Les JUnits

Contenu

Les tests

Quand on écrit une application, il faut toujours la tester. Il existe plusieurs sortes de tests :

- les **tests unitaires** : ils permettent de tester si une méthode remplit bien son contrat ;
- les **tests d'intégration** : ils permettent de tester les interactions entre les objets, les services,… ;
- les **tests de fonctionnalité** : ils permettent de tester la réaction de l'application à une requête d'un utilisateur ;
- les **tests de performance** : cela regroupe entre autre les tests de charge (traiter un grand nombre de données ou de requêtes,…) et les tests de stress (retirer les ressources de l'application et effectuer des tests de récupération).
- les **tests d'acceptation** : ce sont des tests exécutés par les clients afin de vérifier que l'application répond bien à leurs besoins.

JUnit est un outil ayant pour but de faciliter l'écriture des tests unitaires. Dans ce chapitre, on va voir comment écrire des tests unitaires en utilisant les **Junit 4**. JUnit 4 utilise des annotations afin de marquer les classes et les méthodes de test.

Les tests unitaires

Les tests unitaires ont pour objectifs de tester si une méthode remplit bien son **contrat** càd qu'elle fait ce qu'elle doit faire. Par exemple :

- elle teste la validité des paramètres
- elle envoie bien une exception dans tel ou tel cas
- elle renvoie la valeur adéquate
- elle met à jour les attributs
- etc

On écrit une classe de test par classe testée. Chaque méthode de la classe peut être testée par plusieurs méthodes de la classe de test. En général, pour une méthode qui s'intitule `uneMethode`, on définira une méthode de test `testUneMethode`. Si plusieurs méthodes testent cette méthode, on les numérotera : `testUneMethode1`, `testUneMethode2`, `testUneMethode3`, …

Outre divers imports expliqués dans le point qui suit, une classe de test possède comme toute classe Java un état et un comportement. Dans l'état, on précise les objets qu'on va utiliser à plusieurs reprises dans les différents tests. Dans le comportement, on indique toutes les méthodes de tests.

Une **méthode de test** est annotée **`@Test`** :

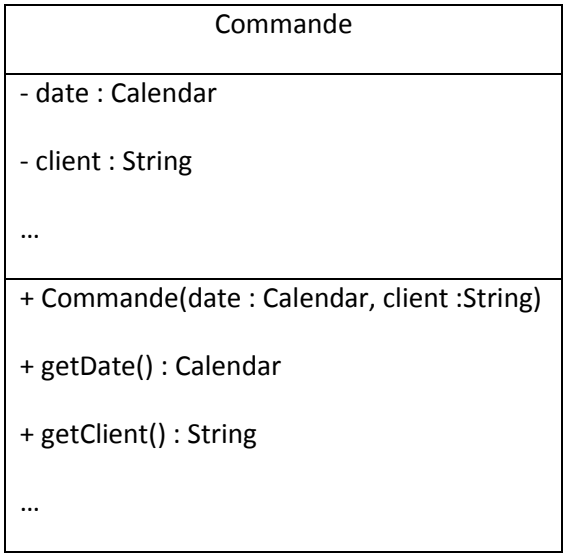
```
@Test
public void testGetClient(){ … }
```

On **initialise** l'état avec une méthode annotée **`@Before`**. Cette méthode sera exécutée avant chaque méthode de test. On peut éventuellement aussi y ouvrir une connexion à une base de donnée ou …

Une méthode annotée **`@After`** permet de préciser un comportement qui sera exécuté **après** chaque méthode de test.

Un exemple de tests avec JUnit 4

Partons d'un exemple afin d'expliquer comment écrire une classe de tests avec JUnit. Supposons qu'on veuille écrire une classe de test pour la classe `Commande` dont le diagramme UML est donné ci-dessous :



Ecrivons maintenant la classe de test :

```
package tests;

import static org.junit.Assert.*;

import java.util.Calendar;
import java.util.GregorianCalendar;
/*pour pouvoir utiliser directement les méthodes de la classe Assert fournie par
JUnit */
import org.junit.Test;
/*nécessaire pour écrire les méthodes de tests*/
import org.junit.After;
/*nécessaire pour écrire une méthode qui s'exécute après chaque méthode de test */

import org.junit.Before;

/*nécessaire si on veut écrire une méthode qui s'exécute avant chaque méthode de
test */

public class TestCommande {

	private Commande commande;
	/*
	 * Déclaration des champs qui seront utilisés dans le test
	 */

	private Calendar date;

	@Before
	/*
	 * annotation qui signifie que la méthode suivante sera exécutée avant
	 * chaque méthode de test.
	 */
	public void setUp() {
		date = new GregorianCalendar();
		commande = new Commande(date, "Leconte");
	}

	@After
	/*
	 * annotation qui signifie que la méthode suivante sera exécutée après
	 * chaque méthode de test. Comme ici il n'y a rien a faire, on n'écrira pas
	 * cette méthode.
	 */
	public void tearDown() {/* … */
	}

	@Test
	/* annotation pour dire qu'il s'agit d'une méthode de test */
	public void testGetClient() {
		/*
		 * méthode qui vérifie que les deux paramètres sont égaux en utilisant
		 * la méthode equals.Il faut mettre en premier lieu ce qui est attendu
		 * et, en second, ce que renvoie la méthode.
		 */
		assertEquals("Leconte", commande.getClient());
	}
	/*
	 * annotation pour dire qu'il s'agit d'une méthode de test et qu'une
	 * IllegalArgumentException devrait être lancée.
	 */

	@Test(expected = IllegalArgumentException.class)
	public void testConstructeurCommande(){
		new Commande(null, "Leconte");
	}
}
```

En théorie, pour écrire une classe de tests avec JUnit 4 :

- Il faut importer **`org.junit.Test`** pour dire pouvoir écrire des tests.
- Chaque méthode de test doit être précédée de l'annotation **`@Test`** ou **`@Test(expected=…)`** où … doit être remplacée l'exception que devrait lancer la méthode. Toutes les méthodes de test doivent être `public` et `void`.
- Afin de pouvoir utiliser les méthodes proposées par JUnit 4, il faut importer **`org.junit.Assert.*`**.
- On peut écrire une méthode qui s'exécutera avant chaque méthode de test. Pour cela, il faut mettre l'annotation **`@Before`** avant cette méthode et il faut importer `org.junit.Before`. Elle doit également être `public` et `void`. Cette méthode est en général utilisée pour initialiser des variables.
- On peut écrire une méthode qui s'exécutera après chaque méthode de test. Pour cela, il faut mettre l'annotation **`@After`** avant cette méthode et il faut importer `org.junit.After`. De nouveau, elle doit être `public` et `void`.
- Il est aussi possible d'écrire une méthode qui s'exécutera une seule fois avant la première méthode de test (ou après la dernière méthode de test) de la classe. Il faut mettre l'annotation **`@BeforeClass`** (`@AfterClass`) avant cette méthode et importer `org.junit.BeforeClass` (`org.junit.AfterClass`). Cette méthode doit être `public`, **`static`** et `void`.

Une suite de test

Parfois, on veut exécuter plusieurs classes de tests en même temps. Si on écrit une classe `TestClient` et une classe `TestCommande` et qu'on désire les exécuter en même temps, il faut écrire une autre classe comme suit :

```
package tests;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ TestClient.class, TestCommande.class })
/*
 * Dans les value, on met toutes les classes qu'on souhaite exécuter. Les mots
 * value= ne sont pas obligatoires.
 */
public class AllTests {
	// ne rien écrire dans cette classe.
}
```

Les méthodes utilisables : les assert

JUnit offre toute une série de méthode qu'on peut utiliser quand on écrit des tests. Vous en avez déjà vu quelques-unes. Voici la liste des principales méthodes existantes :

- `assertEquals(type attendu, type calculé);`
  `assertEquals(String msg, type attendu, type calculé);`

  où type peut être n'importe quoi.

  Si le type est un type primitif (`byte`, `char`, `int`, …), ces méthodes vérifient que la valeur calculée est bien égale à la valeur attendue. Sinon, ces méthodes utilisent la méthode

`equals` pour vérifier qu'il s'agit du même « `Object` ». Pour `double` ou `float`, ces méthodes sont « deprecated » : utilisez plutôt les deux suivantes.

- `assertEquals(type attendu, type calculé, type delta);`
  `assertEquals(String msg, type attendu,type calculé, type delta);`

  où type peut être `float` ou `double`.

  Ces méthodes vérifient que la valeur attendue est la même que la valeur calculée avec une tolérance de delta.

- `assertSame(Object attendu, Object calculé);`
  `assertSame(String msg, Object attendu, Object calculé);`

  Ces méthodes vérifient que l'objet attendu et l'objet calculé ont la même adresse (**==**)

- `assertNotSame(Object attendu, Object calculé);`
  `assertNotSame(String msg, Object attendu, Object calculé);`

  Ces méthodes vérifient que l'objet attendu et l'objet calculé n'ont pas la même adresse (!=)

- `assertNull(Object o);`
- `assertNull(String msg, Object o);`

- `assertNotNull(Object o);`
- `assertNotNull(String msg, Object o);`

- `assertTrue(boolean condition);`
- `assertTrue(String msg, boolean condition);`

- `assertFalse(boolean condition);`
- `assertFalse(String msg, boolean condition);`

- `fail();`
- `fail(String msg);`

  L'exécution d'une de ces deux dernières méthodes provoque l'échec du test.

Remarque :

Vous avez pu remarquer que pour toutes les méthodes, vous pouvez éventuellement mettre un **message** expliquant la **raison** de l'échec (c'est à cela que sert le paramètre `String msg`). **Il est vivement conseillé de le faire !**

Bonnes pratiques pour l'écriture de tests

JUnit offre un outil facilitant l'écriture mais il n'offre pas de recettes pour écrire de bons tests. Quand on écrit des tests, il y a certaines règles à respecter le plus possible afin d'améliorer la qualité des tests :

- ☐ écrire une classe de test par classe testée;
- ☐ écrire une méthode de test pour tester une seule chose. On ne fait pas plusieurs tests dans une méthode !
- ☐ choisir des noms significatifs pour les méthodes de test ;
- ☐ expliquer la raison d'un échec à l'aide d'un message ;
- ☐ tester tout ce qui peut échouer ;
- ☐ tester que tout ce qui doit être mis à jour par une méthode l'a vraiment été ;
- ☐ tester tous les cas où une exception doit être lancée. Pour cela, utiliser toujours l'annotation `@Test(expected = …)`.
- ☐ ne pas « catcher » les exceptions qui ne doivent pas se produire mais les propager.
- ☐ mettre la classe de test dans le même package que la classe testée mais dans des répertoires différents (par exemple dans un autre projet contenant les même package que le projet à tester et le relier à ce dernier) !
- ☐ écrire les tests avant d'écrire le code ;
- ☐ Quand dans un test une exception doit se produire mais qu'on doit encore tester quelque chose d'autre après, on « catche » l'exception comme dans l'exemple suivant :

```
@Test
public void testPasDeModificationQuandLeConstructeurSePlante() {
	Client lec = new Client("Leconte");
	try {
		new Commande(null, lec);
		fail("il y aurait dû y avoir une exception");
	} catch (ArgumentInvalideException e) {
		assertTrue(true);
		// pour indiquer que c'est ce qu'on attend.
	}
	assertEquals(0, lec.nombreDeCommandes());
}
```

Exécution des tests

Lorsqu'on exécute une classe de tests JUnit, toutes les méthodes de test sont exécutées même si certains tests échouent. Lorsqu'un test échoue, JUnit le signale par :

- une **failure** si on est passé par un **fail()** ou si un « **assert** » a raté ;
- une **error** si une exception inattendue s'est produite (`NullPointerException`,…).

Ecrire des tests JUnit avec Eclipse

Avec Eclipse, si on veut écrire une classe de tests, il suffit de choisir « new JUnit Test Case ». Dans la fenêtre qui s'ouvre, il faut :

- sélectionner « new JUnit 4 Test » ;
- préciser le nom de la classe ;
- cocher les méthodes que vous souhaitez créer (setUp, tearDown,…);
- indiquer dans « classe under Test » la classe que vous allez tester dans votre classe de test

Ensuite, il faut cliquer sur « next ». Enfin, il reste à sélectionner les méthodes à tester et « Finish ».