R3.04 Qualité de développement
TD6 Partie 1 – Patron de conception Singleton, Interface, Tests de substitution

Vous devrez terminer le TD5 avant de réaliser celui-ci.

1. ***Patron de conception « Singleton ».***

Reprendre votre code de l'application « Calculatrice » contenant les tests unitaires (TD5).

Transformation d'une classe statique en une classe instanciable

L'utilisation de classes statiques est une mauvaise manière de programmer, car cela ne respecte pas le paradigme objet.
Modifier le code de la classe `Calcul` de façon à remplacer la classe statique par une classe instanciable (supprimer le `static` dans tous le code). Modifier également l'application WPF (créer une property de type `Calcul` et l'instancier dans le constructeur de l'application WPF).
Modifier les tests unitaires :

- Créer une property de type `Calcul`
- La property pourra être instanciée dans le constructeur de la classe de test (utiliser le snipet `ctor`)

Vérifier que l'application fonctionne.
Vérifier que les tests unitaires fonctionnent.

Application du patron « Singleton »

En outre, quand on ne veut créer qu'une seule instance d'un objet, la bonne pratique est d'appliquer le patron de conception « Singleton ».
Code du patron de conception « Singleton » :
https://fr.wikipedia.org/wiki/Singleton_(patron_de_conception)
https://refactoring.guru/fr/design-patterns/singleton/csharp/example#example-1

C# [modifier | modifier le code]

```
public class Singleton
{
    private static Singleton _instance;
    static readonly object instanceLock = new object();

    private Singleton()
    {

    }

    public static Singleton Instance
    {
        get
        {
            if (_instance == null) //Les locks prennent du temps, il est préférable de vérifier d'abord la
nullité de l'instance.
            {
                lock (instanceLock)
                {
                if (_instance == null) //on vérifie encore, au cas où l'instance aurait été créée entretemps.
                   _instance = new Singleton();
                }
            }
            return _instance;
        }
    }
}
```

On remarque que le constructeur de la classe est privé, ce qui empêche de l'utiliser dans la classe appelante. La seule façon d'instancier un objet est donc d'utiliser la méthode publique `Instance`.

Appliquer le patron « Singleton ». Dans votre cas, votre classe se nommera `Calcul` et non `Singleton` (le code à ajouter est donc à insérer dans la classe `Calcul`).

Modifier le code de l'application WPF :

- Créer la property suivante dans l'application WPF :

```
public Calcul ObjCalcul
    {
      get { return Calcul.Instance; }
    }
```

  Le principe est le suivant : la méthode `Calcul.Instance` vérifie si une instance de la classe `Calcul` est déjà créée, sinon elle la créé.
- Chaque méthode (`Addition`, etc.) sera à préfixer par `ObjCalcul` qui contiendra l'instance unique de la classe `Calcul`.
- Supprimer l'instanciation `Calcul … = new Calcul();`

Modifier de la même façon les tests unitaires.

Bonne pratique : remplacer toutes les classes statiques par des classes instanciables intégrant un singleton. Le singleton permettra, par exemple, de limiter les instances de web services ou d'accès aux bases de données, afin de limiter les ressources utilisées.
Exemple : quand vous utilisez un Web Service (Bing Maps, Google Maps ou autre), la tarification se fait souvent à l'instanciation du service et/ou aux appels de méthodes. Avec un singleton, il n'y aura qu'une seule instanciation.

Remarque : il est préférable d'utiliser le Singleton thread-safe (avec lock), plutôt que le Singleton naïf. *https://refactoring.guru/fr/design-patterns/singleton/csharp/example#example-1*

2. Interface

Une interface permet :
- Un code découplé de son implémentation, respectant les principes **SOLID** (nous y reviendrons en partie 2).
- A tout architecte d'imposer des contrats et donc des méthodes à être présentes dans des classes.

Ajouter l'interface `ICalcul` au projet `Calculatrice`. La classe `Calcul` devra implémenter cette interface.
Rappels création d'une interface :
- *https://learn.microsoft.com/fr-fr/dotnet/csharp/language-reference/keywords/interface*
- *Seuls les membres publics doivent être définis dans une interface.*

Le code de la property dans l'application WPF et dans la classe de tests unitaires devient :

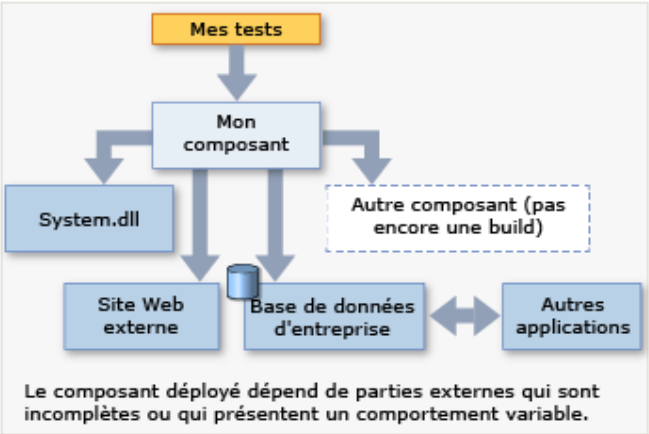
```
public ICalcul ObjCalcul
    {
      get { return Calcul.Instance; }
    }
```

3. Tests de substitution (mocks)

On développe des tests unitaires pour vérifier l'utilisation de nos classes. Ensuite, ces classes vont être « insérées » dans un système qui va à la fois les consommer mais qui va également les chaîner à d'autres. Nos classes devront donc communiquer avec leurs voisines. Mais peut-être que les classes voisines ne seront pas développées en même temps ! Alors comment faire pour valider notre travail d'intégration ?

La solution n'est pas de faire le développement à la place de vos collègues mais plutôt de simuler le comportement de leurs objets dans des *stubs* ou des *shims* qui sont des objets contrôlés par nos tests.

Un *stub* remplace une classe « voisine » en implémentant son interface. L'utilisation de *stubs* n'est donc possible que si l'application est architecturée autour d'interfaces (d'où l'intérêt de créer des interfaces). Un *shim* modifie votre objet au moment de son utilisation pour que ses appels à une classe « voisine » soient redirigés vers du code de test. Le *shim* s'utilise donc dans le cas où les objets à appeler n'ont pas d'interface ou ne sont pas des composants de votre solution (notamment les composants .NET fournis par le framework).



Nous verrons uniquement les tests de substitution utilisant des *stub*s, car nous nous appuierons sur les interfaces créées. Pour cela, nous allons ajouter une méthode `Moyenne` dans une nouvelle classe. Le calcul de la moyenne utilisera les méthodes `Addition` et `Division` de la classe `Calcul`. Le test de la méthode `Moyenne` se fera en utilisant un stub.

Ajout d'une nouvelle classe

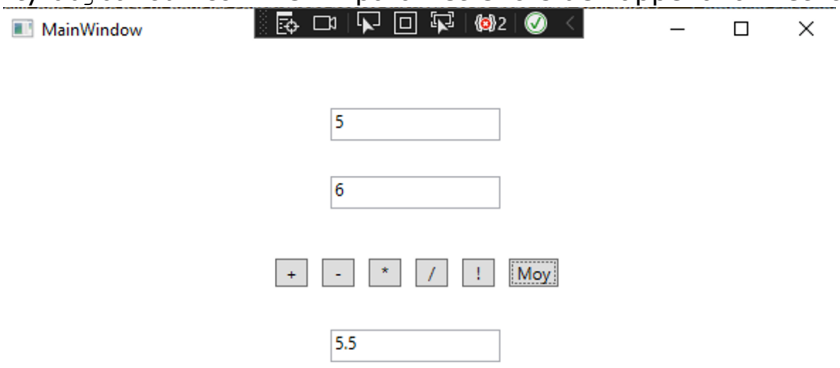
Créer la classe `CalculAvance` dans la bibliothèque de classes `Calculatrice`. Ajouter la méthode `Moyenne`. Elle appelle les méthodes `Division` et `Addition` de la classe `Calcul` sur un objet `calcul` de type `ICalcul` passé en paramètre.

```
public double Moyenne(ICalcul calcul, double nb1, double nb2)
{
  return calcul.Division(calcul.Addition(nb1, nb2), 2);
}
```

Créer une interface. Appliquer le patron Singleton.

Modifier votre application WPF pour gérer le calcul de la moyenne :

- Créer une property de type `ICalculAvance` (code similaire à la property `ObjCalcul`. Cf. page 2).
- Passer la property `ObjCalcul` comme 1er paramètre lors de l'appel à la méthode `Moyenne`.



Test unitaire sans stub

Coder le test unitaire de la méthode `Moyenne` avec comme valeur nb1=5 et nb2=6. Vous devrez créer un objet de type `ICalcul`.

Ajouter une 2nde instanciation d'un objet de type `ICalcul` dans la méthode de tests pour tester le singleton. Mettre un point d'arrêt et lancer le débogage des tests. Vous verrez que lors de la 2nde instanciation, l'objet de type `Calcul` sera retourné.

```
19          [TestMethod()]
20          public void MoyenneTest()
21          {
22              //Arrange
23              Double a = 5.0;
24              Double b = 6.0;
25              ICalcul calcul = Calcul.Instance;
26              ICalcul calcul2 = Calcul.Instance;
27              //Act
28              Double resultat = ObjCalculAvance.Moyenne(calcul, a, b);
29              //Assert
30              Assert.AreEqual(5.5, resultat, "Test non OK. La valeur doit être égale à 5.5.");
31          }
```

```
5   public class Calcul:ICalcul
6   {
7       private static Calcul? instance;
8       static readonly object instanceLock = new object();
9
10      private Calcul(){}
11
12      public static Calcul Instance
13      {
14          get
15          {
16              if (instance == null) //Les locks prennent du temps, il est préférable de vérifier d'abord la nullité de l'instance.
17              {
18                  lock (instanceLock)
19                  {
20                      if (instance == null) //on vérifie encore, au cas où l'instance aurait été créée entretemps.
21                          instance = new Calcul();
22                  }
23              }
24              return instance; ≤ 1 ms écoulées
25          }
```

Vous verrez également que le calcul de la moyenne utilise bien la classe `Calcul` :

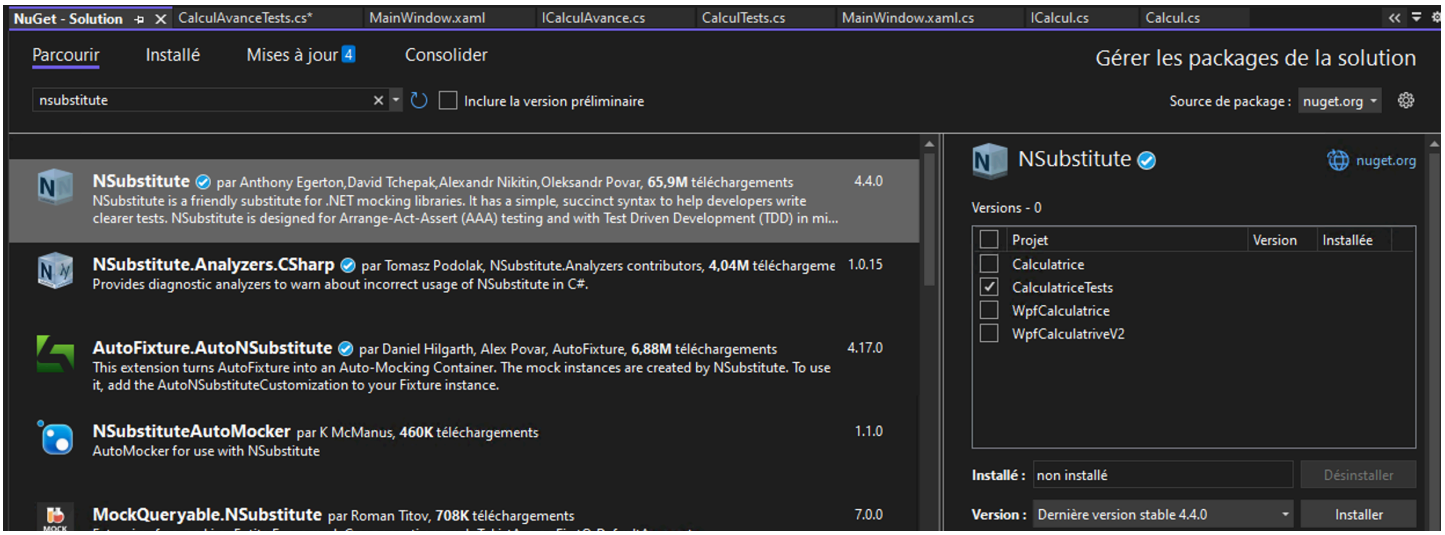
```
3   namespace Calculatrice
4   {
5       public class Calcul:ICalcul
6       {
7           private static Calcul? instance;
8           static readonly object instanceLock = new object();
9
10          private Calcul(){}
11
12          public static Calcul Instance
13          {
14              get
15              {
16                  if (instance == null) //Les locks prennent du temps, il est préférable de vérifier d'abord la nullité de l'instance.
17                  {
18                      lock (instanceLock)
19                      {
20                          if (instance == null) //on vérifie encore, au cas où l'instance aurait été créée entretemps.
21                              instance = new Calcul();
22                      }
23                  }
24                  return instance;
25              }
26          }
27
28          public Double Addition(Double a, Double b)
29          {
30              return a + b; ≤ 1 ms écoulées
31          }
```

Supprimer la 2nde instanciation dans la méthode de tests.

Test unitaire avec stub

Nous allons considérer que la classe `Calcul` n'a pas encore été développée. Pourtant elle est utilisée par la méthode `Moyenne`. Il faut donc utiliser un test de substitution.

Installer le package Nuget *NSubstitute* (Menu *Outils > Gestionnaire de package Nuget > Gérer les packages NuGet pour la solution*) :



La classe `Calcul` respecte les principes de modularité et expose une interface parfaitement définie. C'est ici que la librairie *NSubstitute* va nous aider en créant un objet de test supportant cette interface et pour lequel nous allons fixer « en dur » le retour des méthodes que `CalculAvance` va utiliser.

```
[TestMethod]
public void MoyenneTestAvecStub()
{
    //Arrange
    Double a = 5.0;
    Double b = 6.0;

    var calculStub = Substitute.For<ICalcul>(); // Définition du stub sur l'interface

    // Implémentation de deux méthodes qui vont être utilisées par le calcul de la moyenne
    calculStub.Addition(a, b).Returns(11);
    calculStub.Division(11, 2).Returns(5.5);

    //Act
    Double resultat = ObjCalculAvance.Moyenne(calculStub, a, b); //Utilisation du stub
    //Assert
    Assert.AreEqual(5.5, resultat, "Test non OK. La valeur doit être égale à 5.5.");
}
```

Mettre un point d'arrêt dans la méthode de test. Lancer le débogage des tests. Vous verrez que le test ne passera pas par la méthode `Addition` (passer de ligne en ligne).

L'utilisation de *NSubstitute* est intuitive. On commence par créer un objet à partir de l'interface cible. Ensuite, on définit les « entrées-sorties » pour les deux méthodes de l'interface qui seront utilisées.
Cette souplesse confirme que l'utilisation d'interfaces pour faire communiquer les modules de l'application est définitivement la bonne solution pour architecturer les applications.

4. ***Travail à faire***

Reprendre votre code de l'application « Comptes bancaires » contenant les tests unitaires (TD5).

- Appliquer le patron Singleton à la classe `DataAccess`. Le service d'accès aux données sera ainsi instancié une seule fois (afin de limiter les accès concurrents à la BD, sources de dégradation des performances) et la même instance sera utilisée par tous les composants de l'application qui en auront besoin. Le service est créé pour le premier composant qui en fait la demande, et utilisé pour le reste.
- Créer des interfaces pour les classes `DataAccess` et `ServiceCompte`.