

COLLECTION DE DONNÉES

Tableaux / listes

- Pour gérer un ensemble d'informations de même type
- Choix entre tableaux et listes:
 - Tableau : taille fixe et multidimensionnel

=> utile pour des constantes ou données paramètres dont le nombre est fixé. Par exemple : jour de la semaine, tarifs

- Liste : taille variable

=> utile pour des listes d'informations variables : liste de produits, de clients ...

Tableaux

- Comme en C : taille fixe
- Initialisation lors de la déclaration : pas de taille

```
String [ ] jours = new String [ ] { "Lundi", "Mardi",  
"Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche" };
```

- Déclaration puis initialisation

```
String [ ] jours = new String[7];  
jours[0] = "Lundi";  
jours[1] = "Mardi";  
jours[2] = "Mercredi";  
jours[3] = "Jeudi";  
jours[4] = "Vendredi";  
jours[5] = "Samedi";  
jours[6] = "Dimanche";
```

new = allocation
mémoire contigue

Parcours de tableau

- Pour le lire :

```
foreach (String jour in jours)  
{  
    Console.WriteLine(jour);  
}
```

```
for (int i=0 ; i < jours.Length ; i++)  
{  
    Console.WriteLine( jours[i] );  
}
```

- Pour le modifier :

```
for (int i=0 ; i < jours.Length ; i++)  
{  
    jours[i] = jours[i] + "s";  
}
```

Méthodes pour les tableaux

Méthodes statiques de classe Array rangées dans l'espace de nom System :

- int IndexOf(Array array, Object value) : retourne la position de la 1ere occurrence de la valeur au sein du tableau
- int LastIndexOf(Array array, Object value) : retourne l'index de la dernière occurrence de la valeur au sein du tableau
- void Reverse (Array) : inverse
- void Sort (Array) : trie

```
String [ ] jours = new String [ ] { "Lundi", "Mardi", "Mercredi", ... "Dimanche" };  
int posMercredi = Array.IndexOf(jours, "Mercredi");  
Console.WriteLine("Mercredi c'est le jour " + (posMercredi+1));
```



```
Array.Sort(jours);
```

Listes : list <T>

- Initialisation lors de la déclaration : pas de taille

```
List<String> prenoms = new List<String> { "Franck", "Aimerick", ... };
```

- Déclaration puis initialisation à l'aide de la méthode Add:

```
List<String> prenoms = new List<String>(); // création de la liste
```



```
prenoms.Add("Franck");  
prenoms.Add("Aimerick");  
prenoms.Add("Elodie");
```

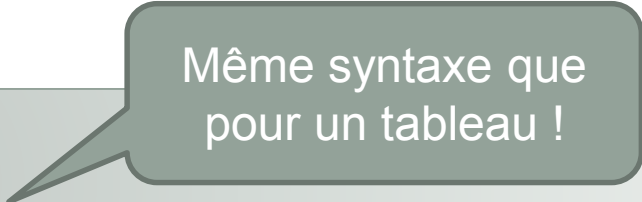
Parcours de liste

- Pour le lire :

```
foreach (String prenom in prenoms)  
{  
    Console.WriteLine(prenom);  
}
```

- Pour le modifier :

```
for (int i = 0; i < prenoms.Count; i++)  
    prenoms[i] = prenoms[i] + "s";
```



Même syntaxe que
pour un tableau !

Méthodes pour les listes : list <T>

Méthodes pour les objets de classe List :

- int IndexOf(T value) : retourne la position de la 1ere occurrence de la valeur au sein du tableau
- int LastIndexOf(T value) : retourne l'index de la dernière occurrence de la valeur au sein du tableau
- void Reverse () : inverse
- void Sort () : trie
- void Insert(int index, T item)
- ...

```
List<String> prenom = new List<String>(); // création de la liste
prenom.Add("Franck");
prenom.Add("Aimerick");
prenom.Add("Elodie");
```



```
prenom.Sort();
```


CLASSE PLUS ELABORÉE

Affiner l'accessibilité

- Jusqu'à présent, on a fait des champs privés avec un accès en lecture/écriture via des propriétés publiques
- Mais on peut faire:
 - Des champs privés avec un accès uniquement en lecture
 - Des champs totalement privés
 - Des champs publics
 - Des champs publics pour le projet : Internal
 - Des champs protected : étudié plus tard

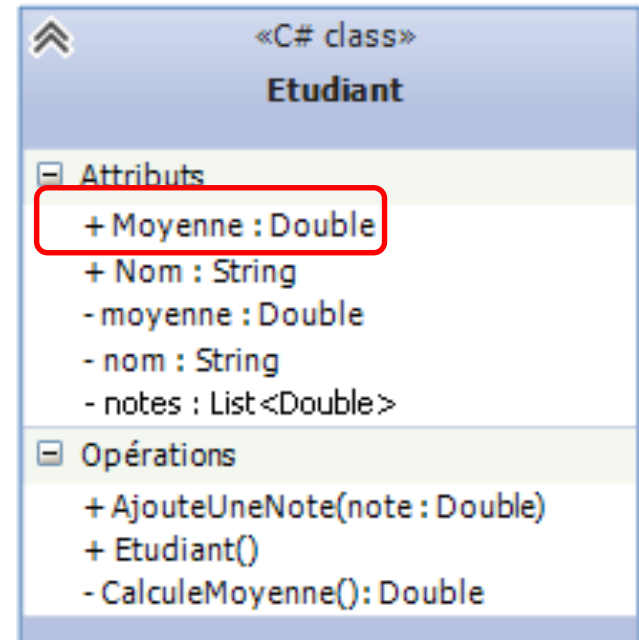
Champs en lecture seule

- Très utiles pour stocker des données calculées. Remarque : on décide ici de stocker une donnée calculée pour optimiser les traitements

```
public double Moyenne
{
    get { return this.moyenne; }

    private set {

        if (value < 0)
            throw new ArgumentException("La moyenne doit
            moyenne = value; }
}
```



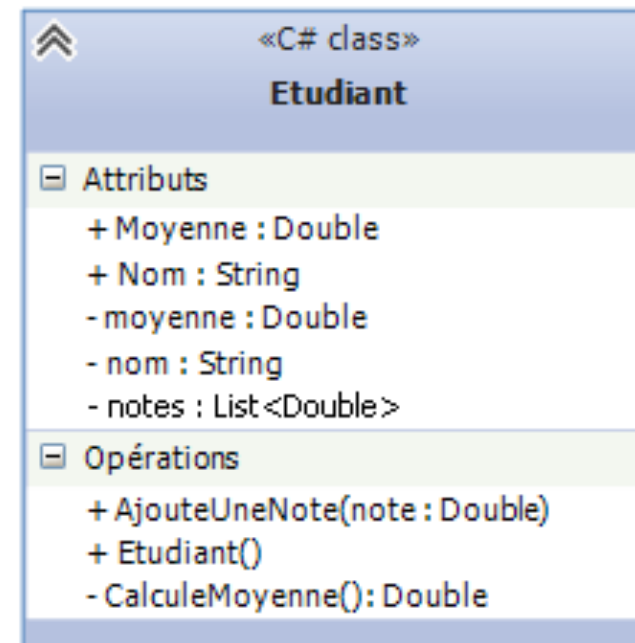
Ici, la moyenne résulte de la liste de notes, elle ne peut donc être modifiée de l'extérieur : trop grand risque d'incohérence !

Champs en lecture seule

- Mis à jour uniquement par des traitements internes à la classe.

```
public double Moyenne
{
    get { return moyenne; }
    private set {
        if (value < 0)
            throw new ArgumentException("La moyenne ne peut pas être négative");
        this.moyenne = value; }
}

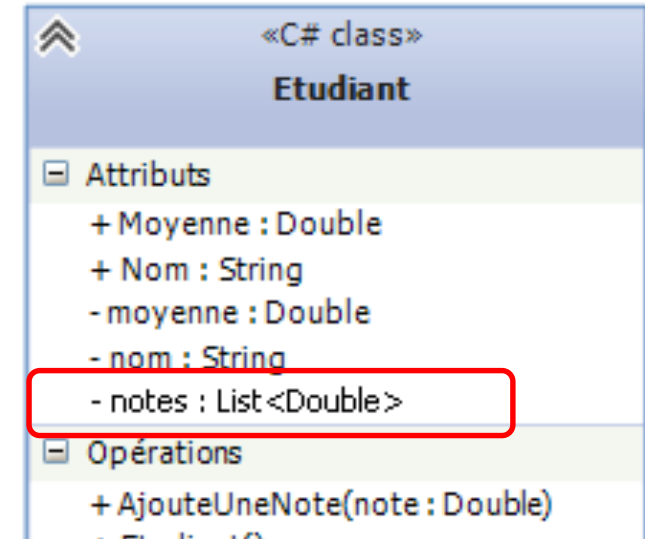
public void AjouteUneNote(double note)
{
    if (note < 0)
        throw new ArgumentException("La note doit être comprise entre 0 et 20");
    this.notes.Add(note);
    this.Moyenne = CalculeMoyenne ( ) ;
}
```



```
Etudiant e = new Etudiant("Dubars");
e.Moyenne = 15;
e.AjouteUneNote(12);
Console.Write ( "Moyenne " + e.Moyenne );
```

Champs totalement privés

- Très utile pour stocker:
 - des données internes à manipuler avec précaution. Exemple : la liste de notes...
 - Des données à ne pas diffuser
- Pas de propriétés publiques !



Champs statiques

- Un champ statique : pour stocker une donnée commune à tous les objets de la classe ! Stockée une seule fois en mémoire, dissociée des objets !

```
class Commande
{
    private static double tarifLivraison = 3;

    public static double TarifLivraison
    {
        get { return Commande.tarifLivraison; }
        set { Commande.tarifLivraison= value; }
    }
}
```

```
Commande c1 = new Commande(1,"alimentation HP", 15.5);
Commande c2 = new Commande(1,"sacoche HP", 29.90);
Console.WriteLine(Commande.TarifLivraison);
Commande.TarifLivraison = 5;
```

On y accède depuis la classe

tarifLivraison :3

- quantite : 1
- libelleArticle : sacochHP
- prixUnitaire : 29,90

- quantite : 1
- libelleArticle : alimentation HP
- prixUnitaire : 15,5

Champs statiques

- On peut jouer sur son accessibilité. Ici, seule la classe pourra modifier sa valeur

```
class Commande
{
    private static double tarifLivraison = 3;

    public static double TarifLivraison
    {
        get { return Commande.tarifLivraison; }
        private set { Commande.tarifLivraison= value; }
    }
}
```

```
Commande c1 = new Commande(1,"alimentation HP", 15.5);
Commande c2 = new Commande(1,"sacoche HP", 29.90);
Console.WriteLine(Commande.TarifLivraison);
```

On y accède depuis la classe

tarifLivraison :3

- quantite : 1
- libelleArticle : sacochéHP
- prixUnitaire : 29,90

- quantite : 1
- libelleArticle : alimentation HP
- prixUnitaire : 15,5

Champs statiques

- Très utile aussi pour faire une donnée auto incrémentée : un identifiant.

```
class Commande
{
    private static int numAuto = 0;

    public static int NumAuto
    {
        get {
            Commande.numAuto++;
            return Commande.numAuto;
        }
        set { Commande.numAuto = value; }
    }

    public Commande(String pLibelle, double pPrixUnit, int pQuantite)
    {
        this.Id = Commande.NumAuto;
        this.LibelleArticle = pLibelle;
        this.PrixUnitaire = pPrixUnit;
        this.Quantite = pQuantite;
    }
}
```

tarifLivraison :3
numAuto : 2

- id : 1
- quantite : 1
- libelleArticle :
sacocheHP
- prixUnitaire :
29,90

- id:2
- quantite : 1
- libelleArticle :
alimentation HP
- prixUnitaire :
15,5

Constantes (champs implicitement statiques)

- Une constante est un champ statique qui ne peut pas être modifié : elle est souvent publique.

```
class Commande  
{  
    public const double MONTANT_FRAIS_PORT_OFFERT = 50;
```

```
Commande c1 = new Commande(1,"alimentation HP", 15.5);  
Commande c2 = new Commande(1,"sacoche HP", 29.90);  
Console.WriteLine(Commande.MONTANT_FRAIS_PORT_OFFERT);
```

On y accède depuis la classe

**MONTANT_FRAIS_PORT
_OFFERT : 50**

- quantite : 1
- libelleArticle :
sacocheHP
- prixUnitaire :
29,90

- quantite : 1
- libelleArticle :
alimentation HP
- prixUnitaire :
15,5

Champs statiques / champs d'instance

- Par défaut, les champs sont d'instances.

Champ d'instance	Champ statique
Donnée propre à un objet	Donnée commune à tous les objets
Appartient à l'objet	Appartient à la classe
Initialisée au sein du constructeur	Initialisée dès sa déclaration

Enum

- Pour définir une énumération

```
public enum SexeOfIndividu { F = 'F', M = 'M'};

private SexeOfIndividu sexe;
public SexeOfIndividu Sexe
{
    get { return sexe; }
    set { sexe = value; }
}
```

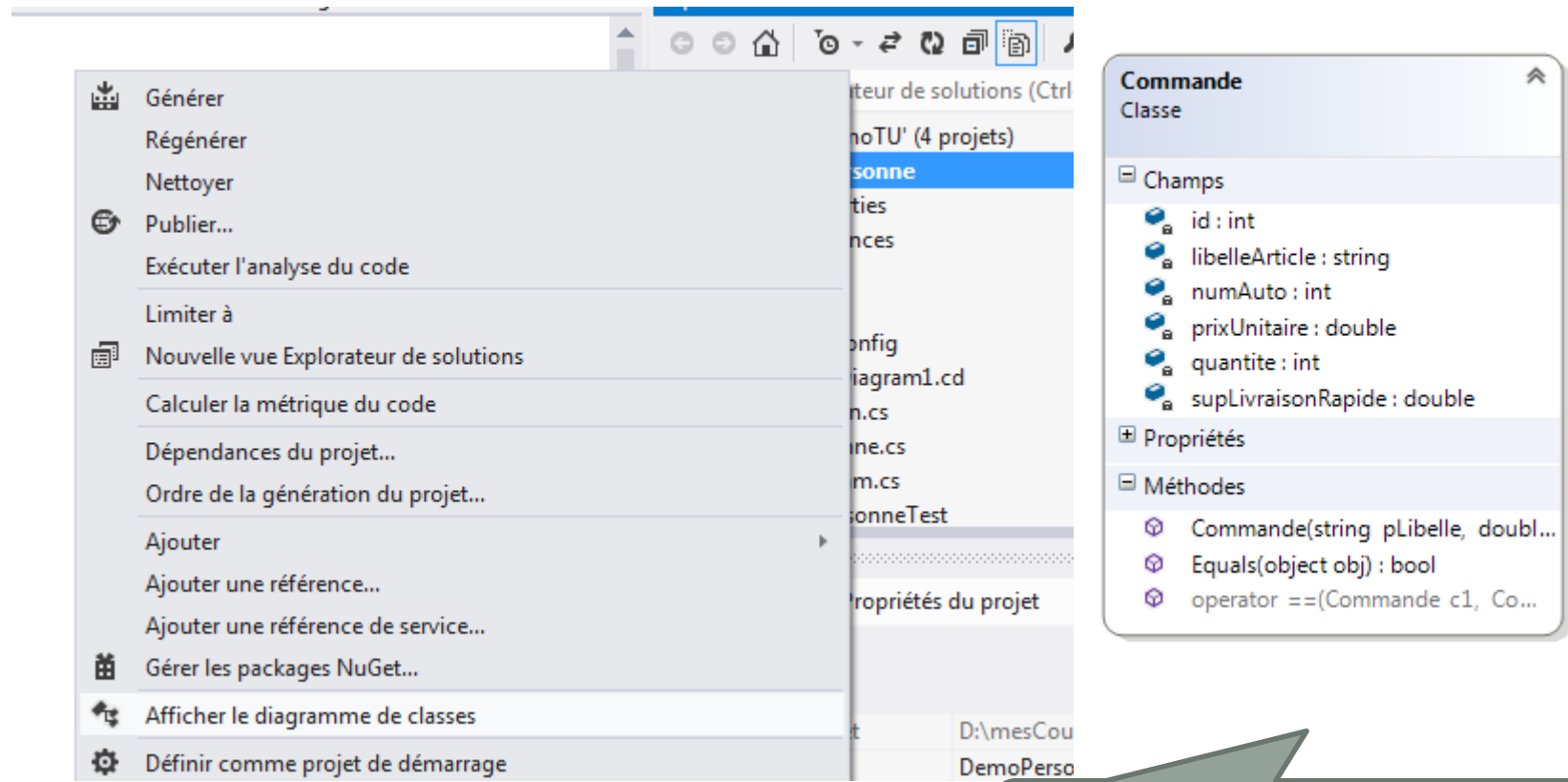
- Et éviter cela :

```
public const char SEXE_F = 'F';
public const char SEXE_M = 'M';

private char sexe;
public char Sexe
{
    get { return sexe; }
    set { if ( value != SEXE_F && value != SEXE_M )
            throw new ArgumentException (« valeur inattendue »);
        sexe = value ;
    }
}
```

Générer un diagramme UML : **rétro ingénierie**

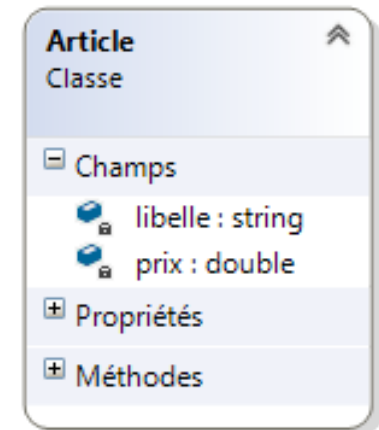
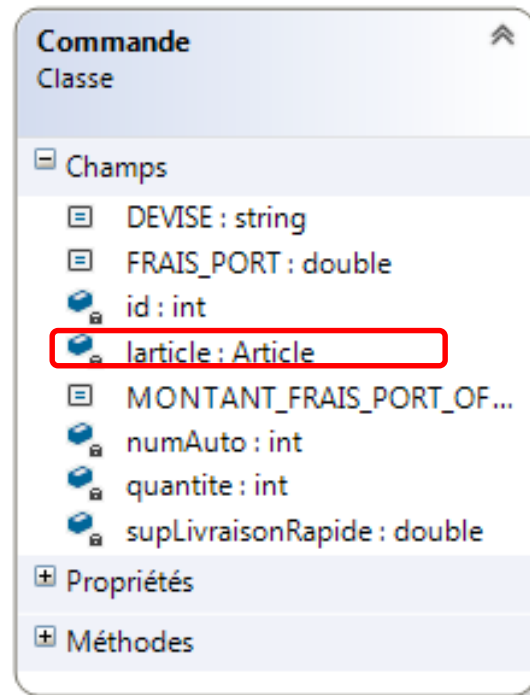
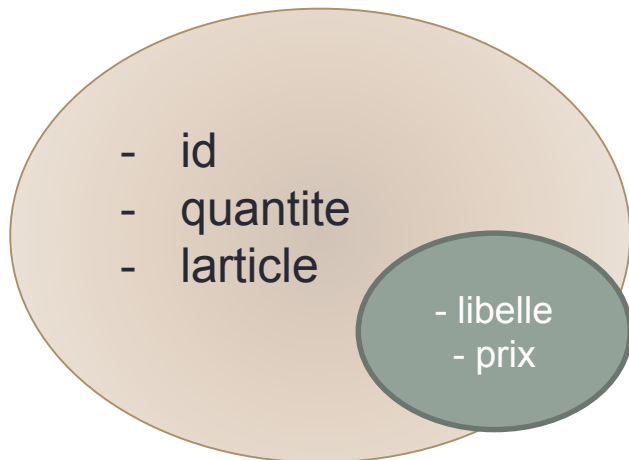
- Visual studio peut générer le diagramme de classe à partir de votre code.



Attention : toute modification entraîne une modification du code

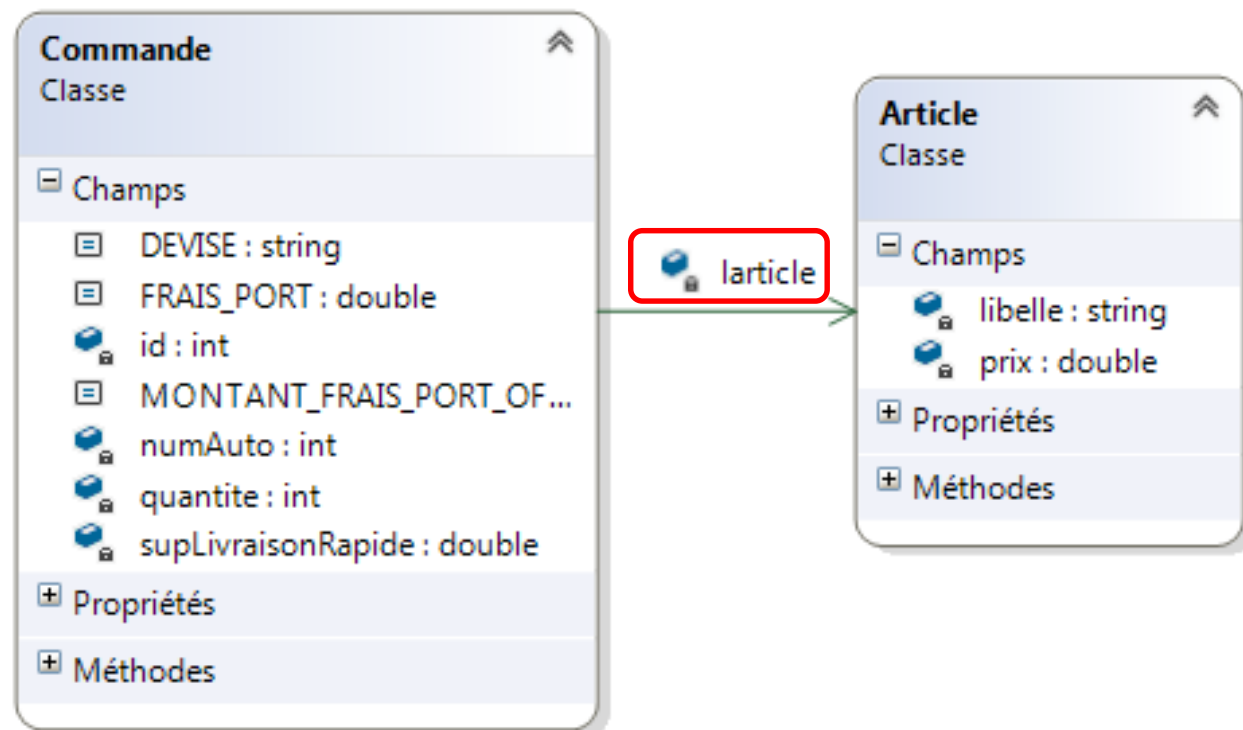
Association simple

- Un objet composé d'un autre objet !



Association simple

- Le champ peut être représenté sous forme d'association.



Association simple

- Généralement, on fera alors une surcharge du constructeur:
 - Une version avec l'objet passé en paramètre

```
public Commande(Article a, int pQuantite)
{
    this.Id = NumAuto;
    this.Quantite = pQuantite;
    this.Larticle = a;
}
```

```
Article a = new Article ( "Alimentation HP", 15.5);
Commande c = new Commande(a, 1);
```

Pour instancier une commande, il faudra déjà instancier l'article


- Une version avec les données nécessaires à la création de l'objet en interne

```
public Commande(String pLibelle, double pPrixUnit, int pQuantite)
{
    this.Id = NumAuto;
    this.Quantite = pQuantite;
    this.Larticle = new Article(pLibelle, pPrixUnit);
}
```

```
Commande c = new Commande("Alimentation HP", 15.5, 1);
```

Association simple

- Rappel pour factoriser le code : mot clef `this`

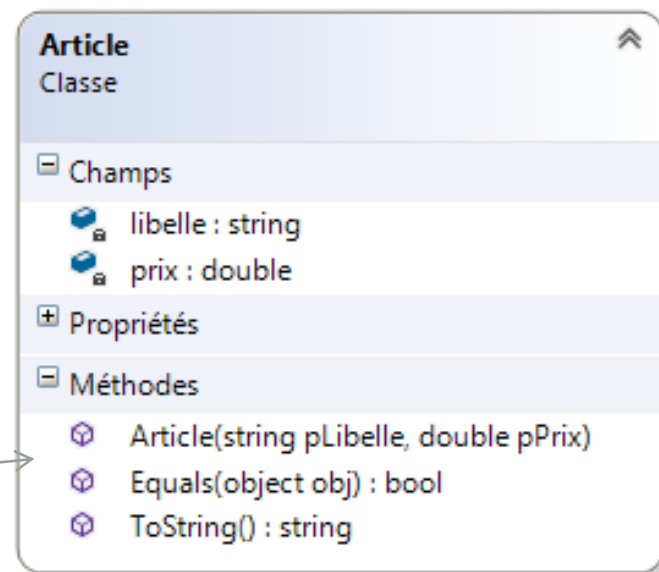


```
public Commande(Article a, int pQuantite )  
{  
    this.Id = Commande.NumAuto;  
    this.Quantite = pQuantite;  
    this.Larticle = a;  
}  
  
public Commande(String pLibelle, double pPrixUnit, int pQuantite)  
: this ( new Article(pLibelle, pPrixUnit), pQuantite ) { }
```


Association simple

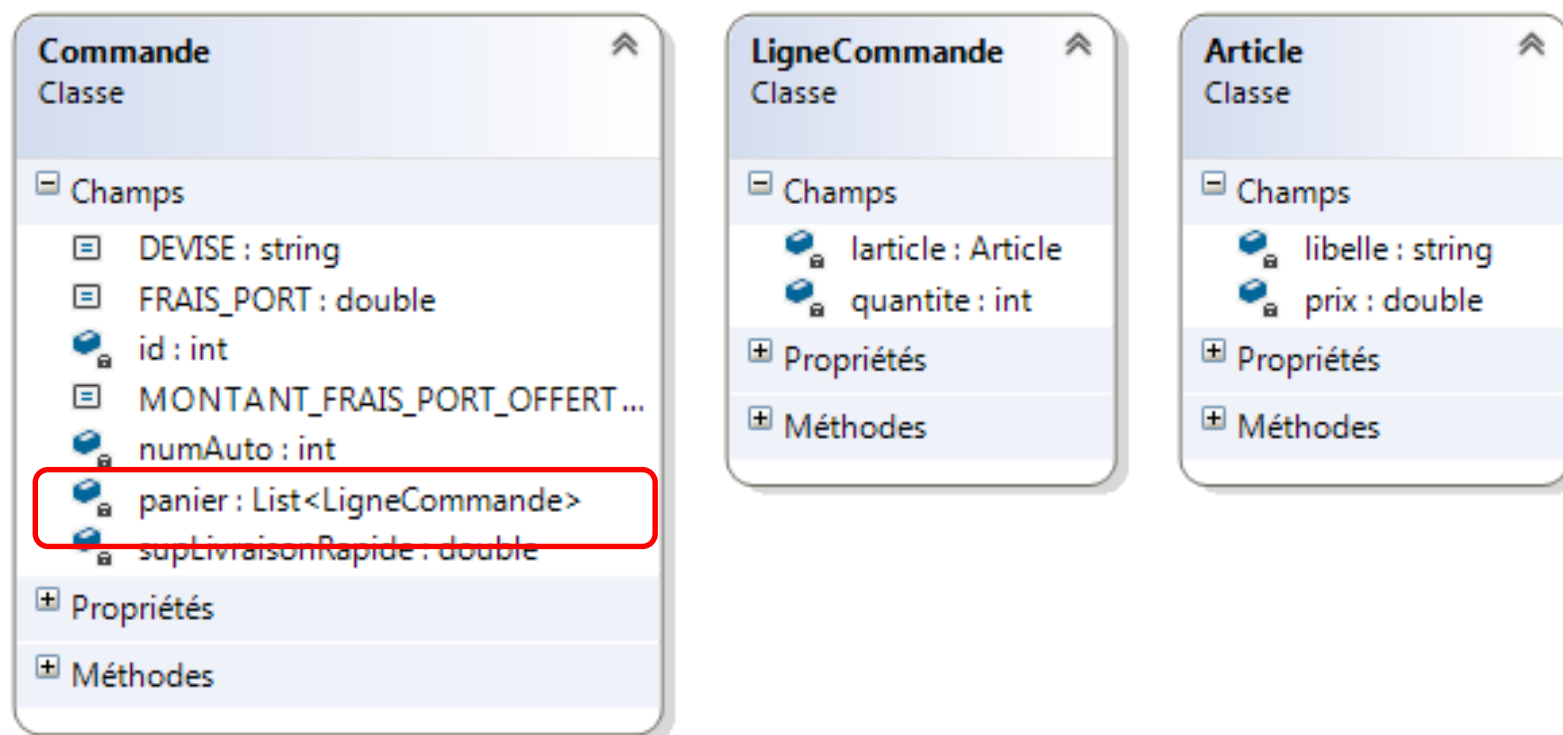
- On pourra déclencher au niveau de la classe Commande, les méthodes de la classe Article par l'intermédiaire de l'objet l'article

```
Class Commande
{
    public override bool Equals(object obj)
    {
        if (obj == null )
            return false;
        if ( obj.GetType() != this.GetType() )
            return false ;
        Commande c = obj as Commande;
        return Larticle.Equals(c.Larticle)
            && this.Quantite == c.Quantite;
    }
}
```



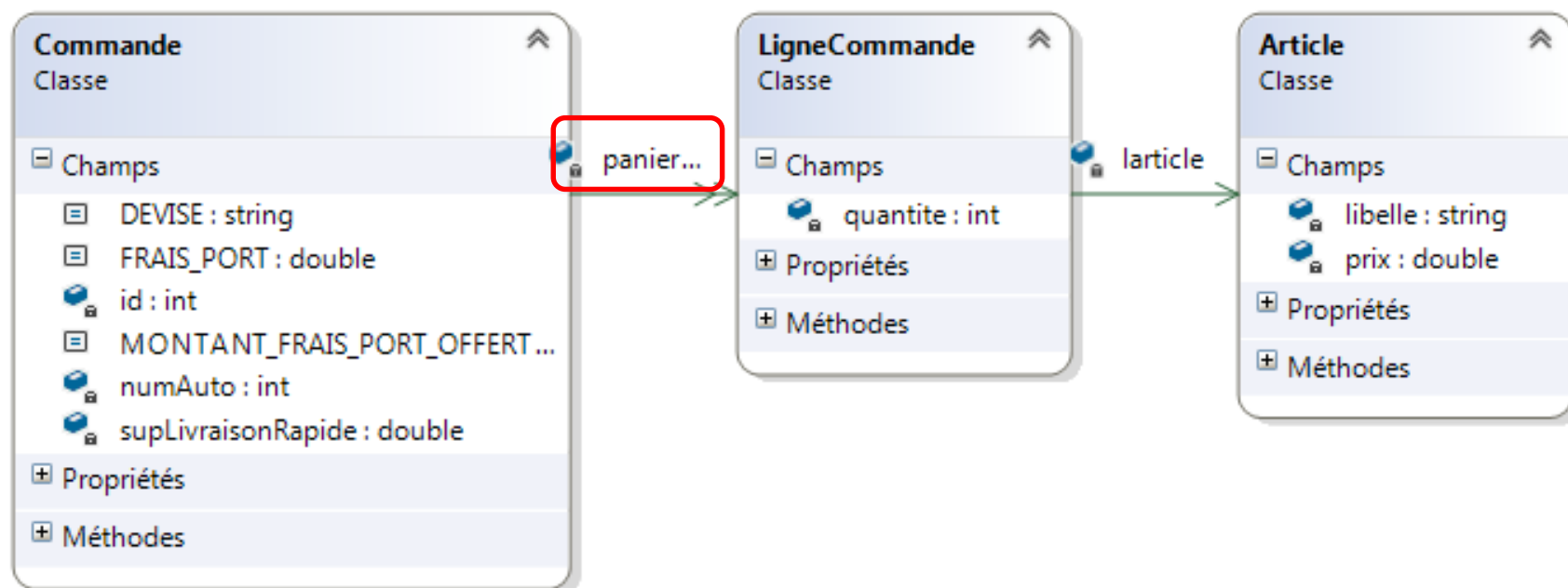
Association multiple

- Les variables peuvent être une collection d'objets.



Association multiple

- La collection peut être représentée sous forme d'une association



Association multiple

- Généralement, on fera alors une surcharge du constructeur:
 - Une version avec la collection passée en paramètre

```
public Commande(List<LigneCommande> l)  
{  
    this.Id = Commande.NumAuto;  
    this.Panier = l;  
}
```

```
List<LigneCommande> l = new List<LigneCommande>();  
l.Add(new LigneCommande(1, new Article("Alimentation HP", 15.5)));  
l.Add(new LigneCommande(2, new Article("Sacoche HP", 29.9)));  
Commande c = new Commande(l);
```

Pour instancier une commande, il faudra déjà instancier une liste de lignes de commande

Association multiple

- Généralement, on fera alors une surcharge du constructeur:
 - Une version « allégée »

```
public Commande()  
{  
    this.Id = Commande.NumAuto;  
    this.Panier = new List<LigneCommande>();  
}  
public void AjouteUneLigne( int pQuantite, String pLibelle, double pPrix)  
{  
    this.Panier.Add(new LigneCommande(pQuantite, new Article(pLibelle, pPrix)));  
}
```

```
Commande c = new Commande();  
c.AjouteUneLigne(1, "Alimentation HP", 15.5);  
c.AjouteUneLigne(2, "Sacoche HP", 29.9);
```

+ facile à
utiliser !

Association multiple

- Il ne faudra pas oublier que cette variable est une collection de données : penser aux boucles. Exemple avec ToString :

```
public override string ToString()
{
    String txt = "\nCommande n°: " + id;
    foreach ( LigneCommande l in panier)
        { txt = txt + l.ToString(); }
    return txt;
}
```

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 !

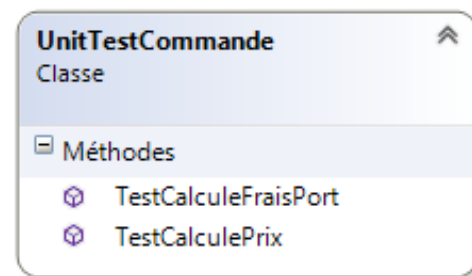
Tests unitaires

- Pour valider une classe : il faut tester la classe.

Ici, il faut tester :

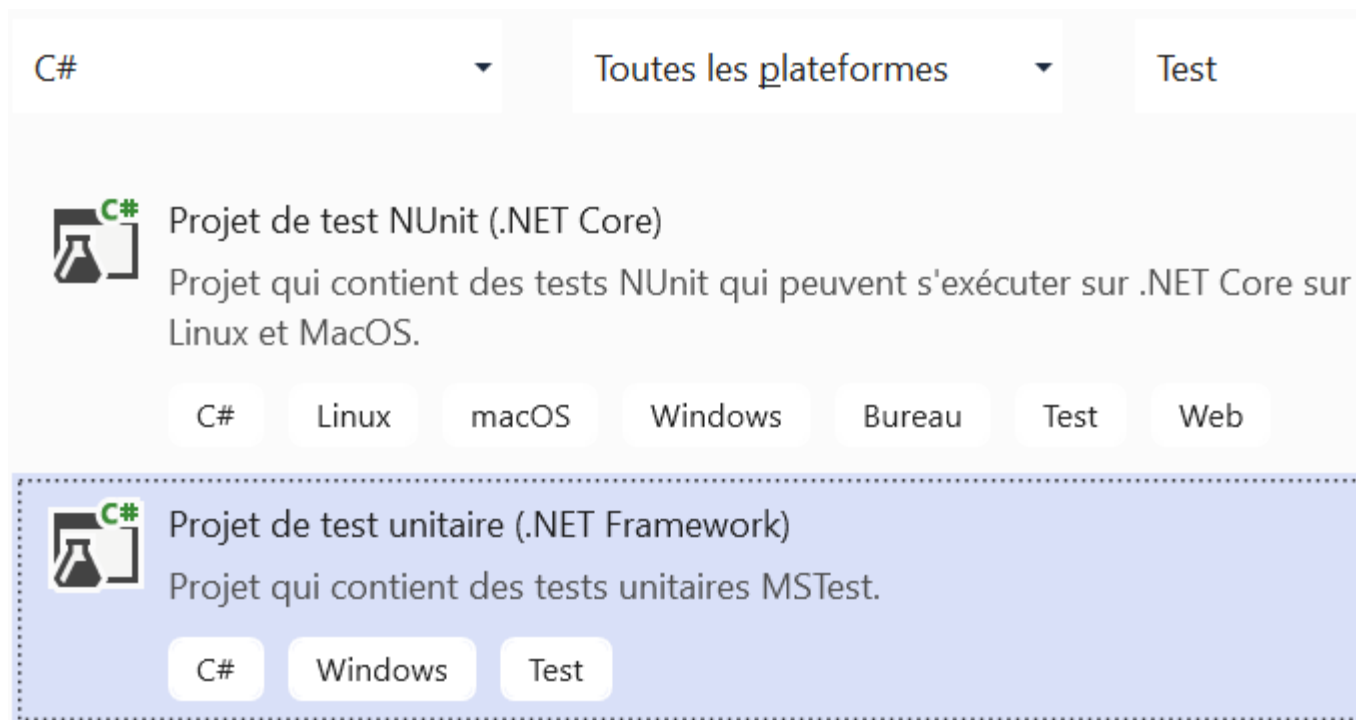
- Le constructeur : voir s'il lance bien des exceptions en cas de valeurs non conformes
- Les méthodes : voir si le résultat retourné est bien le résultat attendu

- 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

Configurer votre nouveau projet

Projet de test unitaire (.NET Framework)

C#

Windows

Test

Nom du projet

UnitTestGestionCommandes

Emplacement

D:\IUT-ACY\mesCours\2019-2020\M21\demo\TP1

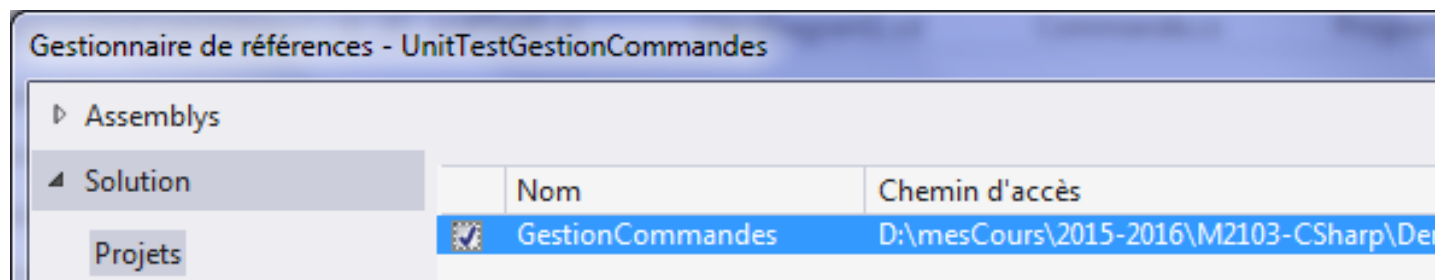


Donnez lui un nom explicite !

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

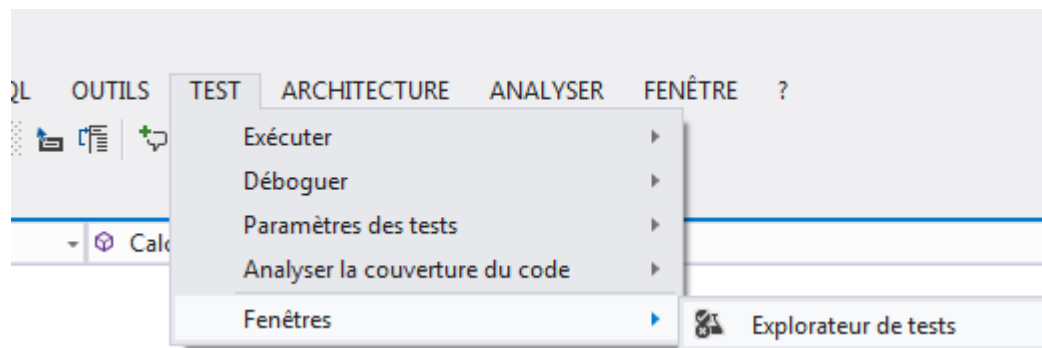
```
[TestClass]
public class UnitTestCommande
{
    [TestMethod]
    public void TestCalculePrix()
    {
        Commande c = new Commande("alimentation HP", 15.5, 3);
        Assert.AreEqual(46.5, c.CalculePrix());
    }
}
```

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

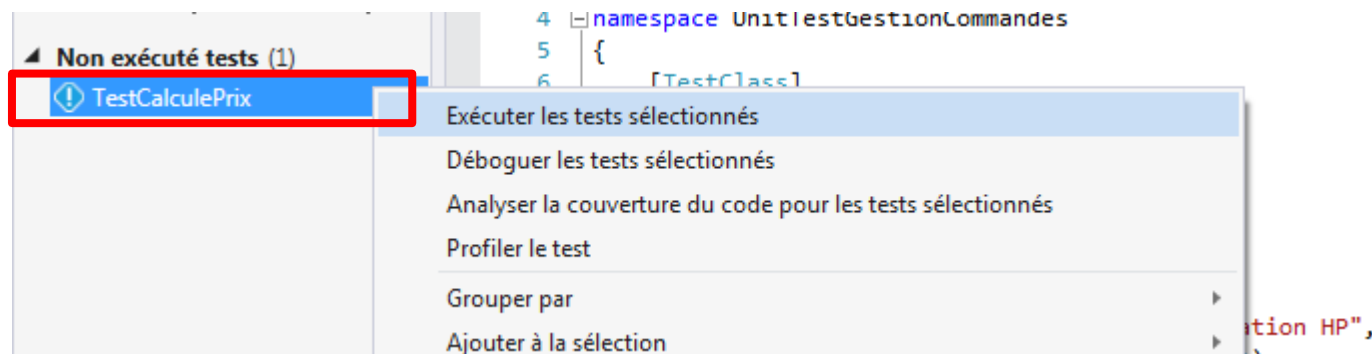
Tests unitaires

Exécutez les tests :

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

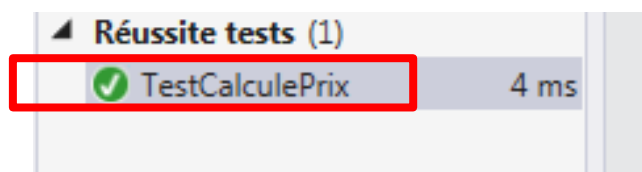


- Lancez le test



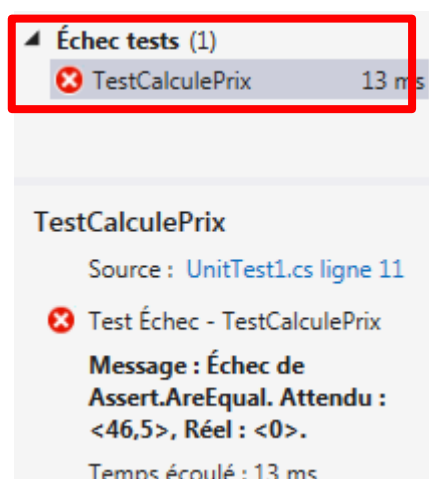
Tests unitaires

- 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
AreEqual	Teste l'égalité de deux objets.
AreNotEqual	Teste l'inégalité de deux objets
AreSame	Teste l'égalité de deux références d'objets
AreNotSame	Teste l'inégalité de deux références d'objets
IsFalse	Teste qu'une condition est fausse.
IsTrue	Teste qu'une condition est vraie.
IsNull	Teste qu'une référence d'objet est nulle.
IsNotNull	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());
}
```



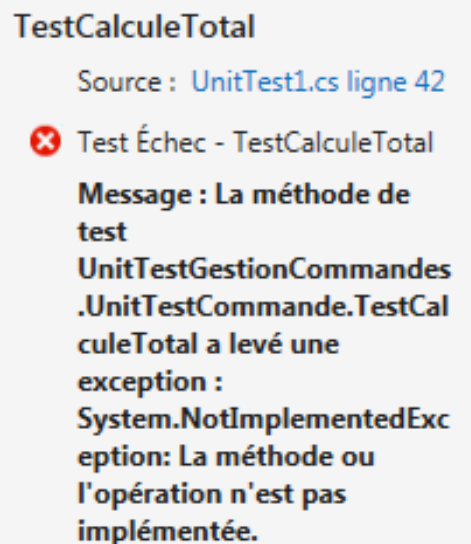
- On demande à VisualStudio de générer le squelette du code de la méthode (générer → stub de la méthode) :

```
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. At the top, it says 'TestCalculeTotal'. Below that, it indicates the source as 'UnitTest1.cs ligne 42'. A red 'X' icon is followed by the text 'Test Échec - TestCalculeTotal'. The message states: '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.TestCal
culeTotal a levé une
exception :
System.NotImplementedExc
eption: 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.