Utilisation de JUnit

Qualité de développement

IUT 45 — Informatique

2021-2022

Résumé

Cette feuille vous initie à l'utilisation de la bibliothèque de référence pour les tests en Java : `JUnit`. Vous allez l'utiliser pour ajouter des tests au *week-end entre amis* que vous avez développé en POO, puis pour l'étendre, dans une démarche TDD.

Dans la première partie de la ressource «Qualité de développement», vous avez découvert la programmation orientée objet en Python, le *Test Driven Development (TDD)* et le travail à plusieurs à l'aide de git. Dans la seconde partie, nous allons découvrir comment tester du code écrit en Java, approfondir le modèle objet de Java avec les exceptions, tout en poursuivant l'utilisation de git.

Ce premier TP vous présente l'outil standard pour écrire des tests en Java : JUnit.

1 Installation de JUnit

Pour pouvoir écrire des tests en Java, nous avons besoin de la bibliothèque JUnit (et d'une bibliothèque auxiliaire appelée hamcrest). Les bibliothèques Java sont distribuées sous forme de fichiers `jar`. Quand elles sont sous licence libre, on peut généralement les récupérer depuis `https://search.maven.org/`.

Question 1

Récupérer sur Célène les fichiers `junit-4.13.2.jar` et `hamcrest-2.2.jar`.

Astuce

Vous pouvez indiquer à `code` que vous allez utiliser ces deux fichiers `.jar` en ouvrant la palette de commandes `Ctrl-Shift-P`, puis en tapant `configure classpath`. Dans la page que vous obtenez, dans la section **Referenced Libraries**, utiliser le bouton **Add** pour ajouter chacun des deux fichiers `.jar`.

À l'aide de ces deux fichiers, nous allons pouvoir écrire notre première classe de tests ! Vérifions par exemple nos connaissances en arithmétique…

Question 2

Recopier le fichier `TestsPetitsCalcul.java` suivant, qui contient vos premiers tests utilisant JUnit.

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

public class TestsPetitsCalculs {
    @Test
    public void testAddition() {
        assertEquals(2, 1 + 1);
    }

    @Test
    public void testMultiplication1() {
        assertEquals(72, 8 * 9);
    }

    @Test
    public void testMultiplication2() {
        assertEquals(3252, 32 * 52);
    }

    @Test
    public void testHexa() {
        assertEquals(1024, 0x10 * 0x10);
    }
}
```

Imports nécessaires pour JUnit

Chaque test est une méthode marquée `@Test`

La méthode `assertEquals` importée depuis `org.junit.Assert` permet de certifier l'égalité entre deux valeurs dans un test.

Ce fichier utilise les deux bibliothèques java `junit` et `hamcrest`. Pour le compiler et l'exécuter, il faut indiquer au compilateur java et à la jvm où trouver ces bibliothèques.

Question 3

Compiler le fichier à l'aide de la commande suivante

```
javac -cp .:junit-4.13.2.jar TestsPetitsCalculs.java
```

Commande pour compiler un programme java

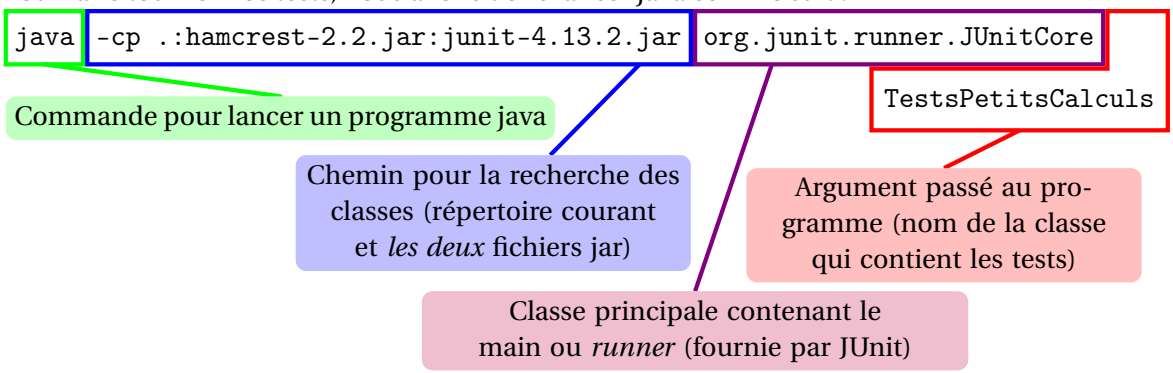
Chemin pour la recherche des classes (dans le répertoire courant et dans `junit-4.13.2.jar`)

Le fichier que l'on compile

Notre fichier contient une classe TestsPetitsCalculs. Les méthodes de cette classe qui correspondent à des tests portent l'annotation `@Test`. Comme vous pouvez le remarquer, cette classe ne contient pas de méthode `main`. Pour lancer les tests, nous allons donc utiliser une classe *fournie par junit* qui

contient un main. Cette classe est un *test runner*; elle s'appelle `org.junit.runner.JUnitCore`, et est contenue dans `junit-4.13.2`. Elle fait appel à des classes contenues dans `hamcrest-2.2.jar`. Sa fonction `main()` attend comme argument le nom de la classe contenant les tests.

Pour faire tourner nos tests, nous allons donc lancer java comme suit :



Question 4

Lancer les tests de la classe qui vous a été donnée. Repérer les messages d'erreur et corriger les tests qui ne passeraient pas.

À retenir

Les tests en java s'écrivent avec JUnit, dans une classe de Test. Cette classe doit se compiler avec les fichiers `.jar` de JUnit et hamcrest. On lance les tests en exécutant la classe `org.junit.runner.Runner` et en passant le nom de la classe de tests en paramètre.

À retenir

En JUnit, les tests sont les méthodes marquées `@Test` dans la classe de tests.

À retenir

Dans un test JUnit, on peut utiliser `assertEquals` pour affirmer que deux valeurs sont égales. Ainsi, `assertEquals(2+2, 4)` permet au test de passer, tandis que `assertEquals(2+2, 5)` provoque l'échec du test qui le contient.

2 Retour du week-end

Nous allons maintenant ajouter des tests sur un code existant pour tester sa qualité. Il nous faut pour celà un bout de code de taille raisonnable, avec lequel vous soyez un peu familièr·es et qui soit susceptible de recevoir quelques améliorations...

Question 5

Récupérer le code du TP3 de conception orientée objet (le week-end entre amis), en créer une nouvelle copie, versionnée par git. Créer un fichier .gitignore pour ignorer les fichiers `.class` et `.jar`. *Créer ensuite votre premier commit qui contiendra le code que vous aviez à la fin du TP de POO.*

2.1 Une classe de tests

Parfait, nous avons un code à tester et nous avons JUnit à notre disposition. Il y a même quelques tests… dans la méthode `main`, n'est-ce pas? Dans la suite, nous voudrions munir votre code d'une méthode `main` qui fasse des choses utiles pour un groupe de bons amis qui font les bons comptes de leur bon week-end. Il faut trouver un meilleur endroit pour mettre les tests. Nous allons utiliser à cet effet la solution standard **Junit**, vue ci-dessus.

Dans notre application Java, «une dépense» ou «toutes les données du week-end» sont représentées par des classes. De même, la *collection des tests* de l'application sera représentée par une classe. Il nous appartient d'écrire une classe de cas de de test. Comme nous l'avons vu ce-dessus, JUnit nous fournit alors le moyen de lancer tous les tests qu'elle contient, de collecter les résultats.

Question 6

Créer une classe de tests `TestsWeekEnd` qui utilise JUnit et qui importe vos classes. Dans un premier temps, cette classe sera vide. Vérifier que vous pouvez compiler et lancer cette classe.

Dans l'exemple de la partie 1, nous avons vu la fonction `assertEquals` pour vérifier que deux valeurs sont égales. D'autres assertions sont courantes en JUnit.

Question 7

Compléter à l'aide de la page `https://github.com/junit-team/junit4/wiki/Assertions` le tableau ci-dessous.

Assertion	Description
`assertEquals(a, b)`	Vérifie que `a` et `b` sont égaux.
`assertEquals(a, b, e)`	Vérifie que les *flottants* `a` et `b` sont égaux à `e` près ($\lvert a-b\rvert < e$).
`assertTrue(cond)`	Vérifie que la condition `cond` est fausse.
`assertSame(obj1, obj2)`	

Question 8

Transférer les tests existants dans le `main` de votre application «week-end entre amis» vers la classe TestsWeekEnd. Chaque test doit être transformé en une méthode séparée de la classe `TestsWeekEnd`.

Question 9

Vérifier —ou faire en sorte— que tous vos tests passent.

2.2 Des tests plus élégants

Voici une liste de propriétés que l'on veut maintenant tester sur votre code :

1. Si deux personnes dans le groupe (`Anais` et `Benjamin`) n'ont réglé aucune dépense, alors les valeurs de `week_end.avoirParPersonne(Anais)` et `week_end.avoirParPersonne(Benjamin)` sont égales;
2. si une personne du groupe n'a fait aucune dépense, son `avoirParPersonne` est plus petit que celui de toutes les autres personnes;
3. s'il n'y a qu'une personne, son `avoirParPersonne` sera nul;

4. la somme des `avoirParPersonne` de toutes les personnes du groupe est nulle ;
5. une personne qui n'a fait aucune dépense a un `avoirParPersonne` négatif.

Réutilisation de valeurs entre tests Avec tous ses tests, votre classe de tests contient probablement beaucoup de répétitions : on crée pour chaque test un week-end, des instances de personne et de dépenses. Certaines personnes et certaines dépenses reviennent dans plusieurs tests. Il est possible de ne les déclarer qu'une fois en ajoutant des attributs à la classe de tests. Ainsi, les valeurs de ces attributs sont accessibles depuis les méthodes marquées `@Test`. Pour que le *runner* donne la bonne valeur à ces attributs avant chaque test, il faut les initialiser dans une méthode marquée `@Before`.

Par exemple, dans le code ci-dessous, l'attribut `this.a` est mis à 3 avant chaque test.

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

public class TestsMethodeBefore {

    private int a;

    @Before
    public void initialisation() {
        this.a = 4;
    }

    @Test
    public void test_valeur_a() {
        assertEquals(a, 2 + 2);
    }

    @Test
    public void test_valeur_a_plus_un() {
        this.a += 1;
        assertEquals(a, 5);
    }

    @Test
    public void test_valeur_a_again() {
        assertEquals(a*a, 0x10);
    }
}
```

Question 10

Ajouter des attributs correspondant aux personnes qui reviennent dans plusieurs tests ; ajouter une méthode `initialisation_personnes` marquée «`@Before`» pour initialiser ces personnes avant chaque test.

Question 11

Ajouter de même une méthode `@Before` nommée `initialisation_depenses` pour initialiser les instances de la classe `Depense`.

3 Extensions de l'application de dépenses

Dans cette partie, vous trouverez deux extensions possibles pour le week-end entre amis. Elles vous sont données sous forme de backlog, à vous de les transformer en tests puis en implémentation, chacune dans une branche.

3.1 Un peu de justice sociale

Certains groupes d'amis peuvent trouver plus juste que chacun paye selon ses moyens plutôt que d'exiger la même contribution de personnes qui ont des ressources différentes. On leur propose que la *contribution* de chacun dépende de son revenu par la formule $c(p) = \frac{D \times r(p)}{R}$, où D est le total des dépenses du groupe, $r(p)$ est le revenu de la personne, et R est la somme des revenus des membres du groupe.

Question 12

Voici un *backlog* pour implémenter cette idée :

1. Les `Personnes` ont un attribut revenu (un flottant)
2. On peut obtenir ou modifier le revenu d'une `Personne`
3. On veut pouvoir obtenir le *revenu* moyen des personnes du groupe
4. Si deux personnes, Karl et Rosa, n'ont fait aucune dépense, mais que le revenu de Karl est de 24% supérieur à celui de Rosa, alors l'avoir de Karl sera égal à 1,24 fois celui de Rosa [a].
5. Si deux personnes, Friedrich et Louise, ont le même revenu, mais que Friedrich a fait une dépense de 100€ alors que Louise n'en a fait aucune, alors l'avoir de Friedrich sera supérieur de 100€ à celui de Louise.

a. et c'est juste : ces deux avoirs seront négatifs, donc Karl paiera plus que Rosa.

3.2 Activités à la carte

Certains groupes d'amis n'ont pas envie de rester collés ensemble tout le week-end. Ces groupes vont, à certains moments, faire des activités auxquelles tout le monde ne participera pas. À la fin du week-end, chaque personne devra payer ou recevoir sa part des dépenses communes à tout le monde, ainsi que sa part pour chacune des activités auxquelles elle a participé.

Question 13

Voici un *backlog* pour les activités à la carte.

1. Un week-end peut contenir des activités ;
2. Un week-end contient toujours une activité `"commun_tlm"` ;
3. Une dépense est associée à une activité, par défaut `"commun_tlm"` ;
4. On peut obtenir la dépense moyenne pour une activité ;
5. Quand on ajoute une troisième personne à une activité, la dépense moyenne par personne de cette activité baisse d'un tiers ;
6. Si Mahaut n'a pas participé à l'activité `"dîner de gala"`, ajouter une dépense dans cette activité ne change pas son avoir.

3.3 Fusion

Question 14

Réaliser la fusion de vos branches `justice_sociale` et `activites_a_la_carte`. Est-ce qu'il est suffisant de résoudre les conflits textuels pour que les tests passent ?