Introduction aux tests

DEV 2.3 (R2.03 Qualité de développement)

Florent Madelaine

IUT de Sénart Fontainebleau
Département Informatique

Année 2023-2024
Cours 2

La semaine dernière

- Rappels programmation défensive.
- Démo test unitaires avec Junit.
- Indicateurs de qualité des tests : couverture.

Quickstart JUnit

Requis

- Sur votre machine personnelle, il faut installer JUnit4 : il faut `JUnit4` et `hamcrest-core` (ce sont des jar)

  `https://github.com/junit-team/junit4/wiki/Download-and-Install`
- faire ce qu'il faut à votre `CLASSPATH` pour que ces jar y apparaissent et que la compilation se fasse sans problème.

Exemple

À l'IUT il suffit de saisir cette ligne dans votre terminal (ou dans votre `.bashrc`)

```
export CLASSPATH=".:/usr/share/java/junit.jar:/usr/share/java/hamcrest-core.jar:$CLASSPATH"
```

Principe de base en JUnit

- Pour chaque classe java on a un test JUnit (qui est lui aussi un fichier java).
- En général on le nomme plus ou moins pareil que la classe avec Test quelque part dans le nom.
- On a au moins un test pour chaque méthode pas trop simple. (revoir TD semaine dernière sur la notion de couverture).

Exemple

- Fichier de notre classe : `Calculator.java`
- Fichier testant cette classe : `CalculatorTest.java`

Hello World en JUnit

Comment faire tourner les tests?

En pratique

1. compiler les fichiers

```
$ javac *.java
```

2. lancer le test

```
$ java org.junit.runner.JUnitCore CalculatorTest0
JUnit version 4.12
.
Time: 0.004

OK (1 test)
```

Exemples plus complexes d'utilisation de Junit

Vous pouvez regarder le code que j'ai tiré de la documentation de junit4 qui est disponible sur le git.

Lisez sur le git

`Junit4Exemples.tar.gz`

Indicateurs de qualité des tests

Nous verrons plus tard comment appliquer des règles empiriques pour établir des tests qu'on espèrera pertinents. En supposant que les tests soient pertinents, il existe des indicateurs pour juger plus ou moins grossièrement si on a testé suffisamment et au bon endroit.

Tester suffisamment et au bon endroit

On peut être plus ou moins exhaustif en fonction du code concerné.

Méthode Est-ce que toutes les méthodes ont été exécutées?

Choix Est-ce que les structures de contrôle (`if`, `while`, etc) ont été testées pour les 2 cas vrai et faux?

Chemin : Est-ce que tous les chemins possibles d'exécution des fonctions d'un programme ont été exécutés?

En pratique

Il convient surtout d'avoir du bon sens.
Si peu de temps, pas besoin de tester des méthodes simples (get et set qui agissent sans surprise sur un attribut), il vaut mieux se concentrer sur les méthodes qui font quelque chose de plus complexe.
Pour les méthodes plus complexes (ayant besoin d'avoir un diagramme d'activité), il convient de faire a minima un test pour chaque choix et idéalement un test pour chaque chemin.
Chaque test correspond donc à un diagramme de séquence système qui est un cas particulier du diagramme d'activité.

Diagramme d'activité

Le *diagramme d'activité* est un diagramme proposé par UML (langage unifié de modélisation, normé).
En gros, nous allons utiliser cette norme pour dessiner un algorigramme.
La suite au tableau / ou démo starUML.

Qu’est-ce-qui fait un bon test

Maintenant que nous avons entrevu les aspects techniques du test avec Junit. Il faut que nous réfléchissions à quels tests nous devons écrire.

Tester correctement n’est pas une tâche simple. Comme la programmation, c’est aussi une activité qui demande de l’entraînement et un peu de réflexion préalable.

Nous avons déjà évoqué la semaine dernière qu’il fallait essayer de couvrir toutes les branches d’une méthode importante.

On peut toutefois couvrir toutes les branches avec des tests sans intérêt. Il convient donc d’en fabriquer qui sont pertinents.

Right BICEP

Jeff Langr, Andy Hunt et Dave Thomas, les auteurs de *Pragmatic Unit Testing in Java 8 with JUnit*, proposent un cadre simple basé sur l'acronyme Right BICEP.

Right Are the results right?
B Are all the boundary conditions CORRECT?
I Can you check inverse relationships?
C Can you cross-check results using other means?
E Can you force error conditions to happen?
P Are performance characteristics within bounds?

Slogan

"Utilise ton biceps droit!"

Right BICEP

Jeff Langr, Andy Hunt et Dave Thomas, les auteurs de *Pragmatic Unit Testing in Java 8 with JUnit*, proposent un cadre simple basé sur l'acronyme Right BICEP.

Right Are the results right?
B Are all the boundary conditions CORRECT?
I Can you check inverse relationships?
C Can you cross-check results using other means?
E Can you force error conditions to happen?
P Are performance characteristics within bounds?

Slogan

"Use your Right BICEP"

Right : que doit faire la méthode?

are the results right?

Ce sont les tests les plus simples à écrire normalement. A priori ils sont implicites dans les spécifications.

- Si ce n'est pas clair, il faut demander au client ou au responsable du projet qui a vu le client.
- En attendant, il faut réfléchir vous-même à ce que vous pensez être la bonne spécification.
- De toute façon la spécification (et donc les tests) risquent de changer en cours de projet.

dh'o

Pas possible de coder la méthode si on n'a pas une petite idée de ce qu'elle doit faire.

B : les cas limites.

Are all the boundary conditions CORRECT?

On teste une forme de robustesse de la méthode.

Ce sont souvent les cas qui ne doivent pas être traités normalement : entrée incohérente ou non conforme ou encore une entrée du type correct mais en dehors des valeurs attendues.

Pour des structures de données sous-jacentes qui sont ordonnées, on regarde ce qui se passe pour des cas particuliers concernant le premier et le dernier élément (erreur borne parcours).

B : les cas limites

Exemples

- fichier avec un nom invraisemblable

  `xyz#"\/toto^~#-([°`
- adresse mail pas bien formée

  `toto.titi.tata@com`
- valeur manquante

  `null`
- type correct mais valeur pas adaptée (e.g. taille d'une personne de 3m)

B : les cas limites.

Are all the boundary conditions CORRECT?

En fait ce cas est une sorte de cas un peu fourre tout pour lequel Jeff Langr, Andy Hunt et Dave Thomas, proposent un acronymes de plus CORRECT.

Conformance

Order

Range

Reference

Existence

Cardinality

Time

Idée générale

Pour les données en entrée de la méthode ou maintenues en internes pour la classe, qu'est-ce-qui pourrait mal se passer?

Can you check inverse relationships?

Certaines méthodes ont un inverse naturel facile à vérifier.

Exemples

- racine carré et produit.
- inverse modulaire et produit modulaire.

À éviter

ne pas coder les 2 méthodes soit-même.

C

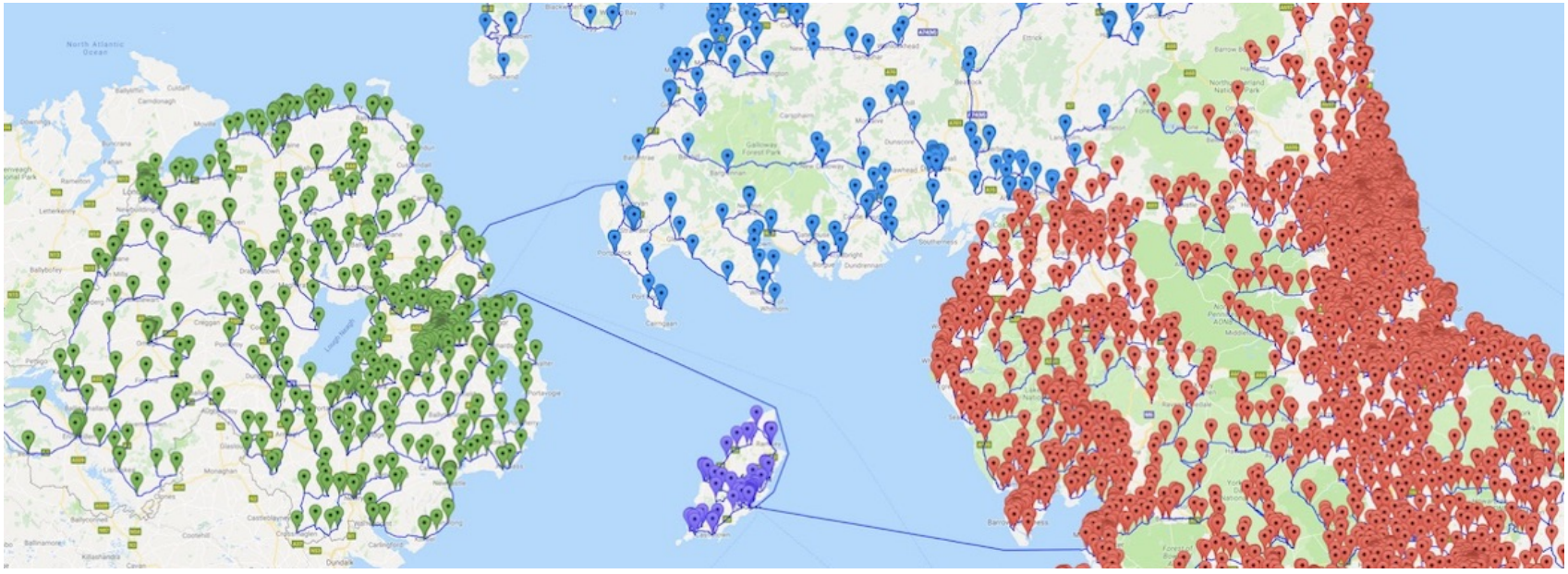
Can you cross-check results using other means?

Idée similaire mais plus générale que la précédente.

Exemple

- Utilisez une méthode intelligente pour résoudre le voyageur de commerce.
- Comparer avec la méthode bête qui teste toutes les permutations de villes possibles.

Digression : le voyageur de commerce



pub crawl des 24,727 pubs du Royaume-Uni.

`http://www.math.uwaterloo.ca/tsp/uk/index.html`

E

Can you force error conditions to happen?

Force ou simule des erreurs pour voir si l'appli tient le coup (disque plein, coupure réseau, bdd inaccessible, problème d'horloge etc).

Dépasse le cadre de ce cours. En pratique, on ne vas pas le faire réellement. On peut même le faire de manière interne à java avec les Mock Objects.

P

Are performance characteristics within bounds?

On étudie par exemple l'efficacité de la méthode avec l'augmentation de la taille de l'entrée.

Exemple

Est-ce-que ma super méthode théoriquement linéaire sur les graphes l'est vraiment en pratique si je passe de 1000 sommets à 100 000 sommets?

Remarque

Pour aller plus loin il y a des outils dédiés come JUnitPerf.

Calcul du plus grand entier dans une liste

```
public class Largest {
/**
* Return the largest element in a list.
*
* @param list A list of integers
* @return The largest number in the given list
*/
    public static int largest(int[] list) {
        int index, max=Integer.MAX_VALUE;
        for (index = 0; index < list.length-1; index++)
            if (list[index] > max) {
                max = list[index];
            }
        }
        return max;
    }
}
```

Remarque finale

Pour éviter les bugs liés au parcours de liste etc, il vaut mieux éviter les index manuels et utiliser la facilité java des boucles permettant d'utiliser l'interface `iterable`.

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html

Exemple

```
int[] numbers =
    {1,2,3,4,5,6,7,8,9,10};
 for (int item : numbers) {
    System.out.println("Count is: " + item);
 }
```

boucle améliorée et itérable

Fonctionne en java plus généralement quand une classe implémente l'interface Iterable.

https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html

boucle améliorée et types énumérés.

Les types énumérés sont naturellement itérable et peuvent avoir des attributs. Un exemple ici pour une monaie fictive, le grisbi.
On déclare un type énuméré avec un attribut.

Exemple

```
public enum Grisbi {
    ARTISTE  (113),
    PRINCE  (57),
    BISEXTILE (29),
    ISCARIOTE (13),
    TUNE   (7),
    PEZE   (3),
    GRISBI (1),
    ;
    private final int denomination;   // in grisbi
    Grisbi(int denomination) {
        this.denomination=denomination;
    }
    public int getDenomination() { return denomination; }
}
```

boucle améliorée et types énumérés.

Pour une classe enum, la méthode statique values() retourne un tableau des “valeurs” du type énuméré dans l’ordre de leur déclaration. Ce tableau peut être traité par la boucle améliorée.

https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

Exemple

itérer sur un type énuméré.

```
for (Grisbi g : Grisbi.values())
   System.out.printf("%s est le grisbi de dénomination %d%n",
                     g, g.getDenomination());
```

Une question

Peut on trouver l'IDE idéal?

Ce que je voudrais dans mon IDE favori.

Un plugin qui me permet

- de donner des couleurs différentes à certains mots-clés
- d'indenter mon code
- de détecter les noms de variables / méthodes inconnus
- de détecter le code mort
- de lancer des tests unitaires
- de vérifier que tous les cas sont couverts dans des branchements.
- de vérifier que 2 méthodes font la même chose (donnent le même résultat)
- de vérifier que le calcul d'une méthode s'arrête tout le temps.

Bibliographie

- documentation de Junit4.
- *Pragmatic Unit Testing in Java 8 with JUnit* par Jeff Langr, Andy Hunt et Dave Thomas.