

Programmation avancée en c#.NET

Ing.Meryem OUARRACHI

Plan du module

Programmation WEB

- ☐ Généralités outils Web
- ☐ ASP MVC
- ☐ ASP.Net core
- ☐ Angular

Génie logiciel en .Net

BI en Self Service

Programmation distribuée avancée

- ☐ Web API
- ☐ GraphQL

CHAPITRE 3:

Génie logiciel en .Net

Génie logiciel en .Net

.Les tests unitaires-

C'est quoi un test unitaire?

- un test unitaire est un procédé permettant de s'assurer du fonctionnement correct d'une partie déterminée d'un logiciel ou d'une portion d'un programme (appelée « unité » ou « module »).
- Un test est dit **unitaire** s'il ne fait pas appel à d'autres ressources que la classe testé. Un test unitaire n'utilise donc pas de base de données, de socket... à l'inverse d'un test d'intégration.

A quoi sert un test unitaire?

- Assurer que notre code fonctionne.
- De plus, cela permet de faire des opérations de maintenance sur le code tout en étant certain que ce code n'aura pas subi de régressions.

Mise en œuvre des tests unitaires

Il y'a deux façons pour tester une application:

-Manuelle :Celle qui semble la plus naturelle est celle qui se fait manuellement. On lance l' application, on clique partout, on regarde si elle fonctionne.

-Automatique: Ecrire un bout de code pour tester que l'application fonctionne correctement .

Etape de test unitaire

un test unitaire est un processus en trois étapes(le concept de 3A):

- Arrange** : Initialisation
- Act** : Appel de méthode à tester
- Assert** : Vérification des résultats

Etape de test unitaire

-Arrange : C'est la première étape d'une application de test unitaire. Ici, nous allons organiser le test, en d'autres termes, nous allons faire la configuration nécessaire du test. Par exemple, pour effectuer le test, nous devons créer un objet de la classe ciblée, si nécessaire, puis nous devons créer des objets fictifs et d'autres initialisations de variables, quelque chose comme ça.

Etape de test unitaire

- **Act:** Dans cette étape, nous allons exécuter le test. En d'autres termes, nous ferons les tests unitaires réels c.à.d. ici, nous allons appeler la fonction ciblée par le test en utilisant l'objet que nous avons créé à l'étape précédente.
- **Assertion:** C'est la dernière étape d'une application de test unitaire. Dans cette étape, nous allons vérifier si le résultat retourné correspond aux résultats attendus.

Framework de test

- Un framework de test est aux tests ce que l'IDE est au développement. Il fournit un environnement structuré permettant l'exécution de test et des méthodes pour aider au développement de ceux-ci.
- Il existe plusieurs frameworks de test. Microsoft dispose de son framework, **MSTest**. D'autres framework de tests existent, comme le très connu **NUnit**.

Test unitaire en ASP WEB Forms

-C'est impossible en WebForms!

- L'interface utilisateur est étroitement couplée au code derrière.

La logique d'interaction utilisateur, la logique de présentation et la logique de transformation de données seront écrites dans le code Behind.

- Le test de code derrière n'est pas possible car le code behind contient des gestionnaires d'événements et l'appel de gestionnaire d'événements en tant que fonction sera une tâche très difficile.

Test unitaire en ASP MVC

-C'est facile à implémenter

En ASP MVC:

-Chaque requête dans Asp.Net MVC passe par Controller.

-Le contrôleur contient la logique d'interaction de l'utilisateur.

-Le contrôleur n'est pas étroitement couplé avec View

→ La structure de ASP MVC prépare un environnement adéquat pour les tests

Mise en place d'un test unitaire

On a toujours deux éléments:

- Le projet à tester
- Le projet de test

- En VS Créer un nouveau projet (exemple ASPMVC)
- Ajouter à la même solution un nouveau projet de type « Projet de test unitaire »
- Dans le projet de test unitaire ajouter une référence de projet précédent(projet1)

Mise en place d'un test unitaire

- **Structure de classe de test :**

```
namespace UnitTestProject1
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

Mise en place d'un test unitaire

- Méthode à tester:

```
public class CalculController : Controller
{
    public int Somme(int nb1,int nb2)
    {
        return nb1 + nb2;
    }
}
```


Mise en place d'un test unitaire

- Méthode de test:

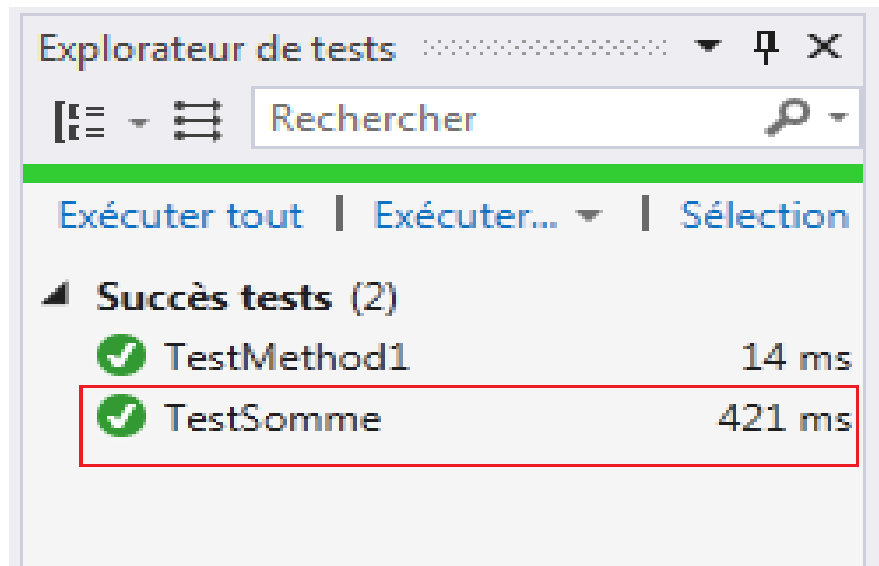
```
[TestMethod]
public void TestSomme()
{
    //Arrange
    CalculController c = new CalculController();
    int num1 = 10;
    int num2 = 30;
    int som = 0;
    int ValeurAttendue = 40;

    //Act
    som = c.Somme(num1, num2);

    //Assert
    Assert.AreEqual(ValeurAttendue, som);
}
```

Mise en place d'un test unitaire

-**Exécution:** Test → Exécuter → Tous les tests



Les assertions

- Dans un test, on cherche toujours à vérifier un comportement. Pour cela on utilise les assertions qui servent à vérifier les conditions dans les tests unitaires à l'aide des propositions true/false
- Donc Il s'agit d'une expression qui doit être vrai pour que le test réussisse.
- Cela est assurée par des classes **Assert** qui vont nous permettre de tester entre autre :l'égalité : `Assert.Equals` /le non null : `Assert.IsNotNull` /une condition `Assert.IsTrue`...

La classe Assert

Méthode	Description
AreEqual	Vérifie l'égalité de deux éléments
AreNotEqual	Vérifie que deux éléments spécifiées ne sont pas égaux
AreSame	Vérifie que deux variables objets spécifiées font référence au même objet.
AreNotSame	Vérifie que deux variables objets spécifiées font référence à des objets différents.
IsTrue	Vérifie que la condition spécifiée est true
IsFalse	Vérifie que la condition spécifiée est false
IsNotNull	Vérifie que l'objet spécifié est null

<https://msdn.microsoft.com/fr-fr/library/microsoft.visualstudio.testtools.unittesting.assert.aspx>

La classe Assert

- Exemple:

```
[TestMethod]
public void TestAssert()
{
    double attendu = 3;
    double actuel = 4;
    //true
    Assert.AreNotEqual(attendu, actuel);
    //false
    Assert.AreEqual(attendu, actuel, "des valeurs différentes");
    //true
    Assert.IsTrue(actuel >= attendu);
    //true
    Assert.IsNotNull(attendu);
    //false
    Assert.IsInstanceOfType(attendu, typeof(int));
}
```

La classe CollectionAssert

-Vérifie des propositions true/false associées aux collections dans les tests unitaires.

Fonction	Description
AreEqual	Vérifie que deux collections spécifiées sont égales
AllItemsAreUnique	Vérifie que tous les éléments de la collection spécifiée sont uniques
Contains	Vérifie que la collection spécifiée contient l'élément spécifié.
AllItemsAreInstanceOf Type()	Vérifie que tous les éléments de la collection spécifiée sont des instances du type spécifié
AllItemsAreNotNull	Vérifie que tous les éléments de la collection spécifiée ne sont pas null.

<https://msdn.microsoft.com/fr-fr/library/microsoft.visualstudio.testtools.unittesting.collectionassert.aspx>

La classe CollectionAssert

-Exemple:

```
[TestMethod]
public void TestCollection()
{
    List<string> first = new List<string>();
    first.Add("a"); first.Add("b");
    List<string> second = new List<string>();
    second.Add("c");
    //false
    CollectionAssert.AreEqual(first, second);
    //true
    CollectionAssert.AllItemsAreUnique(first);
    //true
    CollectionAssert.Contains(first, "a");
    //true
    CollectionAssert.DoesNotContain(first, "x");
    //true
    CollectionAssert.AllItemsAreInstancesOfType(first, typeof(string));
    //true
    CollectionAssert.AllItemsAreNotNull(first);
}
```

ExpectedException

C'est un attribut utilisé lorsque nous savons qu'une fonction peut déclencher une sorte d'exception

```
public int Devide(int a, int b)
{
    return a / b;
}
```

```
[TestMethod]
public void TestDivision()
{
    CalculController ObjMath = new CalculController();

    int Result = ObjMath.Devide(10, 0);

    Assert.AreEqual(10, Result);
}
```


ExpectedException

TestDivision

[Copier tout](#)

Source : [UnitTest1.cs](#) ligne 81

❌ Test Échec - TestDivision

Message : La méthode de test
UnitTestProject1.UnitTest1.TestDivision a levé une
exception :
System.DivideByZeroException: Tentative de division par
zéro.

```
[TestMethod]
[ExpectedException(typeof(DivideByZeroException), "erreur")]
public void TestDivision()
{
    CalculController ObjMath = new CalculController();

    int Result = ObjMath.Devide(10, 0);

    Assert.AreEqual(10, Result);
}
```

Maintenant ,la méthode sait que si la méthode ciblée renvoie une exception "Divide by zero", alors c'est notre exception attendue.

Test dans les projets ASP MVC

Les tests d'interaction avec l'utilisateur Asp.Net MVC ou les tests du contrôleur consistent à faire des tests pour:

- ✓ ViewResult
- ✓ ViewData / ViewBag
- ✓ RedirectResult

Test dans les projets ASP MVC

- Test de `viewResult`:

```
public ActionResult GetView(int id)
{
    if (id == 0)
    {
        return View("View1");
    }
    else
    {
        return View("View2");
    }
}
```

Test dans les projets ASP MVC

-Méthode de test:

```
[TestMethod]
public void TestForViewWithValue0()
{
    //Arrange
    CalculController t = new CalculController();
    //Act
    ViewResult r = t.GetView(0) as ViewResult;
    //Assert
    Assert.AreEqual("View1", r.ViewName);
}

[TestMethod]
public void TestForViewWithValueOtherThanZero()
{
    //Arrange
    CalculController t = new CalculController();
    //Act
    ViewResult r = t.GetView(1) as ViewResult;
    //Assert
    Assert.AreEqual("View2", r.ViewName);
}
```

Test dans les projets ASP MVC

- Test de ViewData:

```
public ActionResult ActionViewData()
{
    ViewData["Name"] = "xxx";
    return View();
}
```

- Méthode de test:

```
[TestMethod]
public void TestForViewData()
{
    //Arrange
    CalculController t = new CalculController();

    //Act
    ViewResult r = t.Action2() as ViewResult;

    //Assert
    Assert.AreEqual("xxx", r.ViewData["Name"]);
}
```

Test dans les projets ASP MVC

- **Test de Redirection:**

```
public ActionResult Details(int id)
{
    if (id < 1)
        return RedirectToAction("Index", "Controlleur2");

    return View("Details");
}
```

Test dans les projets ASP MVC

-Méthode de test:

```
[TestMethod]
public void TestDetailsForRedirect()
{
    C1Controller controllerx = new C1Controller();
    var result = controllerx.Details(-1) as RedirectToRouteResult;
    Assert.AreEqual("Index", result.RouteValues["Action"]);
    Assert.AreEqual("Controlleur2", result.RouteValues["Controller"]);
}

[TestMethod]
public void TestDetailsForViewResult()
{
    C1Controller controller = new C1Controller();
    ViewResult result1 = (ViewResult)controller.Details(2);
    Assert.AreEqual("Details", result1.ViewName);
}
```

Test unitaire et le IOC

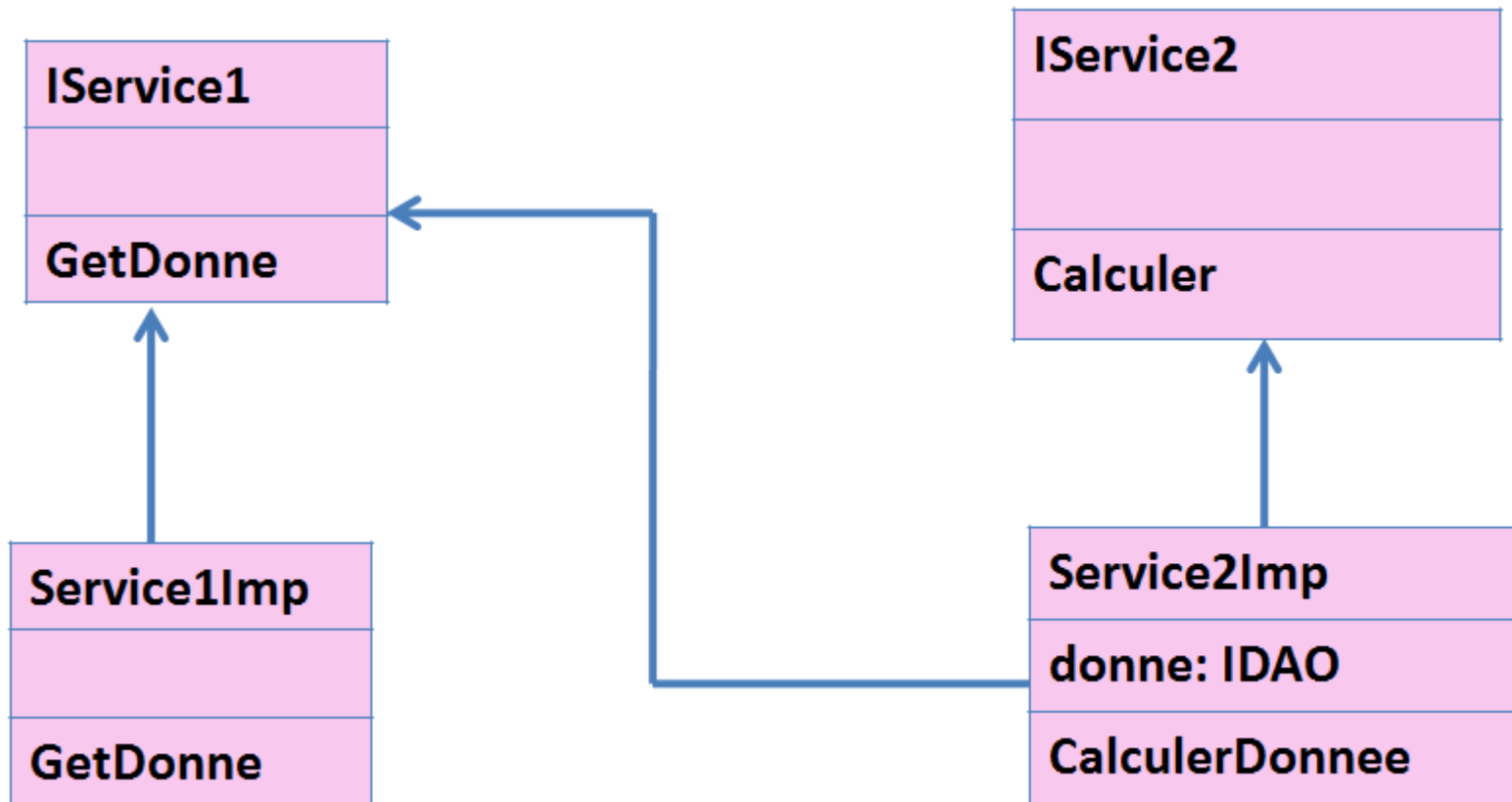
Soit l'architecture suivante:



-Service2 contient une méthode qui utilise des données récupérées à partir d'une méthode de service1

Test unitaire et le IOC

L'implémentation de l'architecture précédente:



Test unitaire et le IOC

•Service1

```
public interface IService1
{
    int GetDonne(int cle);
}
```

```
public class Service1Imp
{
    private static Dictionary<int, int> ValeurParametre = new Dictionary<int, int>
    {
        { 1, 10 },
        { 3, 100 },
        { 4, 27 },
        { 5, 31 }
    };

    //méthode qui récupère la valeur selon l'index
    int GetDonne(int cle)
    {
        return (int)(ValeurParametre.ContainsKey(cle) ? ValeurParametre[cle] : 1);
    }
}
```

Test unitaire et le IOC

•Service2

```
public interface IService2
{
    int calcul(int cle, int x);
}
```

```
public class Service2Impl
{
    IService1 _s;
    public Service2Impl(IService1 s)
    {
        _s = s;
    }
    public int calcul(int cle, int x)
    {
        return x*(_s.GetDonne(cle));
    }
}
```

Test unitaire et le IOC

Problématique: tester la méthode 'calcul' de service2 sans faire un test de la méthode 'GetDonne' de service1

« Car un test est dit **unitaire** s'il ne fait pas appel à d'autres ressources que le module testé »

Solution: Utiliser la notion des objets faux « **Fakes objets** » pour simuler l'application de test

Faking

- C'est le fait de créer une implémentation de test d'une classe qui peut dépendre d'une infrastructure externe. (Il faut que le test d'unité n'interfère pas avec l'infrastructure externe.)
- Autrement dit créer une implémentation de test d'*une classe* dont ses éléments utilisés par *une autre classe* qui représente l'objectif de test

Exemple:

- Pour l'exemple précédent on va créer une classe qui implémente l'interface IService1 et on va redéfinir la fonction GetDonnee de telle façon, me donne un résultat que l'on peut l'exploiter dans la méthode calcul

Faking

- **Etape1:** Définition de la classe fakes

```
class Stub : IService1
{
    public int GetDonne(int cle)
    {
        if (cle == 1)
            return 10;
        return 1;
    }
}
```

Faking

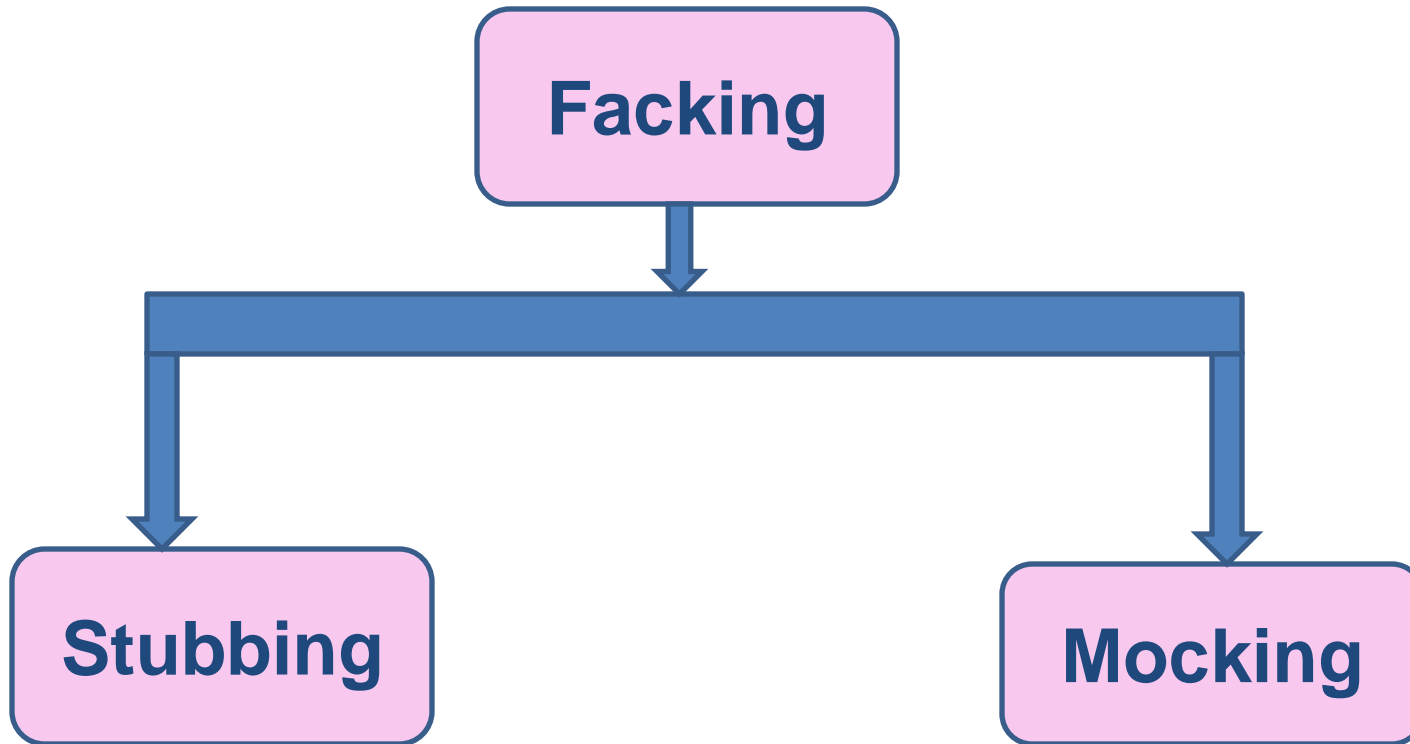
- **Etape2:** Définition de la méthode de test en utilisant un faux objet

```
[TestMethod]
public void TestMethod1()
{
    //Arrange
    const int cle = 1;
    var stub = new Stub();

    //Act
    var x = new Service2Impl(stub);
    var resultat = x.calcul(cle, 3);

    //Assert
    Assert.AreEqual(30, resultat);
}
```

Faking, Stubbing, Mocking



Faking, Stubbing, Mocking

- **Facke:** classe qui implémente une interface mais contient des données fixes et aucune logique. Renvoie simplement les données "bonnes" ou "mauvaises" en fonction de l'implémentation..

Exemple: Créer une fausse implémentation pour accéder à une base de données, la remplacer par une collection en mémoire.

Faking, Stubbing, Mocking

- **Stub:** remplace les méthodes pour renvoyer des valeurs codées en dur

Exemple1: l'exemple précédent

Exemple2: Notre classe de test dépend d'une méthode

Calculate () prenant 5 minutes à compléter. Plutôt que d'attendre 5 minutes, nous pouvons remplacer son implémentation réelle par un stub qui renvoie des valeurs codées en dur; prendre seulement une petite fraction du temps.

Faking, Stubbing, Mocking

- **Mock:** très similaire à Stub mais basé sur l'interaction plutôt que sur l'état. Cela signifie que nous n'attendons pas à ce que Mock retourne une valeur, mais nous voudrions vérifier que l'ordre spécifique des appels de méthode est correct.

- Càd Mock permet de vérifier si des méthodes particulières ont été appelées / non appelées.

Exemple: nous testons une classe d'inscription d'utilisateur. Après avoir appelé Enregistrer, il doit appeler SendConfirmationEmail.

Mock

- Faire un mock pour l'exemple précédent afin de vérifier si la méthode GetDonne est appelé lors de l'exécution de la méthode calcul de service1

Etape1: Définition de la classe fakes

```
class Mock : IService1
{
    public bool IsCalled = false;
    public int GetDonne(int cle)
    {
        IsCalled = true;

        return 1;
    }
}
```

Mock

- **Etape2:** Définition de la méthode de test en utilisant un faux objet de type Mock

```
[TestMethod]
public void TestMethod2()
{ //Arrange
    const int cle = 1;
    var m = new Mock();

    //Act
    var x = new Service2Impl(m);
    var resultat = x.calcul(cle, 3);

    //Assert
    Assert.IsTrue(m.IsCalled);
}
```

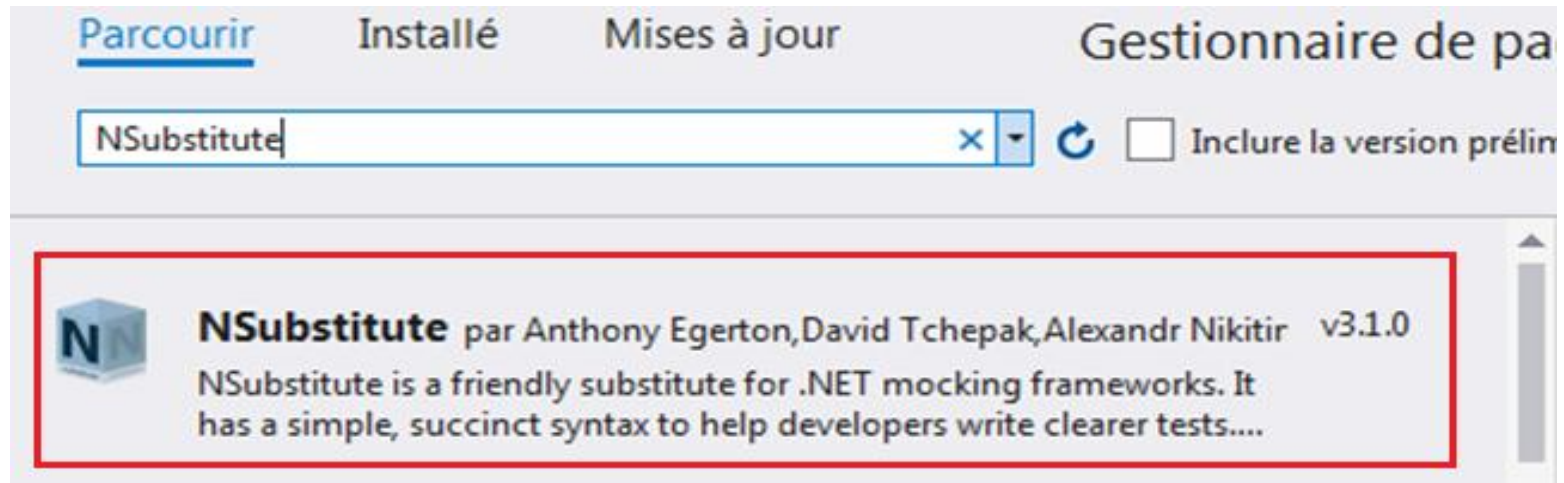
Framework de mocking et stubbing

- A chaque fois on veut faire un stub ou un mock on doit définir manuellement une classe fake ce qui rend le travail lourd
- C'est pour cela on pourra passer par un framework qui nous génère les méthodes de stub et mock d'une manière automatique
- Exemple des framework: NSubstitute, Moq, FakeItEasy, Rhino Mocks, NMock3

Framework NSubstitute

-Framework qui permet de générer automatiquement un stub et un mock

-Installation:



-Site officiel

<http://nsubstitute.github.io/help.html>

Framework NSubstitute

❑ Faire un Stubbing

-Etape1: Instancier la classe Fake générée par le framework en utilisant la méthode **For**

-Etape2: Déterminer les valeurs à récupérer qui vont être utilisées par la méthode qui représente l'objectif de test en utilisant les méthodes d'extensions **Return** de Nsubstitute

Framework NSubstitute

❑ Faire un stubbing

```
[TestClass]
public class UnitTest3
{
    IService1 _stub;
    public UnitTest3()
    {
        _stub = Substitute.For<IService1>();//Create
    }

    [TestMethod]
    public void TestMethod1()
    {
        //Arrange
        const int cle = 1;
        _stub.GetDonne(cle).Returns(10);//Set a return value:

        //Act
        var x = new Service2Impl(_stub);
        var resultat = x.calcul(cle, 3);
        //Assert
        Assert.AreEqual(30, resultat);
    }
}
```

Framework NSubstitute

❑ Faire un Mocking

-Etape1: Instancier la classe Fake générée par le framework

-Etape2: Utiliser les méthodes d'extensions **Receive** afin de vérifier si une méthode est appelée ou non

Framework NSubstitute

❑ Faire un mocking

```
public class UnitTest4
{
    IService1 _stub;
    public UnitTest4()
    {
        _stub = Substitute.For<IService1>();//Create
    }

    [TestMethod]
    public void TestMethodMock()
    {
        //Arrange
        const int cle = 1;
        _stub.GetDonne(cle).Returns(10);//Set a return value:

        //Act
        var x = new Service2Impl(_stub);
        var resultat = x.calcul(cle, 3);
        //Assert
        //Vérifier que la méthode de création est bien appelé
        _stub.Received().GetDonne(cle);
    }
}
```

Framework NSubstitute

❑ Faire un mocking

- `_stub.Received().GetDonne(cle):` vérifie que la méthode `GetDonne` est appelée avec les paramètres spécifiés
- `_stub.Received().GetDonne(Arg.Any<int>()):` vérifie que la méthode `GetDonne` est appelée avec n'importe quel argument de type `int`
- `_stub.Received(2).GetDonne(cle):` vérifie si la méthode `GetDonne` est appelée 2 fois .
- `_stub.DidNotReceive().GetDonne(cle):` vérifie que la méthode `GetDonne` n'est pas appelée

La couverture de code

Après la création des tests unitaires penser à utiliser un outil de couverture de code :C'est une métrique utilisée en génie logiciel pour décrire le taux de code source testé d'un programme. Ceci permet de mesurer la qualité des tests effectués.



-En .NET on utilise **NCover** pour mesurer cette couverture de code. NCover va calculer à partir des tests joués et du code initial l'ensemble des chemins qui ont été parcourus et fournir un pourcentage du code parcouru par rapport au code non parcouru.






















-Lien officiel de Ncover:

<https://www.ncover.com/support/docs/desktop/visual-studio-extension/vse-quick-start>

La couverture de code

- Exemple des rapports générés par cet outil

Project	Sequence Points			Branch Points			C. C.
	Acceptable	U.V.	Coverage	Acceptable	U.V.	Coverage	Avg / Max
New Project	95.0 %	64	73.3 % 	95.0 %	21	78.1 % 	1.6 / 6

Module	Sequence Points			Branch Points			C. C.
	Acceptable	U.V.	Coverage	Acceptable	U.V.	Coverage	Avg / Max
BusinessObjects	95.0 %	11	87.9 % 	95.0 %	16	77.8 % 	1.6 / 3
Namespace / Classes							
BusinessObjects	95.0 %	11	87.9 % 	95.0 %	16	77.8 % 	1.6 / 3
Customer	95.0 %	1	91.7 % 	95.0 %	1	88.9 % 	1.4 / 3
get_CustomerId		0	100.0 % 		0	100.0 % 	1
get_CustomerName		0	100.0 % 		0	100.0 % 	1
set_CustomerId		0	100.0 % 		0	100.0 % 	1
set_CustomerName		1	83.3 % 		1	80.0 % 	3
Order	95.0 %	3	90.9 % 	95.0 %	3	85.0 % 	1.4 / 2
<TotalCost>b__4		0	100.0 % 		0	N/A	1
<TotalItems>b__0		0	100.0 % 		0	N/A	1
<TotalPrice>b__2		0	100.0 % 		0	N/A	1
ClearCustomer		0	100.0 % 		0	100.0 % 	1