Conception d'un framework de test

Lorque l'on développe une application, il est essentiel de définir des programmes de test. Leur objectif est d'essayer de trouver des erreurs dans l'application, mais aussi d'être rejoués à chaque évolution de cette dernière afin de prévenir toute éventuelle régression.

Seulement, il est souvent fastidieux d'écrire les programmes de test en partant de rien. C'est justement là l'intérêt des frameworks. Un « framework » est un ensemble de classes éventuellement incomplètes qui peuvent être réutilisées et adaptées, ou bien servant de base pour l'application.

Dans ce TP, nous allons réaliser une application qui se résumera à une classe, puis nous re-créerons un mini framework de test ressemblant à JUnit3.

Exercice 1

Une monnaie représente une certaine valeur d'argent dans une devise particulière. On considèrera que la valeur est représentée par un entier, et la devise par une chaîne de caractères. Il est possible d'ajouter à la valeur d'une monnaie la valeur d'une autre monnaie. Il est également possible de retrancher à une monnaie la valeur d'une autre monnaie. Ces deux opérations n'ont de sens que si les deux monnaies ont la même devise. Dans le cas contraire, elles lèvent une exception.

Réaliser les classes `Money` et `InvalidCurrencyException`.

Exercice 2

Nous nous intéressons ici à la notion de test élémentaire. Pour pouvoir lancer automatiquement des tests contre une classe, il faut non seulement spécifier le format que devront prendre les tests, mais aussi la manière d'évaluer si un test à réussi ou échoué.

Bien que ça soit le testeur (le programmeur qui écrit des programmes de test) qui explique ce qui doit être fait lorsque le test est lancé, en général, il voudra comparer le résultat d'un calcul à un résultat attendu. Par exemple, vérifier que le pgcd de 10 et 6 est bien 2. Pour se faire une méthode assertTrue doit être définie. Sa signature est : `public static void assertTrue(boolean expression)`.

Cette méthode vérifie que l'expression est effectivement vraie. Dans notre exemple, le testeur pourrait écrire `assertTrue(2 == pgcd(10, 6))`.

Un test peut échouer pour deux raisons :

- Soit parce qu'une vérification échoue (assertTrue sur une expression booléenne fausse). Cette erreur est qualifiée d'erreur fonctionnelle.
- Soit parce qu'une erreur de programmation s'est produite (indice non valide pour un tableau, méthode appliquée sur une poignée nulle, division par zéro, ...)

Réaliser les classes suivantes permettant de mettre en place le système de vérification :

- Une classe `Failure` représentant une erreur fonctionnelle d'un test.
- Une classe utilitaire (abstract, final, ne comportant que des méthodes statiques) nommée `Assert` implémentant la méthode `assertTrue`.

Exercice 3

Il s'agit maintenant d'écrire un programme de test comportant deux tests de la classe `Money`.

Ces deux tests construisent deux monnaies, `m1` qui correspond à `5 Euro` et `m2` à `7 Euro`. Le premier ajoute `m2` à `m1` et vérifie que la valeur de `m1` est bien `12 Euro`. Le second retranche `m2` à `m1` et vérifie que la valeur de `m1` est bien `-2 Euro`.

Voici une description des aspects à prendre en compte pour automatiser les tests :

1. Une suite de test est une classe quelconque qui doit être publique.
2. Un test élémentaire est une méthode publique de cette classe, dont le nom commence par `test`, et qui ne possède aucun paramètre.

3. Lancer une suite de tests consiste à lancer chaque test à la suite.
4. Il est intéressant d'avoir le résultat des tests, c'est-à-dire des informations qui indiquent :
   - le nombre total de tests lancés ;
   - le nombre de tests qui ont échoué ;
   - les tests qui ont échoué, avec la cause de leur échec. À la fin de la suite de tests, les résultats seront affichés dans la console.

5. Peu importe la cause de l'échec, une suite de test ne peut pas échouer et s'arrêter avant d'avoir effectué tous les tests.
6. Étant donné que chaque test risque de modifier les données, il est nécessaire de les initialiser avant chaque nouveau test. Pour cela, on définit dans notre suite de test une méthode publique `setUp()` dont le but sera d'initialiser les données. Cette méthode sera appelée avant chaque test.
7. De manière symétrique, lorsque le test est terminé, il peut être nécessaire de libérer les ressources allouées lors de l'initialisation. On définit pour cela une méthode `tearDown()`. Tout comme `setUp()`, cette méthode sera appelée après chaque test.

Avec ces informations, réaliser la classe de test de la classe `Money`. On pourra l'appeler `MoneyTest`.

Exercice 4

À partir des information précédentes, réaliser une classe `Launcher`, chargée d'identifier toutes les méthodes de test, de construire une suite de tests, lancer les tests et enfin afficher les résultats.

Bien entendu, pour que ceci fonctionne, la classe doit pouvoir découvrir dans la classe donnée en argument toutes les méthodes de tests, la méthode `setUp()` et la méthode `tearDown()`.

Pour faire cela, nous nous reposerons sur l'introspection de Java.

Réfléchir aussi au point suivants :

- Faut-il préférer utiliser `getMethods` ou bien `getDeclaredMethods` ?
- La classe de test peut ne pas définir les méthodes `setUp()` et `tearDown()`. Que faut-il faire dans ce cas ?

Ne pas hésiter à écrire plusieurs autres classes de test pour tester le lanceur.
Bien entendu, vous pourrez également faire en sorte que votre framework se teste lui-même ! (cela revient à écrire une classe de test contre les entités du framework, et de les lancer grâce au lanceur)

Il est conseillé d'avancer petit à petit en compilant et en exécutant régulièrement et en vérifiant que vous obtenez bien les résultats attendus grâce à des traces placées dans le code.

Pour vous aider, voici les différentes étapes à réaliser pour lancer une suite de tests :

- Récupérer la classe
- Récupérer les méthodes `setUp()` et `tearDown()`
- Instancier l'objet qui sera le récepteur des tests
- Récupérer et exécuter les méthodes de test
- Afficher les résultats dans la console (statistiques et erreurs pour chaque test)