



Concepteur Développeur en Informatique

Développer une application x-tiers

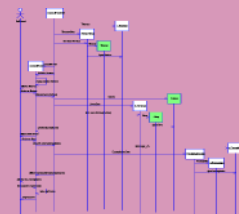
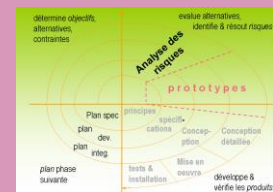
Introduction aux tests Unitaires

Accueil

Apprentissage

PAE

Evaluation



Localisation : U04-E02-S01

SOMMAIRE

1.	Introduction.....	3
2.	Les contraintes	4
3.	Une démarche en 3 Actes	5
4.	Premier exemple simple.....	5
4.1.	Les attributs.....	7
4.1.1.	TestFixture	7
4.1.2.	Test.....	7
4.1.3.	TestCase.....	7
4.2.	La classe Assert	7
5.	Documentation Framework Nunits.....	9
6.	Documentation sur les tests	9

1. Introduction

Un bon développeur doit s'assurer que chacun de ses composants fonctionne correctement. En programmation informatique, le test unitaire est un procédé permettant de s'assurer du fonctionnement correct d'une partie déterminée d'un logiciel ou d'une portion d'un programme (appelée « unité » ou « module »).

Le principe de la mise en place des tests unitaires est d'isoler chaque portion de code à tester et de s'assurer que chaque portion de code fonctionne correctement indépendamment du reste de l'application.

Par principe :

- Une application ne fonctionnera correctement que si chaque portion de code fonctionne directement.
- Cela n'implique pas pour autant que si chaque portion de code fonctionne correctement l'application fonctionne correctement.

Rappel logique sur l'implication et la contraposition ?

De nombreuses méthodes ou démarches préconisent de placer les tests au centre de nos préoccupations, qu'elles relèvent de l'agilité ou de l'Extrem Programming.

Certaines méthodes préconisent même de générer les méthodes réelles à partir des cas de tests : il est alors question de développement piloté par les tests. Ces méthodes sont alors affublées de l'acronyme TDD qui signifie Test Driven Development.

Quelle que soit la démarche retenue, un test vise à confronter une réalisation à sa spécification. Il est donc indispensable que la spécification du produit à réaliser soit disponible !

Le test définit un critère qui permet de statuer sur le succès ou sur l'échec d'une vérification.

Grâce à la spécification, on est en mesure de faire correspondre un état d'entrée donné à un résultat ou à une sortie.

Le test permet de vérifier que la relation d'entrée / sortie donnée par la spécification est bel et bien réalisée.

Petit rappel de définitions :

- Test : il s'agit d'une vérification par exécution.
- Vérification : ce terme est utilisé dans le sens de contrôle d'une partie du logiciel.
- Une « unité » peut ici être vue comme « le plus petit élément de spécification à vérifier »

Le développeur doit donc tester une unité, indépendamment du reste de l'application, ceci afin de s'assurer que celle-ci répond aux spécifications fonctionnelles et qu'elle fonctionne correctement en toutes circonstances.

Au niveau d'une application, nous devons ainsi nous assurer d'un bon **ratio de vérification de la couverture du code**.

Cela consiste à s'assurer que le test conduit à exécuter l'ensemble (ou une fraction déterminée) des instructions présentes dans le code à tester.

Un taux de couverture de 100 % est illusoire.

L'ensemble des tests unitaires doit pouvoir être rejoué après une modification du code afin de vérifier qu'il n'y a pas de régressions (l'apparition de nouveaux dysfonctionnements).

L'emploi d'une « stratégie de test » particulière peut alors limiter les tests à rejouer, par exemple : une analyse d'impact des modifications, corrélée à une preuve d'indépendance des modules, permet de cibler les cas de test unitaire à rejouer.

Des outils existent aujourd'hui pour rejouer tout ou partie des tests programmés.

2. Les contraintes

Nous devons nous assurer par les tests de la non régression de nos composants. Ainsi, toute fonctionnalité présente dans une version X doit être opérationnelle dans une version ultérieure à X.

Chaque modification apportée à une portion de code doit donc nécessiter de tester à nouveau cette portion de code.

Il est dès lors nécessaire d'utiliser des outils de programmation des tests qui nous permettront de rejouer automatiquement ces tests et ainsi conserver un bon niveau de productivité.

Il existe de nombreux outils de tests unitaires. Certains sont libres d'accès d'autres payants.

Certains sont bien intégrés à l'environnement de développement intégré Visual Studio, notamment à partir des versions professionnelles.

Des outils de tests peuvent être couplés avec des outils d'analyse du cycle de vie d'une application.

Ils permettent alors de s'assurer de la qualité du logiciel en proposant des statistiques sur les ratios de couverture de code des tests. L'idéal est de tendre vers un taux de couvertures des tests de 100% ...

Mais l'idéal comme la perfection ne font pas vraiment parties du monde réel.

Un des principaux outils de tests NUnits : issu de la version Java J2EE Units, c'est une des références du marché. Il est gratuit et disponible pour toutes les versions de Visual Studio.

Nous allons travailler avec l'outil intégré dans notre version EDI et son template qui permet de créer un projet dédié aux tests.

3. Une démarche en 3 Actes

La grande majorité des tests unitaires se réalise en 3 phases, selon un acronyme AAA (qui ne qualifie pas la qualité de l'andouillette...).

Cet acronyme correspond à des termes anglais qui précisent ces 3 phases :

1. A pour Arrange : Il s'agit de la première étape de préparation où nous allons définir les objets nécessaires au bon déroulement du test. Nous pouvons traduire Arrange par Préparer ou Arranger.
2. A pour Act : La méthode à tester est exécutée dans le contexte préparée à l'étape précédente. Nous pouvons traduire cette étape par Agir.
3. A pour Assert : Nous pouvons le traduire par Affirmer. C'est la même racine que nous retrouvons dans Assertion.

Nous pourrions aussi traduire ces verbes par :

Arranger : Il s'agit dans un premier temps de définir les objets, les variables nécessaires au bon fonctionnement de son test (initialiser les variables, initialiser les objets à passer en paramètres de la méthode à tester, etc.).

Agir : Ensuite, il s'agit d'exécuter l'action que l'on souhaite tester (en général, exécuter la méthode que l'on veut tester, etc.)

Auditer : Et enfin de vérifier que le résultat obtenu est conforme à nos attentes.

4. Premier exemple simple

Ci-dessous un test extrêmement simple pour comprendre la base du test unitaire. Nous avons un premier projet qui contient la logique métier d'un composant permettant d'additionner deux nombres.

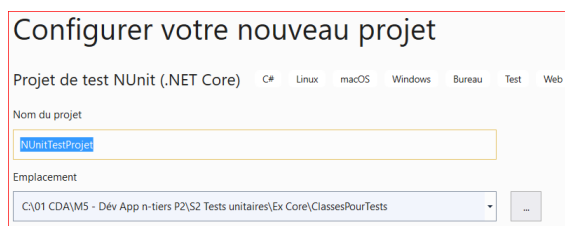
Il convient ici de vérifier le bon fonctionnement de la méthode Additionner de la calculatrice. Classe à tester :

```
public class Calculatrice
{
    1 référence | 1/1 ayant réussi
    public int Additionner(int a, int b)
    {
        return a + b;
    }
}
```

Elle se trouve dans un projet Calculatrice.BLL.

Nous devons installer le projet de tests unitaires.

Le plus simple est de disposer du modèle de projet. Choisissez projet de test Nunit pour core.



Configurer votre nouveau projet

Projet de test NUnit (.NET Core) C# Linux macOS Windows Bureau Test Web

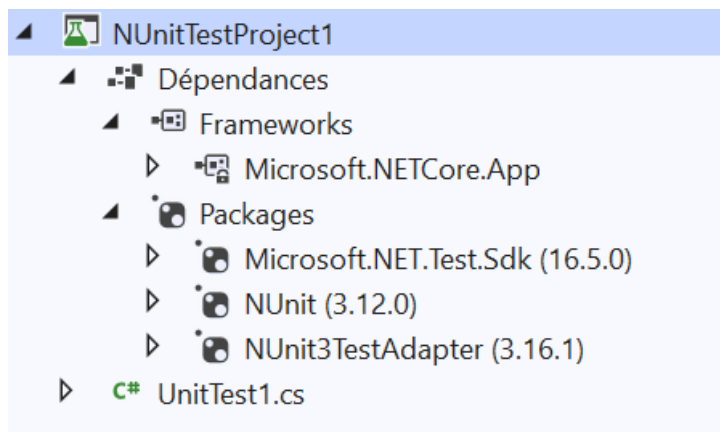
Nom du projet

NUnitTestProjet

Emplacement

C:\01 CDA\M5 - Dev App n-tiers P2\S2 Tests unitaires\Ex Core\ClassesPourTests

Vous devriez avoir les packages suivants installés :



Liez votre projet de classes à tester à votre projet de tests en ajoutant une dépendance.

La classe de test :

```
[TestFixture]
0 références
public class TestCalculatrice
{
    [Test]
    0 références
    public void AdditionnerDeuxEntiersQuelconques()
    {
        int a = 3, b = 4; // Arrange
        int expected = 7;
        Calculatrice.BLL.Calculatrice calculatrice
        = new Calculatrice.BLL.Calculatrice();
        Assert.AreEqual(expected, calculatrice.Additionner(a, b));
    }
}
```

Nous retrouvons bien les 3 temps Arrange, Act et Assert.

La variable **expected** représente le résultat attendu, alors que la variable **actual** représente le résultat obtenu.

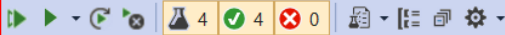
L'attribut **[TestFixture]** désigne une classe qui contient des tests unitaires. L'attribut **[Test]** indique une méthode qui est une méthode de test.

Il est possible de créer une liste pour les cas pilotés par les données. Préférable à l'usage de boucles avec un tableau dans la méthode de test.

```
[TestCase(-1,2,1)]
[TestCase(0,3,3)]
[TestCase(1,2,3)]
0 références
public void AdditionnerDeuxEntiersQuelconques(int a, int b, int expected)
{
    Calculatrice.BLL.Calculatrice calculatrice
    = new Calculatrice.BLL.Calculatrice();
    Assert.AreEqual(expected, calculatrice.Additionner(a, b));
}
```

Cet exemple est correct car l'objectif du test ne change pas.

Exécutons les tests et observons les résultats :

Explorateur de tests			
			
Test	Durée	Caractéristiques	Message d'erreur
✓ Calculatrice.Tests (4)	10 ms		
▶ ✓ Calculatrice.Tests.TestCalculatrice.AdditionnerDeuxEntiersQuelconques (4)	10 ms		

4.1. Les attributs

Ils en existent de très nombreux pour des objectifs très différents.

Vous y trouverez les différents attributs qui vous permettront de configurer vos tests et modifier le comportement des méthodes de tests avec une approche non intrusive.

<https://docs.nunit.org/articles/nunit/writing-tests/attributes/apartment.html>

4.1.1. TestFixture

Cet attribut indique que la classe contient des tests et qu'elle doit être prise en compte lors de l'exécution des tests.

4.1.2. Test

C'est bien sûr l'attribut incontournable.

Le test est généralement défini selon le principe des 3 A présenté en introduction.

4.1.3. TestCase

Cet attribut indique que la méthode sera appelée avec les différents cas définis. Permet de jouer le même test avec plusieurs cas définis en entrée.

4.2. La classe Assert

La classe statique **Assert** présente plusieurs méthodes qui peuvent être sollicitées lors de la vérification. Le tableau ci-dessous récapitule les principales méthodes résumées :

Nom	Description
AreEqual	Vérifie que les valeurs spécifiées sont égales.
AreNotEqual	Vérifie que des valeurs spécifiées ne sont pas égales.
AreNotSame	Vérifie que des variables objets spécifiées font référence à des objets différents.
AreSame	Vérifie que des variables objets spécifiées font référence au même objet.
Inconclusive	Indique qu'une assertion ne peut pas être prouvée true ou false. Permet également d'indiquer une assertion qui n'a pas encore été implémentée.
IsFalse	Vérifie qu'une condition spécifiée est false .
IsInstanceOfType	Vérifie qu'un objet spécifié est une instance d'un type spécifié.
IsNotInstanceOfType	Vérifie qu'un objet spécifié n'est pas une instance d'un type spécifié.
IsNotNull	Vérifie qu'un objet spécifié n'est pas null .
IsNull	Vérifie qu'un objet spécifié est null .
IsTrue	Vérifie qu'une condition spécifiée est true .

A cette forme classique tend à se substituer une autre forme plus souple et extensible.

La forme classique de l'assertion avec Assert a été complétée par un modèle Assert basé sur des contraintes qui utilise une seule méthode de la classe Assert pour toutes les assertions.

La forme classique reste opérationnelle mais ne fera plus l'objet de nouvelles implémentations.

La logique nécessaire pour effectuer chaque assertion est intégrée dans l'objet de contrainte passé en tant que deuxième paramètre à cette méthode.

```
Assert.That(message, Is.EqualTo("Bienvenue aux CDA pour tests"));
```

Autre exemple ici avec une expression régulière.

```
[Test]
[TestCase("Les CDA recherchent des entreprises pour PEE")]
public void VerifierTermesContenusPhrase(string phrase)
{
    Assert.That(phrase, Does.Match("CDA.*entreprise.*PEE"));
    Assert.That(phrase, Does.Match("cda.*entreprise.*pee").IgnoreCase);
    Assert.That(phrase, Does.Not.Match("PEE.*recher.*CDA"));
}
```

Plusieurs formes possibles pour atteindre le même résultat. Exemples pour vérifier le numéro de Siret

```
[Test]
[TestCase("0123456", false)]
[TestCase("0123456ABC", false)]
[TestCase("82436343600737", true)]
public void VerifierNumeroSiret(string numSiret, bool attendu)
{
    Assert.That(Siret.IsSiretValide(numSiret), Is.EqualTo(attendu));
}
```

Ou :

```
[Test]
[TestCase("0123456")]
[TestCase("0123456ABC")]
public void VerifierNumeroSiretFalse(string numSiret)
{
    Assert.That(Siret.IsSiretValide(numSiret), Is.False);
}

[Test]
[TestCase("82436343600737")]
public void VerifierNumeroSiretTrue(string numSiret)
{
    Assert.That(Siret.IsSiretValide(numSiret), Is.True);
}
```


5. Documentation Framework Nunit

Vous retrouverez toute la documentation nécessaire de Nunit sur le site en ligne.
Nous travaillons ici avec la version 3.

<https://docs.nunit.org/articles/nunit/intro.html>

Par exemple, cette page fournit des exemples de l'utilisation de la classe Assert :

6. Documentation sur les tests

Si dans votre futur professionnel vous aviez besoin d'informations complémentaires sur les tests, ce site, extrêmement riche, aborde le testing sous de nombreux angles et dans toutes ses dimensions.

<https://www.guru99.com/software-testing.html>