Concepteur Développeur en Informatique

Développer une application x-tiers

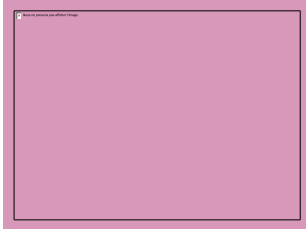
Introduction à la création de simulacres pour tests

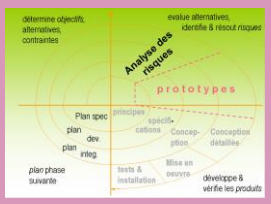
Accueil

Apprentissage

PAE

Evaluation





Localisation : U04-E02-S01

SOMMAIRE

1. Introduction

Il n'est pas toujours possible, et encore moins souhaitable, d'envisager de tester nos composants avec les dépendances réelles nécessaires à la bonne réalisation des opérations qu'ils implémentent.

Ainsi, si un composant de la couche logique métier détient une référence sur un composant assurant la persistance, nous devons exclure de le tester en mettant en œuvre ces mécanismes de sauvegarde d'états.

Imaginons que dans notre scénario de tests nous envisagions de tester la méthode qui créée un nouvel objet entité. Si nous demandons que l'état de ce dernier soit persisté dans la base de données, une deuxième exécution de ce même test conduira à une exception levée pour présence d'un doublon dans la base.

Le test n'est alors plus reproductible. Son résultat dépend d'un élément extérieur, la base de données.

Pour tester unitairement un composant, nous devons le faire dans un contexte isolé qui ne sera pas soumis à des changements de contexte extérieurs.

Dans l'exemple précédent, nous serons amenés à remplacer les composants de la couche d'accès aux données par des composants factices en lieu et place du composant d'accès aux données.

Le recours à des objets factices permet de s'assurer du caractère reproductif des tests : nous pouvons nous assurer d'exécuter les tests toujours dans les mêmes conditions et d'obtenir ainsi toujours les mêmes résultats.

Vous trouverez dans la littérature des experts qui s'évertuent à distinguer l'objet Mock du Stub ou du Fake. Je retiens pour ma part que l'objectif réside toujours dans la substitution d'un objet réel par un objet factice.

La plupart s'accorde sur le fait que le Mock serait le type de simulacre offrant le plus de possibilités. Nous pouvons le programmer dynamiquement pour des scénarios où l'objet factice retourne des objets ou valeurs différenciés en fonction des entrées. Là où le stub quant à lui se contenterai de retourner des valeurs constantes.

Le mock dispose donc de fonctionnalités supplémentaires par rapport au stub simple qui sont :

- Un paramétrage dynamique lors de l'exécution
- La possibilité de vérifier que l'objet a bien été utilisé comme prévu. On peut ainsi vérifier que la méthode testée est bien été invoquée.

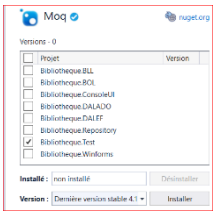
Il faut bien comprendre que le Mock n'est pas la cible du test mais permet de préciser des conditions susceptibles de produire le test.

Les objets Mock sont toujours définis sur la base d'une abstraction, classe ou interface.

2. Installer le produit de « mocking »

Comme les frameworks d'injection ou de testing unit, ils sont très nombreux, certains présents dans différentes architectures logicielles.

Nous utiliserons le produit open source Moq qui présente l'avantage de pouvoir manipuler des génériques et utiliser les expressions lambdas.



Le produit de mocking s'installe dans le projet de tests.

3. Un premier exemple de recours aux simulacres

Dans notre application en couches, nous souhaitons tester les couches BOL et BLL. Pour les raisons évoquées précédemment, nous devrons nous abstenir de tenir compte de la couche de persistance. Nous la remplacerons par un objet simulacre.

Il n'est possible de « mocker » un objet que si celui-ci implémente une interface ou hérite d'une classe abstraite. Nous pouvons éventuellement simuler des méthodes d'objets concrets si ces dernières ont été définies comme virtuelles.

Nous allons ici comprendre l'intérêt de l'inversion de contrôle mis en œuvre lors des ateliers précédents.

Nous allons réaliser un premier exemple qui pose les bases du test d'un objet de type contrôleur (manager) de la couche BLL, ici AdherentBLL.

En lieu et place de la couche de données, nous allons créer un simulacre basé sur l'interface du Repository IAdherentRepository.

Notre classe de tests doit donc connaître :

1. L'assemblage de description des objets métiers Bibliotheque.BOL
2. L'assemblage de description des objets de la couche logique métier Bibliotheque.BLL
3. L'assemblage du Repository
4. Les composants pour tests unitaires et moq

Soit, dans la classe de test, les directives using suivantes :

```
using Bibliotheque.BOL;
using Bibliotheque.BLL;
using Bibliotheque.Repository;
using NUnit.Framework;
using Moq;
```

Définition de l'objet factice qui se substituera au repository associé à la DAL.

```
Mock<IAdherentRepository> mockAdhRepo =
    new Mock<IAdherentRepository>();
```

Nous souhaitons tester la méthode Lister du composant AdherentManager.
Cette méthode produit une liste d'adhérents que nous préparons ici.

```
List<Adherent> adherents = new List<Adherent>()
{
    new Adherent { NomAdherent="Bost",PrenomAdherent="Vincent",IdAdherent="1234567"},
    new Adherent { NomAdherent="Arnaud Puyraveau",PrenomAdherent="Romain",IdAdherent="1234568"}
};
```

Nous devons maintenant configurer les membres du Mock qui seront sollicités lors du test de la méthode. Il s'agit ici d'associer à la méthode Lister la liste préparée.

```
mockAdhRepo.Setup(adhR => adhR.ListerAdherents())
    .Returns(adherents);
```

Nous instancions ensuite le composant à tester en lui injectant le simulacre et invoquons la méthode que nous souhaitons tester.

```
AdherentManager adherentManager =
    new AdherentManager(mockAdhRepo.Object);

var res = adherentManager.Lister();
```

Le résultat du test est ici sans grand intérêt ...Si ce n'est que nous sommes assurés que la méthode configurée a été invoquée. Vous remarquerez la possibilité de produire un message qui pourra facilement consultable dans le résultat des tests quand celui-ci échoue.

```
Assert.AreSame(adherents, res, "Les objets ne sont pas égaux");
```

De plus nous pouvons ici nous assurer que la méthode configurée ait bien été invoquée. Il suffit d'invoquer la méthode Verify de l'objet Mock en précisant la méthode tracée.

Cette action doit être définie après l'exécution de la méthode testée.

```
mockAdhRepo.Verify(ad => ad.ListerAdherents());
```

Si la méthode ListerAdherents n'a pas été invoquée lors du test, une exception sera levée.

4. Quelques précisions sur la manipulation des simulacres

Le Framework Moq permet l'usage de types génériques et des expressions lambdas.

4.1. Instanciation du mock

Les objets mock peuvent être instanciés en précisant le comportement (behavior) de l'objet factice. L'énumération `MockBehavior` comporte deux valeurs, Loose ou Strict.

- Loose, option par défaut, aura pour conséquence que l'invocation d'une méthode ou propriété de l'objet factice qui n'aurait pas été configurée ne provoquera pas d'exception. Elle retournera alors la valeur par défaut du type.
- L'autre option de l'énumération, Strict, conduit à la génération d'une exception en cas de sollicitation d'un membre non configuré d'un objet Mock.

Dans l'exemple précédent, si la méthode ListerAdherents n'a pas été configurée, elle retournera null, valeur par défaut, et fera échouer le test.

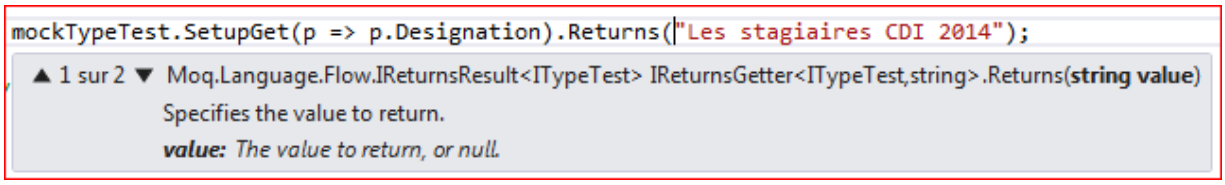
Pour éviter ce comportement, lors de l'instanciation, indiquer un comportement strict. L'invocation d'une méthode non configurée provoquera alors une exception.

```
Mock<IAdherentRepository> mockAdhRepo =
    new Mock<IAdherentRepository>(MockBehavior.Strict);
```

4.2. Configurer l'accesseur Get

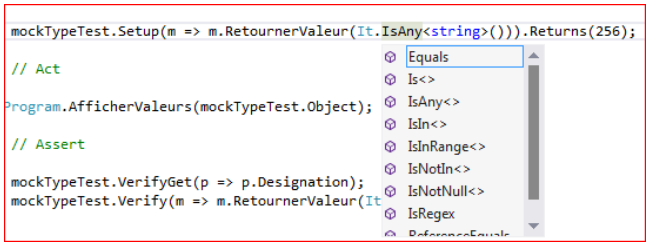
Nous allons ici paramétrer notre mock pour qu'il retourne des valeurs prédéfinies lors de l'accès à une propriété ou lors de l'invocation d'une méthode.

Dans l'exemple suivant, lorsque le composant à tester accédera en lecture à la propriété Designation, une chaine constante sera retournée.



4.3. Paramétrage d'une méthode

Dans cet exemple, lorsque le composant à tester invoquera la méthode, celle-ci renverra une valeur entière constante quelle que soit la valeur du paramètre.
It permet de faire référence au paramètre. Vous pouvez donc envisager



Mais nous pourrions différencier les valeurs retournées en fonction de la valeur du paramètre :

```
mockTypeTest.Setup(m => m.RetournerValeur(It.IsInRange("A", "Z", Moq.Range.Inclusive))).Returns(128);
```

Ici, si la valeur du paramètre est comprise entre « A » et « Z » la méthode retourne 128.

4.4. Vérifier que les options configurées soient utilisées

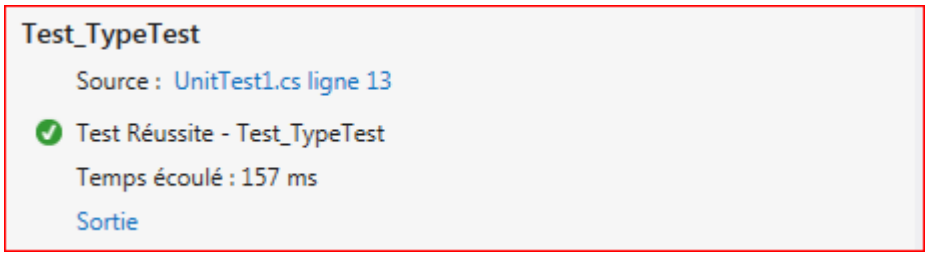
Il peut s'avérer utile de vérifier que les options programmées du mock aient bien été invoquées.Vous disposez des méthodes Verify implémentées sous diverses formes à cette fin.

```
// Assert

mockTypeTest.VerifyGet(p => p.Designation);
mockTypeTest.Verify(m => m.RetournerValeur(It.IsAny<string>()));
```

La méthode testée réalise bien un accès à la propriété Designation et exécute la méthode RetournerValeur.

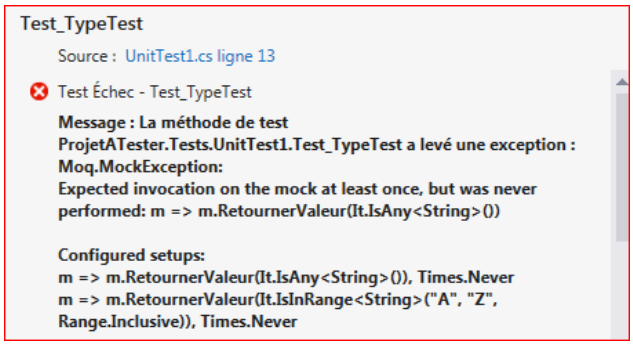
Le test est alors passé avec succès :



Que se passe-t-il alors si je commente l'appelle à la méthode ?

```
public static void AfficherValeurs(ITypeTest oTypeTest)
{
    Console.WriteLine(oTypeTest.Designation);
//  Console.WriteLine(oTypeTest.RetournerValeur("CDI"));
}
```

Nouvelle exécution des tests :



Le test n'est pas réussi.

Cette approche permet de s'assurer que les options programmées lors du setup ont bien été exécutées.

Il existe une méthode VerifyAll() qui permet de s'assurer que tous les comportements configurés ont bien été sollicités.

```
mockTypeTest.VerifyAll();
```

5. Référence de la documentation

La documentation sur le framework Moq se trouve ici :

https://github.com/Moq/moq4/wiki/Quickstart