Tests Unitaires

Lors de la création d'un programme, le principal souci du développeur est de savoir si son code est fonctionnel. Il cherche à tester son application en imaginant tout ce que ses futurs utilisateurs pourraient effectuer (clic en tout genre, saisie de caractères exotiques, etc.).

Chaque développeur effectue d'instinct une série de tests de son application. La plupart du temps, il utilise la manière la plus naturelle, à savoir lancer son application et faire comme s'il était un utilisateur. Il corrige les bugs trouvés et s'amuse à tester à nouveau, jusqu'au moment où il pense que tout est fonctionnel.

Cette procédure présente l'avantage qu'elle est directement accessible et rend rapidement l'état de l'application. Par contre, elle ne garantit pas l'efficacité des résultats (ai-je déjà testé ce cas ?) et elle n'est pas réutilisable en cas de changement de code (tous les tests faits manuellement devront être refaits).

Est-ce possible d'automatiser les tests à effectuer et surtout de les réutiliser en cas de changement ou maintenance du code ? La réponse est OUI ;-)

Définition

Un test est le fait de créer un bout de code extérieur à l'application, qui permet de tester un bout de code de l'application.

Exemple : J'ai une méthode de mon application qui me permet d'additionner deux nombres :

```
public class OperationMath
{
    /// <summary>
    /// Main
    /// </summary>
    static void Main(string[] args)
    {

    }

    /// <summary>
    /// Sum two numbers
    /// </summary>
    /// <param name="a">First number</param>
    /// <param name="b">Second number</param>
    public static int Sum(int a, int b)
    {
        return a + b;
    }
}
```

Je désire additionner les chiffres 4 et 6, normalement le résultat doit me donner 10. Comment puis-je être sûr que le résultat soit correct ? Je peux écrire un bout de code qui permet de le vérifier pour moi.

```
public class OperationMath
{
    /// <summary>
    /// Main
    /// </summary>
    static void Main(string[] args)
    {
        int numberOne = 4;
        int numberTwo = 6;
        int result = 0;

        result = Sum(numberOne, numberTwo);

        if (result == 10)
        {
            "Le résultat est correct";
        }
    }

    /// <summary>
    /// Sum two numbers
    /// </summary>
    /// <param name="a">First number</param>
    /// <param name="b">Second number</param>
    public static int Sum(int a, int b)
    {
        return a + b;
    }
}
```

C'est le principe des tests. Maintenant, nous allons utiliser ce qui se nomme des Frameworks de test. Les langages de programmation ont chacun le(s) leur(s). Pour C#[1], il est possible d'utiliser directement celui fournit dans Visual Studio (VS) Community, il se nomme MSTest.

[1] Il est aussi possible d'utiliser d'autres Framework comme NUnit qui s'intègre dans Visual Studio.

Principe des tests

La mise en place des tests s'effectue selon trois principes :

- Arrange : définir ce que nous avons besoin, par exemple les variables, leurs valeurs …
- Act : exécuter l'action, par exemple, appeler la méthode à tester
- Assert : vérifier que le résultat obtenu soit conforme à nos attentes.

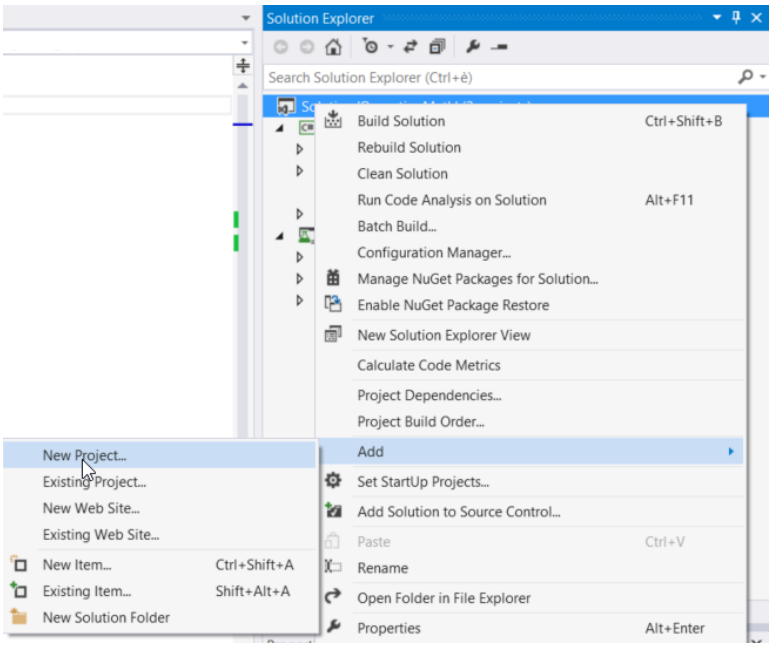
Et il est possible de faire autant de tests d'une fonctionnalité que nécessaire, le but étant de choisir ce qui est déterminant et ne pas faire des choses inutiles.

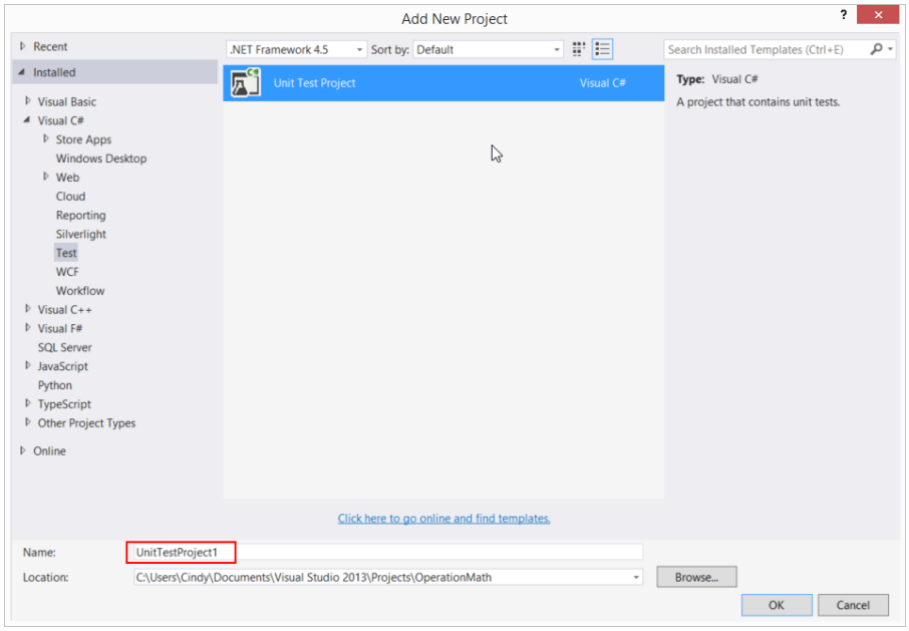
Mise en place avec Visual Studio

Reprenons l'exemple sur la méthode pour additionner deux nombres. Nous avons un projet qui se compose de notre classe pour faire des opérations de math :

```
public class OperationMath
{

    /// <summary>
    /// Main
    /// </summary>
    static void Main(string[] args)
    {

    }

    /// <summary>
    /// Sum two numbers
    /// </summary>
    /// <param name="a">First number</param>
    /// <param name="b">Second number</param>
    public static int Sum(int a, int b)
    {
        return a + b;
    }
}
```

Maintenant, nous ajoutons un nouveau projet dans la même solution qui nous permettra d'effectuer des tests.





Une bonne pratique est de nommer le nom du projet de test par le nom du projet suivi de test. Ici, comme notre projet est OperationMath, le projet contenant les tests est OperationMath.Tests.

De base, le fichier ".cs" contenant la classe de test de s'appelle "UnitTest1.cs", renommons-le "OperationMathTests.cs". Dans un projet, beaucoup de fichiers de tests peuvent être créés, il est nécessaire de les appeler correctement dès le départ. Une bonne pratique est de le nommer du même nom que le fichier a testé et le terminant par "Tests".

En ouvrant le fichier "OperationMathTests.cs", Visual Studio a utilisé une structure qui permet d'effectuer des tests. En effet, il y a des attributs qui décorent les éléments. Ces derniers sont importants pour le fonctionnement du test.

```
[TestClass]
public class OperationMathTests
{
    [TestMethod]
    public void TestMethod1()
    {
    }
}
```

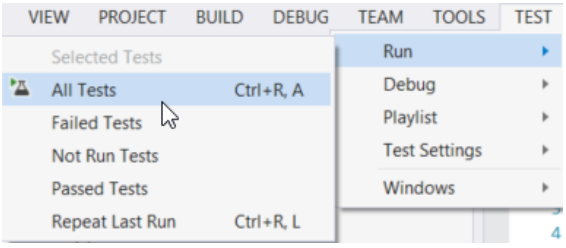
Il est possible d'ajouter autant de méthodes de tests que nécessaire à l'intérieur de cette classe. Une bonne pratique pour la nomenclature des méthodes est de la préfixer par le nom de la méthode à tester, suivi des paramètres d'entrées et du résultat attendu. De cette manière, il est plus agréable d'analyser le résultat des tests.

```
[TestClass]
public class OperationMathTests
{
    [TestMethod]
    public void Sum_WithValue4_And_6_Result_10()
    {
        // Arrange
        int numberOne = 4;
        int numberTwo = 6;
        int result = 0;

        // Act
        result = OperationMath.Sum(numberOne, numberTwo);

        // Assert
        Assert.AreEqual(10, result , "Le resultat doit etre de 10");
    }
}
```

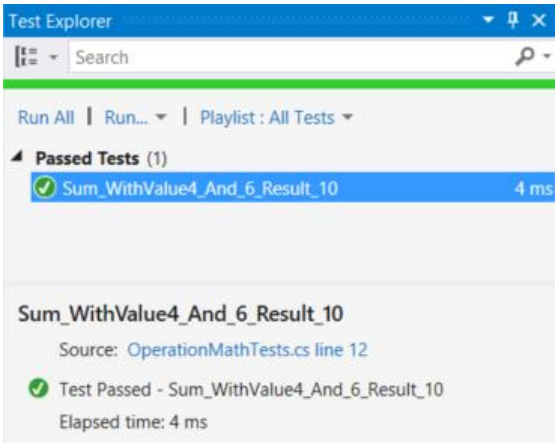
Il est important de ne pas oublier d'ajouter la référence à la classe OperationMath dans le projet de Test. Dans le code précédent, vous pouvez remarquer les commentaires reprenant les 3 principes des tests, à savoir Arrange, Act et Assert.

Il existe plusieurs moyens de comparaison en utilisant Assert. N'hésitez pas à le tester par vous-même en utilisant l'auto-complétion proposée par VS.

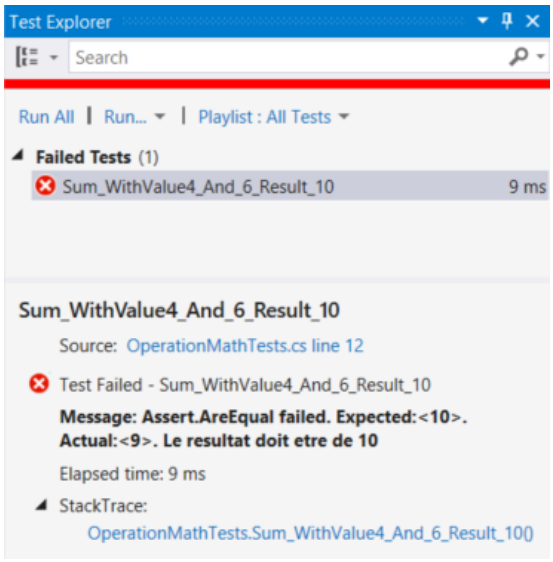
Pour effectuer les tests, il suffit de le demander à Visual Studio.



Si le test est concluant, son résultat sera vert.



Si une faute s'est glissée dans notre test, par exemple, les valeurs ne donnent pas le bon résultat, le test est un échec. Il est possible de visualiser un message indiquant ce qui n'a pas fonctionné dans le test. Par exemple, ici, un 10 était attendu, et le résultat a été 9.



L'exemple ci-dessus utilise la vérification d'égalité de deux nombres. On vérifie que le nombre retourné par la méthode est égal à une valeur à laquelle on s'attend. Il existe toutefois des situations où la condition à vérifier n'est pas l'égalité de deux nombres mais par exemple : le fait que la méthode retourne une valeur null, que la valeur retournée est plus grande ou plus petite qu'une autre valeur numérique, que la méthode lance une exception, …

La classe Assert met plusieurs autres méthodes à disposition pour ces cas-là. Voir la documentation Microsoft

Test Driven Development (TDD)

Le Test Driven Development est une pratique utilisée lors de la conception d'un programme. Le principe est de commencer par coder les méthodes de tests (donc penser à tous les cas possibles) et par la suite implémenter les classes et méthodes nécessaires à la réalisation du programme.