

Test Driven Development (TDD)

Tutoriel

version 1.0

Hans W. Schwendimann

`hws@idec.ch`

mars 2019

1 Introduction

TDD (Test-Driven Development) est un processus de développement logiciel qui repose sur la répétition d'un cycle de développement très court :

- le développeur écrit d'abord un cas de test automatisé (initialement défaillant) qui définit une amélioration ou une nouvelle fonction, puis génère le minimum de code pour passer ce test ...
- enfin le développeur factorise le nouveau code à des normes acceptables ...

Les cinq étapes suivantes sont généralement suivies :

1. écrire un test,
2. exécuter le test et vérifier qu'il échoue,
3. écrire le code **suffisant** pour que le test passe,
4. vérifier que le test passe,
5. re-factoriser le code et vérifier qu'il n'y ait pas de régression,

Pour simplifier cette logique on peut regrouper ces cinq étapes en trois grandes idées :

- **Tester d'abord**, qui correspond aux deux premières étapes.
- **Rendre fonctionnel**, qui englobe les points 3 et 4.
- **Rendre meilleur**, qui n'est autre que l'étape 5.

Commencer par vous imposer des pratiques de développement rigoureuses est impératif mais les tests seront un complément indispensable à la qualité de votre code.

2 Créer un calculateur simple - Fonctionnalités

L'objectif de ce tutoriel est de réaliser une calculatrice **String** simple en ligne de commande en utilisant le processus de développement logiciel **TDD**.

Liste d'exigences de notre calculatrice :

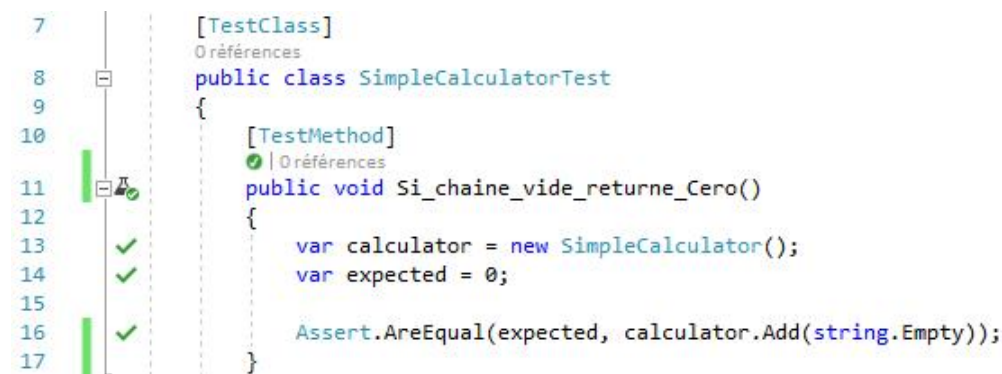
1. créer une calculatrice string simple avec une méthode `int Add(string s)`
2. la méthode peut prendre 0, 1 ou 2 nombres, et retournera leur somme (pour une chaîne vide, elle retournera 0) par exemple "" ou "1" ou "1,2".
3. autoriser la méthode Add à gérer un nombre inconnu de nombres.
4. autoriser la méthode Add à gérer les retours à la ligne entre les nombres (au lieu de virgules).
5. l'entrée suivante est ok : "1\n2,3" (sera égal à 6).
6. Supporte différents délimiteurs.
7. Pour changer un délimiteur, le début de la chaîne contiendra une ligne distincte qui ressemble à ceci : "// [délimiteur] \n [nombres ...]" par exemple "//;\n1; 2" devrait retourner trois où le délimiteur par default est ','.
8. La première ligne est facultative. Tous les scénarios existants doivent toujours être pris en charge
9. Appeler la méthode Add() avec un nombre négatif lèvera une exception "négatifs non autorisés" - et le négatif qui a été passé. S'il y a plusieurs négatifs, montrez-les tous dans le message d'exception.
10. Les nombres supérieurs à 1000 doivent être ignorés, ce qui ajoute $2 + 1001 = 2$.
11. Les délimiteurs peuvent être de n'importe quelle longueur avec le format suivant : "// [délimiteur] n" par exemple : "// [-] n1-2-3" devrait retourner 6
12. Autoriser plusieurs délimiteurs comme ceci : "// [delim1] [delim2] n" par exemple "// [-] [n1-2

3 Créer un calculateur simple - Développement

3.1 Itération 1

Exigence 2a : Pour une chaîne vide, la méthode retourne 0.

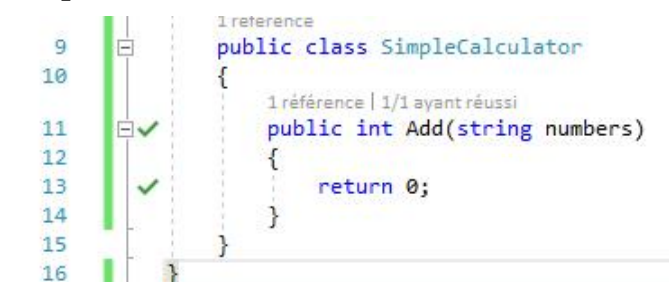
-Test :



```
[TestClass]
0 références
public class SimpleCalculatorTest
{
    [TestMethod]
    0 références
    public void Si_chaine_vide_returne_Cero()
    {
        var calculator = new SimpleCalculator();
        var expected = 0;

        Assert.AreEqual(expected, calculator.Add(string.Empty));
    }
}
```

-Implémentation :

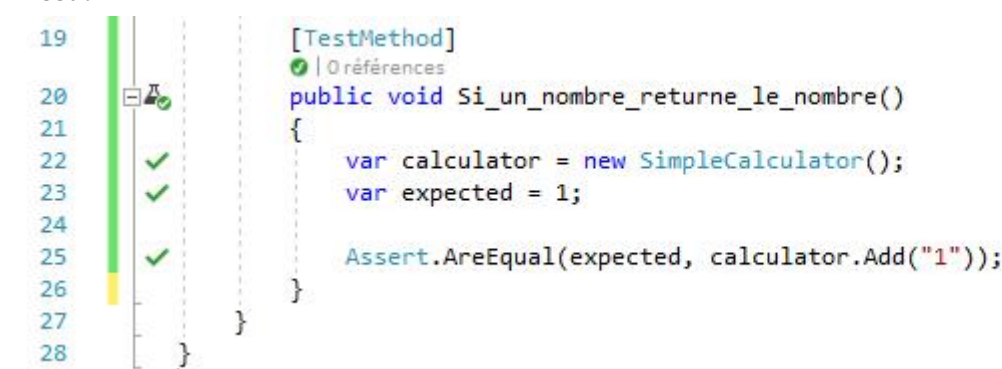


```
1 référence
public class SimpleCalculator
{
    1 référence | 1/1 ayant réussi
    public int Add(string numbers)
    {
        return 0;
    }
}
```

3.2 Itération 2

Exigence 2b : Si un seul nombre, la méthode retourne le nombre.

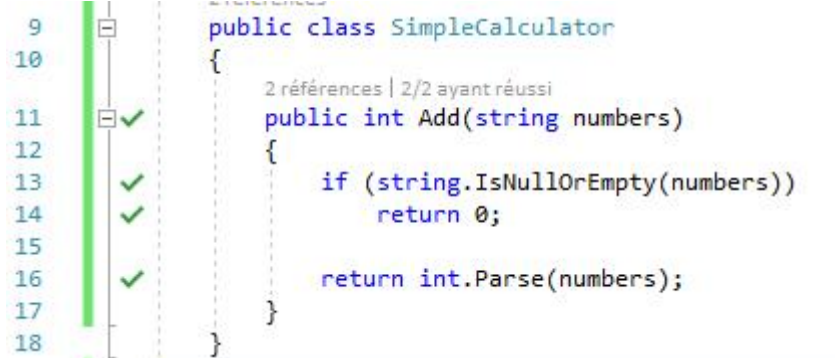
-Test :



```
[TestMethod]
0 références
public void Si_un_nombre_returne_le_nombre()
{
    var calculator = new SimpleCalculator();
    var expected = 1;

    Assert.AreEqual(expected, calculator.Add("1"));
}
```

-Implémentation :

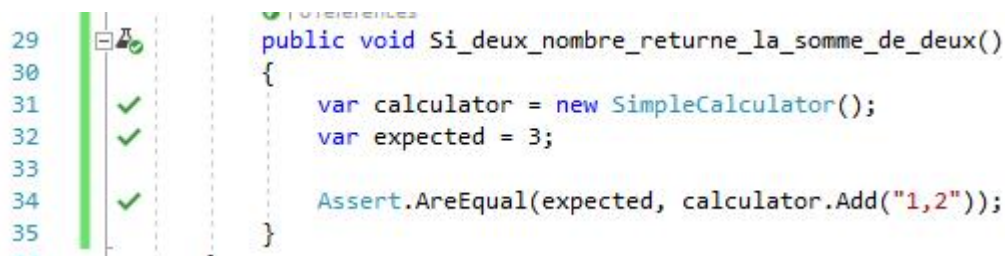


```
9 public class SimpleCalculator
10 {
11     public int Add(string numbers)
12     {
13         if (string.IsNullOrEmpty(numbers))
14             return 0;
15
16         return int.Parse(numbers);
17     }
18 }
```

3.3 Itération 3

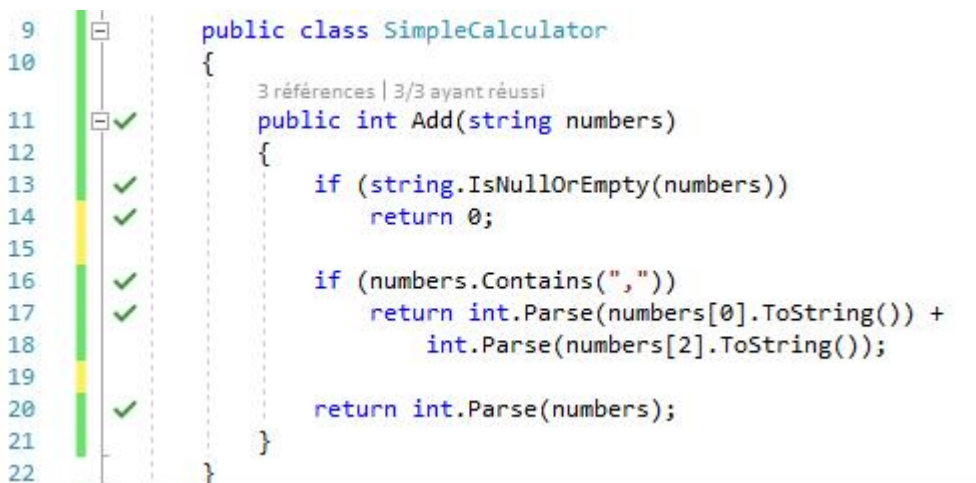
Exigence 2c : La méthode peut prendre 0, 1 ou 2 nombres séparés par une virgule (,).

-Test :



```
29 public void Si_deux_nombre_returne_la_somme_de_deux()
30 {
31     var calculator = new SimpleCalculator();
32     var expected = 3;
33
34     Assert.AreEqual(expected, calculator.Add("1,2"));
35 }
```

-Implémentation :



```
9 public class SimpleCalculator
10 {
11     public int Add(string numbers)
12     {
13         if (string.IsNullOrEmpty(numbers))
14             return 0;
15
16         if (numbers.Contains(","))
17             return int.Parse(numbers[0].ToString()) +
18                 int.Parse(numbers[2].ToString());
19
20         return int.Parse(numbers);
21     }
22 }
```

3.4 Itération 4

Exigence 3 : La méthode peut prendre plusieurs nombres.

-Test :

```
37 [TestMethod]
38 | 0 références
39 public void Si_plusieurs_nombres_returne_la_somme_de_tous_les_nombres()
40 {
41     var calculator = new SimpleCalculator();
42     var expected = 6;
43     Assert.AreEqual(expected, calculator.Add("1,2,3"));
44 }
```

-Implémentation :

```
9 public class SimpleCalculator
10 {
11     4 références | 4/4 ayant réussi
12     public int Add(string numbers)
13     {
14         if (string.IsNullOrEmpty(numbers))
15             return 0;
16
17         if (numbers.Contains(","))
18         {
19             return numbers.Split(',').Select(x => int.Parse(x)).Sum();
20         }
21         return int.Parse(numbers);
22     }
```

3.5 Itération 5

Refactorisation :

```
8 public class SimpleCalculatorTest
9 {
10     [TestMethod]
11     0 références
12     public void Si_chaine_vide_returne_Cero()
13     {
14         string.Empty.ShouldCalculateTo(0);
15     }
16
17     [TestMethod]
18     0 références
19     public void Si_un_nombre_returne_le_nombre()
20     {
21         "1".ShouldCalculateTo(1);
22     }
23
24     [TestMethod]
25     0 références
26     public void Si_deux_nombre_returne_la_somme_de_deux()
27     {
28         "1,2".ShouldCalculateTo(3);
29     }
30
31     [TestMethod]
32     0 références
33     public void Si_plusieurs_nombres_returne_la_somme_de_tous_les_nombres()
34     {
35         "1,2,3".ShouldCalculateTo(6);
36     }
37 }
38
39 0 références
40 internal static class TestHelper
41 {
42     4 références | 4/4 ayant réussi
43     public static void ShouldCalculateTo(this string input, int expected)
44     {
45         var calculator = new SimpleCalculator();
46
47         Assert.AreEqual(expected, calculator.Add(input));
48     }
49 }
```

3.6 Itération 6

Exigences 4-5 : La méthode autorise le retour à la ligne comme séparateur.

-Test :

```
34 [TestMethod]
35 | 0 références
36 public void Si_return_a_la_ligne_comme_separateur_returne_la_somme_de_tous_les_nombres()
37 {
38     "1\n2\n3".ShouldCalculateTo(6);
}
```

-Implémentation :

```
9 public class SimpleCalculator
10 {
11     1 référence
12     public int Add(string numbers)
13     {
14         if (string.IsNullOrEmpty(numbers))
15             return 0;
16         return numbers.Split(',', '\n').Select(x => int.Parse(x)).Sum();
17     }
18 }
```

3.7 Itération 7

Exigence 6 : Plusieurs séparateurs consécutifs lèvent une exception.

-Test :

```
39 [TestMethod]
40 [ExpectedException(typeof(ArgumentException))]
41 | 0 références
42 public void Si_deux_separateurs_consecutifs_returne_exception()
43 {
44     ",,".ShouldCalculateTo(-1);
45 }
46
```

-Implémentation :

```
9 public class SimpleCalculator
10 {
11     1 référence
12     public int Add(string numbers)
13     {
14         var delimiters = ",\n";
15         if (string.IsNullOrEmpty(numbers))
16             return 0;
17
18         var items = numbers.Split(delimiters.ToCharArray());
19
20         if (items.Any(x => string.IsNullOrEmpty(x)))
21             throw new ArgumentException();
22
23         return items.Select(x => int.Parse(x)).Sum();
24     }
25 }
```

3.8 Itération 8

Exigences 6-7 : La méthode autorise différents séparateurs personnalisés.

-Test :

```
46
47 [TestMethod]
48 0 références
49 public void Si_separateur_personalise_retourne_la_somme_de_tous_les_nombres()
50 {
51     "//;\n1;2".ShouldCalculateTo(3);
52 }
```


-Implémentation :

```
9      public class SimpleCalculator
10     {
11         1 référence
12         public int Add(string numbers)
13         {
14             var delimiters = ",\n";
15             if (string.IsNullOrEmpty(numbers))
16                 return 0;
17
18             if(numbers.Contains("//"))
19             {
20                 delimiters += numbers[2];
21                 numbers = numbers.Substring(4, numbers.Length - 4);
22             }
23
24             var items = numbers.Split(delimiters.ToCharArray());
25
26             if (items.Any(x => string.IsNullOrEmpty(x)))
27                 throw new ArgumentException();
28
29             return items.Select(x => int.Parse(x)).Sum();
30         }
31     }
```

3.9 Itération 9

Exigence 8 : La première ligne est facultative.

Voir test antérieurs...

3.10 Itération 10

Exigence 9 : Appeler la méthode avec un nombre négatif lèvera une exception

-Test :

```
53     [TestMethod]
54     [ExpectedException(typeof(ArgumentOutOfRangeException))]
55     0 références
56     public void Si_un_nombre_negatif_est_donnee_retourne_un_exception()
57     {
58         "-1,2".ShouldCalculateTo(1);
59     }
```

-Implémentation :

```
11  public int Add(string numbers)
12  {
13      var delimiters = ",\n";
14
15      if (string.IsNullOrEmpty(numbers))
16          return 0;
17
18      if(numbers.Contains("//"))
19      {
20          delimiters += numbers[2];
21          numbers = numbers.Substring(4, numbers.Length - 4);
22      }
23
24      var items = numbers.Split(delimiters.ToCharArray());
25
26      if (items.Any(x => string.IsNullOrEmpty(x)))
27          throw new ArgumentException();
28
29      var integers = items.Select(x => int.Parse(x));
30      if (integers.Any(x => x < 0))
31          throw new ArgumentOutOfRangeException();
32
33      return items.Select(x => int.Parse(x)).Sum();
34  }
35  }
```

3.11 Itération 11

Exigence 9b : Le message de l'exception des nombres négatifs doit contenir le ou les nombres négatifs passés en paramètre.

-Test :

```
60  [TestMethod]
61  public void Message_de_erreur_doit_contenir_les_nombres_negatifs()
62  {
63      try
64      {
65          new SimpleCalculator().Add("-1,2");
66      }
67      catch (ArgumentOutOfRangeException ex)
68      {
69          Assert.IsTrue(ex.Message.Contains("-1"));
70      }
71  }
```

-Implémentation :

```
9 public class SimpleCalculator
10 {
11     4 références | 3/3 ayant réussi
12     public int Add(string numbers)
13     {
14         var delimiters = ",\n";
15         if (string.IsNullOrEmpty(numbers))
16             return 0;
17
18         if (numbers.Contains("//"))
19         {
20             delimiters += numbers[2];
21             numbers = numbers.Substring(4, numbers.Length - 4);
22         }
23
24         var items = numbers.Split(delimiters.ToCharArray());
25
26         if (items.Any(x => string.IsNullOrEmpty(x)))
27             throw new ArgumentException();
28
29         var integers = items.Select(x => int.Parse(x));
30         var negatives = integers.Where(x => x < 0);
31         if (negatives.Count() > 0)
32         {
33             var message = "Negatives not allowed: {0}";
34             throw new ArgumentOutOfRangeException(
35                 string.Format(message,
36                     string.Join(" ", negatives.Select(x => x.ToString()).ToArray())));
37         }
38         return integers.Sum();
39     }
40 }
41
```

4 Il reste à faire...

Implémentez les exigences 10, 11, 12 manquantes ainsi que le client console utilisant la classe SimpleCalculator...

5 Conclusion

L'idée est que coder est un processus incrémental et que chaque nouveau cycle doit être initié par un besoin spécifique défini par un test dédié.

L'écriture des tests est simple : on décompose notre script en une suite d'affirmations correspondant chacune à une fonctionnalité précise de notre algorithme.

Grâce à ce processus on évite :

- **les régressions :** la suite valide de tests est la garantie que le code reste fonctionnel malgré les évolutions de l'algorithme.
- **le code mort :** chaque morceau de code écrit est testé et a son utilité.
- **le code non documenté :** chaque comportement est décrit de manière fonctionnelle.

Le TDD est destiné à être incorporé à un processus d'intégration continue pour s'assurer du bon fonctionnement de l'application sur tous les environnements de production après chaque nouveau commit.