TP C#4 : UnitTest

Consignes de rendu

À la fin de ce TP, vous devrez rendre une archive respectant l'architecture suivante :

```
|-- csharp-tp4-{login}/
    |-- AUTHORS
    |-- README
    |-- TP4.sln
    |-- TP4/
        |-- Assert.cs
        |-- AssertArrayEqual.cs
        |-- AssertArrayNotEqual.cs
        |-- AssertEqual.cs
        |-- AssertNotEqual.cs
        |-- ConsoleMain.cs
        |-- Misc.cs
        |-- Test.cs
        |-- Everything except bin/ and obj/
```

N'oubliez pas de vérifier les points suivants avant de rendre :

— Le fichier `README` est obligatoire.
— Pas de dossiers `bin` ou `obj` dans le projet.
— Respectez scrupuleusement les prototypes demandés.
— **Le code doit compiler !**

README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les bonus que vous aurez implémentés. Un `README` vide sera considéré comme une archive invalide (malus). Le README est le meilleurs moyens d'expliquer pourquoi et comment vous avez implémenté vos bonus.

1 Introduction

Depuis le début de l'année vous avez programmé sans vraiment tester sérieusement votre code. Ce Tp va vous montrer l'importance d'une telle pratique. Pendant ce Tp vous allez étudier la notion de généricité, d'interface et d'héritage.

2 Cours

2.1 Les tests unitaires

Un test unitaire est un test qui a pour but de tester un bout élémentaire de votre code. Par exemple si vous codez une fonction qui affiche un labyrinthe en 3d, les tests unitaires d'un tel programme seraient de tester individuellement chaque étape critique (affichage des murs, parsing du fichier de sauvegarde, teste d'affichage des vertex).

En effet si vous testez méticuleusement votre code vous serez obligés de factoriser votre code en suite de fonctions élémentaires car elles seront plus faciles à tester. Or, partitionner son code en fonctions élémentaires est une manière saine d'écrire du code (le code est plus compréhensible, plus facile à débuguer et enfin plus modulaire).

Cependant tester son code sans l'aide d'un framework de test peut être particulièrement ardu. C'est pour cela que dans ce tp, nous allons écrire un "framework" basique de test unitaire. L'idée est que vous compreniez mieux la logique de ces frameworks et que vous puissiez en même temps vous habituer à ces concepts.

2.2 Objet

2.2.1 Rappels sur l'héritage

Vous utilisez la programmation objet depuis quelques mois maintenant, vous devez avoir compris que le but de la POO est de permettre une abstraction et une factorisation avancée du code.

C'est là où l'héritage prend toute son importance. Imaginez que vous avez le squelette d'une classe qui représente un humain et que vous devez implémenter une class élève. De toute évidence la plupart des méthodes et variables de la classe seront inchangées. Cependant vous aurez peut-être besoin de changer la méthode `Sleep` ou `Work`. En effet, un élève ne peut pas dormir pendant les cours ! Mais comment faire.

Nous avons la classe `parente`, dans notre cas humain.

```
1  public class Humain
```

La classe `élève` hérite de la classe `humaine`.

```
1  public class Élève: Humain
```

Donc si nous utilisons la méthode `Run` ou `Talk`, d'`élève` en réalité nous appelons la méthode du père. Si nous voulons réécrire le comportement de ces méthodes nous avons besoin de 2 choses. La première est que la méthode du père doit être virtuelle. Et la deuxième est que nous utilisons dans le fils le mot-clef **override**.

```
1  public void virtual Sleep(int time); // Humain
```

```
1  public void override Sleep(int time); // Élève
```

Il reste encore 2 subtilités nécessaires à comprendre sur l'héritage. Si vous voulez garder une méthode privée mais que vous avez besoin que le fils de l'objet puisse y accéder remplacez le `private` par `protected`. Enfin la syntaxe pour initialiser l'objet qui hérite de son père est :

```
1 public Élève(string name) : base(name)
2 {
3 }
```

Pour aller plus loin

`https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/inheritance`

2.2.2 Généricité

La généricité est le fait d'abstraire son code à la notion de type. Plus simplement vous vous utilisez une classe générique lorsque vous allez stocker des types indéterminés mais que cela ne gêne par votre implémentation. La généricité est un bon moyen de factoriser votre code.

```
1 public static bool SameType<A, B>(A a, B b)
2 {
3     return a.GetType() == b.GetType();
4 }
```

Les chevrons permettant de définir un type A et un type B, on sait donc que 'a' est du type A et que 'b' est du type B. Cependant dans la fonction on teste si les 2 variables sont de même type. On peut aussi définir des classes génériques.

Pour aller plus loin

`https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-classes`

2.2.3 Interface

Les interfaces sont utilisées pour permettre à un programme d'utiliser indifféremment plusieurs classes. Par exemple : si on a 2 classes génériques qui ont donc possiblement des types différents mais que nous utilisons dans les 2 classes la méthode `Run`. Alors il suffit de définir une interface pour cette méthode. On pourra donc faire une liste d'objets possédant cette interface. Bien entendue nous pourrons utiliser seulement les méthodes définies dans l'interface.

Pour aller plus loin

`https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/`

3 Exercices

3.1 UnitTest

Dans cette section nous allons implémenter un **framework** rudimentaire de test unitaire.

3.1.1 Assert

Tout le code de cette section doit être dans `Assert.cs`. L'interface suivante est défini pour vous. En effet cette classe est la base de tous les **asserts** de notre framework.

```
1  public interface IAssert
2  {
3      void Run();
4  }
```

Le squelette de la classe est :

```
1  public class Assert<T>
2  {
3      protected String _name;
4      protected T _a;
5      protected T _b;
6
7      public Assert(String name, T a, T b)
8      {
9          // TODO
10     }
11
12     public bool Run()
13     {
14         // TODO
15     }
16
17     protected virtual bool Test()
18     {
19         return false;
20     }
21
22     protected virtual void Success()
23     {
24         // TODO
25     }
26
27     protected virtual void Exception()
28     {
29         // TODO
30     }
31 }
```

L'initialisateur doit assigner tous les paramètres aux attributs.
`Run` doit exécuter `Test` et exécuter `Succès` en cas de réussite ou `Exception` en cas d'échec. Bien entendu la valeur renvoyée par `Test` doit être retournée par `Run`.

`Success` doit afficher : *"Test '<Nom du test>' succès."*
`Exception` doit afficher : *"Test '<Nom du test>' fail."*

3.1.2 AssertEqual

Tout le code de cette section doit être dans `AssertEqual.cs`. Cette classe teste si la variable `a` est égale à `b`
Le squelette de la classe est :

```
1  public class AssertEqual<T>: Assert<T>, IAssert
2  {
3      public AssertEqual(String name, T a, T b): base(name, a, b)
4      {
5      }
6
7      protected override bool Test()
8      {
9          // TODO
10     }
11
12     protected override void Exception()
13     {
14         // TODO
15     }
16
17     public new void Run()
18     {
19         base.Run();
20     }
21 }
```

La méthode `Test` doit tester l'égalité entre `a` et `b`.
La méthode `Exception` doit afficher : *"Test '<Nom du test>' fail : <a> is not equal to <b>."*

3.1.3 AssertNotEqual

Le squelette d'`AssertNotEqual` est minimal et ne vous sera pas donné car au fond cette classe est l'opposé de `AssertEqual`. N'oublier pas de changer le message d'`Exception`.

3.1.4 AssertArrayEqual

Tout le code de cette section doit être dans `AssertArrayEqual.cs`. Le squelette d'`AssertArrayEqual` est :

```
1  public class AssertArrayEqual<T>:Assert<T[]>, IAssert
2  {
3      protected int _size;
4
5      public AssertArrayEqual(string name, T[] a, T[] b, int size)
6                                                      : base(name, a, b)
7      {
8          _size = size;
9      }
10
11     protected override bool Test()
12     {
13         // TODO
14     }
15
16     protected override void Exception()
17     {
18         // TODO
19     }
20
21     public new void Run()
22     {
23         base.Run();
24     }
25 }
```

`Exception` affiche si `a = {1, 2}` et `b = {1, 2, 3}` : *"Test '<Nom du test>' fail : [1, 2] is not equal to [1, 2, 3]."*

3.1.5 AssertArrayNotEqual

Tout le code de cette section doit être dans `AssertArrayNotEqual.cs`. Le squelette ne vous sera pas donné.

3.1.6 Test

La classe `Test` doit contenir une liste d'assertion et un nom. Lorsqu'on appelle la méthode Run, l'ensemble des assertions doivent être testées. Vous devez obtenir ce résultat si 3 tests (`strip basique`, `sort basique` et `fibo basic`) ont réussi et que l'objet test s'appelle MISC. La méthode pour ajouter une assertion doit s'appeler `AddAssert`.

```
Start test suite: MISC
Start test number: 0
Test 'strip basic' success.
Start test number: 1
Test 'sort basic' success.
Start test number: 2
Test 'fibo basic' success.
```

3.2 Misc

Dans cette partie, ce n'est pas vraiment votre capacité à résoudre les exercices qui nous intéresse mais comment vous allez faire pour tester la validité de votre code.Vous devez utiliser Test et les asserts pour tester dans `ConsoleMain.cs` vos résultats. Un code non testé n'est pas considéré comme valide.

3.2.1 StripSpace

Cette fonction enlève tout les `whitespace` inutiles en fin de `string`.

```
1 public static String StripSpace(String str);
```

3.2.2 Swap

Cette fonction échange les valeurs que référence `a` et `b`.

```
1 public static void Swap(ref int a, ref int b);
```

3.2.3 Sort

Cette fonction trie en ordre croissant un tableau.

```
1 public static void Sort(int[] arr);
```

3.2.4 FiboIter

Renvoie la nième valeur de la suite de fibonacci.

```
1 public static int FiboIter(int n);
```

4 Bonus

4.1 More

Dans cette partie, rajouter des types d'assertions et justifiez dans le README votre choix.

4.2 Time

Dans cette partie, rajoutez un timer qui indique à la fin de chaque de test combien de temps le test a duré.

4.3 Pretty

Rajouter de la couleur aux sorties des assertions et de `Test`. Vous pouvez rajouter tout ce qui vous semble bon pour rendre la sortie des tests plus claire.

4.4 Verbose

Dans cette partie, rajoutez une méthode `RunVerbose` dans la classe `Test` qui fonctionne comme `Run` cependant elle affiche beaucoup plus d'informations sur les tests (Vous choisirez les informations supplémentaires à afficher).

4.5 Dépendance

Rajoutez à la notion de test la notion de dépendance. Il ne faut pas tester l'assertion `b` qui dépend de l'assertion `a` si celle si n'a pas été un succès. Cependant l'assertion `c` qui est indépendant de ces 2 assertions doit être exécuté.

4.6 GUI

Implémentez une interface graphique qui récapitule l'ensemble de vos tests.

4.7 Fork

Dans cette partie vous pouvez essayer de paralléliser tout les tests.

4.8 Vous avez vraiment beaucoup de temps

Surprenez nous.

If you knew time as well as I do, You wouldn't talk about wasting it