

BTS SIO
Bloc 2 – TP2
Programmation C#

Prestations soins
Tests unitaires

Objectif

L'idée de ce TP est de mettre en place les tests unitaires dans une solution Visual Studio 2019.

Contraintes

- ✓ Créer une nouvelle solution à partir de votre solution Soins2021
- ✓ Nom de la solution : *CabinetMedical*
- ✓ *Vous pourrez donc réutiliser le code que vous avez écrit dans la solution Soins2021*
- ✓ *Mettre en place le versionning de votre projet avec dépôt Git local et un dépôt sur github.*

Rappel

Les tests unitaires sont faits pour tester une classe indépendamment du reste de l'application.

Plus précisément ils permettent de tester des méthodes publiques qui retournent une information (donc, de type fonction au sens algorithmique du terme). Le reste du code peut éventuellement être testé suite à des appels dans des méthodes "testables" : par exemple, une méthode privée ou qui ne retourne rien peut tout de même être testée si elle est appelée par une autre méthode qui, elle, est publique et retourne une information.

Présentation

Si on revient sur le TP Soin précédent, on peut se poser la question de savoir à quoi il sert fonctionnellement La réponse est ... à rien

On a implémenté des classes qui devront s'intégrer dans une application fonctionnelle, donc cette partie est importante.

Par contre le programme contenu dans la méthode Main ou dans la classe Traitement n'a strictement aucun intérêt fonctionnel, son seul intérêt est de tester les classes implémentées.

Le but de ce TP est donc de vous faire prendre conscience qu'en programmation, on distingue clairement :

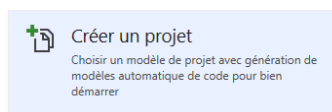
- ✓ La partie fonctionnelle

et

- ✓ La partie tests.

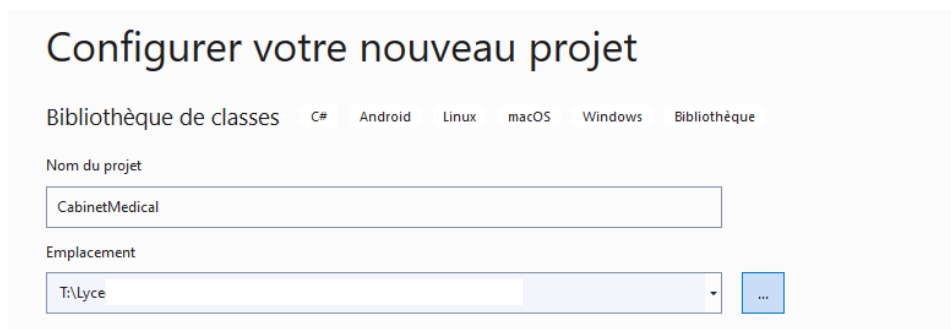
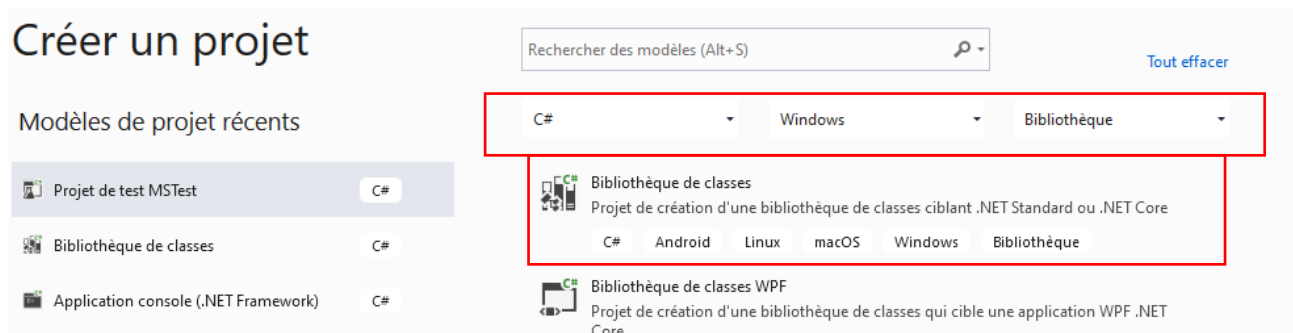
Partie 1 : MISE EN PLACE DU PROJET

Créer un nouveau projet de type bibliothèque de classes .Net Framework ayant comme nom *CabinetMedical*



Mon idée est de vous faire construire une bibliothèque de classes portable de type **classes .Net Framework** et de vous faire créer une bibliothèque de classes

Vous allez donc créer un projet de type bibliothèque :



Informations supplémentaires

Bibliothèque de classes C# Android Linux macOS Windows Bibliothèque

Framework cible ⓘ

.NET Core 3.1 (Prise en charge à long terme)

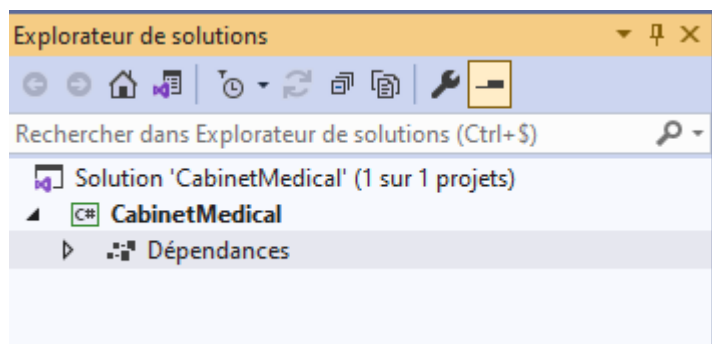


Répondez aux questions suivantes :

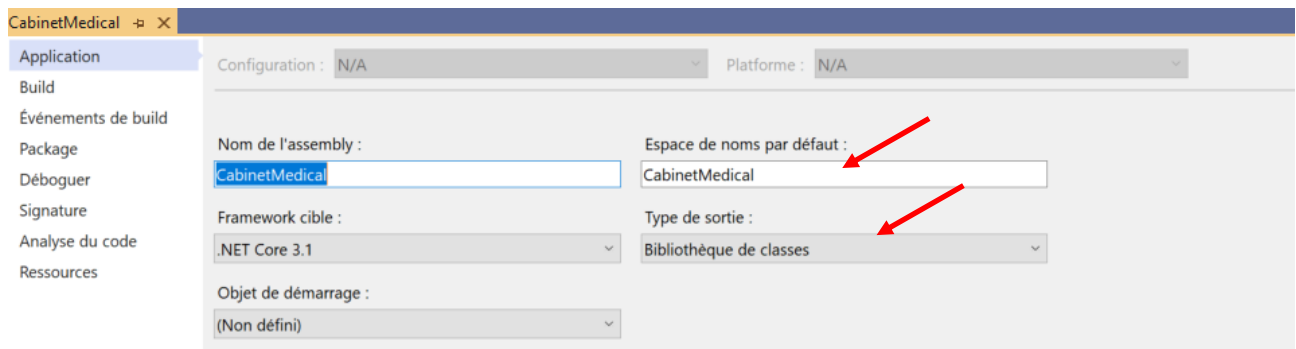
Que signifie .Net Core ?

Que signifie le terme *Prise en charge à long terme* ?

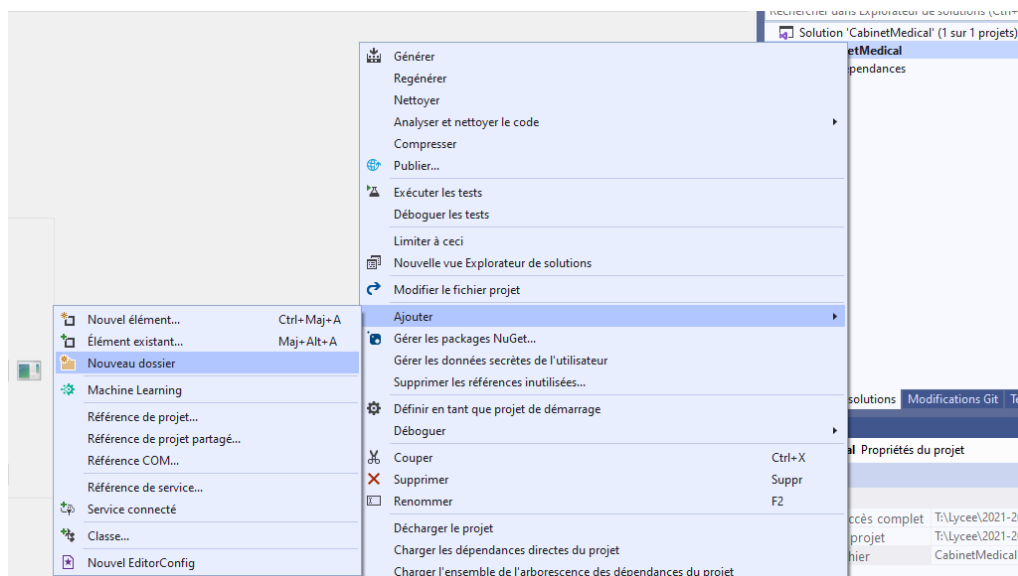
Vous supprimerez la classe Class1



Vérifiez les propriétés de votre projet :



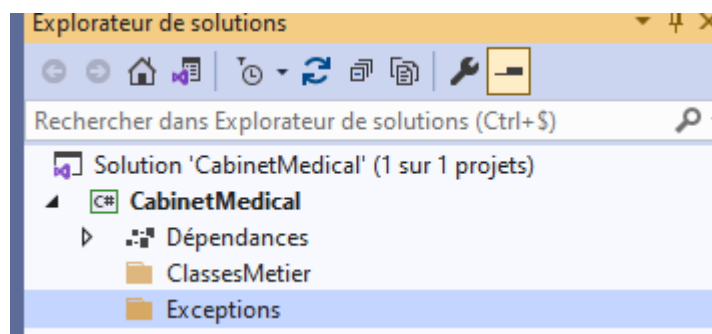
Vous ajouterez un nouveau dossier :



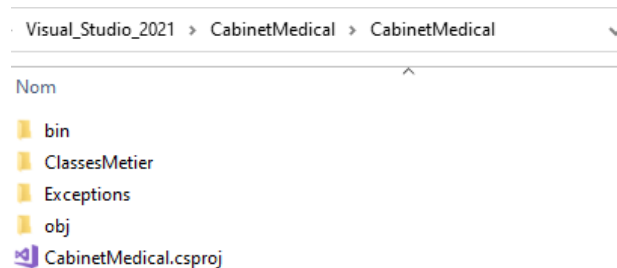
Vous l'appellerez **ClassesMetier**.

Vous ferez pareil pour les exceptions, vous créerez un dossier **Exceptions**.

Au final vous devriez retrouver ceci :



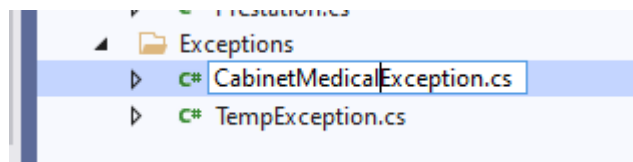
Vérifiez sur le disque :



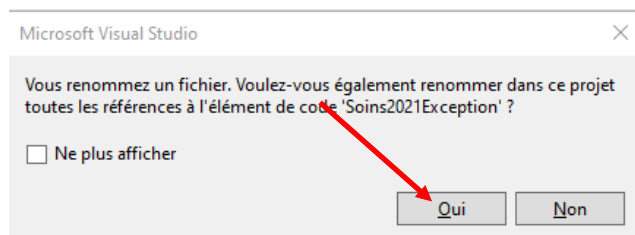
Vous allez recopier les classes Dossier, Intervenant, IntervenantExterne, Prestation dans le dossier ClassesMetier et les classes Soins2021Exception et TempException dans le dossier Exceptions.

Vous aurez à rajouter ces classes dans le projet.

Vous renommerez depuis l'explorateur de solutions la classe Soins2021Exception en CabinetMedicalException :



Vous aurez à renommer la classe et le fichier. Répondez Oui à la question posée.



Vous renommerez l'espace de noms de chaque classes :

- ✓ CabinetMedical.Exceptions pour les exceptions
- ✓ CabinetMedical.ClassesMetier pour les classes métier

```
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.Configuration;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

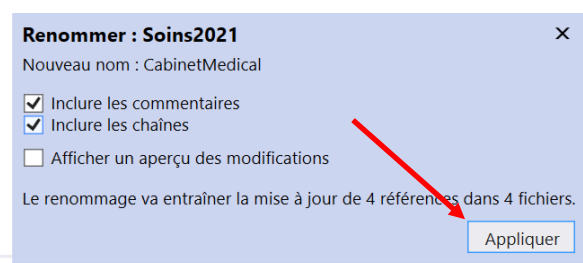
namespace CabinetMedical.Exceptions
{

```

Ou

```
namespace CabinetMedical.ClassesMetier
{
    11 références
    public class Dossier
    {

```



Vous aurez aussi à corriger un certain nombre d'erreurs nouvellement apparues :

Par exemple :

```
var lesAppSettings = new AppSettingsReader();
string pathToLogFile = lesAppSettings.GetValue("pathToLogFile", typeof(string)).ToString();
String contenuJson = File.ReadAllText(pathToLogFile);
List<TempException> tempExceptions = JsonConvert.DeserializeObject<List<TempException>>(contenuJson);
tempExceptions.Add(tempException);
```

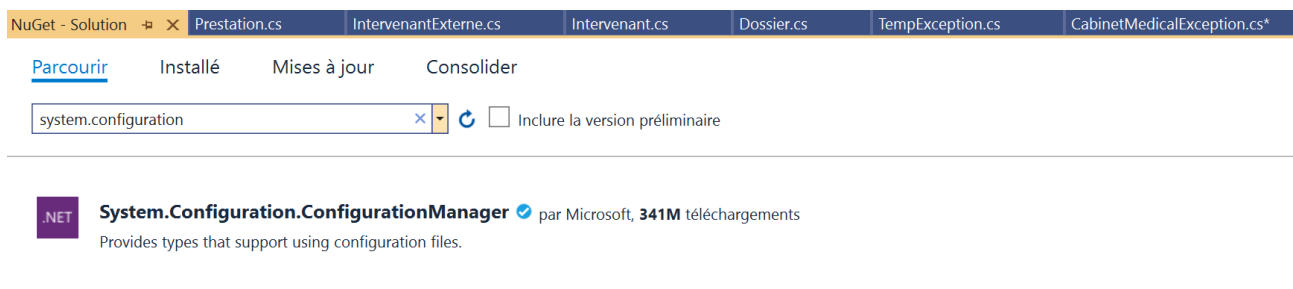
Erreur 1 :

```
var lesAppSettings = new AppSettingsReader();
```

Cette erreur est logique et ennuyeuse.... En effet, vous travaillez avec le framework .net Core 3.1 et ce framework n'intègre pas la bibliothèque *System.Configuration*, donc pas de fichier App.config. C'est normal, si le projet est une bibliothèque de classes .core, elle doit pouvoir être utilisée en cross plateforme (linux, windows,...) et dans des environnement variés (console, winform, webforms)

Donc ici vous aurez 2 solutions :

✓ Solution 1



Importer la bibliothèque *System.Configuration.ConfigurationManager*

Inconvénient : votre bibliothèque n'est plus vraiment .net Core donc plus compatible sur toutes les plateformes

Solution 2

Créer une méthode *GetExceptionJson* qui renvoie un objet de la classe *tempException* sérialisé au format Json indenté (formaté).

Vous laisserez les développeurs utilisant votre bibliothèque de classes *CabinetMedical* libres d'utiliser cette méthode pour tenir un journal de logs.

Erreur 2 :

```
List<TempException> tempExceptions = JsonConvert.DeserializeObject<List<TempException>>(contenuJson);
tempExceptions.Add(tempException);
```

Il vous manque la bibliothèque *Newtonsoft.Json*

Je vous invite toutefois à faire un peu de veille technologique et à aller ici :


<https://www.newtonsoft.com/json> vérifier que la bibliothèque est compatible avec le framework standard ou le framework Core :

Cette bibliothèque supporte bien le framework .Net Standard mais pas .Net Core.....

- Supports .NET Standard 2.0, .NET 2, .NET 3.5, .NET 4, .NET 4.5, Silverlight, Windows Phone and Windows 8 Store

Et de chercher un peu pour trouver ceci : <https://docs.microsoft.com/fr-fr/dotnet/standard/serialization/system-text-json-migrate-from-newtonsoft-how-to?pivots=dotnet-5-0>

Comment migrer de Newtonsoft.Json vers System.Text.Json

25/08/2021 • 25 minutes de lecture • 

Choisir une version .NET

.NET 5

.NET Core 3.1

Version préliminaire de .NET 6

Cet article explique comment migrer de [Newtonsoft.Json](#) vers [System.Text.Json](#).

Vous abandonnerez donc la bibliothèque Newtonsoft.Json pour utiliser la bibliothèque System.Json
Vous

Vous aurez donc à :



Ecrire la méthode `GetExceptionJson` en tenant compte des 2 remarques ci-dessus

Pour vous aider :

<https://docs.microsoft.com/fr-fr/dotnet/api/system.io.directory.getcurrentdirectory?view=netcore-3.1>

<https://docs.microsoft.com/fr-fr/dotnet/standard/serialization/system-text-json-how-to?pivots=dotnet-5-0>




Votre bibliothèque de classes étant destinée à être diffusée, il vous est imposé d'écrire un code propre.

Vous installerez donc le package Nugget StyleCop.Analyzers

Parcourir Installé Mises à jour Consolider


stylecop ☐ Inclure la version préliminaire

 **StyleCop.Analyzers** par Sam Harwell et. al., 50,3M téléchargements 1.1.118

An implementation of StyleCop's rules using Roslyn analyzers and code fixes

Parcourir Installé Mises à jour Consolider

stylecop ☐ Inclure la version préliminaire

 **StyleCop.Analyzers** par Sam Harwell et. al.

An implementation of StyleCop's rules using Roslyn analyzers and code fixes



Et vous corrigerez toutes vos erreurs de saisie ...

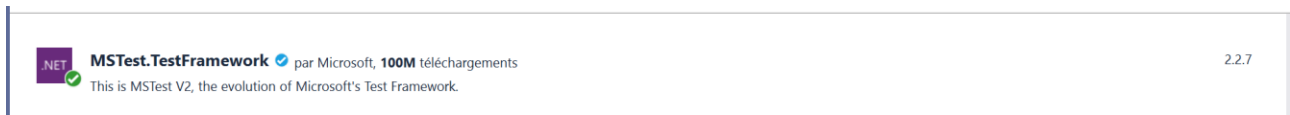
Partie 2 : MISE EN PLACE DES TESTS

L'idée maintenant est de créer un projet de tests qui ne servira QU'A tester vos classes. Il ne sera jamais déployé dans les applications.



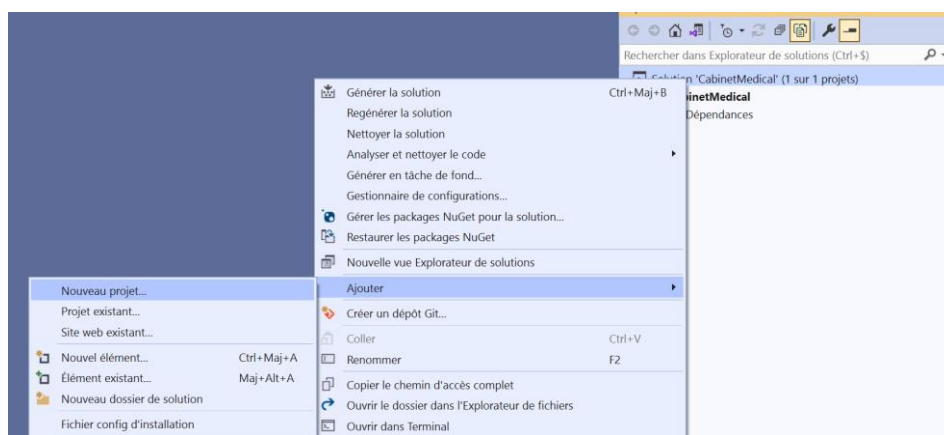
Visual Studio vous facilite grandement la mise en place des tests C'est la plus-value d'une bonne IDE par rapport à un simple éditeur ...

Vous commencerez à installer la bibliothèque MsTest à partir de NuGet :

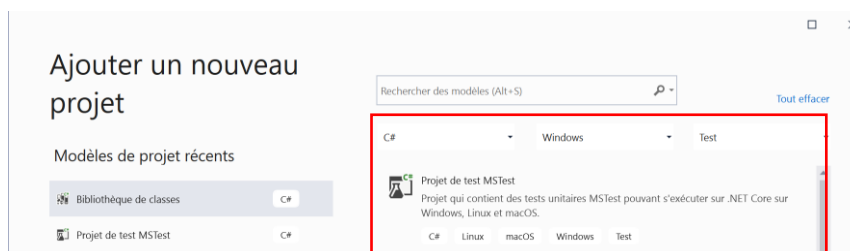


Vous aurez besoin de créer un deuxième projet dans votre solution.

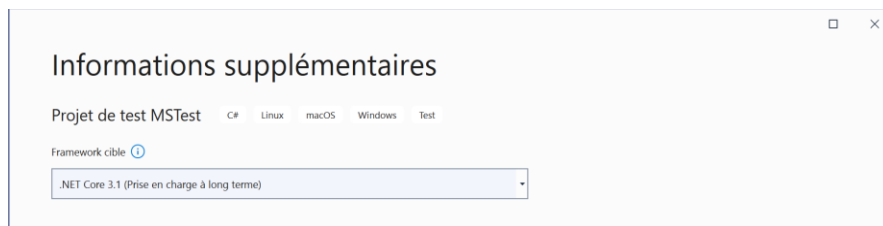
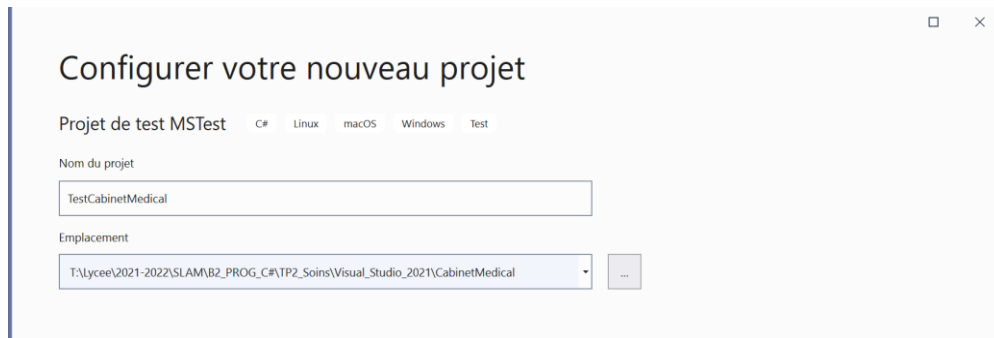
Vous allez donc vous positionner sur la solution dans l'explorateur de solutions et vous ajouter un projet de tests à votre solution.



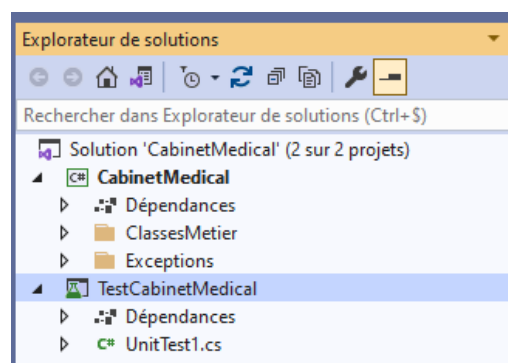
Il s'agira d'un projet MsTests :



Vous l'appellerez TestCabinetMedical pour le framework Net Core 3.1

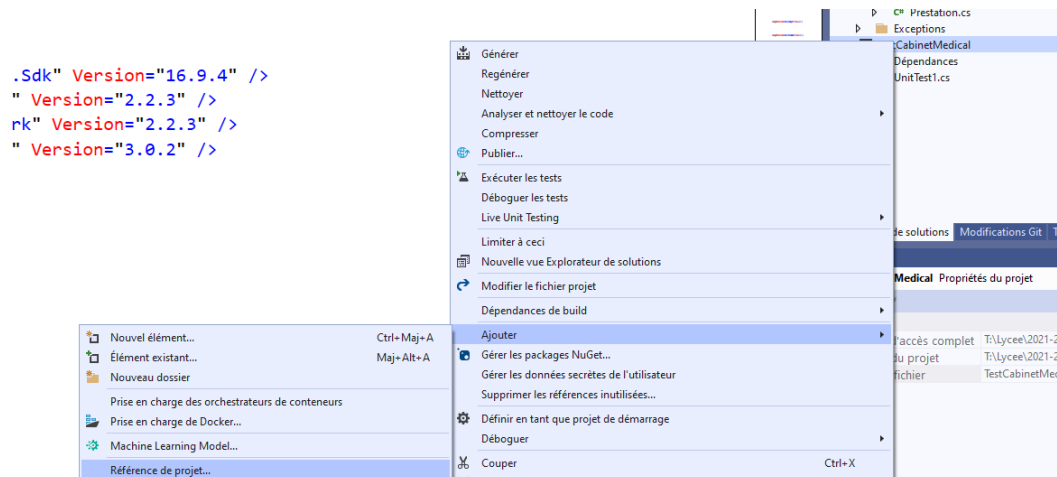


Un deuxième projet figure dans votre explorateur de solutions :



Pour associer le projet de tests au projet de bibliothèque de classes du cabinet médical, vous poserez une référence dans le projet de tests :

Click droit sur le projet de tets/Ajouter/Référence de projet

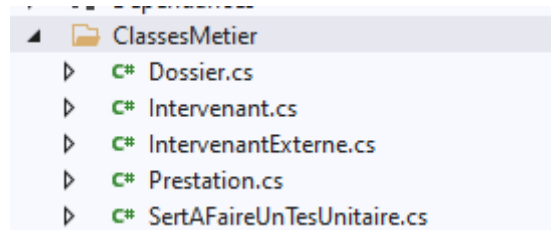


Vous sélectionnez le projet du cabinet médical :



Un petit exemple de test unitaire

Vous allez créer la classe SertAFaireUnTesUnitaire dans le projet CabinetMedical :



Et vous allez dans cette classe (après avoir corrigé les fautes de style), ajouter la méthode SommePourRien

```
internal class SertAFaireUnTesUnitaire
{
    /// <summary>
    /// Effectue la somme de deux nombres entiers.
    /// </summary>
    /// <param name="a">entier 1 à additionner.</param>
    /// <param name="b">entier 2 à additionner.</param>
    /// <returns>la somme des nombres passés en paraètre.</returns>
    0 références
    public int SommePourRien(int a, int b)
    {
        return a + b;
    }
}
```

Rendez-vous dans la classe UnitTest et renommez la méthode TestMethod1 par SommePourRienTest

Vous remarquerez l'annotation de classe : `[TestClass]`

Et l'annotation de méthode : `[TestMethod]`

```

[TestClass]
0 références
public class UnitTest1
{
    [TestMethod]
    0 références
    public void SommePourRienTest()
    {
    }
}

```

Vous allez maintenant implémenter la méthode SommePourRienTest.

L'idée de cette méthode est :

- ✓ D'appeler la méthode SommePourRien (a,b) de la classe SertAFaireUnTesUnitaire
- ✓ De tester si le résultat obtenu est égal au résultat attendu.
- ✓ Et s'il est différent, de déclencher une erreur :



Vous allez utiliser la classe Assert pour effectuer les tests

Allez voir ici :

<https://docs.microsoft.com/fr-fr/dotnet/api/microsoft.visualstudio.testtools.unittesting.assert?view=visualstudiosdk-2019>



Répondez aux questions suivantes :

Quel est le type de la classe ?

Quel est le type des méthodes ?

Vous allez donc rajouter le code suivant à votre méthode de test :

```

public void SommePourRienTest()
{
    int a = 3;
    int b = 5;
    int somme = new SertAFaireUnTesUnitaire().SommePourRien(a,b);
    Assert.AreEqual(8, somme);
}

```



Remarquez le prototype de la méthode AreEquals et l'ordre des paramètres....

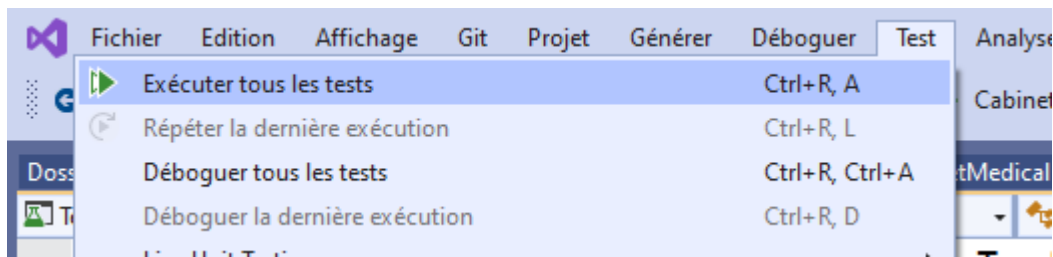
```
Assert.AreEqual(8, somme);
```

void Assert.AreEqual<int>(int expected, int actual) (+ 17 surcharges)

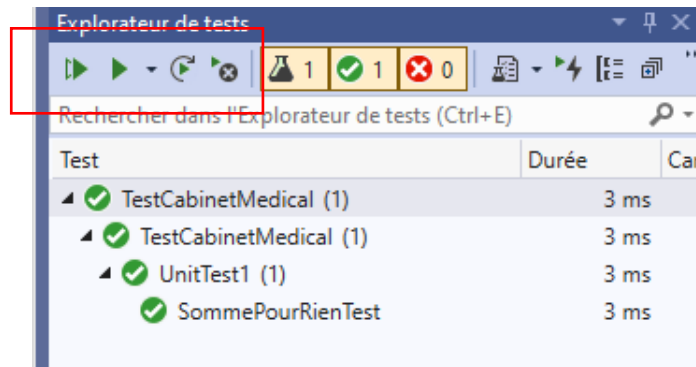
Teste si les valeurs spécifiées sont identiques, et lève une exception si les deux valeurs sont différentes. Les types numériques distincts sont considérés comme différents même si les valeurs logiques sont identiques. 42L n'est pas égal à 42.

Exceptions :
AssertFailedException

Vous allez pouvoir maintenant effectuer le test :



La fenêtre Explorateur de test s'ouvre et donne le résultat des tests :



Si tous les boutons sont verts, c'est que tous les tests ont réussi....



Je vous conseille d'ancrer cette fenêtre pour avoir toujours à l'œil les tests



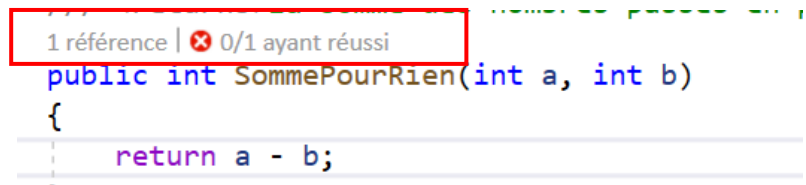
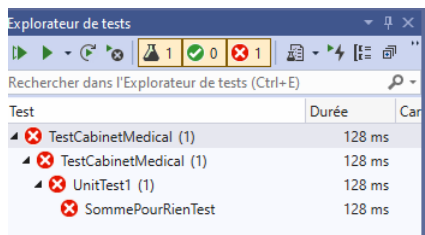
Exercice:

Modifiez la méthode SommePourRien :

```
public int SommePourRien(int a, int b)
{
    return a - b;
}
```

Exécutez à nouveau les tests :





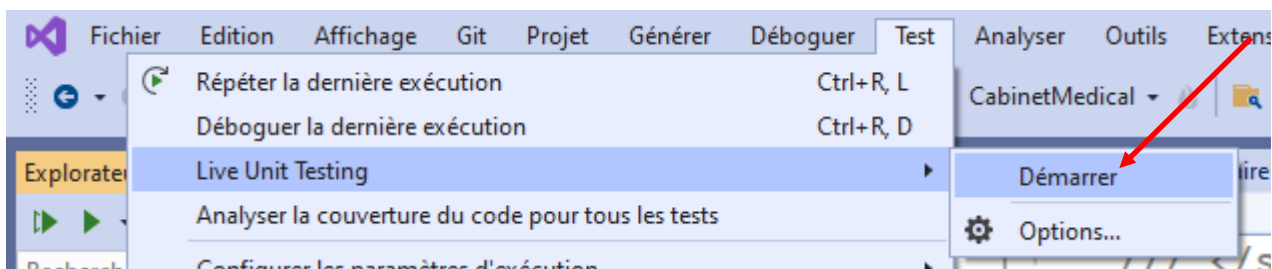
Corrigez et exécutez à nouveau le test.

Partie 3 : LES TESTS EN DIRECT : LIVE UNIT TESTING

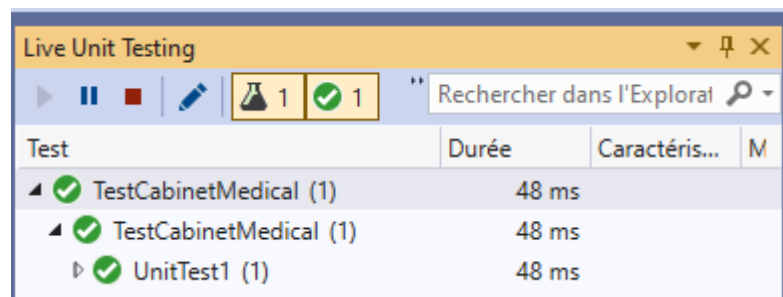
Encore mieux les tests unitaires en direct

Live Unit Testing est une technologie introduite dans Visual Studio 2017. Elle exécute automatiquement les tests unitaires en temps réel lorsque l'on apporte des modifications au code. Cela permet de gagner du temps et d'avoir une vision globale des tests en temps réel.

Il suffit d'aller dans le menu de Visual Studio et d'activer les live tests unit :

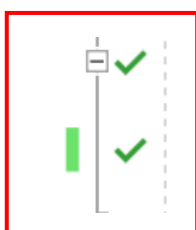


Une nouvelle fenêtre de tests apparaît (vous pouvez fermer la *fenêtre Explorateur de tests*, elle est remplacée par la fenêtre *Live Unit Testing*). Vous pouvez ancrer cette fenêtre ou la laisser flottante.



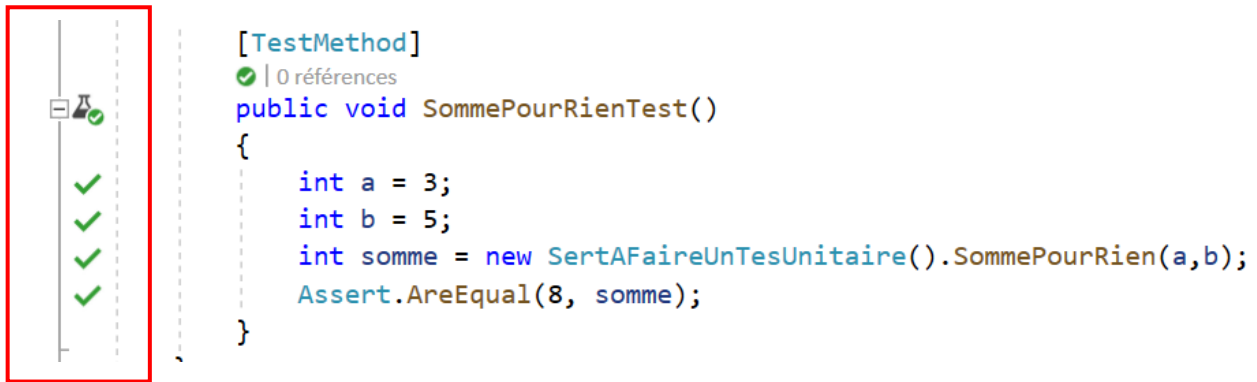
Il y a aussi eu des modifications

- ✓ au niveau de votre fenêtre d'édition de code d'application :



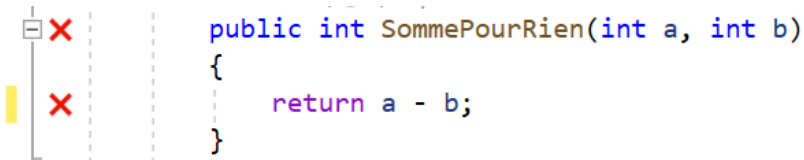
```
public int SommePourRien(int a, int b)
{
    return a + b;
}
```

✓ au niveau de votre fenêtre d'édition de test :

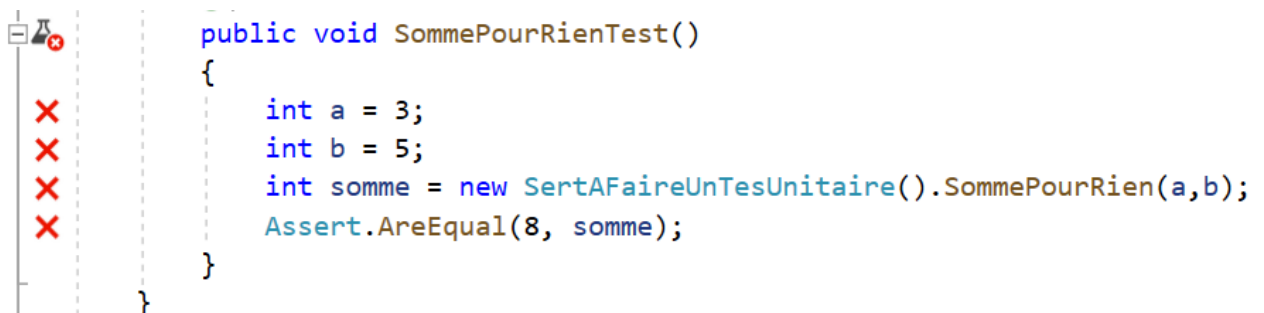


Modifiez la méthode SommePourRien :

Le test unitaire se fait automatiquement et vous ne pouvez pas ignorer qu'il a échoué :

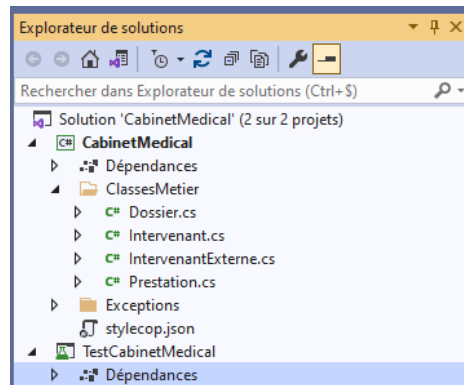


Live Unit Testing				
Test	Durée	C..	Message d'erreur	
✗ TestCabinetMedical (1)	75 ms			
✗ TestCabinetMedical (...)	75 ms			
✗ UnitTest1 (1)	75 ms			
✗ SommePourRien...	75 ms		Assert.AreEqual failed....	



Corrigez la méthode SommePourRien.... Les tests se remettent au vert !

Vous pouvez maintenant supprimer les classes SertAFaireUnTesUnitaire et UnitTest1, elles ne vous serviront plus pour la suite du projet.



Partie 4 : MISE EN PLACE DES TESTS

Vous allez maintenant mettre en place les vrais tests unitaires pour votre application CabinetMedical.

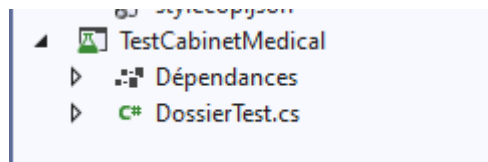
Vous aurez le choix :

- ✓ D'écrire vous-même vos classes de tests en les organisant comme vous le souhaitez
- ✓ De laisser Visual Studio générer les classes de tests classe par classe, puis les modifier pour couvrir tous les tests voulus. Il suffit de faire un click droit dans la fenêtre d'édition de la classe à tester



Vous allez écrire entièrement les classes de tests sur les classes métier sur lesquelles il y a des tests à écrire.

Dans le projet TestCabinetMedical, vous rajouterez la classe DossierTest :



Et vous rajouterez

Le using qui va bien :

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
```

les annotations de tests.

```
[TestClass]
0 références
class DossierTest {

    [TestMethod]
    0 références
    public void TesteGetNbPrestationsI() {

    }
}
```



N'oubliez pas de déclarer la classe en **public** sinon VS ne la verra pas et n'effectuera pas les tests

```
public class DossierTest
```

Il ne vous reste plus qu'à écrire la méthode en vous aidant de ce que vous avez fait dans le premier TP :

```
[TestMethod]
0 références
public void TestGetNbPrestationsI() {
    Intervenant intervenant = new Intervenant("Dupont", "Pierre");
    intervenant.AjoutePrestation(new Prestation("Presta 10", new DateTime(2019, 6, 12), intervenant));
    intervenant.AjoutePrestation(new Prestation("Presta 11", new DateTime(2019, 6, 15), intervenant));
    Assert.AreEqual(2, intervenant.GetNbPrestations());
}
```



Travail à faire :

Implémenter la méthode **TesteGetNbPrestationsIE** . Vérifiez le test

Vérifiez la bonne exécution avec une mauvaise valeur

TestCabinetMedical (2)	4 ms
TestCabinetMedical (...)	4 ms
DossierTest (2)	4 ms
TesteGetNbPrest...	< 1 ms
TestGetNbPresta...	4 ms

Test	Durée	C
TestCabinetMedical (2)	13 ms	✗
TestCabinetMedical (2)	13 ms	✗
DossierTest (2)	13 ms	✗
TesteGetNbPrestationsIE	9 ms	✗
TestGetNbPrestationsI	4 ms	✓

Nous allons maintenant gérer la date de création d'un dossier.

Nous allons faire 2 tests :

- ✓ **TestDateCreationDossierOK** : la date de création d'un dossier est inférieure ou égale à la date du jour, le dossier est créé.
- ✓ **TestDateCreationDossierKO** : la date de création d'un dossier est supérieure à la date du jour, une exception de type `CabinetMedicalException` est levée.

TestDateCreationDossierOK nous allons utiliser la méthode `IsInstanceOfType` de la classe `Assert` pour vérifier que si la date est bonne, l'objet créé est bien de la classe `Dossier` :

```
public void TestDateCreationDossierOK()
{
    Dossier dossier = new Dossier("Dupont", "Jean", new DateTime(1990, 11, 12), new DateTime(2019, 3, 31));
    Assert.IsInstanceOfType(dossier, typeof(Dossier));
}
```

Rechercher dans l'Explorateur de tests (Ctrl+E)			
Test	Durée	C..	M
TestCabinetMedical (3)	4 ms		
TestCabinetMedical (3)	4 ms		
DossierTest (3)	4 ms		
TestDateCreationDossierOK	< 1 ms		
TesteGetNbPrestationsIE	< 1 ms		
TestGetNbPrestationI	4 ms		

Essayez avec `typeof(Prestation)` : le test passe en rouge



```
Assert.IsInstanceOfType(dossier, typeof(Prestation));
```

TestDateCreationDossierKO Dans ce cas, la date de création d'un dossier testée sera supérieure à la date du jour, une exception de type `CabinetMedicalException` doit être levée. Vous allez annoncer en annotation que le méthode doit lever une exception de type `CabinetMedicalException`

```
[TestMethod]
```

```
[ExpectedException(typeof(CabinetMedicalException))]
```

Rappelez-vous le cours Il faut que le test marche quel que soit le moment où il est exécuté. Je vais vous proposer une astuce ...



1 – consistance

Quoi ? Ils doivent toujours ramener la même valeur à chaque exécution

Pourquoi ? Ce qui est vrai à un instant t peut être faux à un autre instant ... ou inversement

```
DateTime dateDuJour = DateTime.Now; // ce qui est vrai aujourd'hui peut être faux demain
int nb = new Random().Next(); // les random peuvent fournir des résultats de tests différents
String contenuJson = File.ReadAllText(@"d:\appli\tests\logfile.json"); // problème si fichier déplacé
```

```
[TestMethod]
```

```
[ExpectedException(typeof(CabinetMedicalException))]
```

```
public void TestDateCreationDossierKO()
```

```
{
```

```
    // quel que soit le moment où on exécute le test, la date de création du dossier
```

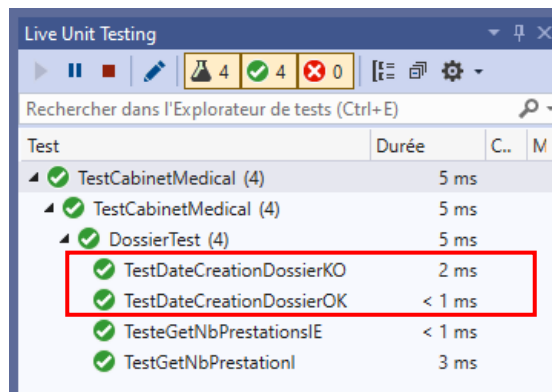
```
    // sera supérieure à la date du jour.
```

```
    DateTime dateCreationDossier = DateTime.Now.AddDays(10);
```

```
    Dossier dossier = new Dossier("Dupont", "Jean", dateCreationDossier, new DateTime(2019, 3, 31));
```

```
}
```

Le test va être immédiatement effectué et il sera validé car l'instanciation de l'objet de la classe `Dossier` lève une exception de type `CabinetMedicalException`



Test	Durée	C..	M
TestCabinetMedical (4)	5 ms		
TestCabinetMedical (4)	5 ms		
DossierTest (4)	5 ms		
TestDateCreationDossierKO	2 ms		
TestDateCreationDossierOK	< 1 ms		
TesteGetNbPrestationsIE	< 1 ms		
TesteGetNbPrestationI	3 ms		

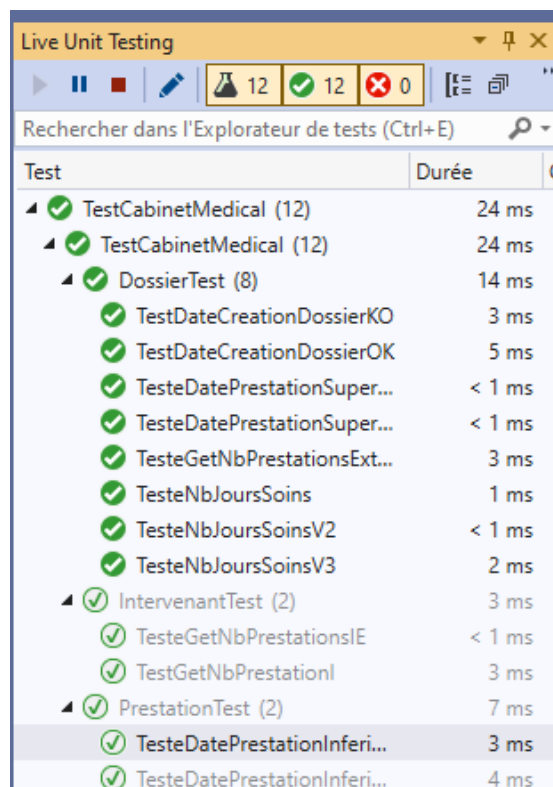


Travail à faire :

Implémenter les autres tests vus dans le TP précédent

Vous pouvez en créer d'autres si vous pensez qu'il en manque

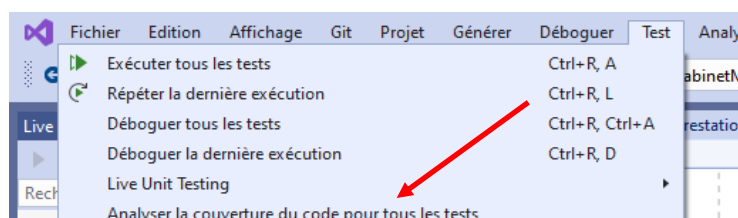
A minima, vous devriez obtenir ceci (après avoir créé une classe de test pour chaque classe métier) :



Test	Durée
TestCabinetMedical (12)	24 ms
TestCabinetMedical (12)	24 ms
DossierTest (8)	14 ms
TestDateCreationDossierKO	3 ms
TestDateCreationDossierOK	5 ms
TesteDatePrestationSuper...	< 1 ms
TesteDatePrestationSuper...	< 1 ms
TesteGetNbPrestationsExt...	3 ms
TesteNbJoursSoins	1 ms
TesteNbJoursSoinsV2	< 1 ms
TesteNbJoursSoinsV3	2 ms
IntervenantTest (2)	3 ms
TesteGetNbPrestationsIE	< 1 ms
TesteGetNbPrestationI	3 ms
PrestationTest (2)	7 ms
TesteDatePrestationInferi...	3 ms
TesteDatePrestationInferi...	4 ms



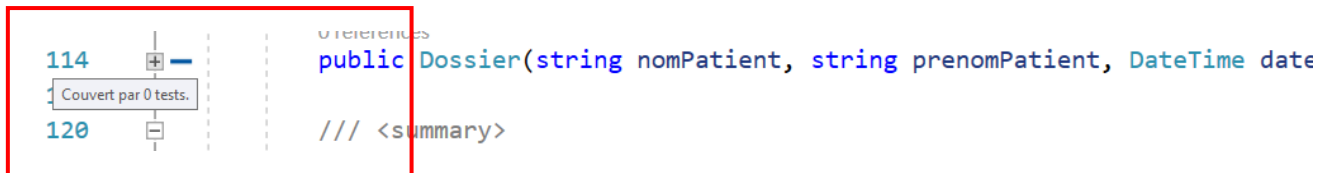
Allez voir dans tests/analyser la couverture de tests :



Résultats de la couverture du code				
benoi_PEGASE 2021-10-01 15_54_53.coverag				
Hiérarchie	Non couverts (blocs)	Non couverts (% blocs)	Couverts (blocs)	Couverts (% blocs)
benoi_PEGASE 2021-10-01 15_54_53.coverage	50	12,41 %	353	87,59 %

Vous verrez qu'il y a encore un peu de travail A vous de terminer le travail
Notamment sur les constructeurs surchargés qui ne sont pas couverts par les tests :

Exemple :



Partie 5 : VERSIONNER

Si votre solution n'est pas versionnée, il est temps de la faire !!!

Travail à faire :

Maintenant que vous avez eu un aperçu des tests unitaires, vous allez compléter votre travail :

- ✓ Supprimer les tests de méthodes inutiles : getters, setters, méthodes qui ne retournent rien, ...
- ✓ Créer les tests pour les méthodes de la classe dossier :
 - ✓ getNbPrestationsExternes()
 - ✓ getNbJoursSoins()

Vous pourrez créer une méthode privée dans la classe DossierTests

```
private Dossier InitialiseDossier()
{
    Dossier unDossier = new Dossier("Robert", "Jean", new DateTime(1980, 12, 3));
    unDossier.AjoutePrestation("Libelle P3", new DateTime(2015, 9, 10, 12, 0, 0), new Intervenant("Dupont", "Jean"));
    unDossier.AjoutePrestation("Libelle P1", new DateTime(2015, 9, 1, 12, 0, 0), new IntervenantExterne("Durand", "Annie", "Cardiologue", "Marseille", "0202020202"));
    unDossier.AjoutePrestation("Libelle P2", new DateTime(2015, 9, 8, 12, 0, 0), new IntervenantExterne("Sainz", "Olivier", "Radiologue", "Toulon", "0303030303"));
    unDossier.AjoutePrestation("Libelle P4", new DateTime(2015, 9, 20, 12, 0, 0), new Intervenant("Maurin", "Joëlle"));
    unDossier.AjoutePrestation("Libelle P6", new DateTime(2015, 9, 8, 9, 0, 0), new Intervenant("Blanchard", "Michel"));
    unDossier.AjoutePrestation("Libelle P5", new DateTime(2015, 9, 10, 6, 0, 0), new Intervenant("Tournier", "Hélène"));

    return unDossier;
}
```



Exercice:

- ✓ Créez les tests pour la méthode de la classe Prestation :
- ✓ CompareTo() : il y a au moins 3 tests à écrire

Dès que vos Tests sont tous au vert, vous pourrez générer votre DLL... dans le TP suivant

Désormais vous pourrez travailler en bâtissant vos tests **AVANT** d'écrire vos méthodes
C'est du TDD : **T**est **D**riven **D**evelopment

Bon courage

