

PIF 6005 Travail I

KHEE23019700
KHELIFI, ELOUANES

ELOUANES.KHELIFI@UQTR.CA

Introduction :	2
Etape 1 (Test de classe SimpleCalculator) :	2
Partie I (testes manuelles) :	2
Partie II (Tests Automatiques) :	4
Conclusion (comparaison) :	8
Etape 2 (programme concret):	9
Ordre d'intégration :	9
Description détaillée des associations :	9
Génération des tests :	10
PostfixGeneratorTest :	10
Test Number.cs :	12
Test Expression.cs :	13
Test PrintVisitor et CalculateVisitor:	14
Test ExpressionBuilder :	15
Test Calculator :	17
Etape 3 (Tests modification) :	17
Modifications Mineurs et leurs impacts :	17
Changements Majeurs et leurs impacts :	19
Conclusion :	21

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 susmentionnée. 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 parties de ce travail.

NB : le but de ces tests n'est pas de tester de manière exhaustive les méthodes mais de mettre en évidence les techniques et stratégies de tests.

Etape 1 (Test de classe SimpleCalculator) :

Dans cette première étape, 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 tâche 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 au long de cette section.

```
namespace Calculatrice.Simple
{
    19 references | 0 changes | 0 authors, 0 changes
    public class SimpleCalculator
    {
        0 references | 0 changes | 0 authors, 0 changes
        public int Add(int _a, int _b) => _a + _b;
        0 references | 0 changes | 0 authors, 0 changes
        public int Sub(int _a, int _b) => _a - _b;
        0 references | 0 changes | 0 authors, 0 changes
        public int Mul(int _a, int _b) => _a * _b;
        1 reference | 0 changes | 0 authors, 0 changes
        public int Div(int _a, int _b)
        {
            return _a / _b;
        }
        1 reference | 0 changes | 0 authors, 0 changes
        public char Operation(int note)
        {
            if (note >= 90 && note <= 100) return 'A';
            if (note >= 70 && note < 90) return 'B';
            if (note >= 60 && note < 70) return 'C';
            if (note >= 50 && note < 60) return 'D';
            if (note >= 0 && note < 50) return 'F';

            throw new ArgumentException();
        }
    }
}
```

Figure 1 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.

```

namespace CalculatriceTests.Simple
{
    [TestClass]
    0 references | 0 changes | 0 authors, 0 changes
    internal class SimpleTests
    {
        SimpleCalculator calculator;

        [TestInitialize]
        0 references | 0 changes | 0 authors, 0 changes
        internal void Init()
        {
            calculator = new SimpleCalculator();
        }

        [TestMethod]
        0 references | 0 changes | 0 authors, 0 changes
        internal void TestDiv()
        {
            //preparation
            int _a = 4;
            int _b = 2;
            int _r = 2;
            //Assert
            Assert.AreEqual(_r, calculator.Div(_a, _b));
        }
    }
}

```

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]` définit les méthodes de test. Dans ces méthodes, nous commençons par préparer le jeu de test ainsi que le résultat attendu. Puis en utilisant la méthode **Assert.AreEqual(methodCall, result)** nous pouvons tester si la méthode retourne bien ce que l'on attends d'elle.

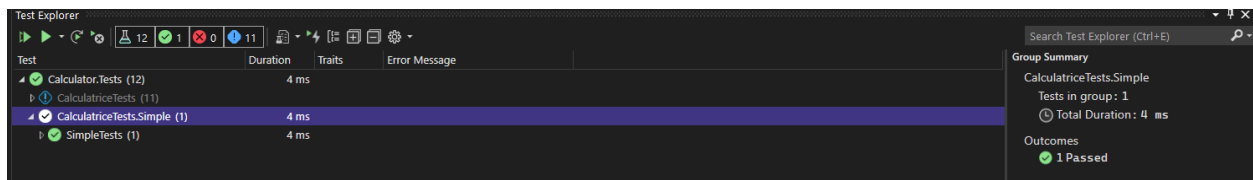


Figure 3 Test passe avec succès.

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 méthode `Div(int, int)` ne traite pas le cas de division par zéro.

Cet exemple illustre l'importance du choix de jeu de test lors de la génération de tests unitaires. En effet si notre jeu de tests ne couvre pas tous les cas limites. 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 à chaque itération. Alternativement, on pourrait aussi créer plusieurs méthodes de test. Avec un nouveau jeu de tests à 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 licences *Entreprise* de Visual Studio.

Cet outil facilite grandement la génération de tests 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 lesquels sont définies les classes à tester ciblent bien le .NetFramework. IntelliTest ne prend toujours pas en charge les projets .Net Core et .Net standard. Il faut aussi s'assurer que les projets ne sont pas destinés à des architectures x64 bit.

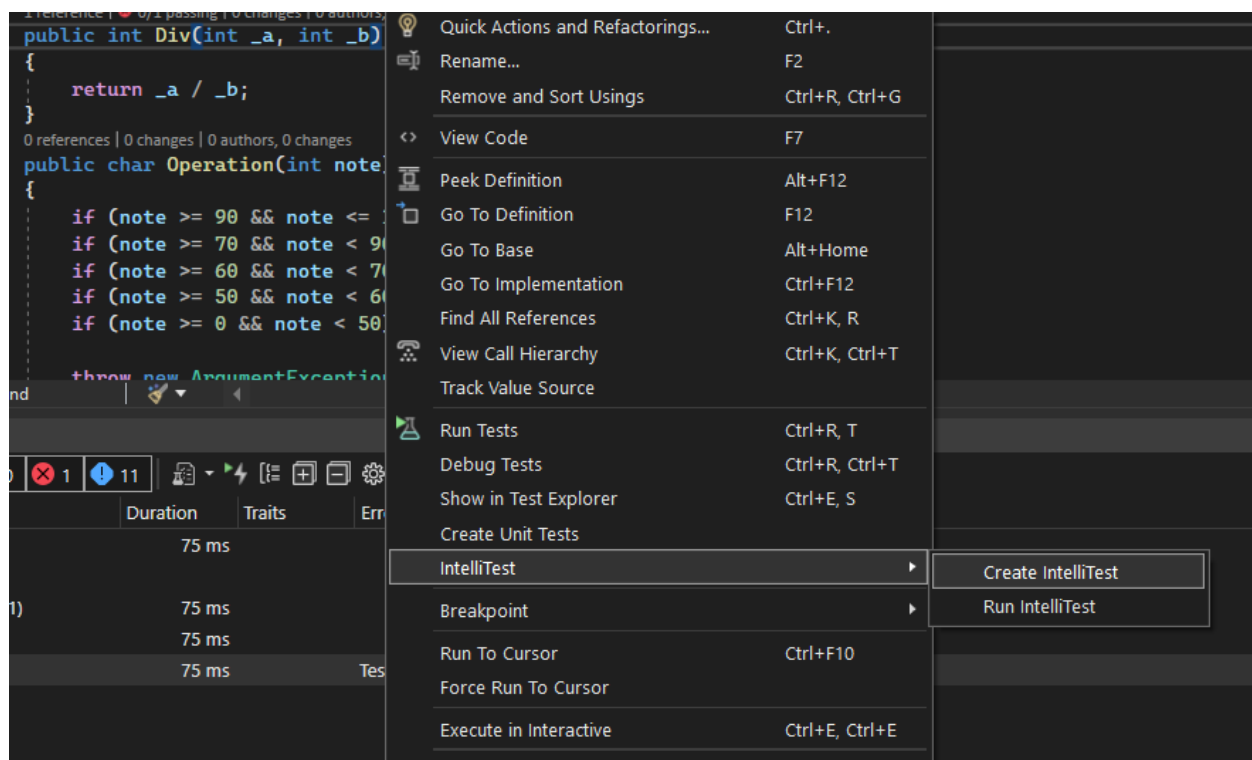


Figure 4 IntelliTest menu contextual

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

Figure 5 shows the 'Create IntelliTest' dialog box. The 'Test Framework' is set to 'MSTest'. The 'Test Project' is set to 'Calculator.Tests'. The 'Name Format for Test Project' is '[Project].Tests'. The 'Namespace' is '[Namespace].Tests'. The 'Name Format for Test Class' is '[Class]Test'. The 'Name Format for Test Method' is '[Method]Test'. There are 'OK' and 'Cancel' buttons at the bottom right.

Figure 5 Formulaire generation automatique de test

Sans trop aller dans les détails, le seul champ intéressant est le test framework qui définit quel extension de test utiliser. Pour cet exemple nous allons utiliser celui de microsoft appelé MSTest, mais intelliTest supporte plusieurs extensions comme UTest et NTest.

```

namespace Calculatrice.Simple.Tests
{
    /// <summary>This class contains parameterized unit tests for SimpleCalculator</summary>
    [TestClass]
    [PexClass(typeof(SimpleCalculator))]
    [PexAllowedExceptionFromTypeUnderTest(typeof(ArgumentException), AcceptExceptionSubtypes = true)]
    [PexAllowedExceptionFromTypeUnderTest(typeof(InvalidOperationException))]
    0 references | 0 changes | 0 authors, 0 changes
    public partial class SimpleCalculatorTest
    {
        /// <summary>Test stub for Div(Int32, Int32)</summary>
        [PexMethod]
        0 references | 0 changes | 0 authors, 0 changes
        public int DivTest(
            [PexAssumeUnderTest] SimpleCalculator target,
            int _a,
            int _b
        )
        {
            int result = target.Div(_a, _b);
            return result;
        }
    }
}

```

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 tous les tests de façon dynamique (à chaque exécution de cette classe une autre classe portant le même nom à l'exception d'un «.g.cs » à la fin, et contenant tous les

tests pour les branches 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.

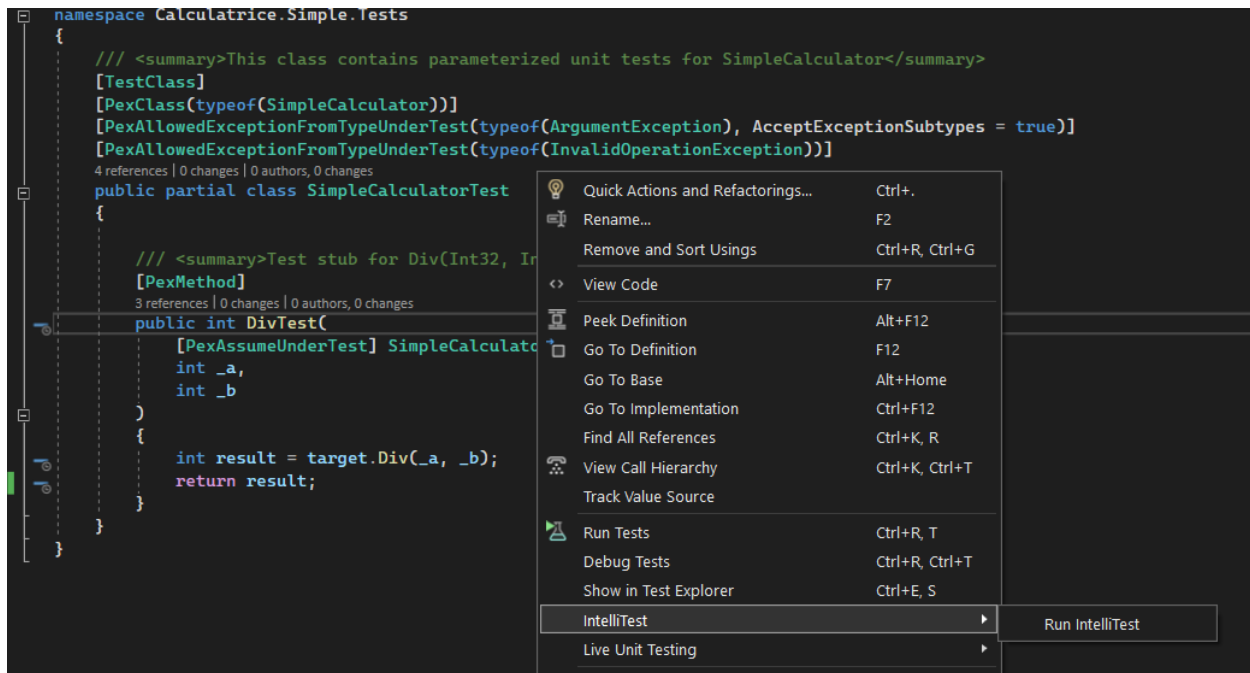


Figure 7 Execution de la classe generatrice via le menu run Intellitest.

SimpleCalculatorTest.DivTest(Simpl...							
1/1 blocks, 0/0 asserts, 2 runs							
	target	_a	_b	result(target)	result	Summary / Exception	Error Message
1	new Simple...	0	0	new Simple...	0	DivideByZeroException	Attempted to divide by zero.
2	new Simple...	0	1	new Simple...	0		
3	new Simple...	int.MinValue	-1			OverflowException	Arithmetic operation resulted in an overflow.

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 n'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.

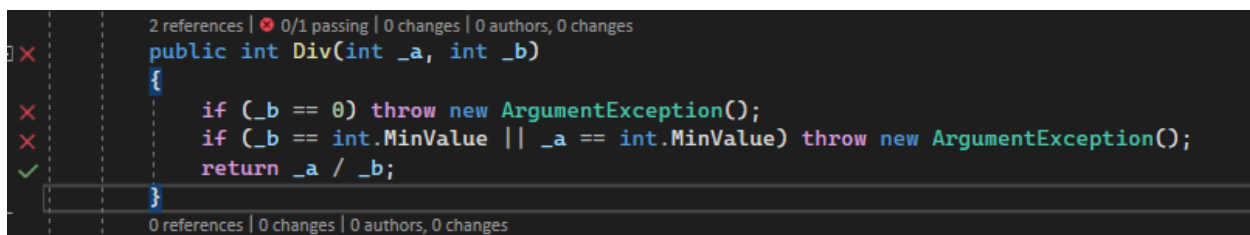


Figure 9 Code après traitement cas limite

IntelliTest Exploration Results - stopped

SimpleCalculatorTest.DivTest(Simpl Run 0 Warnings

4 0 9/9 blocks, 0/0 asserts, 4 runs

	target	_a	_b	result(target)	result	Summary / Exception	Error Message
1	new Simple...	0	0			ArgumentException	Value does not fall within the expected ran...
2	new Simple...	0	1	new Simple...	0		
3	new Simple...	int.MinValue	1			ArgumentException	Value does not fall within the expected ran...
4	new Simple...	0	int.MinValue			ArgumentException	Value does not fall within the expected ran...

Figure 10 Nouveau resultat de test

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

```
public partial class SimpleCalculatorTest
{
    [TestMethod]
    [PexGeneratedBy(typeof(SimpleCalculatorTest))]
    public void DivTest297()
    {
        int i;
        SimpleCalculator s0 = new SimpleCalculator();
        i = this.DivTest(s0, 0, 1);
        Assert.AreEqual<int>(0, i);
        Assert.IsNotNull((object)s0);
    }

    [TestMethod]
    [PexGeneratedBy(typeof(SimpleCalculatorTest))]
    [ExpectedException(typeof(ArgumentException))]
    public void DivTestThrowsArgumentException914()
    {
        int i;
        SimpleCalculator s0 = new SimpleCalculator();
        i = this.DivTest(s0, 0, 0);
    }

    [TestMethod]
    [PexGeneratedBy(typeof(SimpleCalculatorTest))]
    [ExpectedException(typeof(ArgumentException))]
    public void DivTestThrowsArgumentException794()
    {
        int i;
        SimpleCalculator s0 = new SimpleCalculator();
        i = this.DivTest(s0, int.MinValue, 1);
    }

    [TestMethod]
    [PexGeneratedBy(typeof(SimpleCalculatorTest))]
    [ExpectedException(typeof(ArgumentException))]
    public void DivTestThrowsArgumentException967()
    {
        int i;
        SimpleCalculator s0 = new SimpleCalculator();
        i = this.DivTest(s0, 0, int.MinValue);
    }
}
```

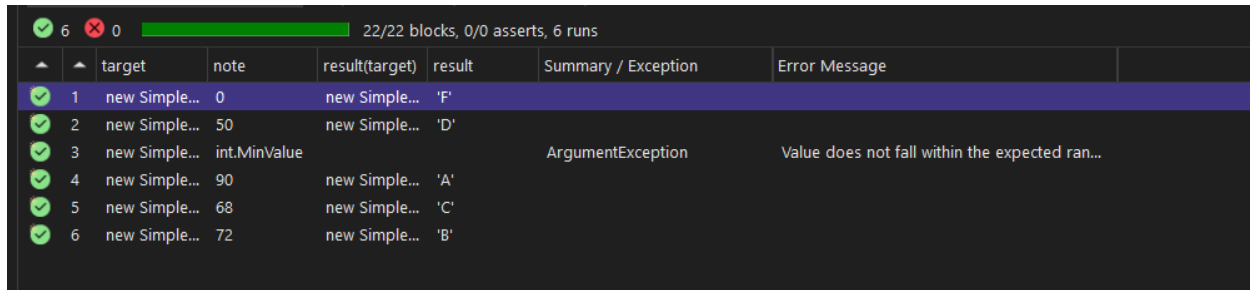
Figure 11 Tests generees par la classe SimpleCalculatorTest.cs

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.



	target	note	result(target)	result	Summary / Exception	Error Message
✓ 1	new Simple...	0	new Simple...	'F'		
✓ 2	new Simple...	50	new Simple...	'D'		
✓ 3	new Simple...	int.MinValue			ArgumentException	Value does not fall within the expected ran...
✓ 4	new Simple...	90	new Simple...	'A'		
✓ 5	new Simple...	68	new Simple...	'C'		
✓ 6	new Simple...	72	new Simple...	'B'		

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.

un calcul de plus en plus complexe de manière non exhaustive. Une expression peut être sous forme de chaîne 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 utilisée par **Expression** pour lire une expression sous forme de chaîne de caractères et la transformer à 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.

Étant 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 calcul 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 à des classes sans avoir à changer leurs codes. Tout en prenant compte du comportement différent de chacune.

C'est pourquoi l'interface **IExpression** oblige les classes qui l'implémentent à 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 calcul 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 tâche 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 à l'exécution des tests unitaires. Dans cette section nous allons tester nos classes de manière « Accendante », c'est-à-dire en commençant des classes tout à la fin de la chaîne de dépendances vers les classes avec le plus de dépendances.

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 méthode **Number.Accept()** pour le moment).

PostfixGeneratorTest :

La figure suivante montre deux tests que l'on peut effectuer sur la méthode **GeneratePostfix()** de la classe **PostfixGenerator**.

```

[TestMethod()]
0 references | 0 changes | 0 authors, 0 changes
public void GeneratePostfix_Test()
{
    var _test = "2 * 4 + 3 - 15 / 2";
    var _expct = "2 4 * 3 + 15 - 2 /";

    var _actual = String.Join(" ", PostfixGenerator.GeneratePostfix(_test).ToArray());

    Assert.AreEqual(_expct, _actual);
}

[TestMethod()]
0 references | 0 changes | 0 authors, 0 changes
public void GeneratePostfix_Test2()
{
    var _test = "2 / 4";
    var _expct = "2 4 /";

    var _actual = String.Join(" ", PostfixGenerator.GeneratePostfix(_test).ToArray());

    Assert.AreEqual(_expct, _actual);
}

```

Figure 14 PostfixGeneratorTest.cs

✓	Calculator.Tests (1...	14 ms
✓	Calculatrice (2)	12 ms
✓	PostfixGenerat...	12 ms
✓	GeneratePos...	6 ms
✓	GeneratePos...	6 ms

Figure 15 Resultat tests PosfixGeneratorTest.cs

La deuxième méthode à 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 effectuée 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 à faire pour la méthode *Calculate()*. Maintenant que la méthode *GeneratePostfix()* a été testée, nous pouvons l'utiliser pour générer le postfix de test pour la méthode *Calculate()*. On voit que les tests passent sans problème.

```

[TestMethod()]
0 references | 0 changes | 0 authors, 0 changes
public void Calculate_Test()
{
    var _test = PostfixGenerator.GeneratePostfix("2 / 4 * 4");
    var _expct = 8;
    var _actual = PostfixGenerator.Calculate(_test);

    Assert.AreEqual(_expct, _actual);
}

[TestMethod()]
0 references | 0 changes | 0 authors, 0 changes
public void Calculate_Test2()
{
    var _test = PostfixGenerator.GeneratePostfix("2 * 4 + 4 * 2");
    var _expct = 24;
    var _actual = PostfixGenerator.Calculate(_test);

    Assert.AreEqual(_expct, _actual);
}

```

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.

```

8  namespace Calculatrice.Expressions
9  {
10     7 references | LoneX, 7 days ago | 1 author, 1 change
11     public class Number : IExpression
12     {
13         3 references | LoneX, 7 days ago | 1 author, 1 change
14         private int Value { get; set; }
15
16         3 references | LoneX, 7 days ago | 1 author, 1 change
17         public Number(int a) { Value = a; }
18
19         4 references | LoneX, 7 days ago | 1 author, 1 change
20         public int Calculate() => Value;
21
22         2 references | LoneX, 7 days ago | 1 author, 1 change
23         public void Accept(IExpressionVisitor visitor) => visitor.Visit(this);
24
25         3 references | LoneX, 7 days ago | 1 author, 1 change
26         public string ToString(bool parenthesis) => Value.ToString();
27     }
28 }

```

Figure 17 Code source Number.cs

Test Expression.cs :

Toutes les dépendances de la classe **Expression** sont maintenant testées. On peut maintenant aller tester celle-ci sans se soucier du bon fonctionnement des classes qui lui sont associée.

```
11 [TestClass()]
12 0 references | 0 changes | 0 authors, 0 changes
13 public class ExpressionTests
14 {
15     Expression expression;
16     [TestInitialize()]
17     0 references | 0 changes | 0 authors, 0 changes
18     public void Init()
19     {
20         expression = new Expression(1);
21     }
22     [TestMethod()]
23     0 references | 0 changes | 0 authors, 0 changes
24     public void Calculate_Test()
25     {
26         var _operand = new Number(1);
27         var _operator = "+";
28         expression.Push(_operand, _operator);
29
30         var _expct = 2;
31         var _actual = expression.Calculate();
32
33         Assert.AreEqual(_expct, _actual);
34     }
35
36     [TestMethod()]
37     0 references | 0 changes | 0 authors, 0 changes
38     public void Calculate_Test_complex()
39     {
40         var _operand1 = new Number(4);
41         var _operand2 = new Number(5);
42         var _operator1 = "+";
43         var _operator2 = "-";
44
45         expression.Push(_operand1, _operator1);
46         expression.Push(_operand2, _operator2);
47
48         var _expct = 0;
49         var _actual = expression.Calculate();
50
51         Assert.AreEqual(_expct, _actual);
52     }
53 }
```

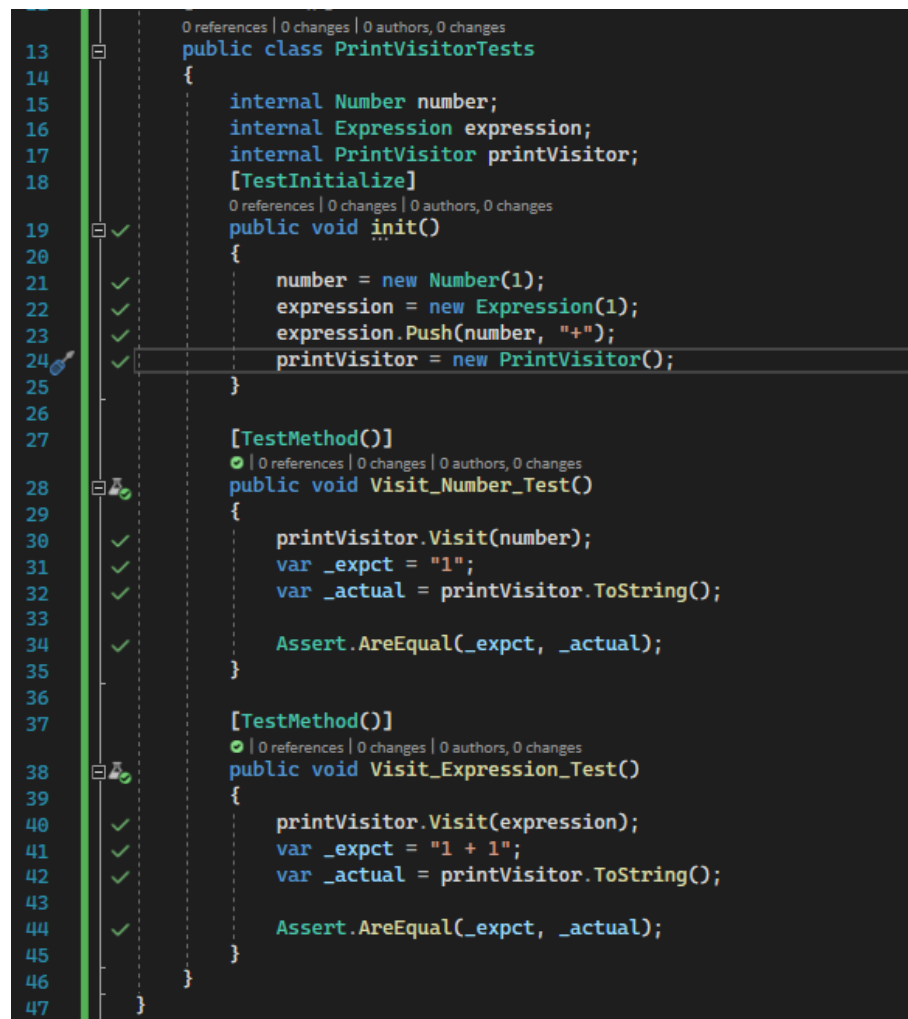
Figure 18 Test Calculate() de Expression.cs

La méthode *ToString()* de **Expression** prend un paramètre « *bool parenthesis* » qui définit si oui ou non il s'agit d'une sous expression ou pas. Si oui, il met celle-ci entre parenthèses. Pour garder les tests assez courts. On essaye les deux cas dans un seul test par plusieurs *Asserts*.

NB : la méthode *Push()* et *Accept()* ne vont pas être testées car elles délèguent leurs comportements à d'autres classes (respectivement **List** et les **IExpressionVisitor**).

Test PrintVisitor et CalculateVisitor:

Ces deux classes sont de type **IExpressionVisitor** qui ajoute des fonctionnalités à **Expression** et à **Number**. Ces deux classes ayant déjà été testées, nous pouvons dorénavant les utiliser pour tester tous les **IExpressionVisitor**. La figure suivante représente les tests de **PrintVisitor** et de **CalculateVisitor** respectivement.



```
13 public class PrintVisitorTests
14 {
15     internal Number number;
16     internal Expression expression;
17     internal PrintVisitor printVisitor;
18     [TestInitialize]
19     public void init()
20     {
21         number = new Number(1);
22         expression = new Expression(1);
23         expression.Push(number, "+");
24         printVisitor = new PrintVisitor();
25     }
26
27     [TestMethod]
28     public void Visit_Number_Test()
29     {
30         printVisitor.Visit(number);
31         var _expct = "1";
32         var _actual = printVisitor.ToString();
33
34         Assert.AreEqual(_expct, _actual);
35     }
36
37     [TestMethod]
38     public void Visit_Expression_Test()
39     {
40         printVisitor.Visit(expression);
41         var _expct = "1 + 1";
42         var _actual = printVisitor.ToString();
43
44         Assert.AreEqual(_expct, _actual);
45     }
46 }
47 }
```

Figure 19 Tests Print Visitor pour number et expression

```

13 public class CalculateVisitorTests
14 {
15     internal Number number;
16     internal Expression expression;
17     internal CalculateVisitor calculateVisitor;
18     [TestInitialize]
19     0 references | 0 changes | 0 authors, 0 changes
20     public void init()
21     {
22         number = new Number(1);
23         expression = new Expression(1);
24         expression.Push(number, "+");
25         calculateVisitor = new CalculateVisitor();
26     }
27
28     [TestMethod()]
29     0 references | 0 changes | 0 authors, 0 changes
30     public void Visit_Number_Test()
31     {
32         calculateVisitor.Visit(number);
33         var _expct = 1;
34         var _actual = calculateVisitor.Result;
35
36         Assert.AreEqual(_expct, _actual);
37     }
38
39     [TestMethod()]
40     0 references | 0 changes | 0 authors, 0 changes
41     public void Visit_Expression_Test()
42     {
43         calculateVisitor.Visit(expression);
44         var _expct = 2;
45         var _actual = calculateVisitor.Result;
46
47         Assert.AreEqual(_expct, _actual);
48     }
49 }

```

Figure 20 CalculateVisitor Tests avec Number et Expression

Test ExpressionBuilder :

Cette class suis le patron *builder* pour créer des expressions de type **Expression** pour tester ses méthodes, il suffit de comparer l'expression créée en utilisant le builder a une autre cree manuellement en utilisant des méthodes incluses dans **Expression** qui ont déjà été testés.

NB : pour éviter de trop charger ce document je ne vais pas tester toutes les méthodes de builder mais plutôt la méthode qui est au cœur de toutes les autres : (*Operate()*) qui ajoute une opération a l'expression en cours de création. Toutes les autres méthodes seraient testés suivant le même principe (comparaison avec une création manuelle de l'expression).

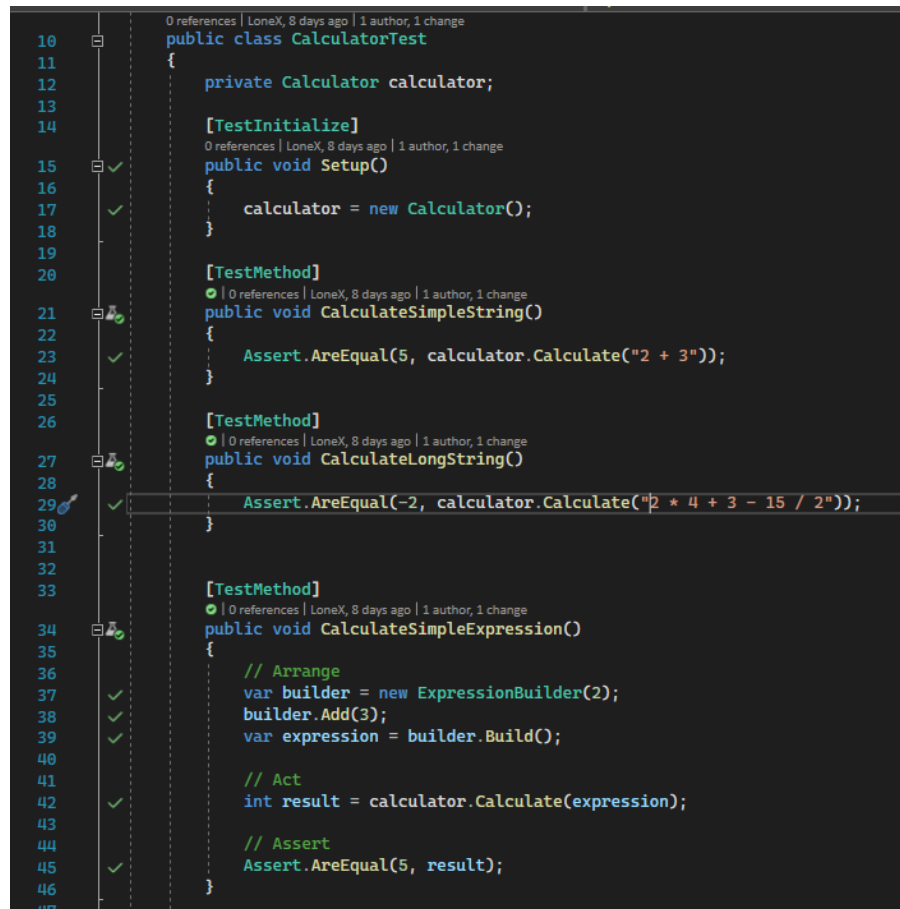

```

12 0 references | 0 changes | 0 authors, 0 changes
13 public class ExpressionBuilderTests
14 {
15     public ExpressionBuilder builder;
16
17     [TestInitialize]
18     0 references | 0 changes | 0 authors, 0 changes
19     public void Init()
20     {
21         builder = new ExpressionBuilder(1);
22     }
23
24     [TestMethod]
25     0 references | 0 changes | 0 authors, 0 changes
26     public void Operate_number_Test()
27     {
28         var _n = new Number(1);
29         //actual expression is built with the builder
30         builder.Operate(_n.Calculate(), "+");
31         var _actual = builder.Build();
32
33         //expcted expression is built manually via already tested methods
34         var _expct = new Expression(1);
35         _expct.Push(_n, "+");
36
37         Assert.AreEqual(_actual.ToString(), _expct.ToString());
38     }
39
40     [TestMethod]
41     0 references | 0 changes | 0 authors, 0 changes
42     public void Operate_builder_Test()
43     {
44         var _n = new Number(1);
45         //actual is built with the builder
46         builder.Operate(builder, "+");
47         var _actual = builder.Build();
48         var _expct = new Expression(1);
49         _expct.Push(_n, "+");
50
51         Assert.AreEqual(_actual.ToString(), _expct.ToString());
52     }
53 }

```

Figure 21 Test de la classe ExpressionBuilder.cs

Test Calculator :



```
10 public class CalculatorTest
11 {
12     private Calculator calculator;
13
14     [TestInitialize]
15     public void Setup()
16     {
17         calculator = new Calculator();
18     }
19
20     [TestMethod]
21     public void CalculateSimpleString()
22     {
23         Assert.AreEqual(5, calculator.Calculate("2 + 3"));
24     }
25
26     [TestMethod]
27     public void CalculateLongString()
28     {
29         Assert.AreEqual(-2, calculator.Calculate("2 * 4 + 3 - 15 / 2"));
30     }
31
32     [TestMethod]
33     public void CalculateSimpleExpression()
34     {
35         // Arrange
36         var builder = new ExpressionBuilder(2);
37         builder.Add(3);
38         var expression = builder.Build();
39
40         // Act
41         int result = calculator.Calculate(expression);
42
43         // Assert
44         Assert.AreEqual(5, result);
45     }
46 }
```

Figure 22 CalculatorTests.cs

Maintenant que tous les composants testables de notre système ont été testés. On peut enfin tester la classe façade qui utilise toutes les autres classes pour implémenter le fonctionnement désiré de notre programme. La figure suivante montre un échantillon de test pour la classe. Et le reste des tests implémentés pour cette classe sont accessibles dans le fichier CalculatorTests.cs (ce sont les vôtres 😊).

Etape 3 (Tests modification) :

Dans cette section nous allons effectuer quelques changements sur notre code afin de mesurer l'impact que cela inflige à nos tests unitaires.

Modifications Mineurs et leurs impacts :

Les changements effectués dans cette sous-section sont supposés avoir un impact mineur sur la conception du programme.

Ajouter un attribut peut signifier la création d'une nouvelle association. Dans ce cas, la classe nouvellement associée doit être testée avant de valider le test de quelque méthode qui dépend de cette dernière.

```

private Number number;

0 references | 0 changes | 0 authors, 0 changes
private string Calculate_Number(IEnumerable<string> postfix)
{
    var _result = Calculate(postfix);
    number = new Number(_result);

    return number.ToString(true);
}

```

Figure 23 Ajout d'une methode ainsi qu'un attribut a la classe PostfixGenerator.cs

L'ajout de cet attribut (Figure 23) génère une nouvelle association entre **PostfixGenerator** et **Number**. Dans la section (**Etape2**) nous aurions pu tester la classe **PostfixGenerator** en premier (avant **Number**). Maintenant qu'il y a une relation de dépendance ces deux classes, nous devons respecter le nouvel ordre d'intégration en commençant par **Number**. Mais celle-ci n'a pas changée donc pour notre cas, cela ne sera pas nécessaire.

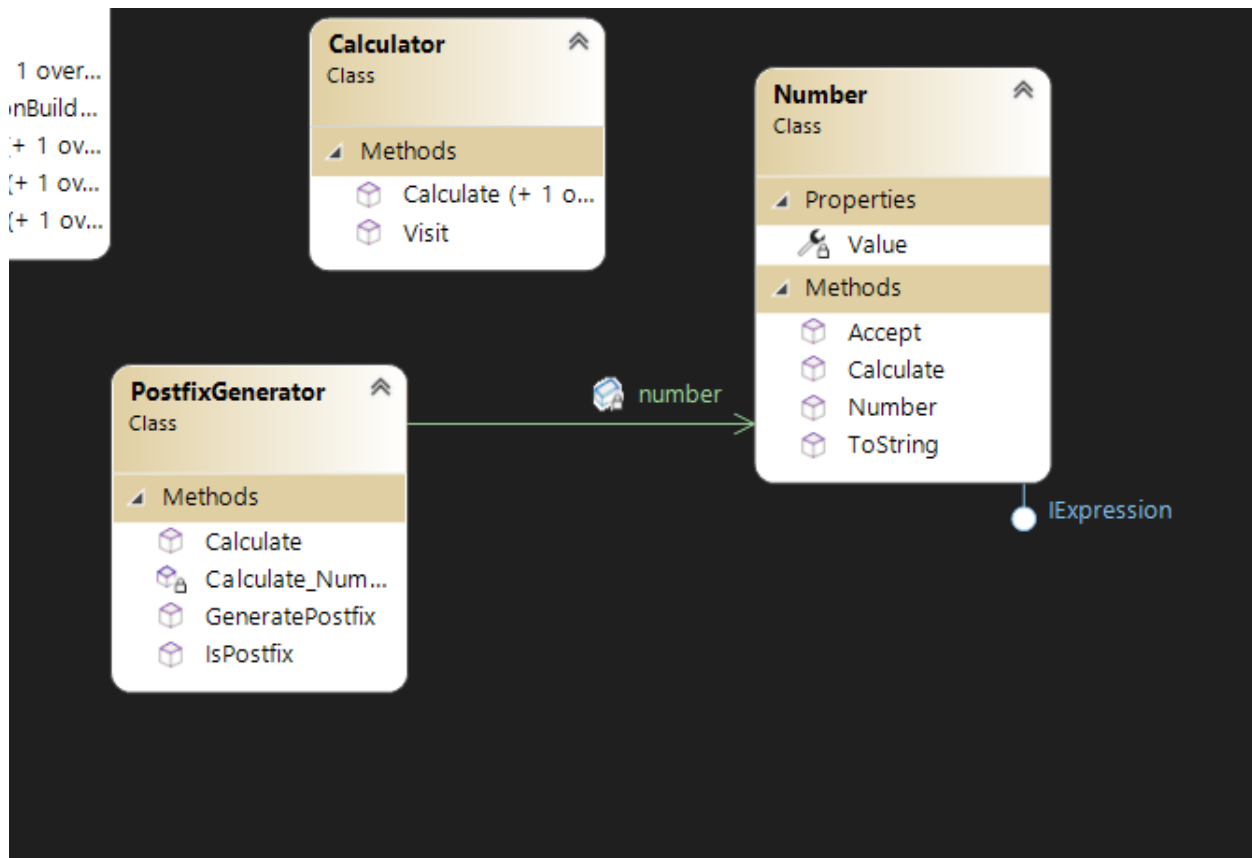


Figure 24 Nouvelle association issue de l'ajout d'un attribut de type Number.

L'ajout de la méthode **CalculateNumber()** dont le rôle est de donner le résultat d'une expression postfix sous forme de chaîne de caractères entre parenthèses, nécessite de nouveaux tests. La méthode **toString()** de **Number** qui a déjà été testée. Et un test unitaire de la méthode elle-même (**CalculateNumber()**). La figure suivante montre les tests ajoutés pour valider le bon fonctionnement de cette méthode.

```

[TestMethod()]
0 references | 0 changes | 0 authors, 0 changes
public void CalculateNumber_Test()
{
    var _test = PostfixGenerator.GeneratePostfix("5 + 5 + 1");
    var _expct = "(11)";
    var _actual = PostfixGenerator.CalculateNumber(_test);

    Assert.AreEqual(_expct, _actual);
}

```

Figure 25 Test de la methode Calculate number de PostfixGenerator.cs

Changements Majeurs et leurs impacts :

Nous allons ajouter une fonctionnalité a notre programme qui sert à analyser une expression pour dire s'il s'agit bien d'un postfix. Cette fonctionnalité touche la classe façade **Calculator** , **PostfixGenerator** et **PrintVisitor**. La figure 26 et 27 montrent le code ajouter dans ces deux classes pour implémenter cette fonctionnalité.

```

1 reference | 1/1 passing | 0 changes | 0 authors, 0 changes
public bool IsPostfix(IExpression expr)
{
    if (expr is Number) return true;

    var _visitor = new PrintVisitor();
    _visitor.Visit(expr as Expression);
    var _strExpr = _visitor.ToString();

    var _postfix = PostfixGenerator.GeneratePostfix(_strExpr);
    return PostfixGenerator.IsPostfix(_postfix);
}

```

Figure 26 IsPostfix at Calculator.cs

```

[TestMethod()]
0 references | 0 changes | 0 authors, 0 changes
public void IsPostfix_Test_true()
{
    var _test = PostfixGenerator.GeneratePostfix("5 + 5 + 1");
    Assert.IsTrue(PostfixGenerator.IsPostfix(_test));
}

```

Figure 27 Tests is postfix() at PostfixGenerator.cs

```

[TestMethod]
0 references | 0 changes | 0 authors, 0 changes
public void IsPostfixTest()
{
    var builder = new ExpressionBuilder(2);
    builder.Add(3);
    builder.Add(4);
    var expression = builder.Build();

    Assert.IsTrue(calculator.IsPostfix(expression));
}

```

Figure 28 Test Is postfix at Calculator.cs

Pour tester cette nouvelle fonctionnalité, étant donné que celle-ci définit une association entre *calculator* et *PrintVisitor*, et une autre entre *Calculator* et *PostfixGenerator* ses classes doivent être testées avant de pouvoir tester la nouvelle fonctionnalité (*PrintVisitor* l'est déjà).

```

2 references | 1/1 passing | 0 changes | 0 authors, 0 changes
public static bool IsPostfix(IEnumerable<String> input)
{
    var _it = 0;
    var _lastWasNum = false;
    foreach(var token in input)
    {
        if (int.TryParse(token, out var _))
        {
            if (_it > 1 && _lastWasNum) return false;
            _lastWasNum = true;
        }
        else
        {
            if (!_lastWasNum) return false;
            _lastWasNum = false;
            if (_it == 0) return false;
        }
        _it++;
    }

    return true;
}

```

Figure 29 isPostfix at PostfixGenerator.cs

D'un point de vue **quantitatif** l'ajout de code avec de nouvelles branches peut entraîner (comme c'est le cas pour nous) de pertes en termes de score de couverture. Nous pouvons mesurer ce métrique dans Visual Studio en allant vers Tests/Analyse code coverage for all tests. Les figures suivantes montrent l'impact qu'a eu l'ajout de la méthode IsPostfix() (qui a plusieurs complexité cyclomatique supérieure) sur le score de couverture pour les tests de cette méthode. C'est du au fait que notre jeu de tests actuelles ne couvre pas la totalité des cas traitées dans la méthode.

kheli_J3-RAPTOR 2022-03-19 ...	44	10.14%	390	89.86%
calculator.dll	41	18.30%	183	81.70%
Calculatrice	2	3.33%	58	96.67%
Calculator	0	0.00%	8	100.00%
PostfixGenerator	2	3.85%	50	96.15%
Calculatrice.Expressions	8	7.34%	101	92.66%

Figure 30 PostfixGenerator.cs coverage score Avant isPostfix()

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
kheli_J3-RAPTOR 2022-03-19 ...	48	9.96%	434	90.04%
calculator.dll	45	17.51%	212	82.49%
Calculatrice	6	6.45%	87	93.55%
Calculator	1	5.88%	16	94.12%
PostfixGenerator	5	6.58%	71	93.42%
Calculatrice.Expressions	8	7.34%	101	92.66%

Figure 31 PostfixGenerator.cs coverage score apres ajout de ispostfix()

Conclusion :

Les relations entre les objets jouent un rôle crucial dans l'élaboration de tests, il est clair que l'on ne doit pas baser nos tests sur des méthodes dont le comportement dépend d'autres classes. Car nos tests pourraient passer pour un certain jeu d'instructions mais il est possible qu'il y ait un certain jeu d'instruction qui provoquerait une erreur dans la classe de laquelle notre méthode testée dépend. Cela dit, on sera amené à prendre en compte les cas limite de cette classe pour chaque méthode que l'on veut tester (ce qui n'est pas pratique). C'est pourquoi on préfère suivre une stratégie (ascendante ou descendante) pour tester les classes desquels nos tests dépendent. Des techniques de génération de tests sont aussi à notre disposition pour cela.