Tests & validation

LES DOUBLURES DE TESTS

Séance 4

Tester des comportements en isolation

LES TESTS UNITAIRES

Problématique

Comme quand on développe, quand on cherche à tester il faut isoler les comportements et fonctionnalités.

Comment faire quand on cherche à valider le comportement d'un objet qui repose sur un ou plusieurs collaborateurs?

C'est à cette problématique que répondent les « doublures » ou « test doubles ».

Exemple

```
public class CommandeTestEtat
{
    private static string BIERE = "Licorne Black";
    private static string SOFT = "Carola";
    private IAssoEtudiante stockBDE = new BDE();
    0 références
    public CommandeTestEtat()
    {
        stockBDE.Add(SOFT, 50);
        stockBDE.Add(BIERE, 25);
    }
    [Fact]
    0 références
    public void TestOrderIsFilledIfEnoughInWarehouse()
    {
        Commande commande = new Commande(SOFT, 50);
        commande.Passer(stockBDE);
        Assert.True(commande.EstValidee);
        Assert.Equal(0, stockBDE.GetInventory(SOFT));
    }
    [Fact]
    0 références
    public void TestOrderDoesNotRemoveIfNotEnough()
    {
        Commande commande = new Commande(SOFT, 51);
        commande.Passer(stockBDE);
        Assert.False(commande.EstValidee);
        Assert.Equal(50, stockBDE.GetInventory(SOFT));
    }
}
```

Ici, on a une classe de tests qui cherche à vérifier l'état de notre classe Commande après qu'elle ait été passée.

Commande est donc notre SUT, mais elle a besoin de la classe BDE pour fonctionner. BDE est donc un collaborateur de Commande.

On fait ici un test par vérification d'état de notre objet commande après avoir fait un traitement dessus.

Problématique

Dans l'exemple précédent, le collaborateur appartient à notre domaine, et il fait partie de notre périmètre fonctionnel et technique. On a donc accès à ses sources, et on peut faire le test en mode « boîte blanche ».

On peut donc se permettre de faire un test en vérifiant l'état de notre SUT en utilisant des collaborateurs « réels ».

Mais comment faire quand le collaborateur fait appel à un système externe (web services, BDD, filesystem…) ? Ou quand il appartient à une bibliothèque externe?

C'est à cette problématique que répondent les doublures de tests.

Les différents types de doublures

DÉFINITIONS

Les doublures de test

LES DUMMY

Les dummy

- C’est le type de doublure le plus simple.
- Ce sont des objets qui sont passés pour remplir les listes de paramètres.
- A l’exécution, ils ne seront pas appelés.

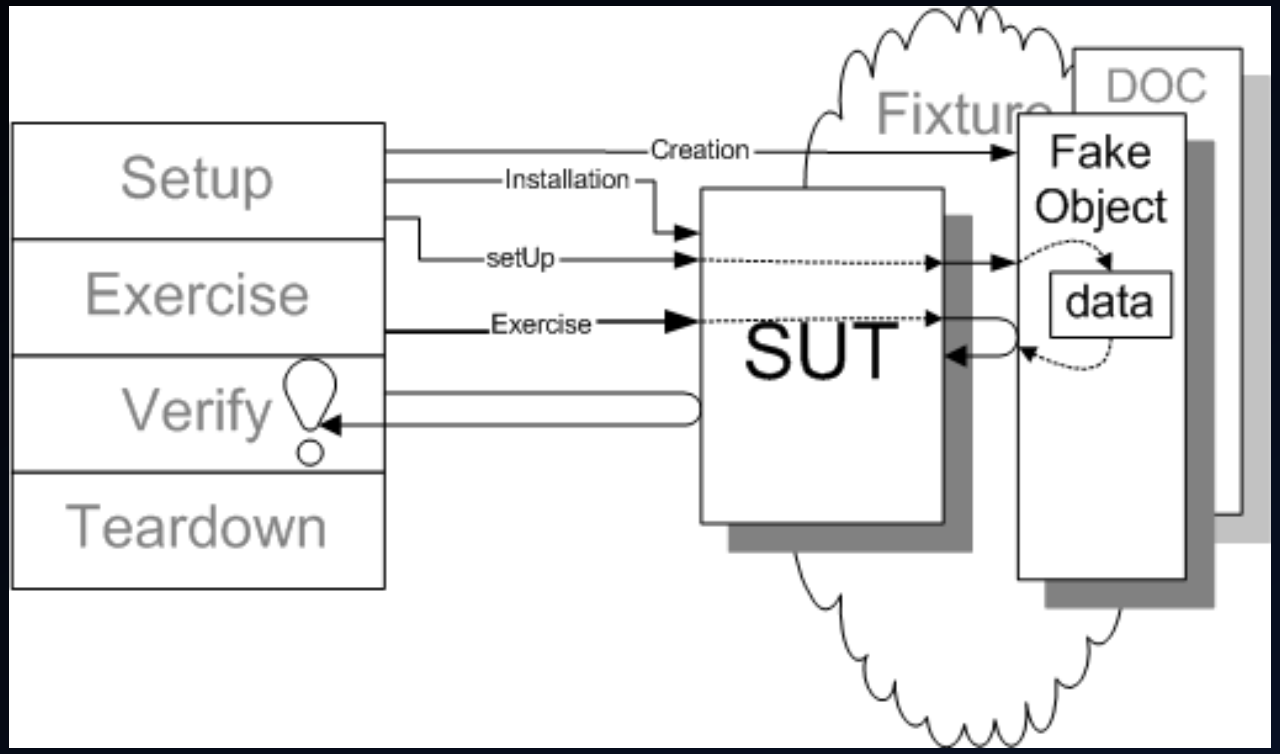
Les doublures de test

LES FAKES

Les fakes

- Ce sont des objets qui ont une implémentation réelle, mais qui est trop simplifiée par rapport à l'implémentation nécessaire pour le code de production.
- Ils simulent des collaborateurs qui ont un comportement attendu. Ce seront souvent les collaborateurs de fin de traitement.
- Peut les utiliser pour simuler une écriture dans une base de données ou un fichier sans le faire réellement.

Les fakes



Ils sont spécialisés pour la récupération de données qui ont été manipulées par un collaborateur comme :
BDD, WebService, Service layer…

Source : http://xunitpatterns.com/Fake%20Object.html

Les doublures de test

LES STUBS

Les stubs

- Ce sont des objets qui sont là pour fournir des données qui seront nécessaires à la réalisation d'un traitement.
- Ils simulent des collaborateurs qui servent à fournir une donnée qui n'est pas présente dans le contexte d'exécution.
- Par exemple on les utilisera pour remplacer un appel à un service externe (service web, API Rest, etc.), ou un appel à la base de données.

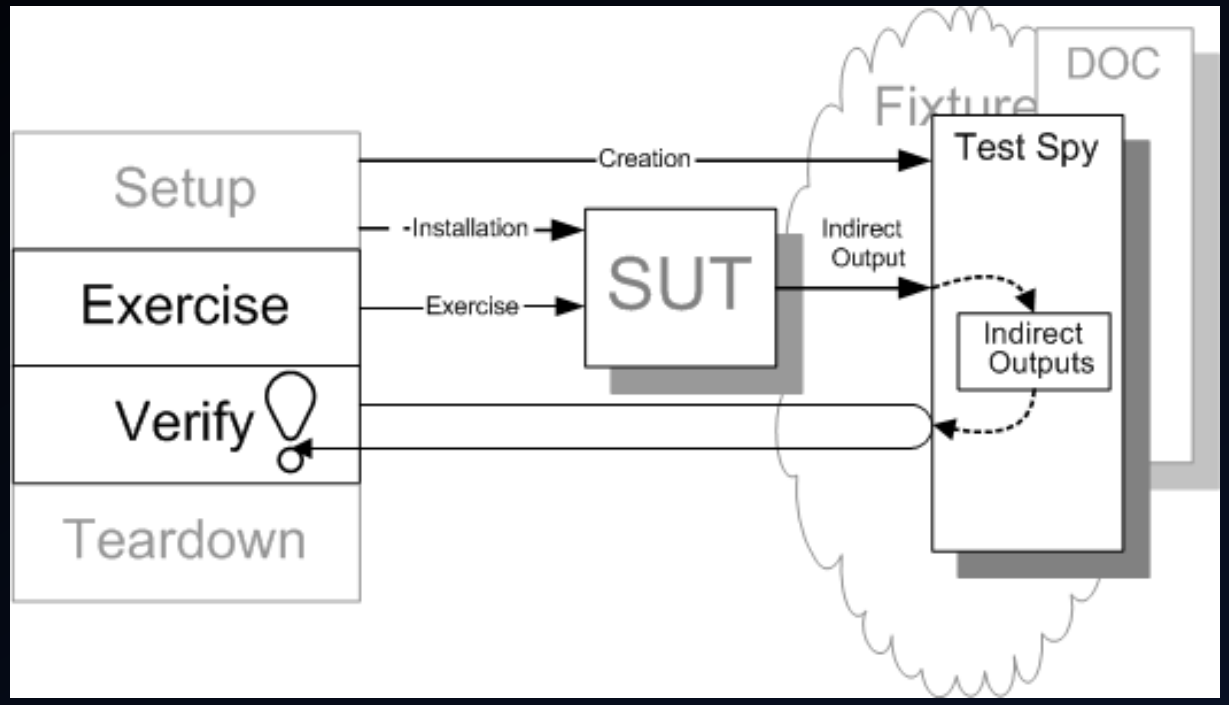
Les doublures de test

LES SPIES

Les spies

- Ce sont des objets qui sont là pour espionner des traitements et enregistrer des données à des états intermédiaires de la réalisation d’un traitement.
- Ils simulent des collaborateurs qui sont impliqués dans une ou plusieurs étapes d’un traitement.
- Ils servent de point d’observation dans les étapes de traitements de notre SUT.

Les spies



Ici les vérifications peuvent s'effectuer directement sur la doublure de type SPY.
On peut coder un spy pour enregistrer des données, ou pour logger combien d'appels ont été reçus par exemple.

Source : http://xunitpatterns.com/Test%20Spy.html

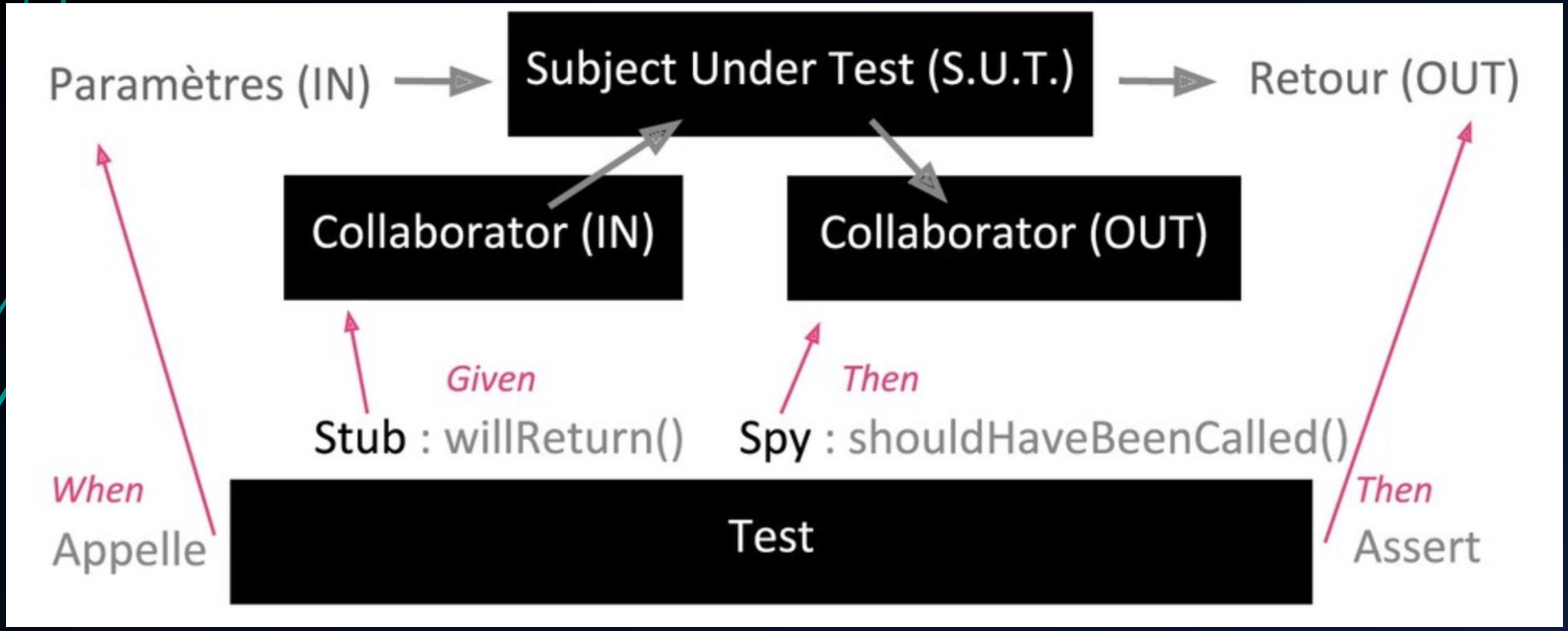
Les doublures de test

LES MOCKS

Les mocks

- Ce sont des objets qui sont là pour fournir un collaborateur qui a un comportement déclenché par notre SUT.
- Ils simulent des collaborateurs et permettent de récupérer les appels qu'ils ont reçu. On vérifiera donc avec eux des comportements.
- Par exemple on les utilisera dans le cas où on cherche à vérifier qu'un traitement est bien déclenché, mais sans vérifier la validité de ce comportement.

Les différentes doublures



Créer un contexte d'exécution avec les différents collaborateurs

COMMENT CRÉER LES DOUBLURES

Relation entre SUT et collaborateurs

La relation entre une entité qu'on cherche à tester est une relation de **dépendance**.

Pour spécifier un type de collaborateur spécifique à chaque contexte d'exécution, nous allons donc recourir à un mécanisme appelé « Injection de dépendances ».

C'est une notion qui peut s'avérer extrêmement compliquée, mais ce terme recouvre un principe plutôt simple.

La notion de dépendance

Une classe A dépend d'une autre classe B quand:

- A est de type B (dépendance par héritage) ;
- A possède un attribut de type B (dépendance par composition) ;
- Une méthode de A appelle une méthode de B.
- A dépend d'un autre objet de type C qui dépend d'un objet de type B (dépendance par transitivité) ;

La notion de dépendance

“… Because the problem with object-oriented languages is they’ve got all this implicit environment that they carry around with them. **You wanted a banana but what you got was a gorilla holding the banana** and the entire jungle. “

—Joe Armstrong,

creator of Erlang progamming language

La notion de dépendance

Dans un programme bien structuré (SOLID) on cherchera à avoir des éléments atomiques. Que ce soit pour les fonctions et méthodes, ou les classes.

On se retrouve donc potentiellement avec beaucoup de classes (SRP).

Problème: Comment pouvons-nous lier ces classes les unes aux autres de façon la plus modulaire possible?

L’injection de dépendances

DÉFINITION DU CONCEPT ET EXEMPLES

Injection de dépendance

Définition :

Le terme est difficile à définir et les définitions qu'on trouve dans l'état de l'art de la littérature sont trop abstraites pour une introduction au concept.

Il est plus facile de l'expliquer en parlant du problème auquel on répond : le couplage fort entre les composants.

Injection de dépendances

```
public interface ICommande
{
    7 références | 4/6 ayant réussi
    bool EstValidee { get; }
    7 références | 4/6 ayant réussi
    void Passer(IStockProduits warehouse);
    1 référence
    void StockerEnBDD();
}
```

Imaginons que la classe Commande ait une interface qui implémente les méthodes suivantes.
On a un couplage très fort entre le métier et implémentation de la sauvegarde.

Comment dissocier le comportement de sauvegarde qui appartient à la logique métier de celui qui appartient aux SGBD?

Exemple de couplage

Injection de dépendances

```
public class Commande : ICommande
{
    private readonly ICommandeRepository Repository;
    1 référence
    public void StockerEnBDD()
    {
        Repository.Sauver(this);
    }
```

Il faut ici utiliser une abstraction supplémentaire qui représente la collaboration entre la Commande et les mécanismes de sauvegarde. (SRP)

Ceci est la première forme d'injection de dépendances : l'injection par constructeur. Comme son nom l'indique, on injecte la dépendance à l'initialisation de l'objet.

Injection de dépendances

```
public class Commande : ICommande
{
    0 références
    public void StockerEnBDD(ICommandeRepository Repository)
    {
        Repository.Sauver(this);
    }
```

On peut aussi choisir de faire varier la dépendance à chaque appel de la méthode.

Ceci est une autre forme d'injection de dépendances : l'injection par méthode. Comme son nom l'indique, on injecte la dépendance à l'appel de la méthode de l'objet métier.

Injection de dépendances

```
public interface IInjectableRepository
{
    0 références
    void SetRepository(IRepository repository);
}
public class Commande : ICommande, IInjectableRepository
{
    private ICommandeRepository Repository;
    1 référence
    public void SetRepository(IRepository repository)
    {
        this.Repository = (ICommandeRepository)repository;
    }
```

Enfin, l'injection peut également se faire par Interfaces.

Cette technique permet de fournir la dépendance à un moment qui n'est pas forcément l'initialisation de l'objet.

Cette forme d'injection de dépendances est très utilisée par les Frameworks qui utilisent des conteneurs d'objets qui gèrent de façon automatique l'injection de dépendance.

Créer un contexte d’exécution avec les différents collaborateurs

EXEMPLES D’UTILISATION DES DOUBLURES

Exemples d’utilisation des doublures

```
public interface ICommande
{
    7 références | 4/6 ayant réussi
    bool EstValidee { get; }
    7 références | 4/6 ayant réussi
    void Passer(IStockProduits warehouse);
    1 référence
    void StockerEnBDD();
}
```

Imaginons que la classe Commande ait une interface qui implémente les méthodes suivantes.

Nous allons voir comment dissocier le comportement de sauvegarde qui appartient à la logique métier de celui qui appartient aux couches inférieures en fonction du comportement qu’on souhaite valider.

Le cas de la sauvegarde

```
public interface ICommande
{
    7 références | 4/6 ayant réussi
    bool EstValidee { get; }
    7 références | 4/6 ayant réussi
    void Passer(IStockProduits warehouse);
    1 référence
    void StockerEnBDD();
}
```

Nous allons ici chercher à valider le comportement de la sauvegarde.

Il nous faut donc un moyen de vérifier que le collaborateur en charge de la sauvegarde a bien reçu les demandes de traitement et les informations de la commande.

Test de la sauvegarde

```
public class Commande : ICommande
{
    private readonly ICommandeRepository Repository;
    1 référence
    public void StockerEnBDD()
    {
        Repository.Sauver(this);
    }
    1 référence
    public Commande(string nomArticle, int quantite, ICommandeRepository repository)
    {
        Identifiant = new Guid();
        OrderLineInformations = new KeyValuePair<string, int>(nomArticle, quantite);
        Repository = repository;
    }
```

En passant au constructeur une instance d'une classe implémentant ICommandeRepository, on peut se soustraire aux contraintes de connexion à un système externe comme un SGBD.

On voit ici que le comportement qu'on cherche à valider est exécuté par l'objet qui implémentera l'interface ICommandeRepository.

Test de la sauvegarde

```
[Fact]
0 références
public void TestOrderIsFilledIfEnoughInWarehouse()
{
    Commande commande = new Commande(BIERE, 25);
    commande.Passer(stockBDE);
    Assert.True(commande.EstValidee);
    Assert.Equal(0, stockBDE.GetInventory(BIERE));
}
```

Problème : on a cassé les tests qui passaient jusqu'ici.

On va donc devoir passer une doublure de test à notre SUT.

On va donc devoir passer une doublure de test à notre SUT. Ici, on ne cherche pas à vérifier que la sauvegarde est effectuée, on peut donc choisir une doublure qui n'implémente pas ce comportement.

Utilisation d’un Dummy

```
[Fact]
0 références
public void TestOrderIsFilledIfEnoughInWarehouse()
{
    Commande commande = new Commande(BIERE, 25, new DummyRepository());
    commande.Passer(stockBDE);
    Assert.True(commande.EstValidee);
    Assert.Equal(0, stockBDE.GetInventory(BIERE));
}
internal class DummyRepository : ICommandeRepository
{
    4 références
    public void Sauver(Commande commande){}
```

La sauvegarde étant exclue du comportement qu’on cherche à tester ici, on utilise un Dummy pour isoler le comportement qui nous intéresse.

Test de la sauvegarde avec un Fake

```
[Fact]
⚠ | 0 références
public void CommandePasseeDoitEtreSauvegardee()
{
    Commande commande = new Commande(BIERE, 25);
    commande.Passer(stockBDE);
    Assert.True(commande.EstValidee);
    // Assert.True( La commande a bien été sauvegardée)
}
```

Une fois les autres tests « réparés », on cherche donc à vérifier que quand la commande est validée, elle est bien sauvegardée.

Nous allons donc commencer par modifier le constructeur qui est appelé ici, pour lui passer un objet implémentant l'interface ICommandeRepository.

Technique : Parametrize Constructor

En ajoutant un paramètre au constructeur, on casse la signature publique de la classe, et il nous faudrait modifier tous les appels pour éviter de casser notre code. Parfois c'est impossible car les clients de notre classe ne sont pas tous dans notre code.

On va donc créer un constructeur supplémentaire, qui sera appelé par le constructeur qui a la signature originale.

Technique : Parametrize Constructor

```
5 références | 1/2 ayant réussi
public Commande(string nomArticle, int quantite) : this(nomArticle, quantite, new CommandeRepository())
{
}
5 références | 4/4 ayant réussi
public Commande(string nomArticle, int quantite, ICommandeRepository repository)
{
    Identifiant = new Guid();
    OrderLineInformations = new KeyValuePair<string, int>(nomArticle, quantite);
    Repository = repository;
}
```

On a maintenant notre constructeur qui prend un 3^{e} paramètre, mais on ne casse pas les appels des clients vers le constructeur à 2 paramètres.

Test de la sauvegarde avec un Fake

```
[Fact]
0 références
public void CommandePasseeDoitEtreSauvegardee()
{
    ICommandeRepository repository = new InMemoryCommandeRepository();
    Commande commande = new Commande(BIERE, 25, repository);
    commande.Passer(stockBDE);
    Assert.True(commande.EstValidee);
    Assert.True(repository.GetAll().First().Equals(commande));
}
```

On a donc ici besoin d'implémenter dans le projet de tests la classe InMemoryCommandeRepository qui implémente notre interface ICommandeRepository.

Création d’un Fake pour la sauvegarde

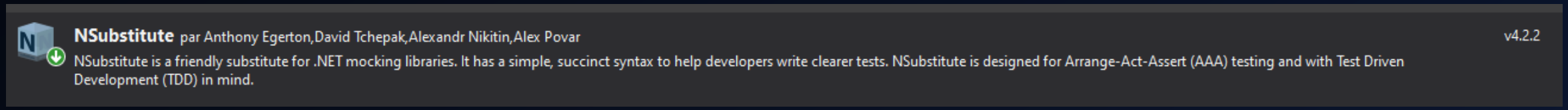
```
internal class InMemoryCommandeRepository : ICommandeRepository
{
    3 références
    internal HashSet<Commande> ListeCommandes { get; private set; } = new HashSet<Commande>();
    3 références | 1/1 ayant réussi
    public IReadOnlyCollection<Commande> GetAll()
    {
        return new ReadOnlyCollection<Commande>(ListeCommandes.ToList());
    }

    3 références
    public void Sauver(Commande commande)
    {
        ListeCommandes.Add(commande);
    }
```

L’implémentation est très simple. On a seulement besoin de coder :

- Une collection qui contiendra nos objets,
- Des méthodes pour travailler sur cette collection.

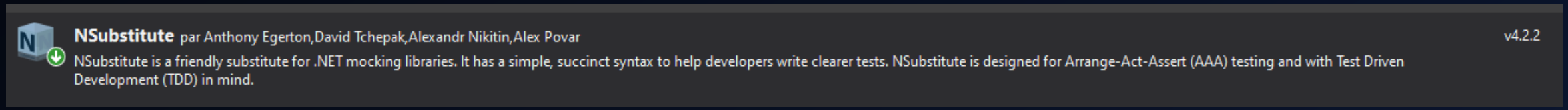
Test de la sauvegarde avec un Mock

On va ici chercher à valider le comportement de sauvegarde d'une manière différente de celle du Fake. Nous allons utiliser un objet de type Mock via le paquet nuGet NSubstitute.



Test de la sauvegarde avec un Mock

On va ici chercher à valider le comportement de sauvegarde d'une manière différente de celle du Fake. Nous allons utiliser un objet de type Mock via le paquet nuGet NSubstitute.



Test de la sauvegarde avec un Mock

Ici on ne cherche pas à vérifier un état, mais à vérifier qu'un comportement a bien été appelé.

On instancie d'abord notre objet de mock, et on va ensuite vérifier que les méthodes de l'interface qu'il mocke ont bien été appelées.

```
[Fact]
0 références
public void TestOrderIsFilledIfEnoughInWarehouse()
{
    repositoryMock = Substitute.For<ICommandeRepository>();
    Commande commande = new Commande(SOFT, 50, repositoryMock);
    commande.Passer(stockBDE);
    Assert.True(commande.EstValidee);

    repositoryMock.Received().Sauver(commande);
}
```