guillaume.dupont@enseeiht.fr

JUnit : un framework de test

Lorsque l'on développe une application (ou même une seule classe, voire un seul sous-programme), il est essentiel de définir des programmes de test. Leur objectif est d'essayer de trouver des erreurs dans l'application. Si les programmes de test sont suffisamment complets et qu'ils réussissent tous, alors on pourra avoir confiance dans l'application. Si certains échouent, il faut alors corriger l'application (ou les programmes de test si ce sont eux qui contiennent l'erreur) et rejouer l'ensemble des programmes de test (tests dits de non régression) pour vérifier que les modifications apportées n'ont pas introduit de nouvelles erreurs. Les tests sont également rejoués quand l'application évolue.

Malheureusement, il est souvent fastidieux d'écrire les programmes de test. L'objectif des exercices qui suivent est d'étudier comment est conçu et fonctionne le « framework » de test JUnit (http ://junit.org). Un « framework » est un ensemble de classes éventuellement incomplètes qui peuvent être réutilisées et adaptées.

Dans un premier temps, nous définissons notre application qui se résume ici à une classe Monnaie (exercice 1). Nous écrivons ensuite quelques programmes de test « traditionnels » pour tester cette classe (exercice 2). Ceci nous permet de voir les limites de l'approche « traditionnelle » et d'introduire l'approche JUnit (exercice 3). JUnit permet de créer des tests élémentaires (exercice 4) qui peuvent être organisés en suites de tests (exercice 5).

Exercice 1 : La classe Monnaie

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 même devise. Dans le cas contraire, elles lèvent une exception `DeviseInvalideException`.

1.1 Dessiner le diagramme de classe qui fait apparaître la classe Monnaie.

1.2 Écrire en Java l'exception `DeviseInvalideException` sous contrôle du compilateur.

1.3 Écrire en Java la classe Monnaie.

Exercice 2 : Test de la classe Monnaie

Il s'agit maintenant d'écrire les programmes de test de la classe Monnaie. Notons que ces programmes devraient être écrits en même temps que la classe, voire avant ! Ici nous nous limiterons à deux programmes de test.

Ces deux programmes construisent deux monnaies, m_1 qui correspond à 5 euros et m_2 à 7 euros. Le premier programme, `TestMonnaie1`, ajoute m_2 à m_1 et affiche la valeur de m_1. L'utilisateur pourra contrôler qu'elle vaut 12. Le deuxième programme, `TestMonnaie2`, retranche m_2 à m_1 et affiche la valeur de m_1 qui devrait être -2.

Écrire seulement le programme `TestMonnaie1` car `TestMonnaie2` en est très proche.

Exercice 3 : Conception du framework de test

Les tests de l'exercice 2 reposent sur l'utilisateur du programme de test qui doit vérifier que le programme affiche bien les résultats attendus. Malheureusement, l'utilisateur n'est pas fiable pour détecter si un test a réussi ou non. L'idée est donc d'automatiser les tests mais aussi leur évaluation pour savoir s'ils ont réussi ou échoué.

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

1. Un test élémentaire décrit un seul test.
2. Un test élémentaire peut être lancé.
3. Ce qui se passe lorsqu'un test est lancé dépend du système à tester et du test envisagé.
4. Faire un seul test n'est pas suffisant. Il faut généralement plusieurs tests pour tester une méthode. Et bien sûr, il faut tester toutes les méthodes de toutes les classes de l'application. En conséquence, ces tests élémentaires sont regroupés sous la forme de suites de tests.
5. Une suite de tests peut être lancée.
6. Lancer une suite de tests consiste à lancer chaque test de la suite.
7. Toujours dans un souci de structuration, une suite de tests peut contenir des suites de tests.
8. 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 ;
9. Chaque test élémentaire lancé doit mettre à jour ces statistiques en fonction de son résultat.

3.1 Pour chaque paragraphe numéroté de la description du framework de test ci-dessus, indiquer en français, en utilisant le vocabulaire UML, ce qui peut en être déduit concernant le diagramme de classe.

3.2 Proposer un diagramme de classe détaillé qui décrit un tel framework de test.

Exercice 4 : Le test élémentaire

Intéressons nous maintenant plus en détail à un test élémentaire. La description UML de la classe TestÉlémentaire est donnée figure 1.

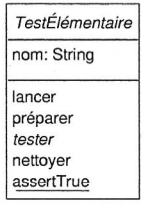


FIGURE 1 – La classe TestÉlémentaire

C'est le testeur (le programmeur qui écrit des programmes de test) qui explique ce qui doit être fait lorsque le test est lancé.

En général, le testeur 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` est définie. Sa signature est :

```
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))`.

Le programmeur peut souhaiter réaliser plusieurs tests sur les mêmes données, par exemple $10 + 6 = 16$, $10 * 6 = 60$... Comme un test pourrait modifier les données, il est nécessaire de les initialiser avant chaque nouveau test. De manière symétrique, lorsque le test est terminé, il peut être nécessaire de libérer les ressources allouées lors de l'initialisation des données. Ainsi lancer un test (`lancer`) consiste à initialiser les données (`préparer`), lancer le test effectif (`tester`) et, enfin, libérer les ressources (`nettoyer`).

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...).

Aussi, les résultats de tests doivent comptabiliser et conserver d'une part les tests en échec (erreur fonctionnelle) et les tests en erreur (erreur de programmation).

4.1 *Utilisation de la classe TestÉlémentaire.* Maintenant que la classe TestÉlémentaire existe, nous souhaitons réorganiser les deux programmes de test de la classe Monnaie (exercice 2). L'objectif est de minimiser le nombre d'instructions à écrire.

Dessiner le diagramme de classe correspondant à ces deux programmes de test et donner le code Java des méthodes dans des annotations UML.

4.2 *Compléter et comprendre la description UML.* Dans un premier temps, nous allons compléter la description UML de la classe `TestÉlémentaire` et comprendre les choix faits.

4.2.1 Indiquer, en le justifiant, le droit d'accès à mettre sur chaque élément de la classe.

4.2.2 Seule la méthode `tester` est déclarée retardée. Les méthodes `lancer`, `préparer` et `nettoyer` ne le sont pas. Expliquer ce qui justifie ces choix. On commencera par expliquer l'intérêt de définir une méthode retardée.

4.2.3 Indiquer s'il est possible de créer une instance de la classe TestÉlémentaire. La réponse doit être justifiée.

4.2.4 Peut-on définir un constructeur sur cette classe ? La réponse doit être justifiée et, le cas échéant, la signature du constructeur donnée.

4.3 *La méthode `assertTrue(boolean)`.* La méthode `assertTrue(boolean)` est utilisée par le testeur pour vérifier une expression booléenne. Si cette expression est fausse, la vérification a échoué et une exception `Échec` est levée. Le test sera comptabilisé en échec.

4.3.1 Expliquer pourquoi il est préférable d'utiliser une exception pour signaler l'échec plutôt qu'un code de retour de la méthode `assertTrue` ?

4.3.2 Expliquer pourquoi la méthode `assertTrue(boolean)` est une méthode de classe.

4.3.3 Étant donné l'objectif de l'exception Échec, indiquer comment la définir en Java et écrire le code correspondant.

4.3.4 Écrire le code de la méthode `assertTrue(boolean)`.

4.4 *Programmation de la classe.* Intéressons nous d'abord à la méthode `lancer` qui est chargée de lancer le test et de le comptabiliser.

4.4.1 Comment une erreur fonctionnelle est-elle détectée ? Et une erreur de programmation ?

4.4.2 Écrire la méthode `lancer`.

4.4.3 Écrire la classe TestÉlémentaire. On se contentera de mettre des ellipses (...) pour le code des méthodes qui ont déjà été définies.

Exercice 5 : Suite de tests

Intéressons nous maintenant à la notion de suite de tests.

5.1 Indiquer si la conception détaillée de la classe `TestÉlémentaire` (exercice 4) nécessite de modifier le diagramme de classe du framework de test (exercice 3). Donner les éventuelles modifications.

5.2 Une suite de tests est un ensemble de tests. Expliquer les différentes possibilités en Java de représenter cette relation avec les avantages et inconvénients associés.

Conclusion : Il est important de construire des jeux de test pour vos classes. JUnit peut vous aider en apportant encore plus de facilités que ce qui a été présenté dans ce sujet...

La devise d'un développeur pourrait alors être : « Si une fonctionalité d'un programme n'a pas de tests automatisés, nous considérons qu'elle ne fonctionne pas ».

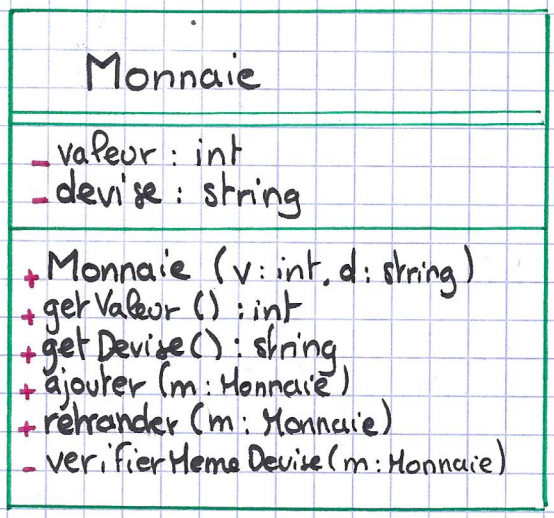
GLS
TD1.1

JUnit : un framework de test

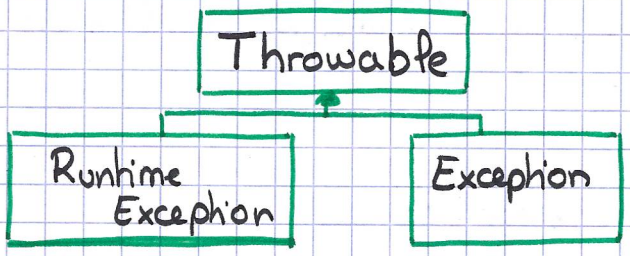
Exercice 1 : La classe Monnaie

Rappel : - privé, + public, # protected

1.1



1.2




```
public class DeviseInvalideException extends Exception {
    rajouter constructeur de la classe mère
}
```

1.3

```
public class Monnaie {

    private int valeur ;
    private String devise;

    public Monnaie ( int valeur, String devise ) {
        this.valeur = valeur;
        this.devise = devise;
    }

    public String getDevise () {
        return this.devise;
    }

    public int getValeur () {
        return this.valeur;
    }

    public void ajouter (Monnaie m) throws DeviseInvalideException {
        verifierMemeDevise ( m );
        this.valeur += m.valeur;
    }

    ...
```

```
    private void verifierMemeDevise(Monnaie m) throws DIE {
        if(!this.devise.equals(m.devise);
            throw new DIE;
    }
}
```

Exercice 2 : Test de la classe monnaie

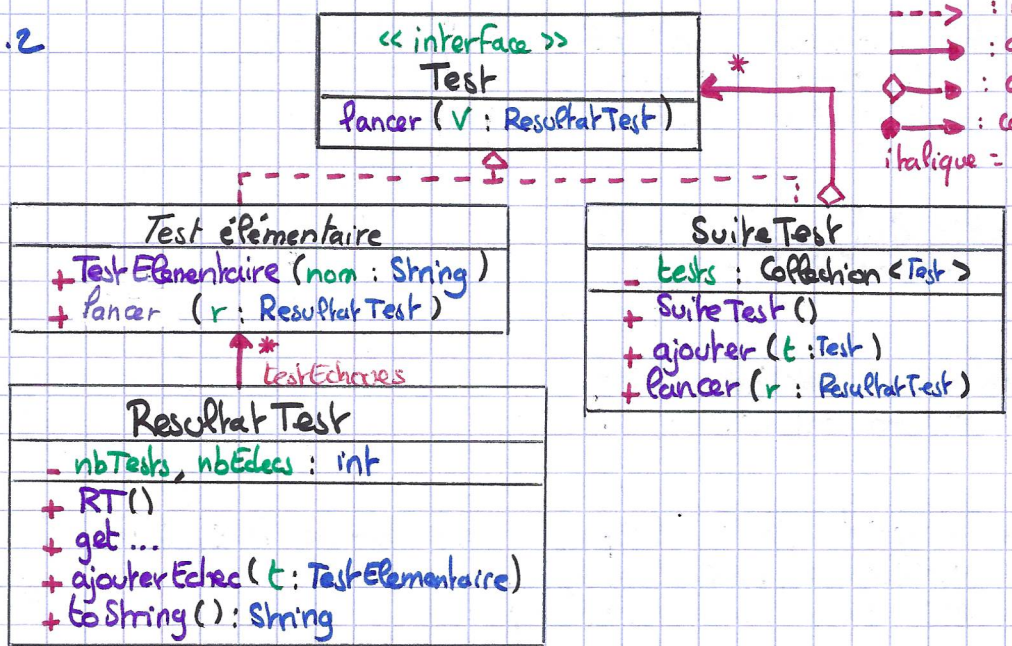
```
public class TestMonnaie1 {

    public static void main(String args[]) {

        Monnaie m1 = new Monnaie(5,"€");
        Monnaie m2 = new Monnaie(7,"€");

        m1.ajouter(m2);

        System.out.println("m1 = "+m1.getValeur()+"m1.getDevise()

    }
}
```

Exercice 3 : Conception de Framework de test

3.1

1. TestElementaire
2. lancer()
3. abstract
4. SuiteTest *
5. ↳ lancer() *
6. Test
7.
8. ResultatTest
9.

3.2



GLS
TD1.2

Exercice 4 : Le test élémentaire

Dans le diagramme UML de TestElementaire, lancer ne doit pas être abstrait
On rajoute 3 méthodes, qui decomposent lancer()

```
# preparer()
# nettoyer()
# tester()
```

On rajoute un attribut à la classe ResultatTest : - nbErreurs

```
public st void assertTrue(bool t){
        if (! t)
        throw TestEchoue()
}
```

Pour ResultatTest on rajoute 2 méthodes

```
+ incTest()
+ ajouterErreur(t : TestElementaire)
```

Ecriture de lancer()

```
void lancer(rt : RT){
    try { rt.incTest();
        try { preparer();
              tester();
        } finally {
              nettoyer();
        }
    } catch (TestEchoue e){
        rt.ajouterEchec(this);
    } catch (Throwable e) {
        rt.ajouterErreur(this);
    }.
}
```