Tests (brièvement)

Les tests.

On appelle **test unitaires** des tests permettant de vérifier le bon fonctionnement d'une partie spécifique d'un logiciel. C'est ce que vous réaliserez en écrivant des tests pour chaque méthode.

Méthodologie. Une fois effectuée l'analyse objet d'un problème on arrive à la phase effective de développement et d'écritude du code. Adopter et respecter une démarche méthodologique rigoureuse favorise la production efficace de code fiable et correct. Il est par exemple conseillé de travailler **une méthode à la fois** en appliquant la démarche suivante :

1. **écrire la signature de la méthode,**
2. **écrire la documentation (javadoc) de la méthode,**
3. **écrire les tests qui permettront de vérifier que le code produit pour la méthode est correct,**
4. **écrire le code,**
5. **exécuter les tests prévus à l'étape 3, en vérifiant que les tests des méthodes précédemment écrites (et testées) restent réussis : test de non régression**[a]**,**
6. **si les tests sont réussis passer à la méthode suivante (étape 1) sinon recommencer à l'étape 4.**

[a] On s'assure que le nouveau code écrit ne remet pas en cause les codes précédents.

On constate donc que documentation et tests s'écrivent **avant** le code du corps des méthodoes. La documentation et les tests, **écrits avant le code**, fournissent la spécification complète, et exécutable pour la partie tests, de la méthode.

JUnit. Nous utiliserons le framework de tests JUnit qui fait référence pour l'écriture et l'exécution des tests. Plus particulièrement nous utiliserons JUnit4.

Classe de tests. Le principe est de placer les tests dans une classe dédiée.
Il faut ajouter en entête de cette classe les lignes d'import nécessaires, liées à JUnit :

```
import org.junit.*;
import static org.junit.Assert.*;
```

Et vous devrez certainement ajouter les `import` vers les classes de votre application utilisées dans les tests.
Précisons, si nécessaire, que les classes de test, comme leur nom l'indique, sont des classes Java. Elles peuvent donc définir des attributs, des méthodes (autres que les méthodes de tests), etc., en fait tuot ce qui peut-être nécessaire à la bonne écriture des tests.

Méthodes de tests. Dans cette classe on crée autant de méthodes que de tests que l'on souhaite réaliser. Il peut y avoir plusieurs méthodes de tests pour une même méthode d'une classe testée. Même si cela amène à écrire un peu plus de code, car plusieurs méthodes de tests, il est conseillé d'écrire un méthode par point testé (principe "unitaire" des tests).
Le nom de ces méthodes est libre, mais il convient de choisir un nom qui exprime ce qui est testé par la méthode.
Ces méthodes doivent obligatoirement :

- être précédées de l'annotation `@Test`
- avoir pour signature : `public void` *`nomMethode`*`()`

Elles contiennent le code exécuté pour le test, en particulier une ou plusieurs **assertions de test** qui doivent être vérifiées pour que le test soit considéré réussi :

- `assertTrue(v)` vérifie que la valeur fournie en paramètre vaut `true`,
- `assertFalse(v)` vérifie que la valeur fournie en paramètre vaut `false`,

- `assertEquals(expected, actual)` vérifie l'égalité de deux valeurs passées en paramètre, en utilisant `equals()` pour les objets.

  Lorsque les valeurs `expected` et `actual` sont des nombres flottants, compte-tenu de l'imprécision des calculs sur ces nombres, il faut rajouter un troisième paramètre, `epsilon`, qui représente l'intervalle de précision tolérée pour l'égalité. Dans ce cas, le test `assertEquals(expected, actual, espilon)` sera considéré comme un succès si |`expected - actual`| `< epsilon`.

- `assertNull(ref)` (respectivement `assertNotNull(ref)`) vérifie que lé référence fournie en paramètre est (respectivement n'est pas) `null`

- `assertSame(expected, actual)` vérifie en utilisant `==` que les deux références fournies en paramètre correspondent au même objet (`assertNotSame(expected, actual)` existe également)

- `fail()` échoue toujours

Pour `assertEquals()` et `assertSame()`, il faut faire attention à distinguer les 2 paramètres (`expected` et `actual`). L'ordre est important car les messages d'erreurs et les outils de comparaison en tiennent compte. Le premier paramètre est la valeur que l'on veut obtenir, le second est celle que l'on teste.
Chacune de ces méthodes peut prendre en premier paramètre une chaîne de caractères qui permet d'identifier le test réalisé, ce message est notamment utile en cas d'échec au test concerné.
Exemple : `assertEquals("disc radius should be 12", 12, disc.getRadius());`

> **Exercice.** *Montrez que la seule assertion `assertTrue` est suffisante en proposant une réécriture de toutes les autres assertions en n'utilisant que celle-ci.*
>
> **Le travail le plus compliqué (et c'est une tâche réellement difficile) est de bien construire les tests réalisés afin qu'ils permettent de s'assurer avec un maximum de certitude du bon fonctionnement de la méthode. Se pose en particulier le problème de la complétude des tests écrits.**

Tester les exceptions. Pour tester la levée d'une exception on peut utiliser une annotation particulière. La méthode de tests doit être précédée par :

`@Test(expected=`*`MaClasseDException`*`.class)`

Le test sera réussi si et seulement si une exception du type mentionné est levée lors de l'exécution de la méthode. Dans le cas contraire il échoue.

> **Exercice.** *Ecrire un autre code de méthode de test équivalente mais n'utilisant que les assertions précédentes et donc sans la forme `(expected=...)` (même si on préférera cette forme dans la pratique.).*

Méthodes complémentaires. L'initialisation des objets nécessaires aux tests se répète le plus souvent d'un test à l'autre. Afin de faciliter l'écriture des méthodes de tests il est possible de définir et désigner des méthodes qui seront exécutées avant et après **chacune** des méthodes de tests.
Il s'agit à nouveau de méthodes dont la signature sera `public void` *`nomMethode`*, mais cette fois elles sont précédées de l'annotation `@Before` pour celles exécutées avant, et `@After` pour celles exécutées après.
Il peut y avoir plusieurs méthodes ainsi annotées, toutes sont exécutées avant, ou après, chacune des méthodes de tests. Il s'agit le plus souvent pour les premières de réaliser des initialisations et pour les secondes de « nettoyer » celles-ci.
En complément il est possible également de définir des méthodes qui sont exécutées **une et une seule** :

- avant l'exécution de l'ensemble des méthodes de tests, elles sont annotées par `@BeforeClass`.

- après l'exécution de l'ensemble des méthodes de tests, elles sont annotées par `@AfterClass`.

Exemple

```
package generics;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class CollectorTest {

   // attributs de la classe de test
   private Collector<Carrot> collector;
   private Carrot carrot;

   // méthode exécutée avant chaque méthode de test
   @Before
   public void setupBefore() {
      collector = new Collector<Carrot>("timoleon");
      carrot = new Carrot(1);
   }

   // méthode de test de "bon fonctionnement" avec assertions
   @Test
   public void testCollectAndGetCollected() {
     // initially nothing is carried
     assertNull(collector.getCarriedObject());
     collector.take(carrot);
     // carrot is now the collected object
     assertSame(carrot, collector.getCarriedObject());
   }

   // méthode de test d'une levée d'exception
   @Test(expected=AlreadyCarryingException.class)
   public void testCollectThrowsExceptionWhenAlkreadyCarrying() {
     collector.take(carrot);
     // collector carries something
     assertNotNull(collector.getCarriedObject());
     Carrot c2 = new Carrot(2);
     // throws the exception
     collector.take(c2);
   }
}
```