonctionnelles.

En général, les étapes suivantes sont respectées :

1. Ecriture des scénarios de tests (à partir du CdC ou des spec. fonctionnelles)

2. Codage des tests. On l'a vu, on a besoin de faire une référence dans le projet de test au namespace contenant la classe à tester et d'appeler des méthodes non encore codées. Cela impose au préalable de créer les méthodes de la classe à tester et de les déclarer non encore implémentées (`throw new NotImplementedException()`) .
3. Exécution des tests (ou Live Unit Testing) -> *Aucun ne doit passer*
4. Codage des méthodes (du projet à tester)
5. Exécution des tests (ou Live Unit Testting) -> *Tous doivent passer*
6. Et ensuite, pour chaque itération de modification ou refactoring du code des méthodes, une réexécution des tests s'impose.

Cette technique est notamment très adaptée voire recommandée pour la réalisation des tests notamment unitaires.

2.1 Exemple : division

2.1.1 Jeux de test

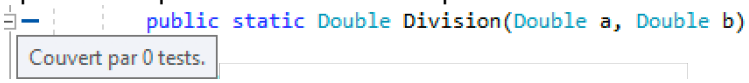
Paramètre 1	Paramètre 2	Résultat
1	2	0.5
1	-2	-0.5
0	1	0
1	0	Exception : `DivideByZeroException("Erreur division par zéro")`

Ici, la couverture des tests est (largement) suffisante. **En principe, il vaut mieux en faire un peu plus que pas assez…**

2.1.2 Codage des tests

1. Créer la méthode `Division` et la déclarer non implémentée.

En live unit testing, on peut voir que la méthode n'est pas encore testée :



2. Coder les tests de la méthode `Division` (les 4 cas possibles du tableau ci-dessus).

Deux possibilités pour coder le test d'une exception :

- 1ère possibilité :

```
// Arrange
…
Exception expectedException = null;

// Act
try
{
        …
}
catch (Exception ex)
{
        expectedException = ex;
}

// Assert
Assert.IsNotNull(expectedException, "Test non OK. Erreur exception.");
```

`Assert.IsNotNull()` vérifie si l'objet en paramètre est bien instancié et donc, ici, si une instance de l'exception a bien été créée. On peut ajouter en 2nd paramètre le message, comme pour les tests précédents.
Si vous exécutez les tests, seul ce test devrait réussir car la classe `NotImplementedException` hérite de la classe `Exception`.
Il est cependant préférable d'être précis et de gérer une exception spécifique `DivideByZeroException` plutôt qu'`Exception`. Modifier ainsi le code (aux 2 endroits !).

- 2ème possibilité (préférable car plus concis) :

```
[TestMethod()]
[ExpectedException(typeof(MonException))]
public void MaMethodeTest_AvecValeur_X_Y_RetourneMonException()
{
    // Arrange
    …
    // Act levant l'exception attendue de type MonException
    …
    // PAS d'Assert
}
```

Remarque : cette 2ème possibilité ne fonctionne qu'avec MSTest (versions 1 & 2). La première fonctionne dans tous les frameworks de tests.

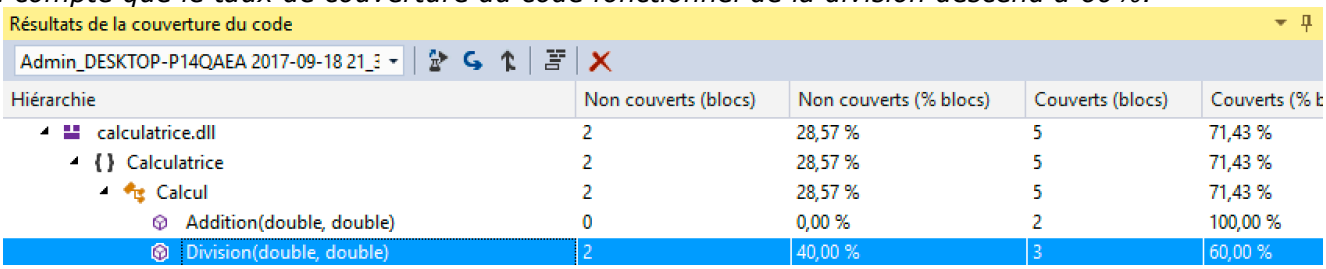
Vous pourrez coder ces 2 possibilités pour tester la division par zéro.

Exécuter les tests. Normalement, si l'exception gérée est bien `DivideByZeroException`, les tests devraient échouer. **Mais c'est l'objectif recherché !**

2.1.3 Codage de la division

Tout est dans le titre…
Quand on divise un nombre par 0, .Net renvoie l'infini (+*Infini*). Vous lèverez donc l'exception `DivideByZeroException("Erreur division par zéro")` en cas de tentative de division par zéro.

Calculer la couverture du code.
Remarque : si l'on met les 2 méthodes de test de l'exception de division par zéro en commentaire, on se rend compte que le taux de couverture du code fonctionnel de la division descend à 60%.



2.2 Exercice : factorielle

Procéder de même que pour la division **en respectant une démarche TDD**. Vous coderez d'abord la factorielle sans utilisation de récursivité (boucle `for` par exemple) et exécuterez les tests. Ce code n'étant pas optimal, le remplacer par une fonction récursive. Ré-exécutez les tests après cette modification majeure afin de s'assurer qu'il n'y a pas de régression dans le code.
Une factorielle prend un entier en paramètre.