C# : tests unitaires avec **MSTest**

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Plan

5. Pre/Post test

6. Quelques décorateurs pour les tests
   - `[TestCategory]`
   - `[Ignore]`
   - `[DataRow]`
   - `[DynamicData]`
   - `[TestProperty]`

Plan

7. Moq
   - `Of`
   - `Setup ... Returns`
   - `SetupSequence ... Returns`
   - `Verify`
   - `Times`
   - `It.IsAny<type>()`
   - `It.IsInRange()`
   - `When`
   - `SetUp ... Throws`

MSTest

Deux catégories de tests

- tests manuels :
  - coûteux
  - répétitif (corriger et retester)
- tests automatiques (via des programmes)

MSTest

Plusieurs niveaux de tests

- **tests unitaires** : permettent de tester une partie isolée de l'application.
- **tests d'intégration** : permettent de vérifier que toutes les parties isolées fonctionnent correctement ensemble.
- **tests de charge** : permettent de vérifier si un système peut gérer une charge spécifiée : le nombre d'utilisateurs simultanés...
- **tests d'acceptation** : permettent de vérifier que l'application respecte bien le besoin fonctionnel.

MSTest

Tests unitaires

Programme permettant de vérifier le bon fonctionnement d'une partie (ou une unité) de l'application.

MSTest

Tests unitaires

Programme permettant de vérifier le bon fonctionnement d'une partie (ou une unité) de l'application.

Objectif

Trouver un maximum d'erreurs pour les corriger.

MSTest

Tests unitaires

Programme permettant de vérifier le bon fonctionnement d'une partie (ou une unité) de l'application.

Objectif

Trouver un maximum d'erreurs pour les corriger.

Remarque

Si le test ne détecte pas d'erreurs $\not\Rightarrow$ il n'y en a pas.

MSTest

Quelques frameworks de tests pour **C#**

- **MSTest** : développé par **Microsoft** pour effectuer des tests unitaires.
- **NUnit** : la version **.NET** de la famille des frameworks de tests **XUnit** inspirée de **JUnit** de **Java** (fonctionnant avec **.NET Framework** et **Xamarin**).
- **xUnit** : écrit par l'auteur de **NUnit** (fonctionnant avec **.NET Framework**, **.NET Core** et **Xamarin**).

MSTest

TestClass (classe de test)

Classe **C#** décorée par `[TestClass]`

- Contenant quelques méthodes de test (décorée par `[TestMethod]`)
- Permettant de tester le bon fonctionnement d'une classe (en testant ses méthodes)

MSTest

Règles de nommage : classe

- Nom de la classe de test = Nom de la classe testée + `"Test"`
- Exemple : `CalculTest`

MSTest

Règles de nommage : classe

- Nom de la classe de test = Nom de la classe testée + `"Test"`
- Exemple : `CalculTest`

Règles de nommage : méthode

- Nom de la méthode de test = Nom de la méthode testée + `"_"` + Inputs + `"_"` + Output
- Exemple : `TestSomme_2_3_5()` ou `Somme_2_3_5()`

MSTest

Étape

- Création d'un Projet Console (appelé `ProjetTest`) : `Fichier > Nouveau > Projet > C# > Application console (.NET Framework)`
- Création d'un Projet Console (appelé `UnitTestProject`) : Faire clic droit sur la solution dans l'`Explorateur de solution` et aller à `Ajouter Nouveau projet > C# > Test > Projet de test unitaire (.NET Framework)`
- Pour chaque classe créée, on lui associe une classe de test dans le projet de test unitaire
- On prépare le test et ensuite on le lance : s'il y a une erreur, on la corrige et on relance le test.

MSTest

Création d'une première classe `Calcul`

```
namespace ProjetTest
{
    public class Calcul
    {
        public int Somme(int x, int y)
        {
            return x + y;
        }

        public int Division(int x, int y)
        {
            if (y == 0)
                throw new DivideByZeroException();
            return x / y;
        }
    }
}
```

MSTest

Le code généré pour la classe de test `UnitTest1`

```
namespace UnitTestProject
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

Renommer le fichier et la classe `CalculTests` pour respecter les conventions.

Préparons une méthode pour tester `Somme`

```
namespace UnitTestProject
{
    [TestClass]
    public class CalculTests
    {
        [TestMethod]
        public void TestSomme_2_2_4()
        {
            var calcul = new Calcul();
            if (calcul.Somme(2, 2) != 4)
            {
                Assert.Fail("problème d'adition");
            }
        }
    }
}
```

Préparons une méthode pour tester `Somme`

```
namespace UnitTestProject
{
    [TestClass]
    public class CalculTests
    {
        [TestMethod]
        public void TestSomme_2_2_4()
        {
            var calcul = new Calcul();
            if (calcul.Somme(2, 2) != 4)
            {
                Assert.Fail("problème d'adition");
            }
        }
    }
}
```

`Fail()` permet de faire échouer le test.

MSTest

Assertions

- Méthodes statiques définies dans la classe `Assert`.
- Permettant de vérifier le bon déroulement d'un test : si la vérification échoue, l'assertion lève une exception et le test échoue.

MSTest

Pour lancer le test

- Aller dans `Test > Exécuter`
- Cliquer sur `Tous les tests`

MSTest

`Assert` **contient des méthodes statiques qui permettent de réaliser certaines comparaisons**

```
namespace UnitTestProject
{
    [TestClass]
    public class CalculTests
    {
        [TestMethod]
        public void TestSomme_2_2_4()
        {
            var calcul = new Calcul();
            Assert.AreEqual(4, calcul.Somme(2, 2));
        }
    }
}
```

MSTest

`Assert` **contient des méthodes statiques qui permettent de réaliser certaines comparaisons**

```
namespace UnitTestProject
{
    [TestClass]
    public class CalculTests
    {
        [TestMethod]
        public void TestSomme_2_2_4()
        {
            var calcul = new Calcul();
            Assert.AreEqual(4, calcul.Somme(2, 2));
        }
    }
}
```

Relancer le test et vérifier qu'il se termine correctement.

MSTest

On peut aussi ajouter un message qui sera affiché en cas d'échec

```
namespace UnitTestProject
{
    [TestClass]
    public class CalculTests
    {
        [TestMethod]
        public void TestSomme_2_2_4()
        {
            var calcul = new Calcul();
            Assert.AreEqual(4, calcul.Somme(2, 2), "Resultat
               attendu = 4");
        }
    }
}
```

MSTest

On peut aussi ajouter un message qui sera affiché en cas d'échec

```
namespace UnitTestProject
{
    [TestClass]
    public class CalculTests
    {
        [TestMethod]
        public void TestSomme_2_2_4()
        {
            var calcul = new Calcul();
            Assert.AreEqual(4, calcul.Somme(2, 2), "Resultat
               attendu = 4");
        }
    }
}
```

Relancer le test et vérifier qu'il se termine correctement.

MSTest

Autres assertions

- `Assert.AreEqual()` : vérifier si deux objets sont égaux (la réciproque `AreNotEqual()`)
- `Assert.AreSame()` : vérifier si deux objets sont identiques (la réciproque `AreNotSame()`)
- `Assert.IsNull()` (la réciproque `IsNotNull()`)
- `Assert.IsTrue()` (la réciproque `IsFalse()`)
- `Assert.IsInstanceOfType()` (la réciproque `IsNotInstanceOfType()`)

MSTest

Et si on voulait vérifier si une méthode lève une exception

```
[TestMethod]
[ExpectedException(typeof(DivideByZeroException))]
public void TestDivision_2_0_Exception()
{
        var calcul = new Calcul();
        calcul.Division(2, 1);
}
```

MSTest

Et si on voulait vérifier si une méthode lève une exception

```
[TestMethod]
[ExpectedException(typeof(DivideByZeroException))]
public void TestDivision_2_0_Exception()
{
        var calcul = new Calcul();
        calcul.Division(2, 1);
}
```

Le test est valide si la méthode lève une exception arithmétique. Sinon, il échoue.

MSTest

Fluent Assertions

- Package **NuGet**.
- Contenant un ensemble de méthodes permettant d'améliorer l'écriture et la lisibilité des assertions en s'approchant du langage naturel.
- Documentation : `https://fluentassertions.com/`.

MSTest

Installation

- Aller dans `Outils > Gestionnaire de package NuGet > Gérer les packages NuGet pour la solution`.
- Chercher `Fluent Assertions` dans l'onglet `Parcourir`.
- Choisir la dernière version stable et cliquer sur `Installer`.

MSTest

Réécrivons `TestSomme_2_2_4` en utilisant Fluent Assertions

```
[TestMethod]
public void TestSomme_2_2_4()
{
      var calcul = new Calcul();
      calcul.Somme(2, 2).Should().Be(4, "2 + 2 = 4");
}
```

MSTest

Pour vérifier que la méthode `division` lève une exception, on utilise `Throw<NomException>` qui s'applique sur des actions

```
[TestMethod]
public void TestDivision_2_0_Exception()
{
      var calcul = new Calcul();
      Action act = () =>  calcul.Division(2, 0);
      act.Should().Throw<DivideByZeroException>();
}
```

MSTest

Pour vérifier que la méthode `division` lève une exception, on utilise `Throw<NomException>` qui s'applique sur des actions

```
[TestMethod]
public void TestDivision_2_0_Exception()
{
      var calcul = new Calcul();
      Action act = () =>  calcul.Division(2, 0);
      act.Should().Throw<DivideByZeroException>();
}
```


Plus besoin du décorateur

```
[ExpectedException(typeof(DivideByZeroException))]
```

MSTest

Nous pouvons aussi filtrer selon le message de l'exception

```
[TestMethod]
public void TestDivision_2_0_Exception()
{
      var calcul = new Calcul();
      Action act = () =>  calcul.Division(2, 0);
      act.Should().Throw<DivideByZeroException>()
                  .WithMessage("Tentative de division par zéro.");
}
```

MSTest

Si on ne connaît pas le message exact de l'exception, on peut faire

```
[TestMethod]
public void TestDivision_2_0_Exception()
{
      var calcul = new Calcul();
      Action act = () =>  calcul.Division(2, 0);
      act.Should().Throw<DivideByZeroException>()
                  .Where(e => e.Message.StartsWith("Tenta"));
}
```

MSTest

Ou aussi

```
[TestMethod]
public void TestDivision_2_0_Exception()
{
      var calcul = new Calcul();
      Action act = () =>  calcul.Division(2, 0);
      act.Should().Throw<DivideByZeroException>()
                  .WithMessage("*division*");
}
```

MSTest

Pour vérifier qu'une méthode ne lève pas d'exception

```
[TestMethod]
public void TestDivision_2_2_NotException()
{
      var calcul = new Calcul();
      Action act = () =>  calcul.Division(2, 2);
      act.Should().NotThrow<DivideByZeroException>();
}
```

MSTest

Considérons la méthode de test suivante

```
[TestMethod]
public void TestSomme_NombresPositifs_NombrePositif()
{
      var calcul = new Calcul();
      5.Should().Be(calcul.Somme(2, 4));
      10.Should().Be(calcul.Somme(5, 4));
}
```

MSTest

Considérons la méthode de test suivante

```
[TestMethod]
public void TestSomme_NombresPositifs_NombrePositif()
{
      var calcul = new Calcul();
      5.Should().Be(calcul.Somme(2, 4));
      10.Should().Be(calcul.Somme(5, 4));
}
```

En lançant le test, seul le premier échec est mentionné.

Pour avoir tous les échecs, on peut utiliser `AssertionScope`

```
[TestMethod]
public void TestSomme_NombresPositifs_NombrePositif()
{
    using (new AssertionScope())
    {
        var calcul = new Calcul();
        5.Should().Be(calcul.Somme(2, 4));
        10.Should().Be(calcul.Somme(5, 4));
    }
}
```

Pour avoir tous les échecs, on peut utiliser `AssertionScope`

```
[TestMethod]
public void TestSomme_NombresPositifs_NombrePositif()
{
    using (new AssertionScope())
    {
        var calcul = new Calcul();
        5.Should().Be(calcul.Somme(2, 4));
        10.Should().Be(calcul.Somme(5, 4));
    }
}
```

En lançant le test, le résultat est

```
TestSomme_NombresPositifs_NombrePositif
   Source: CalculTests.cs ligne 35
   Durée: 238 ms

  Message:
    Expected value to be 6, but found 5.
    Expected value to be 9, but found 10.
```

MSTest

Assertions pour les types numériques

- `Be(number)`
- `NotBe(number)`
- `BeGreaterOrEqualTo(number)`
- `BeGreaterThan(number)`
- `BeLessOrEqualTo(number)`
- `BeLessThan(number)`
- `BePositive()`
- `BeNegative()`
- `BeInRange(inf, sup)`
- `NotBeInRange(inf, sup)`
- `Match(action)`

MSTest

Exemple avec `Should Match`

```
[TestMethod]
public void TestSomme_3_3_NombrePair()
{
      var calcul = new Calcul();
      calcul.Somme(3, 3).Should().Match(x => x % 2 == 0);
}
```

MSTest

Exemple avec `Should Match`

```
[TestMethod]
public void TestSomme_3_3_NombrePair()
{
      var calcul = new Calcul();
      calcul.Somme(3, 3).Should().Match(x => x % 2 == 0);
}
```


Lancer le test et vérifier qu'il se termine correctement.

MSTest

Assertions pour les chaînes de caractères

- `Be(string)` et `NotBe(string)` (sensible à la casse)
- `BeEquivalentTo(string)` et `NotBeEquivalentTo(string)` (insensible à la casse)
- `BeNull()` et `NotBeNull()`
- `BeEmpty()` et `NotBeEmpty()`
- `HaveLength(number)`
- `BeNullOrWhiteSpace()` et `NotBeNullOrWhiteSpace()`
- `StartWith(string)`, `NotStartWith(string)`, `StartWithEquivalent(string)` et `NotStartWithEquivalentOf(string)`
- `Match(string)`, `NotMatch(string)`, `MatchEquivalentOf(string)` et `NotMatchEquivalentOf(string)` (`string` peut contenir le joker `*`)
- ...

MSTest

Les surcharges de la méthode `Contain`

- `Contain(string)`
- `Contain(string, Exactly.Once())`
- `Contain(string, AtLeast.Once())`
- `Contain(string, MoreThan.Twice())`
- `Contain(string, AtMost.Times(number))`
- `Contain(string, LessThan.Twice())`
- ...

MSTest

Les surcharges de la méthode `Contain`

- `Contain(string)`
- `Contain(string, Exactly.Once())`
- `Contain(string, AtLeast.Once())`
- `Contain(string, MoreThan.Twice())`
- `Contain(string, AtMost.Times(number))`
- `Contain(string, LessThan.Twice())`
- ...

Les mêmes surcharges existent pour `ContainEquivalentOf`.

MSTest

Les extensions de la méthode `Contain`

- `ContainAll(string1, string2,... stringN)`
- `ContainAny(string1, string2,... stringN)`
- `NotContainAll(string1, string2,... stringN)`
- `NotContainAny(string1, string2,... stringN)`
- `NotContain(string)`
- ...

MSTest

Autres assertions

- Basiques : `Be()`, `NotBe()`, `BeNull()`, `NotBeNull()`, `BeOfType<type>()`, `BeSameAs`, `NotBeSameAs`...
- DateTime : `Be()`, `BeAfter()`, `BeBefore()`, `BeSameDateAs()`, `BeOneOf()`, `HaveDay`, `HaveYear`...
- Collections : `Equal()`, `BeEquivalentTo()` (pas forcément dans l'ordre), `HaveCountGreaterThan()`, `HaveCountLessThan()`, `StartWith()`, `Contain`, `ContainInOrder`...
- Dictionaries : `ContainKey()`, `ContainKeys()`, `ContainValue()`, `ContainValues()`, `HaveCount()`...
- ...

MSTest

Dans certains cas

- Avant de démarrer un test, il faut faire certains traitements :
  - instancier un objet de la classe,
  - se connecter à une base de données,
  - ouvrir un fichier...
- Après le test, il faut aussi fermer certaines ressources : connexion à une base de données, socket...

MSTest

Dans certains cas

- Avant de démarrer un test, il faut faire certains traitements :
  - instancier un objet de la classe,
  - se connecter à une base de données,
  - ouvrir un fichier...
- Après le test, il faut aussi fermer certaines ressources : connexion à une base de données, socket...

Solution

Utiliser des méthodes décorées par `[ClassInitialize]`, `[ClassCleanup]`, `[TestInitialize]` et `[TestCleanup]`.

Pour vérifier, ajoutons les quatre méthodes suivantes

```
[ClassInitialize()]
public static void ClassInit(TestContext context) {
    Debug.WriteLine("Class Initialize");
}
[ClassCleanup()]

public static void ClassCleanup()
{
    Debug.WriteLine("Class Cleanup");
}

[TestInitialize()]
public void Initialize() {
    Debug.WriteLine("Test Initialize");
}

[TestCleanup()]
public void Cleanup() {
    Debug.WriteLine("Test Cleanup");
}
```

MSTest

Allez dans le menu `Test` et cliquer sur `Déboguer tous les tests` et vérifier l'affichage suivant

```
Class Initialize
Test Initialize
Test Cleanup
Test Initialize
Test Cleanup
Test Initialize
Test Cleanup
Test Initialize
Test Cleanup
Test Initialize
Test Cleanup
Class Cleanup
```

MSTest

Comprenons les annotations de méthodes précédentes

- `[ClassInitialize()]` : la méthode décorée sera exécutée avant tous les tests de cette classe.
- `[ClassCleanup()]` : la méthode décorée sera exécutée seulement après exécution du dernier test de cette classe.
- `[TestInitialize()]` : la méthode décorée sera exécutée avant chaque test.
- `[TestCleanup()]` : la méthode décorée sera exécutée après chaque test.

MSTest

Utilisons ces méthodes pour restructurer la classe `CalculTest` (Première partie)

```
[TestClass]
public class CalculTests
{
    Calcul calcul;

    [ClassInitialize()]
    public static void ClassInit(TestContext context) {}

    [ClassCleanup()]
    public static void ClassCleanup() {}

    [TestInitialize()]
    public void Initialize() {
        calcul = new Calcul();
    }

   [TestCleanup()]
   public void Cleanup() {
        calcul = null;
   }
```

(Deuxième partie)

```
    [TestMethod]
    public void TestSomme_2_2_4()
    {
        calcul.Somme(2, 2).Should().Be(4, "2 + 2 = 4");
    }
    [TestMethod]
    public void TestDivision_2_0_Exception()
    {
        Action act = () =>  calcul.Division(2, 0);
        act.Should().Throw<DivideByZeroException>()
                    .WithMessage("*division*");
    }
    [TestMethod]
    public void TestDivision_2_2_NotException()
    {
        Action act = () => calcul.Division(2, 2);
        act.Should().NotThrow<DivideByZeroException>();
    }
    [TestMethod]
    public void TestSomme_NombresPositifs_NombrePositif()
    {
        using (new AssertionScope())
        {
            6.Should().Be(calcul.Somme(2, 4));
            9.Should().Be(calcul.Somme(5, 4));
        }
    }
    [TestMethod]
    public void TestSomme_3_3_NombrePair()
    {
        calcul.Somme(3, 3).Should().Match(x => x % 2 == 0);
    }
}
```

MSTest

Pour définir les caractéristiques d'un test, on utilise `[TestCategory]`

```
[TestMethod]
[TestCategory("On vérifie que la somme de 2 et 2 est bien 4")]
public void TestSomme_2_2_4()
{
    calcul.Somme(2, 2).Should().Be(4, "2 + 2 = 4");
}
```

MSTest

Pour désactiver un test, on utilise `[Ignore]`

```
[Ignore]
[TestMethod]
public void TestDivision_2_0_Exception()
{
    Action act = () =>  calcul.Division(2, 0);
    act.Should().Throw<DivideByZeroException>()
                .WithMessage("*division*");
}
```

MSTest

Pour paramétrer le test et le répéter plusieurs fois, on utilise `[DataRow()]`

```
[DataTestMethod]
[DataRow(1, 1, 2)]
[DataRow(2, 2, 4)]
[DataRow(3, 3, 6)]
[DataRow(4, 4, 8)]
[DataRow(5, 0, 5)]
public void TestSomme_NombresPositifs_NombrePositif(
   int a, int b, int c)
{
    c.Should().Be(calcul.Somme(a, b));
}
```

Le résultat du test

```
Le test a plusieurs sorties de résultat
    6 Réussite

Résultats

    1)  TestSomme_NombresPositifs_NombrePositif
      Durée: 70 ms

    2)  TestSomme_NombresPositifs_NombrePositif (1,1,2)
      Durée: < 1 ms

    3)  TestSomme_NombresPositifs_NombrePositif (2,2,4)
      Durée: < 1 ms

    4)  TestSomme_NombresPositifs_NombrePositif (3,3,6)
      Durée: < 1 ms

    5)  TestSomme_NombresPositifs_NombrePositif (4,4,8)
      Durée: < 1 ms

    6)  TestSomme_NombresPositifs_NombrePositif (5,0,5)
      Durée: < 1 ms
```

MSTest

Pour alimenter le test avec des valeurs dynamiques, on utilise `[DynamicData()]`

```
[DataTestMethod]
[DynamicData(nameof(DataSource), DynamicDataSourceType.Method)]
public void TestSomme_NombresPositifs_NombrePositif(int a, int b, int c
    )
{
    c.Should().Be(calcul.Somme(a, b));
}
```

MSTest

Pour alimenter le test avec des valeurs dynamiques, on utilise `[DynamicData()]`

```
[DataTestMethod]
[DynamicData(nameof(DataSource), DynamicDataSourceType.Method)]
public void TestSomme_NombresPositifs_NombrePositif(int a, int b, int c
   )
{
    c.Should().Be(calcul.Somme(a, b));
}
```

La méthode source de données

```
public static IEnumerable<object[]> DataSource()
{
    yield return new object[] { 1, 1, 2 };
    yield return new object[] { 2, 2, 4 };
    yield return new object[] { 3, 3, 6 };
    yield return new object[] { 4, 4, 8 };
    yield return new object[] { 5, 0, 5 };
}
```

Le résultat du test

```
Le test a plusieurs sorties de résultat
   6 Réussite

Résultats

    1)  TestSomme_NombresPositifs_NombrePositif
      Durée: 38 ms

    2)  TestSomme_NombresPositifs_NombrePositif (1,1,2)
      Durée: < 1 ms

    3)  TestSomme_NombresPositifs_NombrePositif (2,2,4)
      Durée: < 1 ms

    4)  TestSomme_NombresPositifs_NombrePositif (3,3,6)
      Durée: < 1 ms

    5)  TestSomme_NombresPositifs_NombrePositif (4,4,8)
      Durée: < 1 ms

    6)  TestSomme_NombresPositifs_NombrePositif (5,0,5)
      Durée: < 1 ms
```

MSTest

Pour définir des propriétés sous forme clé-valeur, on utilise `[TestProperty()]`

```
[TestMethod]
[TestProperty("i", "-3")]
[TestProperty("j", "-5")]
public void TestSomme_NombresNegatifs_NombreNegatif()
{
    int i = int.Parse(TestContext.Properties["i"].ToString());
    int j = int.Parse(TestContext.Properties["j"].ToString());
    calcul.Somme(i, j).Should().BeNegative();
}
```

MSTest

Pour définir des propriétés sous forme clé-valeur, on utilise `[TestProperty()]`

```
[TestMethod]
[TestProperty("i", "-3")]
[TestProperty("j", "-5")]
public void TestSomme_NombresNegatifs_NombreNegatif()
{
    int i = int.Parse(TestContext.Properties["i"].ToString());
    int j = int.Parse(TestContext.Properties["j"].ToString());
    calcul.Somme(i, j).Should().BeNegative();
}
```


Sans oublier de déclarer `TestContext` **comme propriété de la classe**

```
[TestClass]
public class CalculTests
{
    public TestContext TestContext { get; set; }

    // + contenu précédent
```

MSTest

Mock ?

- Doublure, objet fictif, objet factice (fake object)
- permettant de reproduire le comportement d'un objet réel non implémenté

MSTest

Mock ?

- Doublure, objet fictif, objet factice (fake object)
- permettant de reproduire le comportement d'un objet réel non implémenté

Moq ?

- Framework open-source pour **C#**
- Générateur automatique de doublures
- Un seul type de Mock possible et une seule façon de le créer

MSTest

Installation

- Aller dans `Outils > Gestionnaire de package NuGet > Gérer les packages NuGet pour la solution`.
- Chercher `Moq` dans l'onglet `Parcourir`.
- Choisir la dernière version stable et cliquer sur `Installer`.

MSTest

Exemple, supposant qu'on

- a une interface `ICalculService` ayant une méthode `Carre()`
- veut développer une méthode `SommeCarre()` dans `Calcul` qui utilise la méthode `Carre()` de cette interface `ICalculService`

MSTest

L'interface `ICalculService`

```
namespace ProjetTest
{
    public interface ICalculService
    {
        int Carre(int x);
    }
}
```

MSTest

Nouveau contenu de la classe `Calcul`

```
namespace ProjetTest
{
    public class Calcul
    {
        ICalculService calculService;

        public Calcul(ICalculService calculService)
        {
            this.calculService = calculService;
        }

        public int SommeCarre(int x, int y)
        {
            return Somme(calculService.Carre(x), calculService.Carre(y));
        }

        public int Somme(int x, int y)
        {
            return x + y;
        }

        public int Division(int x, int y)
        {
            if (y == 0)
                throw new DivideByZeroException();
            return x / y;
        }
    }
}
```

Pour tester la classe `Calcul` dans `CalculTests`, il faut commencer par instancier `ICalculService`

```
namespace UnitTestProject
{
    [TestClass]
    public class CalculTests
    {

        Calcul calcul;

        ICalculService calculService;

        public TestContext TestContext { get; set; }

        [TestInitialize()]
        public void Initialize()
        {
            calcul = new Calcul(calculService);
        }

        [TestCleanup()]
        public void Cleanup()
        {
            calcul = null;
        }

        [TestMethod]
        public void TestSommeCarre_2_3_13()
        {
            calcul.SommeCarre(2, 3).Should().Be(13, "2*2 + 3*3 = 13");
        }

        // + le code précédent
```

MSTest

En testant, on aura l'erreur suivante

```
La méthode de test UnitTestProject.CalculTests.TestSommeCarre_2_3_13 a
    levé une exception :
   System.NullReferenceException: La référence d'objet n'est pas dé
     finie à une instance d'un objet.
 Arborescence des appels de procédure:
   Calcul.SommeCarre(Int32 x, Int32 y) ligne 20
   CalculTests.TestSommeCarre_2_3_13() ligne 37
```

MSTest

En testant, on aura l'erreur suivante

```
La méthode de test UnitTestProject.CalculTests.TestSommeCarre_2_3_13 a
    levé une exception :
   System.NullReferenceException: La référence d'objet n'est pas dé
     finie à une instance d'un objet.
 Arborescence des appels de procédure:
   Calcul.SommeCarre(Int32 x, Int32 y) ligne 20
   CalculTests.TestSommeCarre_2_3_13() ligne 37
```


Explication

- La source de l'erreur est :
- l'interface n'a pas été instanciée,
- l'appel d'une méthode (`Carre(x)`) non implémentée.

MSTest

Solution

On peut utiliser les mocks pour créer un objet factice de `CalculService`.

MSTest

Commençons par importer le namespace relatif à `Moq` dans la classe `CalculTests`

```
using Moq;
```

MSTest

Commençons par importer le namespace relatif à `Moq` dans la classe `CalculTests`

```
using Moq;
```

Créons un mock de `ICalculService`

```
ICalculService calculService = Mock.Of<ICalculService>();
```

MSTest

Utilisons `Setup ... Returns` pour indiquer ce qu'il faut retourner lorsque la méthode `Carre` est appelée avec les paramètres 2 ou 3

```
[TestMethod]
public void TestSommeCarre_2_3_13()
{
    Mock.Get(calculService).Setup(m => m.Carre(2)).Returns(4);
    Mock.Get(calculService).Setup(m => m.Carre(3)).Returns(9);
    calcul.SommeCarre(2, 3).Should().Be(13, "2*2 + 3*3 = 13");
}
```

MSTest

Utilisons `Setup ... Returns` pour indiquer ce qu'il faut retourner lorsque la méthode `Carre` est appelée avec les paramètres 2 ou 3

```
[TestMethod]
public void TestSommeCarre_2_3_13()
{
    Mock.Get(calculService).Setup(m => m.Carre(2)).Returns(4);
    Mock.Get(calculService).Setup(m => m.Carre(3)).Returns(9);
    calcul.SommeCarre(2, 3).Should().Be(13, "2*2 + 3*3 = 13");
}
```

Lancez le test et vérifiez qu'il se termine correctement.

MSTest

Nous pouvons également factoriser l'écriture précédente en utilisant `SetupSequence ... Returns` **et** `It.IsAny<int>()`

```
[TestMethod]
public void TestSommeCarre_2_3_13()
{
    Mock.Get(calculService).SetupSequence(m => m.Carre(It.IsAny<int>())
       )
                .Returns(4)
                .Returns(9);
    calcul.SommeCarre(2, 3).Should().Be(13, "2*2 + 3*3 = 13");
}
```

MSTest

Nous pouvons également factoriser l'écriture précédente en utilisant `SetupSequence ... Returns` **et** `It.IsAny<int>()`

```
[TestMethod]
public void TestSommeCarre_2_3_13()
{
    Mock.Get(calculService).SetupSequence(m => m.Carre(It.IsAny<int>())
       )
                .Returns(4)
                .Returns(9);
    calcul.SommeCarre(2, 3).Should().Be(13, "2*2 + 3*3 = 13");
}
```

Lancez le test et vérifiez qu'il se termine correctement.

Pour vérifier que le mock a bien été appelé, on peut utiliser la méthode `verify`

```
[TestMethod]
public void TestSommeCarre_2_3_13()
{
    Mock.Get(calculService).SetupSequence(m => m.Carre(It.IsAny<int>())
       )
                .Returns(4)
                .Returns(9);
    calcul.SommeCarre(2, 3).Should().Be(13, "2*2 + 3*3 = 13");
    Mock.Get(calculService).Verify(m => m.Carre(2));
    Mock.Get(calculService).Verify(m => m.Carre(3));
}
```

Pour vérifier que le mock a bien été appelé, on peut utiliser la méthode `verify`

```
[TestMethod]
public void TestSommeCarre_2_3_13()
{
    Mock.Get(calculService).SetupSequence(m => m.Carre(It.IsAny<int>())
       )
                .Returns(4)
                .Returns(9);
    calcul.SommeCarre(2, 3).Should().Be(13, "2*2 + 3*3 = 13");
    Mock.Get(calculService).Verify(m => m.Carre(2));
    Mock.Get(calculService).Verify(m => m.Carre(3));
}
```


Remarque

Remplacez `Mock.Get(calculService).Verify(m => m.Carre(2))` par `Mock.Get(calculService).Verify(m => m.Carre(4))` et vérifiez que le test échoue.

MSTest

Pour vérifier que `carre(2)` a été appelée une seule fois, on peut utiliser la méthode `times`

```
[TestMethod]
public void TestSommeCarre_2_3_13()
{
   Mock.Get(calculService).SetupSequence(m => m.Carre(It.IsAny<int>()))
                .Returns(4)
                .Returns(9);
   calcul.SommeCarre(2, 3).Should().Be(13, "2*2 + 3*3 = 13");
   Mock.Get(calculService).Verify(m => m.Carre(2), Times.Once());
   Mock.Get(calculService).Verify(m => m.Carre(3), Times.Once());
}
```

MSTest

Pour vérifier que `carre(2)` a été appelée une seule fois, on peut utiliser la méthode `times`

```
[TestMethod]
public void TestSommeCarre_2_3_13()
{
   Mock.Get(calculService).SetupSequence(m => m.Carre(It.IsAny<int>()))
                .Returns(4)
                .Returns(9);
   calcul.SommeCarre(2, 3).Should().Be(13, "2*2 + 3*3 = 13");
   Mock.Get(calculService).Verify(m => m.Carre(2), Times.Once());
   Mock.Get(calculService).Verify(m => m.Carre(3), Times.Once());
}
```


Remarque

Relancer le test et vérifier que le test passe correctement.

MSTest

Autres méthodes de `Times`

- `AtLeastOnce()` : au moins une fois.
- `AtLeast(n)` : au moins `n` fois.
- `AtMost(n)` : au plus `n` fois.
- `AtMostOnce()` : au plus une fois.
- `Exactly(n)` : exactement `n` fois.
- `Never()` : jamais.
- `Between(n, m)` : entre `n` et `m` fois.
- ...

MSTest

Pour vérifier que `Carre` a été appelée deux fois avec des paramètres de type `int`, on peut utiliser `It.IsAny<int>()`

```
[TestMethod]
public void TestSommeCarre_2_3_13()
{
   Mock.Get(calculService).SetupSequence(m => m.Carre(It.IsAny<int>()))
               .Returns(4)
               .Returns(9);
   calcul.SommeCarre(2, 3).Should().Be(13, "2*2 + 3*3 = 13");
   Mock.Get(calculService).Verify(m => m.Carre(It.IsAny<int>()), Times.
      Exactly(2));
}
```

MSTest

Pour vérifier que `Carre` a été appelée deux fois avec des paramètres de type `int`, on peut utiliser `It.IsAny<int>()`

```
[TestMethod]
public void TestSommeCarre_2_3_13()
{
   Mock.Get(calculService).SetupSequence(m => m.Carre(It.IsAny<int>()))
                .Returns(4)
                .Returns(9);
   calcul.SommeCarre(2, 3).Should().Be(13, "2*2 + 3*3 = 13");
   Mock.Get(calculService).Verify(m => m.Carre(It.IsAny<int>()), Times.
      Exactly(2));
}
```


Remarque

Remplacer `Times.Exactly(2)` par `Times.Exactly(3)` et vérifier que le test échoue.

MSTest

Pour vérifier que `carre` a été appelée deux fois avec des paramètres appartenant à un certain intervalle, on peut utiliser `It.IsInRange()`

```
[TestMethod]
public void TestSommeCarre_2_3_13()
{
   Mock.Get(calculService).SetupSequence(m => m.Carre(It.IsInRange(2,
      3, Range.Inclusive)))
                  .Returns(4)
                  .Returns(9);
   calcul.SommeCarre(2, 3).Should().Be(13, "2*2 + 3*3 = 13");
   Mock.Get(calculService).Verify(m => m.Carre(It.IsInRange(2, 3, Range
      .Inclusive)), Times.Exactly(2));
}
```

MSTest

Pour vérifier que `carre` a été appelée deux fois avec des paramètres appartenant à un certain intervalle, on peut utiliser `It.IsInRange()`

```
[TestMethod]
public void TestSommeCarre_2_3_13()
{
   Mock.Get(calculService).SetupSequence(m => m.Carre(It.IsInRange(2,
      3, Range.Inclusive)))
                  .Returns(4)
                  .Returns(9);
   calcul.SommeCarre(2, 3).Should().Be(13, "2*2 + 3*3 = 13");
   Mock.Get(calculService).Verify(m => m.Carre(It.IsInRange(2, 3, Range
      .Inclusive)), Times.Exactly(2));
}
```


Remarque

`Range.Inclusive` veut dire les bornes d'intervalle incluses. Sinon, on peut utiliser `Range.Exclusive`.

MSTest

Autres méthodes similaires

- `anyByte()`
- `anyShort()`
- `anyChar()`
- `anyFloat()`
- `anyDouble()`
- ...

MSTest

Utilisons `When` pour retourner une valeur si une condition est respectée

```
[DataTestMethod]
[DataRow(2, 3, 13)]
[DataRow(2, 2, 8)]
public void TestSommeCarre_NombresPositifs_NombrePositif(int a, int b,
   int c)
{
    Mock.Get(calculService).When(() => a > 0)
                           .Setup(m => m.Carre(a))
                           .Returns(a * a);
    Mock.Get(calculService).When(() => b > 0)
                           .Setup(m => m.Carre(b))
                           .Returns(b * b);
    calcul.SommeCarre(a, b).Should().Be(c, "2*2 + 3*3 = 13");
}
```

MSTest

Utilisons `When` pour retourner une valeur si une condition est respectée

```
[DataTestMethod]
[DataRow(2, 3, 13)]
[DataRow(2, 2, 8)]
public void TestSommeCarre_NombresPositifs_NombrePositif(int a, int b,
   int c)
{
    Mock.Get(calculService).When(() => a > 0)
                           .Setup(m => m.Carre(a))
                           .Returns(a * a);
    Mock.Get(calculService).When(() => b > 0)
                           .Setup(m => m.Carre(b))
                           .Returns(b * b);
    calcul.SommeCarre(a, b).Should().Be(c, "2*2 + 3*3 = 13");
}
```

Lancez le test et vérifiez qu'il se termine correctement.

MSTest

Ajoutons une méthode `Racine` dans l'interface `ICalculService`

```
namespace ProjetTest
{
    public interface ICalculService
    {
        int Carre(int x);
        double Racine(double x);
    }
}
```

MSTest

Ajoutons une méthode `Racine` dans l'interface `ICalculService`

```
namespace ProjetTest
{
    public interface ICalculService
    {
        int Carre(int x);
        double Racine(double x);
    }
}
```


Ajoutons une méthode `RacineSomme` dans la classe `Calcul`

```
public double RacineSomme(double x, double y)
{
    return calculService.Racine(x + y);
}
```

MSTest

Utilisons `SetUp ... Throws` pour lever une exception si une valeur est présente

```
[TestMethod]
public void TestRacineSomme_NombresNegatifs_Exception()
{
    Mock.Get(calculService).Setup(m => m.Racine(-5)).Throws(new
        InvalidOperationException());
    Exception e = null;
    try
    {
        calcul.RacineSomme(-2, -3);
    }
    catch(Exception ex)
    {
        e =ex;
    }
    e.Should().BeOfType<InvalidOperationException>();
    Mock.Get(calculService).Verify(m => m.Racine(-5));
}
```