eBook Gratuit

APPRENEZ unit-testing

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#unit-testing

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: unit-testing

It is an unofficial and free unit-testing ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official unit-testing.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec les tests unitaires

Remarques

Les tests unitaires décrivent le processus de test des unités de code individuelles indépendamment du système dont elles font partie. Ce qui constitue une unité peut varier d'un système à l'autre, allant d'une méthode individuelle à un groupe de classes étroitement liées ou d'un module.

L'unité est isolée de ses dépendances à l'aide de tests doubles si nécessaire et configurée dans un état connu. Son comportement en réaction aux stimuli (appels de méthode, événements, données simulées) est ensuite testé par rapport au comportement attendu.

Les tests unitaires de systèmes entiers peuvent être réalisés à l'aide de faisceaux de test écrits personnalisés. Cependant, de nombreux frameworks de test ont été écrits pour simplifier le processus et prendre en charge une grande partie des tâches de plomberie, répétitives et banales. Cela permet aux développeurs de se concentrer sur ce qu'ils veulent tester.

Lorsqu'un projet a suffisamment de tests unitaires, toute modification apportée à l'ajout de nouvelles fonctionnalités ou à la refactorisation de code peut être effectuée facilement en vérifiant à la fin que tout fonctionne comme avant.

La couverture du code , normalement exprimée en pourcentage, est la mesure type utilisée pour indiquer la part du code dans un système couverte par les tests unitaires; notez qu'il n'y a pas de règle stricte quant à la hauteur que cela devrait être, mais il est généralement admis que plus le niveau est élevé, mieux c'est.

Le développement piloté par les tests (TDD) est un principe qui spécifie qu'un développeur doit commencer à coder en écrivant un test unitaire défaillant et ensuite seulement pour écrire le code de production qui fait passer le test. En pratiquant le TDD, on peut dire que les tests eux-mêmes sont le premier consommateur du code en cours de création; Par conséquent, ils aident à auditer et à piloter la conception du code de manière à ce qu'il soit aussi simple à utiliser et aussi robuste que possible.

Versions

Le test unitaire est un concept qui n'a pas de numéro de version.

Examples

Un test élémentaire de base

Dans sa forme la plus simple, un test unitaire comprend trois étapes:

- Préparer l'environnement pour le test

- Exécutez le code à tester
- Valider le comportement attendu correspond au comportement observé

Ces trois étapes sont souvent appelées «Arrange-Act-Assert» ou «Given-When-Then».

Vous trouverez ci-dessous un exemple en C # qui utilise le framework NUnit .

```
[TestFixture]
public CalculatorTest
{
   [Test]
   public void Add_PassSevenAndThree_ExpectTen()
   {
       // Arrange - setup environment
       var systemUnderTest = new Calculator();

       // Act - Call system under test
       var calculatedSum = systemUnderTest.Add(7, 3);

       // Assert - Validate expected result
       Assert.AreEqual(10, calculatedSum);
  }
}
```

Si nécessaire, une quatrième étape de nettoyage facultative est prévue.

Un test unitaire avec une dépendance par écrasement

Les bons tests unitaires sont indépendants, mais le code a souvent des dépendances. Nous utilisons différents types de doubles de test pour supprimer les dépendances à tester. L'un des tests les plus simples est un stub. Ceci est une fonction avec une valeur de retour codée en dur appelée à la place de la dépendance du monde réel.

```
// Test that oneDayFromNow returns a value 24*60*60 seconds later than current time

let systemUnderTest = new FortuneTeller()       // Arrange - setup environment
systemUnderTest.setNow(() => {return 10000})    //   inject a stub which will
                                                //   return 10000 as the result

let actual = systemUnderTest.oneDayFromNow()    // Act - Call system under test

assert.equals(actual, 10000 + 24 * 60 * 60)     // Assert - Validate expected result
```

Dans le code de production, `oneDayFromNow` appellerait Date.now (), mais cela provoquerait des tests incohérents et peu fiables. Donc, ici, nous le supprimons.

Un test unitaire avec un espion (test d'interaction)

Test de l'unité classique teste l' *état* , mais il peut être impossible de tester correctement les méthodes dont le comportement dépend d'autres classes via l'état. Nous testons ces méthodes à l'aide de *tests d'interaction* , qui vérifient que le système testé appelle correctement ses collaborateurs. Étant donné que les collaborateurs ont leurs propres tests unitaires, cela est suffisant et constitue en fait un meilleur test de la responsabilité réelle de la méthode testée. Nous

ne testons pas que cette méthode retourne un résultat donné à une entrée, mais appelle plutôt ses collaborateurs.

```
// Test that squareOfDouble invokes square() with the doubled value

let systemUnderTest = new Calculator()          // Arrange - setup environment
let square = spy()
systemUnderTest.setSquare(square)               //   inject a spy

let actual = systemUnderTest.squareOfDouble(3)  // Act - Call system under test

assert(square.calledWith(6))                    // Assert - Validate expected interaction
```

Test simple Java + JUnit

JUnit est la principale infrastructure de test utilisée pour tester le code Java.

La classe testée modélise un compte bancaire simple, qui facture une pénalité lorsque vous êtes à découvert.

```
public class BankAccount {
    private int balance;

    public BankAccount(int i){
        balance = i;
    }

    public BankAccount(){
        balance = 0;
    }

    public int getBalance(){
        return balance;
    }

    public void deposit(int i){
        balance += i;
    }

    public void withdraw(int i){
        balance -= i;
        if (balance < 0){
            balance -= 10; // penalty if overdrawn
        }
    }
}
```

Cette classe de test valide le comportement de certaines méthodes publiques `BankAccount` .

```
import org.junit.Test;
import static org.junit.Assert.*;

// Class that tests
public class BankAccountTest{

    BankAccount acc;
```

```
    @Before                         // This will run **before** EACH @Test
    public void setUptestDepositUpdatesBalance(){
        acc = new BankAccount(100);
    }

    @After                         // This Will run **after** EACH @Test
    public void tearDown(){
    // clean up code
    }

    @Test
    public void testDeposit(){
       // no need to instantiate a new BankAccount(), @Before does it for us

        acc.deposit(100);

        assertEquals(acc.getBalance(),200);
    }

    @Test
    public void testWithdrawUpdatesBalance(){
        acc.withdraw(30);

        assertEquals(acc.getBalance(),70); // pass
    }

    @Test
    public void testWithdrawAppliesPenaltyWhenOverdrawn(){

        acc.withdraw(120);

        assertEquals(acc.getBalance(),-30);
    }
}
```

Test unitaire avec paramètres utilisant NUnit et C

```
using NUnit.Framework;

namespace MyModuleTests
{
    [TestFixture]
    public class MyClassTests
    {
        [TestCase(1, "Hello", true)]
        [TestCase(2, "bye", false)]
        public void MyMethod_WhenCalledWithParameters_ReturnsExpected(int param1, string
param2, bool expected)
        {
        //Arrange
        var foo = new MyClass(param1);

        //Act
        var result = foo.MyMethod(param2);

        //Assert
        Assert.AreEqual(expected, result);
        }
```

```
    }
}
```

Un test élémentaire de base en python

```
import unittest

def addition(*args):
    """ add two or more summands and return the sum """

    if len(args) < 2:
        raise ValueError, 'at least two summands are needed'

    for ii in args:
        if not isinstance(ii, (int, long, float, complex )):
            raise TypeError

    # use build in function to do the job
    return sum(args)
```

Maintenant la partie test:

```
class Test_SystemUnderTest(unittest.TestCase):

    def test_addition(self):
        """test addition function"""

        # use only one summand - raise an error
        with self.assertRaisesRegexp(ValueError, 'at least two summands'):
            addition(1)

        # use None - raise an error
        with self.assertRaises(TypeError):
            addition(1, None)

        # use ints and floats
        self.assertEqual(addition(1, 1.), 2)

        # use complex numbers
        self.assertEqual(addition(1, 1., 1+2j), 3+2j)

if __name__ == '__main__':
    unittest.main()
```

Un test XUnit avec des paramètres

```
using Xunit;

public class SimpleCalculatorTests
{
    [Theory]
    [InlineData(0, 0, 0, true)]
    [InlineData(1, 1, 2, true)]
    [InlineData(1, 1, 3, false)]
    public void Add_PassMultipleParameters_VerifyExpected(
        int inputX, int inputY, int expected, bool isExpectedCorrect)
```

```
    {
        // Arrange
        var sut = new SimpleCalculator();

        // Act
        var actual = sut.Add(inputX, inputY);

        // Assert
        if (isExpectedCorrect)
        {
            Assert.Equal(expected, actual);
        }
        else
        {
            Assert.NotEqual(expected, actual);
        }
    }
}

public class SimpleCalculator
{
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

Lire Démarrer avec les tests unitaires en ligne: https://riptutorial.com/fr/unit-testing/topic/570/demarrer-avec-les-tests-unitaires

Chapitre 2: Guide des tests unitaires dans Visual Studio for C

Introduction

Comment créer des projets unitaires et des tests unitaires et comment exécuter les tests unitaires et l'outil de couverture de code.

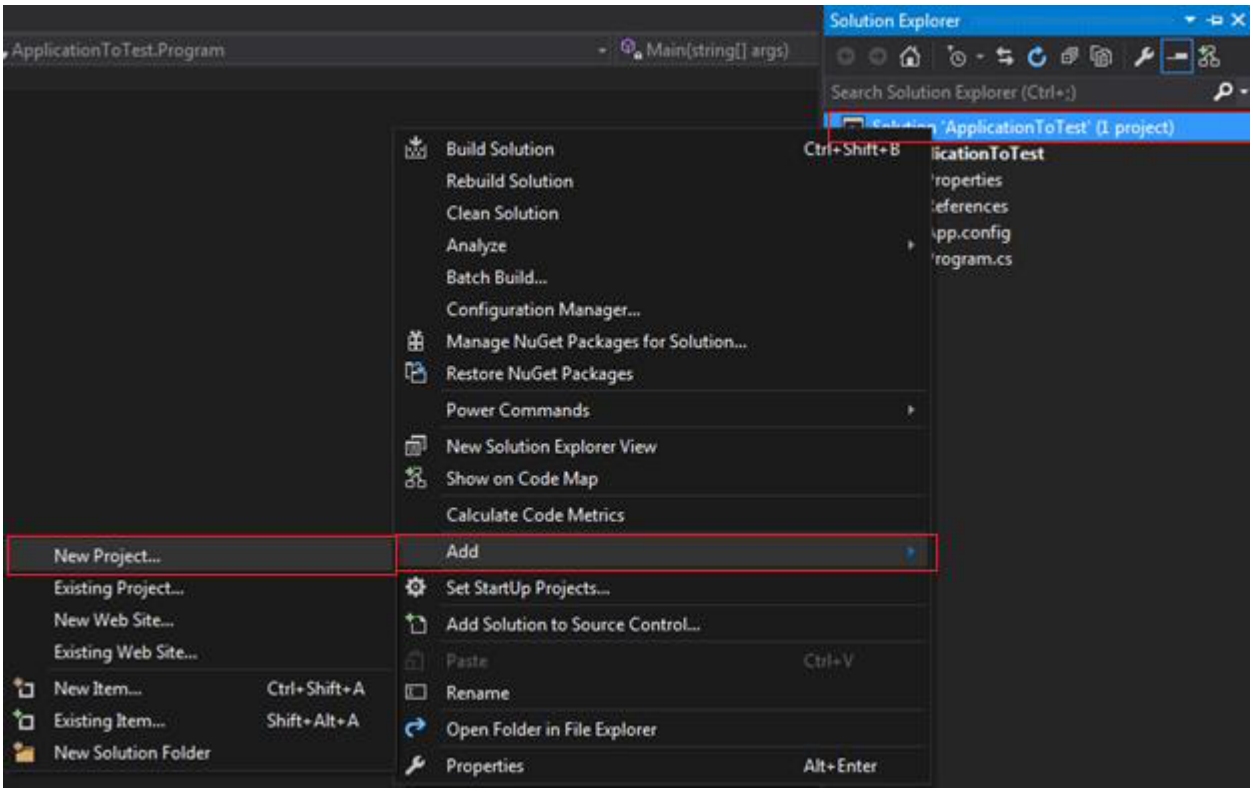
Dans ce guide, l'infrastructure standard de MSTest sera utilisée et l'outil standard d'analyse de la couverture de code, disponible dans Visual Studio.

Le guide a été écrit pour Visual Studio 2015, il est donc possible que certaines choses soient différentes dans d'autres versions.

Examples

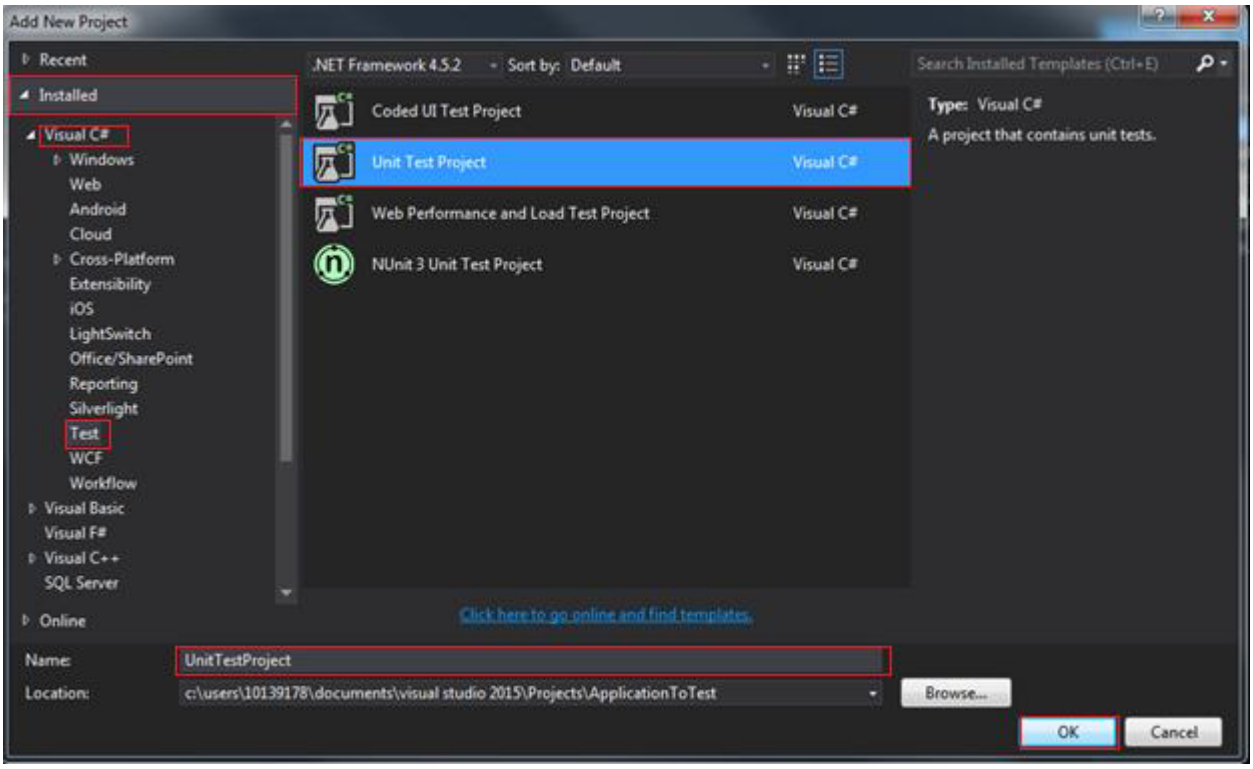
Création d'un projet de test unitaire

- Ouvrez le projet C #
- Faites un clic droit sur la solution -> Ajouter -> Nouveau projet…
- (Figure 1)

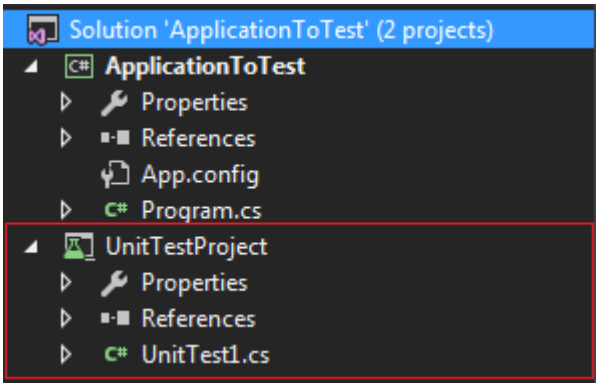


- Aller à Installé -> Visual C # -> Test
- Cliquez sur Projet de test unitaire
- Donnez-lui un nom et cliquez sur OK

- (Figure 2)

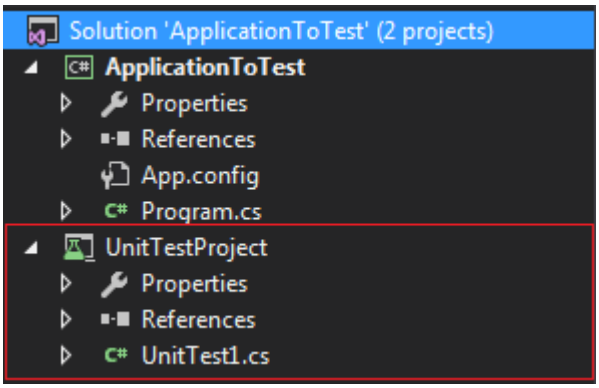


- Le projet de test unitaire est ajouté à la solution
- (Figure 3)

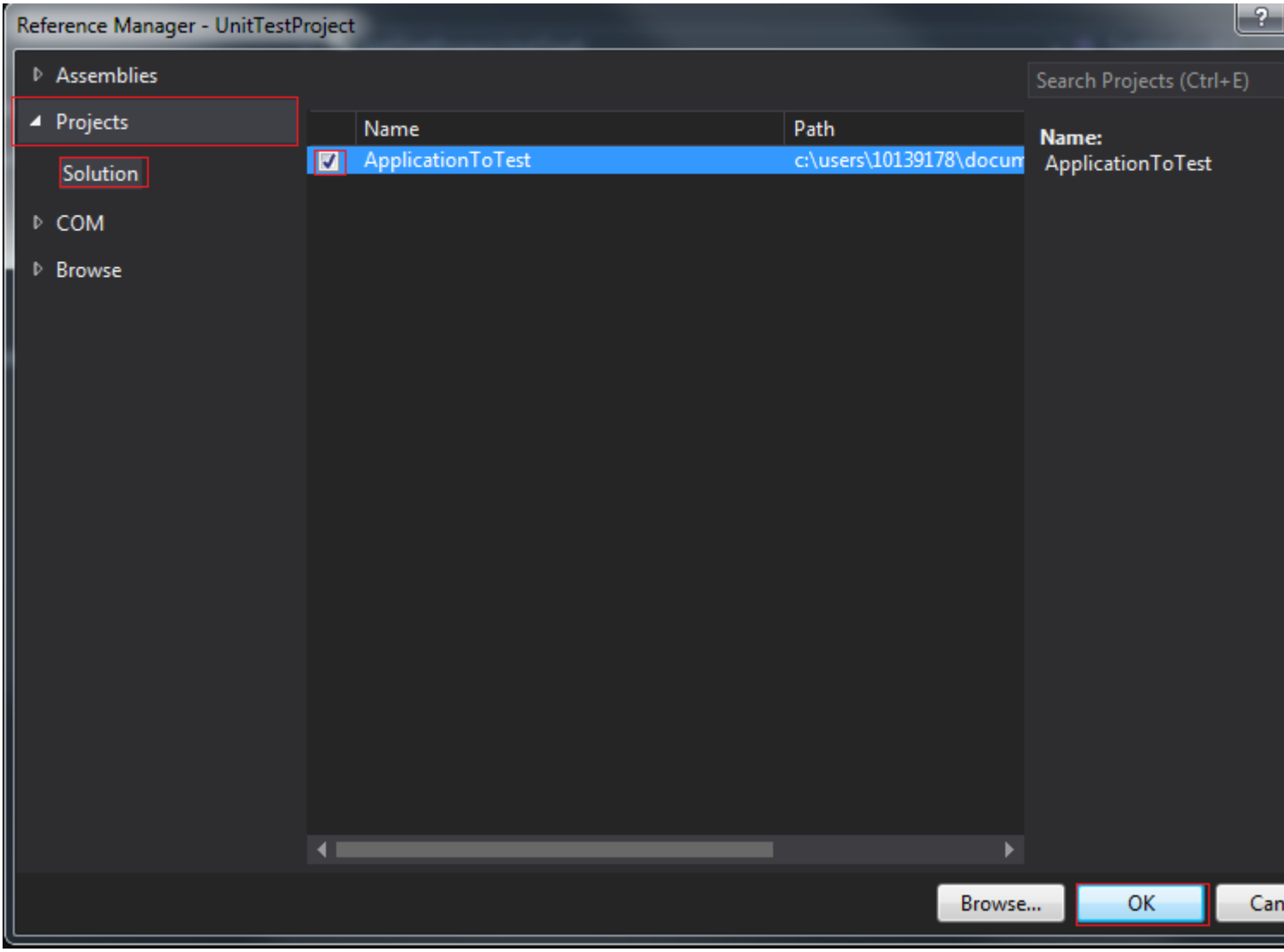


Ajout de la référence à l'application que vous souhaitez tester

- Dans le projet de test unitaire, ajoutez une référence au projet que vous souhaitez tester
- Faites un clic droit sur Références -> Ajouter une référence…
- (Figure 3)



- Sélectionnez le projet que vous souhaitez tester
- Aller à Projets -> Solution
- Cochez la case du projet que vous souhaitez tester -> cliquez sur OK
- (Figure 4)



Deux méthodes pour créer des tests unitaires

Méthode 1

- Accédez à votre classe de test unitaire dans le projet de test unitaire

- Ecrire un test unitaire

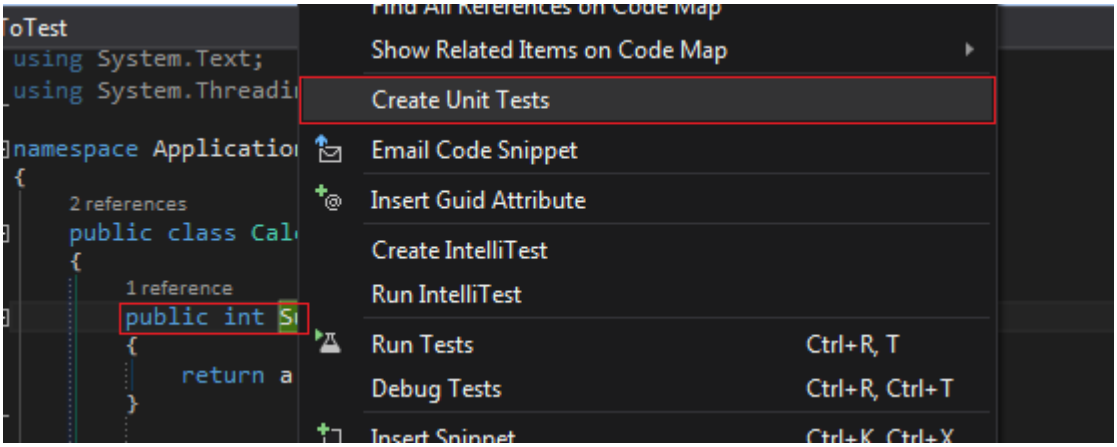
```
[Testclass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
        //Arrange
        ApplicationToTest.Calc ClassCalc = new ApplicationToTest.Calc();
        int expectedResult = 5;

        //Act
        int result = ClassCalc.Sum(2,3);

        //Assert
        Assert.AreEqual(expectedResult, result);
    }
}
```

Méthode 2

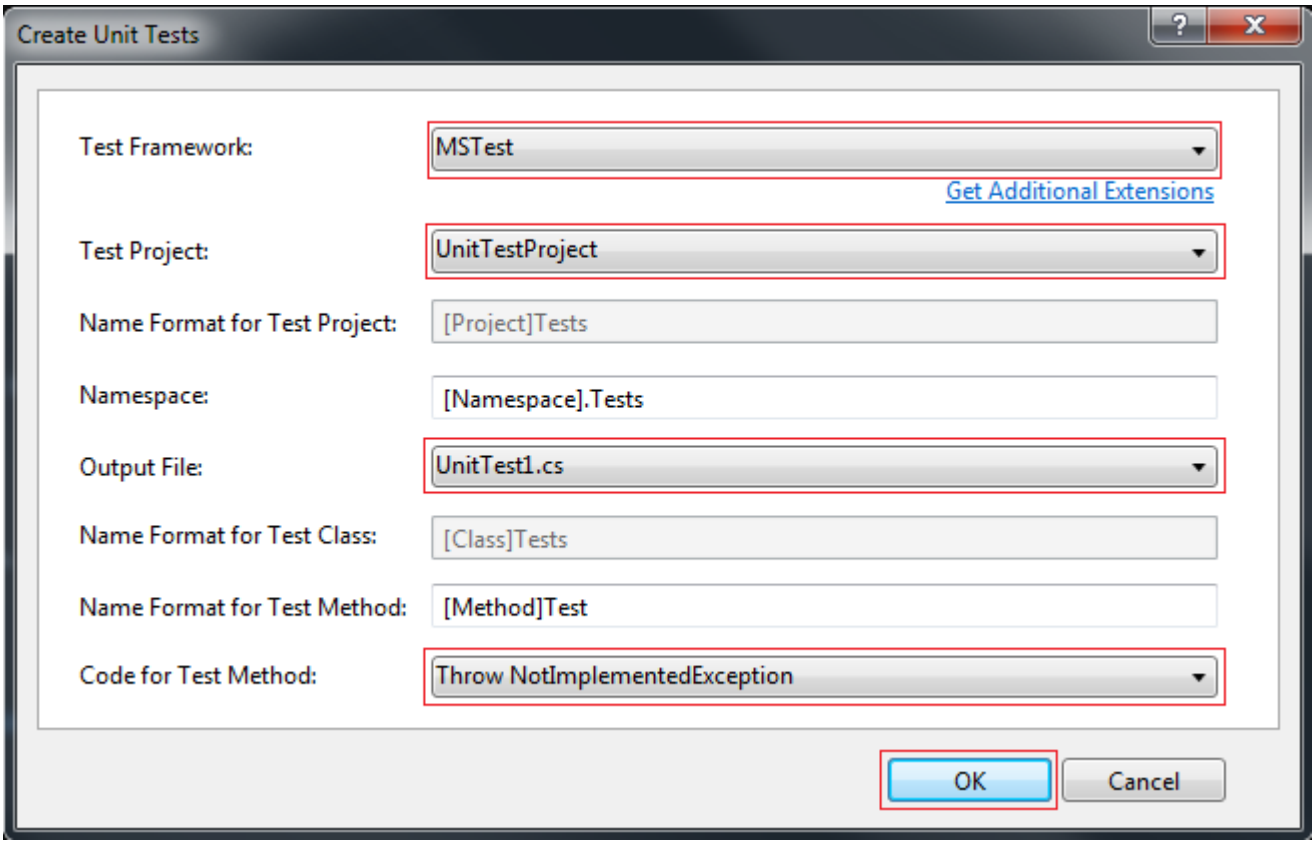
- Aller à la méthode que vous voulez tester
- Faites un clic droit sur la méthode -> Créer des tests unitaires
- (Figure 4)



- Définir le framework de test sur MSTest
- Définissez le projet de test sur le nom de votre projet de test unitaire
- Définir le fichier de sortie sur le nom de la classe des tests unitaires
- Définissez le code de la méthode de test sur l'une des options répertoriées que vous préférez
- Les autres options peuvent être éditées mais ce n'est pas nécessaire

(Astuce: Si vous n’avez pas encore réalisé de projet de tests unitaires, vous pouvez toujours utiliser cette option. Il suffit de définir Test Project sur et Output File to. Il créera le projet de test unitaire et ajoutera la référence du projet au projet de test unitaire)

- (Figure 5)

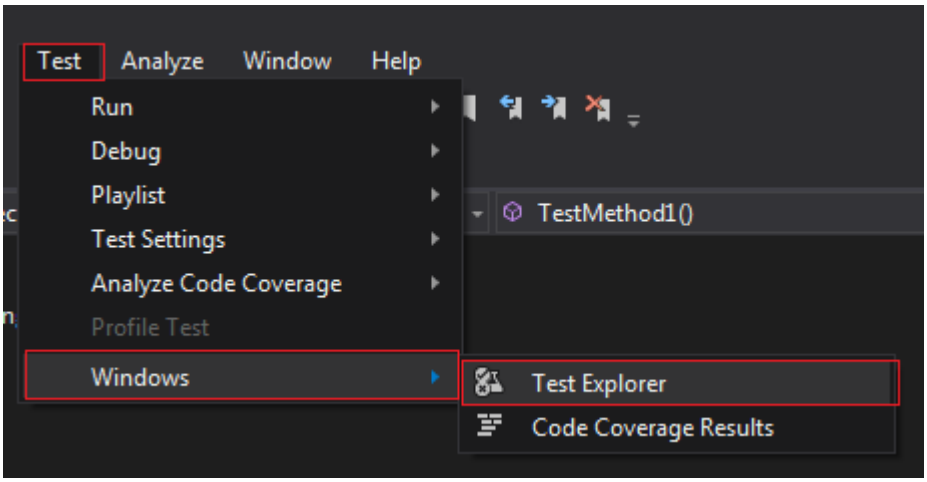


- Comme vous le voyez ci-dessous, il crée la base du test unitaire à remplir
- (Figure 6)

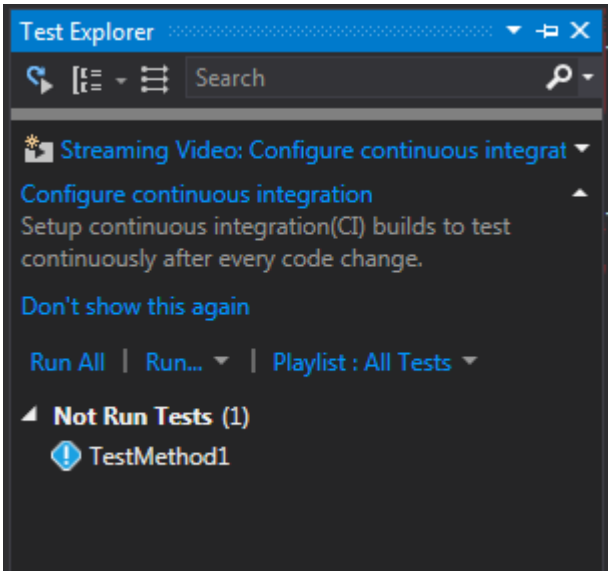
```
namespace ApplicationToTest.Tests
{
    [TestClass()]
    0 references
    public class UnitTest1
    {
        [TestMethod()]
        0 references
        public void SumTest()
        {
            throw new NotImplementedException();
        }
    }
}
```

Exécution de tests unitaires dans Visual Studio

- Pour voir vos tests unitaires, allez dans Test -> Windows -> Test Explorer
- (Figure 1)



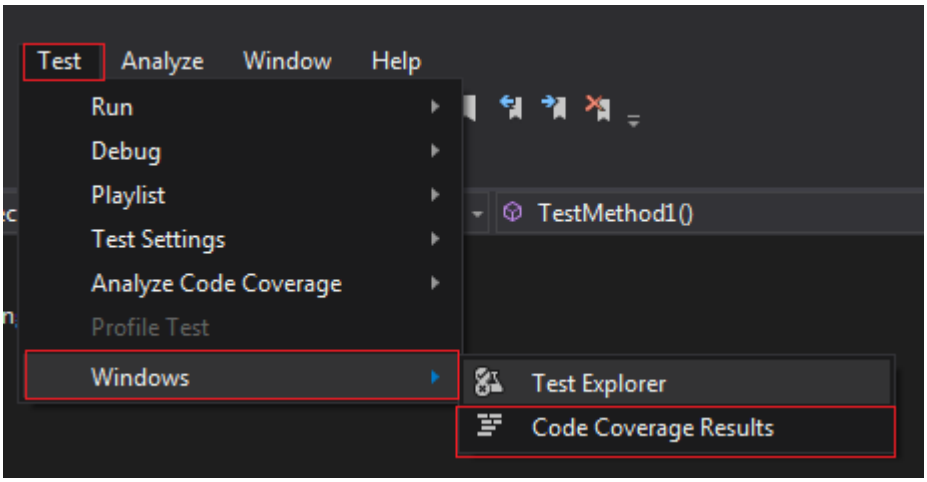
- Cela ouvrira un aperçu de tous les tests de l'application
- (Figure 2)



- Dans la figure ci-dessus, vous pouvez voir que l'exemple a un test unitaire et qu'il n'a pas encore été exécuté
- Vous pouvez double-cliquer sur un test pour accéder au code où le test unitaire est défini
- Vous pouvez exécuter un ou plusieurs tests avec Run All or Run…
- Vous pouvez également exécuter des tests et modifier les paramètres à partir du menu Test (Figure 1).

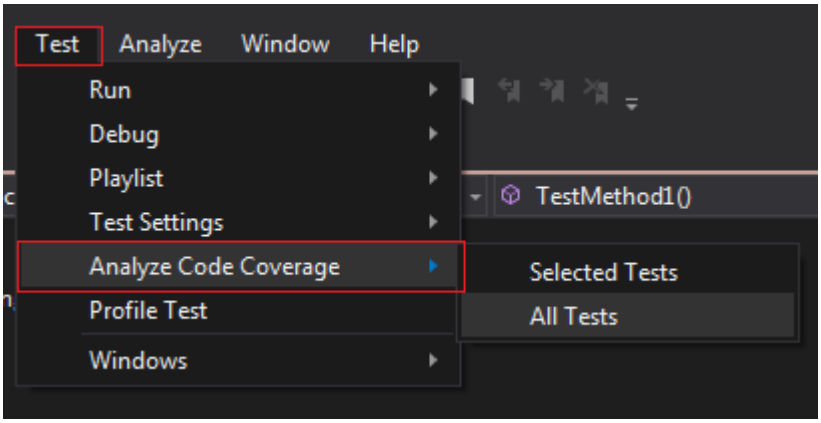
Analyse de la couverture du code dans Visual Studio

- Pour voir vos tests unitaires, allez dans Test -> Windows -> Résultats de la couverture du code
- (Figure 1)

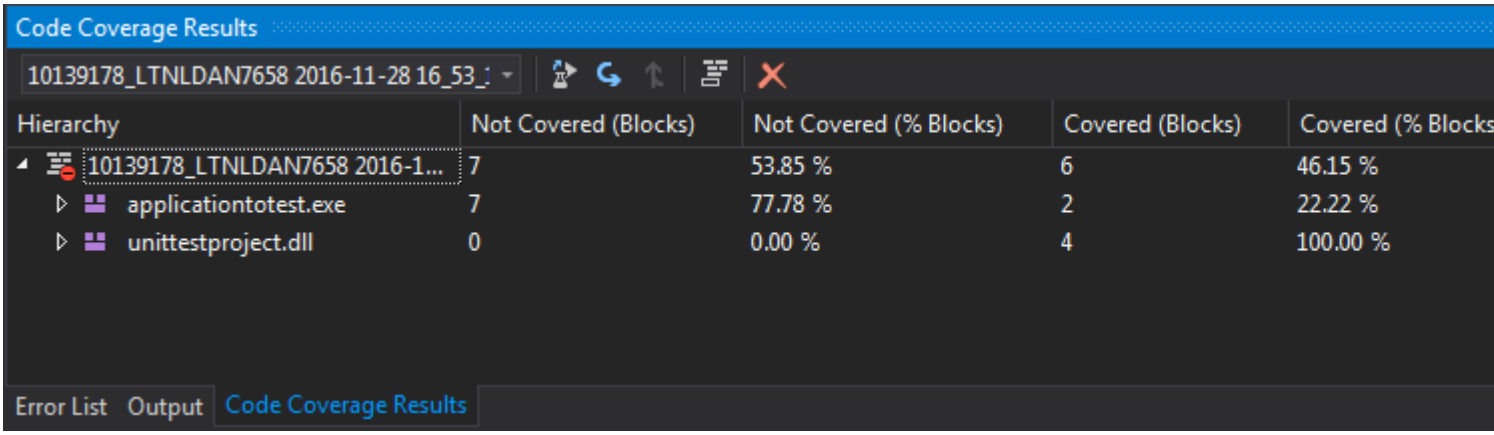


- Il va ouvrir la fenêtre suivante
- (Figure 2)

- La fenêtre est maintenant vide
- Allez dans le menu Test -> Analyser la couverture du code
- (Figure 3)



- Les tests seront désormais exécutés également (voir les résultats dans l'explorateur de tests)
- Les résultats seront affichés dans un tableau, vous pourrez voir quelles classes et méthodes sont couvertes par les tests unitaires et celles qui ne le sont pas.
- (Figure 4)



Lire Guide des tests unitaires dans Visual Studio for C # en ligne: https://riptutorial.com/fr/unit-testing/topic/9953/guide-des-tests-unitaires-dans-visual-studio-for-c-sharp

Chapitre 3: Injection de dépendance

Remarques

Une approche qui peut être prise pour écrire un logiciel consiste à créer des dépendances au besoin. C'est une manière assez intuitive d'écrire un programme et c'est la manière dont la plupart des gens auront tendance à apprendre, en partie parce que c'est facile à suivre. L'un des problèmes de cette approche est qu'il peut être difficile à tester. Considérons une méthode qui effectue un traitement en fonction de la date actuelle. La méthode peut contenir du code comme celui-ci:

```
if (DateTime.Now.Date > processDate)
{
    // Do some processing
}
```

Le code a une dépendance directe à la date actuelle. Cette méthode peut être difficile à tester car la date actuelle ne peut pas être facilement manipulée. Une méthode pour rendre le code plus vérifiable consiste à supprimer la référence directe à la date actuelle et à fournir (ou injecter) la date actuelle à la méthode qui effectue le traitement. Cette injection de dépendance permet de tester plus facilement des aspects du code en utilisant des doubles de test pour simplifier l'étape d'installation du test unitaire.

Systèmes IOC

Un autre aspect à considérer est la durée de vie des dépendances; dans le cas où la classe elle-même crée ses propres dépendances (également appelées invariants), elle est alors responsable de leur disposition. Dependency Injection inverse (et c'est pourquoi nous faisons souvent référence à une bibliothèque d'injection comme un système "Inversion of Control") et que, au lieu que la classe soit responsable de la création, la gestion et le nettoyage de ses dépendances, un agent externe cas, le système IoC) le fait à la place.

Cela rend beaucoup plus simple d'avoir des dépendances partagées entre des instances de la même classe; Par exemple, considérez un service qui récupère des données à partir d'un point de terminaison HTTP pour une classe à consommer. Comme ce service est sans état (c'est-à-dire qu'il n'a pas d'état interne), nous n'avons besoin que d'une seule instance de ce service dans notre application. Bien qu'il soit possible (par exemple en utilisant une classe statique) de le faire manuellement, il est beaucoup plus simple de créer la classe et d'indiquer au système IoC qu'elle doit être créée en tant que *Singleton* .

Un autre exemple serait les contextes de base de données utilisés dans une application Web, un nouveau contexte étant requis par demande (ou thread) et non par instance de contrôleur; Cela permet d'injecter le contexte dans chaque couche exécutée par ce thread, sans avoir à être passé manuellement.

Cela libère les classes consommatrices de la gestion des dépendances.

Examples

Constructeur Injection

L'injection de constructeur est le moyen le plus sûr d'injecter des dépendances dont dépend toute une classe. De telles dépendances sont souvent appelées *invariants* , car il est impossible de créer une instance de la classe sans les fournir. En exigeant que la dépendance soit injectée lors de la construction, il est garanti qu'un objet ne peut pas être créé dans un état incohérent.

Considérons une classe qui doit écrire dans un fichier journal dans des conditions d'erreur. Il a une dépendance à un `ILogger` , qui peut être injecté et utilisé si nécessaire.

```
public class RecordProcessor
{
    readonly private ILogger _logger;

    public RecordProcessor(ILogger logger)
    {
        _logger = logger;
    }

    public void DoSomeProcessing() {
        // ...
        _logger.Log("Complete");
    }
}
```

Parfois, lors de l'écriture de tests, vous pouvez noter que le constructeur nécessite plus de dépendances que ce qui est réellement nécessaire pour un cas testé. Plus vous avez de tests de ce type, plus il est probable que votre classe casse le *principe de responsabilité unique* (SRP). C'est pourquoi il n'est pas très judicieux de définir le comportement par défaut de toutes les dépendances injectées lors de la phase d'initialisation de la classe de test, car il peut masquer le signal d'avertissement potentiel.

Le plus important pour cela serait le suivant:

```
[Test]
public void RecordProcessor_DependencyInjectionExample()
{
    ILogger logger = new FakeLoggerImpl(); //or create a mock by a mocking Framework

    var sut = new RecordProcessor(logger); //initialize with fake impl in testcode

    Assert.IsTrue(logger.HasCalledExpectedMethod());
}
```

Injection de propriété

L'injection de propriété permet aux dépendances d'une classe d'être mises à jour après sa création. Cela peut être utile si vous souhaitez simplifier la création d’objets, tout en permettant à vos tests de remplacer les dépendances par des tests doubles.

Considérons une classe qui doit écrire dans un fichier journal dans une condition d'erreur. La classe sait comment construire un `Logger` par défaut, mais permet de la remplacer par une injection de propriété. Cependant, cela vaut la peine de noter que l'utilisation de l'injection de propriété de cette façon vous permet de coupler étroitement cette classe avec une implémentation exacte de `ILogger` qui est `ConcreteLogger` dans cet exemple. Une solution de contournement possible pourrait être une fabrique qui renvoie l'implémentation ILogger nécessaire.

```
public class RecordProcessor
{
    public RecordProcessor()
    {
        Logger = new ConcreteLogger();
    }

    public ILogger Logger { get; set; }

    public void DoSomeProcessing()
    {
        // ...
        _logger.Log("Complete");
    }
}
```

Dans la plupart des cas, l'injection de constructeur est préférable à l'injection de propriété car elle fournit de meilleures garanties sur l'état de l'objet immédiatement après sa construction.

Méthode injection

L'injection de méthode est une manière fine d'injecter des dépendances dans le traitement. Considérons une méthode qui effectue un traitement en fonction de la date actuelle. La date actuelle est difficile à modifier depuis un test, il est donc beaucoup plus facile de transmettre une date à la méthode que vous souhaitez tester.

```
public void ProcessRecords(DateTime currentDate)
{
    foreach(var record in _records)
    {
        if (currentDate.Date > record.ProcessDate)
        {
            // Do some processing
        }
    }
}
```

Cadres Conteneurs / DI

Tandis que l'extraction des dépendances de votre code pour qu'elles puissent être injectées rend votre code plus facile à tester, il pousse le problème plus haut dans la hiérarchie et peut également générer des objets difficiles à construire. Divers cadres d'injection de dépendance / Inversion des conteneurs de contrôle ont été écrits pour aider à résoudre ce problème. Celles-ci permettent d’enregistrer les mappages de types. Ces enregistrements sont ensuite utilisés pour résoudre les dépendances lorsque le conteneur est invité à créer un objet.

Considérons ces classes:

```
public interface ILogger {
    void Log(string message);
}

public class ConcreteLogger : ILogger
{
    public ConcreteLogger()
    {
        // ...
    }
    public void Log(string message)
    {
        // ...
    }
}
public class SimpleClass
{
    public SimpleClass()
    {
        // ...
    }
}

public class SomeProcessor
{
    public SomeProcessor(ILogger logger, SimpleClass simpleClass)
    {
        // ...
    }
}
```

Pour construire `SomeProcessor` , une instance de `ILogger` et de `SimpleClass` est requise. Un conteneur comme Unity aide à automatiser ce processus.

D'abord, le conteneur doit être construit et les correspondances sont enregistrées avec lui. Cela se fait généralement une seule fois dans une application. La zone du système où cela se produit est communément appelée *racine de composition*

```
// Register the container
var container = new UnityContainer();

// Register a type mapping.  This allows a `SimpleClass` instance
// to be constructed whenever it is required.
container.RegisterType<SimpleClass, SimpleClass>();

// Register an instance.  This will use this instance of `ConcreteLogger`
// Whenever an `ILogger` is required.
container.RegisterInstance<ILogger>(new ConcreteLogger());
```

Une fois le conteneur configuré, il peut être utilisé pour créer des objets, en résolvant automatiquement les dépendances selon les besoins:

```
var processor = container.Resolve<SomeProcessor>();
```

Lire Injection de dépendance en ligne: https://riptutorial.com/fr/unit-testing/topic/597/injection-de-dependance

Chapitre 4: Les règles générales pour les tests unitaires pour toutes les langues

Introduction

Lorsque vous commencez avec les tests unitaires, toutes sortes de questions se posent:

Qu'est-ce qu'un test unitaire? Qu'est-ce qu'un SetUp et TearDown? Comment gérer les dépendances? Pourquoi faire des tests unitaires? Comment puis-je faire de bons tests unitaires?

Cet article répondra à toutes ces questions afin que vous puissiez commencer les tests unitaires dans la langue de votre choix.

Remarques

Qu'est-ce que le test unitaire?

Le test d'unité consiste à tester le code pour s'assurer qu'il exécute la tâche qu'il est censé effectuer. Il teste le code au niveau le plus bas possible - les méthodes individuelles de vos classes.

Qu'est-ce qu'une unité?

Tout module de code discret pouvant être testé isolément. La plupart des classes de temps et leurs méthodes. Cette classe est généralement appelée "Class Under Test" (CUT) ou "System Under Test" (SUT).

La différence entre les tests unitaires et les tests d'intégration

Les tests unitaires consistent à tester une classe unique de manière totalement indépendante de ses dépendances. Les tests d'intégration consistent à tester une classe unique avec une ou plusieurs de ses dépendances réelles.

The SetUp et TearDown

Lorsqu'elle est créée, la méthode SetUp est exécutée avant chaque test unitaire et le TearDown après chaque test.

En général, vous ajoutez toutes les étapes préalables du SetUp et toutes les étapes de nettoyage du TearDown. Mais vous ne faites ces méthodes que si ces étapes sont nécessaires pour chaque test. Si ce n'est pas le cas, ces étapes sont prises dans les tests spécifiques de la section

"Arranger".

Comment gérer les dépendances

Plusieurs fois, une classe a la dépendance des autres classes pour exécuter ses méthodes. Pour pouvoir ne pas dépendre de ces autres classes, il faut faire semblant. Vous pouvez créer ces classes vous-même ou utiliser un framework d'isolement ou de maquette. Un cadre d'isolation est un ensemble de codes permettant de créer facilement de fausses classes.

Faux cours

Toute classe fournissant des fonctionnalités suffisantes pour prétendre qu'il s'agit d'une dépendance requise par une découpe. Il existe deux types de contrefaçons: les talons et les mocks.

- Un talon: Un faux qui n'a aucun effet sur la réussite ou l'échec du test et qui existe uniquement pour permettre l'exécution du test.
- Un simulacre: Un faux qui garde la trace du comportement de la CUT et réussit ou échoue au test en fonction de ce comportement.

Pourquoi les tests unitaires?

1. Les tests unitaires trouveront des bugs

Lorsque vous écrivez une suite complète de tests définissant le comportement attendu pour une classe donnée, tout ce qui ne se comporte pas comme prévu est révélé.

2. Les tests unitaires éloigneront les bogues

Faites un changement qui introduit un bogue et vos tests peuvent le révéler la prochaine fois que vous exécuterez vos tests.

3. Les tests unitaires permettent de gagner du temps

L'écriture de tests unitaires permet de s'assurer que votre code fonctionne comme prévu dès le départ. Les tests unitaires définissent ce que votre code doit faire et vous ne passerez donc pas de temps à écrire du code qui fait des choses qu'il ne devrait pas faire. Personne ne vérifie que le code ne fonctionne pas et que vous devez faire quelque chose pour vous faire croire que cela fonctionne. Passez ce temps à écrire des tests unitaires.

4. Les tests unitaires donnent la tranquillité d'esprit

Vous pouvez exécuter tous ces tests et savoir que votre code fonctionne comme il se doit. Connaître l'état de votre code, son fonctionnement, sa mise à jour et son amélioration sans crainte est une très bonne chose.

5. Les tests unitaires documentent l'utilisation correcte d'une classe

Les tests unitaires deviennent des exemples simples du fonctionnement de votre code, de ce qu'il est censé faire et de la manière correcte d'utiliser votre code en cours de test.

Règles générales pour les tests unitaires

1. Pour la structure d'un test unitaire, suivez la règle AAA

Organiser:

Mettre en place la chose à tester. Comme les variables, les champs et les propriétés pour permettre l'exécution du test ainsi que le résultat attendu.

Act: Appelez la méthode que vous testez

Affirmer:

Appelez le framework de test pour vérifier que le résultat de votre "acte" est ce qui était attendu.

2. Tester une chose à la fois isolément

Toutes les classes doivent être testées isolément. Ils ne devraient dépendre que des moqueries et des talons. Ils ne devraient pas dépendre des résultats d'autres tests.

3. Ecrivez d'abord les tests simples "au centre"

Les premiers tests que vous écrivez devraient être les tests les plus simples. Ils devraient être ceux qui illustrent les fonctionnalités que vous essayez d'écrire. Ensuite, une fois ces tests réussis, vous devez écrire les tests les plus compliqués qui testent les contours et les limites de votre code.

4. Rédiger des tests pour tester les contours

Une fois que les bases sont testées et que vous savez que vos fonctionnalités de base fonctionnent, vous devez tester les contours. Un bon ensemble de tests explorera les limites extérieures de ce qui pourrait arriver à une méthode donnée.

Par exemple:

- Que se passe-t-il si un débordement se produit?
- Que faire si les valeurs vont à zéro ou en dessous?
- Et s'ils vont à MaxInt ou MinInt?
- Et si vous créez un arc de 361 degrés?
- Que se passe-t-il si vous passez une chaîne vide?
- Que se passe-t-il si une chaîne mesure 2 Go?

5. Test à travers les frontières

Les tests unitaires doivent tester les deux côtés d'une limite donnée. Se déplacer au-delà des frontières sont des endroits où votre code peut échouer ou fonctionner de manière imprévisible.

6. Si vous le pouvez, testez l’ensemble du spectre

Si c'est pratique, testez l'ensemble des possibilités de votre fonctionnalité. S'il implique un type énuméré, testez la fonctionnalité avec chacun des éléments de l'énumération. Il peut être peu pratique de tester toutes les possibilités, mais si vous pouvez tester chaque possibilité, faites-le.

7. Si possible, couvrir tous les chemins de code

Celui-ci est également difficile, mais si votre code est conçu pour des tests et que vous utilisez un outil de couverture de code, vous pouvez vous assurer que chaque ligne de votre code est couverte par des tests unitaires au moins une fois. Couvrir chaque chemin de code ne garantit pas qu'il n'y a pas de bogues, mais cela vous donne sûrement des informations précieuses sur l'état de chaque ligne de code.

8. Ecrivez des tests qui révèlent un bogue, puis corrigez-le

Si vous trouvez un bug, écrivez un test qui le révèle. Ensuite, vous corrigez facilement le bogue en déboguant le test. Ensuite, vous avez un bon test de régression pour vous assurer que si le bogue revient pour une raison quelconque, vous le saurez immédiatement. Il est très facile de corriger un bogue lorsque vous avez un test simple et direct à exécuter dans le débogueur.

Un avantage secondaire est que vous avez testé votre test. Comme vous avez vu le test échouer et que vous l'avez vu passer, vous savez que le test est valide car il a été prouvé qu'il fonctionne correctement. Cela en fait un test de régression encore meilleur.

9. Faites chaque test indépendamment l'un de l'autre

Les tests ne doivent jamais dépendre les uns des autres. Si vos tests doivent être exécutés dans un certain ordre, vous devez modifier les tests.

10. Ecrivez un assert par test

Vous devriez écrire un assert par test. Si vous ne pouvez pas faire cela, réfractez votre code afin que vos événements SetUp et TearDown soient utilisés pour créer correctement l'environnement afin que chaque test puisse être exécuté individuellement.

11. Nommez clairement vos tests. N'ayez pas peur des noms longs

Puisque vous faites une affirmation par test, chaque test peut être très spécifique. Ainsi, n'ayez pas peur d'utiliser des noms de test longs et complets.

Un nom long et complet vous permet de savoir immédiatement quel test a échoué et

exactement ce que le test essayait de faire.

Des tests longs et clairement nommés peuvent également documenter vos tests. Un test nommé "DividedByZeroShouldThrowException" documente exactement ce que fait le code lorsque vous essayez de diviser par zéro.

12. Tester que chaque exception soulevée est effectivement soulevée

Si votre code déclenche une exception, écrivez un test pour vous assurer que toutes les exceptions que vous avez générées sont déclenchées.

13. Évitez d'utiliser CheckTrue ou Assert.IsTrue

Évitez de vérifier une condition booléenne. Par exemple, si vous vérifiez si deux choses sont égales avec CheckTrue ou Assert.IsTrue, utilisez plutôt CheckEquals ou Assert.IsEqual. Pourquoi? À cause de ce:

CheckTrue (Attendu, Réel) Cela rapportera quelque chose comme: "Un test a échoué: le résultat attendu était Vrai mais le résultat réel était Faux."

Cela ne vous dit rien.

CheckEquals (Attendu, Réel)

Cela vous dira quelque chose comme: "Certains tests ont échoué: 7 attendus mais le résultat réel était 3."

Utilisez uniquement CheckTrue ou Assert.IsTrue lorsque votre valeur attendue est en réalité une condition booléenne.

14. Exécutez constamment vos tests

Exécutez vos tests pendant que vous écrivez du code. Vos tests doivent être rapides, ce qui vous permet de les exécuter même après des modifications mineures. Si vous ne pouvez pas exécuter vos tests dans le cadre de votre processus de développement normal, quelque chose ne va pas. Les tests unitaires sont supposés fonctionner presque instantanément. Si ce n'est pas le cas, c'est probablement parce que vous ne les exécutez pas isolément.

15. Exécutez vos tests dans le cadre de chaque génération automatisée

Tout comme vous devriez effectuer des tests pendant votre développement, ils devraient également faire partie intégrante de votre processus d'intégration continue. Un test échoué devrait signifier que votre build est cassé. Ne laissez pas les tests échoués s'attarder. Considérez cela comme un échec de construction et corrigez-le immédiatement.

Examples

Exemple de test unitaire simple en C

Pour cet exemple, nous allons tester la méthode de somme d'une calculatrice simple.

Dans cet exemple, nous allons tester l'application: ApplicationToTest. Celui-ci a une classe appelée Calc. Cette classe a une méthode Sum ().

La méthode Sum () ressemble à ceci:

```
public void Sum(int a, int b)
{
    return a + b;
}
```

Le test d'unité pour tester cette méthode ressemble à ceci:

```
[Testclass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
        //Arrange
        ApplicationToTest.Calc ClassCalc = new ApplicationToTest.Calc();
        int expectedResult = 5;

        //Act
        int result = ClassCalc.Sum(2,3);

        //Assert
        Assert.AreEqual(expectedResult, result);
    }
}
```

Lire Les règles générales pour les tests unitaires pour toutes les langues en ligne: https://riptutorial.com/fr/unit-testing/topic/9947/les-regles-generales-pour-les-tests-unitaires-pour-toutes-les-langues

Chapitre 5: Test Doubles

Remarques

Lors des tests, il est parfois utile d'utiliser un test double pour manipuler ou vérifier le comportement du système testé. Les doublons sont passés ou injectés dans la classe ou la méthode testée au lieu d'instances de code de production.

Examples

Utiliser un stub pour fournir des réponses prédéfinies

Un talon est un double test de poids léger qui fournit des réponses prédéfinies lorsque les méthodes sont appelées. Lorsqu'une classe testée repose sur une interface ou une classe de base, une classe de remplacement peut être implémentée pour les tests conformes à l'interface.

Donc, en supposant l'interface suivante,

```
public interface IRecordProvider {
    IEnumerable<Record> GetRecords();
}
```

Si la méthode suivante devait être testée

```
public bool ProcessRecord(IRecordProvider provider)
```

Une classe de stub qui implémente l'interface peut être écrite pour renvoyer des données connues à la méthode testée.

```
public class RecordProviderStub : IRecordProvider
{
    public IEnumerable<Record> GetRecords()
    {
        return new List<Record> {
            new Record { Id = 1, Flag=false, Value="First" },
            new Record { Id = 2, Flag=true, Value="Second" },
            new Record { Id = 3, Flag=false, Value="Third" }
        };
    }
}
```

Cette implémentation de stub peut alors être fournie au système testé pour influencer son comportement.

```
var stub = new RecordProviderStub();
var processed = sut.ProcessRecord(stub);
```

Utiliser un cadre moqueur comme souche

Les termes Mock et Stub peuvent souvent devenir confus. Cela s'explique en partie par le fait que de nombreux frameworks moqueurs prennent également en charge la création de stubs sans l'étape de vérification associée à Mocking.

Plutôt que d'écrire une nouvelle classe pour implémenter un stub, comme dans l'exemple "Utilisation d'un stub pour fournir des réponses prédéfinies", l'exemple peut être utilisé à la place des frameworks moqueurs.

En utilisant Moq:

```
var stub = new Mock<IRecordProvider>();
stub.Setup(provider => provider.GetRecords()).Returns(new List<Record> {
    new Record { Id = 1, Flag=false, Value="First" },
    new Record { Id = 2, Flag=true, Value="Second" },
    new Record { Id = 3, Flag=false, Value="Third" }
});
```

Cela permet d'obtenir le même comportement que le stub codé à la main et peut être fourni au système sous test de la même manière:

```
var processed = sut.ProcessRecord(stub.Object);
```

Utiliser un cadre moqueur pour valider le comportement

Les simulations sont utilisées lorsqu'il est nécessaire de vérifier les interactions entre le système testé et les tests doubles. Des précautions doivent être prises pour éviter de créer des tests trop fragiles, mais se moquer peut être particulièrement utile lorsque la méthode de test consiste simplement à orchestrer d'autres appels.

Ce test vérifie que lorsque la méthode testée est appelée (`ProcessRecord`), la méthode de service (`UseValue`) est appelée pour le `Record` où `Flag==true` . Pour ce faire, il configure un stub avec des données prédéfinies:

```
var stub = new Mock<IRecordProvider>();
stub.Setup(provider => provider.GetRecords()).Returns(new List<Record> {
    new Record { Id = 1, Flag=false, Value="First" },
    new Record { Id = 2, Flag=true, Value="Second" },
    new Record { Id = 3, Flag=false, Value="Third" }
});
```

Ensuite, il installe une maquette qui implémente l'interface `IService` :

```
var mockService = new Mock<IService>();
mockService.Setup(service => service.UseValue(It.IsAny<string>())).Returns(true);
```

Ceux-ci sont ensuite fournis au système sous test et la méthode à tester est appelée.

```
var sut = new SystemUnderTest(mockService.Object);

var processed = sut.ProcessRecord(stub.Object);
```

Le simulacre peut alors être interrogé pour vérifier que l'appel prévu a été effectué. Dans ce cas, un appel à `UseValue` , avec un paramètre "Second", qui correspond à la valeur de l'enregistrement où `Flag==true` .

```
mockService.Verify(service => service.UseValue("Second"));
```

Lire Test Doubles en ligne: https://riptutorial.com/fr/unit-testing/topic/615/test-doubles

Chapitre 6: Test unitaire des boucles (Java)

Introduction

Boucles considérées comme l'une des structures de contrôle importantes dans tout langage de programmation. Il existe différentes manières de réaliser une couverture en boucle.

Ces méthodes diffèrent en fonction du type de boucle.

Boucles simples

Boucles imbriquées

Boucles concaténées

Examples

Test en boucle unique

Ce sont des boucles dans lesquelles leur corps de boucle ne contient pas d'autres boucles (la boucle la plus interne en cas d'imbrication).

Pour avoir une couverture en boucle, les testeurs doivent exercer les tests indiqués ci-dessous.

Test 1: Concevoir un test dans lequel le corps de la boucle ne devrait pas être exécuté du tout (c.-à-d. Des itérations nulles)

Test 2: Concevoir un test dans lequel la variable de contrôle de boucle est négative (nombre négatif d'itérations)

Test 3: Concevoir un test dans lequel la boucle ne se répète qu'une seule fois

Test 4: Concevoir un test dans lequel la boucle effectue deux itérations

Test 5: Concevoir un test dans lequel la boucle répète un certain nombre de fois, par exemple m où m <nombre maximal d'itérations possible

Test 6: Concevoir un test dans lequel la boucle effectue une itération inférieure au nombre maximal d'itérations

Test 7: Concevoir un test dans quelle boucle itère le nombre maximum d'itérations

Test 8: Concevoir un test dans lequel la boucle effectue une itération de plus que le nombre maximal d'itérations

Considérons l'exemple de code ci-dessous qui applique toutes les conditions spécifiées.

classe publique SimpleLoopTest {

private int [] numbers = {5, -77,8, -11,4,1, -20,6,2,10};

```
/** Compute total of  positive numbers in the array
 *  @param numItems number of items to total.
 */
public int findSum(int numItems)
{
    int total = 0;
    if (numItems <= 10)
    {
        for (int count=0; count < numItems; count = count + 1)
        {
          if (numbers[count] > 0)
             {
                total = total + numbers[count];
             }
        }
    }
    return total;
}
```

}

la classe publique TestPass étend TestCase {

```
public void testname() throws Exception {

    SimpleLoopTest s = new SimpleLoopTest();
    assertEquals(0, s.findSum(0));    //Test 1
    assertEquals(0, s.findSum(-1));   //Test 2
    assertEquals(5, s.findSum(1));    //Test 3
    assertEquals(5, s.findSum(2));    //Test 4
    assertEquals(17, s.findSum(5));   //Test 5
    assertEquals(26, s.findSum(9));   //Test 6
    assertEquals(36, s.findSum(10));  //Test 7
    assertEquals(0, s.findSum(11));   //Test 8
}
```

}

Test des boucles imbriquées

Une boucle imbriquée est une boucle dans une boucle.

La boucle externe ne change que lorsque la boucle interne est complètement terminée / interrompue.

Dans ce cas, les cas de test doivent être conçus de telle manière que

Commencez par la boucle la plus profonde. Définissez toutes les boucles externes à leurs valeurs minimales. Effectuez un test de boucle simple sur la boucle la plus interne (Test3 / Test4 / Test5 / Test6 / Test7). Continuer jusqu'à ce que toutes les boucles soient testées

Test des boucles concaténées

Deux boucles sont concaténées s'il est possible d'en atteindre une après avoir quitté l'autre sur le même chemin depuis l'entrée jusqu'à la sortie. Parfois, ces deux boucles sont indépendantes l'une de l'autre. Dans ces cas, nous pouvons appliquer les techniques de conception spécifiées dans le cadre du test en boucle unique.

Mais si les valeurs d'itération d'une boucle sont directement ou indirectement liées aux valeurs d'itération d'une autre boucle et qu'elles peuvent se produire sur le même chemin, nous pouvons les considérer comme des boucles imbriquées.

Lire Test unitaire des boucles (Java) en ligne: https://riptutorial.com/fr/unit-testing/topic/10116/test-unitaire-des-boucles--java-

Chapitre 7: Tests unitaires: meilleures pratiques

Introduction

Un test unitaire est la plus petite partie testable d'une application, comme les fonctions, les classes, les procédures, les interfaces. Le test unitaire est une méthode par laquelle des unités individuelles de code source sont testées pour déterminer si elles sont aptes à être utilisées. Les tests unitaires sont essentiellement écrits et exécutés par les développeurs de logiciels pour s'assurer que le code correspond à sa conception et à ses exigences et se comporte comme prévu.

Examples

Bonne appellation

L'importance d'une bonne dénomination peut être mieux illustrée par quelques mauvais exemples:

```
[Test]
Test1() {...} //Cryptic name - absolutely no information

[Test]
TestFoo() {...} //Name of the function - and where can I find the expected behaviour?

[Test]
TestTFSid567843() {...} //Huh? You want me to lookup the context in the database?
```

Les bons tests nécessitent de bons noms. Un bon test ne teste pas les méthodes, les scénarios de test ou les exigences.

Une bonne dénomination fournit également des informations sur le contexte et le comportement attendu. Dans l'idéal, lorsque le test échoue sur votre machine de génération, vous devriez être capable de décider ce qui ne va pas, sans regarder le code de test, ou encore plus dur, avec la nécessité de le déboguer.

Un bon nommage vous évite d'avoir à lire du code et à déboguer:

```
[Test]
public void GetOption_WithUnkownOption_ReturnsEmptyString() {...}
[Test]
public void GetOption_WithUnknownEmptyOption_ReturnsEmptyString() {...}
```

Pour les débutants, il peut être utile de lancer le nom du test avec **EnsureThat_** ou un préfixe similaire. Commencez avec un "EnsureThat_" pour commencer à réfléchir au scénario ou à l'exigence, qui nécessite un test:

```
[Test]
public void EnsureThat_GetOption_WithUnkownOption_ReturnsEmptyString() {...}
[Test]
public void EnsureThat_GetOption_WithUnknownEmptyOption_ReturnsEmptyString() {...}
```

Le nommage est également important pour les appareils de test. Nommez le banc d'essai après la classe testée:

```
[TestFixture]
public class OptionsTests //tests for class Options
{
    ...
}
```

La conclusion finale est la suivante:

Une bonne dénomination conduit à de bons tests qui conduisent à une bonne conception du code de production.

Du simple au complexe

Identique à, avec les classes d'écriture - commencez par les cas simples, puis ajoutez la condition (aka tests) et l'implémentation (aka code de production) au cas par cas:

```
[Test]
public void EnsureThat_IsLeapYearIfDecimalMultipleOf4() {...}
[Test]
public void EnsureThat_IsNOTLeapYearIfDecimalMultipleOf100 {...}
[Test]
public void EnsureThat_IsLeapYearIfDecimalMultipleOf400 {...}
```

N'oubliez pas l'étape de refactoring, lorsque vous avez terminé avec les exigences - refactorisez d'abord le code, puis modifiez les tests

Une fois terminé, vous devriez avoir une documentation complète, à jour et lisible de votre classe.

Concept MakeSut

Testcode a les mêmes exigences de qualité que le code de production. MakeSut ()

- améliore la lisibilité
- peut être facilement restructuré
- soutient parfaitement l'injection de dépendance.

Voici le concept:

```
[Test]
public void TestSomething()
{
    var sut = MakeSut();

    string result = sut.Do();
```

```
    Assert.AreEqual("expected result", result);
}
```

Le plus simple MakeSut () renvoie simplement la classe testée:

```
private ClassUnderTest MakeSUT()
{
    return new ClassUnderTest();
}
```

Lorsque des dépendances sont nécessaires, elles peuvent être injectées ici:

```
private ScriptHandler MakeSut(ICompiler compiler = null, ILogger logger = null, string
scriptName="", string[] args = null)
{
    //default dependencies can be created here
    logger = logger ?? MockRepository.GenerateStub<ILogger>();
    ...
}
```

On pourrait dire que MakeSut est juste une alternative simple aux méthodes d'installation et de démontage fournies par les frameworks Testrunner et pourrait être une meilleure solution pour la configuration et le démontage spécifiques aux tests.

Tout le monde peut décider par lui-même de la manière à utiliser. Pour moi, MakeSut () offre une meilleure lisibilité et beaucoup plus de flexibilité. Enfin, le concept est indépendant de toute structure testrunner.

Lire Tests unitaires: meilleures pratiques en ligne: https://riptutorial.com/fr/unit-testing/topic/6074/tests-unitaires--meilleures-pratiques

Chapitre 8: Types d'assertion

Examples

Vérification d'une valeur retournée

```
[Test]
public void Calculator_Add_ReturnsSumOfTwoNumbers()
{
    Calculator calculatorUnderTest = new Calculator();

    double result = calculatorUnderTest.Add(2, 3);

    Assert.AreEqual(5, result);
}
```

Test basé sur l'état

Étant donné cette classe simple, nous pouvons tester que la méthode `ShaveHead` fonctionne correctement en affirmant que l'état de la variable `HairLength` est défini sur zéro après l' `ShaveHead` méthode `ShaveHead` .

```
public class Person
{
    public string Name;
    public int HairLength;

    public Person(string name, int hairLength)
    {
        this.Name = name;
        this.HairLength = hairLength;
    }

    public void ShaveHead()
    {
        this.HairLength = 0;
    }
}

[Test]
public void Person_ShaveHead_SetsHairLengthToZero()
{
    Person personUnderTest = new Person("Danny", 10);

    personUnderTest.ShaveHead();

    int hairLength = personUnderTest.HairLength;

    Assert.AreEqual(0, hairLength);
}
```

La vérification d'une exception est déclenchée

Parfois, il est nécessaire d'affirmer qu'une exception est lancée. Différents frameworks de tests unitaires ont des conventions différentes pour affirmer qu'une exception a été lancée (comme la méthode Assert.Throws de NUnit). Cet exemple n'utilise aucune méthode spécifique au framework, juste construite dans la gestion des exceptions.

```
[Test]
public void GetItem_NegativeNumber_ThrowsArgumentInvalidException
{
    ShoppingCart shoppingCartUnderTest = new ShoppingCart();
    shoppingCartUnderTest.Add("apple");
    shoppingCartUnderTest.Add("banana");

    double invalidItemNumber = -7;

    bool exceptionThrown = false;

    try
    {
        shoppingCartUnderTest.GetItem(invalidItemNumber);
    }
    catch(ArgumentInvalidException e)
    {
        exceptionThrown = true;
    }

    Assert.True(exceptionThrown);
}
```

Lire Types d'assertion en ligne: https://riptutorial.com/fr/unit-testing/topic/6330/types-d-assertion

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec les tests unitaires	Andrey, Carl Manaster, Community, Farukh, forsvarir, Fred Kleuver, mahei, mark_h, Quill, silver, Stephen Byrne, Thomas Weller, zhon
2	Guide des tests unitaires dans Visual Studio for C #	DarkAngel
3	Injection de dépendance	forsvarir, kayess, mrAtari, Pavel Voronin, Stephen Byrne
4	Les règles générales pour les tests unitaires pour toutes les langues	DarkAngel
5	Test Doubles	forsvarir
6	Test unitaire des boucles (Java)	Remya
7	Tests unitaires: meilleures pratiques	mrAtari, RamenChef, Shrinivas Patgar, user2314737
8	Types d'assertion	Danny