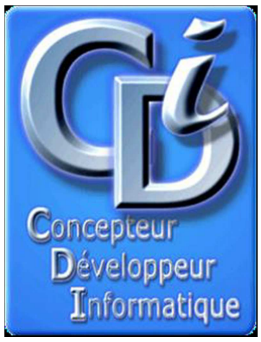
Les tests unitaires
avec NUnit et C#



Les tests unitaires

Une des grandes préoccupations des créateurs de logiciels est d'être certains que leur application informatique fonctionne et surtout qu'elle fonctionne dans toutes les situations possibles.

Qu'est-ce qu'un test unitaire et pourquoi en faire ?

Un test constitue une façon de vérifier qu'un système informatique fonctionne.
Tester son application c'est bien. Il faut absolument le faire. C'est en général une pratique plutôt laissée de côté, car rébarbative. Il y a plusieurs façons de faire des tests.
Celle qui semble la plus naturelle est celle qui se fait manuellement. On lance son application, on clique partout, on regarde si elle fonctionne.
Il existe aussi une pratique automatisée visant à s'assurer que des bouts de code fonctionnent comme il faut et 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 notre code fonctionne. De plus, cela permet de faire des opérations de maintenance sur le code tout en étant certain qu'il n'aura pas subi de régressions. De la même façon, les tests sont un filet de sécurité lorsqu'on souhaite refactoriser son code (retravailler son code sans ajouter de fonctionnalité ni corriger des bogues mais pour améliorer sa lisibilité pour la maintenance) ou l'optimiser.
Cela permet dans certains cas d'avoir un guide pendant le développement, notamment lorsqu'on pratique le TDD. Le Test Driven Development (TDD), en français Développement piloté par les tests, est une méthode de développement de logiciel qui préconise d'écrire les tests unitaires avant d'écrire le code source d'un logiciel.

Un test est donc un bout de code qui permet de tester un autre code.

En général, un test se décompose en trois parties, suivant le schéma « AAA », qui correspond aux mots anglais Arrange, Act, Assert, que l'on peut traduire en français par : Arranger, Agir, Auditer.

– Arranger : il s'agit dans un premier temps de définir les objets, les variables nécessaires au bon fonctionnement de son test (initialiser les variables, initialiser les objets à passer en paramètres de la méthode à tester, etc.).
– Agir : ensuite, il s'agit d'exécuter l'action que l'on souhaite tester (en général, exécuter la méthode que l'on veut tester, etc.).
– Auditer : enfin, il faut vérifier que le résultat obtenu est conforme à nos attentes.

Par définition (Cf. Wikipédia) un test unitaire est un procédé permettant de s'assurer du fonctionnement correct d'une partie déterminée d'un logiciel ou d'une portion d'un programme (appelée « unité » ou « module »).
Un test est dit **unitaire** s'il ne fait pas appel à d'autres ressources que la classe testé. Un test unitaire n'utilise donc pas de base de données, de socket, etc… à l'inverse d'un test d'intégration.
Un test d'**intégration** est donc par extension un test qui peut utiliser des ressources externes : bases de données par exemple.

Notre premier test

Imaginons que nous voulions tester une méthode toute simple qui fait l'addition entre deux nombres : par exemple la méthode suivante :

```
private static int Addition(int a, int b)
{
    return a + b;
}
```

Faire un test consiste à écrire des bouts de code permettant de s'assurer que le code fonctionne. Cela peut-être par exemple :

```
static void Main(string[] args)
{
    // arranger
    int a = 1;
    int b = 2;
    // agir
    int resultat = Addition (a, b);
    // auditer
    if ( resultat != 3)
        Console . WriteLine ("Le test a raté");
}
```

Ici, le test passe bien. 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. Dans notre cas, cela pourrait être avec a et b égaux à 0, a = -5 et b = 5.

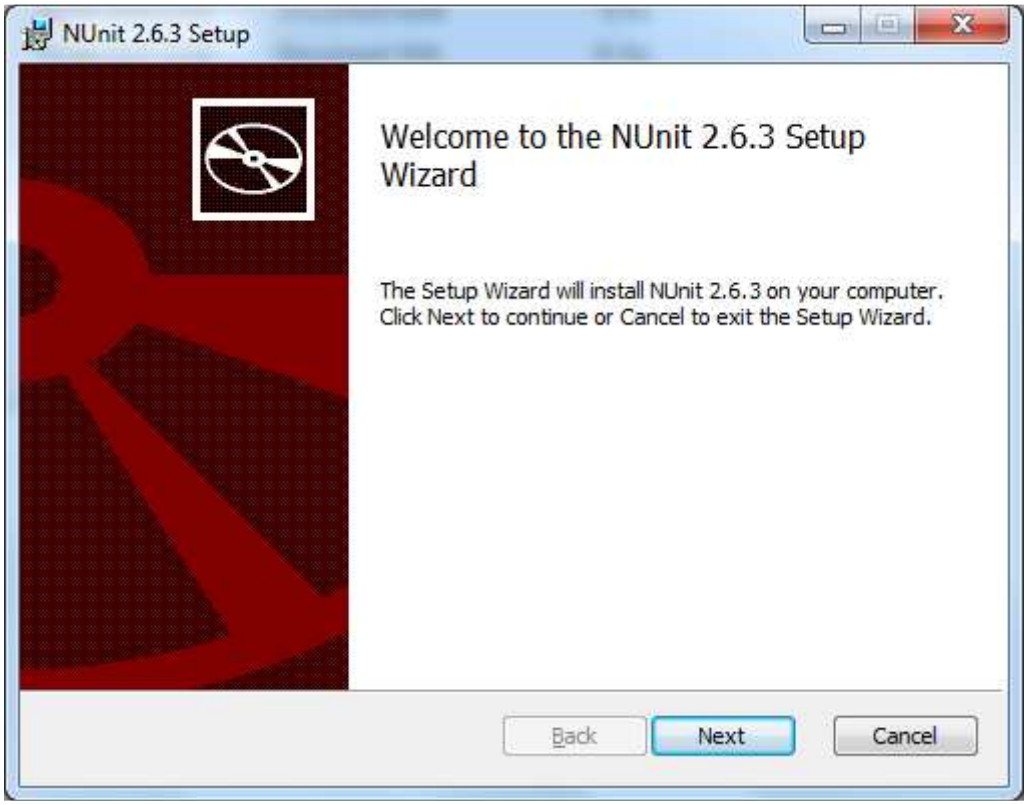
Utiliser une application console pour faire ses tests, ce n'est pas très pratique; Nous avons besoin d'outils !

Le Framework de test

Un Framework de test est aux tests ce qu'un IDE est au développement. Il fournit un environnement structuré permettant l'exécution de tests, ainsi que des méthodes pour aider au développement de ceux-ci. Il existe plusieurs Frameworks de test. Microsoft dispose de son Framework, mstest, qui est disponible dans les versions payantes de Visual Studio. Son intérêt est qu'il est fortement intégré à l'IDE. Son défaut est qu'il ne fonctionne pas avec les versions gratuites de l'environnement de développement.

Par contre, il existe d'autres Frameworks de test, gratuits, comme NUnit.
NUnit est la version .NET du Framework XUnit, qui se décline pour plusieurs environnements, avec par exemple PHPUnit pour le langage PHP, JUnit, pour java, etc.

Première chose à faire : télécharger et installer NUnit. Dans la version 2012, pas besoin de l'installer, il est inclus dans Visual Studio. Il suffira de créer un projet de type test.



Cliquez sur Next puis, après avoir accepté la licence, vous pouvez choisir l'installation classique (Typical).

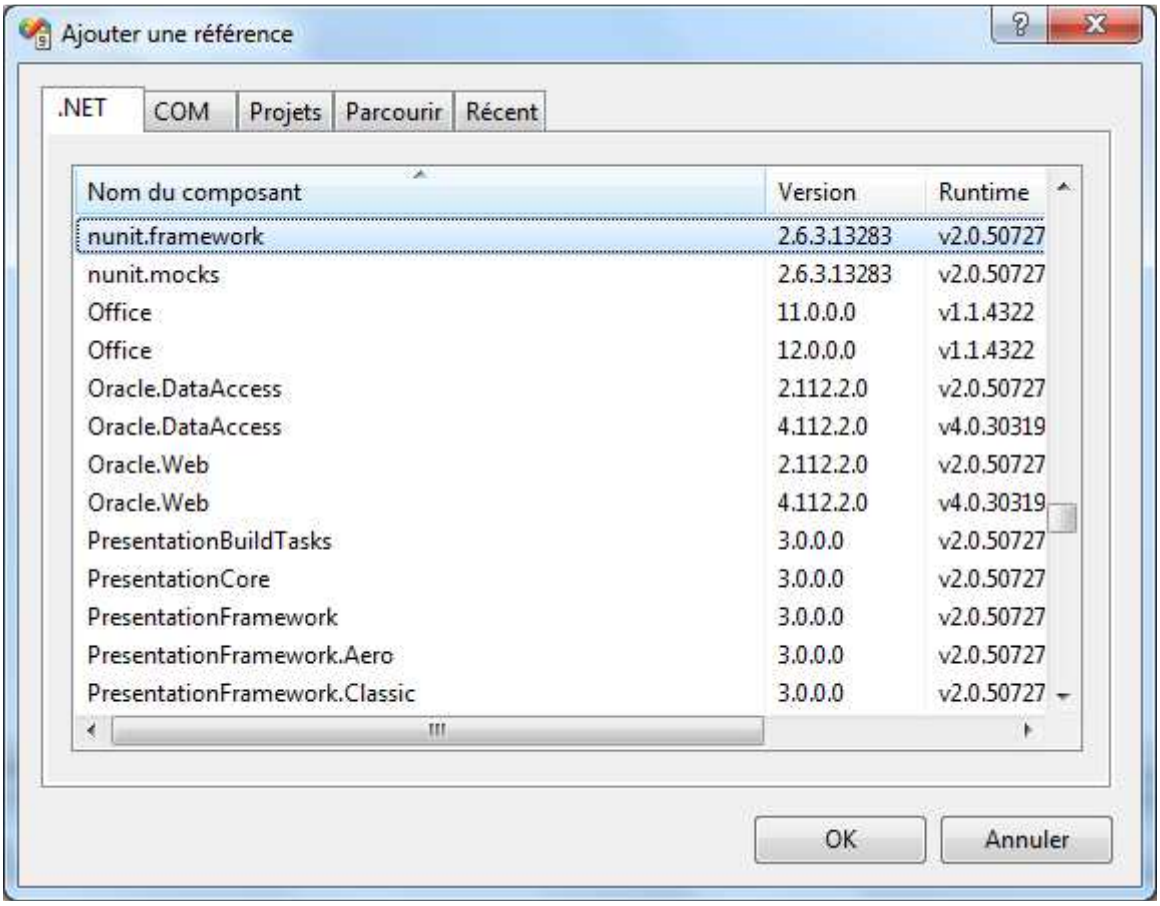
Une fois le Framework de test installé, nous pouvons créer un nouveau projet, dans Visual Studio, qui contiendra une fonctionnalité à tester. Je l'appelle MaBibliothequeATester. En général, nous allons surtout tester des assemblys (unité de déploiement indivisible) avec NUnit. Je crée donc un projet de type bibliothèque de classes. Ce projet ne sera pas exécutable, car il ne s'agit pas d'une application console. À l'intérieur, je vais pouvoir créer une classe utilitaire, disons Math, qui contiendra notre méthode de calcul de factorielle :

```
public static class Math
{
    public static int Factorielle (int a)
    {
        if (a <= 1)
            return 1;
        return a * Factorielle (a - 1);
    }
```

Ensuite, ajoutons un nouveau projet de type bibliothèque de classes où nous allons mettre nos tests unitaires, appelons-le MathTests.Unit. Ce n'est pas une norme absolue, mais je vous conseille de suffixer vos projets de test avec .Unit, afin de les identifier facilement.

Les tests doivent se mettre dans une classe spéciale. Par convention, une classe de test correspond généralement aux tests de notre code.
Ici aussi, pas de règle de nommage obligatoire, mais il est intéressant d'avoir une norme pour s'y retrouver facilement. Je vous propose de nommer les classes de tests en commençant par le nom de la classe que l'on doit tester, suivie du mot Tests. Ce qui donne : MathTests. Pour être reconnue par le framework de test, la classe doit respecter un certain nombre de contraintes. Elle doit dans un premier temps être décorée de l'attribut [TestFixture] (stéréotype). Il s'agit d'un attribut qui permet à NUnit de reconnaître les classes qui contiennent des tests.

Cet attribut étant dans une assembly de NUnit, vous devez rajouter une référence à l'assembly nunit.framework.



Vous devez ensuite inclure l'espace de noms adéquat :

```
using NUnit;
```

Nous allons pouvoir créer des méthodes à l'intérieur de cette classe. De la même façon, une méthode pourra être reconnue comme une méthode de test si elle est décorée de l'attribut [Test]. Ici aussi, il est intéressant de suivre une règle de nommage afin de pouvoir identifier rapidement l'intention de la méthode de test. Je vous propose le nommage suivant :

MethodeTestee_EtatInitial_EtatAttendu().

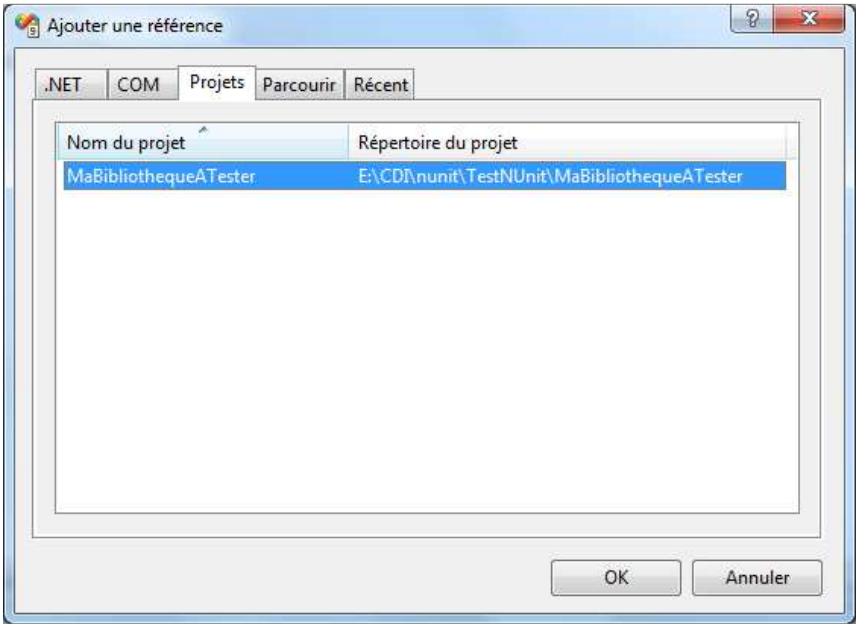
Par exemple, une méthode de test permettant de tester la factorielle pourrait s'appeler :

```
using NUnit;

namespace MathsTests.Unit
{
    [TestFixture]
    class MathTest
    {
        [Test]
        public void Factorielle_ent3_result6
        {
        }
    }
}
```

Dans un test, nous cherchons à vérifier un comportement. On utilise donc des assertions, ces assertions définissent le comportement attendu. En français, une assertion est un énoncé considéré comme vrai. Pour un test unitaire, il s'agit d'une expression qui doit être vrai pour que le test réussisse. NUnit utilise une classe statique *Assert* pour les assertions.
Pour l'écriture du test et la vérification du résultat, on utilise des méthodes de NUnit qui nous permettent de vérifier par exemple qu'une valeur est égale à une autre attendue. Cela se fait grâce à la méthode Assert.AreEqual() :

```
using NUnit.Framework;

namespace MathTests.Unit
{
    [TestFixture]
    class MathTest
    {
        [Test]
        public void Factorielle_ent3_result6()
        {
            int valeur = 3;
            int resultat = MaBibliothequeATester.Math.Factorielle(valeur);
            Assert.AreEqual(6, resultat);
        }
    }
}
```

Pour accéder à la méthode Math du projet MaBibliothequeATester, il faut rajouter la référence du projet.



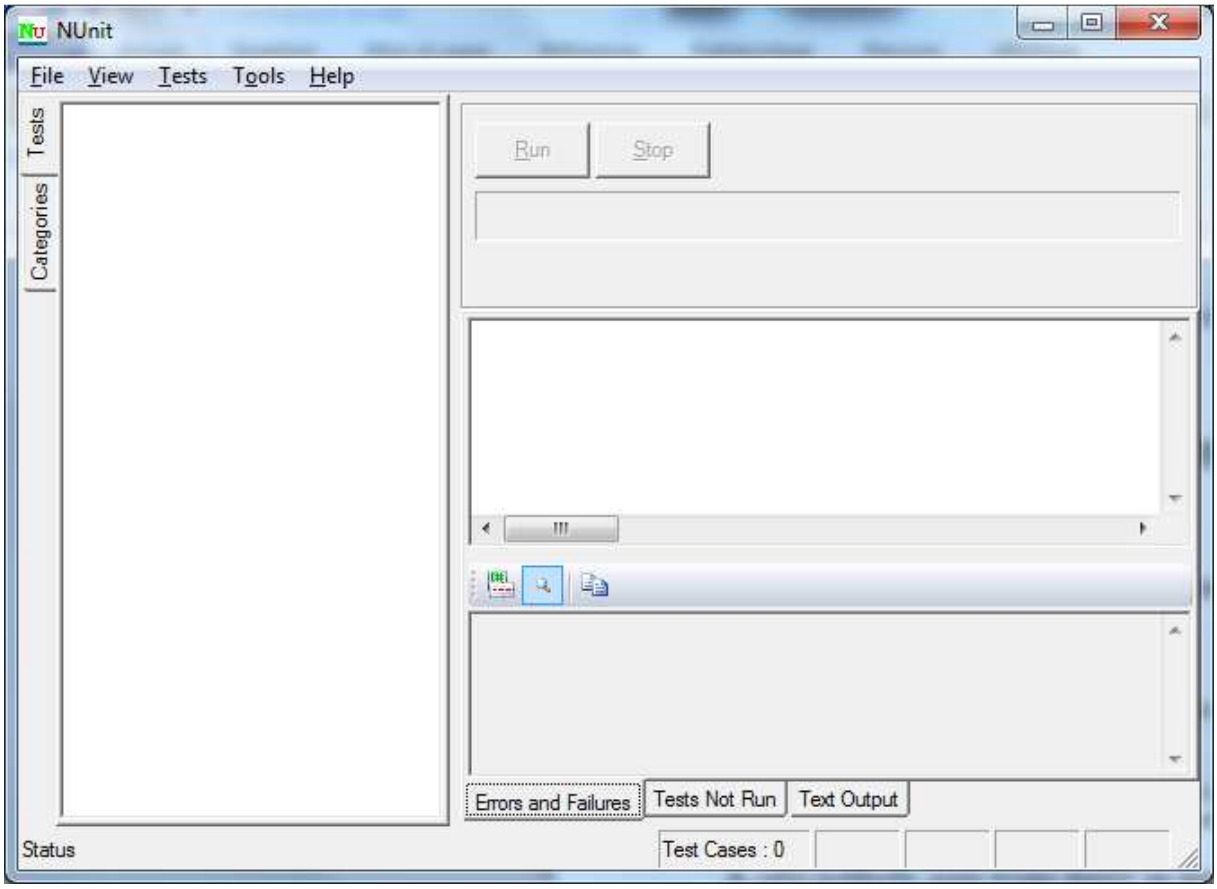
Elle permet de vérifier que la variable valeur vaut bien 6. Rajoutons une méthode de test qui échoue :

```
[Test]
public void Factorielle_ent10_result1()
{
    int valeur = 10;
    int resultat = MaBibliothequeATester.Math.Factorielle(valeur);
    Assert.AreEqual(1, resultat,"La valeur doit être 1");
}
```

A cette méthode, nous avons ajouté un message qui permet d'indiquer des informations complémentaires si le test échoue.

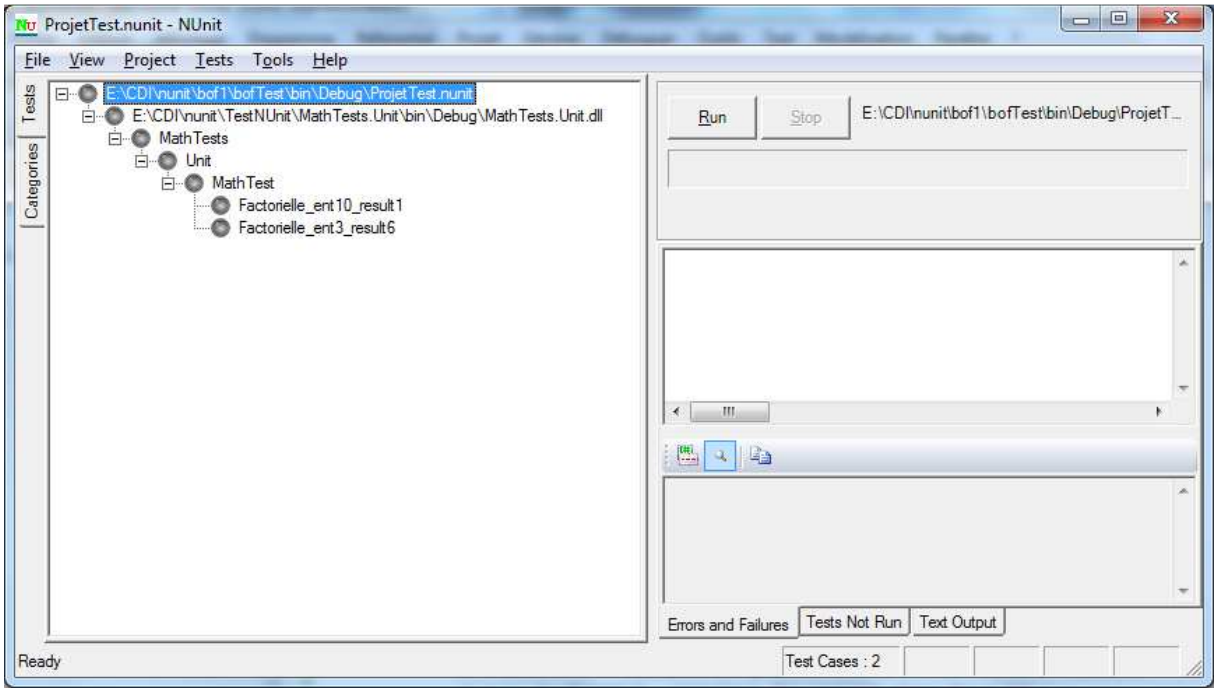
Il est important que chaque méthode qui s'occupe de tester une fonctionnalité, le fasse à l'aide d'un cas unique comme illustré juste au-dessus. La première méthode teste la fonctionnalité Factorielle pour le cas où la valeur vaut 3 et la seconde s'occupe du cas où la valeur vaut 10. Nous pouvons rajouter autant de méthodes de tests que vous le souhaitez tant qu'elles sont décorées de l'attribut [Test]. Si nous mettons plusieurs assertions par méthode de test, une erreur empêche l'exécution des autres tests

Compilons le projet et lançons l'application nunit.exe.

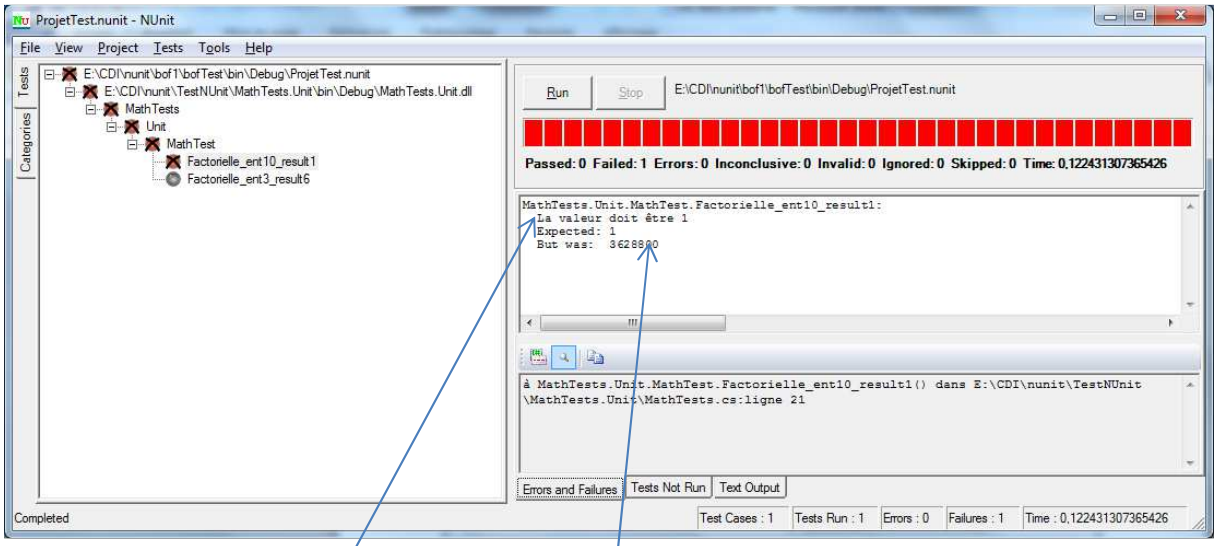


Créons un nouveau projet par exemple ProjetTest puis ajoutons une assembly en allant dans Project > Add Assembly.

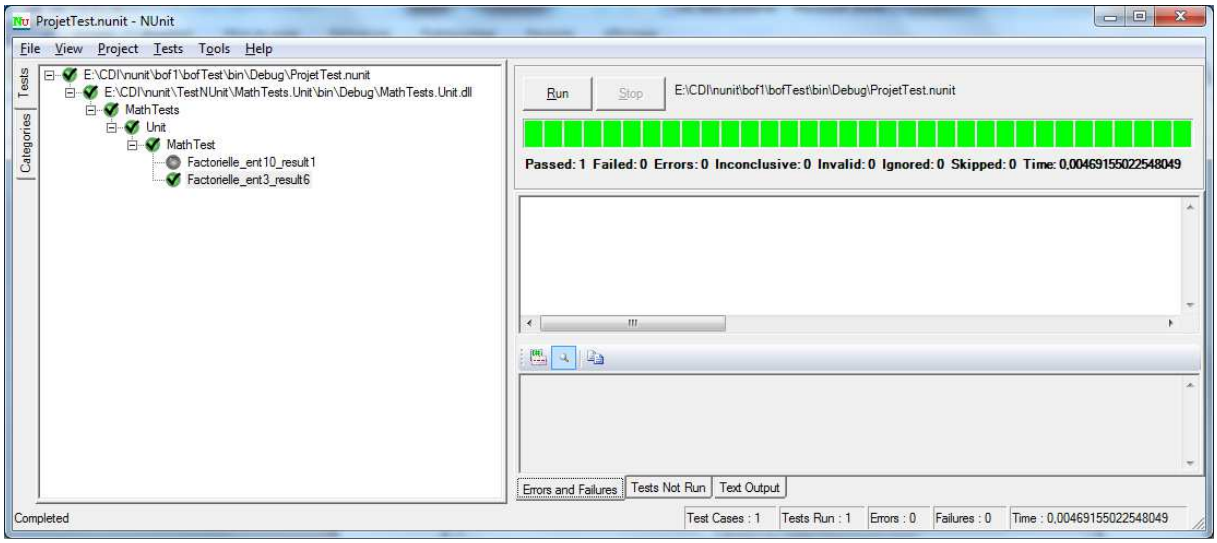
Pointez l'assembly de tests, à savoir MathTests.Unit.dll. NUnit analyse l'assembly et fait apparaître la liste des tests qui composent notre assembly, en se basant sur les attributs TestFixture et Test.



Nous pouvons à présent lancer les tests en cliquant sur Run ! On s'aperçoit qu'il y a un test qui passe (icône verte) et un test qui échoue (icône rouge).



Notre message apparait ici et la valeur obtenu là.



Nous devons modifier notre test mais le souci avec NUnit est qu'à partir du moment où il a chargé la dll pour lancer les tests, il n'est plus possible de faire de modifications, car toute tentative de compilation provoquera une erreur où il sera mentionné qu'il ne peut pas faire de modifications car le fichier est déjà utilisé ailleurs. Nous serons donc obligés de fermer puis de rouvrir NUnit. À noter que dans les versions payantes de Visual Studio, nous avons la possibilité de configurer NUnit en tant qu'outil externe.

Il y a beaucoup de méthodes permettant de vérifier si un résultat est correct. Regardons les assertions suivantes :

```
bool b = true;
Assert.IsTrue(b);
string s = null;
Assert.IsNull(s);
int i = 10;
Assert.Greater(i, 6);
```

La première permet de vérifier qu'une condition est vraie.
La deuxième permet de vérifier la nullité d'une variable.
La dernière permet de vérifier qu'une variable est bien supérieure à une autre.
Elles ont chacune leur pendant (IsFalse, IsNotNull, Less).

Il est également possible d'utiliser un attribut pour vérifier qu'une méthode lève bien une exception, par exemple :

```
[Test]
[ExpectedException(typeof(FormatException))]
public void ToInt32_AvecChaineNonNumerique_LeveUneException()
{
   Convert.ToInt32("abc");
}
```

Dans ce cas, le test passe si la méthode lève bien une FormatException.

Il est parfois nécessaire d'effectuer des opérations avant et après chaque test. Une classe peut donc définir des méthodes appelées avant et après chaque test avec les stéréotypes SetUp et TearDown. C'est l'endroit idéal pour factoriser des initialisations ou des nettoyages dont dépendent tous les tests.

```
[ TestFixture ]
public class MathTests
{
        [ SetUp ]
        public void InitialisationDesTests ()
        {
                // rajouter les initialisations
        }

        [ Test ]
        public void Factorielle_ent3_result6 ()
        {
                // test à faire
        }

        [ TearDown ]
```

```
        public void NettoyageDesTests ()
        {
                // nettoyer les variables , ...
        }
}
```

En résumé

- ✓ Les tests unitaires sont un moyen efficace de tester des bouts de code dans une application afin de garantir son bon fonctionnement.

- ✓ Ils sont un filet de sécurité permettant de faire des opérations de maintenance, de refactoring ou d'optimisation sur le code.

- ✓ Les frameworks de tests unitaires sont en général accompagnés d'outils permettant de superviser le bon déroulement des tests et la couverture de tests.