Programmation structurée

Université de Nice - Sophia Antipolis
Richard Grin
Version 1.1.3 – 1/10/11

Plan

- Algorithme
- Principe de base de la programmation structurée
- Les structures
- Traduction en Java

Objectif

- Maîtriser la complexité lors de l'écriture de programmes/algorithmes comportant de nombreuses instructions

Algorithme

Algorithme

- L'enchaînement des opérations élémentaires qui permettent de résoudre un problème s'appelle un algorithme
- N'importe quelle machine ou individu qui suit à la lettre l'algorithme obtiendra le bon résultat, même s'il ne comprend pas ce qu'il fait

Programme et algorithme

- Un programme est écrit dans un langage qui peut être exécuté par une machine
- Un algorithme est écrit dans un pseudo-langage, souvent proche d'un langage informatique mais débarrassé de détails techniques liés aux langages informatiques ou à la machine, qui gênent la compréhension du cheminement logique

Exemple : algorithme d'Euclide

- On cherche le PGCD de deux entiers naturels *a* et *b* (*a* > *b*)
- Description informelle de l'algorithme :

1. Si b = 0, le PGCD est a ; FIN
2. sinon,
   calculer r le reste de la division de *a* par *b*,
   puis remplacer *a* par *b*, puis *b* par *r,*
   puis recommencer l'étape précédente

Exemple de calcul de PGCD

- PGCD de 18 et de 14 (a = 18 et b = 14)
- b ≠ 0 ; r = 18 modulo 14 = 4 ; a = 14 ; b = 4
- b ≠ 0 ; r = 14 modulo 4 = 2 ; a = 4 ; b = 2
- b ≠ 0 ; r = 4 modulo 2 = 0 ; a = 2 ; b = 0
- b = 0 donc on s'arrête
- 2 est le PGCD de 18 et de 14

Transcription plus formelle

- Il faut maintenant transcrire cette description informelle dans les termes de la programmation structurée
- Ça n'est pas toujours évident
- Il peut exister plusieurs transcriptions possibles

Exemple de pseudo-langage

```
Entrer a, b
tant que (b ≠ 0) répéter {
  r := a modulo b
  a := b
  b := r
}
Afficher a
```


Preuve de programme/d'algorithme

- Pour les programmes les plus simples il est possible de démontrer que le résultat de l'algorithme est correct
- Le plus souvent la preuve est trop complexe et on se contente d'effectuer des tests sur le programme écrit dans le langage informatique
- La preuve de programme s'appuie essentiellement sur les invariants de boucle (voir section dans la suite de ce cours)

Complexité d'un algorithme

- Il existe le plus souvent plusieurs algorithmes pour résoudre un problème
- La complexité d'un algorithme mesure l'efficacité d'un algorithme en temps de calcul ou en quantité de mémoire
- On recherche des algorithmes qui nécessitent moins de temps de calcul ou (souvent incompatible) moins d'espace mémoire

Calcul de la complexité

- La complexité dépend le plus souvent de la taille *n* des données à traiter
- Le calcul de la complexité d'un algorithme donne une évaluation du nombre d'opérations significatives en fonction de la taille *n*
- Par exemple, on dira qu'en moyenne (notion probabiliste car dépend des données) le tri par insertion de n éléments nécessite $(n^2 - n) / 4$ opérations de comparaison entre éléments à trier

Notation O()

- La complexité est surtout importante pour les grands nombres (toujours rapide si petit nombre)
- Pour les grands nombres n est très petit par rapport à n^2
- On dit que le tri par insertion est en $O(n^2)$, ce qui donne un ordre de grandeur pour l'efficacité du tri par insertion
- Les meilleurs algorithmes de tri sont en $O(n \log n)$

Classes de complexité

- Les algorithmes les plus rapides sont en O(1) : ne dépend pas de la taille des données à traiter
- Algorithme linéaire : O(n)
- Logarithmique : O(log n), meilleur que O(n)
- Moins bons :
  - Polynomial : en $O(n^p)$ pour un certain p
  - Exponentiel : en O(exp n) (catastrophique !)

Exemple

- Pour trier 30.000 éléments, il faudra environ
  - 900.000.000 comparaisons avec le tri par insertion qui est en $O(n^2)$
  - 1.600.000 comparaisons avec un tri « quicksort » qui est en $O(n \log n)$

Principe de la programmation structurée

Principe général de base

- Un programme peut comporter des milliers, et même des millions d'instructions
- Impossible pour un développeur d'avoir une vision d'ensemble et de gérer cette complexité
- Il faut décomposer le programme en éléments plus simples à appréhender
- Le discours de la méthode de Descartes : « *Diviser chacune des difficultés que j'examinerai en autant de parcelles qu'il se pourrait, et qu'il serait requis pour les résoudre.* »

Programmation structurée

- Une mauvaise décomposition peut rendre le problème encore plus difficile à résoudre
- La programmation structurée permet de bien décomposer un programme complexe en parties plus simples à comprendre

Principe

- Un programme est décomposé en modules
- Chaque module a une des structures de la programmation structurée
- En plus de permettre une bonne décomposition, ce principe offre un bon guide lors de l'écriture du programme puisque les structures de la programmation structurée sont simples et peu nombreuses

Spécification des modules

- Lorsqu'on introduit un nouveau module dans la décomposition, il faut le définir de façon précise
- En effet, lorsqu'on cherchera à le décomposer à son tour, il ne faut pas avoir à regarder ce qui l'entoure
- Il faut donner une spécification du module qui décrit
  - la situation de départ (les pré-conditions)
  - ce qui sera obtenu à la fin de l'exécution du module (les post-conditions)

Spécification des modules

- La spécification dit ce que l'on veut obtenir mais pas comment l'obtenir
- Le « comment » sera donné lorsque le module sera décomposé à son tour

Programmation par contrat

- Le contrat du module est le suivant :
  si on fournit au module des données qui satisfont aux pré-conditions, le module assure qu'il rendra un résultat qui satisfera aux post-conditions

Exemple de spécification

- Pour une méthode qui insère un nombre entier `n` dans une liste de nombres entiers `l` déjà triée par ordre croissant
  `void inserer(Liste l, int n)`
- Pré-condition :
  la liste d'entiers `l` est triée par ordre croissant
- Post-condition :
  le nombre `n` est ajouté à la liste `l`
  la liste `l` est triée par ordre croissant

Les structures de la programmation structurée

Structures de la programmation structurée

- Chaque module a une des ces 3 structures :
  - suite de modules
  - alternative
  - répétition

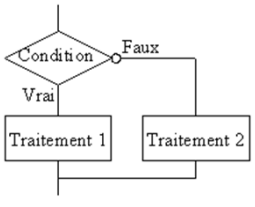
Suite

- Faire Traitement 1 puis Traitement 2
- Chaque module a lui-même une des structures de la programmation structurée
- C'est le système des poupées russes qui s'emboîtent les unes dans les autres
- Se généralise en une suite de n traitements (n > 2)

Exemple

Saisir une fraction (x / y)
Réduire la fraction (on obtient a / b)
m = a * b

Alternative



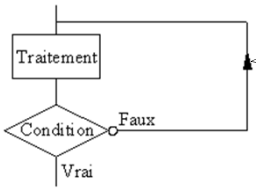
- Si la condition est vraie, faire Traitement 1, sinon, faire Traitement 2

Exemple

- ```
  si (x > y) {
    max = x;
  }
  sinon {
    max = y;
  }
  ```
- Variante (pour les cas où on ne fait rien si la condition est fausse) :

```
si (x > max) {
  max = x;
}
```
```

Répétition « jusqu'à »



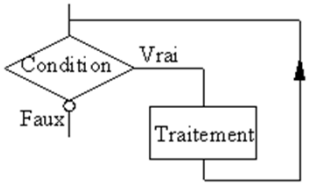
L'ordre « naturel » d'exécution est de haut en bas et de gauche à droite ; on ajoute une flèche pour indiquer un autre ordre

- Répéter Traitement
  jusqu'à ce que la condition (« de fin ») soit vraie

Exemple

```
répéter {
  x = x + 10;
} jusqu'à (x < y)
```


Répétition « tant que »

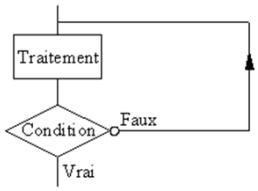


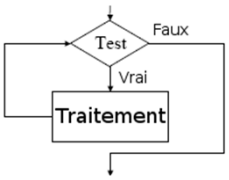
- Tant que condition est vraie,
  répéter Traitement

Exemple

```
tant que (x < y) répéter {
  x = x + 10;
}
```


Différence importante entre « jusqu'à » et « tant que »





- Avec jusqu'à, le traitement est exécuté au
  moins une fois

Exemple

```
répéter {
  x = x + 10;
} jusqu'à (x >= y)
```

On commence par augmenter x de 10, même si x >= y au début

Répétition « pour »

- Cas particulier de la boucle « tant que »
- Utilisé quand on sait au moment de l'écriture de l'algorithme combien de fois la boucle va être parcourue
- Pour i de 1 à n répéter Traitement
- i s'appelle la variable de boucle

Exemple

```
pour i de 1 à 20 répéter {
  s = s + i;
}
```


Complément sur les invariants de boucle

- Cette section est réservée aux curieux qui souhaiteraient avoir une idée de la façon de prouver un algorithme en utilisant des invariants de boucle
- Il peut être intéressant de prouver des parties de programmes si on veut être certain d'avoir bien pris en compte tous les cas de figure

Rappel sur les boucles

- Essentiellement 2 types de boucle :
  - Tant que (condition) répéter traitement
  - Répéter traitement jusqu'à (condition_de_fin)
- On peut se passer du 2ème type qui peut être remplacé par :
  traitement suivi de tant que(non condition_de_fin) répéter traitement
- La suite de cette section ne considérera que les boucles « tant que »
- Appelons « condition de répétition » la condition d'une boucle « tant que »

Invariant de boucle

- Un invariant de boucle est une expression booléenne toujours vraie au début et à la fin d'une certaine boucle
- Pour démontrer qu'une expression booléenne est un invariant de boucle, il suffit de démontrer que l'invariant est vrai juste avant la boucle et que
  - si l'expression est vraie au début de la boucle,
  - et si la condition de répétition est vraie,
  - alors l'expression est vraie à la fin de la boucle

Invariant de boucle

```
Entrer a, b
a0 = a; b0 = b
tant que (b ≠ 0) répéter {
  r := a modulo b
  a := b
  b := r
}
Afficher a
```

Invariant de boucle : les diviseurs de a0 et de b0 sont les mêmes que ceux de a et de b.
a0 et b0 ont été introduits pour faciliter l'expression de l'invariant de boucle

Preuve de la boucle

- Il faut prouver que la boucle se terminera, ce qui est évident puisque b est un entier positif qui décroît (définition du reste de la division entière : a = bq + r avec r < b)
- A la fin on doit trouver que a est bien le PGCD des 2 nombres du départ, compte tenu de l'invariant de boucle et que la condition de répétition est fausse (puisqu'on est sorti de la boucle)
- C'est vrai car l'ensemble des diviseurs du a final est l'ensemble des diviseurs des a et b du départ

Traduction en Java

Suite de traitements

- Suite d'instructions élémentaires :

```
int x = 3;
int y = x + 8;
```

- Bloc avec des accolades
- Un traitement peut être représenté en Java par un appel de méthode (étudié plus loin dans le cours)

Alternative

-
```
if (x >= 0) {
  x = x + 1;
}
else {
  x = -x + 1;
  y++;
}
```

- Variante :

```
if (x >= 0) {
  x = x + 1;
}
```


Alternative

- Possible de ne pas mettre d'accolades s'il n'y a qu'une seule instruction dans le if ou le else
-
```
if (x >= 0)
  x = x + 1;
else
  x = -x + 1;
```

-
```
if (x >= 0)
  x = x + 1;
```

- Pas recommandé, car source d'erreurs !

Alternative

- Lorsque plusieurs **`if`** sont emboîtés les uns dans les autres, un bloc **`else`** se rattache au dernier **`if`** qui n'a pas de **`else`**

Exemple

```
x = 3;
y = 8;
if (x == y)
  if (x > 10)
    x = x + 1;
else
  x = x + 2;
```

Quelle valeur pour x à la fin de ce code ?

Facile de se tromper si on ne met pas d'accolades, surtout si on indente mal son code !

Exemple

```
x = 3;
y = 8;
if (x == y) {
  if (x > 10) {
    x = x + 1;
  }
}
else {
  x = x + 2;
}
```

Mettre des accolades évite de se tromper !

Expressions booléennes

- Ce sont des expressions dont le type Java est **`boolean`** ; elles peuvent avoir 2 valeurs : **`true`** (vrai) ou **`false`** (faux)
- Elles sont utilisées par **`if`** ou par les répétitions (comme condition pour refaire une boucle)
- Opérateurs « et », « ou », « non » ; en Java « **`&&`** », « **`||`** », « **`!`** »

Exemples

- **`x >= 0`**
- **`(x != 0) && (y / x > 2.3)`** // raccourci
- **`(x == 0) || (y / x > 2.3)`** // raccourci
- **`! ((x == 0) || (y / x > 2.3))`**

Distinction de cas suivant une valeur

- Java a une instruction qui permet de simplifier du code avec des if emboîtés dans le cas où la condition dépend de la valeur d'une expression

switch

```
switch(expression) {
case val1: instructions;
           break;
...
case valn: instructions;
           break;
default:   instructions;
}
```

Attention, sans break, les instructions du cas suivant sont exécutées !

expression est de type **char**, **byte**, **short**, ou **int** (ou **String** depuis le JDK 7)

- S'il n'y a pas de clause **default**, rien n'est exécuté si *expression* ne correspond à aucun **case**

Exemple de switch

```
char lettre;
int nbVoyelles = 0, nbA = 0,
    nbT = 0, nbAutre = 0;
. . .
switch (lettre) {
case 'a' : nbA++;
case 'e' :                  // pas d'instruction !
case 'i' : nbVoyelles++;
           break;
case 't' : nbT++;
           break;
default :  nbAutre++;
}
```

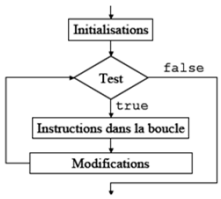

Répétition jusqu'à

- ```
  do {
    x++;
  } while (x < 8)
  ```
- Quelle valeur aura x après ce code ?

Exemples en Java – tant que

- ```
  while (x < 8) {
    x++;
  }
  ```
- Quelle valeur aura x après ce code ?

Répétition « pour »



- En Java :

```
for (initialisations ; test ; modifications) {
  instructions dans la boucle
}
```


Répétition « pour »

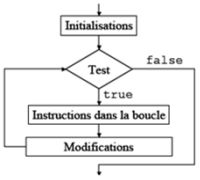
- La boucle « for » de Java est plus générale que « pour » ; la boucle for peut faire tout autre chose que d'incrémenter la variable de boucle
- Le plus souvent la variable de boucle est déclarée à l'intérieur de l'instruction d'initialisation de for (sa portée est alors limitée à la boucle) : **for (int i = 0; … ; …)**

Exemple typique de répétition « pour »

initialisations test modifications

```
for (int i = 0; i < 3; i++) {
   instructions dans la boucle
}
```

- Question : combien de fois les instructions dans la boucle seront-elles exécutées ?



Exercice

- Reconstituer la structure « pour » à l'aide des autres structures de la programmation structurée
- Utiliser les « poupées russes » en donnant plusieurs étapes, chaque étape n'utilisant qu'une seule structure de la programmation structurée

Saut depuis une boucle en Java

- Pas prévu dans la programmation structurée ; il faut jouer sur la condition de sortie pour sortir d'une sortie en son milieu
- Prévu dans la plupart des langages informatique car bien pratique
- En Java, 2 instructions : **`break`** et **`continue`**
- **`break`** fait sortir de la boucle ; l'exécution se poursuit juste après la fin de la boucle
- **`continue`** reste dans la boucle mais saute la fin de la boucle en cours

Exemple de **`continue`** et **`break`**

```
int somme = 0;
for (int i = 0; i < 10; i++) {
  if (i % 2 == 0) continue;
  if (somme > 4) break;
  somme = somme + i;
}
System.out.println(somme);
```

Qu'affiche ce code ?