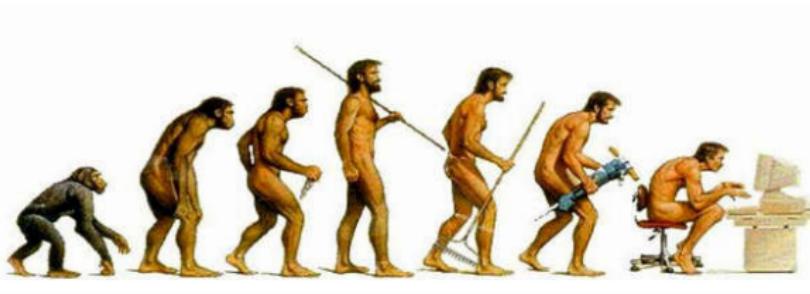




Module d'informatique

Programmation et algorithmique

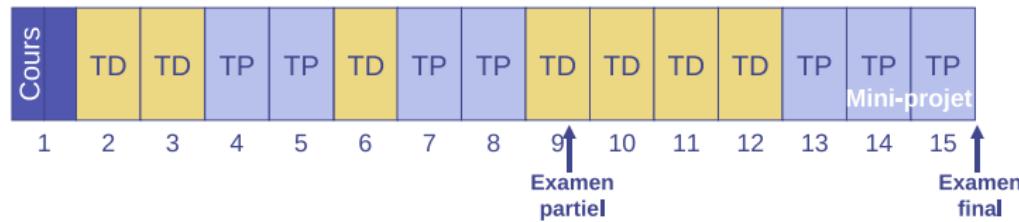
Avant propos



L'informatique à l'ENSMM

- Première année (semestre bleu)
 - Programmation et algorithmique (C)
- Deuxième année (semestre vert, parcours EAO)
 - Programmation Orientée Objet (Java)
- Troisième année (modules transverses)
 - Systèmes d'informations et application Web (SQL, HTML, PHP)
 - Méthodes d'Optimisation et de Décision

Organisation du module

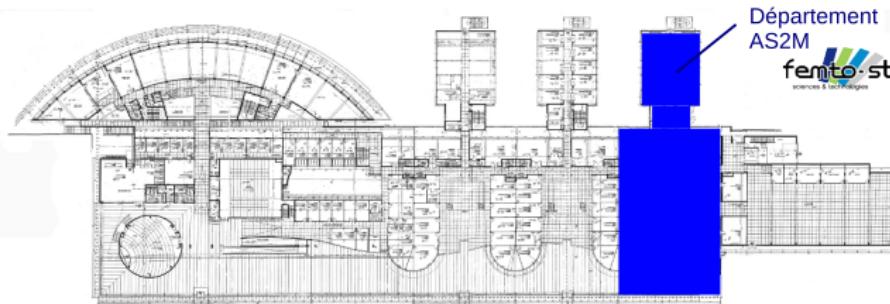


Évaluation :

- 1/3 examen partiel
- 1/3 examen final
- 1/3 mini-projet

Équipe pédagogique

- Enseignants-chercheurs au département AS2M de l'institut FEMTO-ST
 - Guillaume Laurent
 - Jean-Marc Nicod
 - Nadine Piat
 - Emmanuel Piat
 - Emmanuel Ramasso
 - Christophe Varnier



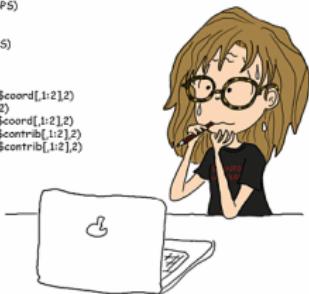
Petits messages...

```
p1<-read.csv("~/Users/lapin/Desktop/Classeur7.csv", sep=",", dec=".","row.names=1")
nrow(p1)
ncol(p1)

names(p)
is.numeric(m$PS)

summary(m$PS)

pa1<-PCA(p1)
round(pa1$var$coord[,1:2],2)
round(pa1$eig,2)
round(pa1$ind$coord[,1:2],2)
round(pa1$ind$contrib[,1:2],2)
round(pa1$var$contrib[,1:2],2)
```



VS



Source : <http://lapinobservateur.over-blog.com/>

Plan du cours

Partie 1 : Introduction à l'algorithmique

Partie 2 : Introduction à la programmation structurée

Partie 1 : Introduction à l'algorithmique

- 1** Définition et historique
- 2** Notion de variable
- 3** Instructions et structures de contrôle
- 4** Conclusion

1 Définition et historique

1 Définition et historique

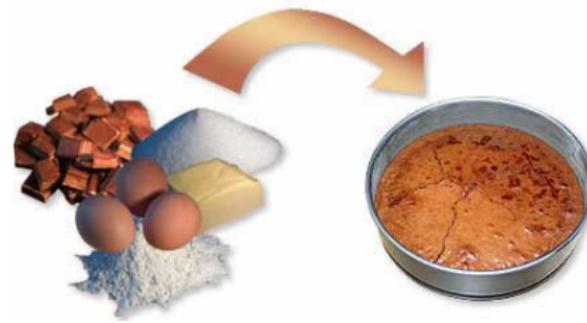
2 Notion de variable

3 Instructions et structures de contrôle

4 Conclusion

Qu'est-ce qu'un algorithme ?

Qu'est-ce qu'un algorithme ?



L'antiquité

L'antiquité

-1900

Époque babylonienne



Descriptions d'algorithmes pour des calculs
concernant le commerce et les impôts

L'antiquité

-1900

-300

Époque babylonienne



Descriptions d'algorithmes pour des calculs concernant le commerce et les impôts

Euclide



Algorithme permettant de trouver une unité de mesure commune à deux longueurs, c-à-d le PGCD (livre VII des Éléments d'Euclide)

Naissance du concept d'algorithme

Naissance du concept d'algorithme

783 - 850

Muhammad Al Khawarizmi



Mathématicien perse, auteur d'un ouvrage qui décrit des méthodes systématiques de calculs algébriques

Naissance du concept d'algorithme

783 - 850

1080 - 1160

Muhammad Al Khawarizmi



Mathématicien perse, auteur d'un ouvrage qui décrit des méthodes systématiques de calculs algébriques

Adelard de Bath



Mathématicien britannique, introduit le terme latin de *algorismus*. Ce mot donnera *algorithme* en français en 1554

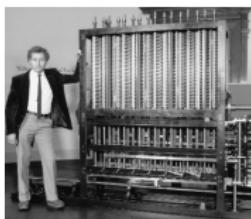
Formalisation du concept d'algorithmique



Formalisation du concept d'algorithmique

1815 - 1852

Ada Lovelace



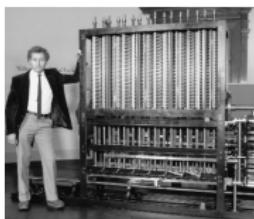
Mathématicienne britannique, propose une méthode très détaillée pour calculer les nombres de Bernoulli considérée comme le premier programme informatique pour la machine à différence de Charles Babbage

Formalisation du concept d'algorithmique

1815 - 1852

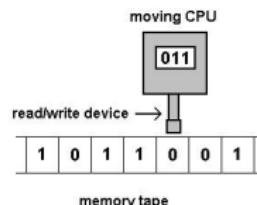
1912 - 1954

Ada Lovelace



Mathématicienne britannique, propose une méthode très détaillée pour calculer les nombres de Bernoulli considérée comme le premier programme informatique pour la machine à différence de Charles Babbage

Alan Turing



Mathématicien britannique, formalise les concepts d'algorithmique et de calculabilité au travers l'invention d'une machine virtuelle : la machine de Turing. Tout problème de calcul basé sur une procédure algorithmique peut être résolu par une machine de Turing (thèse Church-Turing)

Formalisation du concept d'algorithmique

Définition (Algorithm)

Un algorithme est la formalisation logique, claire et complète d'une suite d'actions qui permettent de passer des données d'un problème au résultat. Cette formalisation nécessite l'utilisation d'un langage simple, non ambigu et indépendant de tout matériel et de tout logiciel.

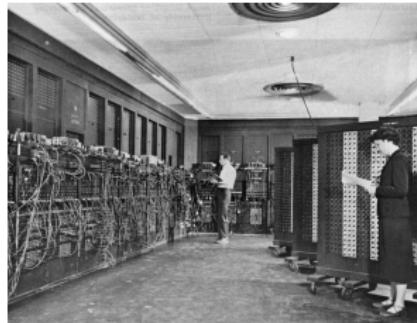
Premiers langages machines



Premiers langages machines

1943 ENIAC

ENIAC



- Premier ordinateur électronique (à tubes) construit pour être complet au sens de Turing.

Premiers langages machines

1943 ENIAC

1954 IBM 704

IBM 704



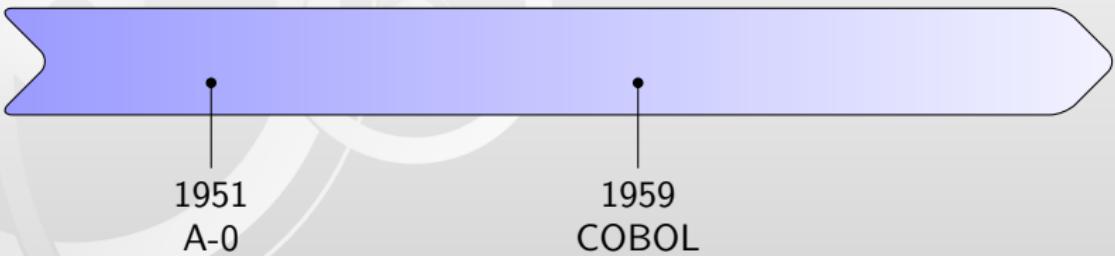
Adresse	Contenu de la mémoire	
00000000	01010101	1ère instruction
00000001	11111111	2ème instruction
00000010	00000000	
00000011	11001100	
00000100	00110011	début de la 3ème instruction

- Programmes écrits avec les instructions de base de la machine (en binaire)
- Programmes difficiles à comprendre et à modifier

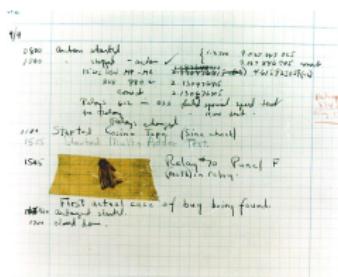
Premiers langages symboliques



Premiers langages symboliques

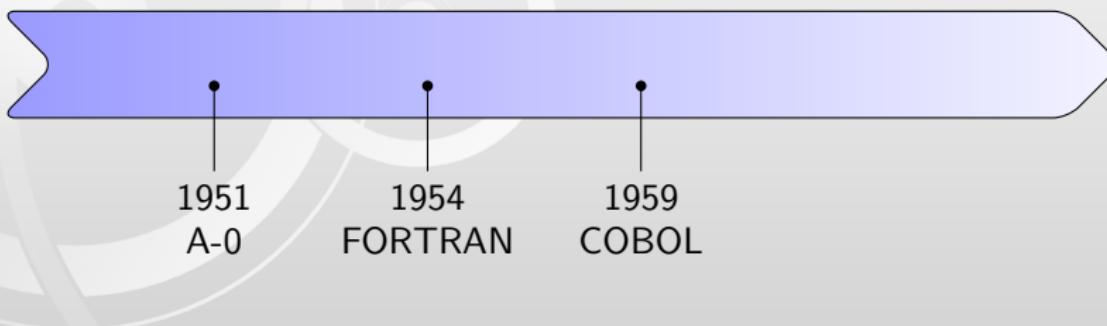


Grace Murray Hopper

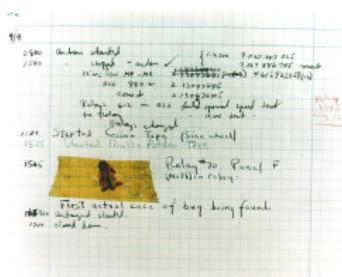


Informaticienne américaine, conceptrice du premier compilateur (A-0 System) et du langage COBOL.

Premiers langages symboliques



Grace Murray Hopper

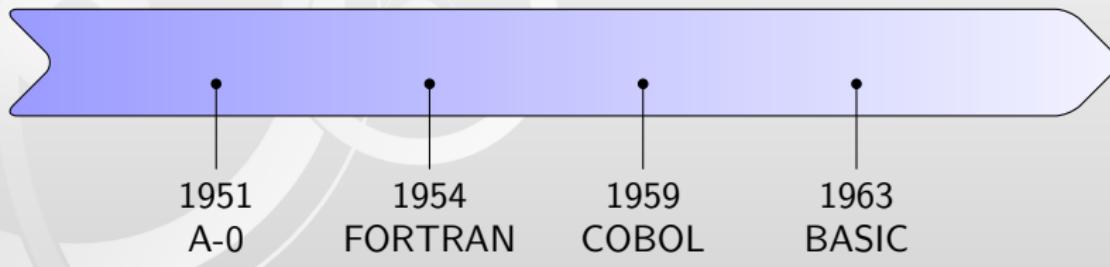


Informaticienne américaine, conceptrice du premier compilateur (A-0 System) et du langage COBOL.

John Backus

Informaticien chez IBM, publie en 1954 un article titré *Specifications for the IBM Mathematical FORmula TRANslating System*

Premiers langages symboliques



Grace Murray Hopper



Informaticienne américaine, conceptrice du premier compilateur (A-0 System) et du langage COBOL.

John Backus

Informaticien chez IBM, publie en 1954 un article titré *Specifications for the IBM Mathematical FORmula TRANslating System*

Kemeny & Kurtz

Conçoivent le BASIC au Dartmouth College

Premiers langages symboliques

BASIC

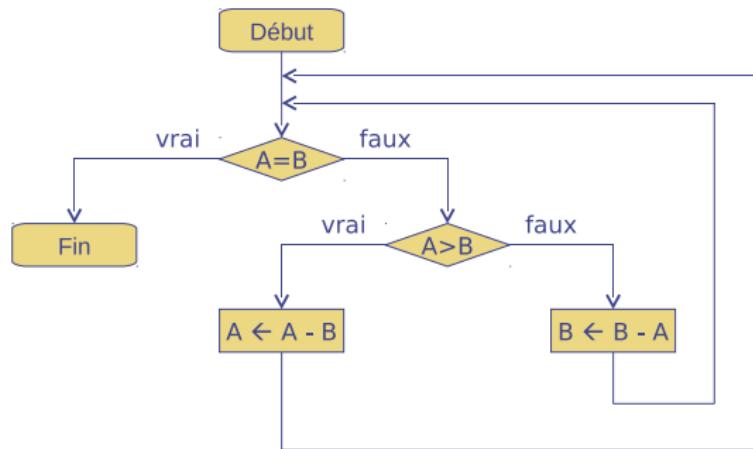
```
10 if A = B goto 70
20 if A < B goto 50
30 A = A - B
40 goto 10
50 B = B - A
60 goto 10
70 end
```

Premiers langages symboliques

BASIC

```
10 if A = B goto 70
20 if A < B goto 50
30 A = A - B
40 goto 10
50 B = B - A
60 goto 10
70 end
```

Organigramme (IBM 1969, norme ISO 5807 en 1985)

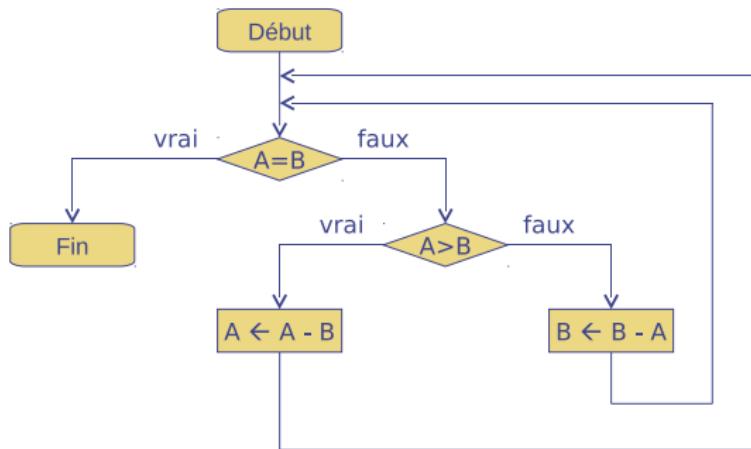


Premiers langages symboliques

BASIC

```
10 if A = B goto 70
20 if A < B goto 50
30 A = A - B
40 goto 10
50 B = B - A
60 goto 10
70 end
```

Organigramme (IBM 1969, norme ISO 5807 en 1985)



- Programmes difficiles à vérifier et à réutiliser

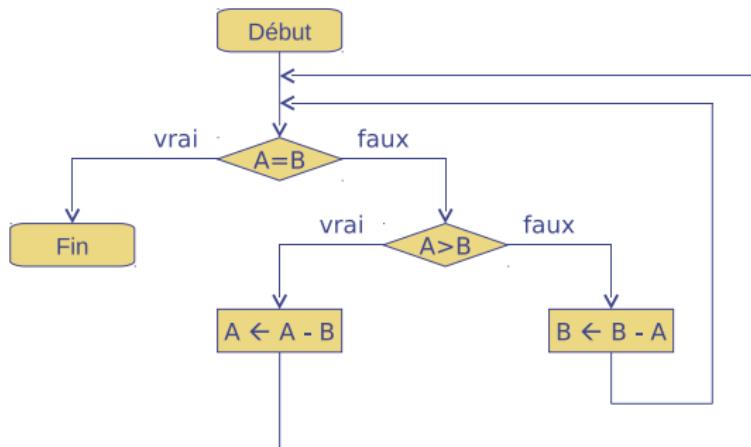
Premiers langages symboliques

BASIC

```

10 if A = B goto 70
20 if A < B goto 50
30 A = A - B
40 goto 10
50 B = B - A
60 goto 10
70 end
    
```

Organigramme (IBM 1969, norme ISO 5807 en 1985)



- Programmes difficiles à vérifier et à réutiliser
- Programmation « spaghetti »

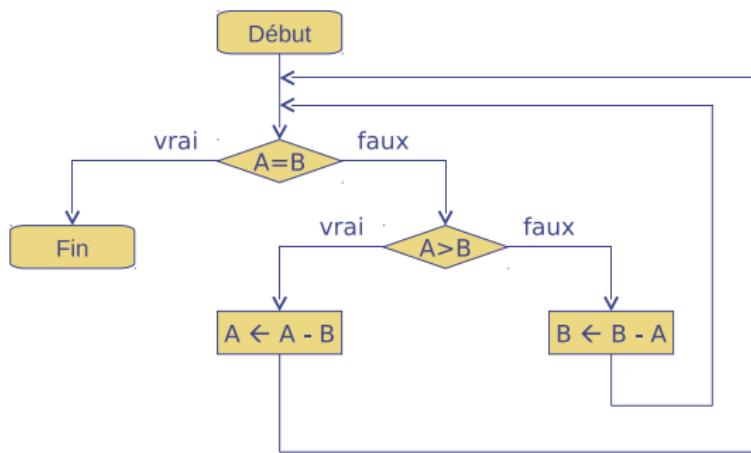
Premiers langages symboliques

BASIC

```

10 if A = B goto 70
20 if A < B goto 50
30 A = A - B
40 goto 10
50 B = B - A
60 goto 10
70 end
    
```

Organigramme (IBM 1969, norme ISO 5807 en 1985)



- Programmes difficiles à vérifier et à réutiliser
- Programmation « spaghetti »
- Organigrammes limités à des algorithmes très simples

Langages structurés de haut niveau

Langages structurés de haut niveau

- Années 70 : crise du logiciel

Langages structurés de haut niveau

- Années 70 : crise du logiciel
- Nombreuses publications sur le génie logiciel
 - 1966 : Böhm et Jiacopini, *Go To Statement Considered Harmful*
 - 1975 : Dijkstra, *Guarded commands, non determinacy and formal derivation of programs*
 - 1970 : Warnier, *Principes de la Logique de Construction de Programmes*
 - 1986 : Brooks, *No Silver Bullet : Essence and Accidents of Software Engineering*

Langages structurés de haut niveau

- Années 70 : crise du logiciel
- Nombreuses publications sur le génie logiciel
 - 1966 : Böhm et Jiacopini, *Go To Statement Considered Harmful*
 - 1975 : Dijkstra, *Guarded commands, non determinacy and formal derivation of programs*
 - 1970 : Warnier, *Principes de la Logique de Construction de Programmes*
 - 1986 : Brooks, *No Silver Bullet : Essence and Accidents of Software Engineering*

~~goto~~

~~break~~

Langages structurés de haut niveau

- Années 70 : crise du logiciel
- Nombreuses publications sur le génie logiciel
 - 1966 : Böhm et Jiacopini, *Go To Statement Considered Harmful*
 - 1975 : Dijkstra, *Guarded commands, non determinacy and formal derivation of programs*
 - 1970 : Warnier, *Principes de la Logique de Construction de Programmes*
 - 1986 : Brooks, *No Silver Bullet : Essence and Accidents of Software Engineering*

goto

break

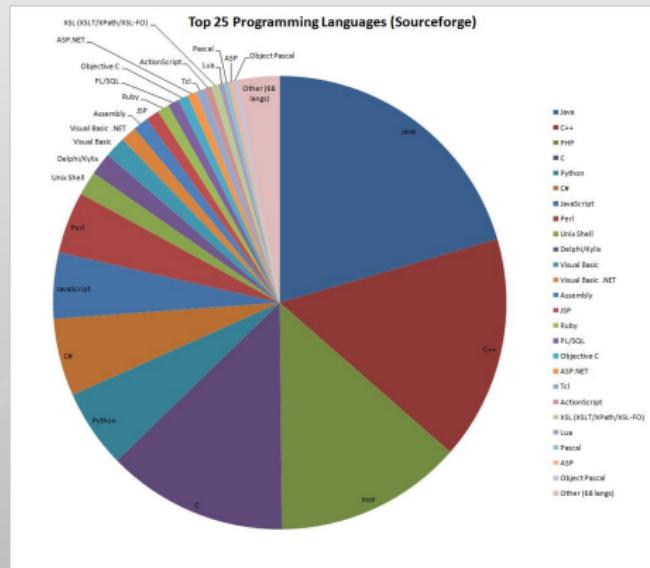
Introduction de nouveaux concepts

- Structures de contrôles
- Structures de données
- Sous-programmes

Situation actuelle

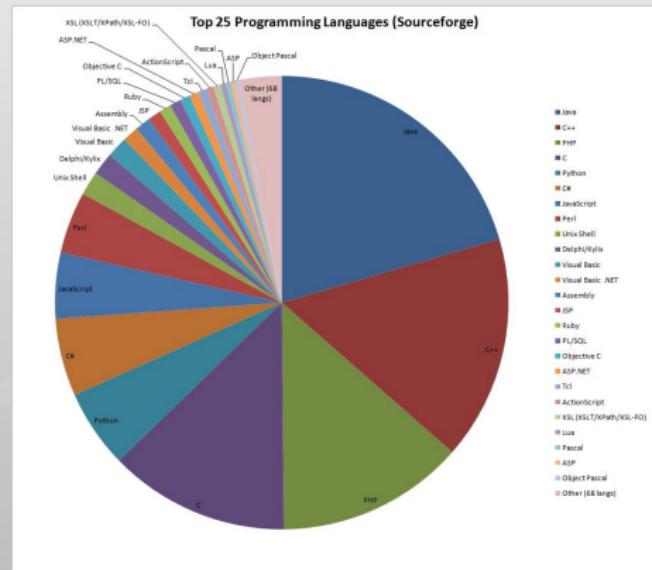
Situation actuelle

- Des centaines de langages



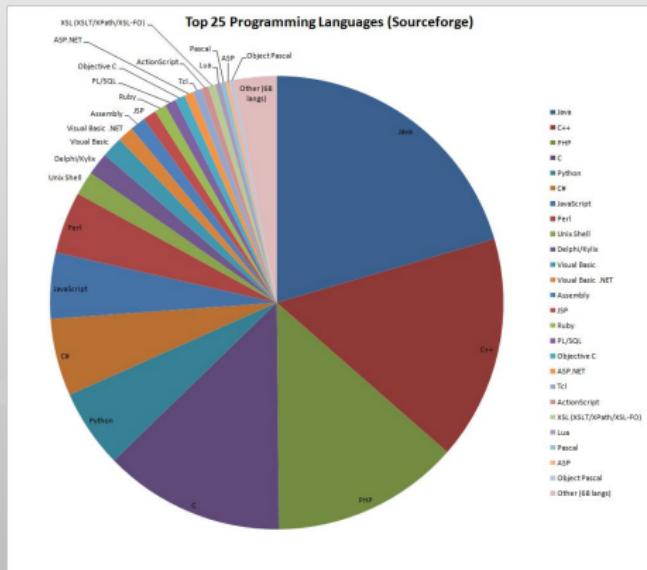
Situation actuelle

- Des centaines de langages
- Langages déclaratifs
 - Programmation fonctionnelle
 - Programmation logique



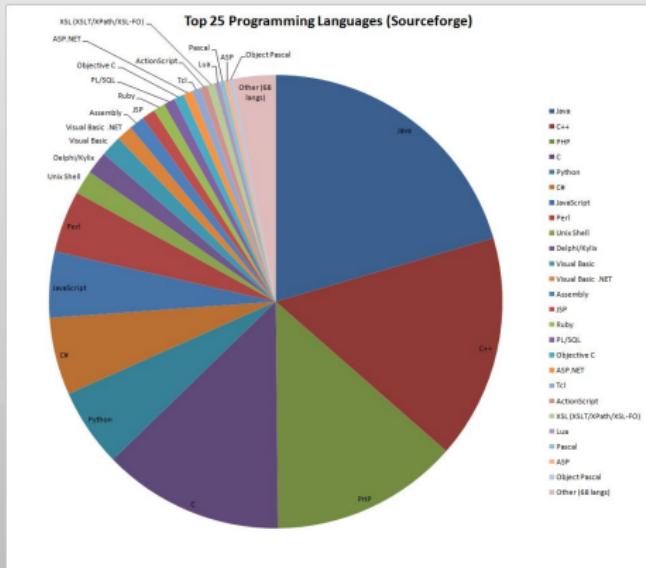
Situation actuelle

- Des centaines de langages
- Langages déclaratifs
 - Programmation fonctionnelle
 - Programmation logique
- Langages impératifs
 - **Programmation procédurale**
 - Programmation orientée objet



Situation actuelle

- Des centaines de langages
- Langages déclaratifs
 - Programmation fonctionnelle
 - Programmation logique
- Langages impératifs
 - **Programmation procédurale**
 - Programmation orientée objet
- Définition d'un pseudo-langage universel



2 Notion de variable

- 
- 1 Définition et historique
 - 2 Notion de variable
 - 3 Instructions et structures de contrôle
 - 4 Conclusion

Qu'est-ce qu'une variable ?

Qu'est-ce qu'une variable ?

Définition (Variable)

Une variable est une cellule mémoire dont le contenu est consultable et modifiable. Une variable est désignée par un identificateur.

Qu'est-ce qu'une variable ?

Définition (Variable)

Une variable est une cellule mémoire dont le contenu est consultable et modifiable. Une variable est désignée par un identificateur.

- Une variable doit être déclarée en début de programme ou de sous-programme

Qu'est-ce qu'une variable ?

Définition (Variable)

Une variable est une cellule mémoire dont le contenu est consultable et modifiable. Une variable est désignée par un identificateur.

- Une variable doit être déclarée en début de programme ou de sous-programme
- Une variable a toujours une valeur

Qu'est-ce qu'une variable ?

Définition (Variable)

Une variable est une cellule mémoire dont le contenu est consultable et modifiable. Une variable est désignée par un identificateur.

- Une variable doit être déclarée en début de programme ou de sous-programme
- Une variable a toujours une valeur
- Une variable doit toujours être initialisée avant d'être consultée

Qu'est-ce qu'un identificateur ?

Qu'est-ce qu'un identificateur ?

Définition (Identificateur)

Un identificateur est le nom associé à une variable, à un type, ou à un sous-programme. Un identificateur est une suite de caractères commençant par une lettre et ne contenant pas d'espace.

Qu'est-ce qu'un identificateur ?

Définition (Identificateur)

Un identificateur est le nom associé à une variable, à un type, ou à un sous-programme. Un identificateur est une suite de caractères commençant par une lettre et ne contenant pas d'espace.

- **A , x , b5 , couleur_bord** sont des identificateurs valides

Qu'est-ce qu'un identificateur ?

Définition (Identificateur)

Un identificateur est le nom associé à une variable, à un type, ou à un sous-programme. Un identificateur est une suite de caractères commençant par une lettre et ne contenant pas d'espace.

- **A , x , b5 , couleur_bord** sont des identificateurs valides
- **3A , couleur fond** ne le sont pas

Qu'est-ce qu'un identificateur ?

Définition (Identificateur)

Un identificateur est le nom associé à une variable, à un type, ou à un sous-programme. Un identificateur est une suite de caractères commençant par une lettre et ne contenant pas d'espace.

- **A , x , b5 , couleur_bord** sont des identificateurs valides
- **3A , couleur fond** ne le sont pas
- **Xi** est un identificateur, mais **i** n'est pas un indice

Comment déclarer une variable ?

Pseudo-langage

Type_de_la_variable identificateur_de_la_variable

Comment déclarer une variable ?

Pseudo-langage

Type_de_la_variable identificateur_de_la_variable

- Les déclarations sont regroupées en début de programme ou de sous-programme

Comment déclarer une variable ?

Pseudo-langage

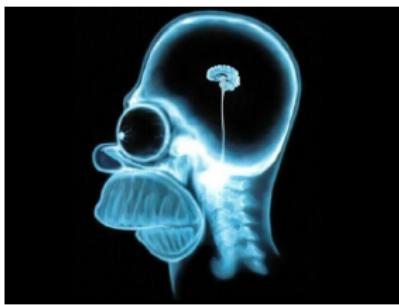
Type_de_la_variable identificateur_de_la_variable

- Les déclarations sont regroupées en début de programme ou de sous-programme
- Le type peut être soit un type prédéfini (entier, réel, booléen, caractère, chaîne) soit un type défini par l'utilisateur

Où sont stockées les variables ?

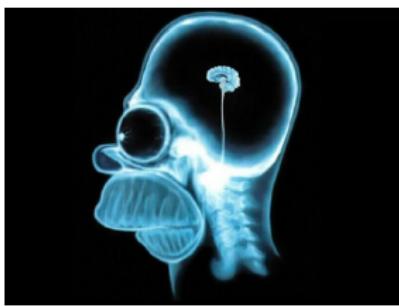
Où sont stockées les variables ?

Mémoire biologique

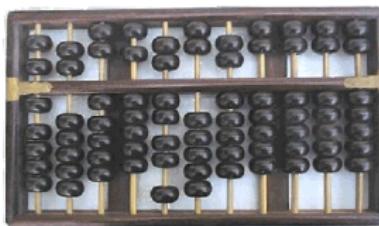


Où sont stockées les variables ?

Mémoire biologique



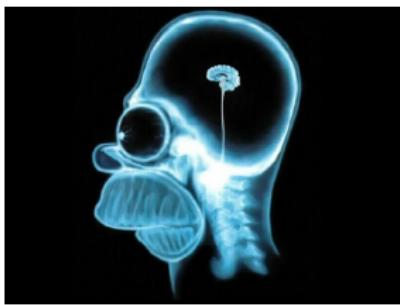
Mémoire mécanique



$$\begin{array}{r} \textcircled{1} \\ 13 \\ + 13 \\ + 13 \\ \hline 52 \end{array} \qquad \begin{array}{r} \textcircled{1} \\ 13 \\ \times 4 \\ \hline 52 \end{array}$$

Où sont stockées les variables ?

Mémoire biologique



Mémoire mécanique



$$\begin{array}{r} \textcircled{1} \\ 13 \\ + 13 \\ + 13 \\ + 13 \\ \hline 52 \end{array}$$
$$\begin{array}{r} \textcircled{1} \\ 13 \\ \times \quad 4 \\ \hline 52 \end{array}$$

Mémoire électronique



Représentation interne et externe

Il y a 10 types de personne dans le monde, ceux qui comprennent le binaire et les autres !

Représentation interne et externe

Il y a 10 types de personne dans le monde, ceux qui comprennent le binaire et les autres !

Représentation externe



- C'est la variable telle que imaginée/vue par l'utilisateur
- Une variable peut avoir plusieurs représentations externes

Représentation interne et externe

Il y a 10 types de personnes dans le monde, ceux qui comprennent le binaire et les autres !

Représentation externe



Représentation interne

A 10x10 grid of binary digits (0s and 1s) representing the internal memory representation of the string 'Hello world!' and the year '2008'. The digits are arranged in a pattern that corresponds to the ASCII values of the characters in the string.

- C'est la variable telle que imaginée/vue par l'utilisateur
- Une variable peut avoir plusieurs représentations externes

- Codage de la variable dans la machine
- Une variable d'un type donné n'aura qu'une seule représentation interne

3 Instructions et structures de contrôle

- 1 Définition et historique
- 2 Notion de variable
- 3 Instructions et structures de contrôle
- 4 Conclusion

L'affectation

Pseudo-langage

```
identificateur_de_la_variable ← expression
```

L'affectation

Pseudo-langage

```
identificateur_de_la_variable ← expression
```

- Affecte une valeur à une variable

L'affectation

Pseudo-langage

```
identificateur_de_la_variable ← expression
```

- Affecte une valeur à une variable
- Change le contenu d'une zone de mémoire

L'affectation

Pseudo-langage

```
identificateur_de_la_variable ← expression
```

- Affecte une valeur à une variable
- Change le contenu d'une zone de mémoire
- Déroulement :
 - 1 Évaluation de l'expression
 - 2 Mise en mémoire du résultat dans la variable

Structures de contrôles

Il existe trois types de structures de contrôle

- La séquence
- Le choix
- L'itération (boucle)

Structure élémentaire : la séquence

Séquence

Les instructions d'un programme s'exécutent dans l'ordre de leur écriture (de gauche à droite et de bas en haut).

Structure élémentaire : la séquence

Séquence

Les instructions d'un programme s'exécutent dans l'ordre de leur écriture (de gauche à droite et de bas en haut).

Une instruction par ligne !

Structure de choix *si-alors-sinon*

Pseudo-langage

```
si ( condition ) alors  
    séquence 1  
sinon  
    séquence 2  
finsi
```

Pseudo-langage

```
si ( condition ) alors  
    séquence 1  
finsi
```

Structure de choix *si-alors-sinon*

Pseudo-langage

```
si ( condition ) alors  
    séquence 1  
sinon  
    séquence 2  
finsi
```

Pseudo-langage

```
si ( condition ) alors  
    séquence 1  
finsi
```

- Condition doit être une expression booléenne

Structure de choix *si-alors-sinon*

Pseudo-langage

```
si ( condition ) alors
    séquence 1
sinon
    séquence 2
finsi
```

Pseudo-langage

```
si ( condition ) alors
    séquence 1
finsi
```

- Condition doit être une expression booléenne
- Si la valeur de la condition est *vrai* alors la séquence 1 est exécutée, lorsqu'elle se termine l'exécution reprend à la première instructions qui suit *finsi*

Structure de choix *si-alors-sinon*

Pseudo-langage

```
si ( condition ) alors
    séquence 1
sinon
    séquence 2
finsi
```

Pseudo-langage

```
si ( condition ) alors
    séquence 1
finsi
```

- Condition doit être une expression booléenne
- Si la valeur de la condition est *vrai* alors la séquence 1 est exécutée, lorsqu'elle se termine l'exécution reprend à la première instructions qui suit *finsi*
- Si la valeur de la condition est *faux* alors la séquence 2 est exécutée, lorsqu'elle se termine l'exécution reprend à la première instructions qui suit *finsi*

Structure d'itération *Tant que*

Pseudo-langage

```
tant que ( condition ) faire  
    séquence  
fintantque
```

Structure d'itération *Tant que*

Pseudo-langage

```
tant que ( condition ) faire  
    séquence  
fintantque
```

- La condition doit être une expression booléenne

Structure d'itération *Tant que*

Pseudo-langage

```
tant que ( condition ) faire  
    séquence  
fintantque
```

- La condition doit être une expression booléenne
- Répète la séquence tant que la valeur de la condition est *vrai*

Structure d'itération *Tant que*

Pseudo-langage

```
tant que ( condition ) faire  
    séquence  
fintantque
```

- La condition doit être une expression booléenne
- Répète la séquence tant que la valeur de la condition est *vrai*
- La condition n'est testée qu'au début de chaque boucle

Structure d'itération *Tant que*

Pseudo-langage

```
tant que ( condition ) faire  
    séquence  
fintantque
```

- La condition doit être une expression booléenne
- Répète la séquence tant que la valeur de la condition est *vrai*
- La condition n'est testée qu'au début de chaque boucle

Structure d'itération *Tant que*

Pseudo-langage

tant

Règles de base

fintant

- La condition est testée au début de la boucle
- Répétition
- La condition n'est testée qu'au début de chaque boucle

Structure d'itération *Tant que*

Pseudo-langage

tant

Règles de base

- Toute structure *tant que* doit être précédée de l'initialisation des variables intervenant dans l'expression booléenne de la condition

- La condition n'est testée qu'au début de chaque boucle

- Rép

Structure d'itération *Tant que*

Pseudo-langage

tant

fint

■ La

■ Rép

Règles de base

- Toute structure *tant que* doit être précédée de l'initialisation des variables intervenant dans l'expression booléenne de la condition
- La séquence doit modifier au moins l'une des variables intervenant dans la condition. Il est indispensable que la condition prenne la valeur *faux* à un moment donné, afin de sortir de la boucle itératif.
- La condition n'est testée qu'au début de chaque boucle

Structure d'itération *Répéter*

Pseudo-langage

répéter

 séquence

 tant que (condition)

Structure d'itération Répéter

Pseudo-langage

répéter

 séquence

 tant que (condition)

- La séquence est exécutée au moins une fois

Structure d'itération Répéter

Pseudo-langage

répéter

 séquence

 tant que (condition)

- La séquence est exécutée au moins une fois
- Répète la séquence tant que la valeur de la condition est *vrai*

Structure d'itération Répéter

Pseudo-langage

répéter

 séquence

 tant que (condition)

- La séquence est exécutée au moins une fois
- Répète la séquence tant que la valeur de la condition est *vrai*
- La condition n'est testée qu'à la fin de chaque boucle

Structure d'itération *Pour*

Pseudo-langage

```
pour var allant de min à max (par pas de incr) faire
    séquence
finpour
```

Structure d'itération *Pour*

Pseudo-langage

```
pour var allant de min à max (par pas de incr) faire
    séquence
finpour
```

- La séquence est exécutée pour les valeurs de *var* successivement égale à *min*, *min+incr*, *min+2*incr*, ... , *max*

Structure d'itération *Pour*

Pseudo-langage

```
pour var allant de min à max (par pas de incr) faire
    séquence
finpour
```

- La séquence est exécutée pour les valeurs de *var* successivement égale à *min*, *min+incr*, *min+2*incr*, ... , *max*
- *incr* peut être négatif, il faut adapter les bornes en conséquences

Structure d'itération *Pour*

Pseudo-langage

```
pour var allant de min à max (par pas de incr) faire
    séquence
finpour
```

- La séquence est exécutée pour les valeurs de *var* successivement égale à *min*, *min+incr*, *min+2*incr*, ... , *max*
- *incr* peut être négatif, il faut adapter les bornes en conséquences
- Le nombre d'itérations est connu à l'avance

Structure d'itération *Pour*

Pseudo-langage

```
pour var allant de min à max (par pas de incr) faire
    séquence
finpour
```

- La séquence est exécutée pour les valeurs de *var* successivement égale à *min*, *min+incr*, *min+2*incr*, ... , *max*
- *incr* peut être négatif, il faut adapter les bornes en conséquences
- Le nombre d'itérations est connu à l'avance
- Le pas d'incrémentation de la boucle peut être omis, il sera alors par défaut de 1

Structure d'itération *Pour*

Pseudo-langage

```
pour var allant de min à max (par pas de incr) faire  
    séquence  
finp
```

Attention !

- La structure de boucle commence par *pour*, suivi du nom de la variable (*var*), puis l'instruction *allant de*, le *min* et le *max*, puis l'*incrémentation* (ou pas) entre parenthèses, puis *faire*, puis la *séquence* de commandes, puis *finp*.
- La variable (*var*) prend toutes les valeurs comprises entre *min* et *max*, et l'incrémentation (*incr*) est effectuée après chaque itération.
- Le nombre d'itérations est connu à l'avance.
- Le pas d'incrémantation de la boucle peut être omis, il sera alors par défaut de 1.

Structure d'itération *Pour*

Pseudo-langage

```
pour var allant de min à max (par pas de incr) faire  
    séquence  
finp
```

Attention !

- La modification de la variable de contrôle est totalement interdite dans la boucle
- La variable de contrôle ne peut être initialisée qu'à l'aide de *min*, *max* et *incr*
- *incr* est obligatoire
- Le nombre d'itérations est connu à l'avance
- Le pas d'incrémentation de la boucle peut être omis, il sera alors par défaut de 1

Structure d'itération *Pour*

Pseudo-langage

```
pour var allant de min à max (par pas de incr) faire  
    séquence  
finp
```

Attention !

- La modification de la variable de contrôle est totalement interdite dans la boucle
- À la fin de la boucle, la variable de contrôle a une valeur indéterminée
- *incr* → à *min*,
- Le nombre d'itérations est connu à l'avance
- Le pas d'incrémentation de la boucle peut être omis, il sera alors par défaut de 1

Structure d'itération *Pour*

Pseudo-langage

```
pour var allant de min à max (par pas de incr) faire  
    séquence  
finp
```

Attention !

- La modification de la variable de contrôle est totalement interdite dans la boucle
- À la fin de la boucle, la variable de contrôle a une valeur indéterminée
- Les bornes ne doivent pas évoluer au cours des itérations
- incr
- Le nombre d'itérations est connu à l'avance
- Le pas d'incrémentation de la boucle peut être omis, il sera alors par défaut de 1

4 Conclusion

- 1 Définition et historique
- 2 Notion de variable
- 3 Instructions et structures de contrôle
- 4 Conclusion

Conclusion : coder proprement !

Conclusion : coder proprement !

- Tout algorithme séquentiel peut s'écrire comme une combinaison de ces trois structures (langage complet au sens de Turing)

Conclusion : coder proprement !

- Tout algorithme séquentiel peut s'écrire comme une combinaison de ces trois structures (langage complet au sens de Turing)
- Un problème peut être résolu par plusieurs algorithmes

Conclusion : coder proprement !

- Tout algorithme séquentiel peut s'écrire comme une combinaison de ces trois structures (langage complet au sens de Turing)
- Un problème peut être résolu par plusieurs algorithmes
- Complexité des algorithmes

Conclusion : coder proprement !

- Tout algorithme séquentiel peut s'écrire comme une combinaison de ces trois structures (langage complet au sens de Turing)
- Un problème peut être résolu par plusieurs algorithmes
- Complexité des algorithmes
- Problème de l'arrêt (Turing) : il n'est pas possible de savoir avec une machine de Turing si une autre machine de Turing s'arrêtera

Conclusion : coder proprement !

- Tout algorithme séquentiel peut s'écrire comme une combinaison de ces trois structures (langage complet au sens de Turing)
- Un problème peut être résolu par plusieurs algorithmes
- Complexité des algorithmes
- Problème de l'arrêt (Turing) : il n'est pas possible de savoir avec une machine de Turing si une autre machine de Turing s'arrêtera



Explosion d'Ariane 5 au cours de son vol inaugural le 4 juin 1996 en raison d'un bug informatique

Partie 2 : Introduction à la programmation structurée

- 5 Introduction
- 6 Types scalaires
- 7 Types structurés
- 8 Sous-programmes
- 9 Quelques mots sur le génie logiciel

5 Introduction

5 Introduction

6 Types scalaires

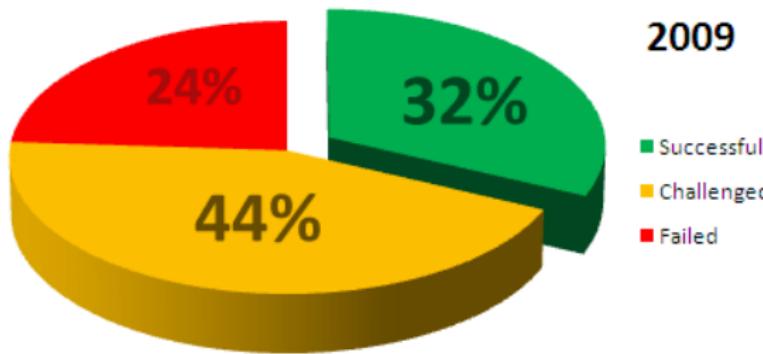
7 Types structurés

8 Sous-programmes

9 Quelques mots sur le génie logiciel

Avant propos

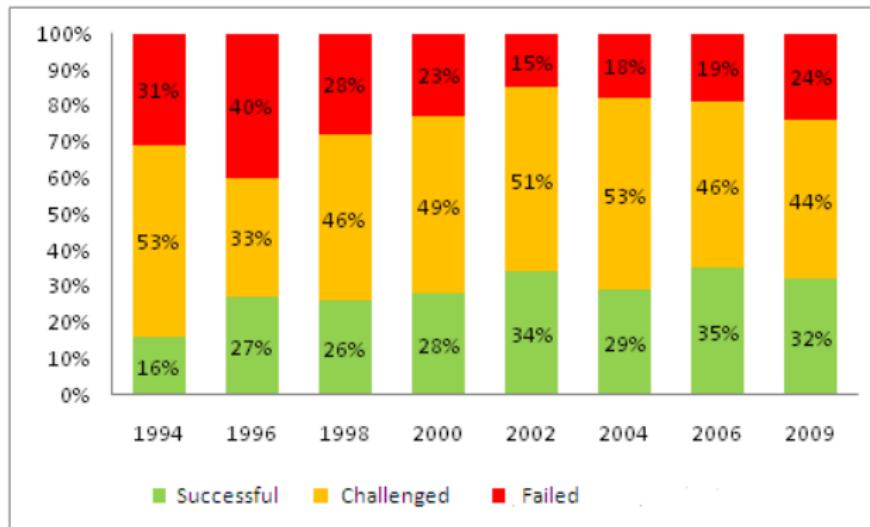
Le succès est rare !



Source : Standish Group 2009 chaos report

Avant propos

Le succès est rare !



Source : Standish Group 2009 chaos report

Défis du génie logiciel



Défis du génie logiciel

- Fuite en avant de la complexité

Défis du génie logiciel

- Fuite en avant de la complexité
- Coût du changement
 - Le coût d'un changement de fonctionnalité dans un logiciel est 10 fois plus élevé s'il a lieu en phase de développement que s'il est connu au départ, 100 fois plus élevé s'il a lieu en phase de production
 - Idem pour les corrections d'erreurs

Défis du génie logiciel

- Fuite en avant de la complexité
- Coût du changement
 - Le coût d'un changement de fonctionnalité dans un logiciel est 10 fois plus élevé s'il a lieu en phase de développement que s'il est connu au départ, 100 fois plus élevé s'il a lieu en phase de production
 - Idem pour les corrections d'erreurs
- L'importance de la maintenance est souvent sous-estimée

Défis du génie logiciel

- Fuite en avant de la complexité
- Coût du changement
 - Le coût d'un changement de fonctionnalité dans un logiciel est 10 fois plus élevé s'il a lieu en phase de développement que s'il est connu au départ, 100 fois plus élevé s'il a lieu en phase de production
 - Idem pour les corrections d'erreurs
- L'importance de la maintenance est souvent sous-estimée
- Faiblesse des tests

Défis du génie logiciel

- Fuite en avant de la complexité
- Coût du changement
 - Le coût d'un changement de fonctionnalité dans un logiciel est 10 fois plus élevé s'il a lieu en phase de développement que s'il est connu au départ, 100 fois plus élevé s'il a lieu en phase de production
 - Idem pour les corrections d'erreurs
- L'importance de la maintenance est souvent sous-estimée
- Faiblesse des tests
- Méthodes de développement inadaptées

Qu'est-ce que la programmation ?

- Avant de décrire l'ensemble des algorithmes, il convient de s'intéresser aux données et aux résultats

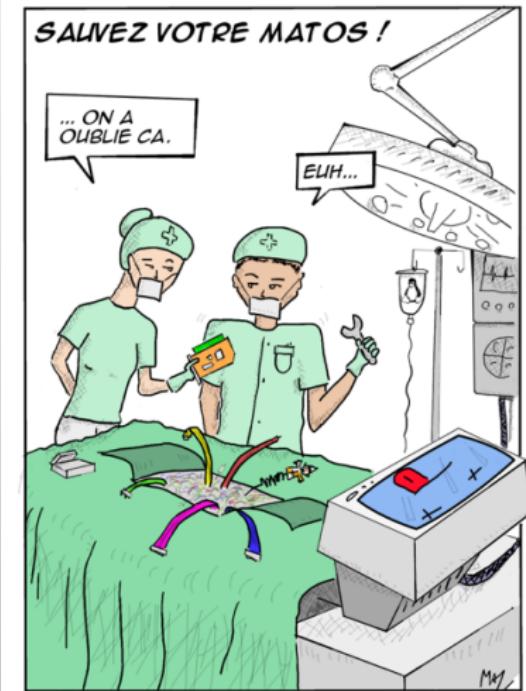


Illustration : Mathieu BRÉTHES
Licence CC : BY-SA

Qu'est-ce que la programmation ?

- Avant de décrire l'ensemble des algorithmes, il convient de s'intéresser aux données et aux résultats
- Comme dans toute activité de conception, on s'intéresse au « Quoi » avant d'expliciter le « Comment »

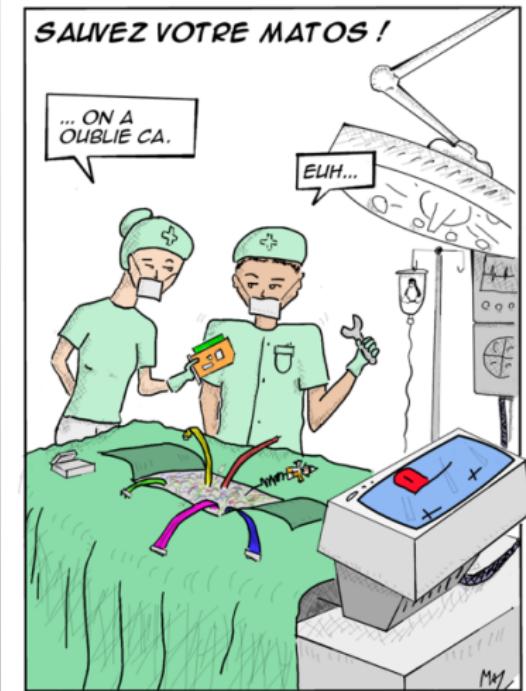
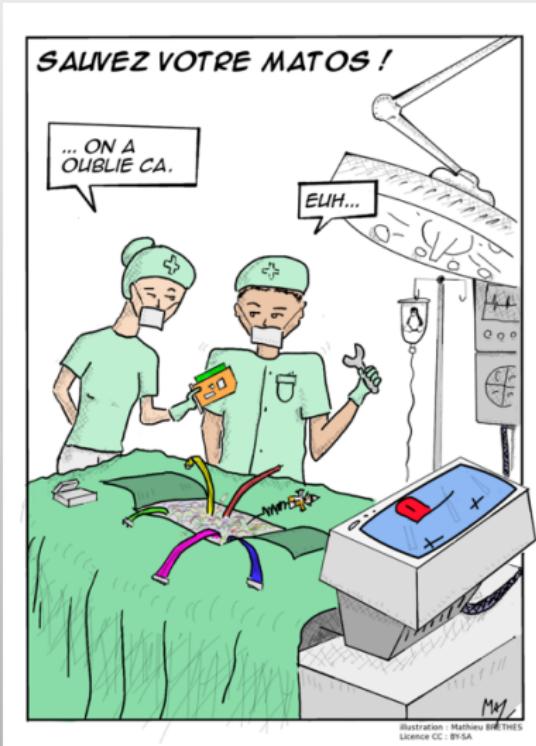


Illustration : Mathieu BRETHES
Licence CC : BY-SA

Qu'est-ce que la programmation ?

- Avant de décrire l'ensemble des algorithmes, il convient de s'intéresser aux données et aux résultats
- Comme dans toute activité de conception, on s'intéresse au « Quoi » avant d'expliciter le « Comment »
- Faire le cahier des charges avant !



Qu'est-ce que la programmation ?

- Avant de décrire l'ensemble des algorithmes, il convient de s'intéresser aux données et aux résultats
- Comme dans toute activité de conception, on s'intéresse au « Quoi » avant d'expliciter le « Comment »
- Faire le cahier des charges avant !

Ne pas confondre !

- Conception d'un logiciel (programmation en général)
- Écriture du programme (codage)

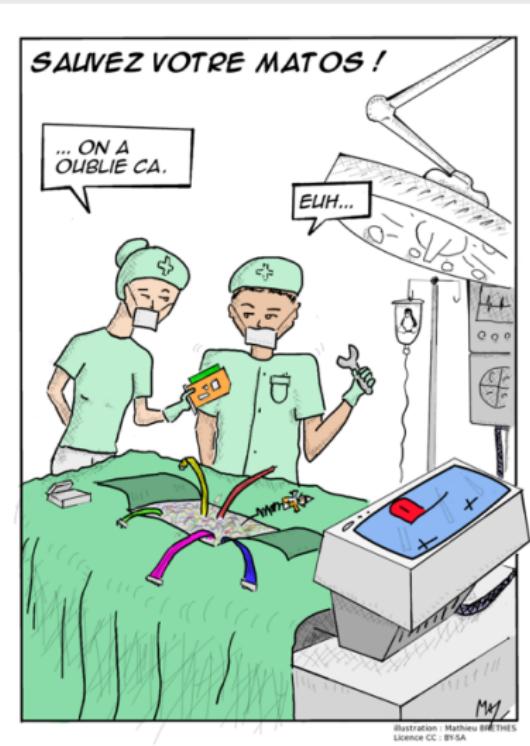


Illustration : Mathieu BIÈTHES
Licence CC : BY-SA

6 Types scalaires

5 Introduction

6 Types scalaires

7 Types structurés

8 Sous-programmes

9 Quelques mots sur le génie logiciel

Notion de type

- Chaque variable manipulée a un type unique et connu
- Un type est la représentation d'un ensemble fini d'information de même nature
- Un ensemble de fonctionnalités (opérations, fonctions) est associé à un type
- Typage fort (pas de mélange entre types)

Types scalaires

- Types scalaires prédéfinis
 - Entiers
 - Réels
 - Caractères
 - Booléens
 - Adresses mémoires (pointeurs)
- Types scalaires non prédéfinis
 - Définitions des propres types scalaires du programmeur.
 - Types énumérés (énumération de toutes les valeurs possibles du type)

Type *entiers*

- Entiers non signés : $\{ 0, 1, \dots, max \}$
- Entiers signés : $\{ -max + 1, -max + 2, \dots, -1, 0, 1, \dots, max \}$
- max dépend de la représentation interne :
 - Entiers signés sur 2 octets : $max = 2^{15} = 32\ 768$
 - Entiers signés sur 4 octets : $max = 2^{31} = 2\ 147\ 483\ 648$
- Opérateurs de calcul : $+$, $-$, $*$, $/$ (division entière) , $\%$ (modulo)
- Opérateurs relationnels : $<$, \leq , $>$, \geq , $=$, \neq

Type réel

- Nombre à virgule flottante de la forme $s.m.2^e$
 - s est le signe ($s \in \{-1, 1\}$)
 - m est la mantisse (entier non signé)
 - e est l'exposant (entier signé)
- Nombre réel sur 4 octets :
 - On ne peut représenter exactement que 4 294 967 296 réels !!
 - Norme IEEE 754 : exposant compris entre -127 et 127 , mantisse avec 7 chiffres significatifs
- Opérateurs de calcul : $+$, $-$, $*$, $/$
- Opérateurs relationnels : $<$, \leq , $>$, \geq
- Fonctions : `sqrt`, `abs`, `cos`, `sin`, `exp`, etc.

Type réel

- Nombre à virgule flottante de la forme $s.m.2^e$
 - s est le signe ($s \in \{-1, 1\}$)
 - m est la mantisse (entier non signé)

Jamais de calculs exacts en informatique !

- Nombre à virgule flottante de la forme $s.m.2^e$
 - s est le signe ($s \in \{-1, 1\}$)
 - m est la mantisse (entier non signé)
- Entrée d'un nombre réel :
 - Entrée : 0.00001
 - Résultat : 0.00000999999974737875
- Opérateurs arithmétiques : +, -, *, /
- Opérateurs relationnels : < , ≤ , > , ≥
- Fonctions : sqrt, abs, cos, sin, exp, etc.

The screenshot shows a Windows command-line window titled "reel.exe". The window contains the following text:
entrer une valeur réelle : 0.00001
vous avez choisi la valeur : 0.00000999999974737875

c 7

Type *booléens*

- Représentation externe : Vrai / Faux
- Représentation interne : 1 / 0
- Opérateurs : et, ou, non

Type caractère

- Représentation externe :

a b ... A B ... 0 1 ... 9 < * + ! ([...

- Représentation interne : code ASCII étendu (entier entre 0 et 255)

- Relation d'ordre :

'0' < '1' < ... < 'a' < 'b' < ... < 'z' < 'A' < ... < 'Z'

- Utilisation des guillemets simples pour éviter la confusion avec un identificateur

7 Types structurés

5 Introduction

6 Types scalaires

7 Types structurés

8 Sous-programmes

9 Quelques mots sur le génie logiciel

Types structurés

- Types scalaires = types de base d'un langage
- Mais comment représenter ?
 - Un vecteur
 - Un polynôme
 - Une adresse
 - ...
- Besoin de modélisation et de structuration de l'information
- Deux structures principales
 - Le tableau
 - L'enregistrement

Structure tableau

Pseudo-langage

```
Nom_du_type nom_du_tableau[dimension]
```

- Un tableau permet de regrouper des données de **même** type
- La dimension d'un tableau est fixe
- Accès à l'élément i avec la notation $A[i]$ et $1 \leq i \leq \text{dimension}$

Type *chaîne de caractère*

- Souvent définie par un tableau de caractères
- N'est pas prédefinie en C, il faut donc définir un nouveau type

Pseudo-langage

```
Type chaîne = tableau [20] de caractères ;
```

Structure *enregistrement*

Pseudo-langage

```
Type Nom_du_type = enregistrement
    type1  champ1
    type2  champ2
    ...
    typeN  champN
```

- Un enregistrement permet de regrouper de données de **differents** types
- Chaque **champ** est nommé et typé
- Accès aux données avec la notation pointée A.champ1, A.champ2, etc.

8 Sous-programmes

5 Introduction

6 Types scalaires

7 Types structurés

8 Sous-programmes

9 Quelques mots sur le génie logiciel

Qu'est-ce qu'un sous-programme ?

Objectif

Représenter, nommer, paramétriser une séquence d'instructions

- Pourquoi ?
 - Simplifier l'écriture d'un long programme
 - Faciliter la mise au point
 - Isoler des fonctionnalités
 - Enrichir le langage et permettre la réutilisation de fonctionnalités

Définition d'un sous-programme

Spécification d'un sous-programme (le Quoi)

- Entête du sous-programme
 - Nom du sous-programme
 - Listes des paramètres (type et nom)
- Spécifications et conditions d'utilisation (commentaires)

Définition d'un sous-programme

Spécification d'un sous-programme (le Quoi)

- Entête du sous-programme
 - Nom du sous-programme
 - Listes des paramètres (type et nom)
- Spécifications et conditions d'utilisation (commentaires)

Implantation d'un sous-programme (le Comment)

- Rappel de l'entête
- les déclarations locales (variables nécessaires à l'exécution du sous-programme)
- Séquence d'instructions correspondant au traitement désiré

Définition d'une fonction

Fonction

- calculent et retournent **un et un seul résultat**
- ne modifient pas de variables existantes à l'extérieur
- s'utilisent dans des expressions

Entier X, Y, R, Z

X \leftarrow abs(-510)

Y \leftarrow pgcd(X, 30)

T \leftarrow 25

R \leftarrow 15

Z \leftarrow 5+pgcd(T , R)

Polynôme A, B, C, D

Entier n;

...

/ initialisation des polynômes */*

/ A et B */*

...

n \leftarrow degré(A)

C \leftarrow somme(A , B)

D \leftarrow somme(A , somme(B,C))

Définition d'une fonction

Spécification d'une fonction



```
fonction type nom_de_la_fonction ( liste des paramètres )
/* commentaire indiquant clairement ce que fait la fonction */
/* éventuellement les conditions d'utilisation de la fonction */
```

Définition d'une fonction

Implantation d'une fonction

```
fonction    type_résultat    nom_de_fonction ( liste des paramètres )
début
    /* déclarations locales au sous-programme */
    type_résultat    res ;

    /* séquence d' instructions calculant le résultat res */
    ...

    retourner    res ;
fin
```

Définition d'une procédure

Procédure

- peuvent calculer et donner **plusieurs résultats**
- peuvent modifier des objets déjà existant en mémoire centrale sans copie
- s'utilisent comme des instructions (**rien à gauche**)

Entier R, S

R ← 5
S ← -17

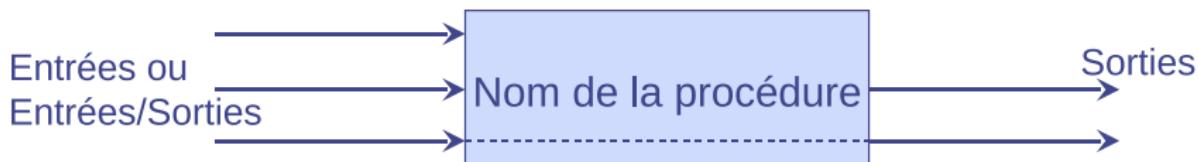
si (S < R) alors
 échangerValeur(R,S)
finsi

afficherEcran (R)

Polynôme A
Complexe Z,W;
...
si (degré(P) = 2) alors
 calculerRacines (A,Z,W)
finsi

Définition d'une procédure

Spécification d'une procédure



```
procédure nom_de_procedure ( liste des paramètres incluant leur nature )
/* nature des paramètres (E , S ou E/S) */
/* commentaire indiquant clairement ce que fait la procedure */
/* éventuellement les conditions d'utilisation de la procedure */
```

Définition d'une procédure

Implantation d'une procédure

```
procédure nom_de_procedure ( liste des paramètres incluant leur nature )
début
    /* déclarations locales au sous-programme */
    /* séquence d' instructions réalisant la spécification */
    ...
fin
```

Passage des paramètres

Passage par valeur

```
Procédure incrA ( entier i (E) )
/* i est passé par valeur */
début
    i ← i + 1 ;
fin
```

Programme appelant

```
entier N
N ← 10
afficherEcran (N)
incrA(N)
afficherEcran (N)
incrB(N)
afficherEcran (N)
```

Passage par adresse

```
Procédure incrB ( entier i (E/S) )
/* i est passé par adresse */
début
    i ← i + 1 ;
fin
```

Résultat à l'écran

```
10
10
11
```

Quelques remarques sur les sous-programmes

Nature du sous-programme

Quelques remarques sur les sous-programmes

Nature du sous-programme

- 1 seul paramètre de sortie et n entrées \Rightarrow fonction

Quelques remarques sur les sous-programmes

Nature du sous-programme

- 1 seul paramètre de sortie et n entrées \Rightarrow fonction
- Au moins 1 paramètre d'entrée/sortie \Rightarrow procédure

Quelques remarques sur les sous-programmes

Nature du sous-programme

- 1 seul paramètre de sortie et n entrées \Rightarrow fonction
- Au moins 1 paramètre d'entrée/sortie \Rightarrow procédure
- 0 ou plus d'un paramètre de sortie \Rightarrow procédure

Quelques remarques sur les sous-programmes

Nature du sous-programme

- 1 seul paramètre de sortie et n entrées \Rightarrow fonction
 - Au moins 1 paramètre d'entrée/sortie \Rightarrow procédure
 - 0 ou plus d'un paramètre de sortie \Rightarrow procédure
-
- Quelques conseils :

Quelques remarques sur les sous-programmes

Nature du sous-programme

- 1 seul paramètre de sortie et n entrées \Rightarrow fonction
 - Au moins 1 paramètre d'entrée/sortie \Rightarrow procédure
 - 0 ou plus d'un paramètre de sortie \Rightarrow procédure
-
- Quelques conseils :
 - Une chose à la fois

Quelques remarques sur les sous-programmes

Nature du sous-programme

- 1 seul paramètre de sortie et n entrées \Rightarrow fonction
 - Au moins 1 paramètre d'entrée/sortie \Rightarrow procédure
 - 0 ou plus d'un paramètre de sortie \Rightarrow procédure
-
- Quelques conseils :
 - Une chose à la fois
 - Maximum 3 paramètres

Quelques remarques sur les sous-programmes

Nature du sous-programme

- 1 seul paramètre de sortie et n entrées \Rightarrow fonction
 - Au moins 1 paramètre d'entrée/sortie \Rightarrow procédure
 - 0 ou plus d'un paramètre de sortie \Rightarrow procédure
-
- Quelques conseils :
 - Une chose à la fois
 - Maximum 3 paramètres
 - Un sous-programme = 10 lignes

Quelques remarques sur les sous-programmes

Nature du sous-programme

- 1 seul paramètre de sortie et n entrées \Rightarrow fonction
 - Au moins 1 paramètre d'entrée/sortie \Rightarrow procédure
 - 0 ou plus d'un paramètre de sortie \Rightarrow procédure
-
- Quelques conseils :
 - Une chose à la fois
 - Maximum 3 paramètres
 - Un sous-programme = 10 lignes
 - Une structure d'itération par sous-programme

Quelques remarques sur les sous-programmes

Nature du sous-programme

- 1 seul paramètre de sortie et n entrées \Rightarrow fonction
- Au moins 1 paramètre d'entrée/sortie \Rightarrow procédure
- 0 ou plus d'un paramètre de sortie \Rightarrow procédure

- Quelques conseils :
 - Une chose à la fois
 - Maximum 3 paramètres
 - Un sous-programme = 10 lignes
 - Une structure d'itération par sous-programme
 - Tester un à un chaque sous-programme

9 Quelques mots sur le génie logiciel

5 Introduction

6 Types scalaires

7 Types structurés

8 Sous-programmes

9 Quelques mots sur le génie logiciel

Génie logiciel

Génie logiciel

Le génie logiciel est l'ensemble des activités de conception et de mise en oeuvre des produits et des procédures tendant à rationaliser la production du logiciel et son suivi.

Cycle de vie d'un logiciel

Cycle de vie d'un logiciel

- Analyse et conception : quel est le problème à résoudre ?
 - Données de départ et résultats attendus
 - Modèle de données : définition d'une représentation des données
 - Méthode de résolution (algorithmes) : description des étapes successives pour arriver aux résultats
 - Affinage progressif

Cycle de vie d'un logiciel

- Analyse et conception : quel est le problème à résoudre ?
 - Données de départ et résultats attendus
 - Modèle de données : définition d'une représentation des données
 - Méthode de résolution (algorithmes) : description des étapes successives pour arriver aux résultats
 - Affinage progressif
- Codage
 - Traduction de l'algorithme dans un langage de programmation : code source
 - Langage de haut niveau : Pascal, C, C++, java, etc.
 - Langage de bas niveau : Assembleur.
 - Résultats du codage est un fichier texte

Cycle de vie d'un logiciel

Cycle de vie d'un logiciel

■ Mise au point

- Compilation : traduction du langage de programmation vers un langage compréhensible par la machine
- Jeu de tests : exécution du programme ou sous-programme avec des données typiques et représentative d'une utilisation future
- Corrections des erreurs
- Réitérations de ces trois étapes autant de fois que nécessaire

Cycle de vie d'un logiciel

■ Mise au point

- Compilation : traduction du langage de programmation vers un langage compréhensible par la machine
- Jeu de tests : exécution du programme ou sous-programme avec des données typiques et représentative d'une utilisation future
- Corrections des erreurs
- Réitérations de ces trois étapes autant de fois que nécessaire

■ Maintenance

- Corrections d'erreur non détectée lors de la mise au point
- Mise à jour pour répondre à de nouveaux besoins

Cycle de vie d'un logiciel

Les dangers du développement en cascade



Proposé par le commercial



Spécifié par le chef de projet



Conçu par l'équipe de conception



Réalisé par les programmeur



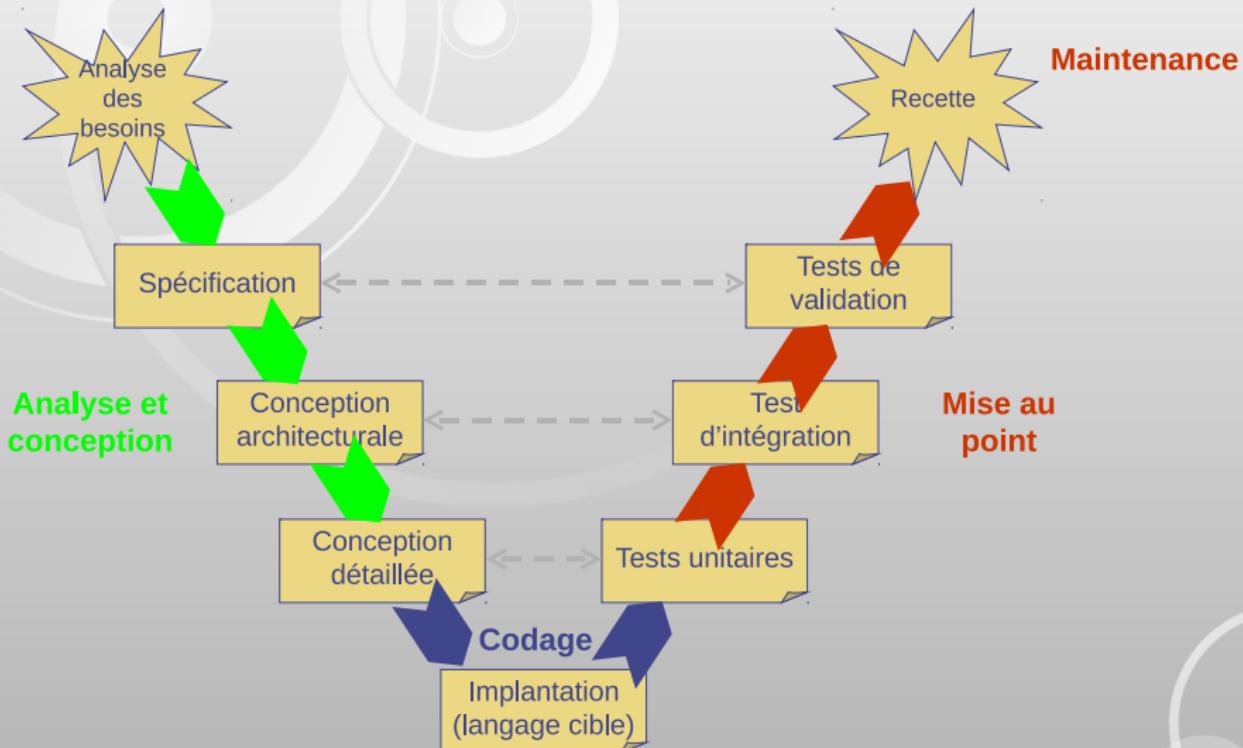
Installé sur le site



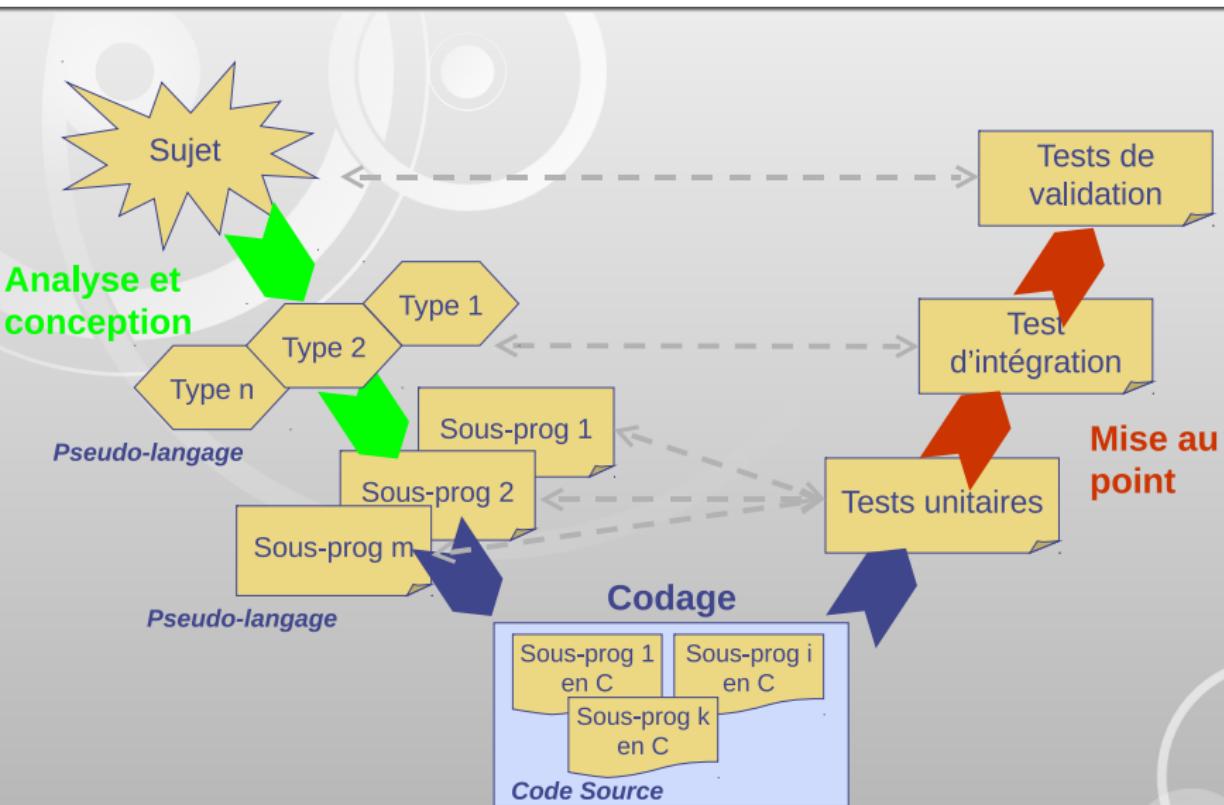
Ce que voulait l'utilisateur

Source : Univ. of London Computer Center Newsletter n°53 march 1973

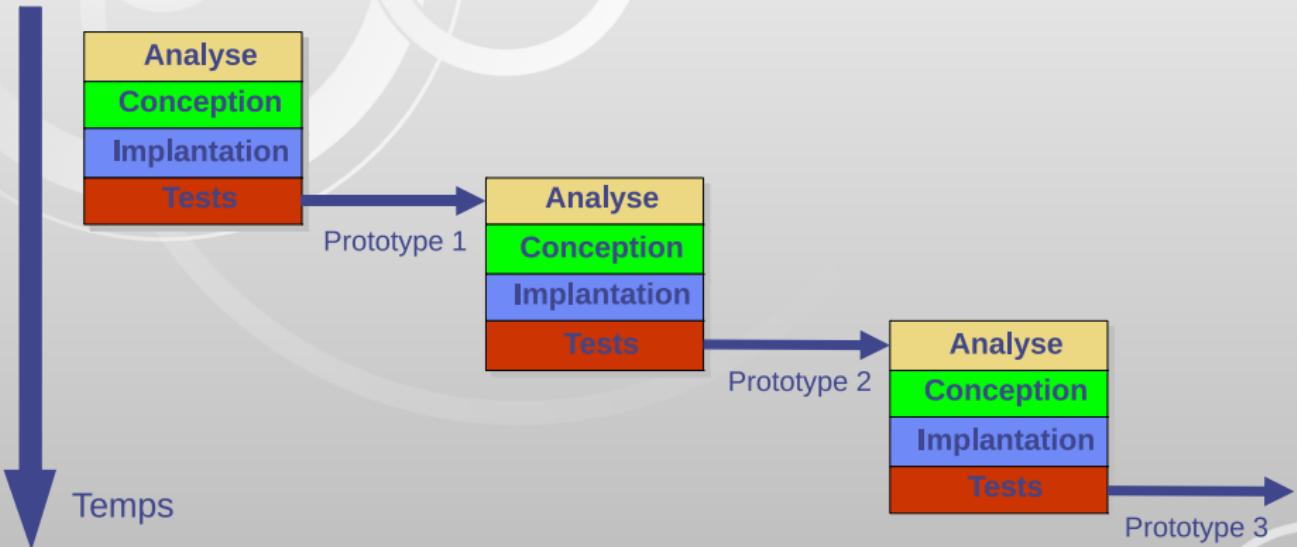
Cycle en V



Cycle en V simplifié



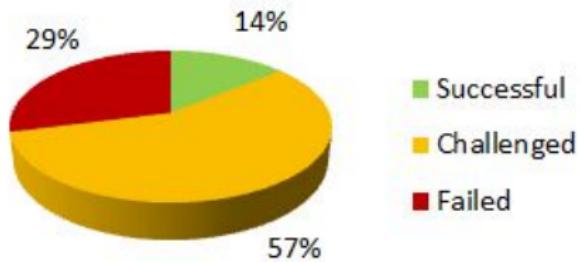
Développement itératif



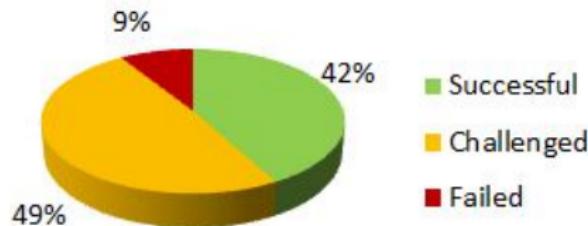
Conclusion

Vive les méthodes agiles !

Cycle projet - méthode classique
(cascade)



Cycle projet - méthode agile



Source : Standish Group 2010 chaos report