TP R2.03 – Les Tests Unitaires avec JUnit et les Contrats

Objectifs du TP (4 X 1h30)

- Savoir coder un bon test unitaire.
- Mettre en œuvre un test unitaire avec l'outil JUnit.
- Comprendre l'intérêt des contrats.

Codage et rendu

On suppose une arborescence de développement habituelle et obligatoire :

/TPTestJUnit				/TPMoney			
/	|	|	\	/	|	|	\
/ws	/src	/class	/lib	/ws	/src	/class	/lib

Les compilations et les exécutions doivent être lancées avec les commandes *javac* et *java* dans */ws* (version de java : **java11 maximum**). Votre *CLASSPATH* doit être déclaré et doit contenir les bons chemins de recherche.

Tous vos sources *java* (fichiers *.java* !) doivent être rassemblés dans une archive à votre nom (*Nom_Prénom.zip*). Ce **rendu individuel** doit être déposé sur Moodle R2.03 (https://moodle.univ-ubs.fr/course/view.php?id=7556) pour le dimanche 04/06 à 23h55 au + tard (**attention malus -2 points si retard**).

Test simple avec *JUnit* (TPTestJUnit)

Rappel

Lors du développement d'une application, à chaque fois qu'UNE classe est développée on lui associe UNE classe de test.

==> UNE classe développée = UNE classe de test <==

Par convention, si la classe développée s'appelle *MaClasse*, la classe de test s'appelle *TestMaClasse*. Cette classe de test contient nécessairement un lanceur car elle doit pouvoir s'exécuter.

Test sans JUnit

Pour créer un test unitaire, nous allons partir d'une classe très simple qui nous sert d'exemple : une classe *Operations* avec une méthode qui additionne deux entiers.

```
public class Operations {

    public int additionne ( int premier, int second ) {
        int ret = premier + second ;
        return ret ;
    }
}
```

La classe de test associée est la suivante :

```
public class TestOperations {

   public static void main ( String[] args ) {
      testAdditionne ();
   }

   private static void testAdditionne() {
      System.out.println ( "Test de la méthode additionne – cas normal" );
      Operations op = new Operations();
      int res = op.additionne( 1, 2 );
      if ( res == 3 ) System.out.println ( "Test réussi" );
      else System.out.println ( "Echec du test" );
   }
}
```

La même chose avec JUnit

JUnit est un outil, avec sa propre syntaxe, utilisé pour le développement et l'exécution de tests unitaires semi-automatisés.

Semi-automatisés car *JUnit* ne propose qu'une coquille vide (mais facile à comprendre et à utiliser !) qui devra être remplie « intelligemment » par le développeur. En d'autres mots, *JUnit* ne jugera pas de la pertinence d'un cas de test, il se contentera juste de l'exécuter (même s'il n'y a rien dedans !). C'est donc à vous d'écrire le bon code de test à l'intérieur de cette coquille vide.

Voici ce qu'il faudrait écrire pour réaliser le simple test de la méthode additionne ci-dessus (**ceci est à mettre en œuvre obligatoirement sur machine, espace TPTestJUnit**) :

Dans votre espace de développement (répertoires */ws*, */class*, */src*, */lib*), on placera les archives *junit-4.11.jar* et *hamcrest-core-1.3.jar* (*ArchiveJUnit* à récupérer sur Moodle R2.03) dans le répertoire */lib*. Modifiez votre CLASSPATH pour permettre l'accès à ces fichiers :

- sous Linux : modifiez dans votre fichier *.bashrc* la ligne ci-dessous :

  export CLASSPATH=../class:../lib/junit-4.11.jar:../lib/hamcrest-core-1.3.jar

- sous Windows : rajouter à votre variable d'environnement CLASSPATH les 2 chemins relatifs **..\lib\junit-4.11.jar** et **..\lib\hamcrest-core-1.3.jar**

Dans le répertoire */src*, créez la classe *Operations* et codez là.

Dans ce même répertoire, créez la classe de test unitaire *TestOperations* selon la syntaxe *JUnit*, ci-dessous :

```
import org.junit.*;                  // accès aux classes JUnit4
import org.junit.runner.*;           // permet l'exécution de la classe de test
import static org.junit.Assert.*;    // accès aux assertions

public class TestOperations {

   private Operations op;

   // méthode exécutée systématiquement AVANT chaque cas de test
   @Before
   public void instancier() {
      this.op = new Operations();
   }

   // Premier cas de test : test de la méthode additionne
   @Test
   public void testAdditionne() {
      System.out.println ( "Test de la méthode additionne – cas normal" );
      int res = this.op.additionne( 1, 2 );
      // automatisation : « res » doit contenir « 3 » sinon le test est en erreur
      // if/else s'écrit en 1 seule ligne avec assertEquals
      assertEquals ( "Echec du test", 3, res );
      System.out.println ( "Test réussi" );
   }

   // lanceur
   public static void main ( String args[]) {
      JUnitCore.main("TestOperations");
   }
}
```

Compiler les classes *Operations* et *TestOperations*.

Exécuter la classe de test *java TestOperations* et constater l'affichage du message « Test réussi ». Exécutez également la classe en provoquant volontairement une erreur de test et dans ce cas bien observer le lancement de l'exception *AssertionError* avec l'affichage du message « Echec du test ».

Faire de bons tests unitaires avec *JUnit* (TPTestJUnit)

Vous le savez déjà, dans une séquence de test il vaut mieux être trop bavard que pas assez. Voici un exemple d'informations intéressantes à afficher à l'écran :

```
@Test
public void testAdditionne() {
   System.out.println( "Test de la méthode additionne(…) – cas normal" );
   int res = this.op.additionne(1,2);
   System.out.println( "Résultat = " + res );
   etc…
}
```

Il est important de clairement signaler si le test s'est bien déroulé ou pas, ce qui se traduit par un message à la console du type :

Test réussi

ou en cas d'erreur

ECHEC du test

Cette syntaxe de sortie d'écran pour avantages :

- De permettre de rendre le résultat de test très visible en affichant un résultat compréhensible qui n'a pas besoin d'être interprété.
- D'être utilisable avec un très grand nombre de tests.
- De produire du texte qui est facilement exploitable par d'autres outils, par exemple, il suffit de rediriger la sortie écran vers un fichier et de recherche le mot clé ECHEC pour afficher toutes les lignes qui contiennent des erreurs.
- Avec *JUnit*, de permettre un test binaire (soit vrai, soit faux) qui s'écrit en une ligne avec une assertion (*AssertEquals* par exemple). Pour connaître la liste de toutes les assertions que *JUnit4.11* propose, consulter la *Javadoc* de la classe *Assert* (voir l'archive *junit-4.11-javadoc.jar* fournie dans *ArchiveJUnit* que vous avez déjà récupéré).

```
@Test
public void testAdditionne() {
      System.out.println( "Test de la méthode additionne(…) – cas normal" );
      int res = this.op.additionne(1,2);
      assertEquals ( "ECHEC du test : somme incorrecte", 3, res );
}
```

En cas d'erreur à l'exécution de *assertEquals*, *JUnit* lance une exception (*AssertionError*) avec un message à l'écran de la forme `java.lang.AssertionError: ECHEC du test : somme incorrecte`.

Il sera donc bien + lisible au niveau de la trace d'exécution de capturer l'exception *AssertionError* pour afficher le message d'erreur « ECHEC du test : somme incorrecte » et continuer une exécution normale sans arrêt brutal de l'application qui rendrait illisible les affichages du test à la console. Ceci doit être codé dans la méthode *testAdditionne* et bien entendu correctement testé.

Bien tester les cas d'erreurs (TPTestJUnit)

Bien tester les valeurs attendues dans des conditions d'utilisation normales est une bonne chose, mais tester aussi que les utilisations anormales sont bien gérées est tout aussi important.

Dans l'exemple qui va illustrer cette idée, nous allons tester une méthode qui calcule la racine carrée d'un nombre réel.

Codage

Avant d'écrire ce cas de test, **rajouter** dans la classe *Operations* la méthode *double calculeRacineCarree (double val)* qui fera usage de *static double sqrt (double a)* de la classe *java.lang.Math*. La méthode *sqrt(…)* renvoie NaN si la racine carrée ne peut pas être évaluée. Le résultat de *sqrt(…)* peut être passé en paramètre à la méthode *statique boolean isNaN (double v)* de la classe *Double* et si elle renvoie vrai la méthode *calculeRacineCarree* doit lancer l'exception *ArithmeticException* (hérite de *RuntimeException*). Cette exception devra être capturée dans un cas de test spécifique.

```
@Before
public void instancier() {
      this.op = new Operations();
}

@Test
public void testCalculeRacineCarree() {
      System.out.println ( "Test de la méthode calculeRacineCarree – cas normal" );
      double res = this.op.calculeRacineCarree ( 4 );
      // consulter la javaDoc de assertEquals dans le cas d'une comparaison de
      // deux nombre réels (double)
      assertEquals ( "ECHEC du test : racine carrée incorrecte", 2d, res, 0d );
      System.out.println ( "Test réussi" );
}
```

Le code ci-dessus n'est PAS suffisant car il ne tient pas compte du fait que la méthode *calculeRacineCarree* peut lancer une exception de type *ArithmeticException*.

```
@Test
public void testCalculeRacineCarree() {
      double test = 4;
      System.out.println( "Test de la méthode calculeRacineCarree – cas normal" );
      try {
         double res = this.op.calculeRacineCarree ( test );
         assertEquals ( "ECHEC du test : racine carrée incorrecte", 2d, res, 0d );
         System.out.println ( "Test réussi" );
      }
      catch ( ArithmeticException e ) {
         System.out.println ( "ECHEC du test : " + e.getMessage()" );
      }
}
```

Codez la méthode de test ci-dessus (cas normal) et essayez de bien comprendre son exécution. En particulier, choisissez une valeur de test de telle sorte que :

1. Le test soit réussi
2. Le test soit en échec avec le message « ECHEC du test : racine carrée incorrecte »
3. Le test soit en échec pour cause de lancement de l'exception *ArithmeticException*

Comme appris en algorithmique sur la première partie de l'année, vous pouvez écrire une méthode *private void testCasCalculeRacineCarree (double test, double attendu)* qui sera appelée autant de fois que nécessaire par le test unitaire *void testCalculeRacineCarree()*.

Vous pouvez noter que le test gère ici 2 cas :

- Si tout se passe bien, on doit récupérer la bonne valeur et si le calcul est erroné, on doit détecter qu'on ne récupère pas la bonne valeur (racine carrée incorrecte).
- Si le calcul se passe mal, on doit attraper l'exception *ArithmeticException* susceptible d'être jetée par la méthode *calculeRacineCarree*.

Exercice

Compléter le test de la méthode *calculeRacineCarree* en rajoutant clairement dans *testCalculeRacineCarree()* :

- le(s) cas d'erreurs
- le(s) cas limites

Privilégier les tests simples

Ne testez qu'une fonctionnalité à la fois.

La règle que l'on se donne, c'est de toujours repartir d'un objet propre et donc nouvellement créé pour chaque test et que chaque test doit minimiser les appels aux méthodes, afin de tester le moins de choses possibles dans un seul cas de test. Il vaut mieux avoir beaucoup de petits tests, facile à comprendre et avec des résultats clairs et lisibles plutôt que peu de tests qui contiennent beaucoup de codes et qui sont difficiles à comprendre (surtout lorsque ça ne se déroule pas comme prévu).

Il existe une syntaxe *JUnit* qui permet d'exécuter toujours la même séquence d'instructions avant même de commencer le test unitaire : c'est l'utilisation du tag *@Before* (avec en-dessous un nom de méthode qui rassemble le code d'initialisation). Dans l'exemple du test de la classe *Operations*, avant chaque test unitaire on effectue une instanciation d'un objet de type *Operations*.

TPMoney : comment bien tester avec JUnit

Ce TP demande le développement d'une classe *Money* et d'une classe *MoneyBag* qui utilise le type *Money*. Elles doivent être codées et le test unitaire se fera obligatoirement avec l'outil *JUnit4*.

La classe Money

Cette classe déclare 2 attributs :

- *int amount* : le montant (somme d'argent) qui peut être positif, négatif ou zéro
- *String currency* : la devise de la somme d'argent ("EUR", "USD", "CHF", "GBP", …)

Et les méthodes :

- *Money (int theAmnt, String theCurr)* : constructeur
- *getAmount()* et *getCurrency()* : accesseurs
- *Money add (Money theM) throws BadCurrencyException* : renvoie un nouvel objet *Money* dont le montant est le montant de l'objet courant + le montant de l'objet passé en paramètre. Ceci à condition que ces deux montants soient exprimés dans la même devise sinon une exception *BadCurrencyException* (code ci-dessous) est lancée.
- *boolean equals (Money otherM)* : renvoie vrai si et seulement si la monnaie passée en paramètre est strictement égale à l'objet courant (montant ET devise).
- *String toString()* : renvoie une chaîne de caractères contenant le montant et la devise de l'objet *Money*.

Code de la classe *BadCurrencyException* :

```
class BadCurrencyException extends Exception {
   public BadCurrencyException ( String msg ) {
      super (msg);
   }
}
```

Créez cette classe et testez-là (*TestMoney*) avec *JUnit4* en respectant bien les consignes présentées aux paragraphes précédents et en traitant les cas normaux, les cas d'erreurs et les cas limites.

La classe MoneyBag

La classe *MoneyBag* prend en charge des devises multiples qui sont stockées dans un tableau. On considère que chaque case du tableau correspond à une somme d'argent exprimée dans une devise (EUR, USD, CHF, …) nécessairement différente de toutes les autres cases du tableau (qui contiennent aussi des montants mais exprimés nécessairement dans une autre devise). Le seul attribut de la classe est donc un tableau de *Money* (*ArrayList<Money> monies*) où chaque case du tableau contient une somme d'argent dans une même devise.

Exemple :

- Case 0 : 118 en devise « EUR »
- Case 1 : 23 en devise « USD »
- Case 2 : 213 en devise « GBP »
- etc.

Les méthodes sont :

- *public MoneyBag()* : constructeur qui crée le tableau *monies* qui est vide au départ.
- *public MoneyBag (Money[] bag)* : constructeur qui crée le tableau des devises, *monies* et le remplit à partir des montants contenus dans le tableau *bag* passé en paramètre. Attention, utiliser la méthode *appendMoney* ci-après pour ajouter les montants.

- *public void appendMoney (Money theM)* : méthode qui ajoute le montant exprimé dans une devise (paramètre *theM*) dans le tableau monies. Attention, l'ajout ne peut pas se faire n'importe comment :
  - si la devise n'existe pas encore dans le tableau, ajouter le nouveau montant à la fin du tableau,
  - si la devise existe déjà dans le tableau, modifier le montant correspondant à cette devise en ajoutant le montant passé en paramètre (montant éventuellement négatif). Si le résultat de cette addition donne zéro alors il faut supprimer cette devise du tableau.
- *public void theSame (MoneyBag otherBag) throws NotTheSameException* : lance une exception *NotTheSameException* si et seulement si l'objet *otherBag* passé en paramètre n'est pas strictement égale à l'objet courant. Avant de coder, il faut donc réfléchir à la notion d'égalité entre 2 objets *MoneyBag*. Ecrire la classe *NotTheSameException* de la même façon que la classe *BadCurrencyException* ci-dessus.
- *public String toString()* : renvoie une chaîne de caractères contenant l'ensemble des données du tableau *monies*.

Créez cette classe et testez-là (*TestMoneyBag*) avec *JUnit4* en traitant les cas normaux, les cas d'erreurs et les cas limites.

Les classes Money et MoneyBag version2

Modifiez dans la classe *Money* la méthode *add (Money theM)* de telle sorte qu'il soit possible d'ajouter un montant qui ne soit pas de la même devise (et donc l'exception *BadCurrencyException* disparaît). Dans ce cas, on va considérer qu'un nouveau *MoneyBag* est créé et retourné. Il sera initialisé avec le montant courant ET le montant dans la nouvelle devise (*theM*).

Dès lors, la méthode *add(…)* de la classe *Money* retourne soit un type *Money* (même devise), soit un type *MoneyBag* (devises différentes). Ce type générique va s'appeler *IMoney* qui sera une interface de la forme suivante :

```
public interface IMoney {
    public abstract IMoney add ( Money m );
}
```

Pour être de type *IMoney*, les classes *Money* et *MoneyBag* doivent implémenter l'interface, ce qui leur impose de redéfinir la méthode abstraite *IMoney add (Money m)*. Cette méthode abstraite est précisément la signature de la méthode *add(…)* de la classe *Money* que l'on doit modifier. Par contre, la redéfinition dans la classe *MoneyBag* se limitera simplement, pour l'instant, au lancement d'une exception de type *UnsupportedOperationException*. En effet, la classe *MoneyBag* n'a pas d'utilité de cette méthode dans une première étape.

Marche à suivre :

1. Coder l'interface *IMoney*.
2. Modifier le code de la classe *Money* pour implémenter l'interface et redéfinir la méthode *add(…)* selon la spécification expliquée ci-dessus. Cette méthode ne lance plus d'exception.
3. Modifier le code de la classe *MoneyBag* pour implémenter l'interface et redéfinir la méthode *add(…)* selon la spécification expliquée ci-dessus.
4. Faire une compilation croisée de *Money* et *MoneyBag* (javac –d ../class ../src/Money.java ../src/MoneyBag.java) et résoudre les problèmes de compilation (notamment au niveau des transtypages).
5. Modifier les tests unitaires de *Money* et *MoneyBag* (cas normaux, cas limites, cas d'erreurs) avec *JUnit4* comme d'habitude.

Pour terminer, redéfinir maintenant la méthode *IMoney add (Money m)* dans la classe *MoneyBag* et donc donnez par vous-même un sens à cette méthode pour cette classe. Testez cette méthode.

Les contrats

Les contrats servent, en très résumé, à mieux garantir qu'une classe développée assure correctement les services qui lui sont demandés.

Il existe 3 types de contrat :

1. la pré-condition : condition à vérifier AVANT l'exécution d'une méthode publique. Consiste, en pratique en *Java*, à tester en tout début de méthode la validité des paramètres passés à celle-ci et à lancer une exception si la valeur de ces paramètres est incorrecte.
2. la post-condition : condition à vérifier APRES l'exécution d'une méthode publique ou privée pour contrôler l'accomplissement correct du travail demandé à la méthode. Consiste, en pratique en *Java*, à tester avec une (ou +sieurs) assertion(s) en fin de méthode, que l'objet modifié se trouve bien dans l'état demandé.
3. l'invariant : condition à vérifier à tout moment pour s'assurer que l'objet reste dans un état cohérent « tout au long de sa vie ». Consiste, en pratique en *Java*, à tester avec une méthode privée *private boolean invariant()* la validité de chacun des attributs de l'objet. L'appel de la méthode *invariant()* doit se faire dès la création de l'objet et en fin d'exécution de chaque méthode de type modificateur. En pratique il s'agit d'écrire une assertion : *assert this.invariant : "Invariant violé !"*.

Première mise en œuvre des contrats dans la classe *Money*

Mise en œuvre de quelques premiers contrats dans la classe *Money* (version2 déjà développée) :

- pré-condition : écrire en tout début de constructeur des pré-conditions (avec lancement d'exceptions) qui vérifient la validité des paramètres passés au constructeur.
- post-condition : écrire en fin de méthode *add* une post-condition (exprimée avec une assertion) qui vérifie que l'addition c'est correctement déroulée.
- invariant : écrire une méthode *private boolean invariant()* qui vérifie la validité des attributs (*amount* et *currency*) de l'objet courant. Renvoie *vrai* si les attributs sont valides, sinon *faux*. Cet invariant sera appelé à la fin de chaque méthode de type « modificateur » avec l'assertion suivante : *assert this.invariant : "Invariant violé !"*.

Dans la classe de test de *Money* (*TestMoney*) vous devez rajouter le test des cas d'erreurs pour le test du constructeur pour mettre en évidence la violation de la *pré-condition*.

Pour les 2 autres contrats (*post-condition* et *invariant*) vous n'avez rien à tester en particulier (sauf à détecter éventuellement que votre code est mal écrit) MAIS vous devez permettre à l'interpréteur *Java* de déclencher les assertions en exécutant votre classe. La commande devient alors *java –ea TestMoney* avec l'option *–ea* qui obligera les assertions à se déclencher si nécessaire.

Mise en œuvre des contrats : la suite

- Pour la classe *Money* (version2) : l'invariant, des pré-conditions (constructeur), une post-condition sont déjà écrits ci-dessus. Rajoutez le maximum de contrats dans cette classe et complétez si nécessaire le code de la classe de test (*TestMoney*).

- Pour la classe *MoneyBag* (version2) : écrire le maximum de contrats (invariant, pré-conditions, post-conditions) et complétez le code de la classe de test (*TestMoneyBag*) pour rajouter le test de cas d'erreurs supplémentaires qui viennent s'ajouter à cause de la présence maintenant de pré-conditions dans la classe *MoneyBag*. Pour déclencher les post-conditions et l'invariant, n'oubliez pas que l'exécution se lance avec la commande *java –ea TestMoneyBag*.