iUT ORSAY
Département Informatique

DUT Informatique
Module JAVA Apprentis
2008 / 2009

Travaux Pratiques n° 2 : JUnit

Nom(s) :
Groupe :
Date :

Objectifs : Apprendre à vérifier une implantation à l'aide de tests unitaires sous JUnit. Pouvoir critiquer une spécification et partir d'elle pour implanter et vérifier une classe.

1 Notes de cours

1.1 Visibilité

Il existe différentes politiques de droits d'accès pour les membres d'une classe, qu'ils soient des constantes, des variables, des méthodes ou même des constructeurs :

- *Publique* (le champ ou la méthode est précédé du mot clé `public`) : le membre est accessible depuis l'extérieur de la classe, depuis toutes les autres classes ; le recours à une instruction `import` en début de fichier sera peut-être cependant nécessaire pour utiliser la classe elle-même.
- *Privée* (le champ ou la méthode est précédé du mot clé `private`) : le membre n'est accessible qu'aux méthodes de la classe.
- *Protégé* (le champ ou la méthode est précédé du mot clé `protected`) : le membre n'est accessible qu'aux méthodes de la classe et de toutes les classes qui en héritent.
- *Paquetage* (le champ ou la méthode n'est précédé d'aucun mot clé relatif à l'accès) : l'accès au membre est limité aux classes du même paquetage (pour mémoire, si aucune mention d'appartenance à un paquetage n'est précisée dans un fichier contenant une classe, celle-ci est considérée comme appartenant au paquetage par défaut).

1.2 JUnit

JUnit est un cadre de développement de tests pour applications JAVA. Il s'agît de créer des tests automatisés pour vérifier qu'une implantation respecte la spécification qui a été prévue. En pratique, pour chaque classe `MaClasse` d'une application va correspondre une classe de test `MaClasseTest` et pour chaque méthode `maMethode` de `MaClasse` qu'on souhaitera tester on écrira une méthode de test `testMaMethode` dans `MaClasseTest`.

Le code des classes et méthodes de test est semblable à celui des classes et méthodes classiques, avec cependant en plus des opérations d'*assertion*. Ces opérations sont des points de contrôle où on vérifie qu'une valeur obtenue est bien celle attendue. Quelques exemples sont :

– `static void assertEquals(int attendu, int observe)`
  prend en argument deux valeurs entières, celle attendue et celle observée, et vérifie que les deux valeurs sont égales.
– `static void assertEquals(float attendu, float observe, float precision)`
  prend en argument deux valeurs réelles, celle attendue et celle observée, et vérifie que les deux valeurs sont égales plus ou moins une troisième valeur de précision.
– `static void assertTrue(boolean condition)`
  vérifie que la valeur booléenne passée en argument est à `true`.
– `static void assertNotNull(Object object)`
  vérifie que la référence passée en argument n'est pas `null`.

Lorsqu'un test échoue, un rapport est envoyé (sous la forme d'une *exception* que l'on étudiera plus tard en détail) qui aidera à découvrir et corriger le(s) bug(s) d'implantation. Pour connaître tous les tests disponibles, vous pourrez vous reporter à la javadoc de la classe `Assert` du paquetage `junit` disponible à l'adresse `http://junit.sourceforge.net/javadoc`.

1.2.1 Un exemple de test

On considère une classe `Entier` destinée à contenir et manipuler un nombre entier. Le code est présenté en Figure 1.

```
/**
 * Classe Entier destinée à contenir un nombre entier.
 * @author Monprenom Monnom
 */
public class Entier {
        /**
         * Nombre entier
         */
        private int nombre ;

        /**
         * Contructeur.
         * @param valeur Pour initialiser l'attribut nombre.
         */
        public Entier(int nombre) {
                this.nombre = nombre;
        }

        /**
         * Double la valeur de l'attribut nombre.
         */
        public void doubler() {
                nombre *= 2;
        }

        /**
         * Getter de l'attribut nombre.
         * @return Renvoie la valeur de l'attribut nombre.
         */
        public int getValeur() {
                return nombre;
        }
}
```

FIG. 1 – Code du fichier *Entier.java*

On souhaite vérifier que le constructeur et la méthode `doubler` de la classe `Entier` fonctionnent comme on le souhaite. En utilisant Eclipse, on doit tout d'abord s'assurer que la bibliothèque JUnit est bien connue. Pour cela, clic droit dans partie exploration, *Build Path→Configure*

Build Path..., dans la fenêtre qui s'affiche, choisir l'onglet *Libraries* pour *Java Build Path*, faire *Add Library...→JUnit→Next* choisir *JUnit 3* et enfin *Finish→OK*.

Pour créer une classe de test JUnit à partir d'Eclipse : se placer dans partie exploration sur la classe que l'on souhaite tester, clic droit, *New→JUnit Test Case*. Choisir *New JUnit 3 test*, cocher *setUp()* et éventuellement *tearDown()* dans la boîte de dialogue qui s'ouvre et cliquez sur *Next*. Ensuite Eclipse propose la liste des méthodes que l'on souhaite tester. Il ne reste qu'à choisir (dans notre exemple, `Entier(int)` et `doubler`) puis enfin *Finish*. Eclipse génère automatiquement une classe `EntierTest` avec ses méthodes qui ne restent qu'à remplir. **N'ajoutez du code qu'à la fin des méthodes**. Les mots clé `extends`, `throws` et `super` vous sont peut être encore inconnus, vous découvrirez bientôt leur rôle. Un exemple complet est présenté en Figure 2.

```
import junit.framework.TestCase;

/**
 * Classe de test JUnit pour la classe Entier.
 * @author Monprenom Monnom
 */
public class EntierTest extends TestCase {

        /**
         * Attribut référence vers un objet de type Entier.
         */
        Entier entier;

        /**
         * Méthode executée avant chaque méthode de test.
         */
        protected void setUp() throws Exception {
                super.setUp();
                System.out.println("Début d'un test.");
                entier = new Entier(2); // Création d'un Entier avec nombre = 2.
        }

        /**
         * Méthode executée après chaque méthode de test.
         */
        protected void tearDown() throws Exception {
                super.tearDown();
                System.out.println("Fin d'un test.");
        }

        /**
         * Méthode de test pour le contructeur de la classe Entier.
         */
        public void testEntier() {
                System.out.println("Test du constructeur.");
                assertNotNull(entier); // L'objet doit bien avoir été créé.
                assertEquals(2,entier.getValeur()); // Son nombre doit valoir 2.
        }

        /**
         * Méthode de test pour la méthode doubler de la classe Entier.
         */
        public void testDoubler() {
                System.out.println("Test de doubler.");
                entier.doubler(); // On double la valeur du nombre.
                assertEquals(4,entier.getValeur()); // Il doit donc valoir 4.
        }
}
```

FIG. 2 – Code du fichier *EntierTest.java*

Pour lancer le test à partir d'Eclipse, clic droit dans la partie navigation sur la classe de test, *Run As→JUnit Test*. Un nouvel onglet *JUnit* s'ouvre alors, contenant le rapport d'erreur (avec notre exemple il ne devrait y avoir aucune erreur), et l'onglet *Console* imprime le résultat de l'exécution. En pratique chaque test se déroule de la manière suivante : (1) la méthode `setUp` est exécutée si elle a été définie, on y fait ce qu'il y a de commun au début de tous les tests, (2) une méthode de test est exécutée (il n'y a pas d'ordre précis) et (3) la méthode `tearDown` est exécutée si elle a été définie. Pour plus d'informations :

`http://www.junit.org`

2 Description du problème

On considère une application dédiée à la saisie et à l'étude des notes d'un groupe d'étudiants. Le programme demande à l'utilisateur le nombre de notes qu'il souhaite entrer puis les notes une à une. Quand toutes les notes ont été entrées, il affiche la liste des notes et des statistiques. Cette application est construite autour de trois objets dont les classes vous sont fournies (voir `http://www.lri.fr/~bastoul/teaching/java`) :

– Un objet de type `Notes` qui est destiné à contenir la liste des notes, les statistiques sur ces notes ainsi qu'à effectuer des opérations sur cette liste. Le diagramme UML de la classe `Notes` ainsi que sa spécification vous sont fournis en Figure 3. Pour rappel, en UML un « + » signifie *publique*, un « - » signifie *privé* et le soulignement désigne les constantes.
– Un objet de type `Clavier` dédié à la saisie d'informations à partir du clavier.
– Un objet utilitaire de type `Principale` qui contient la méthode principale et orchestre l'utilisation des deux premiers objets.

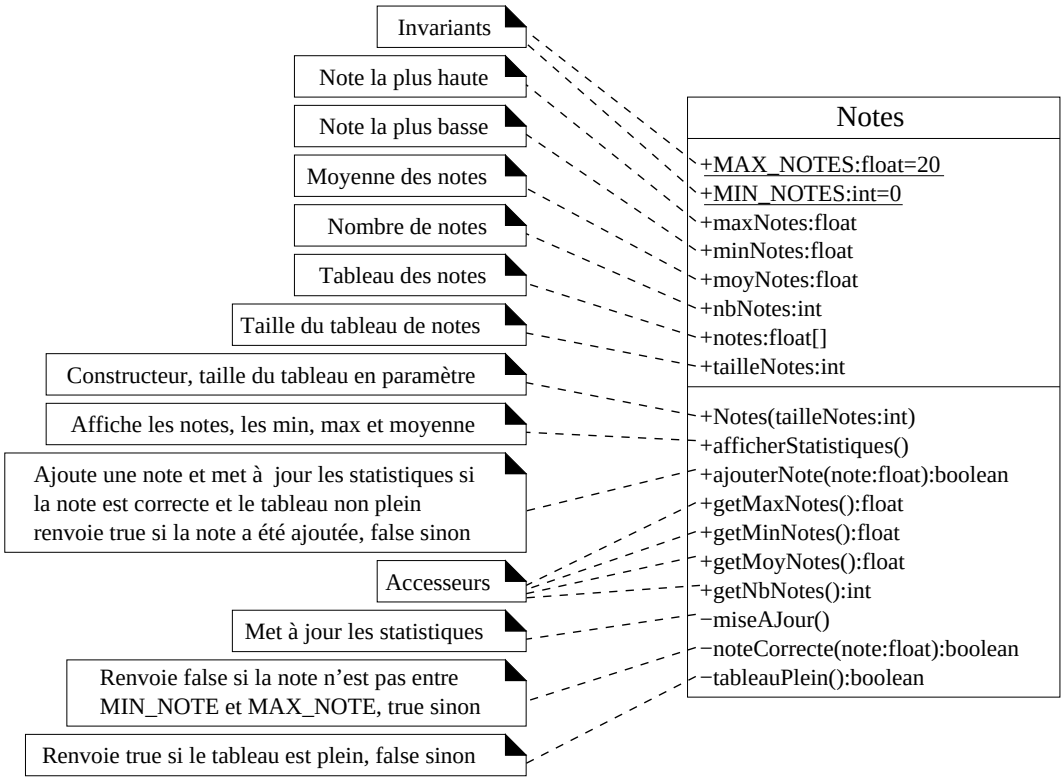

FIG. 3 – Spécification UML de la classe `Notes`

3 Vérification du respect d'une spécification

Respectez l'ordre des étapes, dans cette section il est interdit de modifier les codes donnés !

Importez l'archive *Notes.zip* sous Eclipse. Pour cela, dans le menu *File*, choisissez *Import*, puis *General→Existing Projects into Workspace*, faites alors *Select archive file*, *Browse* jusqu'au fichier et enfin *Finish*. Lancez une première exécution du programme et observez son comportement. Copiez ci-dessous le résultat d'une exécution.

Conformément à la spécification définie en Section 2, écrire un test JUnit pour la classe `Notes` du paquetage `notation` (ne pas tester les accesseurs et les routines d'affichage) en suivant la spécification de l'objet. **Vous joindrez le code de cette classe de test au compte-rendu**. La classe `Notes` du paquetage `notation` passe-t-elle avec succès le test que vous avez défini ?

Plusieurs autres équipes de programmeurs ont écrit leur propre version de la classe `Notes` dans des paquetages `notation_v`x avec $1 \leq x \leq 4$. Pour les utiliser dans la classe `Principale` à la place de celle du paquetage `notation`, il suffit de modifier l'instruction `import` en tout début de fichier.

Exécutez le programme en utilisant chacune de ces quatre classes et **les notes (exactement et dans cet ordre) 0, 5, 10 15 et 20**. Pour chaque version, est-ce que le programme semble fonctionner correctement ?

Utilisez votre classe de test, **telle que vous l'avez écrite**, pour tester chaque version. Donner honnêtement (jouez le jeu, cette partie n'a aucune incidence sur une quelconque note !) les rapports d'erreur pour chaque version de la classe `Notes`.

Améliorez votre classe de test, normalement aucune classe, sauf celle du paquetage `notation`, ne devrait passer le test. **Vous joindrez le code de cette nouvelle classe de test au compte-rendu**.

4 Respect de l'encapsulation

À la fin de la méthode `main` dans la classe `Principale`, essayez de modifier directement des attributs de l'objet référencé par `listeNotes` puis de réafficher les statistiques, par exemple changez la première note en ajoutant les instructions suivantes (et n'entrez que des valeurs supérieures à 2 lors de l'exécution) :

```
listeNotes.notes[0] = 2 ;
listeNotes.afficherStatistiques() ;
```

Est-ce qu'une erreur est détectée à la compilation ? À l'exécution ? Les valeurs affichées sont-elles cohérentes (par exemple, la moyenne affichée est-elle bien correcte) ?

On peut remarquer que la politique d'accès pour tous les attributs est publique. Corrigez cela de manière à ce que l'utilisateur de la classe ne puisse pas faire n'importe quoi. Que doit-on modifier dans le diagramme UML ?

Tentez à nouveau de modifier directement des attributs rendus privés ou d'appeler des méthodes rendues privées. Que se passe-t-il ? À quel niveau (compilation, exécution) une erreur est-elle repérée ?

Quelles sont vos conclusions sur l'intérêt de l'encapsulation ? Sur les compétences de celui qui a écrit la spécification ? Un bon programmeur aurait-il dû corriger l'erreur de spécification dans le programme ?

5 Changement d'implantation

La programmation orientée objet permet de facilement remplacer un objet par un autre sans se soucier du reste de l'application, du moment que l'interface avec l'extérieur (ce qui n'est pas privé) ne change pas de nom ou de fonctionnement. Dans l'application présentée dans la première partie du TP, une implantation à base de tableau pour maintenir la liste des notes était proposée pour la classe `Notes`. Cette implantation a pour limitation d'obliger à définir un nombre maximum de notes à entrer (la taille du tableau).

On souhaite écrire une nouvelle version plus puissante de la classe `Notes`, pour cela on souhaite utiliser la classe `ArrayList` proposée par JAVA pour stocker les notes (voir la Javadoc pour plus d'informations).

Réaliser une nouvelle classe `Notes` utilisant `ArrayList`. **Vous joindrez le code de cette nouvelle classe au compte-rendu**.

Remplacer l'ancienne classe par la nouvelle et vérifier que le fonctionnement du programme est toujours correct, en particulier, utilisez votre test JUnit. **Il est interdit de modifier les autres objets**. Votre nouvelle classe passe-t-elle votre test ?

6 Second changement d'implantation (facultatif)

Écrire une troisième classe `Notes` (et son test JUnit) qui a une spécification plus riche : elle permet d'afficher toutes les notes dans l'ordre où elles ont été entrées et permet le remplacement de la $i^{ème}$ note par une nouvelle et ne change rien si la position i n'est pas cohérente. Vous joindrez le code de cette nouvelle classe et de sa classe de test au compte-rendu.