TDD

Les Tests Unitaires en C#

Antoine Dieudonné

UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Qu'est-ce que le TDD ?

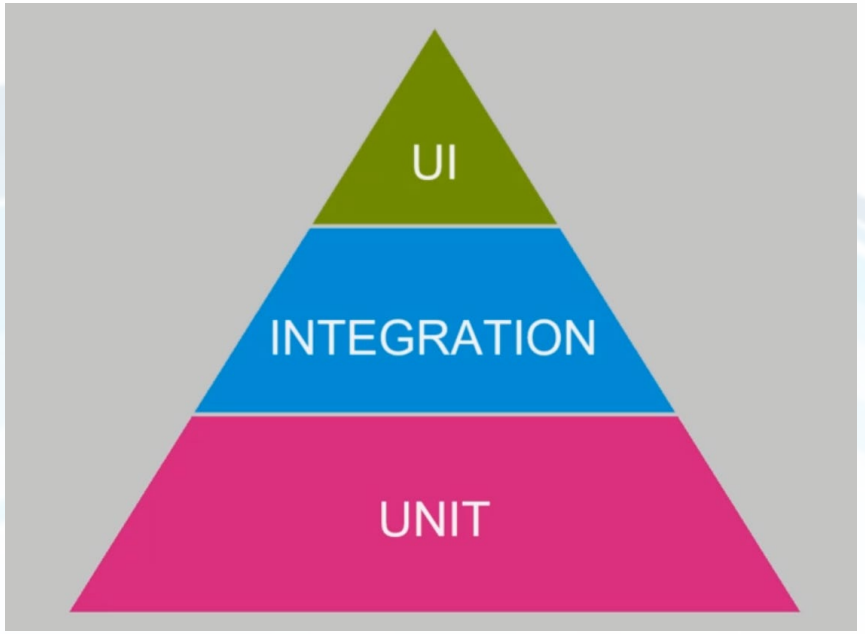
Le **Test Driven Development** est une façon de sécuriser la production d'un projet en réaliser plusieurs familles de tests au cours du développement d'un projet. Les tests les plus faciles et les plus répétés sont en règle générale les Tests unitaires, qui peuvent en C# être réaliser avec plusieurs frameworks spécialisées qui sont par exemple :

- ***MSTest***
- ***NUnit***
- ***xUnit***

L'intérêts des tests unitaires est de permettre la vérification de l'implémentation des fonctionnalités en fonction de leur objectif. En effet, prenons le cas d'une classe visant à faire une simulation de jeu de bowling. Dans une classe de ce genre, il faut par exemple vérifier qu'il n'est pas possible de faire tomber plus de 10 quilles par round, qu'un round dans lequel le joueur fait tomber les 10 quilles du premier coup correspond à un strike, etc…

Pour réaliser des tests unitaires, il faut créer des méthodes de tests, qui sont par convention nommées
NomDeMethode_ValeursEnvoyéesALaMéthode_ResultatAttendu()
de sorte à permettre une lecture rapide de tous les tests.

Les différentes familles de tests sont répartis selon la pyramide des tests, qui permet de facilement observer l'importance des tests et la fréquence de leur réalisation (les tests unitaires sont plus accessibles car moins chers et plus rapides)



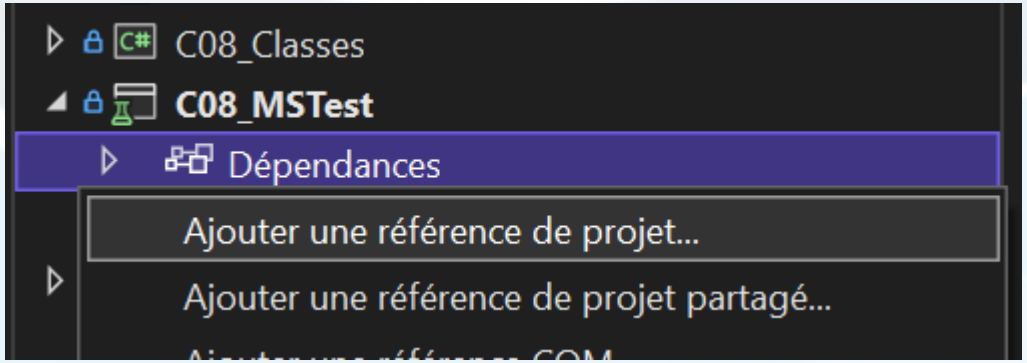
La nécessité des tests

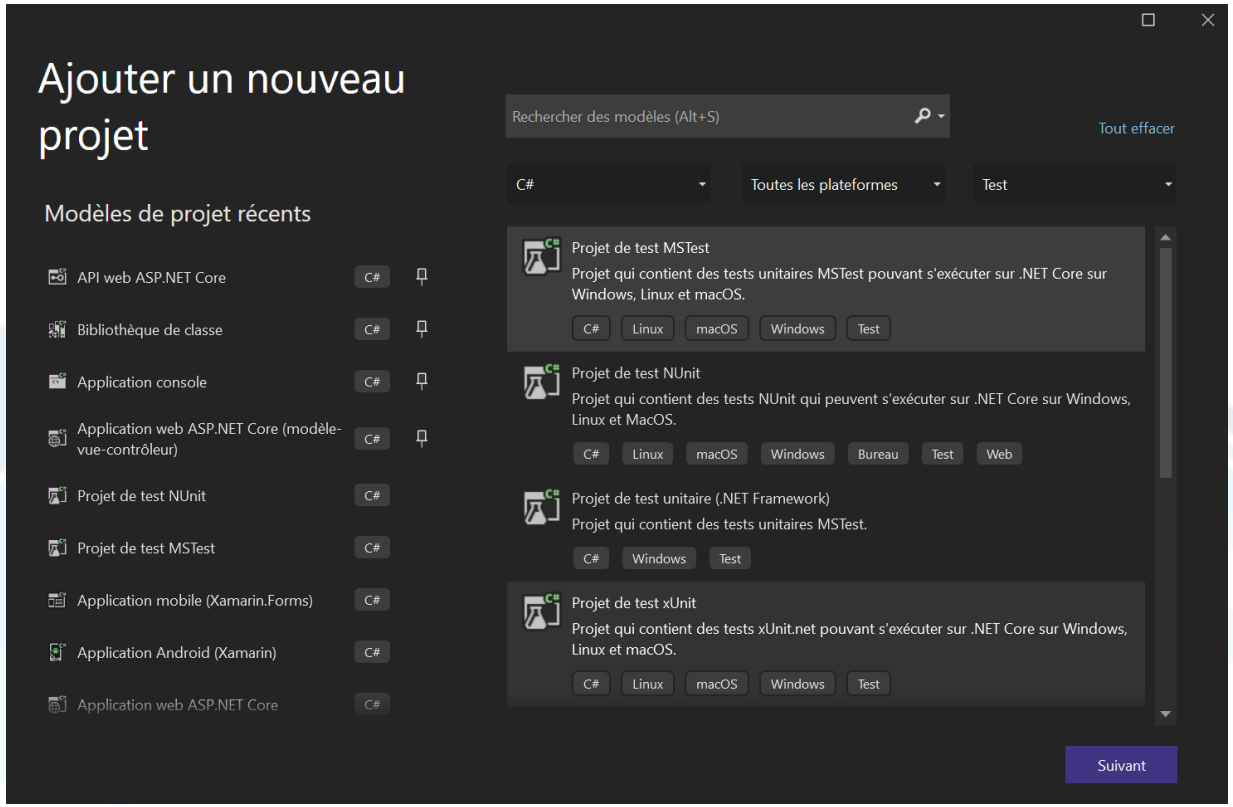
Un projet de tests unitaires se fait en parallèle du projet de développement. Par exemple on aura un projet qui se nommerai **JeuDuPendu** et un autre qui se nommerai **JeuDuPendu.Tests**

Les tests unitaires pouvant être réalisés plusieurs fois et de façon répétées, il sera ainsi facile d'implémenter les tests en amont de la création des classes, de sorte à confirmer qu'elles fonctionnent tel que l'on le souhaitait. De plus, lors de la modification future de l'une de ces classes, on pourra, via les tests, être ainsi certain que la classe répond toujours aux demandes en voyant qu'elle passe tous les tests malgré son évolution.

Pour réaliser un projet de test, il suffit de créer un nouveau projet de type Bibliothèque de tests, par exemple MSTest :

Une fois le projet créé, il ne faudra pas oublier d'ajouter une référence de projet entre ce nouveau projet et celui que l'on cherche à tester.





MSTest

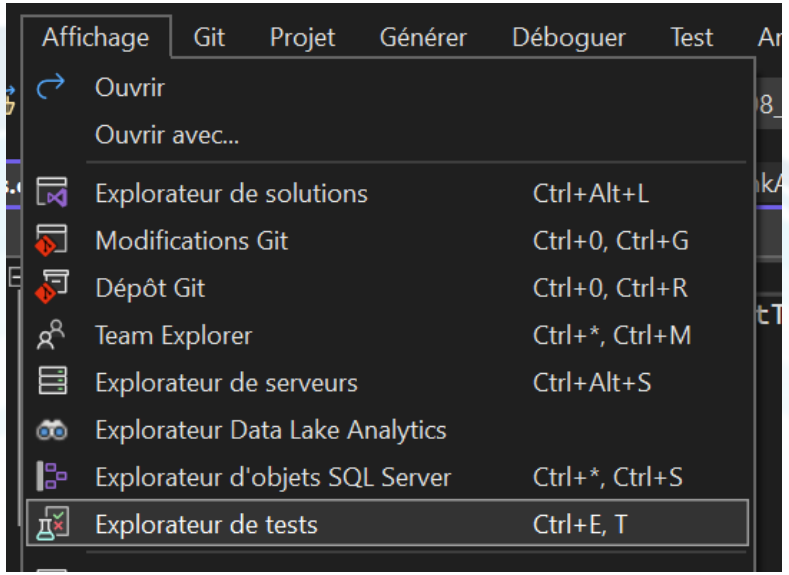
MSTest est le framework de test de base fourni dans Visual Studio. Il est cependant peu utilisé, car on lui préfère les framework **NUnit** et **xUnit**, qui sont plus performants et plus faciles d'utilisation.

Dans MSTest, la création d'une classe de test se fait par la simple création d'une classe que l'on précède d'une annotation [**TestClass**] et dont les méthodes publiques seront précédées d'une annotation [**TestMethod**].

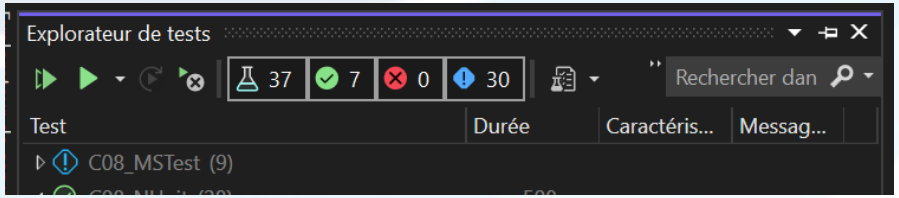
La réalisations des tests de fait par la succession de trois phases :

- **Arrange**, dans laquelle on planifie les classes et les résultats voulus
- **Act**, dans laquelle on fait appel à la méthode que l'on souhaite tester
- **Assert**, dans laquelle on vérifie le résultat de la méthode dans le but de valider le test

Lorsque l'on réalise un **TDD**, il faut être sur que **tous les tests soient échoués avant l'implémentation des méthodes**, de sorte à éviter les faux positifs.
Une fois l'implémentation des méthodes effectuées, on doit ainsi voir s'allumer petit à petit les feux verts signalant l'accomplissement de tous les test.



Pour voir les tests et les lancer, il faut ouvrir la fenêtre des tests :



La classe Assert

Assert est une classe qui possède de nombreuses méthodes statiques utilisées dans le cadre des tests unitaires. Un test unitaire devra toujours posséder une assertion. Il est également possible de réaliser une succession d'Assertions, mais attention, car le test peut en devenir confus, et il est préférable de réaliser ces assertions dans des méthodes spécialisées. De plus, MSTest, à contrario de NUnit, ne permet pas de voir le détail de l'échec des tests, et stoppera les assertions à la première ayant échouée.

Il est par exemple possible de vérifier que le résultat d'une méthode est égal à une valeur saisie par le développeur via l'utilisation de la méthode ***Assert.AreEqual()***, de vérifier que le résultat est vrai ou faux via l'utilisation des méthodes ***Assert.IsTrue()*** et ***Assert.IsFalse()***

```
public void ExecuteOperation_InputTwoIntege
{
    // Arrange
    Calculator calculator = new();

    // Act
    double result = calculator.ExecuteOpera

    // Assert
    Assert.AreEqual(210, result);

}
```

```
[TestMethod]
0 références
public void IsOddNumber_InputEvenNumber_GetFalse()
{
    // Arrange
    Calculator calculator = new();

    // Act
    bool result = calculator.IsOddNumber(8);

    // Assert
    Assert.IsFalse(result);
}
```

NUnit

NUnit est le framework de tests que l'on utilisera car il permet plus de fonctionnalités que MSTest, et permet dans de nombreux cas de ne pas avoir à passer autant de temps dans la phase d'arrangement des classes et d'éviter une répétition inutile du code. NUnit permet ainsi d'annoter une méthode de [**SetUp**] pour spécifier un point commun entre tous les tests.

```
private Calculator _calculator;

[SetUp]
0 références
public void Setup()
{
    _calculator = new Calculator();
}

[Test]
| 0 références
public void AddNumbers_InputTwoIntegers_GetCorrectAddition()
{

    // Act
    int result = _calculator.AddNumbers(5, 7);

    // Assert
    Assert.AreEqual(12, result);

}
```

De plus, NUnit permet d'ajouter des annotations supplémentaires des méthodes, comme par exemple [**TestCase**], qui permet de réaliser plusieurs configurations des paramètres de la méthode pour éviter d'avoir à la réécrire à chaque fois comme on aurait à le faire dans MSTest.

Les paramètres du test doivent ainsi être passés à la méthode de test, puis utilisés en tant que **variables** dans les méthodes testés.

```
[Test]
[TestCase(12)]
[TestCase(14)]

public void IsOddNumber_InputEvenNumber_GetFalse(int nbA)
{
```

```
// Act
bool result = _calculator.IsOddNumber(nbA);
```

Il est également possible de spécifier le retour attendu à ces tests, en modifiant les méthodes et en ajoutant un élément supplémentaire au **TestCase** tout en retournant le résultat que l'on veut vérifier

```
[TestCase(12, ExpectedResult = false)]
| 0 références
public bool IsOddNumber_InputNumber_GetTrueIfOdd(int nbA)
{
```

Extention d'Assert

NUnit ajoute également de nouvelles fonctionnalités à Assert, comme par exemple ***Assert.That()***, qui permet de nombreuses fonctionalités en plus de ce que proposait de base la classe via l'utilisation du framework MSTest.

```
    int result = _customer.Discount;
    Assert.That(result, Is.InRange(10, 25));
}
```

Lors de l'utilisation de Assert.That(), on commence par mettre en premier paramètre le résultat, puis on en spécifie les contraintes par utilisation de classes comme par exemple **Is**, **Does**, **Has**, etc… ce qui revient à écrire quasiment en anglais ce que l'on veut obtenir comme résultat de la méthode.

```
// Assert
Assert.That(result, Is.TypeOf<BasicCustomer>());
```

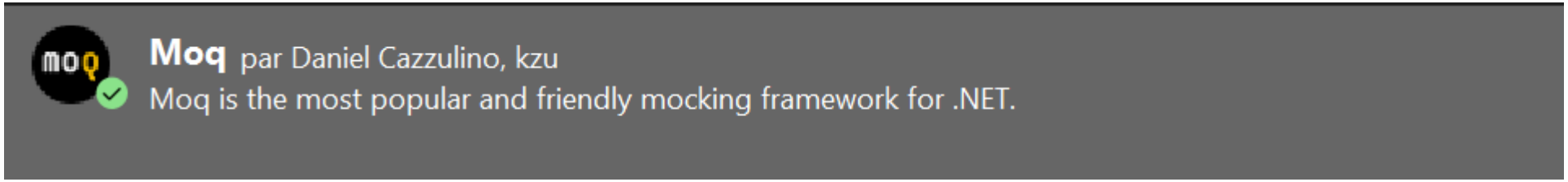
De plus, l'assertion multiple est possible lorsque l'on se sert de NUnit, via l'utilisation de la méthode ***Assert.Multiple()***, dans laquelle on passe une fonction fléchée contenant les assertions que l'on souhaite réaliser. A contrario de MSTest, les assertions multiples réalisées via cette méthode ne seront pas bloquantes, et peuvent être déroulées pour voir le détails de quelle assertion a échoué et lesquelles ont réussi.

```
// Assert
Assert.Multiple(() =>
{
    Assert.AreEqual(_customer.GreetMessage, "Hello, John SMITH");
    Assert.That(_customer.GreetMessage, Is.EqualTo("Hello, John SMITH"));
    Assert.That(_customer.GreetMessage, Does.Contain("S"));
```

Le Mocking

Le **Mocking** est un procédé utilise énormément dans les tests unitaires, car il permet d'éviter la transformation d'un test unitaire en tant que **test d'intégration**. En effet, bien souvent, les classes de notre programme interagissent entre elles et doivent attendre les résultats de méthodes d'autres classes pour fonctionner. Ceci est en particulier vrai dans les classes ayant recours à des appels API ou à des récupérations de données en base de données.

Pour réaliser du Mocking, il faut se servir d'un package NuGet. Il en existe plusieurs, mais le plus utilisé est sans doute **Moq**.



Moq permet ainsi de fabriquer des faux-semblant de classes implémentant des **interfaces** dans le but de leur fixer un **fonctionnement** et un **retour**. Dans le cadre d'un test unitaire, il faut par exemple tester le fonctionnement d'une classe faisant une modification de données en BdD, mais bien évidemment **éviter de réaliser réellement cette modification**.

```
var logMock = new Mock<ILogger>();
logMock.Setup(x => x.LogToDb(It.IsAny<string>())).Returns(true);
logMock.Setup(x => x.LogBalanceAfterWithdrawal(It.Is<decimal>(x => x > 0))).Returns(true);
_bankAccount = new BankAccount(logMock.Object);
```

Pour ce faire, on passe par exemple par l'interface de cette classe, et l'on **fixe le résultat** comme étant un résultat convenable de sorte à vérifier que la classe que l'on cherche à tester fonctionne correctement dans le cadre où la modification est réalisée.

Utiliser Moq

Pour utiliser Moq, il faut d'abord créer le Mock, c'est-à-dire instancier un Mock d'interface (ou de classe) via l'utilisation du constructeur de Mock, qui est de type générique.

```
var logMock = new Mock<ILogger>();
_bankAccount = new BankAccount(logMock.Object);
```

Une fois le mock créé, on peut le passer à l'objet (par exemple un objet qui demande une injection de dépendance comme c'est le cas des applications **ASP.NET**).

```
logMock.Setup(x => x.LogToDb(It.IsAny<string>())).Returns(true);
logMock.Setup(x => x.LogBalanceAfterWithdrawal(It.Is<decimal>(x => x > 0))).Returns(true);
```

Une fois l'objet ajouté à l'injection, on peut en fixer le fonctionnement via l'utilisation de la méthode ***Setup()***, puis le retour via la méthode ***Returns()***

Au sein de la méthode **Setup** ou de la méthode **Returns**, on peut spécifier des contraintes, comme par exemple que le paramètre est un ***Int32*** de n'importe quelle valeur positive, ou que le **retour est le même que le paramètre envoyé à la méthode**.

```
logMock.Setup(x => x.MessageWithReturnStr(It.IsAny<string>())).Returns((string str) => str.ToLower());
```