Sensibilisation et mise en pratique de l'approche agile

ACDA – Méthodologie (M3301)

Florent Madelaine

IUT de Sénart Fontainebleau
Département Informatique

Année 2018-2019
Cours 2

Des techniques simples à essayer de mettre en oeuvre

pair programming deux sur une machine, facilite la montée en compétence, en faisant tourner les paires, permet une meilleure homogénéité des pratiques

pomodoro technique du *timer* pour ne pas se disperser tout en restant réactif sur les mails etc.

`https://en.wikipedia.org/wiki/Pomodoro_Technique`

Tester

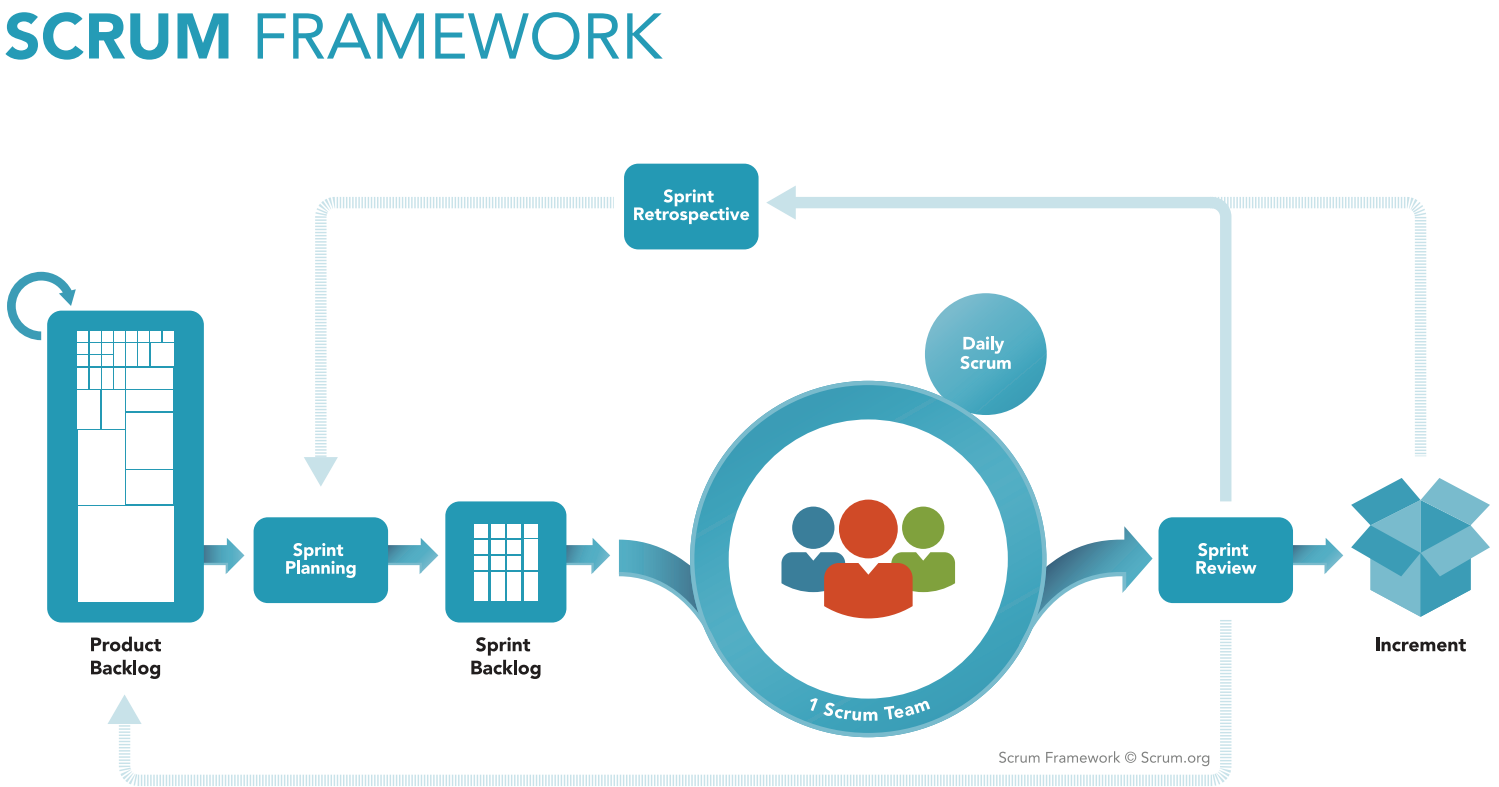
- Il faut mettre en oeuvre des tests.
- C’est aussi important que de coder quelque chose.
- Il faut trouver un équilibre.
- Un bon début : les tests unitaires.
- Version plus poussée : *test driven* / *extreme programming*.

Méthodes agiles, l'exemple de scrum

Vocabulaire

- *Product owner*
- *Product backlog*
- *Sprint*
- *Sprint planning*
- *Sprint Backlog*
- *Sprint review*
- *Sprint retrospective*
- *Daily scrum*
- *Scrum master*
- *Dev Team*

Scrum



Scrum.org

https://www.scrum.org/resources/what-is-scrum

Sprint Retrospective

- Aspect méthodologique / organisationnel.
- En gros 3 heures pour un *sprint* de 4 semaines.
- Qu'est-ce-qui a bien marché?
- Qu'est-ce-qu'on peut améliorer?
- Qu'est-ce-que l'équipe s'engage à mettre en oeuvre en plus ou différemment au prochain sprint.
- Peut être fait à différents moments, par exemple au milieu du sprint.

`https://www.scrum.org/resources/what-is-a-sprint-retrospective`

Sprint Review

- À la fin d'un *sprint*
- En gros 4 heures pour un *sprint* de 4 semaines.
- Concerne le produit sur lequel on travaille.
- En présence des parties prenantes (*stakeholders*).
- Inspection de l'incrément.
- Explication des évolutions du *product backlog*.
- Réflexion, projection et révision à moyen et à long terme (dates, coûts).
- Alimente le prochain *sprint planning*.

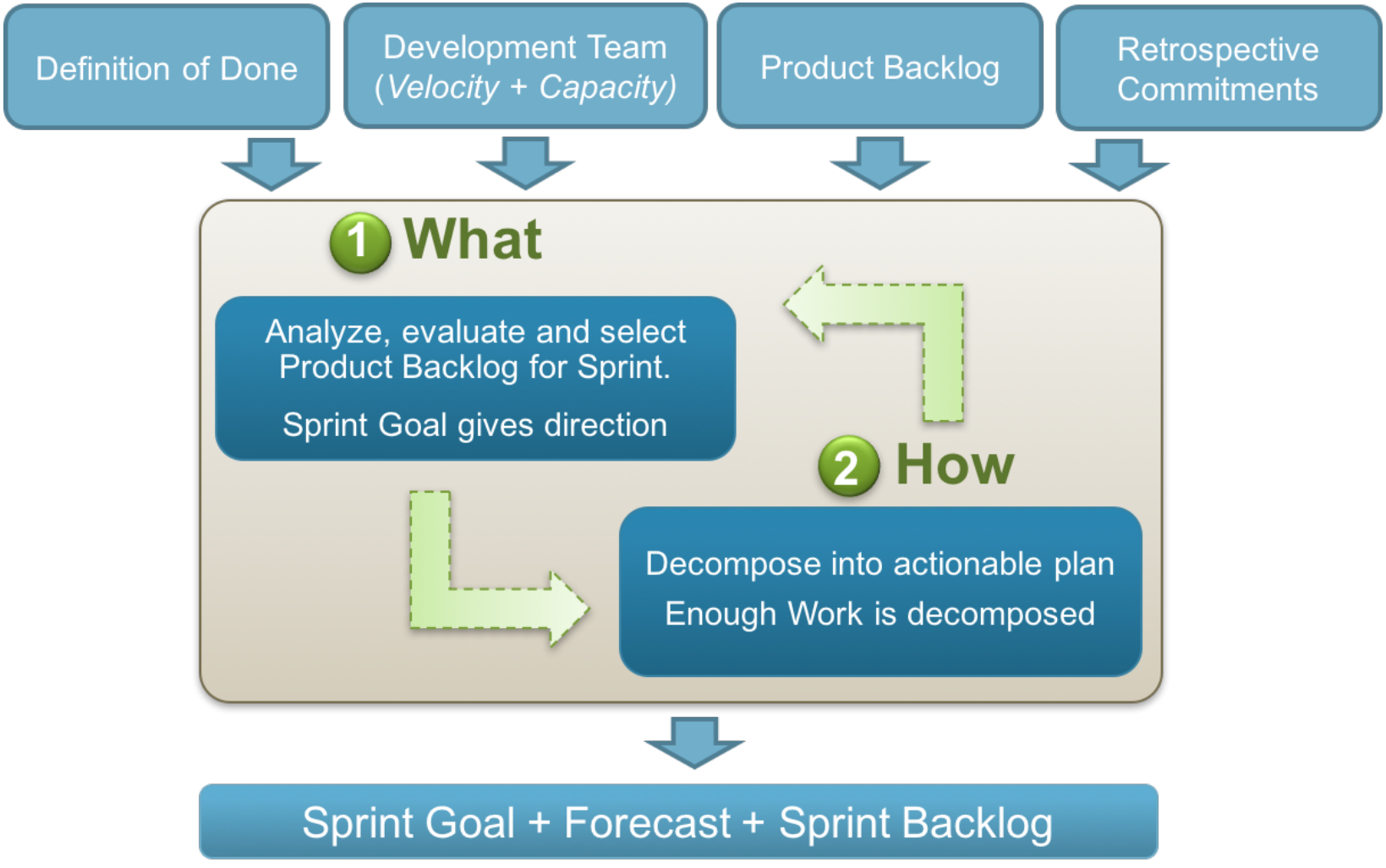
https://www.scrum.org/resources/what-is-a-sprint-review

Sprint planning

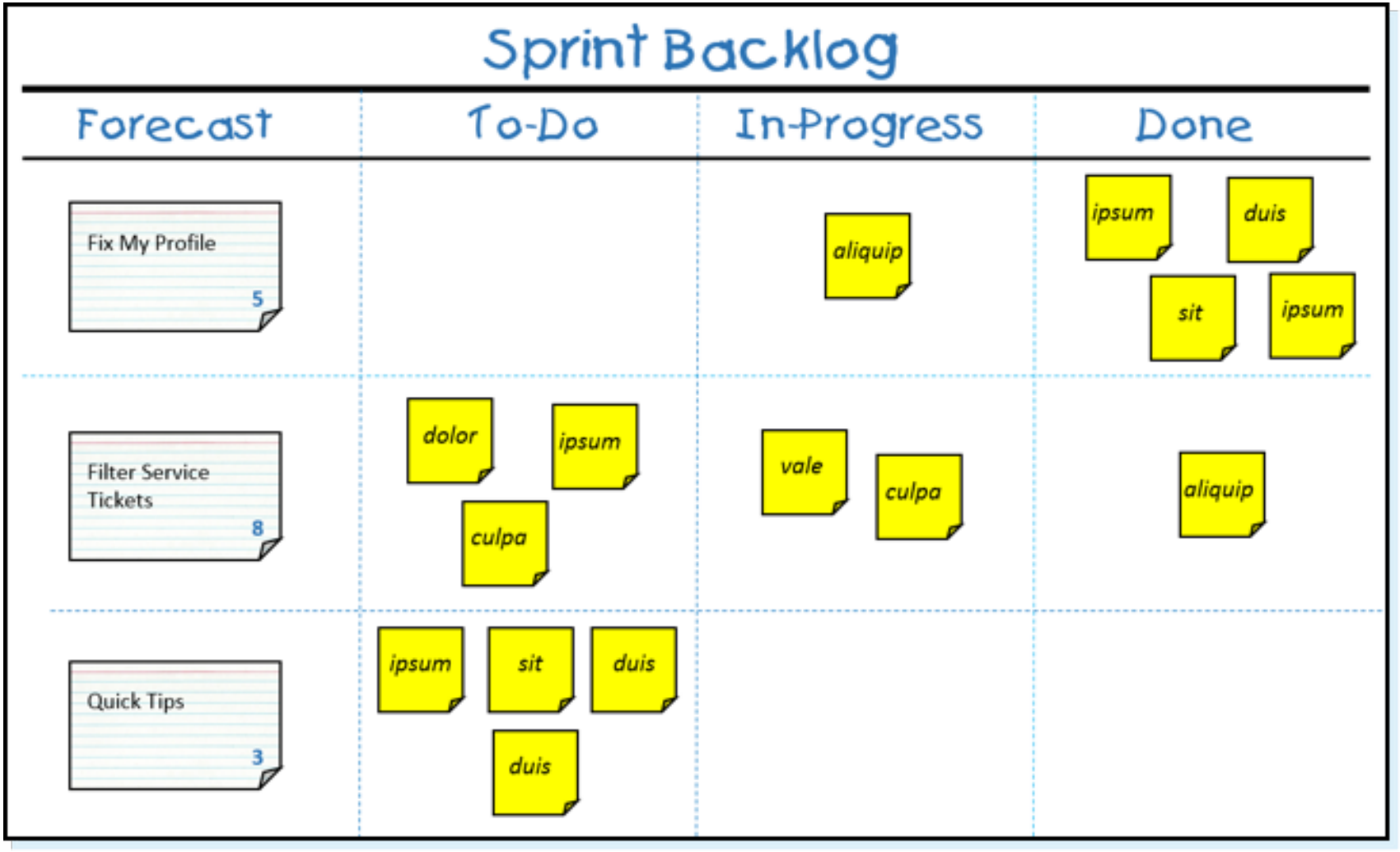
- Au début du *sprint*.
- En gros 8 heures pour un *sprint* de 4 semaines.
- Concerne les tâches spécifiques sur lesquelles on va travailler avec une estimation du temps.
- En gros, on choisit des éléments importants et assez bien décris du *product backlog* qu'on décompose et pour lequel on évalue le temps nécessaire.

`https://www.scrum.org/resources/what-is-sprint-planning`

Sprint planning



Sprint Backlog



Estimation du temps

- Pendant le *sprint planning*, on peut utiliser un jeu simple pour estimer le temps nécessaire pour réaliser une tâche.
- L'équipe dispose de cartes avec des coûts.
- Le *product owner* décrit la tâche.
- Chaque membre de l'équipe fait une enchère cachée.
- On révèle les enchères et on cherche un consensus.
- Typiquement on utilise une échelle inspirée de la suite de Fibonacci.

$0, \frac{1}{2}, 1, 2, 3, 5, 8, 13, 20, 40, 100, ?, \infty$

https://en.wikipedia.org/wiki/Planning_poker

Done

- Pour pouvoir planifier une tâche précise, il faut savoir ce que signifie de la réaliser correctement.
- Il faut que tous les membres de l'équipe soient d'accord sur ce que ça signifie.
- Une manière un peu restrictive mais simple consiste à dire qu'on peut écrire des tests unitaires et que ce qu'on a réalisé passe ces tests.

https://www.scrum.org/resources/what-is-an-increment

https://github.com/junit-team/junit5/wiki/Definition-of-Done

Ce qui est attendu et calendrier

- Rendu / soutenance du projet agile : dernière séance.
- Ce n'est pas grave si vous n'avez pas fini (MoScoW).
- Il me faut un rapport de 6 pages environ avec deux parties a peu près égales.
- La première partie est classique et concerne le travail rendu.
  - Tout ce qui est rendu doit être fonctionnel.
  - J'attends un accès à un git.
  - Votre prototype doit fonctionner sur une machine de l'IUT sans effort particulier de ma part.
- La seconde partie concerne la méthodologie.
  - Il faut idéalement avoir : mis en oeuvre une javadoc et des tests, utilisé un git et fait du pair programming.
  - Je veux des détails sur l'organisation
  - C'est là qu'il faudra montrer le recul que vous avez sur la méthode scrum après usage (en gros c'est le résultat de vos *sprint retrospective*

Les tests

- Je ne vais pas revenir en détail sur tout le cours de première année.
- Je peux vous aider en TP si vous avez des questions techniques.
- Les supports de cours de l'an dernier sont disponibles sur eprel.

Tests unitaires pour nous en java

Un test unitaire c'est quelque chose de très simple qui s'applique à une toute petite partie du code et qui permet à un développeur de se convaincre que cette partie du code fait ce qu'il pense qu'elle doit faire.
En pratique :

- Pour chaque classe un ensemble de tests.
- Pour chaque méthode un ensemble de tests.

En pratique

Pour commencer on peut juste écrire des tests unitaires à la main dans le *main* de chaque classe.
On peut aussi utiliser le mécanisme des assertions en java
Plus généralement, je vous invite à utiliser JUnit qui est l'outil dédié pour les tests unitaires en Java.

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.

Que choisir?

Point de détail sur les exceptions

`IllegalArgumentException` argument KO (jamais correct)

`IllegalStateException` argument KO (maintenant)

Exemple

- Jamais le droit de payer avec ma carte sur un terminal chez un marchand.
- Pas le droit aujourd'hui car plafond dépassé.

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 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.

Exemple

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

Hello World en JUnit

Comment faire tourner les tests?

En pratique

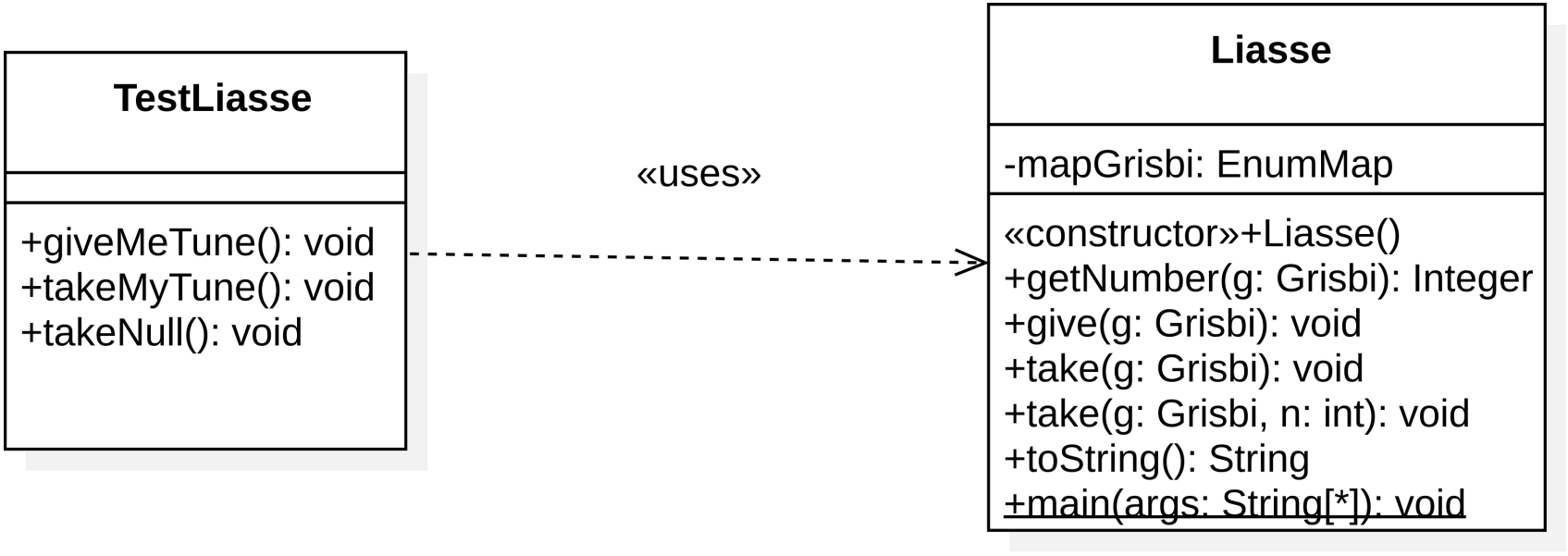
1. compiler les fichiers

   ```
   $ javac *.java
   ```

2. lancer le test

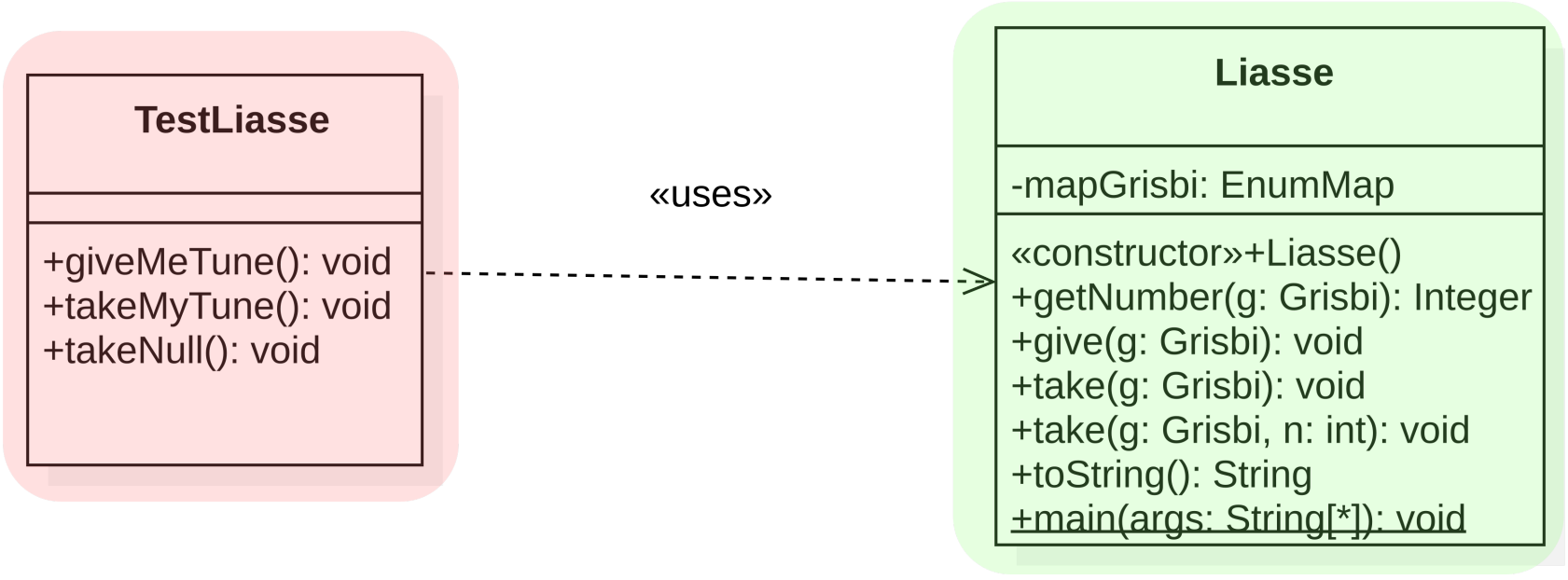
```
$ java org.junit.runner.JUnitCore CalculatorTest0
JUnit version 4.12
.
Time: 0.004

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

```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 de
    @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'avoi
écrire de quelle classe on parle)
        assertEquals(6, sum);
    }
}
```

Élaborer les bon tests

- Nommer les tests avec des noms utiles (ça facilite la compréhension, test34 est moins clair que TestBougerPionSurCaseInterdite).
- On a évoqué préconditions, postconditions et invariants.
- Pour les méthodes complexes, il faut essayer de couvrir tous les cas (couverture de branches) mais avec des tests 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"

Ce qui est attendu et calendrier

- Rendu / soutenance du projet agile : dernière séance.
- Ce n'est pas grave si vous n'avez pas fini (MoScoW).
- Il me faut un rapport de 6 pages environ avec deux parties a peu près égales.
- La première partie est classique et concerne le travail rendu.
  - Tout ce qui est rendu doit être fonctionnel.
  - J'attends un accès à un git.
  - Votre prototype doit fonctionner sur une machine de l'IUT sans effort particulier de ma part.
- La seconde partie concerne la méthodologie.
  - Il faut idéalement avoir : mis en oeuvre une javadoc et des tests, utilisé un git et fait du pair programming.
  - Je veux des détails sur l'organisation
  - C'est là qu'il faudra montrer le recul que vous avez sur la méthode scrum après usage (en gros c'est le résultat de vos *sprint retrospective*