



# TESTS UNITAIRES

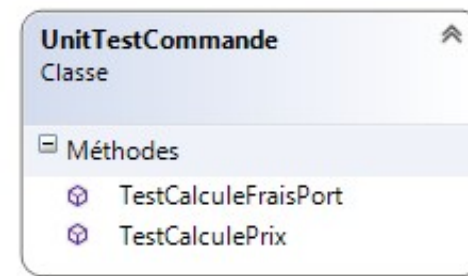
---

# Tests unitaires

- Jusqu'à présent, pour tester vos classes, vous avez travaillé dans le main pour :
  - Créer des instances
  - Déclencher les méthodes d'instances
- Mais le main a pour but de contenir les instructions d'un programme (éventuellement à destination d'un utilisateur), et non des instructions de tests.
- Il faut donc pour valider une classe : créer des classes de tests !

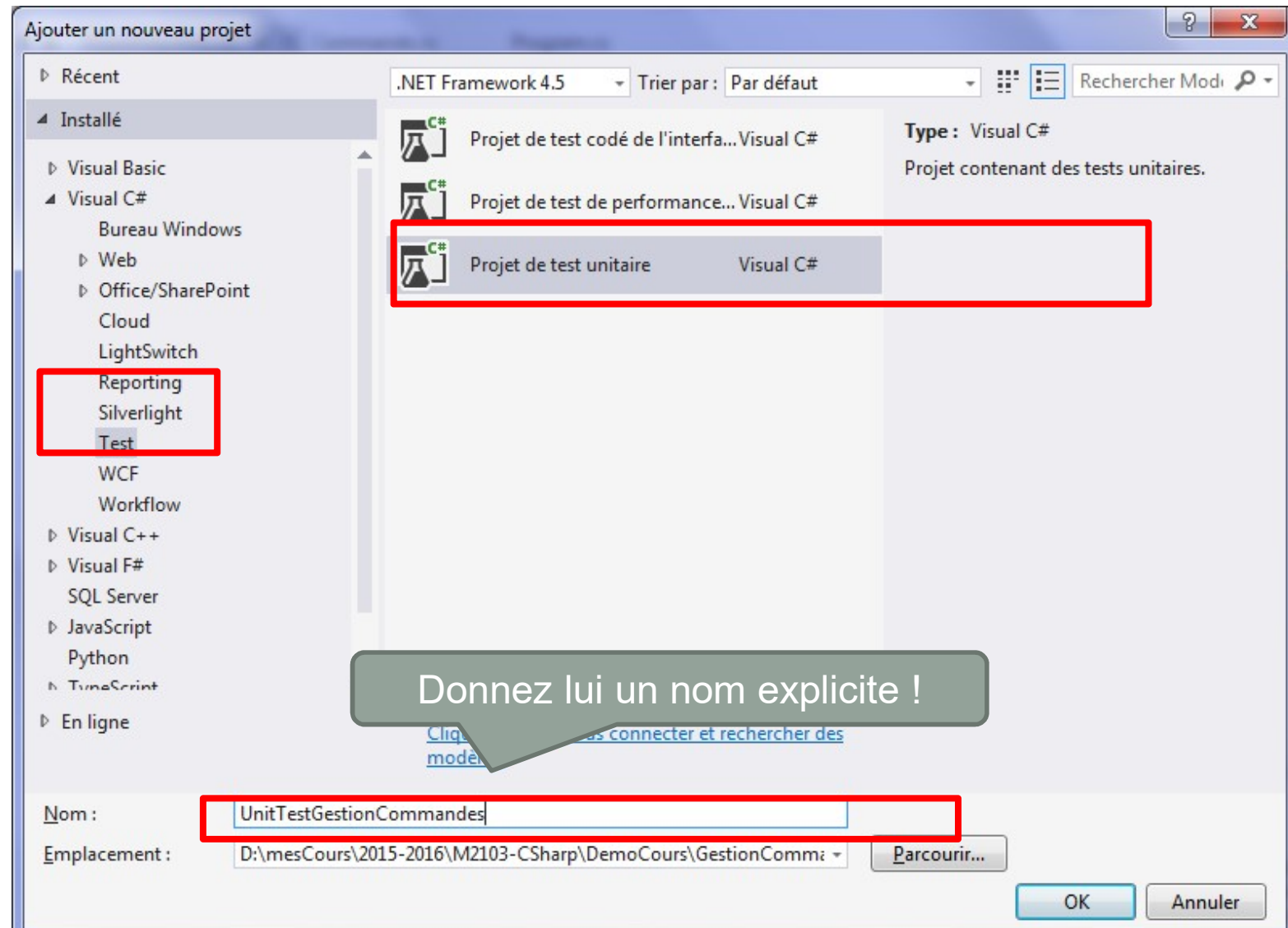
# Tests unitaires

- Pour valider une classe : il faut tester la classe.  
Ici, il faut tester :
  - Le constructeur : voir s'il lance bien des exceptions en cas de valeurs non conformes
  - Les méthodes : voir si le résultat retourné est bien le résultat attendu
- On peut automatiser les tests en faisant des classes de Test
- Une classe de test pour chaque classe ! Une méthode de test pour chaque méthode



# Tests unitaires

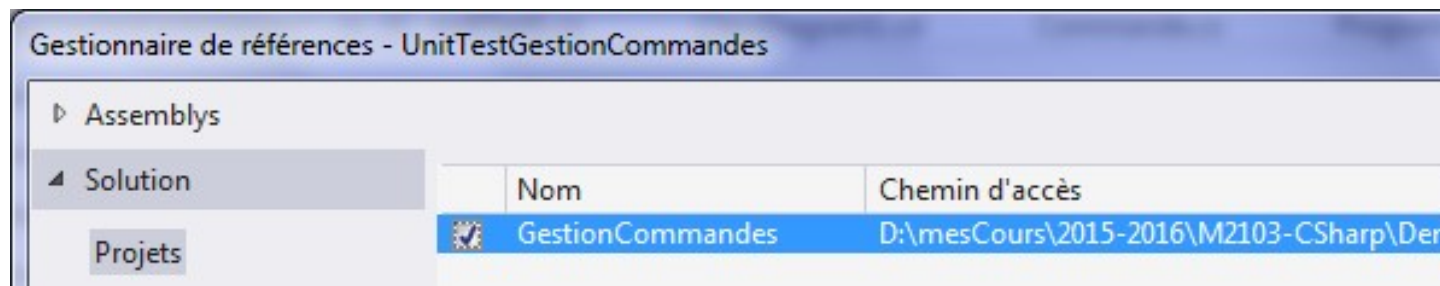
A partir de votre solution, ajoutez un projet de tests unitaires



# Tests unitaires

Liez votre projet de test à votre projet contenant le code source à tester :

- Ajoutez une référence à votre projet de test



- Indiquez dans le code de test le lien vers l'espace de nom contenant la classe Commande pour pouvoir l'utiliser sans préfixer ! Et indiquez à votre classe Commande qu'elle est publique !

```
using Microsoft.VisualStudio.TestTools.UnitTesting;  
using GestionCommandes;  
namespace UnitTestGestionCommandes  
{  
    [TestClass]  
    {  
        public class Commande  
        {
```

# Tests unitaires

Modifiez les noms donnés par défaut à votre classe de test et à la méthode de test pour être explicite !

```
namespace UnitTestGestionCommandes
{
    [TestClass]
    public class UnitTestCommande
    {
        [TestMethod]
        public void TestCalculePrix()
        {
        }
    }
}
```

On teste la classe  
Commande

On teste la méthode  
CalculePrix()

# Tests unitaires

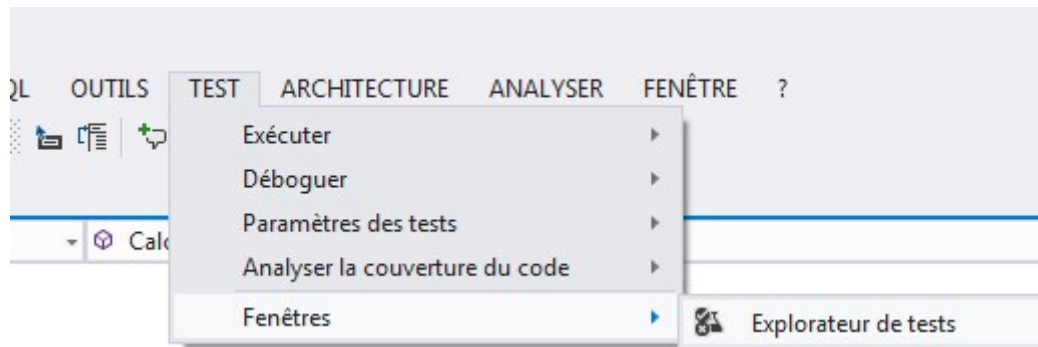
```
[TestClass]
public class UnitTestCommande
{
    [TestMethod]
    public void TestCalculePrix()
    {
        Commande c = new Commande("alimentation HP", 15.5, 3);
        Assert.Equals(46.5, c.CalculePrix());
    }
}
```

On teste si le retour de la méthode est bien égal (AreEquals) au résultat attendu

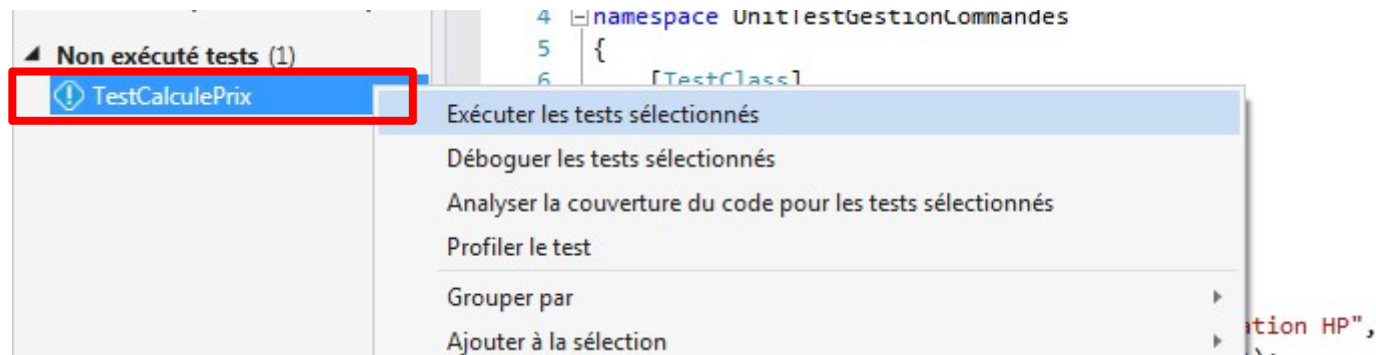
# Tests unitaires

Exécutez les tests :

- Générez la solution
- Ouvrez la fenêtre d'explorateur de tests



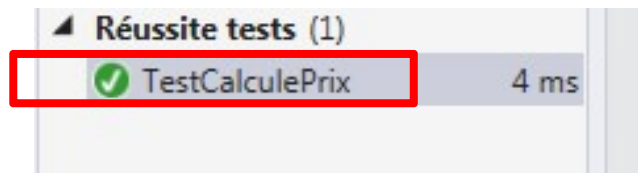
- Lancez le test





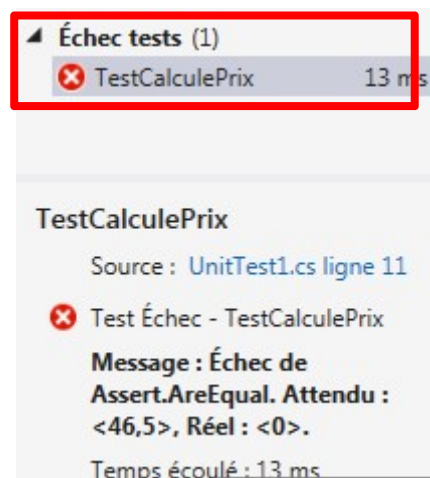
# Tests unitaires

- Soit le test a réussi : la méthode CalculeIMC est donc valide !



Flag indiquant la réussite du test : valide ou non la méthode !

- Soit le test a échoué : la méthode n'est pas valide. Il faut revoir le code car elle ne retourne pas le résultat attendu !



# Tests unitaires

- Bien penser son jeu de test ! Valeurs médianes mais aussi valeurs limites, il ne faut rien omettre, valeurs entières, non entières, ...
- Exemple pour la méthode CalculeFraisPort : les frais de port sont offerts à partir de 50 euros

Nature du jeu	Qte	Prix	Résultat attendu
Montant n'occasionnant pas de frais de port	2	45,50	0
Montant occasionnant frais de port	3	15,5	5
Montant à la limite	1	50	0

# Tests unitaires

Certains jeux de tests vont être commun aux différentes méthodes de test, il faut alors factoriser !

```
public void TestCalculePrix()
{
    Commande c = new Commande("alimentation HP", 15.5, 3);

    Assert.AreEqual(46.5, c.CalculePrix());
}
[TestMethod]
public void TestCalculeFraisPort()
{
    Commande c = new Commande("alimentation HP", 15.5, 3);

    Assert.AreEqual(5, c.CalculeFraisPort());
}
```

# Tests unitaires

Pour factoriser, il faut :

- Déclarer les variables du jeu de test dans la zone des variables de classe
- Initialiser les variables dans une méthode préfixée par le tag **[TestInitialize()]**

```
public class UnitTestCommande
{
    private Commande c;

    [TestInitialize()]
    public void init()
    {
        c = new Commande("alimentation HP", 15.5, 3);
    }

    [TestMethod]
    public void TestCalculePrix()
    {
        Assert.AreEqual(46.5, c.CalculePrix());
    }
}
```

# Tests unitaires

Les autre méthodes statiques de la classe Assert :

Méthode	Description
<b>AreEqual</b>	Teste l'égalité de deux objets.
<b>AreNotEqual</b>	Teste l'inégalité de deux objets
<b>AreSame</b>	Teste l'égalité de deux références d'objets
<b>AreNotSame</b>	Teste l'inégalité de deux références d'objets
<b>IsFalse</b>	Teste qu'une condition est fausse.
<b>IsTrue</b>	Teste qu'une condition est vraie.
<b>IsNull</b>	Teste qu'une référence d'objet est nulle.
<b>IsNotNull</b>	Teste qu'une référence d'objet est non nulle

# Tests unitaires

Pour tester le bon déclenchement d'une exception: il faut :

- utiliser le tag [ExpectedException
- Indiquer l'exception attendue
- Indiquer le message si l'exception n'a pas lieu
- Ecrire une méthode de test dans laquelle les instructions sont sensées déclencher une exception

```
[TestMethod]
[ExpectedException(typeof(ArgumentException),
    "La quantité est < 1. Cela devrait lancer une exception")]

public void QuantiteInf1InConstructor()
{
    Commande c = new Commande("alimentation HP", 15.5, 0);
}
```


# TDD : développement piloté par les tests

- Le Test Driven Development (TDD) une technique de développement de logiciel qui préconise d'écrire les tests unitaires avant d'écrire le code source d'un logiciel.
- Le cycle préconisé par TDD comporte cinq étapes :
  - écrire un premier test ;
  - vérifier qu'il échoue (car le code qu'il teste n'existe pas), afin de vérifier que le test est valide ;
  - écrire juste le code suffisant pour passer le test ;
  - vérifier que le test passe ;
  - puis refactoriser le code, c'est-à-dire l'améliorer tout en gardant les mêmes fonctionnalités.

# TDD : VisualStudio

- On code d'abord le test :

```
[TestMethod]
Oréférences
public void TestCalculeTotal()
{
    Assert.AreEqual(51.5, c.CalculeTotal());
}
```



- On demande à VisualStudio de générer le squelette du code de la méthode (générer → stub de la méthode) :

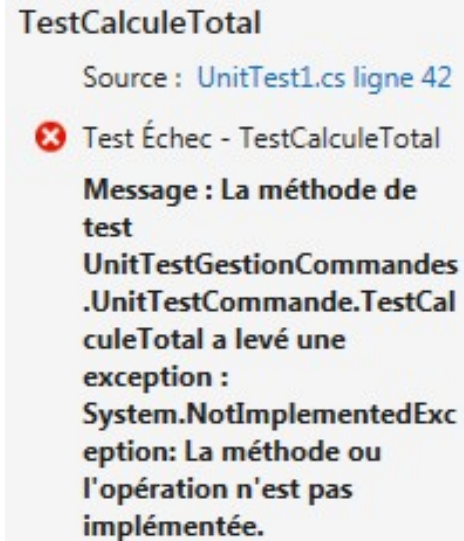
```
public object CalculeTotal()
{
    throw new NotImplementedException();
}
```

Précise que le code n'est pas encore implémenté



## TDD : VisualStudio

- On peut alors lancer le test qui ne marche pas immédiatement car la méthode n'est pas encore codée :



The screenshot shows a test failure in Visual Studio. The title is 'TestCalculeTotal'. Below it, the source is listed as 'Source : UnitTest1.cs ligne 42'. A red 'X' icon indicates a failure. The message reads: 'Test Échec - TestCalculeTotal', 'Message : La méthode de test', 'UnitTestGestionCommandes', '.UnitTestCommande.TestCalculeTotal a levé une exception :', 'System.NotImplementedException: La méthode ou l'opération n'est pas implémentée.'

TestCalculeTotal  
Source : UnitTest1.cs ligne 42  
❌ Test Échec - TestCalculeTotal  
Message : La méthode de test  
UnitTestGestionCommandes  
.UnitTestCommande.TestCalculeTotal a levé une exception :  
System.NotImplementedException: La méthode ou l'opération n'est pas implémentée.

- Il suffit alors de la coder et tester à nouveau et cela jusqu'à ce que le test soit réussi.