# Ingénierie du logiciel : cours 3

Frédéric Gava

Master ISIN, Université de Paris-Est Créteil

*Cours Ingénierie du logiciel du M2 ISIN*

# Plan

# Plan

# Déroulement du cours

# Introduction

## Bug

Pourquoi ne veut on plus de bug ? Avantages et inconvénients

## Généralités

- Rajouter des annotations/assertions/contrats pour la correction du code
- Correction partielle ou totale et fonctionelle
- Totale $\equiv$ partielle+terminaison
- Fonctionelle $\equiv$ qu'est-ce qui a été calculé ?
- Méthodes formelles $\equiv$ long long long

## Outils

- Krakatoa, ESC/Java2, Key, Jack, VCG, *etc.* Aussi pour C, C#, *etc.*
- Génération d'obligations de preuves pour prouveurs (automatiques ou non) : Simplify, Z3, CVC3, Alt-Ergo, *etc.*

# Notion d'annotation logique (JML)

## Notion de pre post conditions

{P}
code
{Q}

Si $P$ est vrai, alors après l'execution du code, $Q$ doit être vrai :

{x≤y} ou bien {pair(x)} ou bien {0≤x≤y}
x=x+1
{x≤y+1} ou bien {impair(x)} ou bien {0≤x≤y+1}

Donc, si on a {P}code{R} alors {P'}code{R'} si et seulement si P'⇒P et R⇒R'

## Affectation

{P[x←e]} /* *substitution de x par e dans P* */
x=e
{P}

Cela est vrai si P[x←e]⇒x'=e⇒P[x←x']

# Les cas d'annotations (2)

## Exemple, affectation

{x+1>0}
x=x+1
{x>0}
Ici, $P[x \leftarrow e]$ est $x + 1 > 0$ et $P$ est $x > 0$. Alors nous aurons :

x: **int**
H1: $x + 1 > 0$
x0: **int**
H2: $x0 = x + 1$
==========
$x0 > 0$

Une autre manière (parfois plus intuitive) de voir la chose est {P}x=e{Q} tel que Q[x←e]≡P. Alors, on aura {x≥0}x=x+1{x>0} et :

x: **int**
H1: $x \geq 0$
x0: **int**
H2: $x0 = x + 1$
==========
$x0 > 0$

# Les cas d'annotations (3)

## La séquence

Si on a {P1}code1{Q1} et {P2}code2{Q2}
alors on a {P1}code1;code2{Q2} si Q1⇒P2. Par exemple, {x≥0}x=x+1;x=x+1{x>1}. Donc :

x: **int**
H1: $x \geq 0$
x0: **int**
H2: $x0 = x + 1$
x1: **int**
H3: $x1 = x0 + 1$
==========
$x1 > 1$

## Conditionnel (1)

Dans le cas d'un {P}**if** B then i1 **else** i2{S}, il est nécessaire les 2 branches. Nous aurons $\{P \wedge B\} i_1 \{S_1\}$ et $\{P \wedge \neg B\} i_2 \{S_2\}$ tel que $S \equiv S_1 \vee S_2$. Alors nous aurons à prouver :

1. $P \wedge B \implies S$
2. $P \wedge \neg B \implies S$

# Les cas d'annotations (4)

## Conditionnel (2), exemple

$\{0 \leq x < n\}$ **if** $(x < n-1)$ then $x = x+1$ **else** $x = 0$ $\{0 \leq x < n\}$. A prouver :

n: **int**
x: **int**
H1: $0 \leq x$ and $x < n$
H2: $x < n - 1$
x0: **int**
H3: $x0 = x + 1$
==========
$0 \leq x0 < n$

n: **int**
x: **int**
H1: $0 \leq x$ and $x < n$
H4: $x \geq n - 1$
x0: **int**
H5: $x0 = 0$
==========
$0 \leq x0 < n$

## Conditionnel (3)

Notons, que nous avons fait un raccourci pour :
$\{0 \leq x < n\}$
**if** $(x < n-1)$ then $\{1 \leq x+1 < n\} x = x+1 \{1 \leq x < n\}$
**else** $\{0 \leq 0 < n\} x = 0 \{0 \leq x < n\}$
$\{0 \leq x < n\}$

# La notion d'invariant (1)

## Introduction

Les boucles (while, for, loop, *etc.*) nécessitent une annotation particulière : un **invariant**. En effet, on ne sait pas combien combien d'itérations aura la boucle et donc ce qu'elle modifiera :

**while** B **do**
{invariant P}
 code
end

**IMPORTANT** : L'invariant P est une propriété qui est vrai au début de la boucle (avant d'y "rentrer") et à chaque itération. Ceci garantie que la propriété sera aussi vrai à la fin de la boucle.

## Exemple

$\{i{=}0 \wedge n \geq 0 \wedge s{=}0\}$
**while** $(i{<}n)$
 $\{\text{invariant } 2{*}s{=}i{*}(i{+}1)\}$
 $i{=}i{+}1;\ s{=}s{+}i$
end
$\{2{*}s{=}n{*}(n{+}1)\}$

# La notion d'invariant (2)

Nous aurons les buts suivants à prouver :

## Loop invariant initialy holds

n: **int**
H1: $n \geq 0$
==========
$(2 \times 0 = 0 \times (0 + 1))$ and $0 \leq n$

## The loop invariant preserved

n: **int** i: **int** s: **int**
H1: $n \geq 0$
H2: $2 \times s = i \times (i + 1)$ and $i \leq n$
H3: $i < n$
i0: **int**
H4: $i0 = i + 1$
s0: **int**
H5: $s0 = s + i0$
==========
$(2 \times s0 = i0 \times (i0 + 1))$ and $i0 \leq n$

## The postcondition holds

n: **int** i: **int** s: **int**
H1: $n \geq 0$
H2: $2 \times s = i \times (i + 1)$ and $i \leq n$
H6: $i \geq n$
==========
$2 \times s = n \times (n + 1)$

# Correction total

## Terminaison : variant

Pour l'instant, nous pouvons prouver qu'un programme vérifie bien une propriété. Mais pas qu'il termine... Pour cela, nous rajoutons aux boucles un **variant**. C'est une mesure **strictement décroissante** et toujours **supérieur à 0**.

## Exemple

{invariant 2∗s=i∗(i+1) variant n−i}. Ce qui nous donnera :

n: **int**
H1: $n \geq 0$
i: **int**
s: **int**
H2: $2 \times s = i \times (i + 1)$ and $i \leq n$
H3: $i < n$
i0: **int**
H4: $i0 = i + 1$
s0: **int**
H5: $s0 = s + i0$
==========
$(0 \leq n - i)$ and $(n - i0 < n - i)$

# Label et ghost code

## Label

Dans les annotations logiques, il est parfois utile d'exprimer la valeur à un endroit donner. On insére un label L:i=i+1 puis on utilise i@L c'.-à-d. la valeur de i avant l'affectation.

On écrit aussi i@ pour parler de la valeur i avant la méthode/procédure (en tant que paramètre).

## Ghost code

**Astuce**. Des fois, il est impossible d'exprimer la propriété logique sans faire des calculs supplémentaires. On ajoute ces calculs dans le code et on parle alors de code fantôme : ce code peut lire les valeurs du programme en cours d'exécution mais PAS les modifier. On utilise ensuite les valeus du code fantôme dans les annotations.

# Déroulement du cours

# Premiers examples (1)

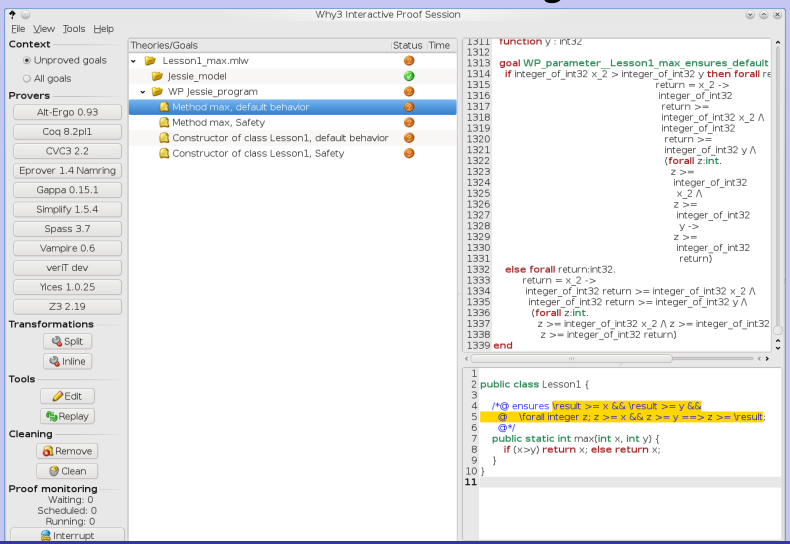
## Première méthode

```
public class Lesson1 {
    /*@ ensures \result >= x && \result >= y &&
      @ \forall integer z; z >= x && z >= y ==> z >= \result;
      @*/
    public static int max(int x, int y) {if (x>y) return x; else return x /* erreur */;}
}
```

"ensures"⇒post-condition de la méthode ; "result" la valeur de retour

## Utilisation

`krakatoa Lesson1.java`. On aura la fenêtre suivante :



# Premiers examples (2)

## Avec invariant

```
/*@ requires x >= 0;
  @ ensures \result>=0 && \result*\result<=x && x<(\result+1)*(\result+1);
  @*/
public static int sqrt(int x) {
    int count = 0, sum = 1;
    /*@ loop_invariant
      @ count>=0 && x>=count*count && sum==(count+1)*(count+1);
      @ loop_variant x-sum; */
    while (sum <= x) {
        count++;
        sum = sum + 2*count+1; }
    return count; }
```

"requires" $\Rightarrow$pre-condition ; "loop_invariant" $\Rightarrow$invariant

## Pourquoi ?

$$\sum_{i=0}^{k-1} 2i+1 = k^2 \Rightarrow \sqrt{(n)} \text{ est le plus petit } k \text{ tel que } \sum_{i=0}^{k} 2i+1 \geq n$$

# Sûreté d'exécution (1)

## Qu'est-ce ?

On trouve aussi les erreurs suivantes possibles : division-par-zero, arithmetic Overflow, Null pointer dereferencing, Out-of-bounds array access.

## Et pour notre example ? Arithmetic Overflow !

Et bien, il est possible d'avoir un débordement arithmétique pour un gros x : sum+2∗count+1 peut dépasser $2^{31}$. 2 solutions possibles :

1. Rajouter une pre-condition sur la valeur max de x
2. Ignorer le problème avec le pragma
   *//@+ CheckArithOverflow = no*

# Sûreté d'exécution (2)

## Out-of-bounds array access

```
public class Arrays {
 /*@ requires t!=null && t.length >= 1;
    @ ensures 0<=\result<t.length
    @ && \forall integer i; 0<=i<t.length==>t[i]<=t[\result]; @*/
 public static int findMax(int[] t) {
  int m = t[0]; int r = 0;
  /*@ loop_invariant
     @ 1 <= i && i <= t.length && 0 <= r && r < t.length &&
     @ m == t[r] && \forall integer j; 0<=j && j<i ==> t[j]<=t[r];
     @ loop_variant t.length-i; @*/
  for (int i=1; i < t.length; i++) {if (t[i] > m) {r = i;m = t[i];}}
       return r;
```

t==NULL assure l'existance d'un max. Plus tard : une exception.

## Prédicats logiques

```
/*@ predicate is_max{L}(int[] t,integer i,integer l) = 0<=i<l
   @ && \forall integer j; 0 <= j < l ==> t[j] <= t[i]; @*/
```

# L'avant, les labels et le lemmes

## Modification d'un tableau

```
/*@ requires t != null;
      @ ensures
      @ \forall integer i; 0 < i < t.length ==> t[i] == \old(t[i-1]); */
public static void shift(int[] t) {
 /*@ loop_invariant
    @ j < t.length &&
    @ (\forall integer i; 0 <= i <= j ==> t[i] == \at(t[i],Pre)) &&
    @ (\forall integer i; j < i < t.length ==> t[i] == \at(t[i-1],Pre));
    @ loop_variant j; */
 for (int j=t.length-1 ; j > 0 ; j--) {t[j] = t[j-1];}
```

Dans les pre/post conditions, "\old" permet de décrire la valeur à l'entrée de la méthode. \at(e,Label) est pour la valeur à un label. Ici "Pre" est pour la valeur à l'entrée de la méthode.

## Lemma et axiom

```
/*@ lemma distr_right: \forall integer x y z; x*(y+z) == (x*y)+(x*z); */
/*@ axiom comm: \exists integer x y; x*y==y*x; */
```

# Programmation objet et constructeurs

```
class NoCreditException extends Exception { public NoCreditException(); }
public class Purse {
  public int balance;
  //@ invariant balance_non_negative: balance >= 0;

  /*@ assigns balance;
    @ ensures balance == 0; */
  public Purse() {balance = 0;}

  /*@ requires s >= 0;
    @ assigns balance;
    @ ensures balance == \old(balance)+s; */
  public void credit(int s) {balance += s;}

  /*@ requires s >= 0 && s <= balance;
    @ assigns balance;
    @ ensures balance == \old(balance) - s; */
  public void withdraw(int s) {balance -= s; }
```

On garantit que la valeur de balance soit toujours positive.

“assigns” indique que la valeur de balance (attribut) est modifiée.

# Les exceptions

```
class NoCreditException extends Exception {
  /*@ assigns \nothing; */
  public NoCreditException(){} }

public class Purse2 {
 /*@ public normal_behavior
   @ requires s >= 0;
   @ assigns balance;
   @ ensures s <= \old(balance) && balance == \old(balance)−s;
   @ behavior amount_too_large:
   @ assigns \nothing;
   @ signals (NoCreditException) s > \old(balance) ;
   @*/
  public void withdraw2(int s) throws NoCreditException {
    if (balance >= s) {balance = balance − s;}
     else {throw new NoCreditException();}
```

“signals” permet d’exprimer les cas d’exception. Avec “behavior”, on donne un nom à ce cas d’exception. Un “behavior” est le nom d’un cas. On pourrait avoir “behavior comportement_normal: ...”

# GhostBuster

Comme déjà expliqué, il faut parfois, pour la preuve, du code fantôme :

```
/* Déclaration */
//@ ghost integer a = 1, b = 0, c = 0, d = 1;
integer toto;
...
/* Code */
//@ ghost a = c-d;

/* Code non-valide */
//@ ghost toto=a
toto=a
```

Ces variables peuvent être utilisées dans les annotations. Mais il interdit d'utiliser les variables fantômes dans les vrai variables.

# Compléments

## Limitations

- Pour l'instant pas de génériques *p. ex.* Set<String>
- Utilisation des casts et object limité
- Pas de cast implicite (*p. ex.* integer à Integer)
- Les exceptions "NullPointerException" et "ArrayOutOfBoundsException" doivent forcement être empêchées, ce qui n'est pas le cas en Java

## En vrac

- On peut écrire dans le "*//@ assert prop*" pour tester "prop"
- Spécifier les affectations d'une boucle "*/*@ loop_assigns tset @*/*"
- invariant de classe $\Rightarrow$ propriété vrai après chaque chaque méthode ou constructeur

# Exercices

## Exo 1

Écrire et prouver une méthode statique qui trouve le plus petit élément d'un tableau d'entiers.

## Exo 2

On suppose une classe de Compte (comportant la somme et le nom d'une personne). On suppose que les comptes ne peuvent pas être dans le rouge. On suppose un tableau de comptes. Prouver une méthode qui prélève 100 euros à chaque compte : si un compte n'a pas assez d'argent, alors une exception est levée donnant le nom de la personne sans le sous.

# A la semaine prochaine