TP 1 – Tests Junit sur fonctions statiques (sans objets)
Algorithmique – Programmation FIP (ING39)

P. Courtieu

Attention l'expérience montre que ce cours/tutoriel est plus difficile qu'il n'y paraît. La principale difficulté vient probablement du fait qu'un *test* est un morceau de code qui sert à vérifier le fonctionnement (dans un certain cas) *d'un autre morceau de code*. Il est dans votre intérêt de *bien lire les explications ci-dessous* et de poser des questions immédiatement si vous ne comprenez pas.

Exercice 1 : premiers tests (`decaleDroite`)

Récupérez sur le site du cours (rubrique « énoncés des Tp ») l'archive (`.tgz`) contenant le projet associé au Tp1. Décompressez le dans votre compte linux (par exemple dans le répertoire `~/workspaceEclipse/`) puis ouvrez le dans Eclipse. Il contient la classe `ShiftTableau`. Ce sera notre **classe à tester**. **Avant de continuer, regardons son contenu :**

- une (seule) méthode statique : `decaleDroite`. C'est notre seule méthode à tester.
- pas de méthode main : nos tests se feront avec Junit, jamais (ou presque) avec main.
- un **commentaire au format javadoc** pour `decaleDroite`. C'est le **contrat** de la méthode : il spécifie le comportement attendu après son invocation.
- un **prototype de code** pour `decaleDroite` : ce n'est pas le véritable code, juste de quoi permettre à cette méthode de compiler. En particulier, ce prototype n'est pas le code à tester.
- et le « vrai » code à tester ? Plus tard, nous vous en proposerons deux versions : une au fonctionnement conforme au contrat ; une autre bugée. C'est sur ces codes-là que vos tests vont s'exécuter.

Rappel sur les principe des tests sur méthodes statiques

La première chose à savoir sur un test en programmation est qu'on teste du code. Si on teste une méthode, il faut l'invoquer en lui donnant un *argument concret*. Si la méthode a plusieurs comportements possibles, il faudra tous les tester avec au moins autant d'invocations d'arguments concrets correspondant à tous ces cas.

Un autre principe d'un test est de savoir à quoi s'attendre afin de déterminer si le test a réussi ou s'il a échoué. Mais, comment savoir à quoi s'attendre quelque soit l'invocation ? Il faut disposer d'un ***contrat*** qui spécifie tous les comportements de la méthode, précis (pas de bla-bla) et complet (tous les cas sont énumérés).

Concrètement, que veut dire tester une méthode `f` qui prend un argument x et retourne un résultat r ? Cela veut dire : (1) je dois invoquer le code de f avec un argument concret ; (2) je suis capable de dire si le résultat obtenu par cet appel correspond bien à ce que la méthode doit réaliser dans ce cas précis ; (3) *je vais écrire du code pour effectuer (automatiser) cette vérification elle-même* ! Si le comportement de l'invocation est celui qu'on attendait, alors le test réussit, sinon, il échoue. Quelques exemples :

- si l'appel `decaleDroite({1,2})` retourne le tableau `{1}` je peux à coup sûr enregistrer ce cas de test comme un « test échoué ».
- Il en va de même si cette invocation se termine par la lévée d'une exception.
- si l'appel `decaleDroite(new int[0])` se termine par l'exception `IllegalArgumentException` alors je peux enregistrer ce cas comme un « test réussi ».
- même chose si `decaleDroite({1,2})` retourne le tableau `{2, 1}`

Insistons sur le fait que ces *vérification seront elles-mêmes écrites en java*. Il 'agit bien d'écrire du code qui vérifie le bon fonctionnement d'un autre code.

Chaque invocation avec un argument concret différent est nommée ***un cas de test***. Les méthodes à comportements multiples requièrent beaucoup d'invocations différentes, donc beaucoup de cas de tests. En Junit chaque cas de test est implanté dans une méthode spécifique.

Rappel du principe des tests avec Junit4

Les tests en Junit4 sont réalisés en principe sur toutes les méthodes d'une classe. On distingue la **classe à tester** (`ShiftTableau` pour nous), de la **classe contenant les tests** à exécuter par Junit4 sur les méthodes de cette première. Par convention, la classe des tests a le nom de la classe à tester suivi de `Test`. Ainsi, notre classe de tests se nommera `ShiftTableauTest`. Dans ces deux classes :

- classe à tester (`ShiftTableau`) : contient toutes les méthodes à tester avec leurs contrats javadoc + leur code ;
- la classe avec les tests (`ShiftTableauTest`) : pour chaque méthode à tester, contient plusieurs méthodes précédées chacune de l'annotation `@Test`.
  Chacune contient un « cas de test » : elle invoque la méthode à tester sur une certaine valeur concrète puis compare avec le résultat attendu à l'aide d'« instructions » Junit4. Ex : pour tester la méthode `decaleDroite`, il faudra écrire une méthode qui appelle `decaleDroite` sur un tableau null, une autre sur un tableau vide, une autre sur un tableau à un seul élément, etc. Dans chacune de ses méthodes de test il faudra programmer la comparaison avec la valeur correcte de retour (où l'exception correcte à lever).

Une fois ces deux classes définies, on peut demander à exécuter les tests du fichier de tests. Junit4 exécute alors chaque méthode précédée de `@Test` et enregistre les succès ou échecs de chacun de ces tests.

Réalisation pas à pas des tests pour `decaleDroite`

Pas 1 : création de la documentation javadoc

Le contrat (commentaire au format javadoc) qui accompagne la méthode à tester est nécessaire pour savoir ce qu'elle doit « répondre » après une invocation. Si nous ne comprenons pas ce contrat, nous ne pourrons pas concevoir les tests. Pour faciliter sa lecture, nous générons avec Eclipse sa version html. Après avoir sélectionné votre classe à tester, cliquez sur *Project/Generate javadoc* pour produire une page HTML à partir des commentaires javadoc. Vous pourrez visualiser le résultat avec *Windows/Show View/Javadoc*.

Pas 2 : création de la classe de tests et d'un répertoire `Test Package`

La classe des tests n'est pas une classe java ordinaire : elle possède un préambule assez long contenant des commandes à exécuter par Junit4 avant de démarrer le moindre test sur la classe à tester. Netbeans, eclipse et autres IDE sont capables de générer la classe des tests associée à une classe à tester. Il y a plusieurs possibilité concernant l'emplacement des classes de tests. Nous faisons le choix suivant (pas par défaut dans Eclipse) : On crée un nouveau répertoire `test` (dans le projet Eclipse) où sont mises toutes les classes de tests du projet. Cela permet de séparer les classes « source » des classes « avec les tests ». Si votre projet est un projet MAVEN, ce répertoire sera `<racine du projet>/src/test/java`. Note importante : on reproduit la même structure de package dans ce répertoire que dans le répertoire source (maven : `<racine du projet>/src/main/java`).

Eclipse permet de créer votre classe de tests en précisant l'emplacement voulu (et en créant le répertoire associé si nécessaire) :

- Si nécessaire (normalement inutile dans ce tp) créez un répertoire `test` à côté du répertoire `src` : clique droit sur le projet / New / Source folder, remplissez le formulaire, validez.
- Sélectionnez la classe à tester (`ShiftTableau.java`) + clic droit + *New / Junit Test Case*.

- Si nécessaire changez le répertoire (≪ Source folder ≫) pour pointer vers le répertoire `test` nouvellement créé. Validez, une nouvelle classe générée par Eclipse, de nom `ShiftTableauTest`, et contenant un *squelette* de test par méthode de `ShiftTableau`.
- **Attention 1** : lorsque vous répétez cette opération il faut cocher les méthodes pour lesquelles vous voulez (re-)générer les squelettes. Pensez à effacer les squelettes dont vous n'avez pas besoin à chaque fois.

— Arrivés ici, vous devriez avoir dans votre projet deux classes : `ShiftTableau` dans `src` et `ShiftTableauTest` dans `test`.

— La classe de test générée contient un premier test Junit4 généré automatiquement (squelette). Il a la forme d'une méthode statique nommée `decaleDroiteTest()`. Ce nom est celui de l'unique méthode à tester suivi de `Test`.

— **Attention 2** : cette méthode de test générée automatiquement **échoue systématiquement**. Allez voir son code et assurez vous de comprendre pourquoi.

Pas 3 : étude du cas de test généré par Eclipse

La nouvelle classe `ShiftTableauTest` (répertoire `Test Packages`) est destinée à contenir tous les tests pour la classe `ShiftTableau`. Eclipse génère le squelette de plusieurs méthodes (avec corps vide) que nous ne regarderons pas pour l'instant. La méthode qui nous intéresse est précedée de l'annotation `@Test`. Elle nous servira de schéma pour créer toutes les méthodes de test pour notre fonction. Son code doit ressembler à ceci :

```
1      /**
2       * Test of decaleDroite method, of class ShiftTableau.
3       */
4     @Test
5      public void testDecaleDroite() {
6          fail("The test case is a prototype.");
7      }
```

Le corps de cette méthode se termine par `fail`, ce qui provoque l'échec systématique de ce test. Vous devez donc modifier le code de cette méthode par un vrai cas de test (un commentaire mis par netbeans vous le rappelle d'ailleurs !).

Caractéristiques d'un cas de test

Pour rappel, *un cas de test* : (1) teste en général **un seul cas de comportement** de la méthode ; (2) est mis dans une méthode **sans paramètre et qui retourne void** et qui est précédée de l'annotation `@Test` (pour que Junit sache qu'il doit l'exécuter) ; (3) on donne à cette méthode un nom informatif sur la nature du cas testé, cela facilite le travail d'énumération des cas à tester. (4) on y ajoute un commentaire qui décrit le cas testé.

Structure d'un cas de test

Examinons le code typique d'une méthode de test (ici on teste la méthode `toto` qui prend un **`int`** et retourne un **`char`**) :

```
1  public void testToto() {
2      System.out.println("decaleDroite sur singleton");
3      int arg1 = ...;
4      char actResult = toto(arg);
5      char expResult = ...;
6      assertArrayEquals(expResult, result);
7  }
```

La structure (en général) d'un cas de test est la suivante :

lignes 3-4 : on invoque la méthode à tester en lui passant des valeurs concrètes pour ses paramètres.

ligne 4 : on récupère le résultat obtenu (`act`ual) par cet appel.

ligne 5-6 : on teste si ce résultat obtenu est égal à celui attendu (`exp`ected) d'après le contrat.
Pour comparer, on utilise des commandes Junit4. Ici, celle pour comparer le contenu de deux valeurs simples est `assertArray`, voir ci-dessous.

Pas 4 : écriture et exécution d'un premier cas de test

Remplacez le code généré par Eclipse par les **deux** méthodes de test suivantes. Elles correspondent à deux cas de test différents. Notez que le nom de la première est parlant.

```
/**
 * Test sur tableau avec une seule case.
 */
@Test
public void testDecaleDroiteSingleton() {
    System.out.println("decaleDroite␣sur␣singleton");
    int[] t = {1};
    int[] expResult = ... // A completer !!!
    int[] result = ...  // A completer !!!
    assertArrayEquals(expResult, result);
}
/**
 * A COMPLETER : Décrivez ici le cas qui est testé ici !!!
 */
@Test
public void testDecaleDroite() {  // Renommez cette methode !!!
    System.out.println("decaleDroite");
    int[] t = {1,2};
    int[] expResult =  {2, 1};
    int[] result = ShiftTableau.decaleDroite(t);
    assertArrayEquals(expResult, result);
}
```

Question 1

1. La première méthode teste le cas d'appel sur un tableau d'une seule case. Comprenez vous pourquoi c'est un cas important à tester ? Son code est incomplet. Complétez ce code.
2. Quel est la cas qui est testé par la deuxième méthode ?
   (a) Complétez le commentaire pour décrire ce cas de test.
   (b) Renommez la méthode pour rendre compte du cas testé.
3. Exécutez ces deux tests en faisant un clic droit sur *Test file*. Il faudra peut-être ouvrir la fenêtre qui donne le rapport des tests. Que se passe-t-il ? Expliquez ce comportement.
4. Vos tests échouent tous très probablement. La raison est que le code de la méthode retourne `null` systématiquement. On va tester du vrai code dans les pas 6 et 7.

Pas 5 : ajout des tests manquant

Ces deux tests ne suffisent pas à couvrir tous les cas de comportements différents de notre fonction (voir le cours *Javadoc & Junit* pour plus de détails). Ajoutez dans la classe `ShiftTableauTest` les tests qui manquent selon vous.

Pas 6 : tester une implantation buggée

Dans cette question vous allez utiliser vos tests pour tester une version bugée de la méthode montrée plus bas. Son code est à trouver dans la classe `ShiftTableuBugge` fournie. Que se passe-t-il en jouant vos testes sur cette classe ? Si vous trouvez une erreur, corrigez la, puis testez/corrigez jusqu'a ce qu'il n'y ait plus d'erreur.

```
public static int [] decaleDroite(int [] t) {
      int [] nouveau = new int [t.length];
      for (int i=0; i<t.length-1; i++){
          nouveau[i] = t[i+1];
      }
      nouveau[0]=t[t.length-1];
      return nouveau;
}
```

Pas 7 : tester une implantation correcte

Le code suivant implante correctement le contrat de la fonction. Testez cette implantation avec vos tests. S'ils sont correctement écrits ils devraient tous réussir !

```
public static int [] decaleDroite(int [] t) {
     if (t==null)
          throw new IllegalArgumentException();
     int [] nouveau = new int [t.length];
     if (t.length==0)
          return nouveau;
     nouveau[0] = t[t.length-1];
     for (int i=0; i<=t.length-2; i++){
          nouveau[i+1] = t[i];
     }
     return nouveau;
}
```

Exercice 2

Question 1

Dans cet exercice va inverser l'ordre préconisé pour la conception de tests : au lieu d'écrire la spécification en premier lieu et ensuite les tests associés à celle-ci et seulement après le code d'implantation, on partira ici d'une implantation que l'on veut documenter. Cela arrive si par exemple, le code a été développé sans documentation et sans tests. Il s'agit donc d'écrire le contrat javadoc qui correspond au comportement de ce code précis, et qui lui servira de documentation. Faites bien attention à documenter touts les cas de l'exécution de ce code. Ecrivez ensuite les tests permettant de tester le contrat javadoc que vous avez écrit et utilisez les pour tester cette fonction.

```
/**
 * ECRIRE LA JAVADOC QUI DOCUMENTE LE COMPORTEMENT DE CE CODE
 */
static boolean estPalindrome(String m) {
        int gauche = 0;
        int droite = m.length() - 1;
        while (gauche < droite) {
            if (m.charAt(droite) != m.charAt(gauche)) {
                return false;
            }
```

```
            gauche++;
            droite--;
        }
        return true;
}
```

Question 2 : écriture de contrats javadoc

Ecrivez le contrat javadoc décrivant le comportement attendu pour la fonction static int estTrie((int[] t) qui prend en paramètre un tableau d'entiers et teste si les éléments du tableau sont triés. Vous aurez à faire des choix : l'ordre testé est il croissant ou décroissant, est-il strict ou pas, quel comportement pour les cas limites. Rappel : un ordre croissant est strict si toute paire de valeurs consécutives du tableau sont en ordre croissant strict. Si c'est un ordre croissant non strict, deux valeurs consécutives pourront être égales. Écrivez ensuite les tests CORRESPONDANT au contrat que vous avez donné, et enfin écrivez une implantation qui respecte ce contrat et passez dessus tous vos tests.

```
/**
 * ECRIRE CE CONTRAT
 */
static boolean estTrie(int[] t){...}
```