Génie Logiciel Avancé

Année 2010-2011

Prof. Burkhart Wolff
wolff@lri.fr

Delphine Longuet, Lina Bentakouk
longuet@lri.fr, bentakou@lri.fr

TP - Test unitaire avec JUnit

Semaine du 15 novembre 2010

L'objectif de ce TP est d'écrire et d'exécuter des tests avec JUnit pour une classe Java implantant un conteneur. À partir de la spécification informelle donnée, vous devez écrire un ensemble de classes JUnit de façon à pouvoir tester des implantations de la classe `Conteneur` en boîte noire. Les squelettes des classes de test ainsi que les implantations à tester sont disponibles à l'adresse `http://www.lri.fr/~longuet/Enseignements/Conteneur.tar.gz`.

Spécification du conteneur On considère une classe `Conteneur` dont les instances stockent des couples clef-valeur. Les clefs sont des instances (garanties non `null`) de `Object`, de même que les valeurs. Une valeur peut être associée à deux clefs distinctes. Un conteneur a une capacité fixée à l'initialisation, strictement supérieure à 1, qu'on peut agrandir quand le conteneur est plein (et seulement dans ce cas). On peut ajouter et retirer des couples clef-valeur d'un conteneur, tester si une clef est présente dans un conteneur, ainsi que retrouver dans ce cas la valeur associée. Si on tente de retirer une clef absente d'une conteneur, rien ne se passe. Si on ajoute un couple clef-valeur pour une clef qui existe déjà, l'ancien couple est écrasé. Si on ajoute un couple clef-valeur à un conteneur plein, l'exception `DébordementConteneur` (sous-classe de `ErreurConteneur`) est signalée. Si on cherche la valeur d'une clef absente, l'exception `ErreurConteneur` est signalée, de même que si on applique une méthode dans un état où elle est interdite.

Quand un conteneur est plein, et uniquement dans ce cas, on peut le redimensionner en fixant une capacité plus grande. Quand le conteneur n'est pas vide, et uniquement dans ce cas, on peut vider le conteneur en supprimant tous les éléments, sans changer sa capacité.

Parmi les autres fonctionnalités, on peut savoir si un conteneur est vide, connaître sa capacité ainsi que son nombre courant de couples clef-valeur.

Le squelette de la classe Java `Conteneur` est le suivant :

```
public class Conteneur {

    public Conteneur(int n) throws ErreurConteneur { }
    public void ajouter(Object C, Object O) throws ErreurConteneur { }
    public void retirer(Object C) { }
    public void raz() throws ErreurConteneur { }
    public void redimensionner(int nouv) throws ErreurConteneur { }
    public boolean present(Object C) { }
    public Object valeur(Object C) throws ErreurConteneur { }
    public boolean estVide() { }
    public int taille() { }
    public int capacite() { }

}
```

Introduction à JUnit JUnit est un outil permettant d'écrire et d'exécuter des tests unitaires sur des programmes Java. Il est disponible à l'adresse `http://www.junit.org/` mais est également directement intégré à Eclipse.

Les tests en JUnit sont regroupés au sein d'une classe Java qui doit hériter de la classe `TestCase`. Le nom des méthodes de test doit commencer par `test`. Le corps d'une méthode de test doit comporter trois parties :

- le *préambule*, qui permet l'initialisation des objets sur lesquels le test va être exécuté ;
- le *corps de test*, dans lequel la méthode à tester est appelée sur les objets créés ;
- le *postambule*, qui permet de délivrer le verdict du test (succès ou échec) en vérifiant un ensemble de propriétés (assertions) sur l'état des objets après le test. Le tableau 1 résume les différentes assertions possibles en JUnit.

▶ Sous Eclipse, créez un nouveau projet à partir du répertoire TestConteneur fourni. Ajoutez JUnit3 au *classpath* : clic droit sur le projet > Build Path > Add Librairies > Junit3. Puis ajoutez une des implantations de la classe `Conteneur` fournies : clic droit sur le projet > Build Path > Add External Archives, puis ajouter le fichier `testEtat7.jar` du répertoire Implantations, par exemple.

▶ Ouvrez la classe `TestConteneur` donnée en exemple. Elle contient trois méthodes de test. La méthode `testCapaciteSup1` vérifie que l'initialisation d'un conteneur avec une capacité strictement supérieure à 1 s'effectue correctement, c'est-à-dire que le conteneur est bien créé, de taille 0, de capacité égale à celle spécifiée en argument, et vide. Comme l'initialisation ne doit pas lever d'exception, le test doit échouer si une exception est levée. On rattrape alors l'exception et on force l'échec du test avec la méthode `fail()`.

La méthode `testAjouterPresentPlein` vérifie que l'ajout d'un élément dont la clé est déjà présente dans un conteneur plein est possible (ne lève pas d'exception) et a pour effet d'écraser la valeur précédemment associée à la clé. Comme dans le cas précédent, on force le test à échouer si une exception est levée au cours de l'exécution.

La méthode `testRazVide` vérifie que la remise à zéro lève une exception dans le cas où le conteneur est vide.

▶ Pour exécuter ces tests sur une implantation en boîte noire (une archive .jar), il faut ajouter le fichier au *classpath* comme vu précédemment. Exécutez ensuite `TestConteneur` en tant que test JUnit sur l'implantation `testEtat7.jar`. Le résultat des tests apparaît dans un nouvel onglet : un test ayant levé une exception non rattrapée est répertorié dans *Errors*, un test ayant échoué (*AssertionFailedError*) est répertorié dans *Failures*.

Il est possible de grouper les tests ayant un préambule commun (c'est-à-dire devant être exécutés dans le même état) en une classe et de définir des méthodes globales appelées `setUp` et `tearDown` permettant respectivement l'initialisation et la remise à zéro des objets avant et après chaque test.

▶ Ouvrez la classe `TestConteneurPlein` qui illustre l'utilisation de ces méthodes. On remarque qu'on rattrape également une éventuelle exception dans le corps de la méthode `setUp`, au cas où l'initialisation échoue. La méthode `suite` sert à construire une suite de tests à partir des méthodes `setUp` et `tearDown` et des méthodes de test. Cette suite de test correspond à l'exécution des méthodes selon l'ordre :

```
setUp(); test1(); tearDown(); setUp(); test2(); tearDown(); ...
```

▶ Exécutez `TestConteneurPlein` sur l'implantation `testEtat7.jar`.

Exercice : écriture des tests Les tests pour le conteneur vont être répartis dans 4 classes selon l'état initial dans lequel les tests doivent être effectués : `TestInitialisation`, `TestConteneurVide`, `TestConteneurNonVide` et `TestConteneurPlein`.

▶ Complétez les squelettes des 4 classes de test fournies en suivant les exemples précédemment donnés. En particulier, pour chacun des tests, précisez en commentaire l'objectif du test ainsi que le résultat attendu. Pensez à tester aussi bien les cas qui doivent réussir que les cas qui doivent lever une exception : l'objectif est de couvrir un maximum de cas.

▶ **À rendre pour le vendredi 26 novembre** (par mail aux chargés de TD) :
- les quatre classes de test complétées ;
- un fichier (texte ou tableau) résumant les résultats des tests sur les 17 implantations fournies : pour chaque implantation, dites si elle réussit ou échoue aux tests et expliquez les fautes trouvées. Vous pouvez suivre le modèle suivant :

Implantation	**Résultats des tests**	**Fautes trouvées**
testEtat7	Échec	L'ajout d'un couple dont la clé est déjà présente n'écrase pas l'ancien couple mais ajoute le couple comme un nouvel élément.

Méthode	**Rôle**
`assertEquals(Object a, Object b)`	Vérifie que les objets a et b sont égaux
`assertSame(Object a, Object b)`	Vérifie que a et b sont des références vers le même objet
`assertNotSame(Object a, Object b)`	Vérifie que a et b ne sont pas des références vers le même objet
`assertNull(Object o)`	Vérifie que l'objet o est null
`assertNotNull(Object o)`	Vérifie que l'objet o n'est pas null
`assertTrue(boolean e)`	Vérifie que l'expression e est vraie
`assertFalse(boolean e)`	Vérifie que l'expression e est fausse
`fail()`	Provoque l'échec du test

FIG. 1 – Méthodes d'assertions en JUnit