TP JUnit 3

En vert, une réponse

Récupérez à l'URL `http://cedric.cnam.fr/~farinone/SMB212/junit3.8.2.zip`, le fichier `junit3.8.2.zip`. Il contient le `junit.jar` utile pour ce TP, qui contient les classes pour JUnit 3.

Le but du problème est de comprendre le fonctionnement de JUnit ainsi que la construction de classes de tests avec JUnit 3. Il permet donc d'illustrer la version JUnit 3.x, une version plus ancienne que JUnit 4 mais encore utilisée. La version ci-dessous n'utilise donc pas les annotations Java ce qui rend le problème plus simple. C'est, en partie, un sujet qui a été donné en examen au CNAM en septembre 2011.

Pour fixer les idées, on donne ci dessous la classe à tester `PolynomeSecondDegre` :

```
package jmf.equation;

public class PolynomeSecondDegre {
	private double a, b, c;
	private double[] racines;

	public PolynomeSecondDegre(double a, double b, double c) {
		super();
		this.a = a;
		this.b = b;
		this.c = c;
	}

	public PolynomeSecondDegre() {
	}

	public double discriminant() {
		return b *b - 4 * a *c;
	}

	public double[] calculRacines() {
		racines = null;
		double discri = discriminant();
		if (discri == 0) {
			racines = new double[1];
			racines[0] = -b / (2 * a);
		} else if (discri > 0) {
			double racineDiscriminant = Math.sqrt(discri);
			racines = new double[2];
			racines[0] = (-b + racineDiscriminant)/ (2 * a);
			racines[1] = (-b - racineDiscriminant)/ (2 * a);
		}
		return racines;
	}
}
```

Question 1)

Lorsqu'on doit tester une classe avec JUnit, faut il ajouter quelque chose à cette classe ? Par exemple si on veut tester l'exactitude de la classe `PolynomeSecondDegre` ci-dessus, y a t-il quelque chose à ajouter à cette classe ?

une réponse : Non, pour tester la classe `PolynomeSecondDegre`, il n'y a rien à ajouter à cette classe.

Question 2)
JUnit est dit un framework. Rappeler ce qu'est un framework. Indiquer, entre autre, une différence entre un framework et une bibliothèque logicielle.

une réponse : Un framework n'est pas seulement une bibliothèque qu'on peut utiliser pour écrire un logiciel, mais aussi des règles d'utilisation de ces bibliothèques beaucoup plus précises et nombreuses qu'une simple utilisation de bibliothèque. Ces règles sont souvent des règles utilisant les notions orientés objets. De plus, la grande différence entre un framework et une bibliothèque, c'est que le framework prend en charge l'exécution du logiciel qui a été développé. Un framework et aussi (voire surtout) un environnement d'exécution. Le logiciel développé par le programmeur est exécuté dans un environnement faisant partie intrinséquement du framework. Ainsi, à l'exécution, le framework assure le cycle de vie des objets (lancement automatique de méthodes dans certaines situations), l'instanciation à bon escient d'objet (réserve d'objets), le lancement approprié de méthodes (sous section critique, dans des threads à part, lors d'arrivée de certains evénements), etc.

On rappelle qu'un polynome du second degré est une fonction de la forme $P(X) = a X^2 + b X + c$, a, b et c étant des constantes et X la variable. Par exemple $X^2 + X - 2$ est un polynome avec a = 1, b = 1 et c = -2. Une racine du polynome P(X) est un réel x_1 tel que $a x_1^2 + b x_1 + c = 0$. Par exemple 1 est racine du polynome $X^2 + X - 2$ car $1^2 + 1 - 2 = 0$.

Question 3)
Indiquez ce que fait la classe `PolynomeSecondDegre` .

une réponse : Cette classe modélise les polynomes du second degré. Elle permet d'en construire un avec le constructeur `public PolynomeSecondDegre(double a, double b, double c)`. De plus elle permet non seulement d'avoir son discriminant `b² - 4ac` permettant de savoir si ce polynome a des racines réelles, mais surtout de retourner ces racines réelles, quel que soit le polynome, à l'aide de la méthode `public double[] calculRacines()`. Plus précisément cette méthode retourne un tableau contenant les racines réelles, tableau qui est `null` si ce polynome n'a pas de racine réelle.

Question 4)
Construire une classe de test `test.TestEquationSecondDegre` qui teste la classe `PolynomeSecondDegre`. On demande d'utiliser la syntaxe JUnit 3. Cette classe de tests doit avoir trois méthodes :
- la méthode `public void test2Racines()` qui construit le polynome $X^2 + X - 2$ et qui vérifie que la méthode `calculRacines()`retourne bien les 2 racines 1 et -2 de ce polynome,
- la méthode `public void test1Racine()` qui construit le polynome $X^2 - 2X + 1$ et qui vérifie que la méthode `calculRacines()`retourne bien la seule racine 1 de ce polynome,
- la méthode `public void testPasDeRacineReelle()` qui construit le polynome $X^2 + X + 1$ et qui vérifie que la méthode `calculRacines()`retourne aucune racine de ce polynome.

une réponse :
Voici cette classe de test :

```
package test;
import jmf.equation.PolynomeSecondDegre;
import junit.framework.TestCase;

public class TestEquationSecondDegre extends TestCase {
     public void setUp(){
          System.out.println("dans setUp()");
     }

     public void tearDown() {
```

```
            System.out.println("dans tearDown()");
      }

      public void test2Racines() {
            PolynomeSecondDegre poly = new PolynomeSecondDegre(1, 1, -2);
            double racinesPressenties[] = new double[2];
            racinesPressenties[0] = 1.0;
            racinesPressenties[1] = -2.0;
            double[] racinesTrouvees = poly.calculRacines();

            assertTrue ((racinesTrouvees.length == 2) &&
                  ((racinesTrouvees[0] == racinesPressenties[0]) &&
(racinesTrouvees[1] == racinesPressenties[1]))
            ||   ((racinesTrouvees[0] == racinesPressenties[1]) &&
(racinesTrouvees[1] == racinesPressenties[0])));
      }


      public void test1Racine() {
            PolynomeSecondDegre poly = new PolynomeSecondDegre(1, -2, 1);
            double racinesPressentie[] = new double[1];
            racinesPressentie[0] = 1.0;
            double[] racinesTrouvees = poly.calculRacines();

            assertTrue ((racinesTrouvees.length == 1) && racinesTrouvees[0]
== racinesPressentie[0]);
      }


      public void testPasDeRacineReelle() {
            PolynomeSecondDegre poly = new PolynomeSecondDegre(1, 1, 1);
            double[] racinesTrouvees = poly.calculRacines();

            assertTrue (racinesTrouvees == null);
      }
}
```

La suite de problème a pour but de vous faire comprendre le fonctionnement de JUnit 3.
Dans la suite du problème, une classe (comme la classe `test.TestEquationSecondDegre`) qui permet de lancer des tests sera appelée une classe de tests.

A l'aide de ces deux classes on veut mettre en évidence le fonctionnement de JUnit.

Question 5)
On peut lancer JUnit 3 dans une fenêtre de commande. Par exemple, avec les classes ci dessus, dans une fenêtre de commande, on écrit :
`java junit.textui.TestRunner test.TestEquationSecondDegre`
Parmi les deux classes `junit.textui.TestRunner` et `test.TestEquationSecondDegre`, laquelle possède la méthode `public static void main(String args[ ])` ?

une réponse : C'est évidemment la classe `junit.textui.TestRunner` qui possède la méthode `main()` de lancement du programme. D'ailleurs la classe `test.TestEquationSecondDegre` est donnée ci dessus et ne dispose pas de méthode `main()` ! En fait la classe `junit.textui.TestRunner` est un environnement d'exécution de JUnit 3.

Pour simplifier le problème, on se met dans l'hypothèse où on a qu'une seule classe de tests qui posséde

éventuellement plusieurs méthodes de tests. C'est le cas dans notre exemple où nous avons une seule classe de tests, la classe `test.TestEquationSecondDegre` qui possède trois méthodes de tests `public void test2Racines()`, `public void test1Racine()` et `public void testPasDeRacineReelle()`.

Java est un langage de programmation qui est capable de découvrir, à l'exécution, les caractéristiques d'une classe ou les caractéristiques d'un objet d'une classe. Par exemple, à l'exécution, on peut indiquer, sur la ligne de commande, le nom d'une classe et le programme Java va trouver les méthodes de cette classe, trouver si elles sont `public`, etc. C'est le mécanisme d'introspection. De plus, à partir du nom de la classe, Java est capable de construire une instance de cette classe et, ayant trouvé une méthode de cette classe, de lancer sur cet objet, cette méthode : c'est le mécanisme de réflexion.
De ces faits, l'environnement d'exécution de JUnit 3 parcourt la classe à tester, recherche si cette classe possède des méthodes `testXXX()`, et, pour chacune de ces méthodes construit un objet de la classe de tests en précisant la méthode de test qui devra être lancé sur cet objet. Ainsi un test est un objet de classe de tests avec le nom d'une (seule) méthode de test qui devra être lancée sur cet objet et il y a autant d'objet test que de méthodes à tester. Sur notre exemple, un test est un objet de la classe `test.TestEquationSecondDegre`. En conclusion, l'environnement d'exécution JUnit 3, appelle successivement les méthodes à tester sur des objets indépendants, le résultat de chacune de ces méthodes est stocké et édité à la fin. La suite du problème consiste à montrer une implémentation de ce fonctionnement.

Question 6)
Comment est indiqué à l'environnement JUnit 3, que la classe donnée est une classe de tests ? Par exemple, qu'est ce qui indique que la classe `test.TestEquationSecondDegre` est une classe de tests ? Indiquer pourquoi cette classe possède un champ qui indique le nom d'une méthode à lancer parmi les méthodes de test. Donner une explication autre que celle indiquant qu'elle possède des méthodes `testXXX()`.

une réponse : Une classe est une classe de tests si elle dérive de `junit.framework.TestCase`. C'est le cas de la classe `test.TestEquationSecondDegre`. Dans la classe de base `junit.framework.TestCase`, il y a un champ (`private String fName`) qui indiquera le nom de la méthode de test à lancer (parmi les méthodes `testXXX()`). Donc les objets d'une classe de tests (ici `test.TestEquationSecondDegre`) possède un tel champ par héritage. Ce champ sera évidemmennt initialisé par l'environnement d'exécution JUnit 3.

Le framework JUnit 3 lance les tests de manière récursive (à l'aide du design pattern Composite qu'on ne demande pas d'étudier dans ce problème) en exécutant une méthode spécifique de nom `runBare()` sur chaque test. Cette méthode n'est d'ailleurs pas banale puisqu'elle a un corps de la forme :
initialisation (la méthode `setUp()`*)*
lance le test (la méthode de nom `runTest()` *qui lancera les méthodes de nom de la forme* `testXXX()`*)*
fin du test (la méthode `tearDown()`*)*

Question 7)
En fait, cette méthode `runBare()` n'est pas définie dans les classes de tests développées par les programmeur. Par exemple, il n'y a pas de méthode de nom `runBare()` dans la classe `test.TestEquationSecondDegre`. Pourquoi cette méthode `runBare()` peut être lancée sur des objets de la classe `test.TestEquationSecondDegre` ? On indiquera une technique orienté objet utilisée pour être sûr que cette méthode existe dans les classes de tests.

une réponse : Comme cette méthode a un corps, pour être sûr que les classes de tests possède cette méthode, il suffit d'imposer qu'une classe de test dérive d'une classe (éventuellement abstraite) possédant cette méthode avec ce corps. C'est en fait le cas. En JUnit 3 les classes de tests doivent être des classes dérivant de la classe abstraite `junit.framework.TestCase` et cette classe possède la méthode `runBare()` avec un corps non banal.

Pour chaque test, JUnit doit donner le résultat du test, c'est à dire si le test a échoué (failure, error, ...) ou non.

Pour cela un objet de la classe `junit.framework.TestResult` est créé au lancement de l'exécution JUnit 3. Cet objet va collecter les résultats des tests.
Le début du lancement du code de l'environnement JUnit 3 est de la forme :

```
TestResult result = new TestResult();
run(result);
```

L'environnement construit donc un `TestResult` qui va contenir les résultats de tous les tests. Pour une exécution, il n'y aura qu'un seul objet référencé `result` qui va contenir tous les résultats des tests. Pour ne pas passer cette argument `result` à toutes les méthodes qui seront ensuite appelées, le code de la méthode `runBare()` est plutôt de la forme :

```
setUp();
try {
     runTest(); //  qui lancera les méthodes de nom de la forme testXXX())
}
catch (AssertionFailedError e) {
     result.addFailure(..., e);
}
catch (Throwable e) {
     result.addError(..., e);
}
tearDown();
```

Comme dans notre exemple (la classe `test.TestEquationSecondDegre`), JUnit 3 utilise souvent des méthodes de la forme `assertXXX(...)`. Par exemple la méthode `assertTrue(boolean condition)` est codée :

```
protected void assertTrue(boolean condition) {
     if (!condition)
          throw new AssertionFailedError();
}
```

et où `AssertionFailedError` est une classe de la forme

```
public class AssertionFailedError extends Error { ... }
```

Question 8)
Comme une des dernières instructions de l'environnement d'exécution de JUnit 3 consiste à faire afficher les données de l'objet de la classe `TestResult` référencé par `result` ci dessus, indiquez comment, au fur et à mesure, de l'exécution des tests, si le test échoue, l'objet référencé par `result` est alimenté par des informations de résultats du test.

une réponse : Chaque fois qu'un test ne vérifie pas une condition `assertXXX()`, une exception est levée et le programme est dérouté dans la méthode `runBare()` dans le traitant d'exécution correspondant. Ces traitants d'exécution sont de la forme `result.addXXX(..., e)`. Au vu du nom de cette méthode, l'objet de la classe `TestResult` référencé par `result` accumule les informations du résultat du test. Ceci est fait pour chaque test et donc en fin d'exécution de tous les tests, l'objet référencé par `result` contient les informations de tous les tests qui ont échoués. C'est donc le mécanisme des exceptions qui permet d'alimenter l'objet référencé par `result` (ce qui est très astucieux ;-)).

Si dessous deux remarques (qui ne sont pas demandées aux étudiants).
Remarque 1 : les tests étant appelés les uns après les autres, l'objet référencé par `result` va contenir les informations de résultat de tous les tests.
Remarque 2 : Si le test passe avec succès, l'objet référencé par `result` n'est pas alimenté par une exception. En fait, seul un compteur est incrémenté. Si un test ne réussit pas, un booléen est levé. Ces instructions n'ont pas été décrites dans le problème.
A la fin de l'éxécution, suivant la valeur de ce booléen, l'environnemengt d'exécution JUnit 3 affiche que tout

s'est bien passé ou qu'il y a eu des errors ou failures (méthode `printFooter(TestResult result)` de la classe `junit.textui.ResultPrinter`).

Remarque :

Ce problème est inspiré de l'exposé se trouvant à l'URL `http://junit.sourceforge.net/doc/cookstour/cookstour.htm`. Mais la lecture du code de JUnit 3.8 qu'on peut téléchargé à l'URL `http://garr.dl.sourceforge.net/project/junit/junit/3.8.1/junit3.8.1.zip` a été bien utile.