Préambule

Les tests unitaires (de classe et/ou de méthode...) permettent de vérifier que la classe et/ou la méthode sont conformes aux spécifications.

Dans la méthode de développement TDD [1], les tests unitaires sont écrits **avant** de coder les classes.

Les environnements de développement (*Visual Studio* par exemple) offrent des générateurs (partiels) de code de tests unitaires.

1 Tests unitaires de la classe `Fraction`

1.1 Introduction

La classe `Fraction` représente une fraction avec les différents constructeurs et méthodes permettant :

- de construire une fraction avec 0, 1, 2 paramètres
- de réduire automatiquement la fraction
- d'afficher la fraction (sous forme de chaîne, de nombre réel)
- d'effectuer des calculs entre fractions

La classe `Fraction` fournira les propriétés, constructeurs et méthodes du diagramme de la figure 1.

1. *Test Driven Development*

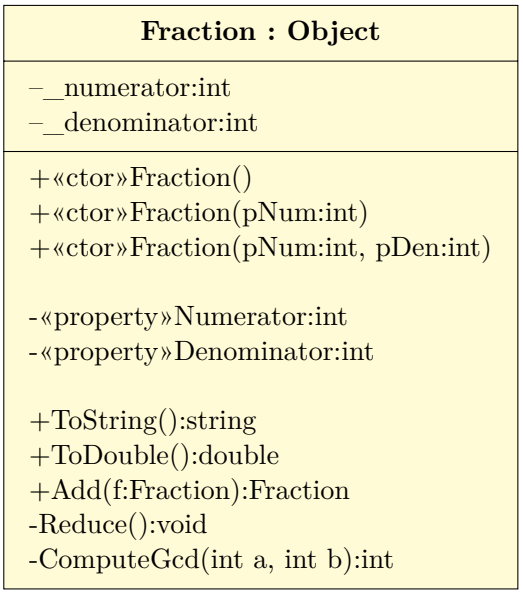

FIGURE 1 – Diagramme de la classe `Fraction`

1.2 Génération des tests unitaires

Remarque : il n'est possible de générer des tests unitaires **QUE** pour les classes `public`.

- clic-droit sur le nom de la classe à tester, « *Créer des tests unitaires* »
- conservez les valeurs par défaut
- OK
- un nouveau projet a été créé, il contient un fichier `FractionTests.cs`, voir listing 1

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Ex3Fraction.Tests {
  [TestClass()]
  public class FractionTests {
    [TestMethod()]
    public void FractionTest() {
      Assert.Fail();
    }

    [TestMethod()]
    public void FractionTest1() {
      Assert.Fail();
    }

 ...

    [TestMethod()]
    public void ToStringTest() {
      Assert.Fail();
    }
  }
}
```

Listing 1 – `FractionTests.cs`

1.3 Structure d'un test unitaire

Un test unitaire est composé de 3 parties :

1.3.1 Organiser (*Arrange*)

Création de l'objet à tester.

1.3.2 Agir (*Act*)

Appel d'une méthode ou modification d'un champ.

1.3.3 Vérifier (*Assert*)

Vérification du résultat.

1.3.4 Exemple

```
[TestMethod()]
public void ToStringTest() {
  // Arrange
  Fraction target = new Fraction(2, 3);

  // Act
  string answer = target.ToString();

  // Assert
  Assert.AreEqual("2 / 3", answer);
}
```

Listing 2 – Extrait de `FractionTests.cs`

1.4 Exécution des tests unitaires

- Menu *Test, Fenêtres, Explorateur de tests*
- dans la fenêtre « *Explorateur de tests* » :
  - cliquez sur le bouton « *Exécuter tout* »
  - la fenêtre « *Explorateur de tests* » visualise les résultats
  - la classe `Fraction` étant vide, certains tests devraient échouer ☹ mais d'autres (paradoxalement) devraient réussir ☺

2 Travail à effectuer

Écrivez les tests unitaires de la méthode `int ComputeGcd(int a, int b)` avec les données suivantes :

- $pgcd(0, 0) = 0$
- $pgcd(0, 1) = 1$
- $pgcd(0, 17) = 17$
- $pgcd(1, 1) = 1$
- $pgcd(10, 10) = 10$
- $pgcd(\texttt{Int32.MaxValue}, \texttt{Int32.MaxValue}) = \texttt{Int32.MaxValue}$
- $pgcd(\texttt{Int32.MaxValue}, 127) = 1$
- $pgcd(1000000007, 1000000009) = 1$
- $pgcd(-10, 10) = 10$
- $pgcd(-10, -10) = 10$
- $pgcd(23, 101) = 1$
- $pgcd(456, 456 * 123) = 456$

Indications : `https://fr.wikiversity.org/wiki/Arithmétique/PGCD#Extension_du_PGCD_aux_entiers_relatifs`

3 Compléments sur les tests unitaires

3.1 `Assert.AreEqual()`

3.1.1 Comparaison de variables de type *valeur*

Dans le cas de variable de type *valeur*, `Assert.AreEqual()` est équivalent à `==`.

Exemple :

```
[TestMethod()]
public void PermissionsTest() {
  Permissions target = new Permissions();
  Assert.AreEqual(false, target.Read);
  Assert.AreEqual(false, target.Write);
  Assert.AreEqual(false, target.Execute);
}
```

Les variables comparées sont de type `bool` (type *valeur*), `Assert.AreEqual(false, target.Read)` est strictement équivalent à `(false == target.Read)`.

3.1.2 Comparaison de variables de type *référence*

Dans le cas de variable de type *référence* (objbet, tableau, chaîne, dictionnaire), la traduction de `Assert.AreEqual()` dépend du type *référence* :

- si le type *référence* ne possède pas de redéfinition de la méthode `Equals(object obj)` alors :
  - `Assert.AreEqual(a, b)` est remplacé par `a == b`, cela signifie que les références (les pointeurs) sont comparées et que le résultat est vrai (`true`) **si** `a` et `b` pointe sur le **même** objet.
- si le type *référence* possède une redéfinition de la méthode `Equals(object obj)` alors :
  - `Assert.AreEqual(a, b)` est remplacé par `a.Equals(b)`, cela signifie que les contenus des objets `a` et `b` sont comparés (à l'aide de la méthode `Equals()`) et que le résultat est vrai (`true`) **si** les contenus sont identiques (même si `a` et `b` pointent sur 2 objets différents).

Exemple :

```
[TestMethod()]
public void FilePermissionsConstructorTest2() {
  FilePermissions target = new FilePermissions();
  Permissions expected = new Permissions();

  Assert.AreEqual(expected, target.User);
  Assert.AreEqual(expected, target.Group);
  Assert.AreEqual(expected, target.Other);
}
```

```
public class Permissions : Object {
  public bool Read { get; set; }
  public bool Write { get; set; }
  public bool Execute { get; set; }

  public override bool Equals(object obj) {
    Permissions p = obj as Permissions;

```

```
    // caution : use of boolean short-circuit evaluation in order ↩
       to avoir null object access
    return (p != null) &&
           (this.Read == p.Read) &&
           (this.Write == p.Write) &&
           (this.Execute == p.Execute);
  }
}
```

Les variables comparées sont de type `Permissions` (type *référence*) et la classe `Permissions` possède une méthode `Equals()`.
`Assert.AreEqual(expected, target.User)` est strictement équivalent à `expected.Equals(target.User)`.

3.1.3 Comparaison de variables de type *référence* : tableau

À compléter : `http://stackoverflow.com/questions/24464452/comparing-two-arrays-in-unit-test-throwing-a-assertfailedexception`

```
You are comparing different instances of an int[]. Assert.AreEqual compares by re

CollectionAssert.AreEqual(expectedPosition, testPlayer.Position);
```

3.2 Ignorer un test

Le mot-clé `Ignore` dans l'attribut (de la classe de test ou de la méthode) permet d'éviter l'exécution du test unitaire mais conserve le nom (grisé) dans la liste des tests.

```

[Ignore, TestMethod()]
public void PermissionsConstructorTest1() {
  for (int octalValue = -16; octalValue < 16; octalValue++)
  {
    Permissions target = new Permissions(octalValue);
    Assert.AreEqual(octalValue & 0x07, target.ToOctal());
  }
}
```

Listing 3 – Ignorer un test unitaire avec le mot-clé `Ignore`

3.3 Accéder à des champs privés

Les tests unitaires ont parfois besoin d'accéder (depuis l'extérieur de la classe à tester) à des champs privés (constantes, variables...). La DLL `Microsoft.VisualStudio.TestTools.UnitTesting` offre des fonctions permettant d'accéder aux champs privés.

```
class StackV3 {
  private const int STACK_SIZE = 1000;
  private int _stackPointer;
  private Object[] _data;

  public StackV3() { }
  public void PushMany(Object[] obj) { }
```

Listing 4 – Extraits de la classe `StackV3`

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

public void PushManyUntilStackFullTest() {
  StackV3 target = new StackV3();

  PrivateType privateType = new PrivateType(target.GetType());
  Object stackMaxSize = ↩
     privateType.GetStaticFieldOrProperty("STACK_SIZE");
  Object[] articles = new Object[Convert.ToInt16(stackMaxSize)];

  target.PushMany(articles);
}
```

Listing 5 – Accès à la constante privée `STACK_SIZE`

3.4 Accéder à un champ privé

`https://stackoverflow.com/questions/10789644/testing-a-private-field-using-mstest`

Ex : lecture du champ privé `_numerator` de la classe `Fraction` :

```
Fraction target = new FractionClass();
PrivateObject obj = new PrivateObject(target);

Assert.AreEqual(3, (int)obj.GetField("_numerator"));
Assert.AreEqual(8, (int)obj.GetField("_denominator"));
```

3.5 Tester une méthode privée

`https://stackoverflow.com/questions/9122708/unit-testing-private-methods-in-c-sharp`

Ex : test unitaire de méthode `private int ComputeGcd(int a, int b)` :

```
Fraction target = new FractionClass();
PrivateObject obj = new PrivateObject(target);

Assert.AreEqual(5,
                obj.Invoke("ComputeGcd", new object[] { 5, 15 })
               );
```

3.6 Détecter une exception

```
[TestMethod()]
[ExpectedException(typeof(StackEmpty), "StackEmpty.")]
public void PopEmptyStackTest()
{
  StackV3 target = new StackV3();
  target.Pop();
}
```

Listing 6 – Détecter une exception

```
class StackEmpty : Exception {
  public StackEmpty() : base("StackEmpty !") { }
}

class StackV3 {
  const int STACK_SIZE = 1000;
...
  public StackV3() { }
  public void Pop() { }
```

Listing 7 – Extraits de la classe `StackV3`

3.7 *Setup* et *TearDown*

Factorisation du code d'initialisation (*setup*, *initialize*) et de nettoyage (*teardown*, *cleanup*)

```
// Utilisez ClassInitialize pour exécuter du code avant ↩
    d'exécuter le premier test dans la classe
[ClassInitialize()]
public static void MyClassInitialize(TestContext testContext) {
}

// Utilisez ClassCleanup pour exécuter du code après que tous les ↩
    tests ont été exécutés dans une classe
[ClassCleanup()]
public static void MyClassCleanup() {
}

// Utilisez TestInitialize pour exécuter du code avant d'exécuter ↩
    chaque test
[TestInitialize()]
public void MyTestInitialize() {
}

// Utilisez TestCleanup pour exécuter du code après que chaque ↩
    test a été exécuté
[TestCleanup()]
public void MyTestCleanup() {
}
```

Listing 8 – *Setup* et *TearDown*

3.8 Tests unitaires paramétrables

Avec *Visual Studio 2017*, il est possible de paramétrer les tests unitaires afin d'utiliser une méthode de test avec différents jeux de paramètres.

```
using Microsoft.VisualStudio.TestTools.UnitTesting

[TestMethod()]
[DataRow(2, 3, "2 / 3")]
[DataRow(8, 16, "1 / 2")]
[DataRow(33, 11, "3")]
public void ToStringTest(int num, int den, string ↩
    expectedFractionString) {
  // Arrange
  Fraction target = new Fraction(num, den);

  // Act
  string answer = target.ToString();

```

```
14    // Assert
15    Assert.AreEqual(expectedFractionString, answer);
16 }
```

Listing 9 – Extrait de `FractionTests.cs`

4 Références

— https://docs.microsoft.com/fr-fr/visualstudio/test/unit-test-basics
— http://c2.com/cgi/wiki?ArrangeActAssert
— https://msdn.microsoft.com/fr-fr/library/hh694602.aspx
— https://docs.microsoft.com/fr-fr/visualstudio/test/unit-test-basics

5 FAQ

5.1 Noms des méthodes de test unitaire

Il existe des conventions de nommage des méthodes de test unitaire :

- `https://dzone.com/articles/7-popular-unit-test-naming`
- `stackoverflow.com/questions/155436/unit-test-naming-best-practices`

1. `MethodName_StateUnderTest_ExpectedBehavior` : There are arguments against this strategy that if method names change as part of code refactoring than test name like this should also change or it becomes difficult to comprehend at a later stage.
   - `isAdult_AgeLessThan18_False`
   - `withdrawMoney_InvalidAccount_ExceptionThrown`
   - `admitStudent_MissingMandatoryFields_FailToAdmit`
2. `MethodName_ExpectedBehavior_StateUnderTest` : Slightly tweeked from above, but a section of developers also recommend using this naming technique. This technique also has disadvantage that if method names get changed, it becomes difficult to comprehend at a later stage.
   - `isAdult_False_AgeLessThan18`
   - `withdrawMoney_ThrowsException_IfAccountIsInvalid`
   - `admitStudent_FailToAdmit_IfMandatoryFieldsAreMissing`
3. `test[Feature being tested]` : This one makes it easy to read the test as the feature to be tested is written as part of test name. Although, there are arguments that the « test » prefix is redundant. However, some sections of developer love to use this technique.
   - `testIsNotAnAdultIfAgeLessThan18`
   - `testFailToWithdrawMoneyIfAccountIsInvalid`
   - `testStudentIsNotAdmittedIfMandatoryFieldsAreMissing`
4. `Feature to be tested` : Many suggests that it is better to simply write the feature to be tested because one is anyway using annotations to identify method as test methods. It is also recommended for the reason that it makes unit tests as alternate form of documentation and avoid code smells.
   - `IsNotAnAdultIfAgeLessThan18`
   - `FailToWithdrawMoneyIfAccountIsInvalid`
   - `StudentIsNotAdmittedIfMandatoryFieldsAreMissing`
5. `Should_ExpectedBehavior_When_StateUnderTest` : This technique is also used by many as it makes it easy to read the tests.
   - `Should_ThrowException_When_AgeLessThan18`
   - `Should_FailToWithdrawMoney_ForInvalidAccount`
   - `Should_FailToAdmit_IfMandatoryFieldsAreMissing`
6. `When_StateUnderTest_Expect_ExpectedBehavior` :
   - `When_AgeLessThan18_Expect_isAdultAsFalse`
   - `When_InvalidAccount_Expect_WithdrawMoneyToFail`
   - `When_MandatoryFieldsAreMissing_Expect_StudentAdmissionToFail`
7. `Given_Preconditions_When_StateUnderTest_Then_ExpectedBehavior` : This approach is based on naming convention developed as part of *Behavior Driven Development* (BDD). The idea is to break down the tests into three part such that one could come up with preconditions, state under test and expected behavior to be written in above format.

— `Given_UserIsAuthenticated_When_InvalidAccountNumberIsUsed-ToWithdrawMoney_Then_TransactionsWillFail`

Je conseille la convention n°4 (`Feature to be tested`) :

— elle permet de s'assurer que la classe a une seule responsabilité (*Single Responsibility Principle*[2])

— elle rend le nom de la méthode de test indépendante du nom de la méthode à tester (cohérence)

5.2 *Fluent Assertions*

Il existe des paquets `NuGet` permettant de simplifier la rédaction des tests unitaires, principalement la partie `Assert`.

Le paquet « `FluentAssertions` »[3] permet de remplacer la syntaxe classique `Assert.AreNotEqual(...);` par une phrase en anglais `target.Should()...;`

5.2.1 Exemples

Syntaxe classique	Syntaxe *FluentAssertions*
`Assert.AreNotEqual(null, target);`	`target.Should().NotBeNull();`
`Assert.AreEqual(1, target.Throw());`	`target.Throw().Should().Equals(1);`
`Assert.AreEqual(3.14F, value, 0.01F);`	`value.Should().BeApproximately(3.14F, 0.01F);`
`Assert.AreEqual("Hello", target);`	`target.Should().Be("Hello");`
`Assert.IsTrue(target.Contains("lo"));`	`target.Should().Contain("lo");`
`Assert.IsTrue(date > new DateTime(2010, 2, 1));`	`date.Should().BeAfter(1.February(2010));`

Table 1 – Comparaison des syntaxes classique et *FluentAssertions*

2. https://en.wikipedia.org/wiki/Single_responsibility_principle
3. http://fluentassertions.com

A Les 10 commandements des tests unitaires

`http://blog.xebia.fr/2008/04/11/les-10-commandements-des-tests-unitaires/`

1. la classe testée doit l'être en isolation complète ;
2. si une classe est difficile à tester, il est temps de faire du *refactoring* ;
3. un test unitaire doit s'exécuter le plus rapidement possible ;
4. le code d'un test unitaire doit respecter des conventions de code ;
5. isolez les dépendances de la classe testée grâce à l'injection de dépendances ;
6. ne testez qu'un comportement à la fois ;
7. utilisez un *framework* de *mocks* ;
8. identifiez précisément les étapes *setup, arrange, act, assert, teardown* ;
9. ne vous concentrez pas sur une couverture de code à 100 % ;
10. ne développez pas vos tests unitaires « plus tard ». Si vous n'utilisez pas l'approche TDD [4], développez-les le plus tôt possible.

4. *Test Driven Development*