TD/TP 4

TESTS UNITAIRES

UTILISATION DE JUNIT 4

UNIVERSITÉ CAEN NORMANDIE

Dernière minute : Dépannage manuel si NetBeans ne fonctionne pas encore

Créer le répertoire testsJava.

On se place dans ce répertoire dans la suite.

Déposer les fichiers junit.jar et hamcrest.jar à télécharger sur ecampus

Créer les répertoires src, test, build et buildTest

Pour compiler vos sources (vers le répertoire build):

```
javac -d "build" src/demotestsunitaires/*.java
```

Pour compiler vos tests (vers le repertoire buildTest):

```
javac -d "buildTest" -cp "build:buildTest:junit.jar" test/demotestsunitaires/*.java
```

Pour lancer vos tests (par exemple ci-dessous sur les deux classes nommées PointTest et DefaultMathetmatiquesTest du package demotestsunitaires) :

```
java -cp build:buildTest:junit.jar:hamcrest.jar org.junit.runner.JUnitCore demotestsunitaires.PointTest demotestsunitaires.DefaultMathematiquesTest
```

1. Création d'un projet NetBeans et de la classe Point.

Dans un but de clarification, nous allons travailler dans un nouveau projet servant uniquement à faire nos premiers pas avec JUnit.

Créer le projet DemoTestsUnitaires.

Créer la classe Point, qui représente un point du plan avec des coordonnées x,y de type double.

Spécifications :

- les coordonnées doivent être positives ou nulles, et le constructeur doit mettre à zéro une coordonnée négative passée en paramètre.
- La méthode toString() doit être sous la forme (1.0,2.0) pour un point situé en x=1 et y=2.

Ecrire la classe et ajouter les getters (ajout de code avec NetBeans...).

Clic-droit sur Point.java, outils, create/update tests
Dans Generated-code, cocher seulement Default Method Bodies, puis valider.

La classe PointTest est créée dans les packages de test.

2. Création de tests unitaires

Dans NetBeans :

- Clic-droit sur Point.java, outils, create/update tests.
- Dans Generated-code, pour simplifier dans un premier temps, cocher seulement Default Method Bodies, puis valider.

La classe PointTest est créée dans les packages de test. Ecrire un premier test :

```
@Test
public void testToString() {
    Point p = new Point(1, 2);
    assertEquals(p.toString(), "(1.0,2.0)");
}
```

Lancer à présent le test par un clic-droit sur le projet → Test (ou shift F6). Si le test échoue, corriger toString(). Dans le cas contraire, modifier provisoirement toString() pour faire échouer le test (et voir ce qui se passe), puis rétablissez la version correcte.

Ecrire à présent un test du constructeur de Point, par exemple :

```
@Test
public void testConstructeur() {
    Point p1 = new Point(-1, 2);
```

```
    assertTrue(p1.getX() == 0);
    assertTrue(p1.getY() == 2);
    Point p2 = new Point(1, -2);
    assertTrue(p2.getX() == 1);
    assertTrue(p2.getY() == 0);
  }
```

À présent, votre procédure de test peut réussir à 0%, 50%, ou 100%, car il y a 2 tests.

3. Problème des dépendances

À présent, nous allons enrichir notre architecture en la dotant d'un module permettant de faire des calculs mathématiques, plutôt que de faire ceux-ci directement dans notre code. En particulier nous allons déléguer la vérification de la valeur positive des coordonnées.

Créer l'interface Mathematiques qui comporte deux méthodes :

```
boolean estPositif(double v) ;
double valeurAbsolue(double v) ;
```

En créer une implémentation DefaultMathematiques

Remarque : on pourrait bien sûr avoir une classe Mathematiques dotée de méthodes statiques, mais nous optons ici pour une architecture modulable à la façon du pattern Strategy, permettant de disposer de différentes implémentations. Cela va particulièrement être pertinent pour la partie 4 de ce TP.

Modifier le constructeur de Point afin qu'il délègue le test à une instance de DefaultMathematiques. Pour ce faire, on va utiliser un mécanisme d'injection de dépendance dans Point via l'attribut et la méthode suivants :

```
private Mathematiques mathematiques;
// Injection d'une dépendance que l'on peut redéfinir dans les sous-classes
protected Mathematiques getMathematiques()
{
   return new DefaultMathematiques();
}
```

Créer une erreur dans la méthode estPositif(...) en lui faisant toujours renvoyer *true*. Relancer le test. On voit que le test échoue sur le constructeur de Point, alors que ce dernier est correct. C'est une erreur située dans DefaultMathematiques, donc une dépendance, qui en est la cause. Il faut donc créer des tests pour DefaultMathematiques. Ceci montre l'importance de tester les briques unitaires afin de pouvoir circonscrire ensuite facilement les erreurs lorsqu'il y a des dépendances.

4. S'affranchir d'une dépendance : utilisation d'un Mock.

Dans certains cas, il existe une dépendance vers un service en cours de développement (par ex. par une autre équipe de développement) ou non encore accessible (accès à une base de données, etc.), ou enfin qui est encore bogué.

Cela rend difficile voire impossible le test de notre partie. On va pour cela créer un Mock, c'est-à-dire une classe simulant partiellement le comportement du service en question sur un nombre restreint d'entrées.

En l'occurrence, comme notre classe DefaultMathematiques est boguée et en supposant que nous ne puissons pas encore disposer de son implémentation correcte, nous allons créer un Mock de l'interface Mathematiques qui sait, en dur, déterminer correctement la valeur absolue de 1, 2, -1 et -2 (et uniquement celles-ci).

Créer la classe MathematiquesMock correspondante.

Créer à présent la classe PointMock qui se base sur le service MathematiquesMock :

```
class PointMock extends Point
    {
      public PointMock(double x, double y)
      {
        super(x,y);
      }
      public Mathematiques getMathematiques()
      {
        return new MathematiquesMock();
      }
    }
```

On peut placer cette classe en interne de la classe PointTest puisque l'on compte ne s'en servir que dans cette dernière.

Ajouter la factory method suivante dans PointTest :

```
public Point createPoint(double x, double y)
{
  return new PointMock(x,y); // replacer par new Point(x,y) lorsque l'on veut tester avec la vraie classe (sans le Mock).
}
```

À présent, relancer le test. Puisque le Mock fonctionne correctement sur les valeurs testées, le test réussit.

Lorsque l'implémentation DefaultMathematiques sera prête, il suffira de remplacer `return new PointMock(x,y);` par `return new Point(x,y);` dans la factory method de la classe de Test. Le code de Point restera complètement inchangé, grâce à l'injection de dépendance réalisée.

5. Complément : test unitaire d'un générateur de nombres aléatoires.

Cette partie consiste à mettre en place le test correspondant à la page 16 du document de CM.

Les valeurs n, m, ainsi que la plage de valeurs (indiquée à +/- 50% dans le CM, c'est-à-dire entre 50 et 150 dans l'exemple donné) seront paramétrables.

On testera les méthodes :

- Math.random() // valide
- 2*Math.random() // non valide car donne des valeurs entre 0 et 2
- Math.sin(Math.random()*Math.PI) // non valide car non homogène

Pour cela, on pourra créer une classe (à tester) comportant une méthode statique double random(), qui s'appuie sur Math.random() comme indiqué ci-dessus.