ADA

- **Objectif** : aborder les concepts fondamentaux du génie logiciel (robustesse du code, modularité, réutilisabilité, portabilité, généricité, portabilité, spécifications, sureté de fonctionnement, ...) et leur illustration dans le langage ADA

- **Intérêts** :
  - sensibilisation aux bonnes pratiques de programmation et aux mécanismes intégrés aux langages permettant de "bien" programmer
  - ADA est utilisé dans l'industrie pour des applications nécessitant robustesse et fiabilité

- **Références** :
  - *Programmation séquentielle avec ADA95*, P. Breguet & L. Zaffalon, Presses Polytechniques et Universitaires Romandes
  - *ADA95*, P. Gabrini, De Boeck University
  - *Méthodes de génie logiciel avec ADA95*, J.P. Rosen, InterEditions

Ressources en ligne sur ADA

- *http://www.adahome.com/* : un site très complet contenant documentation, cours, tutoriels, le manuel de référence, liens vers d'autres ressources de développement (compilateurs , ...), information autour du langage et de ses applications, offres d'emploi, ...

- *http://www.ada-france.org/* : un site francophone du même type que *adahome*

- *http://directory.fsf.org/devel/compilers/* : page du site de la Free Sofware Fondation où on peut trouver entre autres un compilateur pour ADA95

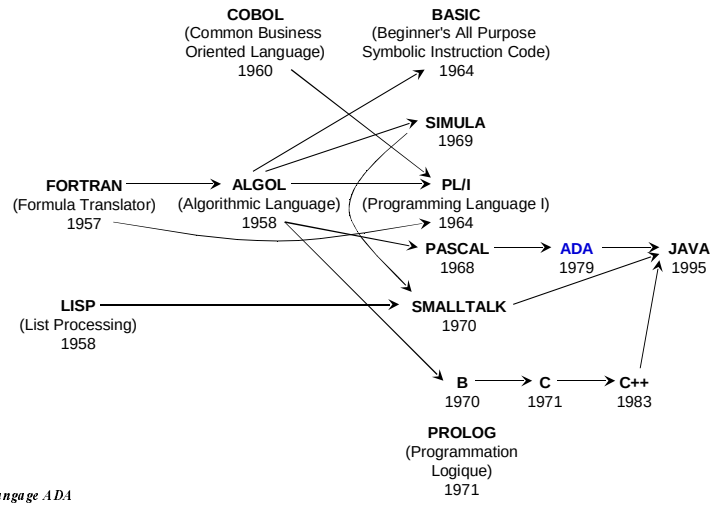
- *http://cuiwww.unige.ch/db-research/Enseignement/analyseinfo/ Ada95/ BNFindex.html#3* : un site avec les diagrammes syntaxiques de ADA95

- *http://www.adaic.com/standards/05rm/html/RM-TTL.html* : manuel de référence en ligne

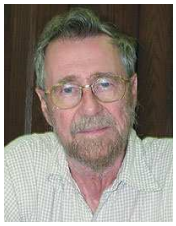
ADA et les autres langages (1/2)

- Langages de bas niveau : les langages assembleur, reflètent les instructions du processeur et leur organisation en mémoire
  - très peu lisible pour les programmeurs, expressivité très pauvre
  - les données sont des registres mémoire, pas de contrôle sur les types
  - organisation du programme à base de jump (goto) source potentielle de bugs
  - un programme ne tourne que sur un type de processeur
- Langages de haut niveau (la plupart des langages)
  - introduisent une organisation du code qui facilite l'écriture, la compréhension, le contrôle des instructions, la maintenance et la réutilisation (modularité, généricité, encapsulation)
  - introduisent la possibilité de vérifier voire de prouver le code sans l'exécuter (types, programmation structurée)
- Langages de très haut niveau (Prolog, Lisp, langages spécialisés)
  - totalement indépendants du matériel et très abstraits
  - souvent déclaratifs
  - adaptés à une tâche particulière

ADA et les autres langages (2/2)



Citations



- **Edsger Wybe Dijkstra,** Prix Turing de l'ACM (Association for Computing Machinery) en 1972 pour sa science et son art des langages de programmation
- *L'informatique n'est pas plus la science des ordinateurs que l'astronomie n'est celle des télescopes*.
- *Si tester un programme peut être une technique très efficace pour montrer la présence de bogues, elle est désespérément incapable d'en montrer l'absence*.
- *La programmation par objets est une idée exceptionnellement mauvaise qui ne pouvait naître qu'en Californie.*

Historique de ADA

- 1975 Le Département of Defense américain décide de développer un nouveau langage pour remplacer les centaines de langages utilisés jusque là dans les logiciels militaires. Ce langage devra être le plus sur possible dans son fonctionnement et faciliter au maximum la programmation pour répondre aux problèmes de qualité et de coût des logiciels. Ce langage devra aussi être le plus général possible pour remplacer la plupart des langages utilisés.

  Un grand concours est lancé, remporté par Honeywell-Bull à Paris pour son langage baptisé ADA
- 1983 Première version d'ADA (ADA83)
- 1995 Deuxième version d'ADA (ADA95), avec en particulier une évolution complète vers la technologie objet (premier langage objet normalisé)

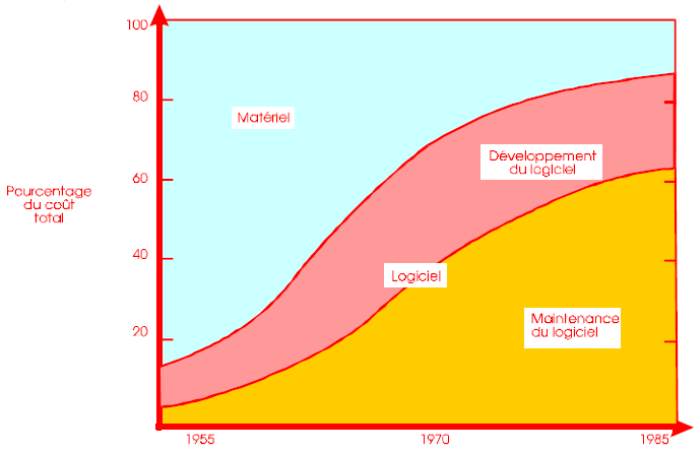
ADA, La première informaticienne

- Lady *Adélaide Augusta Lovelace* (1815-1852), fille de Lord Byron, poête fameux. Elle étudia les mathématiques et collabora avec Charles Babbage sur sa Machine Analytique, qui implémentait mécaniquement les principaux mécanismes qu'on trouve dans les ordinateurs. Ada (son diminutif) est réputée avoir écrit le premier programme informatique, qui devait calculer les termes de la suite de Bernoulli.

ADA dans l'industrie

- ADA est largement utilisé partout où sont requis des logiciels très fiables :
  - transports (Airbus, Boeing, TGV, ...)
  - systèmes spatiaux (ArianeV, Colombus, ...)
  - logiciels des systèmes bancaires
  - systèmes militaires (en particulier américains à partir des années 80)
  - ...
- ADA est utilisé pour les systèmes répartis, du fait de la qualité et de la robustesse de sa gestion des processus
  - systèmes de télécommunication (GPS, ...)
- ADA est aussi utilisé comme langage généraliste, en particulier pour les très gros logiciels du fait de sa modularité

De l'intérêt de bien programmer...



Principes du génie logiciel (1/2)

- Naissance du génie logiciel (ou ingénierie des logiciels) :
  - *première période du développement de logiciel*, l'artisanat (jusque fin des années 70) : chacun écrit son code dans son coin, sans norme, sans insérer son travail dans un cycle industriel et sans trop se préoccuper de ce qui se passe une fois le logiciel écrit
  - *deuxième période du développement de logiciel*, l'industrie : un logiciel est développé en équipe, selon des spécifications pré-existantes, selon des méthodes de développement précises en tenant compte des nécessités de fiabilité, évolutivité, réutilisabilité, ...
  - le terme génie logiciel (software engineering) est né en octobre 1968 à l'occasion d'une conférence à Garmisch-Partenkirchen

- Le génie logiciel vise :
  - à développer les bons logiciels (correspondant aux besoins, fiables, ...)
  - à bien développer les logiciels (facilité de développement, de maintenance et d'évolution, ...)

Principes du génie logiciel (2/2)

- Le génie logiciel fournit :
  - des *modèles de développement*, permettant d'encadrer et d'assister la conception, le développement, l'évolution d'un logiciel
  - des *méthodes d'analyse et de conception*
  - des *méthodes de spécification et de test*
  - des *méthodes et outils pour évaluer la qualité* des logiciels
- Modèles de développement : modèle en cascade, modèle en V, modèle en spirale, ...
- Méthodes d'analyse et de conception : Merise, OMT, méthodes basées sur UML, ...
- Méthodes de spécification et de test : méthode VDM, langage Z et méthode B, ...
- Méthodes d'évaluation de la qualité : facteurs de qualité

Facteurs de qualité coté utilisateur

- Fiabilité : pas de plantage, réponses correctes du logiciel
  - *Nécessite : robustesse du langage, performance des compilateurs*
- Intégrité : protection des données
  - *Nécessite : traçage et contrôle des accès*
- Ergonomie : facilité d'utilisation
  - *Nécessite : techniques de l'IHM*
- Efficacité : minimisation des ressources machine utilisées
  - *Nécessite : efficacité du codage, de la compilation*

Facteurs de qualité coté développeur (1/2)

- Maintenabilité : facilité de correction du code sans cascade de modifications ou apparition de nouvelles erreurs
  - *Nécessite : modularité, documentation, normalisation du code*
- Extensibilité : possibilité de modifier et de faire évoluer facilement le logiciel, et de façon fiable
  - *Nécessite : modularité, documentation, normalisation du code*
- Réutilisabilité : possibilité d'utiliser en partie le logiciel pour une autre application
  - *Nécessite : modularité, généricité du code, langage indépendant des plateformes*

Facteurs de qualité coté développeur (2/2)

- Portabilité : indépendance du logiciel par rapport à l'environnement matériel, au système d'exploitation, au compilateur
  - *Nécessite : indépendance du langage par rapport aux plateformes*
- Lisibilité : accessibilité du code qui doit être compréhensible rapidement, documentation facile à construire
  - *Nécessite : langage verbeux, normalisation du code, outils de documentation*
- Testabilité : facilité de vérification du code et de test fonctionnels
  - *Nécessite : modularité, tracabilité, documentation*

La normalisation

- Un langage est *normalisé* quand il répond à des règles de syntaxes formelles reconnues et imposées par une norme
- Un langage normalisé assure que n'importe quel programme écrit (correctement) dans ce langage pourra être compilé par n'importe quel compilateur conforme à la norme
- La plupart des langages ne sont pas normalisés :
  - C possède une norme mais qui n'est pas toujours respectée
  - Java possède une norme qui n'est pas respectée par Microsoft
- ADA est un des rares langages généraliste normalisé
  - les compilateurs ADA sont certifiés par des tests de validité

Syntaxe de base de ADA (1/3)

- Le jeu de caractères de ADA est celui de la norme ISO 10646:2003
- Les *identificateurs* :

```
identifier ::= letter { underline | alphanumeric }

underline ::= _

alphanumeric ::= digit | letter

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

letter ::= A | B | ... | Z | a | b | ... | z
```

- ATTENTION : pas de différence entre majuscule et minuscule!
- Dans les chaines de caractères on peut aussi utiliser d'autres symboles de la norme
- Les *caractères spéciaux* (délimiteurs et opérateurs) :

```
" # & ' ( ) * + , - . / : ; < > = _
  |
```

Syntaxe de base de ADA (2/3)

- Les *délimiteurs et opérateurs longs* : `=>  ..  **  :=  /=  >=  <=  <<  >>  <>`

- Les *séparateurs* sont : l'espace, la tabulation, la fin de ligne

- Les *caractères et chaînes de caractères* :

```
character ::= 'graphic_character'

string ::= "{string_element}"

string_element ::= "" | non_quotation_mark_graphic_character

graphic_character ::= un des symboles de la norme ISO 10646:2003

non_quotation_mark_graphic_character ::= un graphic_character qui n'est pas "
```

- Les *commentaires* : débutent par --

Syntaxe de base de ADA (3/3)

- Les *nombres* :

```
numeric ::= decimal | based

decimal ::= numeral [ .numeral ] [ exponent ]

numeral ::= digit { [underline] digit }

underline ::= _

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

exponent ::= e_symbol [+] numeral | e_symbol - numeral

e_symbol ::= e | E

based ::= base # based_numeral  [.based_numeral] # exponent

base ::= numeral

based_numeral ::= extended_digit { [ underline ] extended_digit }

extended_digit ::= digit | A | B | C | D | E | F
```

- *Exemples* :

```
1_000
10E4
10e+4
1000.00
1_000.000_0
2#111110_1000#
```

Mots réservés de ADA 95

else	new	return
elsif	not	reverse
end	null	select
entry	of	separate
exception	or	subtype
exit	others	synchronized
for	out	tagged
function	overriding	task
generic	package	terminate
goto	pragma	then
if	private	type
in	procedure	until
interface	protected	use
is	raise	when
limited	range	while
loop	record	with
mod	rem	xor
	renames	
	requeue	

Structure d'un programme ADA (1/2)

- Un programme ADA comporte 4 parties :
  - une clause de contexte : importation de paquetages
  - un en-tête : déclaration du nom du programme
  - une partie déclarative
  - les instructions : cette partie commence par le mot-clé **begin** et finit par le mot-clé **end** suivi du nom du programme

```
with TEXT_IO,INT_IO, PPCM; use TEXT_IO, INT_IO; -- contexte
procedure mon_program is -- entête
   a,b,r : INTEGER; -- déclarations
begin -- début des instructions
   put("entrez deux entiers positifs : ");
   get(a);get(b);
   r := ppcm(a,b);
   put("le ppcm de "); put(a); put(" et "); put(b); put(" est "); put(r);
   new_line;
end mon_program; -- fin des instructions
```

Structure d'un programme ADA (2/2)

- *Note : une procédure est aussi un sous-programme (mais alors elle a des paramètres!)*
- Le corps d'un programme contient des appels à des sous-programmes (procédures et fonctions), écrits à part. ADA permet de séparer la déclaration des sous-programmes et leur implémentation (body).
- ADA est destiné à écrire un programme sous forme d'unités de compilation séparées (modularité) et de librairies de programmes
  - => découper au maximum les programmes, limiter au maximum la taille des unités de compilation, réutiliser au maximum du code existant
- La notion de paquetage existe en ADA et permet de regrouper des sous-programmes en ne laissant voir à l'extérieur que l'interface des procédures et fonctions (maintenabilité)

Structures de contrôle en ADA (1/4)

- *if then else* :

```
-- var est une variable entière
if var >= 2 then put("supérieur à 2");
elseif var = 0 then put("nul");
elseif var >= 0 then put("entre 0 et 2");
elseif var >= -2 then put("entre -2 et 0");
else put("inférieur à -2");
endif;
```

- *case* :

```
-- var est une variable entière
case var is
when 2 .. SYSTEM.MAX_INT => put("supérieur à 2");
when 0 => put("nul");
when 0 .. 1 | 1 .. 2 => put("entre 0 et 2");
when -2 .. 0 => put("entre -2 et 0");
when others => put("inférieur à -2");
encase;
```

Structures de contrôle en ADA (2/4)

- Les boucles de ADA

- Structure générale des boucles de ADA :

```
loop_statement ::= [ loop_name : ] [ iteration_scheme ] loop
                                    sequence_of_statements
                                end loop [ loop_name ] ;

iteration_scheme ::= while condition | for loop_parameter_specification

loop_parameter_specification ::= identifier in [ reverse ] discrete_interval

exit_statement ::= exit [ loop_name ] [ when condition ]  ;
```


Structures de contrôle en ADA (3/4)

- Les boucles peuvent être nommées, ce qui permet de préciser quelle boucle se termine en cas de boucles imbriquées.
- *boucle simple* : elle se termine
  - en cas d'instruction de retour si elle est utilisée dans une fonction
  - au cas où une exception est levée
  - en cas d'utilisation d'une condition **`exit`** vérifiée

```
var : INTEGER := 10;
loop
   put(var);put(" avant mise à feu"); var := var - 1;
   exit when var = -1;
end loop;
put("boum");
```

- *boucle "tant que"* :

```
var : INTEGER := 10;
mise_a_feu : while var >= 0 loop
   put(var);put(" avant mise à feu"); var := var - 1;
end loop mise_a_feu;
put("boum");
```

Structures de contrôle en ADA (4/4)

- *boucle "pour"* :
  - le paramètre de boucle est **local** (même si une variable de même nom a été déclarée auparavant) et sa valeur ne peut être modifiée dans la boucle (détection à la compilation).
  - Le paramètre de boucle doit prendre ses valeurs dans un **intervalle discret** (il est donc obligatoirement de type discret). L'intervalle est évalué en début de boucle et ne peut pas être modifié en cours de boucle.

```
for var in reverse 0 .. 10 loop
   put(var);put(" avant mise à feu");
end loop;
put("boum");
```

- *Note : en Java, on peut décrire l'intervalle de boucle à l'aide d'une variable, et le modifier en cours de boucle, on peut également modifier la valeur du paramètre dans la boucle.*

Affectation

- l'opérateur d'affectation est **:=**
- Le type de la variable cible doit être le **même** que celui de l'expression à évaluer (pas de conversion implicite!)
- Le contrôle de l'identité de type (entier, caractère, ...) est réalisé à la compilation
- Le contrôle des contraintes sur les types (intervalle, ...) se fait le plus souvent à l'exécution, le compilateur ajoutant dans le code des lignes de test.
- Il y a un contrôle de l'initialisation des variables (comme en Java) mais cela n'empêche pas la compilation (warning).