PIF 6005 Travail I

elouanes.khelifi@uqtr.ca

khee23019700

Khelifi, Elouanes

2022

# Introduction :

Dans ce travail, nous avons pour but d’approfondir notre compréhension du fonctionnement des tests unitaires et leur importance dans un processus de développement. Le travail sera divisé en trois étapes, ayant chacune pour objet de couvrir un certain aspect de la problématique susmentionné. Tout le travail sera programmé en C# dans des environnement ciblant le (.Net framework v4.8). Il est nécessaire d’utiliser une version Entreprise de visual studio pour reproduire quelques partie de ce travail.

# Etape 1 (Test de classe SimpleCalculator) :

Dans cette première étapes, nous allons créer une classe de test contenant une batterie de tests unitaires, a fin de tester les méthodes d’une classe SimpleCalculator.cs. Nous allons d’abord faire la tache manuellement en définissant une classe de test contenant un seul test unitaire. Puis générer le reste des tests en utilisant des modules de génération automatique de tests.

## Partie I ( testes manuelles ) :

La figure 1 montre représente le code source de la classe que l’on veut tester. Elle se comporte de quatre méthodes définissant les opérations arithmétique addition, soustraction…etc. Ainsi qu’une méthode un peu plus complexe (en termes de complexité cyclomatique). Nous allons en détailler un peu plus le long de cette section.

A picture containing graphical user interface

Description automatically generated

Figure Class SimpleCalculator.cs (a tester)

Pour un test manuelle tester la méthode Div(int , int) qui divise un nombre a d’un nombre b. la figure suivante montre la classe de test avec le test unitaire en question.

Text

Description automatically generated

Figure 2 Classe SimpleTests.cs qui définit les tests manuelles

Analysons un peu ce code. La ligne avant la déclaration de classe représente un attribut. Les attributs ajoutent des métadonnées à nos classes pour qu’il soit traité différemment à l’exécution. L’attribut [TestClass] en l’occurrence, définit une class comme étant une classe de test. L’attribut *TestClass* se trouve dans le *namespace* Microsoft.VisualStudio.TestTools.UnitTesting. En l’ajoutant à notre class l’IDE Visual Studio saura qu’il s’agit d’une classe de test à compiler comme tel.

A l’intérieure d’une classe de test. La méthode précédée par un attribut [TestInitialize] sera exécutée en premier (avant d’exécuter les tests). Dans cette méthode on pourra créer des instances des classes a tester et les préparer avant de les passer au tests.

Enfin, l’attribut [TestMethod] definit les methode de test. Dans ces méthodes, nous commençons par préparer le jeu de test ainsi que le resultat attendu. Puis en utilisant la methode **Assert.AreEqual( methodCall, result)** nous pouvons tester si la methode retourne bien ce que l’on attends d’elle.

A picture containing graphical user interface

Description automatically generated

Figure 3 Test passe avec succes.

On peut voir que le test passe parfaitement pour deux valeurs arbitraires. Mais on sait bien que cela n’est pas une si bonne nouvelle. Car le code de la methode Div(int , int) ne traite pas le cas de division par zero.

Cet exemple illustre l’importance du choix de jeu de test lors de la génération de teste unitaires. En effet si notre jeu de teste de couvre pas tout les cas limite. Nos tests retourneront des résultats faussement positifs.

**Solution :** on pourrait ajouter une boucle for qui itère les couples (valeurs, résultat) et applique la fonction **Assert** sur ces derniers a chaque itération. Alternativement, on pourrait aussi créer plusieurs méthodes de test. Avec un nouveau jeu de test a chaque fois.

## Partie II (Tests Automatiques) :

Une façon plus efficace de faire les choses est de créer les tests automatiquement. Pour ce faire, nous allons utiliser le plugin *IntelliTest* offert dans les licenses *Etreprise* de Visual studio.

Cet outil facilite grandement la génération de test en prenant en suivant des stratégies de détection de cas limites en analysant toutes les branches du code de la méthode en question. Pour profiter des avantages offerts par ce il faut s’assurer que les projets dans les quels sont définies les classes a tester ciblent bien le .NetFramework. IntelliTest ne prend toujours pas en charge les projet .Net Core et .Net standard. Il faut aussi s’assurer que les projets ne sont pas destinés à des architectures x64 bit.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 4 Intellitest menu contextual

Si notre configuration suit les conditions mentionnées précédemment on devrait pouvoir voir l’option intellitest sur le menu comme le montre la figure 4. On cliquant sur create intellitest, le plugin nous affiche un formulaire pour configurer l’emplacement et les convention de nomination des tests générés. (figure 5)

Graphical user interface, text, application, email

Description automatically generated

Figure 5 Formulaire generation automatique de test

Sans trop aller dans les details, le seul champ interessant est le test framework qui definit quel extension de test utiliser. Pour cet example nous allons utiliser celui de microsoft appele MSTest, mais intellitest supporte plusieurs extensions comme UTest et NTest.

Text

Description automatically generated

Figure 6 Résultat génération de test

La classe générée par cette configuration (figure 5), est comme le montre la figure 6. Celle-ci possède les métadonnées nécessaires a pour générer des tout les tests de façon dynamique (à chaque exécution de cette classe une autre classe portant le même nom a l’exception d’un «.g.cs » à la fin, et contenant tous les tests pour les braches actuels). Si le code de la méthode testée change, plus besoin de réécrire tous les tests qui en sont affectés. La classe en génère les nouveaux tests figure 7.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 7 Execution de la classe generatrice via le menu run Intellitest.

Graphical user interface, text

Description automatically generated

Figure 8 Resultat de l'execution.

On voit immédiatement que la classe générée et exécute des tests sur le cas limites division par zéro, ainsi qu’un autre cas limite dont on a même pas pensé (dans le cas ou un argument prends la valeur minimum que peut prendre un entier de type *int32*, cela provoque un *over-flow* lors du calcul).

Ajoutons maintenant le code qui traite ce cas. Puis lançons de nouveau les tests.

Graphical user interface, application

Description automatically generated

Figure 9 Code après traitement cas limite

A screenshot of a computer

Description automatically generated with medium confidence

Figure 10 Nouveau resultat de test

On peut voir que les tests qui échouaient fonctionne à présent. Mais aussi qu’un nouveau test a été ajouter pour une couverture de code maximale (après ajout de nouvelles branches). La figure 11 montre les tests générés.

Text

Description automatically generated

Figure 11 Tests generees par la classe SimpleCalculatorTest.cs

Dans un cas de figure un peu plus complexe. La méthode Cota(int) qui retourne la Cote étant donne la note de cours, représente un défi un peu plus difficile. A cause de sa complexité cyclomatique relativement haute.

Dans des cas plus complexes il est facile d’avoir des tests qui ne couvre pas la totalité du code. On parle ici de niveau de couverture du code qui peut être quantifier en allant sur :

***Tools/Analyze code coverage on all tests.***

L’utilisation de intellisense nous offre un jeu de test très intéressant. Offrant une couverture de 100 % du code de la méthode.

A screenshot of a computer

Description automatically generated

Figure 12 Jeu de test pour methode Cota(int)

## Conclusion (comparaison) :

L’utilisation du générateur automatisé Intellisense apporte un gain très important en termes de temps. En effet la stratégie d’analyse offre un moyen rapide de générer du code de test en prenant en compte les différentes branches (chemins) possibles du code. Ce qui garantit une couverture maximale de code.

Mais du point de vue formel, on ne peut toujours pas arriver à des résultats objectivement complets. Cela dit, vu que les tests sont générés automatiquement, pour arriver à la certitude absolue, il faudra tester les tests pour s’assurer qu’il couvre bien tout les cas limite possibles. Chose qui n’est pas réalisable.

Il est aussi a prendre en compte que les tests générés automatiquement ne peuvent pas couvrir des cas complexe comme l’analyse d’une chaine de caractères.

En outre, une combinaison de tests manuelle et automatique (en ajoutant des assertions) est la stratégie la plus sûr à envisager. Il existe des méthodes formelles issue de la théorie de graphs qui peuvent être utile pour tester de manière exhaustive la logique derrière notre programme (e.g. Réseaux de Petri, automotates d’états fini etc…). Ce serait intéressant de les utiliser pour évaluer la qualité de nos tests.

# Etape 2 (programme concret):

Dans cette section nous allons étudier et tester un programme calculatrice capable de lire une chaine de caractères « string » représentant une expression arithmétique, et d’en calculer la valeur. Le programme est implémenté en C# et utilise toujours le framework .Net framework pour profiter des avantages intellitest.

## Ordre d’intégration :

La figure suivante représente le diagramme de classes utilise dans la conception du programme en question. Ce diagramme de classe a été généré en utilisant l’extension ClassDesignr de Visual Studio. Les associations en **vert** ont été détecté automatiquement par cet outil en scannant les attributs (fields) des classes (c’est donc des aggregations). Par conséquent, étant de donné que ce projet utilise massivement les design patterns pour diminuer le couplage, l’outil n’a pu détecter que deux associations. Les associations de couleur **orange** ont été ajoutes manuellement. Il s’agit des associations ou la classe associe est soit passe en paramètre dans une méthode. Sois instanciée dans le corp de cette dernière.

Graphical user interface

Description automatically generated

Figure 13 Diagramme de class projet Calculator.csproj

### Description détaillée des associations :

Le programme en question est destiné à calculer la valeur d’expressions arithmétiques composées d’opérations diverses (Addition, soustraction etc…). Les expressions peuvent être imbriquées pour former un calcul de plus en plus complexe de manière non exhaustive. Une expression peut être sous forme de chaine de caractère comme ceci :

***5+2\*4-2/4***

Dans cette conception, une expression peut être une opération entre deux nombres, une opération entre expression et un nombre, ou encore une opération entre deux expressions. Mais peut également être un nombre seul (sans opération). C’est possible via le patron **Composite**. Une expression implémente l’interface ***IExpression*** mais a aussi une association avec celle-ci. Ce qui justifie l’agrégation : ***Expression/IExpressions****,*et le fait que ***Expression*** et ***Number*** implémentent l’interface ***IExpression.***

La classe ***PostfixGenerator*** étant aussi est utilisée par Expression pour lire une expression sous forme de chaine de caractères et la transformer a la forme postfix qui facilite ensuite la transformation vers le type ***Expression****.*

Le patron de conception *Builder* est utilisé pour créer des expressions de type ***Expression*** et les imbriquer facilement en utilisant les méthodes *Add() Subtract()* etc.. de celui-ci.

Etant donné que ***Number*** et ***Expression*** sont toutes les deux des **IExpression**. Celles-ci doivent pouvoir être calculées de manières différentes pour avoir leurs valeurs. Pour ajouter des fonctionnalités de calcule à une ***IExpression*** (pour avoir la valeur d’une expression) nous devons trouver un moyen d’ajouter cette fonctionnalité sans avoir à changer son code, tout en prenant compte la façon par laquelle on doit se comporter chaque type. Nous avons opté pour l’utilisation du pattern Visitor, avec lequel on peut ajouter des fonctionnalités a des classes sans avoir à changer leurs codes. Tout en prenant compte du comportement diffèrent de chacune.

C’est pourquoi l’interface ***IExpression*** oblige les classes qui l’implémentent a implémenter un comportement pour la méthode *Accept(IExpressionVisitor).* Chacune des deux classes ***Expression*** et ***Number*** implémentent l’interface et doivent maintenant accepter un visiteur qui se chargera d’exécuter une fonctionnalité dont le code dépend du type de l’objet qui l’accepte.

Enfin, la classe ***Calculator*** sert d’interface pour la création et le calcule d’expressions. Elle suit le patron *Facade* qui tout simplement l’aide de plusieurs associations, synchronise le comportement de plusieurs types d’objets, en les faisant collaborer en harmonie pour effectuer une tache plus complexe.

## Génération des tests :

Ces relations entre nos classes nous offrent un programme facilement compréhensible et très bien organisé, qui correspond à notre intuition (ce qui est la devise de la programmation orienté objet). Cependant ces dépendances nous compliquent la vie quand a l’exécution des tests unitaires. Dans cette section nous allons tester nos classes de manière « hybride », c’est-à-dire on combinant les test ascendant (en commençant des classes tout à la fin de la chaine de dépendances vers les classes avec le plus de dépendances) et descendant (en commençant par une classes ayant une ou plusieurs depandances).

Les classes atomiques qui ne dépendent d’aucune autre classe pour fonctionner dans notre code sont les classes ***PostfixGenerator*** et ***Number* (sans prendre en compte la methode Accept pour le moment)*.***

### PostfixGeneratorTest :

La figure suivante montre deux tests que l’on peut effectuer sur la la methode *GeneratePostfix*() de la class ***PostfixGenerator***.

Text

Description automatically generated

Figure PostfixGeneratorTest.cs

Graphical user interface, text, application

Description automatically generated

Figure 15 Resultat tests PosfixGeneratorTest.cs

La deuxième méthode a tester dans cette classe est *Calculate*() qui étant donné une expression arithmétique sous la forme postfix calcule la valeur de celle-ci. Cette méthode est référencée une fois par la classe ***Expression*** d’où l’association ***Expression/PostfixGenerator.*** Une fois effectue ce test nous permettra de commencer les tests de la classe ***Expression*** qui en dépend, sans se soucier du bon fonctionnement de ***PostfixGenerator***. La figure 16 montre un exemple de test a faire pour la méthode *Calculate()*. Maintenant que la méthode *GeneratePostfix*() a été testee, nous pouvons l’utiliser pour générer le postfix de test pour la methode *Calculate*(). On vois que les tests passent sans problème.

Text

Description automatically generated

Figure 16 Test Calculate() de la classe PostfixGenerator.cs

### Test Number.cs :

Les tests pour cette classe ne sont pas nécessaire car celle-ci possède seulement des attributs et des délégation de code (*Accept(IExpressionVisitor)* Délègue son comportement au visiteur donc pour la tester il suffit de tester un visiteur pour une instance de ***Number***) Figure 17.

Text

Description automatically generated

Figure Code source Number.cs