Cyril Seguenot

2016

Résumé

Support d’accompagnement d’une formation à la programmation avec C# faite en présentiel

Tests unitaires

Support de cours

Toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l’auteur ou de ses ayants droit ou ayants cause est illicite selon le Code de la propriété intellectuelle (Art L 122-4) et constitue une contrefaçon réprimée par le Code pénal.

Seules sont autorisées (Art L122-5) les copies ou reproductions strictement réservées à l’usage privé du copiste et non destinées à une utilisation collective, ainsi que les analyses et courtes citations justifiées par le caractère critique, pédagogique ou d’information de l’œuvre à laquelle elles sont incorporées, sous réserve, toutefois, du respect des dispositions des articles L 122-10 à L 122-12 du même Code, relatives à la reproduction par reprographie.

©Cyril Seguenot 2016

SOMMAIRE

[1 Introduction 4](#_Toc502525578)

[2 Création d’une suite de tests dans Visual Studio 4](#_Toc502525579)

[2.1 Création d’un projet de tests 4](#_Toc502525580)

[2.2 Structure d’un projet de tests 6](#_Toc502525581)

[2.3 Ecriture de tests 9](#_Toc502525582)

[2.4 Bonnes pratiques 10](#_Toc502525583)

[3 Exécution de tests unitaires 10](#_Toc502525584)

[4 Références bibliographiques 10](#_Toc502525585)

# Introduction

Le test d’une application est une étape essentielle pour s’assurer de sa fiabilité avant sa mise en production.

Les tests sont généralement réalisés au fur et à mesure des développements, pour ne pas avoir à tester l’intégralité d’une version de l’application juste avant sa sortie. Dans une approche itérative, les tests font partie de chaque itération.

Les tests peuvent être de plusieurs natures :

* Alphatests réalisés par le développeur au fur et à mesure ou à la fin d’un développement
* Tests réalisés par des testeurs (éventuellement développeurs) sur base d’un plan de test complet écrit au préalable, et portant principalement sur les aspects fonctionnels de l’application
* Tests unitaires mis au point par des développeurs/testeurs, et joués de façon automatique par un système tel que TFS. Ces tests permettent de s’assurer que les résultats produits par l’application sont conformes à un jeu de résultats défini à l’avance (appelé master). Ils sont utiles pour mettre au point le code au départ, et pour s’assurer ensuite de façon récurrente que le code n’a pas subi de régressions suite à d’éventuelles évolutions / corrections.

**Définitions :**

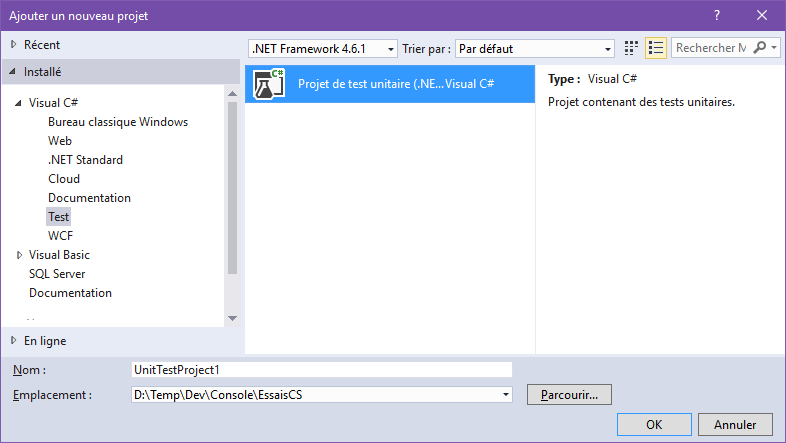
**Une suite de tests** est un enchaînement de tests unitaires appliqués aux classes présentes dans les projets d’une solution.

**Un test unitaire** est une méthode dont le but est de tester une petite unité de code homogène et isolable, et qui renvoie un résultat OK ou erreur. Typiquement, un test unitaire teste une seule méthode à la fois, mais il peut aussi en tester plusieurs.

# Création d’une suite de tests dans Visual Studio

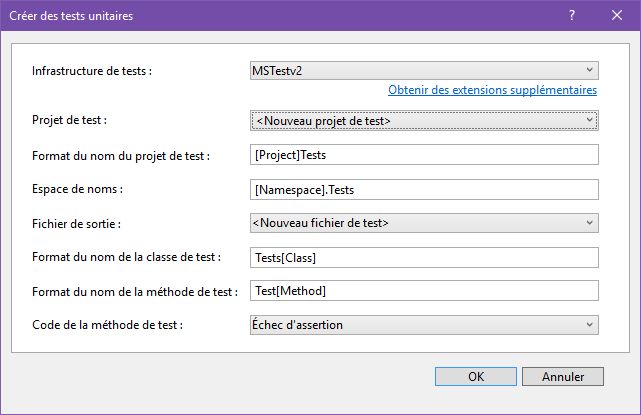
## Création d’un projet de tests

Une suite de tests est créée dans Visual Studio au moyen d’un **projet de tests unitaires**, ajouté dans la solution contenant les projets à tester. L’image ci-dessous montre l’ajout d’un projet de tests :



Le projet de tests unitaire doit référencer tous les projets dont il souhaite tester les classes.

Une autre façon de créer le projet de tests, est de partir du projet contenant les classes à tester. Dans le menu contextuel de l’une des classes, choisir « Créer des tests unitaires », ce qui ouvre la boite de dialogue suivante :



Dans la zone « Projet de test », le choix par défaut permet de créer automatiquement un nouveau projet de tests dans la solution. Les formats des noms du projet, des classes et des méthodes de tests peuvent être personnalisés, comme le montre l’image.

## Structure d’un projet de tests

Le diagramme ci-dessous représente la hiérarchie d’éléments qui figurent dans un projet de tests.

Un projet de tests peut contenir plusieurs classes de tests, qui peuvent elles-mêmes contenir plusieurs méthodes de tests.

A chaque niveau, il est possible de définir des méthodes d’initialisation et de nettoyage, qui seront exécutées respectivement juste après le chargement et juste avant le déchargement du projet, de la classe ou de la méthode.

Ces méthodes sont identifiées au moyens d’attributs déclarés dans l’espace de noms Microsoft.VisualStudio.TestTools.UnitTesting. Elles sont toutes facultatives.

Le diagramme ci-dessous représente l’enchaînement de ces méthodes. Les noms indiqués sont ceux des attributs qui décorent les méthodes. Les noms des méthodes elles-mêmes sont libres.

Dans un projet de tests, il peut y avoir

* Une seule méthode d’initialisation d’assemly
* Une seule méthode de nettoyage d’assemly

Ces méthodes peuvent être placées dans n’importe quelle classe de test

Dans une classe de tests, il peut y avoir :

* Une seule méthode d’initialisation de classe
* Une seule méthode d’initialisation de test
* Plusieurs méthodes de tests
* Une seule méthode de nettoyage de test
* Une seule méthode de nettoyage de classe

Les méthodes d’initialisation et de nettoyage des tests sont exécutées avant et après **chaque** méthode de test, c’est-à-dire autant de fois qu’il y a de méthodes de tests dans la classe.

L’exemple ci-dessous illustre les signatures et l’enchaînement des méthodes. Il contient deux classes de tests dans le même projet.

namespace Tests

{

    [TestClass]

    public class TestClass1

    {

        [ClassInitialize]

        public static void InitClass(TestContext context)

        {

            Debug.WriteLine("Init class 1");

        }

        [TestInitialize]

        public void InitTest()

        {

            Debug.WriteLine("Init test class 1");

        }

        [TestMethod]

        public void TestMethod1()

        {

            Debug.WriteLine("Test method 1 of class 1");

        }

        [TestMethod]

        public void TestMethod2()

        {

            Debug.WriteLine("Test method 2 of class 1");

        }

        [TestCleanup]

        public void CleanTest()

        {

            Debug.WriteLine("Clean test class 1");

        }

        [ClassCleanup]

        public static void CleanClass()

        {

            Debug.WriteLine("Clean class 1");

        }

        [AssemblyCleanup()]

        public static void CleanAssembly()

        {

            Debug.WriteLine("Clean Assembly");

        }

    }

    [TestClass]

    public class TestClass2

    {

        [AssemblyInitialize()]

        public static void InitAssembly(TestContext context)

        {

            Debug.WriteLine("Init Assembly ");

        }

        [ClassInitialize]

        public static void InitClass(TestContext context)

        {

            Debug.WriteLine("Init Class 2");

        }

        [TestMethod]

        public void TestMethod()

        {

            Debug.WriteLine("Test Method of class 2");

        }

    }

}

Lorsqu’on exécute l’ensemble des tests en debug (CTRL+R, CTRL+A), Ce code produit la sortie suivante :

Init Assembly

Init class 1

Init test class 1

Test method 1 of class 1

Clean test class 1

Init test class 1

Test method 2 of class 1

Clean test class 1

Init Class 2

Test Method of class 2

Clean class 1

Clean Assembly

Il est intéressant de noter que même si les méthodes d’initialisation et de nettoyage de l’assembly ne se trouvent pas dans la même classe de test, cela ne pose pas de problème.

Les méthodes d’initialisation de l’assembly et de la classe sont statiques et possèdent un paramètre de type TestContext, qui stocke des informations contextuelles sur les tests.

## Ecriture de tests

Une classe de test contient une ou plusieurs méthodes de tests unitaires, et optionnellement une ou plusieurs des méthodes d’initialisation et de nettoyage présentées dans l’exemple précédent.

Les méthodes de tests doivent être décorées de l’attribut TestMethod. Elles utilisent les méthodes statiques de la classe Assert pour implémenter les tests.

Exemple :

[TestClass]

public class TestClass1

{

    private static IDataContext \_dataContext;

    [ClassInitialize]

    public static void InitClass(TestContext context)

    {

        \_dataContext = new ContexteNorthwind();

    }

    [TestMethod]

    public void TestMethod1()

    {

        Assert.AreEqual(5, \_dataContext.GetNbProduits("France"));

    }

}

Dans certains cas, on s’attend à ce que la méthode testée lève une exception spécifique dans certaines circonstances. Pour tester cela, il faut décorer la méthode de test de l’attribut ExpectedException.

Exemple :

[ExpectedException(typeof(FormatException))]

[TestMethod]

public void TestMethod2()

{

    DateTime reveillon = Convert.ToDateTime("31-12--2017");

}

**Débogage**

Pour déboguer les méthodes de tests, mettre des points d’arrêts sur les méthodes à déboguer et lancer la suite de tests en debug en cliquant sur un des sous-menus du menu Tests \ Déboguer

## Bonnes pratiques

Un test unitaire doit être véritablement unitaire. La classe testée doit être isolable, et ses méthodes doivent être testées une par une, ou en nombre réduit si on doit en tester plusieurs à la fois.

Ne tester qu’un comportement à la fois de façon à ce que le test reste simple et facile à comprendre.

Si une classe est difficile à tester car trop volumineuse ou possédant trop de dépendances, cela peut être l’occasion de la refactoriser.

Il faut s’efforcer de faire des tests courts, pour limiter le temps global de test. Si une suite de tests est trop longue à jouer, elle risque de ne pas pouvoir être jouée aussi souvent qu’on le souhaiterait.

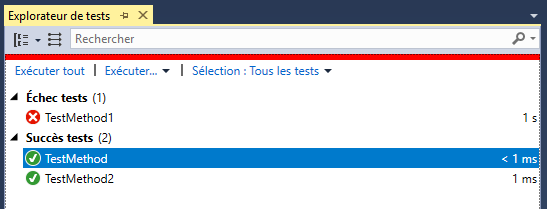
Respecter les conventions de codage aussi dans les tests unitaires.

Ne pas essayer de couvrir 100% du code par des tests unitaires. Choisir les parties de code les plus pertinentes à tester. On ciblera en priorité les tests difficiles à réaliser par un humain, tels que la vérification de résultats de calculs complexes, ou de requêtes complexes sur une base de données.

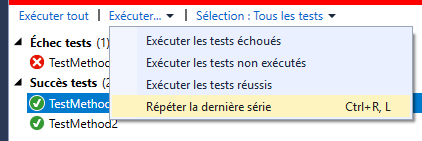
Ne pas remettre le développement des tests unitaires à plus tard. Si on n'utilise pas l’approche TDD (Test Driven Development), développer tout de même les tests unitaires le plus tôt possible, sinon ils risquent de ne jamais être faits.

# Exécution de tests unitaires

[L’explorateur de tests](https://docs.microsoft.com/fr-fr/visualstudio/test/run-unit-tests-with-test-explorer) est une fenêtre qui permet de sélectionner les tests à réaliser et de voir les résultats. On y accède par le menu Tests \ Fenêtres \ Explorateur de tests.



Le lien « Exécuter tout » permet de lancer l’ensemble des tests de tous les projets de tests de la solution, tandis que le lien « Exécuter… » ouvre un menu permettant de sélectionner seulement certains tests :



Une méthode de tests peut également être exécutée directement depuis la classe de tests, au moyen du menu contextuel « Exécuter les tests », qui s’affiche lorsqu’on fait un clic droit sur une méthode.

# Références bibliographiques

[Doc Microsoft](https://docs.microsoft.com/fr-fr/visualstudio/test/unit-test-your-code) sur les tests unitaires

Article « [Les 10 commandements des tests unitaires](http://blog.xebia.fr/2008/04/11/les-10-commandements-des-tests-unitaires/) » sur le blog Xebia