EasyMock 3.6

Source : easymock.org/user-guide.html

Installation d'EasyMock dans un « Environnement de Développement » (IDE)

Librairies à ajouter, si nécessaire, pour la compilation du code et des tests : EasyMock, et Objeneis.
1 librairie à ajouter, si nécessaire, à la compilation des tests : Hamcrest en sus de Junit.

EasyMock est un framework open source qui permet de créer et d'utiliser des doublures.
Les doublures créées par EasyMock peuvent avoir plusieurs utilités allant du simple fantôme qui renvoie une valeur à l'espion qui permet de vérifier le comportement des invocations de méthodes selon les paramètres fournis.
La classe principale d'EasyMock est la classe EasyMock : toutes ses méthodes sont statiques.

Usage d'une doublure se fait en 3 étapes

- création de la doublure d'une interface ou d'une classe
- définition de son comportement (voir ci-dessous)
- placer le mock en situation de reproduire le comportement défini et d'enregistrer les invocations : **`EasyMock.replay(« nom de la doublure »).`**

L'objet **`PortefolioLine`** crée via l'instruction est la « doublure » d'un objet de la classe **`PortefolioLine`** :

```
PortefolioLine line = EasyMock.creatMock(PortefolioLine.class);
```

Définition du Comportement

Définition d'un comportement d'une doublure

Pour qu'une doublure autorise un appel à une méthode il faut utiliser la méthode statique **`EasyMock.expect()`** l'argument de cette méthode doit être **`doublure.methode(arguments autorisées).`**

Pour préciser le comportement d'une méthode d'une doublure qui renvoie une valeur, il faut :

- utiliser la méthode **`andReturn()`** de l'objet retourné par **`EasyMock.expect()`** pour préciser la valeur de retour.

Exemple : l'instruction suivante définit que l'appel à la méthode **`getValue()`** de la doublure **`line`** va retourner la valeur **`50.00`**.

```
EasyMock.expect(line.getValue()).andReturn(50.00)
```

Lever d'exception par une doublure

EasyMock simule la levée d'une exception. Dans la définition du comportement d'une méthode, il faut utiliser la méthode **`andThrow()`** avec pour paramètre l'instance de l'exception à lever. La simulation d'une levée d'exception est particulièrement utile pour tester des cas d'erreurs difficiles à automatiser comme une coupure réseau par exemple.

Exemple : l'instruction suivante définit que l'appel à la méthode **`divide(10.0, 0.0)`** de la doublure **`calcSer`** lève l'exception **`IllegalArgumentException.`**

```
EasyMock.expect(calcSer.divide(10.0,0.0)).
    andThrow(new IllegalArgumentException())
```

Durant l'exécution de **`mathApp.divide(10.0, 0.0)`** la méthode **`divide(10.0,0.0)`** de la doublure **`calcSer`** est appelée. Le test suivant passe car l'appel de **`mathApp.divide(10.0, 0.0)`** lève l'exception **`IllegalArgumentException.`**

```
@Test(expected=IllegalArgumentException.class)
public void testException() {
     CalService calcSer = EasyMock.createMock (CalService.class);
     Calculatrice mathApp = new Calculatrice (calcSer);

     EasyMock.expect(calcSer.divide(10.0,0.0)).
                          andThrow(new IllegalArgumentException());

     EasyMock.replay (calcSer);
     mathApp.divide (10.0, 0.0);    }
```

Simulation d'une méthode retournant void

Inutile de préciser le comportement d'une méthode qui ne retourne aucune valeur (void), il faut simplement invoquer la méthode **`doublure.methode(arguments)`**. EasyMock va simplement enregistrer l'appel.

`EasyMock.expectLastCall()` permet de définir que le comportement du dernier appel d'une méthode sur une doublure. Par exemple, le code ci-dessous spécifie que l'appel de **`meth`** de l'objet **`doub`** va générer la levée d'une exception du type **`IllegalArgumentException.`**

```
doub.meth();
EasyMock.expectLastCall().andThrow(new IllegalArgumentException());
```

Vérification des invocations d'une doublure

L'appel à la méthode **`EasyMock.verify(mock)`** fait échoué le test si la séquence des comportements définies sur **`mock`** n'est pas correctement exécutée ; correctement voulant dire : aucun oubli d'appel, aucune duplication d'appel, et pas d'ajout d'appel.

Pour que l'ordre des appels soit vérifié la doublure doit être « stricte » c'est à dire créée via l'instruction :

```
CalculatriceSer calcSer = EasyMock.strictMock(CalculatriceSer.class);
```

Exemple : le test suivant **échoue** car une seule addition est effectuée sur les 2 définies. Durant, l'exécution de **`mathAppl.add(x, y)`** seule la méthode **`add(x, y)`** de la doublure est appelée.

```
@Test
public void testAddComportement () {
   CalService calcSer = EasyMock.createMock (CalService.class);
   Calculatrice mathApp = new Calculatrice (calcSer);

    EasyMock.expect (calcSer.add(10.0,20.0)).andReturn(30.0);
    EasyMock.expect (calcSer.add(20.0,100.0)).andReturn(120.0);
    EasyMock.replay (calcSer);

    Assert.assertEquals (mathAppl.add(10.0, 20.0), 30.0,0) ;
    EasyMock.verify (calcSer);  }
```

Exemple : le test suivant passe car il n'y a pas de vérification des appels aux méthodes de la doublure.

```
@Test
public void testAddComportement (){
   CalService calcSer = EasyMock.createMock (CalService.class);
   Calculatrice mathApp = new Calculatrice (calcSer);

    EasyMock.expect (calcService.add(10.0,20.0)).andReturn(30.0);
    EasyMock.expect (calcService.add(20.0,100.0)).andReturn(120.0);
    EasyMock.replay (calcService);
    Assert.assertEquals (mathApplication.add(10.0, 20.0), 30.0,0)   }
```

Exemple : le teste suivant passe uniquement car la doublure n'est pas "stricte".

```
@Test
public void testAddComportementBis (){
   CalService calcSer = EasyMock.createMock (CalService.class);
   Calculatrice mathApp = new Calculatrice (calcSer);

      EasyMock.expect (calcService.add(10.0, 20.0)).andReturn(30.00);
      EasyMock.expect (calcService.add(20.0, 30.0)).andReturn(50.00);
      EasyMock.replay (calcService);
      Assert.assertEquals (mathApplication.add(20.0, 30.0), 50.0, 0);
      Assert.assertEquals (mathApplication.add(10.0, 20.0), 30.0, 0);
      EasyMock.verify (calcService); }
```

Répétition d'un même comportement

La méthode **`time(count)`** permet de définir la répétition d'un comportement. L'instruction

```
EasyMock.expect(calcSer.add(1.0,2.0)).andReturn(3.0).times(3)
```

est équivalente au bloc des 3 lignes suivantes

```
EasyMock.expect(calcSer.add(1.0,2.0)).andReturn(3.0);
EasyMock.expect(calcSer.add(1.0,2.0)).andReturn(3.0);
EasyMock.expect(calcSer.add(1.0,2.0)).andReturn(3.0);
```

`anyTimes()` autorise un nombre d'appels quelconque.

`Time (min, max)` exige un nombre d'appels compris entre **`min`** et **`max`** inclus.

`atLeastOnce()` exige au moins un appel et autorise plusieurs appels.

Changer le comportement d'une doublure

Il est possible de spécifier plusieurs comportements pour les appels successifs à une méthode en enchainant les appels aux méthodes **`times()`**, **`andReturn()`**, and **`andThrow()`**.

Exemple : le premier appel de la méthode **`decouvrir`** retourne **`true`** les autres appels retournent **`false`**.

```
EasyMock.expect(mockCase.decouvrir()).andReturn(true).times(1).
                        andReturn(false).anyTimes()
```

Doublure partielle

Parfois, il faut "doubler" certaines méthodes d'une classe mais pas d'autres ces dernières méthodes devant garder leur comportement normal. Normalement, ce besoin existe lorsque l'on veut tester une méthode appelant d'autres méthodes de la même classe. La solution est :

```
ToMock mock = partialMockBuilder(ToMock.class)
 .addMockedMethod("mockedMethod").createMock()
```

La méthode (ou les méthodes) **`addMockedMethod(s)`** seront doublés (**`mockedMethod()`** dans l'exemple). Les autres méthodes ne sont pas doublées.

La méthode **`partialMockBuilder`** retourne un objet implémentant l'interface **`IMockBuilder<ToMock>`**

Exemple,

```
IMockBuilder<CalculatriceService> calcSerBuilder =
                    EasyMock.partialMockBuilder(CalculatriceService.class);
CalculatriceService calcSer =
                        calcSerBuilder.addMockedMethod("add").createMock();
EasyMock.expect(calcSer.add(10.0, 20.0)).andReturn(30.00);
EasyMock.expect(calcSer.add(30.0, 30.0)).andReturn(60.00);
```

Class Mocking Limitations

- To be coherent with interface mocking, EasyMock provides a built-in behavior for equals(), toString(), hashCode() and finalize() even for class mocking. It means that you cannot record your own behavior for these methods. This limitation is considered to be a feature that prevents you from having to care about these methods.
- EasyMock provides a default behavior for Object's methods (*equals*, *hashCode*, *toString*, *finalize*). However, for a partial mock, if these methods are not mocked explicitly, they will have their normal behavior instead of EasyMock default's one.
- Final methods cannot be mocked. If called, their normal code will be executed.
- Private methods cannot be mocked. If called, their normal code will be executed. During partial mocking, if your method under test is calling some private methods, you will need to test them as well since you cannot mock them.