Outils de gestion de projets

Tests unitaires avec JUnit - Travaux pratiques

Pendant le TP, faites attention à l'alternance codage/test. L'idée est ici d'écrire peu de lignes de code, et puis de tester que cela fonctionne, ou mieux, d'écrire un test qui ne fonctionne pas, et ensuite écrire le code qui le fera fonctionner.

Le programme à écrire doit résoudre le problème de la représentation arithmétique avec plusieurs devises. L'addition arithmétique entre deux objets de même devise est triviale, il suffit d'ajouter les deux montants. De simple nombres suffisent ; les devises peuvent être ignorées. Les choses deviennent intéressantes une fois plusieurs devises impliquées. On ne peut pas simplement convertir une devise en une autre pour effectuer des opérations arithmétiques puisqu'il n'y a pas un unique taux de change.

Classe `Money`

Commençons simplement et définissons une classe `Money` pour représenter une valeur pour une devise donnée. Cette classe est capable d'ajouter deux valeurs ayant la même devise (méthode `add`).

```
package fr.emse.test;

class Money {
	private int fAmount;
	private String fCurrency;

	public Money(int amount, String currency) {
		fAmount = amount;
		fCurrency = currency;
	}

	public int amount() {
		return fAmount;
	}

	public String currency() {
		return fCurrency;
	}

	public Money add(Money m) {
		return new Money(amount() + m.amount(), currency());
	}
}
```

A partir de maintenant, nous allons suivre un processus « coder ; tester ; coder ; tester », en utilisant JUnit.

1 – Créer la classe `Money` dans un projet Eclipse. Pour tester notre classe `Money`, nous allons créer une classe de test `MoneyTest` sous Eclipse : **File > New > JUnit Test Case**.

Penser à spécifier la classe à tester, `Money` (figure 1), choisir **« New JUnit 4 test »** et cliquer sur « Finish ».

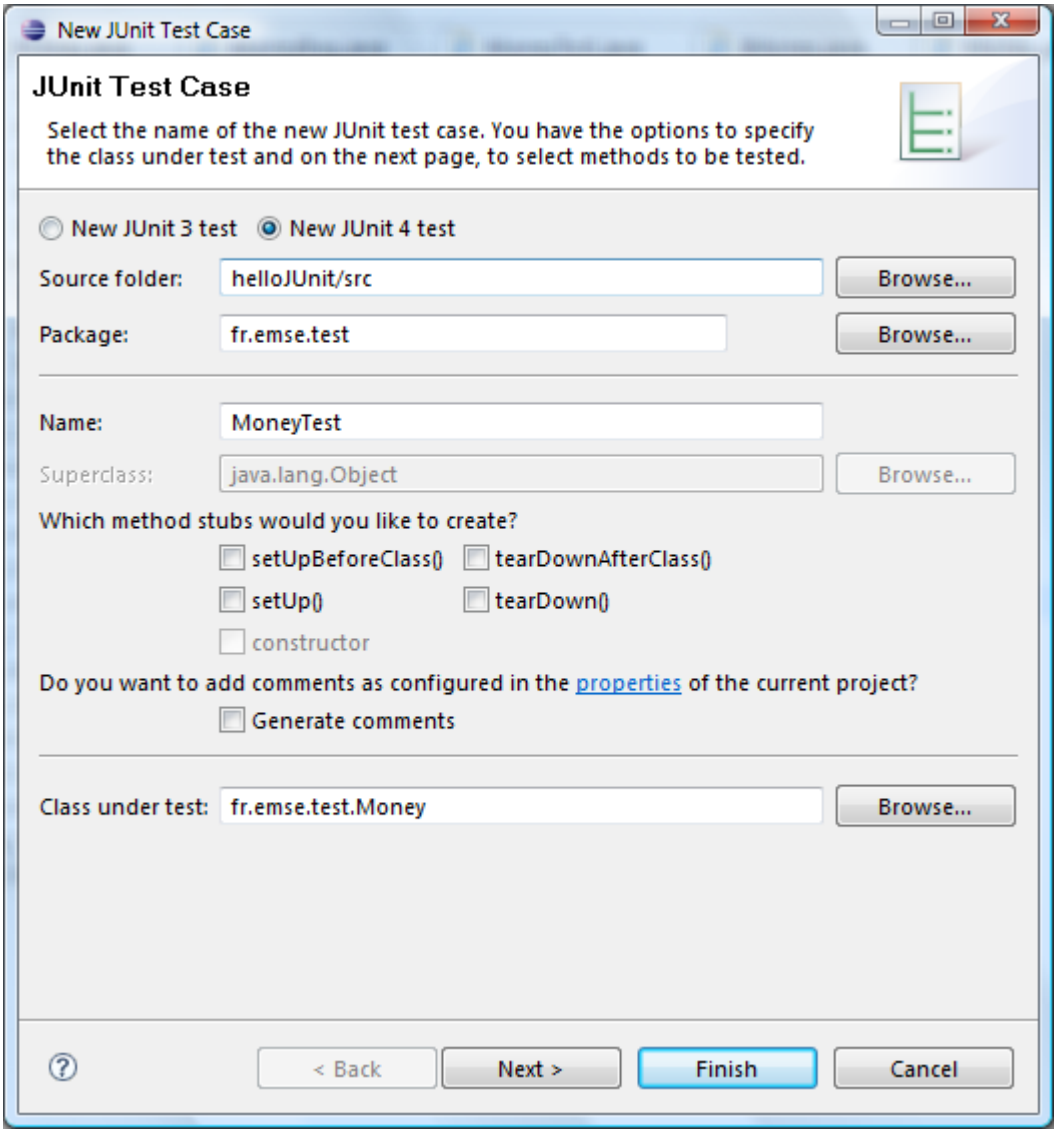

Figure 1

2 – Ajouter une méthode de test à la classe `MoneyTest` :

```
package fr.emse.test;

import static org.junit.Assert.*;

import org.junit.Test;

public class MoneyTest {

	@Test
	public void testSimpleAdd() {
		Money m12CHF = new Money(12, "CHF"); // création de données
		Money m14CHF = new Money(14, "CHF");
		Money expected = new Money(26, "CHF");
		Money result = m12CHF.add(m14CHF); // exécution de la méthode testée
		assertTrue(expected.equals(result)); // comparaison

	}

}
```

3 – Tester la méthode `add` avec le cas de test précédemment défini : « **Run > Run As > JUnit test** ». Que

se passe-t-il ? Comment l'expliquez-vous ?

4 – Ajouter une méthode de test d'égalité dans la classe de test `MoneyTest` :

```
    @Test
    public void testEquals() {
        Money m12CHF= new Money(12, "CHF");
        Money m14CHF= new Money(14, "CHF");

        assertTrue(!m12CHF.equals(null));
        assertEquals(m12CHF, m12CHF);
        assertEquals(m12CHF, new Money(12, "CHF"));
        assertTrue(!m12CHF.equals(m14CHF));
    }
```

5 – Relancer la classe de test `MoneyTest`. Que se passe-t-il ?

6 – Surcharger la méthode `equals` de la classe `Money` pour passer ce test avec succès.

7 – Nous pouvons remarquer la duplication de code dans les méthodes `testSimpleAdd` et `testEquals`. Remédier à ce problème en utilisant l'annotation `@Before`. Relancer le test pour vérifier que vos modifications n'ont pas altéré le résultat.

Classe `MoneyBag`

Maintenant que la classe `Money` semble fonctionner pour une unique devise, nous allons prendre en charge des devises multiples. Pour cela, introduisons la classe `MoneyBag` permettant d'agréger des valeurs de différentes devises.

```
package fr.emse.test;

import java.util.Vector;

class MoneyBag {
    private Vector<Money> fMonies = new Vector<Money>();

    MoneyBag(Money m1, Money m2) {
        appendMoney(m1);
        appendMoney(m2);
    }

    MoneyBag(Money bag[]) {
        for (int i = 0; i < bag.length; i++)
            appendMoney(bag[i]);
    }

    private void appendMoney(Money m) {
        if (fMonies.isEmpty()) {
            fMonies.add(m);
        } else {
            int i = 0;
            while ((i < fMonies.size())
                    && (!(fMonies.get(i).currency().equals(m.currency()))))
                i++;
            if (i >= fMonies.size()) {
                fMonies.add(m);
            } else {
                fMonies.set(i, new Money(fMonies.get(i).amount() +
                        m.amount(),
```

```
                                    m.currency()));

                }
            }
        }
}
```

8 – Écrire la méthode `equals` de la classe `MoneyBag`, pour éviter les mêmes erreurs que `Money` dans la question 5.

9 – Créer une classe de test `MoneyBagTest` avec les méthodes suivantes :

```
    @Before
    public void setUp() {
        f12CHF= new Money(12, "CHF");
        f14CHF= new Money(14, "CHF");
        f7USD=  new Money( 7, "USD");
        f21USD= new Money(21, "USD");
        fMB1= new MoneyBag(f12CHF, f7USD);
        fMB2= new MoneyBag(f14CHF, f21USD);
    }

    @Test
    public void testBagEquals() {
        assertTrue(!fMB1.equals(null));
        assertEquals(fMB1, fMB1);
        assertTrue(!fMB1.equals(f12CHF));
        assertTrue(!f12CHF.equals(fMB1));
        assertTrue(!fMB1.equals(fMB2));
    }
```

10 – Tester la classe `MoneyBag`, et la modifier si nécessaire.

11 – Afin de tester `Money` et `MoneyBag`, créer une nouvelle de suite de test `AllTests`:

```
package fr.emse.test;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses(value = { MoneyTest.class, MoneyBagTest.class })
public class AllTests {

}
```

Il suffit maintenant de lancer `AllTests` en tant que « JUnit test » sous Eclipse pour effectuer tous les tests les uns à la suite des autres.

Intégration des classes `Money` et `MoneyBag`

Maintenant que `MoneyBag` est créée, nous pouvons corriger la méthode `add` de la classe `Money` :

```
    public Money add(Money m) {
        if (m.currency().equals(currency()))
            return new Money(amount() + m.amount(), currency());
        return new MoneyBag(this, m);
    }
```

Cependant cette méthode ne va pas compiler à cause d'un problème de typage. Avec l'introduction de `MoneyBag`, nous avons 2 types pour représenter les monnaies. Afin de rendre cette distinction invisible au code client, introduisons une **interface `IMoney` que doivent implémenter `Money` et `MoneyBag`** :

```
package fr.emse.test;

interface IMoney {
      public IMoney add(IMoney aMoney);
}
```

Afin de vraiment cacher les 2 types aux utilisateurs, il convient de prendre en charge toutes les combinaisons arithmétiques possibles entre `Money` et `MoneyBag`. Mais avant de coder, définissons un nouveau jeu de test :

```
      @Test
      public void testMixedSimpleAdd() {
            // [12 CHF] + [7 USD] == {[12 CHF][7 USD]}
            Money bag[] = { f12CHF, f7USD };
            MoneyBag expected = new MoneyBag(bag);
            assertEquals(expected, f12CHF.add(f7USD));
      }
```

12 – Écrire les méthodes de tests suivant le même schéma que la méthode `testMixedSimpleAdd` :

- `testBagSimpleAdd` : pour ajouter un `MoneyBag` à un simple `Money`
- `testSimpleBagAdd` : pour ajouter un simple `Money` à un `MoneyBag`
- `testBagBagAdd` : pour ajouter deux `MoneyBags`

Les cas de test étant définis, nous pouvons commencer à implémenter les différentes combinaisons de `Money` et de `MoneyBag`. Une solution est l'utilisation d'un appel supplémentaire pour découvrir le type d'argument à gérer. Nous appelons une méthode sur l'argument avec le nom de la méthode originale suivi du nom de la classe du récepteur. Les méthodes `add` de `Money` et `MoneyBag` deviennent :

```
package fr.emse.test;

class Money implements IMoney {
      //...
      public IMoney add(IMoney m) {
            return m.addMoney(this);
      }
      //...
}
```

```
package fr.emse.test;

import java.util.Vector;

class MoneyBag implements IMoney {
      //...
      public IMoney add(IMoney m) {
        return m.addMoneyBag(this);
       }
      //...
}
```

13 – Modifier `IMoney`, `Money` et `MoneyBag` afin que cela compile et que les tests unitaires soient passés.

Simplification

Maintenant que les classes `Money` et `MoneyBag` passent les test précédemment définis, nous pouvons noter l'anomalie suivante : que se passe-t-il lorsque que le résultat de l'addition est un `MoneyBag` avec une seule valeur ? Par exemple, ajouter -12 CHF à un `MoneyBag` contenant 7 USD et 12 CHF résulte en un `MoneyBag` avec uniquement 7 USD, alors qu'il serait souhaitable qu'il résulte un `Money` simple de valeur 7 USD.

14 – Définir un jeu de test pour vérifier la simplification des `MoneyBag` en `Money` lorsque cela est nécessaire.

15 – Modifier les classes en conséquence, jusqu'à ce que les tests soient vérifiés.

Références

Ces exemples sont issus du site officiel de JUnit : http://junit.sourceforge.net/