Introduction aux tests

DEV 2.3

Florent Madelaine

IUT de Sénart Fontainebleau
Département Informatique

Année 2021-2022

Les deux dernières semaines

- Assertion Java.
- Programmation défensive.
- Test unitaires avec Junit.

Assertion Native de Java

- On écrit directement dans le code.
  `assert <test>;`
  ou si on veut un message détaillé.
  `assert <test> : <message>;`
- On lance avec l'option `-ea` (pour *enable assertion*)
- Si on valide le test, on continue après l'assertion
- Sinon l'exécution est interrompue par une `AssertionError`

Exemple

```
$ java -ea Liasse
Exception in thread "main" java.lang.AssertionError:
la liasse doit contenir un nombre positif de chaque dénomination
at Liasse.take(Liasse.java:22)
at Liasse.main(Liasse.java:61)
```

En bref

Les assertions c'est un mode debug.

Programmation Défensive

Principe qui consiste à restreindre l'utilisation des méthodes, en particulier des méthodes publiques.

précondition On restreint les paramètres / valeurs des attributs avant l'appel de la méthode.

On va envoyer des exceptions adaptées `IllegalArgumentException`, `NullPointerException`, `IllegalStateException` etc

invariant On teste les invariants d'une méthode/ d'une classe.

Normalement un assert suffit.

postcondition On vérifie après exécution de la méthode que les propriétés adéquates sont satisfaites.

Normalement un assert suffit.

Programmation Défensive

Pour les méthodes publiques

`Exception` vs `AssertionError`

- Il convient de toujours protéger l'usage d'une méthode publique donc lever une exception dans ce cas.
- Pour tous les autres cas, si le code est correct il n'y a pas d'erreur possible (les invariants et postconditions sont satisfaites). Une assertion suffit.

Quels tests unitaires?

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

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

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, un peu de bon sens

Pas besoin de tester des méthodes trop simples.
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.

Set-up 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 mettre cette ligne dans votre `.bashrc`

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

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

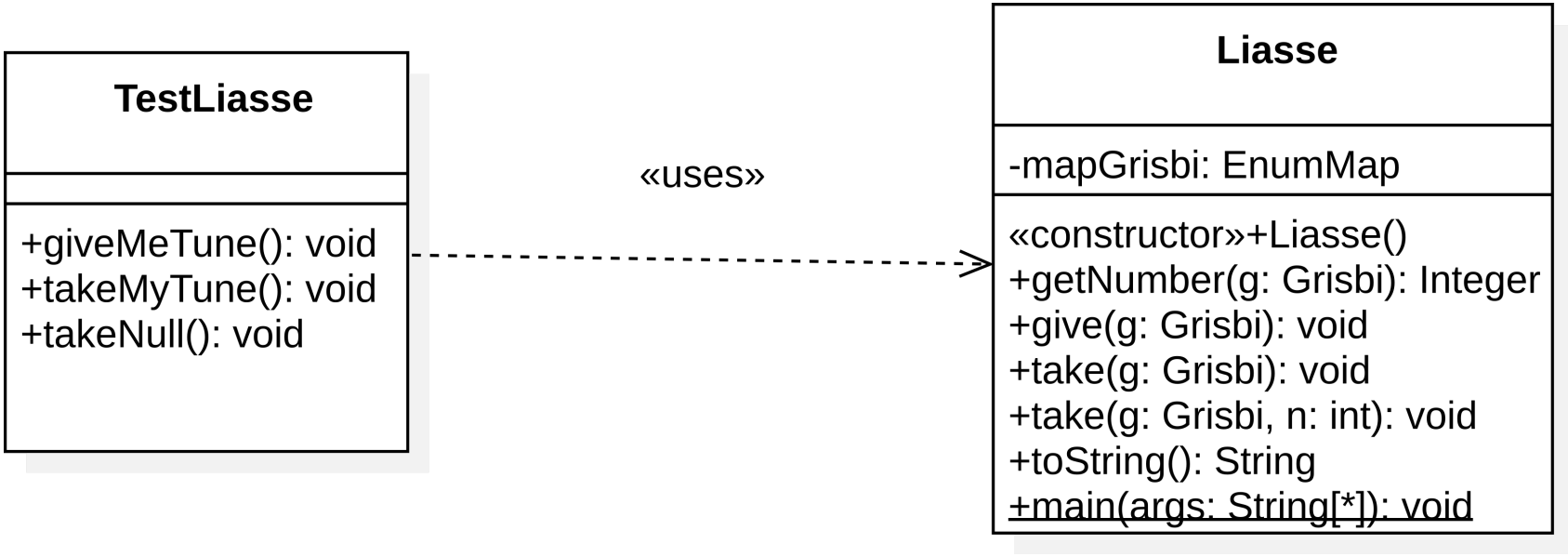
1. compiler les fichiers

```
$ javac *.java
```

2. lancer le test

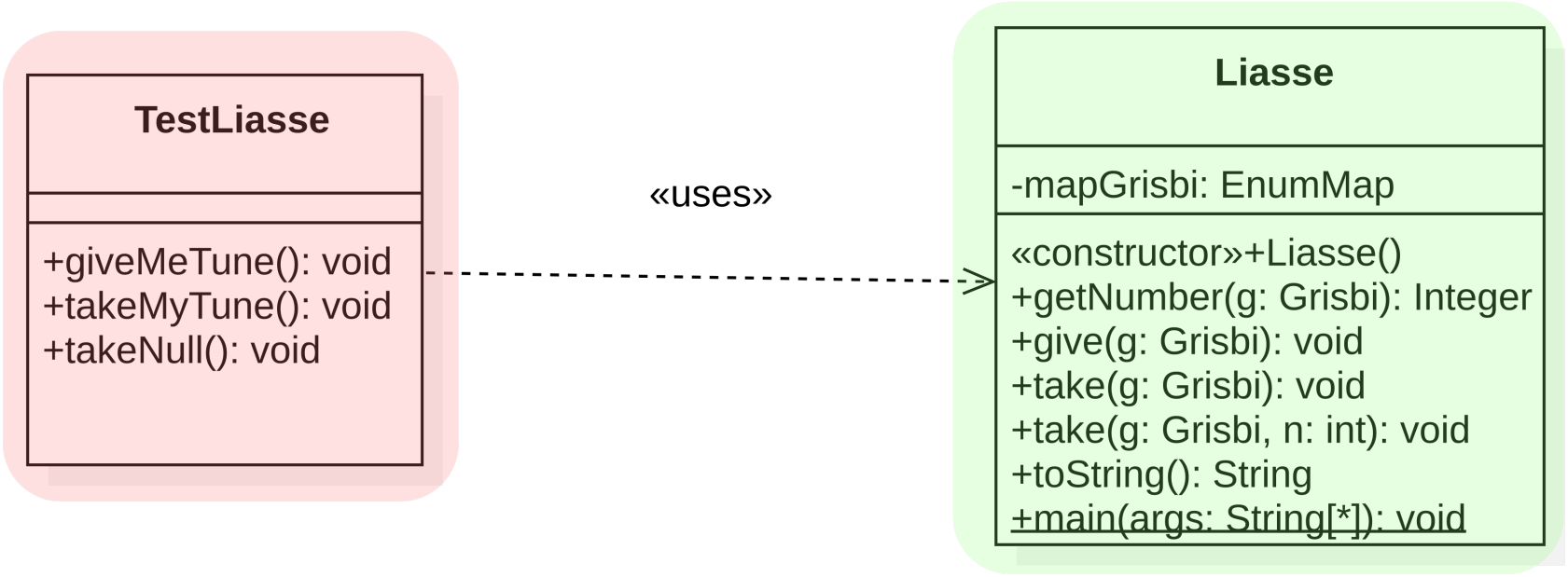
```
$ java org.junit.runner.JUnitCore CalculatorTest0
JUnit version 4.12
.
Time: 0.004

OK (1 test)
```

Junit sur une image



On exécute la classe de test, qui elle va appeler la classe qu'on souhaite tester.

Junit sur une image



On exécute la classe de test, qui elle va appeler la classe qu'on souhaite tester.

Hello World en JUnit

Fichier Calculator.java

```
/**
   Calculator est une classe offrant une seule méthode qui évalue une
   somme, donnée sous la forme d'une chaîne de caractère listant des
   opérandes  séparées par des +
*/
public class Calculator {
    /**
       somme les opérandes passées sous forme d'une chaîne de
       caractères et retourne le résultat sous forme d'entier.
       @param expression : chaîne de caractères
         ("nombres" séparés par des + sans espaces).
         Par exemple "42+3" ou encore "-42+42"
         (le moins unaire est autorisé).
         plus spécifiquement nombre est à comprendre au sens de
         parseInt(java.lang.String)
       @throws NumberFormatException :
         si l'expression n'est pas dans ce format
         (par exemple "x+2" ou " 1 +2" -- il y a des espaces --
         ou encore "9999999990").
     */
    public int evaluate(String expression) {
        int sum = 0;
        for (String summand: expression.split("\\+"))
            sum += Integer.valueOf(summand);
        return sum;
    }
    /**
       Pour appeler cette super méthode depuis la ligne de commande
      (on ne regarde que le premier argument, les autres sont ignorés).
     */
    public static void main(String[] args) {
        Calculator calculator = new Calculator();
        System.out.println(calculator.evaluate(args[0]));
    }
}
```

Hello World en JUnit

fichier CalculatorTest

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {


    // un test pour Junit4 c'est une méthode avec l'annotation suivante devant la méthode.
    @Test
    public void evaluatesGoodExpression() {
        Calculator calculator = new Calculator();
        int sum = calculator.evaluate("1+2+3");
        // on peut stipuler que des choses sont normalement égales
        // charger de manière statique les Assert si on veut éviter d'avoir à écrire de quelle classe on parle)
        assertEquals(6, sum);
    }
}
```

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 eprel.

Lisez sur eprel

`Junit4Exemples.tar.gz`

Attention

Il ne faut pas confondre le mécanisme d'assertion de JUnit (fait pour fonctionner dans un fichier de test JUnit avec celui de Java.

- AssertTrue (équivalent du assert natif java)
- AssertFalse (le dual du précédent)
- AssertEquals
- AssertSame / AssertNotSame
- AssertNull / AssertNotNull
- etc

NB. Ne pas oublier de charger en ajoutant la bonne ligne en haut du fichier JUnit. Par exemple :

```
import static org.junit.Assert.assertEquals;
```

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;
    }
}
```

Acronyme

Right Bicep

Right

- **B** (boundary conditions CORRECT)
- **I** (inverse)
- **C** (cross-check)
- **E** (error conditions)
- **P** (performance)

CORRECT

Conformance
Order
Range
Reference
Existence
Cardinality
Time

Jeff Langr, Andy Hunt et Dave Thomas, les auteurs de *Pragmatic Unit Testing in Java 8 with JUnit*

Version simplifiée pour le TP noté

- Pour chaque classe une classe de test JUnit.
- Pour chaque méthode, des tests pour les exceptions / vérification préconditions

  [NB. En pratique dans le futur pas forcément nécessaire, on met dans le code directement, voir prog défensive, on a autant de chance d'oublier le test que d'oublier une condition dans une méthode. Je vous demande de le faire pour pouvoir revoir les exceptions et bien comprendre comment JUnit gère les exceptions]
- a minima un test pour chaque type de réponse de la méthode si nombre fini
- plus généralement, couverture par chemin : tenter de faire un test par cas.
- si il y a une boucle penser à vérifier les cas aux bornes.
- si utilisation d'une structure de donnée, pensez aux cas extrèmes (structure pleine, structure vide)

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.