Algorithmique Programmation FIP

ING39 : Programmation en Java

Virginia Aponte

CNAM-Paris

Principes du test automatique

Tester du code

- Du code déjà écrit ou à écrire, qu'on souhaite tester.
- Avec contrats javadoc expliquant ce qu'il est censé faire.
- On sait donc quoi tester.

```
/**
  * Calcule a puissance n (qu'on notera a^n)
  * @param a entier
  * @param n entier positif ou nul.
  * @return a ^ n
  * @throws IllegalArgumentException si n negatif.
 */
 public static int puissance(int a, int n) {
   if (n < 0) throw new IllegalArgumentException();
   int resultat= 0;
       ...
  }
```

Solution simpliste : affichages d'appels dans un `main`

- Appels pour les cas à tester + affichage résultats.
- Des commentaires indiquent les résultats attendus.
- On pourra les comparer aux résultats affichés

```
public static void main(String[] args) {
  // Test 0^0 = 1
  System.out.println("0^0 => "+Calcul.puissance(0, 0));
  // Test 10^1 = 10
  System.out.println("10^1 => "+Calcul.puissance(10, 1))
  System.out.println("2^3 => "+Calcul.puissance(2, 3));
  // Test 2^(-3) -> IllegalArgumentException
  System.out.println("2^-3 => ");
  Calcul.puissance(2, -3);
}
```

Affichages du `main` pour notre code buggé

```
0^0 => 0
10^1 => 0
2^3 => 0
2^-3 =>
Exception in thread "main" java.lang.IllegalArgumentExce
  at cours2_demojunit.Calcul.puissance(Calcul.java:23)
  at cours2_demojunit.TestBasical.main(TestBasical.java:
```

Test Manuel : on doit examiner (avec nos petits yeux) ces affichages en les comparant aux résultats attendus afin de déterminer quels tests ont réussit/échoué.

Les tests manuels ne passent pas à l'échelle

- On a déjà tous testé du code de cette manière.
- Ici, nous testons une seule méthode dont le résultat est simple à comparar (un entier).
- Nous la testons sur 4 cas de fonctionnement.
- Si notre code contient des centaines de méthodes, dont certaines retournent ou modifient des structures complexe (tableaux, collections), cette manière de faire ne passe pas à l'échelle.
- Sans parler des tests qu'on doit ajouter/modifier au fil des évolutiuons de notre code.

Tester du code de manière automatique

- On ne veut de tests manuels, p.e. via une suite d'affichages qu'on doit comparer nous mêmes avec les résultats attendus.
- On veut qu'ils soient automatiques (on lance une commande, ou un script et les tests son joués)
- On veut qu'ils soient systématiques (on pourra les *re-jouer* à la demande, p.e. avant chaque intégration du code dans le repository global)

Comment automatiser un test ?

- On **écrit une méthode qui réalise chaque test** ! C'est un *cas de test*
- son but : tester que notre code se comporte comme attendu.
- Grosso modo elle va :
  - invoquer une méthode à tester sur des entrées bien particulières,
  - récupérer le résultat de cette invocation
  - le comparer à ce qui est attendu dans ce cas.
  - si le résultat est « conforme », le test réussit, sinon il échoue.

1 Test = méthode qui invoque et teste une de nos fonctions à tester

Que fait une méthode de test ?

Elle teste un cas particulier de fonctionnement de notre code.

Exemple : on veut tester la fonction

```
static int puissance(int n, int exposant){ ....}
```

- Il y aura autant de méthodes de test que de cas de fonctionnement :
  - tester si l'appel `puissance(100, 1)` ⇒ 100
  - tester si l'appel `puissance(100, 0)` ⇒ 1
  - tester si l'appel `puissance(2, 3)` ⇒ 8
  - tester si l'appel `puissance(2, -3)` lève l'exception `IllegalArgumentException` (à supposer que ce soit le comportément attendu si l'exposant est négatif)
  - ...
- Nous devons coder chacun de ses cas par une méthode différente, écrite en pseudo-java, et exécutable par JUNIT.

Comment se passer du `main` ?

Une fois le code du test écrit, comment le lancer (et le relancer ?) sans passer par le `main` ?

- On se procure une plateforme capable d'exécuter le code des tests qu'on a écrit et de nous informer des succès et échecs de chacun des tests joués.
- JUNIT est une plateforme
  - capable d'éxécuter des méthodes de tests écrits en pseudo-java
  - la plateforme garde une trace de la réussite des tests effectués
  - Elle est intégrée par des nombreux IDEs

Test unitaire

Morceau de code chargé de tester qu'une *fonctionnalité élémentaire* du code se comporte comme prévu.

Particularités :

- on teste uniquement les fonctions, c.a.d., les sous-programmes dont les entrées et résultat sont bien identifiés. Ex : on ne teste pas *unitairement* du code d'entrées/sorties, une interface graphique, etc.
- Indispensable : une specification précise du comportement attendu. Autrement on ne saura quoi tester !

Test unitaire = test sur une fonction

Incontournable : une spécification précise (Javadoc)

Nous utiliserons le format Javadoc afin de spécifier précisément les fonctions à tester.

- une fonction f à tester, avec paramètres $x_1, x_2, \ldots x_n$ et un résultat r bien identifiés ;
- une spécification précise du comportement de f dans tous les cas possibles de ses paramètres :
  - que fait/renvoie la fonction si tel argument est dans un cas limite (indice en dehors d'un tableau, pointeur nul, tableau vide, etc).
  - comportement/résultat obtenu pour TOUS les cas :
    - Ex : pour tel cas la fonction lève l'exception UnTel ; pour tel autre, elle retourne un nombre positif, etc.

Tests unitaires (utilisation)

Utilisés à plusieurs moments du *développement* :

- Avant de la coder :
  - sert de *spécification préliminaire* au code qui sera écrit ;
  - permet de ce concentrer sur les comportements attendus ;
- Pendant son développement et avant intégration dans le logiciel : tester que le code en cours de développement fonctionne comme prévu ;
- Après son intégration : ré-tester le code après chaque changement ;

JUnit

Environnement dédié à l'automatisation de tests unitaires pour Java :

- pour exécuter certaines méthodes Java :
  - identifiées via annotations `@Test` ;
  - souvent mises dans une classe séparée dédiée au test
- assertions : pour tester/signaler la réussite de tests ;
- utilise une bibliothèque à charger au préalable

JUnit intégré aux IDE

Plusieurs environnement de développement proposent un module d'extension dédié à JUnit :

- réjouer les tests via l'interface de l'IDE ;
- afficher graphique de résultats, statistiques, etc.
- fabriquer des *templates* de cas de test : un seul par méthode de la classe à tester ;

Qu'est-ce qu'un cas de test ?

Cas simple : tester *une fonction f* qui prend 2 paramètres x, y ;

- un cas de test pour f est une configuration :

x	y	résultat attendu
v_x	v_y	r

  - v_x et v_y : valeurs concrète pour les paramètre de f,
  - r : résultat attendu en retour de la fonction pour ses entrées.

- Jeu de tests : table avec plusieurs cas de test.
- Cette table contient une colonne par paramètre de la fonction + une colonne pour le résultat attendu.

Un cas de test

On veut coder une méthode de test sur notre fonction à tester f :

x	y	résultat attendu
v_x	v_y	r

1. Invoquer f sur les entrées données pour ce cas...
2. ... tout en récupérant le résultat retourné :
   `result =` $f(v_x, v_y)$`;`
3. comparer résultat obtenu `result` avec résultat attendu r ;
4. envoyer un rapport de réussite ou d'échec selon qu'ils soient égaux ou différents.

JUnit sous Eclipse

Pour tester le fichier `ABC.java` (répertoire **src**) :

1. se placer sur le fichier `ABC.java` à tester (et non pas sur son paquetage, ou répertoire)
2. sélectionner *New JUNIT Test Case*. Cela déclenchera l'installation du plugin si ce n'est pas déjà fait (ne marche pas toujours !) ;
   - une nouvelle classe `ABCTest.java` contient au minimum un *template* de méthode de test ;
3. récopier ces templates autant de fois que des cas de test à programmer et les instancier sur les cas de test souhaités.

La suite dans le cours dédié à JUNIT !