Prof. Burkhart Wolff
wolff@lri.fr

Kostia Chardonnet
kostia@lri.fr

TP - Test unitaire avec JUnit

Semaine du 1er fevrier 2020

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 et les implantations à tester sont disponibles ici : `https://www.lri.fr/~wolff/tmp/TestConteneur.zip`.

Spécification du conteneur. On considère une classe `Conteneur` dont les instances stockent des couples clé-valeur. Les clés sont des instances de `Object`, de même que les valeurs. On supposera que les clés et les valeurs ne sont jamais `null`. Une valeur peut être associée à deux clés 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 clé-valeur d'un conteneur, tester si une clé est présente dans un conteneur, ainsi que retrouver dans ce cas la valeur associée. Si on tente de retirer une clé absente d'une conteneur, rien ne se passe. Si on ajoute un couple clé-valeur pour une clé qui existe déjà, l'ancien couple est écrasé. Si on ajoute un couple clé-valeur à un conteneur plein, l'exception `DebordementConteneur` (sous-classe de `ErreurConteneur`) est signalée. Si on cherche la valeur d'une clé 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 clé-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 4. JUnit est un outil permettant d'écrire et d'exécuter des tests unitaires sur des programmes Java. Il est intégré à Eclipse mais est également disponible à l'adresse `http://www.junit.org/`.

Un test en JUnit 4 est une méthode annotée par `@Test`. Les méthodes de test sont généralement regroupées en une classe dédiée aux tests. Le corps d'une méthode de test doit comporter quatre parties :

- le *préambule*, qui permet de créer les objets et de les amener dans l'état nécessaire pour le test ;
- le *corps de test*, dans lequel la méthode à tester est appelée sur les objets créés ;
- l'*identification*, 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.
- le *postambule*, qui réinitialise les objets.

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 une méthode qui exécutera ce préambule avant chacun des tests de la classe. Cette méthode doit être annotée par `@Before`. De la même manière, si tous les tests d'une classe ont un postambule commun, on peut définir une méthode annotée par `@After` qui sera exécutée après chacun des tests de la classe.

Mise en route.

▶ Démarrez Eclipse dans un shell (pas via le KDE). Sous Eclipse, créez un nouveau projet Java. Créez un package `test` dans `src` et importez dans ce package les fichiers du répertoire TestConteneur fourni : clic droit sur le package > Import > File System puis dans le répertoire TestConteneur, sélectionnez tous les fichiers. Ajoutez ensuite JUnit 4 au *classpath* : clic droit sur le projet > Build Path > Add Librairies > JUnit4. Enfin, ajoutez une des implantations de la classe `Conteneur` fournies : clic droit sur le projet > Build Path > Add External Archives, puis ajoutez le fichier `testEtat7.jar` du répertoire Implantations, par exemple.

▶ Ouvrez les quatre classes Java fournies. La classe `TestPlein` contient la méthode `ajouterPresentPlein`, qui 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. Le test doit échouer si une exception est levée, on rattrape donc l'exception et on force l'échec du test avec la méthode `fail()`.

Le préambule du test se trouve dans la méthode `creerConteneurPlein`. On remarque qu'on rattrape également une éventuelle exception dans le corps de la méthode `creerConteneurPlein`, au cas où l'initialisation échoue.

▶ 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 `TestPlein` 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*. Un double clic sur le nom du test qui a échoué montre l'assertion qui a été violée.

Questions

1. À partir de la spécification de la classe `Conteneur` donnée, écrivez un ensemble de tests JUnit pour cette classe. Vous compléterez les squelettes des quatre classes fournies. Pour chaque test, vous préciserez *obligatoirement* en commentaire l'objectif du test et 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 différents parmi les cas possibles. Pensez également aux cas aux limites.
2. Exécutez vos tests sur chacune des implantations fournies et rédigez un rapport de test sous la forme d'un tableau : pour chaque implantation, dites si les tests ont réussi ou échoué et donnez les raisons apparentes des 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.

3. Déposez dans la section "travaux" de Dokeos, dans le sous-dossier correspondant à votre groupe de TP, une archive `.zip` contenant uniquement les quatre classes de test complétées (les quatre fichiers .java se trouvant dans votre workspace sous `TestConteneur/src/test/`) et votre rapport dans un format texte ou pdf.

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

FIGURE 1 – Méthodes d'assertions en JUnit