Test Driven Development avec Visual Studio 2012

Dans nos précédents articles, nous nous sommes intéressés à certains aspects liés à la qualité du code. Nous avons abordé les principes S.O.L.I.D. Dans ce nouvel article, nous souhaitons continuer en abordant la qualité du développement ; un sujet qui est plus que jamais d'actualité avec l'utilisation des méthodologies Agile et particulièrement le Test Driven Development.

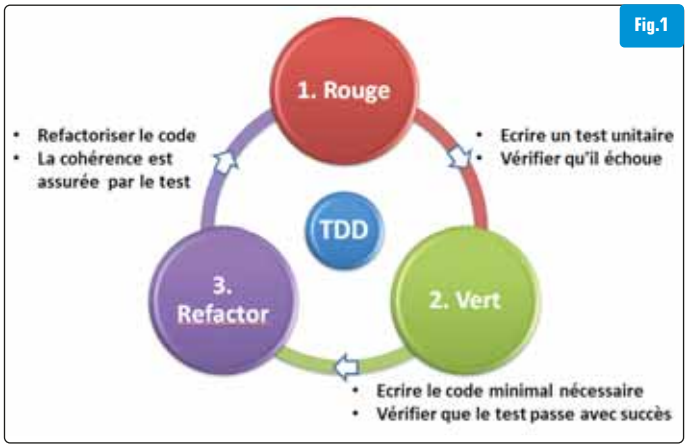


Fig.1

L'objectif de cet article est d'être pragmatique et orienté projet : nous souhaitons montrer que les nouveautés apportées dans la nouvelle version de Visual Studio 2012 permettront aux développeurs de faciliter et de rendre naturel le développement TDD. Il s'agit également de simplifier et limiter l'intégration des différents outils dans un processus d'intégration continue.

Le Test Driven Development (développement piloté par les tests) décrit une technique de développement de logiciel qui consiste à écrire des tests unitaires avant l'écriture du code source.

Voici les différentes étapes du Test Driven Development [Fig.1] :

- Ecriture du premier test pour une nouvelle fonctionnalité puis vérification de l'échec du test (l'implémentation de la nouvelle fonctionnalité n'étant pas encore réalisée)
- Implémentation du code minimal nécessaire pour passer le test puis vérification du succès du test (l'implémentation fournit alors le comportement attendu)
- Refactorisation et optimisation du code, les tests assurent la cohérence des fonctionnalités

L'utilisation du Test Driven Development permet ainsi l'obtention d'un code source très fiable, prévisible, robuste et, après refactorisation, hautement optimisé. Les tests assurent un comportement correct de celui-ci indépendamment des situations auxquelles il pourra être exposé. Le code source devient alors valide en toutes circonstances.

Pour créer de bons tests unitaires, il faut d'abord réfléchir à la conception du programme et à la façon dont il va être utilisé. Il faut éviter une précipitation dans l'implémentation avant d'avoir défini les objectifs. Des erreurs de conception peuvent ainsi être identifiées et résolues plus vite. L'implémentation passe seulement après la validation de la conception complète via les tests unitaires. Les tests unitaires deviennent une sorte de spécification générale décrivant les fonctionnalités du programme de manière unitaire.

Quant à la refactorisation, l'optimisation et la restructuration du code peuvent se faire sans risque car ils sont vérifiables ; les tests unitaires assurant alors la non-régression technique et la cohérence du comportement. Le comportement étant exprimé dans les tests unitaires, ils valident que le programme se comporte toujours de la même façon si les tests passent avec succès. De plus, en associant la méthode d'Extreme Programming (XP) et la programmation en binôme, on obtient un code de très bonne qualité.

Contrairement à d'autres méthodes où les tests sont abordés en fin du cycle, le Test Driven Development met donc le focus sur les tests dès les premières étapes du cycle de développement. Aucun développement ne peut commencer avant que les tests soient conçus et implémentés. Cette méthode a donc un large impact sur l'organisation de l'équipe de développement.

Support des frameworks de tests

Les versions précédentes de Visual Studio permettent l'utilisation d'autres frameworks de tests, mais cela implique certaines limitations comme par exemple :

- L'utilisation d'une application tierce pour jouer les tests unitaires comme avec Gallio : l'utilisation de l'application Icarus Runner est indispensable pour pouvoir exécuter les tests unitaires.
- La couverture de code native ne fonctionne pas sans l'utilisation d'autres plugins (NCover).
- L'exécution des tests unitaires diffère en fonction des plugins : un test utilisant MSTest ne s'exécute pas de la même manière s'il est lancé depuis Visual Studio 2008 ou depuis Resharper, ceci peut poser problème lors de l'utilisation d'un processus d'intégration continue.

Au final avec l'ancienne version de Visual Studio 2010, on finit par multiplier les plugins et les outils pas toujours compatibles les uns avec les autres, ce qui ne permet pas vraiment de tirer avantage des fonctionnalités avancées des frameworks de tests spécialisés.

Dans un environnement où le nombre de projets est conséquent, on a besoin de pouvoir utiliser un framework de tests plutôt qu'un autre en fonction des spécificités du projet et des fonctionnalités de celui-ci. Il est donc important de pouvoir proposer aux équipes de développement un large choix de frameworks de tests avec un coût d'intégration limité (dans les outils de développement comme dans les outils d'intégration continue). L'une des belles surprises de cette nouvelle version de Visual Studio est justement le support de plusieurs frameworks de tests unitaires tels que :

Pour .NET :

- NUnit (http://nunit.org)
- xUnit.net (http://xunit.codeplex.com)
- MBUnit (https://github.com/Gallio/Gallio-VS2011-Integration)

Pour Javascript/HTML :

- QUnit & Jasmine (http://chutzpah.codeplex.com)

Pour C++ :

• MSTest Native (http://aka.ms/mstest-native-vs11)

Prenons, par exemple l'intégration du framework de tests NUnit. Après l'installation de NUnit via NuGet [Fig.2], l'adaptateur de tests est téléchargeable sous la forme d'un plugin via le gestionnaire d'extension [Fig.3], disponible dans le menu « Tools/Extensions and Updates… » de Visual Studio 2012 [Fig.4].

Cette fonctionnalité est complètement intégrée à Visual Studio et les adaptateurs de tests sont disponibles gratuitement sur internet. Ensuite, vous pouvez écrire vos tests unitaires en utilisant les fonctionnalités intégrées et les exécuter directement dans Visual Studio 2012. Vous pouvez ainsi obtenir les résultats des tests et d'autres informations comme l'analyse de la couverture du code [Fig.5].

Le support des tests unitaires pour les programmes en C++ est également une nouveauté non négligeable [Fig.6].

Les développeurs C++ pourront ainsi se mettre au TDD en utilisant directement MSTest Native. Voici un exemple d'implémentation des tests unitaires en utilisant C++ et MSTest Native :

```
#include «stdafx.h»
#include «CppUnitTest.h»
#include <Calculator.h>

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace NativeUnitTests
{
 TEST_CLASS(UnitTests)
 {
 public:
   TEST_METHOD(Calculator_AdditionTest)
   {
     int value1, value2, expectedValue, actualValue;
     Calculator::Calculator calculator;
     a=1, b=2;
     expectedValue = 3;
     actualValue = calculator.Addition(value1, value2);
     Assert::AreEqual(expectedValue, actualValue);
   }
 };
}
```

Support de « async » et « await » dans les tests

Avec l'arrivée du framework .Net 4.5 et surtout de Windows 8, il est évident que la programmation asynchrone est l'une des nouveautés les plus importantes et incontournables ; Visual Studio 2012 et son framework de tests unitaires supportent parfaitement cette nouvelle approche. Les mots clé async et await introduits par .Net 4.5 peuvent désormais être utilisés pour faire des tests unitaires sur des méthodes asynchrones. Voici un exemple d'une méthode asynchrone à tester :

```
public async Task<int> AdditionAsync(int value1, int value2)
{
    // Simulate computing time
    await Task.Delay(3000);
    return  value1 + value2;
}
```

La méthode de test correspondante, implémentée en utilisant NUnit, pourrait être la suivante :

```
[Test]
public async void AdditionAsync_PositiveValues()
{
    var calculator = new Calculator();
    var result = await calculator.AdditionAsync(1, 2);
    Assert.AreEqual(3, result);
}
```

Gestion des tests via Test Explorer

La première impression à l'ouverture de la fenêtre « Test Explorer » est très positive. Voici les principaux changements apportés :

- Les panneaux « Test View » et « Test Results » ont été supprimés et remplacés par le « Test Explorer », ceci améliore l'interaction entre le développement (le code) et les tests.
- L'interface est simple mais efficace : toutes les informations sont accessibles par simple clic de la souris de manière parfaitement intuitive.
- Les tests sont regroupés en fonction de leur statut (failed, passed,…) et les premiers tests visibles sont ceux en échec : ceux qui intéressent en priorité les développeurs. De plus, il n'y a plus besoin de passer par le panneau « Test Results » pour accéder aux sources du test, un simple double clic permet d'atteindre les sources.


Fig.2


Fig.3

- L'exécution de l'analyse de la couverture du code est simplifiée. Dans les versions précédentes, lancer une couverture de test était certes simple mais pas très intuitif : vous deviez créer un fichier de configuration, le configurer, le lancer depuis le menu Visual Studio et ouvrir la fenêtre adéquate pour obtenir le résultat. Avec Visual Studio 2012, tout cela est plus facile car tout est intégré dans l'interface du « Test Explorer ». Le lancement d'une analyse de la couverture du code se fait directement avec la souris.

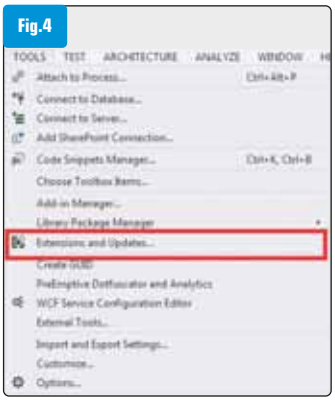
Post Build Test Runs

Une bonne habitude à adopter lors de l'application du TDD est d'exécuter les tests unitaires le plus souvent possible pour s'apercevoir au plus tôt d'un dysfonctionnement ou d'une éventuelle régression. Il existe dans la nouvelle version de Visual Studio 2012 la possibilité d'activer l'exécution des tests unitaires après chaque compilation. Ceci est disponible dans le menu « Test/Test Settings » [Fig.7], mais aussi directement depuis le « Test Explorer ». De plus, les tests unitaires tournent sur un Thread d'arrière-plan, la productivité des développeurs n'étant donc pas impactée.

Compatibilité ascendante entre Visual Studio 2010 et Visual Studio 2012

Ceux qui, comme nous, ont déjà effectué la migration des tests unitaires de Visual Studio 2008 vers Visual Studio 2010, ont pu rencontrer quelques difficultés et quelques bugs car la migration n'était pas transparente et parfois complexe. Cette opération était fort pénible, dans certains cas, la migration des applications devait même être forcée vers le framework .NET 4.0 ! Rassurez-vous, avec Visual Studio 2012, vous n'aurez aucun problème de ce type, car les composants utilisés sont les mêmes que ceux utilisés par Visual Studio 2010. En effet, le composant actuel est encore « Microsoft.VisualStudio.QualityTools.unitTestFramework.dll » dans la version 10.0.0.0 et utilise toujours le runtime .NET v2.0.50727.

Fig.4



Fakes Framework (Stubs et Shims)

Visual Studio 2012 a également intégré le « Fakes Framework » issu du projet « Moles » créé par l'équipe Microsoft Research. Ce framework sera disponible uniquement avec la version Ultimate de Visual Studio 2012.

Le but de cet outil est de permettre aux équipes de développement de produire rapidement et facilement des tests unitaires. Pour cela, le « Fakes Framework » introduit deux notions :

- Les Stubs : ce sont des implémentations d'interfaces ou de classes abstraites automatiques pouvant être utilisées par les tests afin d'isoler la partie à tester unitairement.
- Les Shims : ce sont des mécanismes qui interceptent des appels de méthodes au run-time et les remplacent par d'autres. Les Shims peuvent être utilisés pour isoler les appels vers des méthodes contenues dans des objets qui ne peuvent normalement pas être « mockés ». Par exemple, il est impossible de tester unitairement des méthodes faisant appel à certains objets du framework .NET. Grâce aux Shims, il est possible de rediriger les appels vers ses propres implémentations.

Les classes du framework .NET incluses dans les namespaces « mscorlib » et « system » ne peuvent pas avoir de « Fake Assembly ». On ne pourra donc malheureusement pas créer de Shims pour la classe « System.Configuration.ConfigurationManager » par exemple. Le « Fakes Framework » apportent un réel plus par rapport à des framework de tests existants (comme RhinoMock) qui poussent souvent les développeurs à modifier leur code fonctionnel pour pouvoir effectuer des tests unitaires.

Test Driven Development avec Visual Studio 2012

Prenons un exemple concret dans Visual Studio 2012 pour implémenter des fonctionnalités d'un calculateur.

Phase 1 : Ecriture du premier test pour une nouvelle fonctionnalité

En respectant le Test Driven Development, il faut d'abord créer les tests unitaires pour les méthodes « Addition » et « Multiplication ». Pour cela, il faut ajouter un nouveau projet de type « Unit Test Project ». Voici un exemple d'implémentation des tests unitaires en utilisant NUnit :

```
using System;
using NUnit.Framework;
using TDD_avec_VS2012;

namespace NUnitTests
{
 [TestFixture]
 public class UnitTests
```

Fig.5

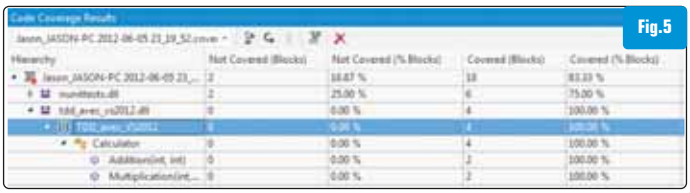
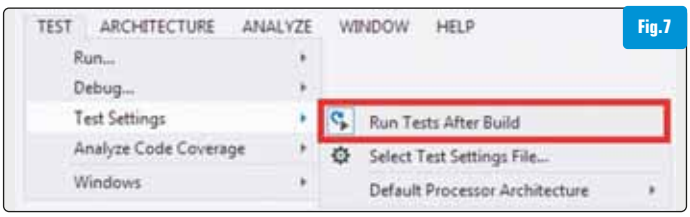


Fig.6

Fig.7



```
  {
    [Test]
    public void Calculator_AdditionTest()
    {
      var calculator = new Calculator();
      Assert.AreEqual(calculator.Addition(2, 3), 5);
    }

    [Test]
    public void Calculator_MultiplicationTest()
    {
      var calculator = new Calculator();
      Assert.AreEqual(calculator.Multiplication(2, 3), 6);
    }
  }
}
```

A noter que la définition de la classe « Calculator » et de ses méthodes « Addition » et « Multiplication » n'existent pas encore à ce stade. Visual Studio 2012 permet la génération du code manquant de manière automatique. Pour cela, il faut cliquer droit sur la définition « new Calculator » dans le projet de tests unitaires et choisir de générer la classe via l'option « Generate/New Type » [Fig.8]. Un assistant s'ouvre permettant de configurer le type (classe, struct, interface, enum), l'accès (public, internal), le projet de destination et le nom du fichier pour la création de la classe manquante.
L'étape suivante consiste donc à générer automatiquement les méthodes manquantes au sein de la nouvelle classe « Calculator ». Ceci se fait quasiment de la même manière, en cliquant droit sur les appels « calculator.Addition(…) » et « calculator.Multiplication(…) » dans le projet de tests unitaires et en choisissant « Generate/Method Stub ». Voici ce qui est généré :

```
public object Addition(int p1, int p2)
{
    throw new NotImplementedException();
}

public object Multiplication(int p1, int p2)
{
    throw new NotImplementedException();
}
```

La dernière étape de cette phase comporte le lancement de « Test Explorer » et l'exécution de tous les tests unitaires via « Run All » (ou utiliser l'option d'exécution des tests unitaires après compilation). Comme attendu, les tests échouent car l'implémentation du code n'a pas encore été faite.

Phase 2 : Implémentation du code minimal nécessaire pour passer le test

Il ne reste plus qu'à écrire le code qui implémente les fonctionnalités attendues. L'idée est de programmer le code le plus simple répondant aux besoins. L'optimisation et l'amélioration interviennent dans un deuxième temps dans la phase suivante (la refactorisation).

Fig.8

```
public int Addition(int value1, int value2)
{
    return value1 + value2;
}

public int Multiplication(int value1, int value2)
{
    return value1 * value2;
}
```

Suite à l'implémentation, il faut alors relancer « Test Explorer » et exécuter tous les tests unitaires via « Run All » (ou utiliser l'option d'exécution des tests unitaires après compilation) et valider que les tests passent avec succès.
A ce stade-là, le code répond aux besoins attendus. La structure finale inclut un projet avec l'implémentation et un autre avec tous les tests unitaires. Par contre, le code n'est peut-être pas optimisé et sa qualité peut laisser à désirer. Il va falloir donc l'améliorer. Les tests unitaires servent dans ce cas comme filet de sauvetage : la refactorisation du code peut se faire sans avoir peur des régressions, car celles-ci seront détectées par les tests unitaires.

Phase 3 : Refactorisation et optimisation du code

La refactorisation décrit le processus d'amélioration de code après son écriture en modifiant sa structure interne sans modifier son comportement extérieur. On transforme un code qui fonctionne en un code qui fonctionne de manière optimale. Souvent, il en résulte un code plus rapide, utilisant moins de mémoire ou simplement présentant une implémentation plus élégante. Cela consiste à :

- Détecter et éliminer toute duplication de code
- Limiter la complexité et le nombre de classes
- Simplifier et optimiser l'algorithmique des méthodes
- Relocaliser, renommer et harmoniser les méthodes
- Améliorer la lisibilité du code
- Supprimer le code non utilisé (dit code mort)
- Ajouter des commentaires sur des sections complexes

Dans l'exemple, il n'y a rien à refactoriser, car il n'y a pas encore beaucoup de méthodes ni de classes implémentées. Cette dernière étape doit quand même être réalisée en fin de chaque itération du cycle. Le cycle incluant d'autres fonctionnalités recommence ensuite depuis la phase 1.

Conclusion

Visual Studio 2012 apporte un grand nombre d'améliorations autours des tests unitaires, aussi bien au niveau du framework que des outils. De plus, la flexibilité de pouvoir utiliser son framework de tests unitaires de préférence est un grand avantage. L'association entre TDD et Visual Studio 2012 se fait très facilement. N'hésitez donc pas à l'utiliser avec vos projets agiles !


Jason **DeOliveira**

Fathi **Bellahcene**