

# Les bouchons de test sous Flex

par Olivier Bugalotto ([Mes articles](#))

Date de publication : 02/09/2007

Dernière mise à jour :



I - Introduction

II - Mise en place du framework de tests unitaires FlexUnit et du composant TestRunner

III - La technique du Self-hunt

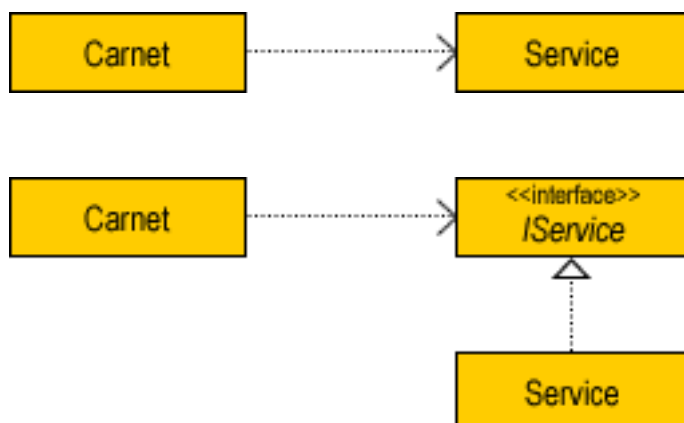
IV - La technique du Mock object

V - Conclusion

## I - Introduction

L'écriture de tests unitaires consiste à écrire un test et d'écrire le code qui passera ce test. C'est aussi notre capacité à écrire un code testable en isolant une classe ou un ensemble de classes. De plus nous savons qu'il est assez rare d'avoir des classes totalement indépendantes du reste du système. C'est pour cela que nous sommes amenés à mettre en place un certain contexte lorsque nous voulons tester certaines classes mais plus le nombre de classes augmentent plus le code devient complexe ce qu'il tente à montrer qu'il faut rompre certaines dépendances.

Une des premières étapes est la rupture de la dépendance à l'aide d'une interface :



Cela devient :

Cette interface nous permet de mettre en place un substitut, c'est ce que nous appelons un *bouchon*. Nous allons voir à travers la diffusion d'évènements d'une classe, la mise en place de technique comme le *Self-hunt* et le *Mock object*.

## II - Mise en place du framework de tests unitaires FlexUnit et du composant TestRunner

Image 1 : Ajout de la librairie flexunit.swc (**FlexUnit**)

Ecriture du code de base, pour utiliser le composant TestRunner :

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
xmlns:it="flexunit.flexui.*" creationComplete="doInit()">
  <mx:Script>
    <![CDATA[
      import tests.TestCarnet;
      import flexunit.framework.TestSuite;
      import flexunit.framework.Test;

      private function doInit():void {
        testRunner.test = suite();
        testRunner.startTest();
      }

      private function suite():TestSuite {
        var s:TestSuite = new TestSuite();
        s.addTestSuite(TestCarnet);
        return s;
      }
    ]]>
  </mx:Script>
  <it:TestRunnerBase width="100%" height="100%" id="testRunner">
  </it:TestRunnerBase>
</mx:Application>
```

Voyons maintenant ces techniques.

### III - La technique du Self-hunt

C'est une technique plutôt simple au la classe de test joue le rôle de bouchon :

```
// L'interface ICarnetDao
package classes
{
    public interface ICarnetDao
    {
        function get count():uint;
        function addContact(contact:*) :void;
    }
}
```

```
// La classe Carnet
package classes
{
    import flash.events.EventDispatcher;
    import classes.events.CarnetEvent;

    public class Carnet extends EventDispatcher
    {
        private var dao:ICarnetDao;

        public function Carnet(dao:ICarnetDao) {
            this.dao = dao;
        }

        public function addContact(contact:*) :void {
            dao.addContact(contact);
        }
    }
}
```

```
// La classe de test
package tests
{
    import flexunit.framework.TestCase;
    import classes.Carnet;
    import classes.events.CarnetEvent;
    import classes.ICarnetDao;

    public class TestCarnet extends TestCase implements ICarnetDao
    {
        private var carnet:Carnet;

        private var numberOfContact:uint;

        public function get count():uint {
            return numberOfContact;
        }

        override public function setUp():void {
            carnet = new Carnet(this);
        }

        public function addContact(contact:*) :void {
            numberOfContact++;
        }

        public function testAddTwoListeners():void {
```

```
    carnet.addContact({});  
    assertEquals(1, count);  
  }  
}
```

C'est aussi une technique qui montre vite ses limites, lorsque la classe à tester :

- demande plusieurs instances du bouchon
- détruit sa dépendance, ici le bouchon (càd la classe de test)
- et d'autres classes au besoin du même bouchon de test.

Pour résoudre ces limites, l'utilisation de la technique du "Mock object" est plus adaptée.

## IV - La technique du Mock object

Le Mock object consiste à écrire une classe pour jouer le rôle du bouchon :

```
package tests
{
    import classes.ICarnetDao;

    public class MockCarnetDao implements ICarnetDao
    {
        private var numberOfContact:uint;

        public function get count():uint {
            return numberOfContact;
        }

        public function addContact(contact:*) :void
        {
            numberOfContact++;
        }
    }
}
```

```
package tests
{
    import flexunit.framework.TestCase;
    import classes.Carnet;
    import classes.events.CarnetEvent;
    import classes.ICarnetDao;

    public class TestCarnet extends TestCase
    {
        private var carnet:Carnet;
        private var mock:ICarnetDao;

        override public function setUp():void {
            mock = new MockCarnetDao();
            carnet = new Carnet(mock);
        }

        public function testAddTwoListeners():void {
            carnet.addContact({});

            assertEquals(1, mock.count);
        }
    }
}
```

## V - Conclusion

Ces deux techniques permettent de tester plus facilement une classe mais entraînent aussi un surplus de travail. Ce travail en plus est un bénéfice plus grand lorsque l'application devient plus complexe et difficile à mettre à jour.

A vous maintenant d'essayer et de faire votre propre opinion en application ces techniques à vos tests unitaires.

