Stubs et Mocks Objects

Motivation

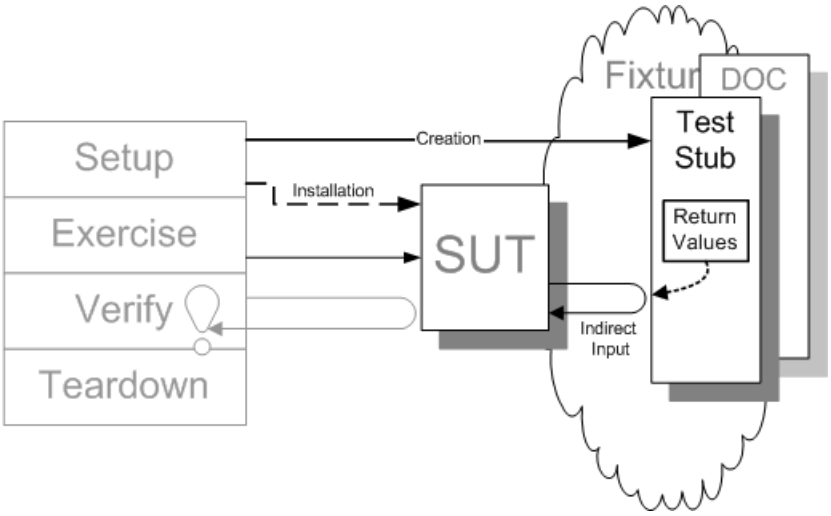
Il est rare lorsque l'on teste une classe que celle-ci ne dépende pas d'autres classes.

Il existe plusieurs raisons pour lesquelles il n'est pas possible d'effectuer le test de notre unité directement avec les « vraies » classes dont elle dépend:

- Une ou plusieurs de ces classes ne sont pas encore développées
- Ces classes contrôlent un équipement physique coûteux, fragile ou unique
- L'exécution de ces classes est très lente
- Ces classes ne sont pas encore testées et on ne peut donc pas leur faire confiance.

De plus on veut dans certains cas faire de « vrais » tests unitaires. C'est à dire tester notre unité en dehors de toute dépendance. En effet si l'on découvre un bug, comment trouver son origine si l'on dépend d'une multitude de classes potentiellement erronées?

Toutes ces raisons nous obligent à travailler avec de « faux » objets remplaçants les classes dont on dépend. Nous obtenons en gros le schéma ci-dessous (SUT = System Under Test):



Stubs et Mocks

Il existe différents types de faux objets, nous n'en n'examinerons que deux ici:

- les stubs
- les mocks objects

On utilise aussi en fonction des besoins des fake objects, ou des spies. Si vous voulez en savoir plus, reportez-vous à la référence en fin de chapitre.

Stubs

Les stubs sont des objets sans aucune intelligence renvoyant systématiquement la même réponse à l'appelant et n'effectuant aucun traitement. Le stub est utile lorsque pour effectuer un test on a besoin des réponses données par ces objets mais que le test ne porte pas sur notre interaction avec ces objets.

Ex: on doit traiter des objets provenant d'une base de données. Le test ne porte pas sur notre interface avec la base de données mais sur le traitement des objets. Les stubs nous renverront des objets à traiter.

Mocks Objects

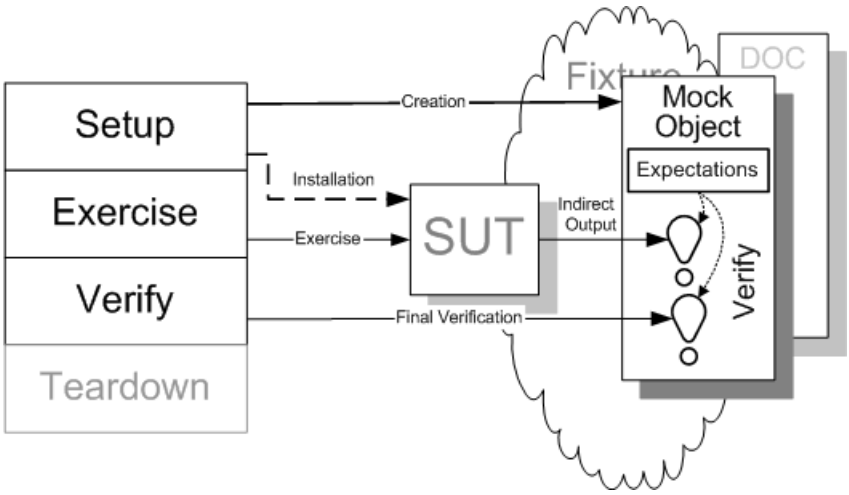
Dans certains cas le test consiste à vérifier si l'on a bien appelé les bonnes méthodes avec les bonnes données.

Ex: on veut persister des objets dans une base de données. Il faut vérifier que l'on envoie bien les bonnes données au DAO.

Dans ce cas le mock object vérifiera que l'on appelle bien les bonnes méthodes dans le bon ordre avec les bonnes données. Si l'on n'appelle pas la bonne méthode au bon moment le mock object déclarera le test échoué.

A la fin il faut encore vérifier que l'on a bien appelé toutes les méthodes nécessaires (on pourrait ne rien avoir fait). C'est pourquoi l'on a une méthode de vérification finale qui déclarera le test échoué si toutes les méthodes attendues n'ont pas été appelées.

La figure suivante illustre l'utilisation des mocks objects:



Il est à noter que contrairement au cas des stubs la vérification ne se fait pas dans la méthode de test mais bien dans le mock objet!

Harcoded vs Configurables Mock Objects

Les appels et paramètres à vérifier peuvent être soit « hardcodés » c'est à dire directement codés dans l'objet sans possibilité de les changer; soit paramétrables via le constructeur ou des setters.

Les objets hardcodés sont évidemment plus simples, avec l'inconvénient qu'il faut écrire une version de l'objet par cas de test.

Référence:

Xunit Test Patterns G. Meszaros

Aussi consultable on-line: http://xunitpatterns.com