Tests Unitaires

I) Concepts

Vérifiez que votre code fonctionne comme prévu en créant et en exécutant des tests unitaires. Il s'agit de tests unitaires, car vous décomposez les fonctionnalités de votre programme en comportements testables discrets que vous pouvez tester en tant qu'unités individuelles.

Utilisez une *infrastructure de tests unitaires* pour créer des tests unitaires, les exécuter et signaler les résultats de ces tests. Réexécutez des tests unitaires quand vous apportez des modifications pour vérifier que votre code fonctionne toujours correctement.

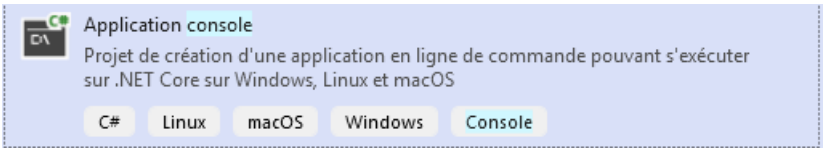
II) Créer un nouveau projet C#

II.1) La solution

Créer un dossier pour votre projet puis clic-droit Ouvrir avec Visual Studio

Fichier, Nouveau, Projet

Sélectionner le modèle



Donner un nom au projet, garder la version courante du Framework et cliquer sur créer

II.2) Les dépendances nuget

Ajouter les dépendances au projet en cliquant sur Outils, Gestionnaire de package NuGet, Gérer les packages nugets pour la solution

```
MySql.EntityFrameworkCore

Microsoft.EntityFrameworkCore.Tools

Microsoft.EntityFrameworkCore
```

Vous pouvez voir les packages inclus en double cliquant sur le nom du projet (fichier nomProjet.csproj)

II.3) Créer une classe compte comme suit

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace TestsUnitaires
{
    class Compte
    {
        public string Client { get; set; }
        public double Solde { get; set; }
        public Compte(string nom, double solde)
        {
            Client = nom;
            Solde = solde;
        }

        /// <summary>
        /// Permet de déduire le montant du solde
        /// </summary>
        /// <param name="montant"></param>
        public void Debit(double montant)
        {
            if (montant > Solde)
            {
                throw new ArgumentOutOfRangeException("montant");
            }

            if (montant < 0)
            {
                throw new ArgumentOutOfRangeException("montant");
            }

            Solde += montant; // code incorrect volontairement
        }

        public void Credit(double montant)
        {
            if (montant < 0)
            {
                throw new ArgumentOutOfRangeException("montant");
            }

            Solde += montant;
        }
    }
}
```

II.4) Mettre à jour le main

Mettre à jour le main avec le code ci-dessous

```
        static void Main(string[] args)
        {
            Compte compte = new Compte("Toto", 200);
            compte.Credit(100);
            compte.Debit(50);
            Console.WriteLine("Le nouveau solde est de : {0}€", compte.Solde);
            Console.ReadKey(); // permet de garder la console ouverte

        }
```

III) Exécuter le code

Exécuter le code et constater l'erreur sur le nouveau solde.

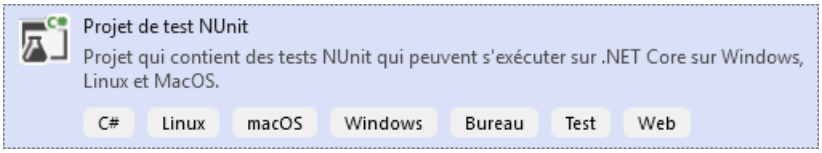
IV) Créer le projet de Tests

IV.1) Ajouter un projet

Créer un dossier pour votre projet puis clic-droit Ouvrir avec Visual Studio

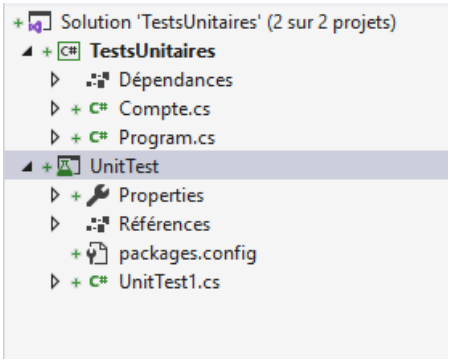
Fichier, **Ajouter**, Nouveau Projet

Sélectionner le modèle



Donner un nom au projet, garder la version courante du Framework et cliquer sur créer

Voici ce que l'on obtient dans l'explorateur de solution



IV.2) Ajouter une référence entre les projets

Sous le projet de Test, clic droit ajouter une référence, retrouver le projet de départ dans la solution actuelle.

V) Créer la classe test

V.1) Créer la classe

Créez une classe de test pour vérifier la classe `compte`. Vous pouvez utiliser le fichier *UnitTest1.cs* qui a été généré par le modèle de projet, mais donnez au fichier et à la classe des noms plus descriptifs.

Voici le résultat

```
using NUnit.Framework;

namespace UnitTest
{
    public class CompteTest
    {
        [SetUp]
        public void Setup()
        {
        }

        [Test]
        public void Test1()
```

```
        {
            Assert.Pass();
        }
    }
}
```

Il existe au moins trois comportements à vérifier :

- La méthode lève une exception ArgumentOutOfRangeException si le montant du débit est supérieur au solde.
- La méthode lève ArgumentOutOfRangeException si le montant du débit est inférieur à zéro.
- Si le montant du débit est valide, la méthode le soustrait du solde du compte.

V.2) Créer les méthodes de tests

Voici les 3 méthodes de tests

```
        [Test]
        public void Debit_MontantValide()
        {
            // Arrange
            double soldeDepart = 11.99;
            double montantDebite = 4.55;
            double attendu = 7.44;
            Compte compte = new Compte("Mr Toto", soldeDepart);

            // Act
            compte.Debit(montantDebite);

            // Assert
            double soldeActuel = compte.Solde;
            Assert.AreEqual(attendu, soldeActuel, 0.001, "Compte mal débité");
        }

        [Test]
        public void Debit_MontantNegatif()
        {
            // Arrange
            double soldeDepart = 11.99;
            double montantDebite = -4;
            Compte compte = new Compte("Mr Toto", soldeDepart);

            // Act et  Assert
            Assert.Throws<ArgumentOutOfRangeException>(() =>
compte.Debit(montantDebite));
        }

        [Test]
        public void Debit_MontantSuperieurSolde()
        {
            // Arrange
            double soldeDepart = 11.99;
            double montantDebite = -44.55;
            Compte compte = new Compte("Mr Toto", soldeDepart);

            // Act et  Assert
            Assert.Throws<ArgumentOutOfRangeException>(() =>
compte.Debit(montantDebite));

        }
```

V.3) Améliorer les messages

- Créer des constantes pour contenir les messages dans la classe compte

```
public const string DebitMontantSuperieurSoldeMessage = "Le montant à débiter est
supérieur au solde";

public const string DebitMontantNegatifMessage = "Le montant à débiter est négatif";
```

- Modifier les conditions dans la méthode débit

```
if (montant > Solde)
{
    throw new ArgumentOutOfRangeException("montant", montant,
DebitMontantSuperieurSoldeMessage);
}

if (montant < 0)
{
    throw new ArgumentOutOfRangeException("montant", montant,
DebitMontantNegatifMessage);
}
```

V.4) Factorisation des valeurs de départ

Pour chacune de ces méthodes de tests, nous devons déclarer un compte.

Celui-ci peut être déclarer avant l'exécution de chaque test s'il est placé dans la méthode setup.

La classe devient alors

```
using NUnit.Framework;
using System;
using TestsUnitaires;

namespace UnitTest
{
    public class CompteTest
    {
        Compte compte;

        [SetUp]
        public void Setup()
        {
            double soldeDepart = 11.99;
            compte = new Compte("Mr Toto", soldeDepart);
        }

        [Test]
        public void Debit_MontantValide()
        {
            // Arrange
            double montantDebite = 4.55;
            double attendu = 7.44;

            // Act
            compte.Debit(montantDebite);

            // Assert
            double soldeActuel = compte.Solde;
            Assert.AreEqual(attendu, soldeActuel, 0.001, "Compte mal débité");
        }
```

```
        [Test]
        public void Debit_MontantNegatif()
        {
            // Arrange
            double montantDebite = -4;

            // Act et  Assert
            Assert.Throws<ArgumentOutOfRangeException>(() =>
compte.Debit(montantDebite));
        }

        [Test]
        public void Debit_MontantSuperieurSolde()
        {
            // Arrange
            double montantDebite = -44.55;

            // Act et  Assert
            Assert.Throws<ArgumentOutOfRangeException>(() =>
compte.Debit(montantDebite));
        }

    }

}
```

L'annotation TearDown permet de supprimer ou détruire les instances créées pour les tests.