



Global Knowledge.



Support de cours

Table des matières

CHAPITRE I INTRODUCTION : QUELQUES DEFINITIONS	1
1. ALGORITHMME	2
2. ALGORITHMIQUE	2
3. METHODOLOGIE - METHODE	3
4. DÉMARCHES DE CONCEPTION	4
4.1. <i>Analyse descendante</i>	4
4.2. <i>Analyse ascendante</i>	5
5. LA PLACE DE L'ALGORITHMIQUE DANS LE PROJET	6
5.1. <i>Les différentes phases d'un projet informatique</i>	6
5.2. <i>La phase de Réalisation – Tests Unitaires</i>	6
CHAPITRE II CONSTITUANTS D'UN ALGORITHMME	9
1. PRESENTATION	10
2. L'ENCHAÎNEMENT	10
2.1. <i>Séquentiel</i>	10
2.2. <i>Sélectif</i>	11
2.3. <i>Répétitif</i>	12
2.4. <i>L'imbrication de structures</i>	12
3. LES DONNEES	13
3.1. <i>Type</i>	13
3.2. <i>Valeur</i>	13
3.3. <i>Identificateur</i>	13
3.4. <i>Déclaration de données</i>	14
4. LES ACTIONS	14
CHAPITRE III FORMALISME	15
1. ORDINOGRAMME	16
2. PSEUDO-CODE	17
2.1. <i>Éléments de pseudo-code</i>	18
2.2. <i>Structure générale d'un algorithme informatique</i>	19
CHAPITRE IV ALGÈBRE DE BOOLE	21
1. PRESENTATION	22
2. GENERALITES	22
3. LA CONJONCTION "ET"	23
4. LA CONJONCTION "OU"	24
5. LA CONJONCTION "NON"	25
6. RELATIONS FONDAMENTALES	26
7. CONCLUSION	27
CHAPITRE V COMPLÉMENTS ALGORITHMIQUES	29
1. DONNEES	30
1.1. <i>Types de données prédéfinis</i>	30
1.2. <i>Les tableaux</i>	32
1.3. <i>Structures de données</i>	34
1.4. <i>Objets</i>	36
2. ENCHAÎNEMENTS	37
2.1. <i>Choix multiple</i>	37
2.2. <i>Itération fixe</i>	38
2.3. <i>Débranchement inconditionnel</i>	39

CHAPITRE VI FICHIERS	41
1. FICHIERS.....	42
1.1. <i>Types d'organisation</i>	42
1.2. <i>Utilisation</i>	44
2. LA PERSISTANCE	48
2.1. <i>Définition</i>	48
2.2. <i>Le modèle des SGBDR</i>	48
2.4. <i>Les transactions</i>	52
2.5. <i>Algorithme d'accès au monde relationnel</i>	54
2.6. <i>Impact de la persistance pour le concepteur-développeur</i>	55
CHAPITRE VII MODULARITE OU « COMMENT DIVISER POUR MIEUX REGNER ».....	57
1. PRESENTATION	58
2. FONDEMENTS SYSTEMIQUES	59
3. TYPES DE MODULES	60
4. TRANSMISSION DE PARAMETRES	61
5. DECLARATIONS / APPELS DE MODULES	62
6. IMBRICATION DES MODULES ET PORTEE DES IDENTIFICATEURS	65
CHAPITRE VIII LES TESTS	69
1. TYPES DE TESTS	71
2. TECHNIQUES DE CONCEPTION.....	71
3. ETAPES DE CONCEPTION DES TESTS.....	72
ANNEXES	75
1. REPRESENTATION INTERNE DES DONNEES	77
1.1. <i>ORGANISATION DE LA MEMOIRE</i>	77
1.2. <i>LE CODAGE</i>	79
1.3. <i>REPRESENTATION INTERNE : DONNEES ALPHANUMERIQUES</i>	82
1.4. <i>REPRESENTATION INTERNE : DONNEES NUMERIQUES</i>	83
2. LES FICHIERS SEQUENTIELS	85
2.1. <i>GENERALITES</i>	85
2.2. <i>ORGANISATION DES DONNEES</i>	86
2.3. <i>LES SUPPORTS DES DONNEES</i>	95
2.4. <i>L'ORGANISATION SEQUENTIELLE</i>	98
2.5. <i>INTEGRITE DES DONNEES</i>	99

CHAPITRE I

INTRODUCTION : QUELQUES

DEFINITIONSERREUR ! SOURCE DU

RENGVOI INTROUVABLE.

1. ALGORITHME

Un algorithme est une description d'une action complexe, décomposée en une suite d'actions qui le sont moins.

Ce terme n'est pas purement informatique et peut s'appliquer à n'importe quelle classe de « problème », dans tous les domaines d'activité.

Par exemple, le mode d'emploi d'un répondeur téléphonique présente des algorithmes : les différentes étapes à suivre pour l'interroger à distance, enregistrer l'annonce...

La plupart du temps, *pour un problème* déterminé, il existe **plusieurs algorithmes possibles**.

Un algorithme peut être exprimé oralement, par un texte écrit, ou à l'aide d'un schéma.

2. ALGORITHMIQUE

L'algorithmique poursuit trois objectifs principaux :

- Représentation compréhensible par le plus grand nombre.
- Représentation aisément traduisible dans n'importe quel langage de programmation.
- Représentation structurée.

Ce dernier point est le plus important.

La programmation structurée n'est pas un phénomène de mode : les actions de maintenance de plus en plus nombreuses (les applications évoluent) ont prouvé qu'un programme coûte plus cher en évolutions qu'en développement initial.

Or bien souvent (c'est le cas de toutes les anciennes applications), les maintenances s'avèrent très difficiles - voire impossibles - à effectuer car les programmes ont été écrits sans aucune méthodologie.

3. METHODOLOGIE - METHODE

La **méthodologie** est l'ensemble des actions à entreprendre pour parvenir à un but. Elle répond aux questions

QUE FAUT-IL FAIRE, QUI LE FAIT et QUAND ?

La **méthode** est un ensemble de démarches raisonnées, suivies pour parvenir à un but. Elle répond à la question :

COMMENT FAUT-IL LE FAIRE ?

Ainsi, le but de la **programmation structurée** est de clarifier le texte d'un programme (le « source »), et ceci pour trois raisons principales :

- **Gagner du temps à l'écriture du programme :**

Une erreur commune est de considérer l'écriture d'un algorithme sur papier comme une perte de temps. En réalité, on s'aperçoit vite que cette remarque n'est justifiée que pour des programmes très simples, ne demandant pas d'effort particulier de réflexion.

Dans tous les autres cas, un **code structuré** permet au programmeur de toujours **maîtriser sa production**, même lorsque celle-ci devient complexe.

- **Améliorer la testabilité des programmes :**

Tester un programme équivaut à l'exécuter plusieurs fois en changeant les paramètres en entrée pour qu'au moins chaque combinaison d'instructions soit exécutée une fois.

Si le programme utilise un grand nombre d'instructions alternatives par exemple, les tests peuvent s'avérer longs et fastidieux.

Afin de faciliter cette étape primordiale, il est impératif que le source ait été écrit de manière logique pour pouvoir dresser une liste exhaustive des groupes d'instructions à tester.

- **Faciliter les maintenances futures :**

Les besoins du service utilisant l'application dans laquelle vient s'insérer ce programme peuvent évoluer.

Par ailleurs, un programme n'est jamais parfait lors de sa mise en production et présente toujours un pourcentage plus ou moins important d'erreurs en fonction de la qualité des tests qui ont été menés.

C'est pourquoi, un source fera toujours l'objet de maintenances (correctives ou évolutives). Le plus souvent, celles-ci sont effectuées par un développeur ne connaissant pas le programme à modifier.

Afin de faciliter le travail de l'équipe de maintenance, le code doit avoir été écrit de façon structurée pour être facilement compréhensible par tout autre développeur.

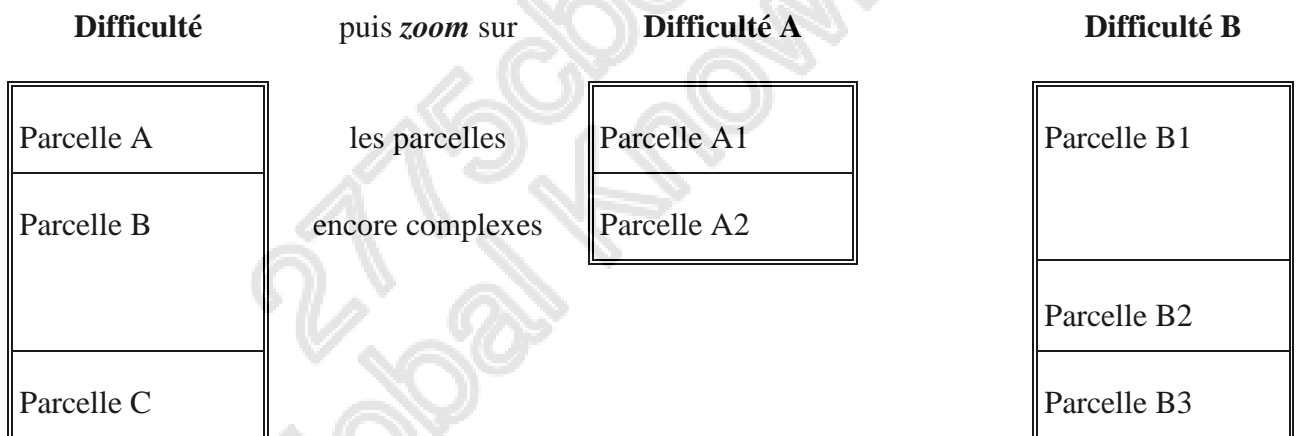
4. DÉMARCHES DE CONCEPTION

4.1. Analyse descendante

L'analyse « descendante » ou « Top-down » est une application de la deuxième règle du Discours de la Méthode de Descartes :

« Diviser chacune des difficultés examinées en autant de parcelles qu'il est requis pour les mieux résoudre »

L'analyse d'une tâche complexe consiste donc à la scinder en quelques sous-problèmes intermédiaires – dont la résolution est reportée à plus tard. Ceci suppose que l'on parte **d'une vue d'ensemble** du problème, pour « descendre » peu à peu **vers le détail** des éléments :



Nous appellerons ces parcelles des « **modules** ».

A chaque étape, il s'agit de **délimiter précisément le rôle qu'on attribue à chaque module**, sans se préoccuper de la façon dont on réalisera ces différentes fonctions.

4.2. Analyse ascendante

Le principe est fondamentalement différent puisque cette fois, on va partir du niveau de détail pour aboutir par regroupements successifs jusqu'à obtenir des fonctions suffisamment évoluées pour remplir les objectifs fixés au départ. »

Cette méthode présente l'inconvénient de ne pas permettre d'avoir une vue d'ensemble à tous les niveaux du développement, mais c'est au fur et à mesure que l'on progresse, que l'on prend conscience de la complexité du problème, ce qui ne permet guère de maîtriser les coûts et les délais de développement du système. Tant que l'on n'a pas pratiquement terminé, on ne sait pas vraiment quand on aura fini.

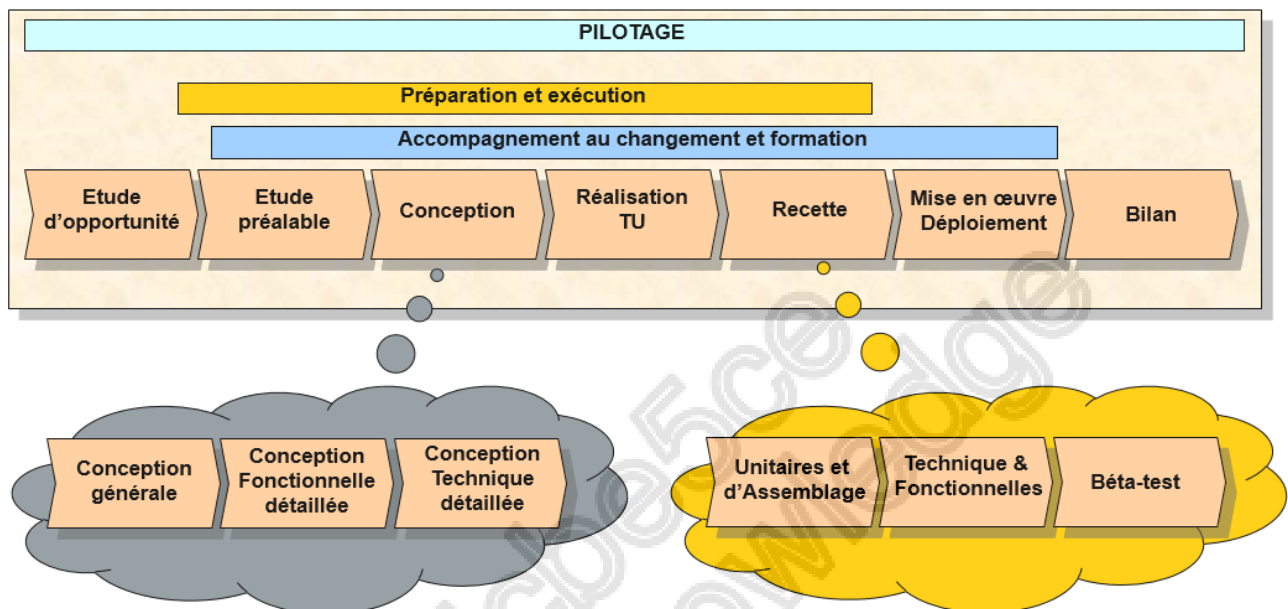
Cette méthode dite « Bottom-up » est utilisée en Conception, avant la phase de réalisation où se situe l'algorithme. C'est ainsi que l'on définit des modules réutilisables.

La méthode de *décomposition par étapes successives* inspirée de Descartes intervient à tous les niveaux : *de la conception des programmes à la définition des tests*.

Elle permet de guetter les éléments susceptibles d'être généralisés, tout en préservant l'autonomie des différents modules (en limitant le nombre de relations entre eux).

5. LA PLACE DE L'ALGORITHMIQUE DANS LE PROJET

5.1. Les différentes phases d'un projet informatique



5.2. La phase de Réalisation – Tests Unitaires

Réalisation du système : cette partie recouvre la construction du produit final respectant les fonctionnalités désirées par l'utilisateur et l'architecture technique mise en place.

La phase de réalisation recouvre les activités de :

- codage des programmes applicatifs, à partir **des algorithmes**,
- production de la documentation de programme, dans le but de faciliter les maintenances ultérieures,
- production des directives d'exploitation du système (politique de sauvegarde et de restauration, par exemple),
- production de guides utilisateurs et d'aides en ligne le cas échéant, voire des dispositifs de formation.

La phase de réalisation du système, acceptée au sens large, recouvre aussi les **tests de l'application**.

Plusieurs activités de test sont nécessaires pour garantir l'adéquation du système aux besoins exprimés :

- des **tests unitaires**, au cours desquels les différents programmes sont testés indépendamment les uns des autres, tests menés à partir de jeux d'essais, c'est-à-dire de cas d'utilisation conçus spécifiquement dans un but de validation,
- des **tests d'intégration**, tests de fonctionnalités qui enchaînent les différents programmes d'une même unité fonctionnelle, puis les unités fonctionnelles entre elles, menées, eux aussi, sur la base de jeux d'essai,
- des **tests systèmes**, dont l'objectif est de vérifier la capacité du système à supporter la future application,
- des **tests opérationnels**, qui sont menés à partir de données et de contextes de production.

Le terme de **recette** désigne l'ensemble des tests que pratiquent les futurs utilisateurs d'un système lors de la livraison de l'application par les informaticiens. Ce terme implique un cadre contractuel.

CHAPITRE II

CONSTITUANTS D'UN ALGORITHME

1. PRESENTATION

Un algorithme est la décomposition d'une action complexe qui, partant de données toutes définies, permet d'obtenir un (des) résultat(s) déterminé(s). Tout algorithme sera donc composé :

- de structures d'enchaînements,
- d'actions, portant sur
- des données

2. L'ENCHAINEMENT

Il existe trois familles d'enchaînement d'actions :

- séquentiel
- sélectif
- répétitif

2.1. Séquentiel

Les actions sont effectuées l'une après l'autre, du début à la fin du **bloc logique** qu'elles forment (l'action plus complexe) :

Début

↓ Exécuter telle action

↓ Exécuter telle autre action

Fin du bloc

Par exemple : Appeler quelqu'un au téléphone :

{
Décrocher le combiné
Attendre la tonalité
Composer le numéro

2.2. Sélectif

L'alternative permet de choisir d'exécuter un bloc d'actions plutôt qu'un autre et peut s'exprimer sous la forme :

SI la condition est vérifiée

Exécuter tel bloc d'actions

SINON

Exécuter tel autre bloc

Par exemple : Multiplier 2 nombres :

SI l'opération paraît simple

La faire de tête

SINON

La poser par écrit

Un bloc peut tout à fait ne comporter aucune action :

Par exemple, en faisant des courses :

SI le produit convient

Le prendre du rayon

SINON

RIEN

Remarque : Dans ce cas, la branche « SINON » peut ne pas être représentée.

2.3. Répétitif

Les « boucles » servent à attendre un état prédéterminé, en répétant un bloc d'actions 0, 1, ou plusieurs fois, selon l'expression choisie :

Les deux boucles les plus courantes sont :

⇒ REPETER

Exécuter tel bloc d'actions	<i>Le bloc est toujours exécuté au moins une fois</i>
JUSQU'À ce que la condition soit vérifiée	<i>Condition d'arrêt de la boucle</i>

Attention : avec un REPETER, le test sur la condition d'arrêt se fait **après** l'exécution du bloc d'action. Cette boucle s'effectue donc de **1 à plusieurs fois**.

⇒ TANT QUE la condition est vérifiée

*Condition de **continuation** de la boucle*

Exécuter tel bloc d'actions	<i>Le bloc peut ne jamais être exécuté</i>
-----------------------------	--

Attention : avec un TANT QUE, le test sur la condition d'arrêt se fait **avant** l'exécution du bloc d'action. Cette boucle s'effectue donc de **0 à plusieurs fois**.

Exemple, prendre un repas :

TANT QUE j'ai faim

Manger un peu

REPETER

Manger un peu

JUSQU'À ne plus avoir faim

2.4. L'imbrication de structures

Attention : L'imbrication de structures d'enchaînement est à utiliser avec discernement et modération :

Trop de niveaux (plus de 3) entraînent de très sérieux problèmes de compréhension.

3. LES DONNEES

Une donnée est une **information nécessaire** au processus décrit. Toute donnée se définit par **trois caractéristiques** :

3.1. Type

Le type d'une donnée désigne

- L'ensemble *des valeurs* qu'elle peut prendre.
Exemples de types : entier, âge, heure, carte de tarot, booléen^(*)...
- Les *opérations* possibles avec elle.
Exemples : nom + prénom (mettre à la suite)
 prix + 20.6 % (ajouter)
 habiter Paris ET avoir moins de 25 ans ^(**).

3.2. Valeur

Celle-ci peut être **constante** (3.1416, 20.6...) ou **varier** pendant l'action – à la manière et dans les limites de son type.

3.3. Identificateur

C'est le **mot** qui symbolise la donnée. Plus les identificateurs sont imagés, plus l'algorithme est compréhensible :

Exemples : prénom, lampe_allumée, pi, taux_TVA...
valent mieux que : X, CDVLI, truc, compteur...

Remarque : En nommant les constantes, vous généraliserez à peu de frais vos algorithmes.

(*) Un booléen n'admet que 2 valeurs qui s'excluent mutuellement : Vrai ou Faux, 0 ou 1, Allumé ou Eteint

(**) Une condition est une expression booléenne : une (suite d')opération(s) dont le résultat est Vrai ou Faux.

3.4. Déclaration de données

Tout algorithme comporte une partie « *déclarative* » qui recense et définit l'ensemble des données qu'il utilise, c'est son *dictionnaire des données*.

Pour chacune, on y indique :

- son identificateur,
- le type et la longueur de la donnée,
- sa valeur - s'il s'agit d'une constante,
- éventuellement son nombre d'éléments

Par exemple :

Prénom prénom	caractères	20	<i>c'est-à-dire 20 caractères maximum pour un</i>
Age	numérique	3	<i>c'est-à-dire 3 chiffres maximum pour l'âge</i>
Lampe_allumée	booléen		
Pi	numérique	= 3.1416	

4. LES ACTIONS

Quand doit-on arrêter de décomposer les actions en actions plus élémentaires ?

- ➔ Quand chacune d'entre elles est connue de celui à qui l'on s'adresse, c'est à dire quand ces actions sont traduisibles dans le langage de programmation concerné.

Vous trouverez dans tout langage de programmation au moins **trois instructions de base** :



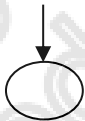
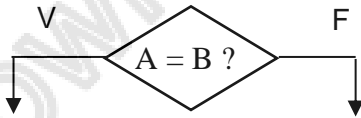
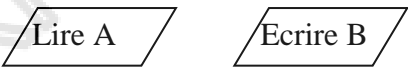
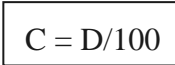
- Lire la valeur d'une variable (au clavier, sur le disque...)
- Mémoriser ou modifier la valeur d'une donnée (faire des calculs...)
- Ecrire la valeur d'une donnée (à l'écran, sur le disque, à l'imprimante...)

CHAPITRE III FORMALISME

1. ORDINOGRAMME

Un « **ordinogramme** », encore appelé « **organigramme** », représente chaque élément de l'algorithme par un schéma.

Il existe des schémas normalisés pour chaque type d'élément.

Début / Fin du processus	
Enchaînement	
Branchement éloigné	
Expression d'une condition	
Opération d'Entrée / Sortie	
Affectation de valeur	

Remarque importante : Ce type de représentation peut parfaitement convenir dans le cadre de l'expression structurée de la pensée. Cependant, si cette dernière ne l'est pas, l'ordinogramme sera brouillon et ne portera pas en lui-même des éléments de structuration. En effet, l'ordinogramme servait à l'analyse ascendante (c'est à dire du détail vers le général). C'est pourquoi ce formalisme est considéré plutôt comme un bon outil de communication – et non comme un outil de conception.

2. PSEUDO-CODE

C'est une représentation écrite des éléments constitutifs d'un algorithme.

Chaque élément constituant l'algorithme **est défini par un terme représentatif et non ambigu**.

Les différents termes du pseudo-code représentent donc les éléments d'un langage qui permet de décrire l'algorithme de *résolution* d'un problème ; par opposition au « langage » de programmation - il s'agit plutôt de code – qui transcrit l'algorithme en instructions exécutables par une machine.

- Le premier est compréhensible par l'homme, qu'il soit utilisateur, analyste ou développeur,
- le second n'est accessible qu'au développeur, qu'à l'ordinateur et, dans le meilleur des cas, à l'analyste.

En programmation classique, l'utilisation de ces différents éléments est peu limitée.

C'est un synonyme de liberté pour le développeur, mais également de difficultés. En effet, le mécanisme de réflexion qui conduit à la résolution d'un problème est loin d'être un mécanisme structuré. Un programme écrit au fur et à mesure de cette réflexion aboutirait vraisemblablement à un codage très complexe dans lequel les instructions figureraient dans un ordre plus ou moins anarchique, et risqueraient d'être fortement dépendantes les unes des autres.

L'expérience prouve qu'à court ou moyen terme, c'est l'utilisation très bien définie d'un petit nombre d'opérations élémentaires de représentation des algorithmes qui simplifie leur expression, facilitant ainsi tant leur conception et leur mise au point, que leur maintenance.

En 1966, Böhm et Jacopini ont démontré qu'il est possible de décrire tout processus algorithmique en n'utilisant que :

- l'enchaînement séquentiel,
- et la boucle « Tant que ».

2.1. Éléments de pseudo-code

Il n'existe pas à l'heure actuelle de norme internationale syntaxique du pseudo-code. Nous en proposerons un exemple qui respecte l'esprit du langage, notamment au niveau des structures d'enchaînement :

Alternative	SI <condition> traitement 1 SINON traitement 2 FIN-SI
Itération	TANT QUE <condition> Traitement FIN-TANT-QUE REPETER Traitement JUSQU'A <condition>
Affectation de valeur	$A \leftarrow B$ $A \leftarrow B / 100$
Lecture d'une valeur (du clavier, du disque...)	LIRE A
Ecriture d'une valeur (à l'écran, sur le disque, ...)	AFFICHER A (à l'écran) ECRIRE A (sur le disque)

Remarques :

- « FIN-SI » clôture toujours l'alternative « SI », « FIN-TANT-QUE » la répétitive « TANT QUE »
- un trait vertical relie SI ... SINON ... FIN-SI, TANT QUE ... FIN-TANT-QUE, REPETER ... JUSQU'A
- les traitements imbriqués sont écrits en retrait

2.2. Structure générale d'un algorithme informatique

Tout algorithme informatique est composé de **3 parties** :

- Un **entête** qui sert notamment à le nommer et préciser son utilité.
- Une **partie déclarative** (ou **dictionnaire des données**) qui recense et définit tous les objets qu'il utilise.

Chaque donnée doit y être déclarée en précisant :

- son identificateur,
 - son type,
 - éventuellement sa longueur,
 - s'il s'agit d'une donnée constante, sa valeur.
- Une **partie exécutable**, le « **corps** », qui indique les actions à exécuter avec les objets décrits.

Remarque : Des (* commentaires *) peuvent et **doivent être** intercalés pour la clarté du programme.

Exemple :

MODULE DEVINETTE

(* Entête*)

(* L'utilisateur doit deviner un nombre secret : *)
 (* S'il le trouve, on lui dit en combien de coups ; s'il abandonne, on divulgue le secret *)

(* Le dictionnaire des données*)

CONSTANTE secret entier = 723

VARIABLES jouer caractère 1 (* données saisies *)
 nombre entier
 nb_coups entier (* donnée calculée *)

(* Le corps*)

nb_coups ← 0

REPETER

```

  (* Demander s'il veut (re)jouer *)
  ECRIRE « Voulez-vous deviner mon nombre secret ? »
  ECRIRE « Tapez O pour oui, n'importe quelle autre touche pour non »
  LIRE jouer
  SI jouer = 'O'
  |
  |   ECRIRE « Tentez votre chance : »
  |   LIRE nombre
  |   nb_coups ← nb_coups + 1
  |   SI nombre < secret
  |   |   ECRIRE « Trop petit ! »
  |   SINON SI nombre > secret
  |   |   ECRIRE « Trop grand ! »
  |   FIN-SI
  FIN-SI
  FIN-SI

```

JUSQU'A nombre = secret OU jouer <> 'O'

SI nombre = secret
 | ECRIRE « Vous avez trouvé en », nb_coups, « coups »
 SINON
 | ECRIRE « Mon nombre secret est », secret
 FIN-SI

FIN-MODULE-DEVINETTE

CHAPITRE IV

ALGEBRE DE BOOLE

1. PRESENTATION

L'Algèbre de Boole est une méthode de raisonnement qui présente deux intérêts importants :

- elle permet de représenter par une notation simple et précise toutes les propositions impliquées dans un raisonnement,
- elle permet de remplacer un raisonnement logique par un calcul et de bénéficier des automatismes que celui-ci permet.

2. GENERALITES

Soit E un ensemble d'éléments $e-i$ et une propriété p .

Chaque élément $e-i$ a ou n'a pas la propriété p .

A chaque $e-i$, on associe une variable, dite variable de Boole, qui vaut :

- 1 si p est vraie,
- 0 si p est fausse.

Exemple :

E est l'ensemble des concepteurs-développeurs d'un service informatique, e est un élément de cet ensemble, p est la propriété « avoir déjà travaillé sur un ordinateur z/OS ». Appelons « a » la variable de Boole correspondante.

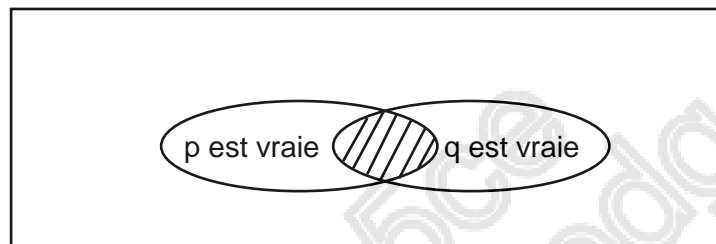
« a » est bien une variable, car sa valeur est fonction de l'individu considéré :

- 1 s'il a travaillé sur z/OS,
- 0 s'il n'a pas travaillé sur z/OS.

3. LA CONJONCTION "ET"

Considérons maintenant deux propriétés de chacun des éléments du même ensemble. Par exemple, connaître z/OS et connaître JAVA.

Nous avons alors un autre sous-ensemble qui est l'ensemble des Concepteurs-Développeurs ayant déjà travaillé sur JAVA. Cette propriété sera notée q , la variable de Boole associée étant « b ».



Dans la zone hachurée, les propriétés p **et** q sont vraies, et là seulement. On peut encore dire que dans cette zone les variables a **et** b sont égales à 1, et là seulement.

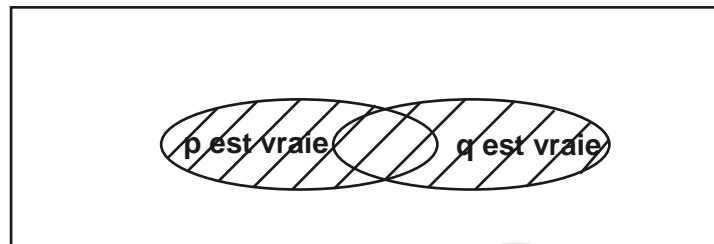
La table de vérité permet de recenser tous les cas de combinaison possibles :

a	b	a ET b
1	1	1
0	1	0
1	0	0
0	0	0

Par convention, la conjonction **ET** est souvent notée « **x** » ou « **.** »

4. LA CONJONCTION "OU"

Nous considérons toujours les deux mêmes propriétés des éléments du même ensemble.



Dans la zone hachurée, les propriétés p ou q sont vraies : chacun des points de cet ensemble représente un concepteur-développeur qui possède au moins une des deux propriétés énoncées (et peut-être les deux) : il s'agit d'un **OU inclusif**.

Table de vérité :

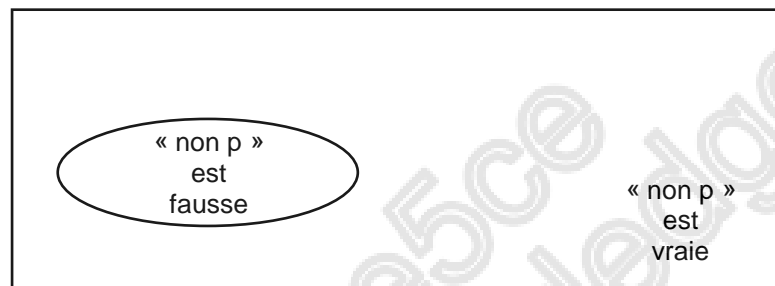
a	b	a OU b
1	1	1
0	1	1
1	0	1
0	0	0

Par convention, la conjonction **OU** est souvent notée « + »

5. LA CONJONCTION "NON"

A chaque propriété p nous associons la propriété « non p ». Nous dirons que « non p » est vraie quand « p » est fausse et réciproquement.

Dans notre exemple « non p » est la propriété « ne jamais avoir travaillé sur z/OS ».



Par convention, $\text{NON}(p)$ est souvent notée \overline{p}

6. RELATIONS FONDAMENTALES

$$\begin{array}{ll}
 a \cdot a = a & a + a = a \\
 1 \cdot a = a & 1 + a = 1 \\
 0 \cdot a = 0 & 0 + a = a \\
 \overline{a \cdot a = 0} & \overline{a + a = 1} \\
 \overline{\overline{a = a}} &
 \end{array}$$

6.1.1. ASSOCIATIVITE

$$\begin{aligned}
 (a + b) + c &= a + (b + c) \\
 (a \cdot b) \cdot c &= a \cdot (b \cdot c)
 \end{aligned}$$

6.1.2. COMMUTATIVITE

$$\begin{aligned}
 a + b &= b + a \\
 a \cdot b &= b \cdot a
 \end{aligned}$$

6.1.3. DISTRIBUTIVITE

$$\begin{aligned}
 a \cdot (b + c) &= (a \cdot b) + (a \cdot c) \\
 a + (b \cdot c) &= (a + b) \cdot (a + c)
 \end{aligned}$$

6.1.4. ABSORPTION

$$\begin{aligned}
 a + (a \cdot b) &= a \\
 a \cdot (a + b) &= a \\
 \overline{a + (a \cdot b)} &= \overline{a + b}
 \end{aligned}$$

Loi de MORGAN :

$$\overline{a \text{ ou } b} = \overline{a} \text{ et } \overline{b}$$

$$\overline{a \text{ et } b} = \overline{a} \text{ ou } \overline{b}$$

Exemple : La négation de $(A = B) \text{ ET } (C = D)$
est $(A \neq B) \text{ OU } (C \neq D)$

7. CONCLUSION

Nous avons défini ci-dessus les règles de calcul qui permettent de simplifier les expressions booléennes.

Nous avons donc la possibilité d'exprimer une propriété à partir de propriétés plus « atomiques », puis par une réduction des termes semblables, une application des identités remarquables, etc. Nous obtiendrons une nouvelle expression de la propriété souvent plus maniable.

Nous avons remplacé le cheminement logique de la pensée par un mécanisme de calcul.

CHAPITRE V

COMPLEMENTS ALGORITHMIQUES

1. DONNEES

1.1. Types de données prédéfinis

Dans tout langage de programmation, il existe des types de données prédéfinis. Les plus courants sont numériques, alphanumériques et booléens.

1.1.1. Le type numérique

Dans quel cas déclare-t-on une variable en numérique ?

→ Uniquement quand cette variable est **destinée à des opérations arithmétiques**.

Une donnée numérique peut contenir une valeur entière ou décimale, signée ou non. Sa longueur doit indiquer le nombre maximal de chiffres (« digits ») pouvant figurer avant et après la virgule.

Par exemple, la déclaration :

Compteur NUMERIQUE 5

indique que cette variable pourra contenir 99999 ou -123
mais pas 123456

En ce qui concerne les réels, l'interprétation dépendra du langage

Montant NUMERIQUE 7, 2

- variable pouvant contenir 7 digits au total **dont** 2 décimales, c'est-à-dire 12345,67 ou -123 mais pas 12345678 ni 1,234
- variable pouvant contenir 7 digits pour la partie entière **plus** 2 décimales.

1.1.2. Le type caractère ou « alphanumérique »

Une donnée de type caractère (encore appelé type « alphanumérique ») peut contenir des lettres, des chiffres et tout autre symbole que l'on peut obtenir au clavier.

Ce type est également **ordonné** par une table où sont recensées toutes les valeurs :

- table ASCII sur la plupart des systèmes
- table EBCDIC dans le monde IBM

C'est pourquoi :

'a' < 'b' 'a' <> 'A' '9' > '8' ...

Remarque : Notons que si 8 (sans apostrophe) représente la valeur entière (constante numérique), '8' (avec apostrophes) représente le caractère (constante alphanumérique), et ne sont donc pas comparables.

La longueur d'une chaîne de caractères indique le nombre maximal de caractères que la chaîne peut comporter.

La plupart des langages de programmation ont défini pour **opérations de type caractère** :

- calculer la longueur réelle d'une chaîne (syntaxe : long(chaine))
- « concaténer » 2 chaînes (mettre bout à bout)
- « extraire » une sous-chaîne (isoler un sous-ensemble d'une chaîne)

1.1.3. Le type booléen

Un booléen n'admet que 2 valeurs qui s'excluent mutuellement :

- **vrai / faux**,
- **0 / 1** (0 signifiant « faux » et 1 signifiant « vrai »).

Par exemple, la déclaration :

Fin-fichier booléen

Si fin-fichier = 0 → ce n'est pas la fin de fichier,

Si fin-fichier = 1 → c'est la fin de fichier.

1.2. Les tableaux

Un tableau est une donnée composée d'un **nombre défini d'éléments** :

- **de même type et**
- **de même longueur.**

Chaque **élément** (ou « **poste** ») est identifié par un numéro : son « **indice** ».

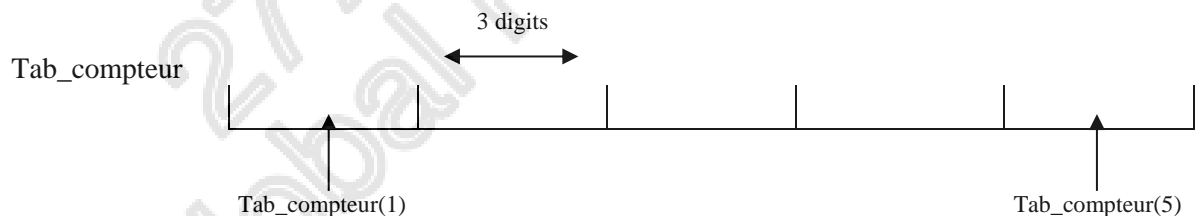
Selon les langages de programmation, le nombre d'éléments est indiqué soit directement, soit en précisant les bornes de l'indice. Nous choisirons ici de ne préciser que le nombre d'éléments – le premier étant à l'indice 1.

Par exemple, la déclaration :

```
Tab_Compteur [ 5 ] NUMERIQUE 3
```

indique que le tableau Tab_compteur est composé de 5 postes, chacun déclaré en numérique de 3.

Représentation du tableau Tab_compteur :



Tab_compteur(1), Tab_compteur(5) sont maintenant des identificateurs d'entiers, qui se manipulent comme des identificateurs habituels, sauf que l'indice peut être remplacé par une variable numérique, ce qui va permettre d'automatiser certains traitements (ex : Tab_compteur(I)).

Attention : Le développeur doit **empêcher les débordements de tableau** car les problèmes engendrés sont imprévisibles. Il est donc interdit de trouver dans un programme Tab_compteur(I) avec $I = 0$ ou, dans notre exemple, avec I supérieur à 6 !

Il est à noter que certains langages démarrent leur comptage à 0. Dans ce cas, dans notre exemple les limites seraient Tab_compteur(0) à Tab_compteur(4).

Exemple : Opérer une remise de 10% sur le montant de 50 factures

Dictionnaire des données :

Nb_max	NUMERIQUE	=	50
No_fac	NUMERIQUE		2
Tab_montants [Nb_max]	NUMERIQUE		10,2

Corps :

.../...

No_fac ← 1

REPETER

	Tab_montants (No_fac)	←	Tab_montants (No_fac) * 0,9
	No_fac	←	No_fac + 1

JUSQU'à No_fac > Nb_max

Beaucoup de langages de programmation admettront des tableaux à **2 dimensions**, certains même à **N dimensions** (en permettant de construire des tableaux de tableaux).

Certains permettent de mettre en œuvre des tableaux « dynamiques » (dont le nombre maximum de postes n'est pas une constante).

1.3. Structures de données

Il est possible dans la plupart des langages informatiques de déclarer des variables qui ont **un sens dans leur globalité**, mais qui ont aussi une signification lorsque l'on s'intéresse aux **caractéristiques élémentaires qui la composent**.

Quel est la différence entre une structure de données et un tableau ?

- Une **structure de données** contient des variables de **nom** et de **type différents**
- Un **tableau** contient des postes de **même nom, même type et même longueur**.

Par exemple, le numéro de sécurité sociale à 13 chiffres représente l'identifiant unique d'un assuré social. Mais...

- le premier chiffre représente le sexe de l'assuré ;
- les quatre suivants, l'année
- puis le mois de sa naissance ;
- les deux suivants son département de naissance ;
- etc.

Pour pouvoir manipuler aussi bien l'identifiant de l'assuré que les quatre chiffres précisant sa date de naissance, plutôt que de déclarer deux variables physiquement distinctes et qu'il faudra valoriser en parallèle, on peut indiquer la décomposition de la variable de la manière suivante :

```
VARIABLE no_sécu
    sexe    CARACTERE    1
    année   NUMERIQUE    2
    mois     NUMERIQUE    2
    départ  CARACTERES   2
    ville    CARACTERES   3
    no_seq   NUMERIQUE    3
```

Nous pouvons améliorer la description de notre structure en imaginant que le mois et l'année vont aussi être manipulés en bloc (à des fins d'affichage par exemple) et que le département, la ville et le numéro séquentiel n'ont par contre pas besoin d'être détaillés puisqu'ils ne seront pas exploités indépendamment des autres caractères du numéro de sécurité sociale. La structure deviendra donc :

```
VARIABLE no_sécu
    sexe          CARACTERE    1
    date_naiss
        année     NUMERIQUE    2
        mois      NUMERIQUE    2
    reste         CARACTERES   8
```

Remarque : Les rubriques d'une donnée structurée peuvent avoir des types différents les uns des autres.

Dans beaucoup de langages de programmation, on peut définir au préalable un nouveau type, en indiquant pour chacun de ses membres son type et sa dimension ; puis déclarer une (ou des) variable(s) ayant ce nouveau type.

Exemple :

```

TYPE noss
    sexe          NUMERIQUE  1
    date_naiss
        année     NUMERIQUE  2
        mois      NUMERIQUE  2
    reste         NUMERIQUE  8

TYPE employe
    no_secu       noss
    nom           CARACTERES  20
    prénom        CARACTERES  15
    adresse
        libellé1   CARACTERES  40
        libellé2   CARACTERES  40
        cod_post   CARACTERES  5
        ville      CARACTERES  20

VARIABLE Var_emp1  employe
VARIABLE Var_emp2  employe

```

Quelle que soit la technique de déclaration, il faudra, pour accéder aux membres eux-mêmes, utiliser une notation préfixée (ou notation “pointée”) :

```

Var_emp1.nom  ←  'Untel'
Var_emp2.adresse.cod_post  ←  '75010'

```

Enfin, il est toujours possible de déclarer un **tableau de structures de données** :

```
Tab_emp [10]  employe
```

Puis, pour l'utiliser :

```
Tab_emp(3).no_secu.sexe  ←  1
```

1.4. Objets

Il est possible dans les langages informatiques orientés objets de déclarer des variables qui ont comme type des classes. La **classe** est une définition, un modèle caractérisant un ensemble d'éléments du système d'information. Ces définitions existent en amont des algorithmes et peuvent être définies par des concepteurs-développeurs et/ou faire parties du langage objet.

En plus de mémoriser des données, que l'on nomme des attributs, la classe stocke aussi les traitements possibles à effectuer, que l'on nomme des méthodes. **L'objet** est un représentant de la classe que l'on stocke en variable, à l'identique des variables de type booléen, numérique...

Par exemple nous disposons d'une classe Date contenant en attributs : le jour, le mois, l'année et le siècle. En méthode, cette classe dispose d'un traitement nommé `retourneLibelle` nous permettant d'obtenir le libellé du mois (sous forme de caractères)¹.

```
La classe Date { * attributs *
                siècle      NUMERIQUE  2
                année      NUMERIQUE  2
                mois       NUMERIQUE  2
                jour       NUMERIQUE  2

                * méthode *
                Alphanumérique retourneLibelle()
            }
```

On peut donc déclarer une variable `dateNaissance` de type `Date`, puis l'initialiser et la manipuler

```
Date dateNaissance
    dateNaissance ← 01072016
    dateNaissance.jour ← 31
```

On peut déclarer une variable `libelle` de type caractères puis l'initialiser par la variable `dateNaissance`

```
Alphanumérique libelle
Libelle ← dateNaissance.retourneLibelle()
```

`Libelle` prend la valeur `Juillet`.

¹ Nous détaillons par la suite les modules, les passages de paramètres et les retours d'informations.

2. ENCHAINEMENTS

2.1. Choix multiple

Il s'agit d'un ensemble de traitements *exclusifs*, conditionnés par la valeur d'une variable commune :

```
CHOIX SELON variable
|   valeur_1   traitement 1
|   valeur_2   traitement 2
|   valeur_3   traitement 3
|   ...
|   AUTREMENT  traitement N
FIN-CHOIX
```

Cette construction est équivalente à plusieurs alternatives *imbriquées* : si une même valeur est citée plusieurs fois, seul le traitement face à sa première apparition est exécuté.

L'option « autrement » est facultative.

Ceci aurait donc pu être exprimé sous la forme :

```
SI   variable = valeur_1
|   traitement 1
SINON SI variable = valeur_2
|   traitement 2
|   SINON SI variable = valeur_3
|   |   traitement 3
|   |   SINON
|   |   traitement N
|   |   FIN-SI
|   FIN-SI
FIN-SI
```

2.2. Itération fixe

Cette boucle est utile quand on connaît le nombre de fois où l'on voudra exécuter un traitement :

```
POUR indice DE valeur_init À valeur_finale
|   Bloc d'instructions
FIN-POUR
```

La valeur finale est déterminée une seule fois, à l'entrée de la boucle.

L'indice reçoit la valeur initiale, puis varie à chaque tour par pas de +1 (par défaut).

Exemple : Lire un livre

```
Nb_pages ← 100
POUR no_page DE 1 À Nb_pages
|   Lire la page
FIN-POUR
```

Aurait dû s'écrire – si la boucle POUR n'existait pas :

```
Nb_pages ← 100
No_page ← 1
TANT QUE no_page ≤ Nb_pages
|   Lire la page
|   No_page ← No_page + 1
FIN-TQ
```

Attention : En général, ni la variable indice, ni la borne finale, ne doivent être modifiés par la ou les instructions du bloc, sous peine de résultats imprévisibles.

2.3. Débranchement inconditionnel

Une des structures de commandes répandues de l'algorithmique est le branchement inconditionnel.

À l'exécution d'un programme, lorsqu'une instruction a été exécutée, le registre d'adresse d'instruction contient l'adresse de l'instruction suivante.

Pour modifier cette adresse, c'est-à-dire pour rompre l'exécution séquentielle, beaucoup de langages informatiques proposent une instruction permettant de se débrancher à un point quelconque du source, repéré par une « étiquette ».

En pseudo-code, cette instruction a la forme suivante :

```
A ← B  
  
ALLER À Suite  
  
...  
Suite  
  
B ← C
```

Le débranchement s'effectue à l'instruction qui suit l'étiquette spécifiée dans l'instruction « **ALLER À** ».

Dans un ordinogramme, le débranchement est simplement représenté par une flèche remontant ou descendant dans le schéma.

Remarques générales sur l'utilisation du « ALLER À » :

Toutes les études faites sur les structures de programme montrent que l'instruction ALLER À est inutile et plutôt néfaste. Disons tout de suite qu'elle n'empêche pas un programme d'être structuré, de même qu'un programme écrit sans instruction de branchement inconditionnel peut être mal structuré.

La polémique qui s'est développée à propos de cette instruction vient du fait que certains langages parmi les plus utilisés (BASIC, COBOL) ne possédaient pas les instructions correspondant aux différentes structures de commande de l'algorithmique, notamment la répétitive, et que dans ces conditions, le ALLER À était nécessaire pour les traduire.

Si le ALLER À est parfois nécessaire, pourquoi vouloir l'éliminer ? Parce qu'il **facilite un raisonnement suivant le libre cours de la pensée qui, par essence, n'est pas structurée.**

En effet, nous avons déjà vu que lorsque l'on travaille sur la résolution d'un problème, surtout s'il est un peu complexe, on éprouve des difficultés à envisager l'ensemble des données de départ, l'ensemble des résultats nécessaires et l'ensemble des conditions et actions à appliquer au premier pour aboutir au second.

Il résulte de ceci que l'on commence par une solution généralement incomplète, que l'on perfectionne à coup de branchements intempestifs qui aboutissent à une solution, satisfaisante fonctionnellement, mais qui sera difficile à mettre au point à cause des interactions multiples et mal cernées entre les groupes d'instructions, et encore plus à modifier ultérieurement.

De plus, ce mode de raisonnement d'apparence très libre, va obliger le programmeur à réinventer une solution nouvelle à chaque fois qu'il rencontrera un nouveau problème à résoudre.

En éliminant les « ALLER À » et en fournissant un petit nombre de structures de base, l'algorithmique va favoriser un mode de raisonnement structuré, standardisé, permettant d'améliorer la productivité du programmeur et de faciliter la mise au point ainsi que les futures maintenances.

Toutefois nous ne supprimerons pas totalement les instructions ALLER À car, dans le cadre des langages dont nous disposons, cela entraîne parfois la répétition inutile de conditionnements tout au long d'un module. Nous n'admettons donc exclusivement qu'une forme de ALLER À :

le ALLER À descendant, se branchant exclusivement à l'intérieur du même module.

Toute autre utilisation sera exclue et nous réfléchirons sur son utilité à chaque fois que le cas se présentera, car, répétons-le, son utilisation peut toujours être évitée.

CHAPITRE VI

FICHIERS

1. FICHIERS

Un fichier est un ensemble d'informations stocké sur un (ou plusieurs) disque(s), délimité par un label de début et un label de fin.

Un fichier peut stocker aussi bien des données que des programmes.

De l'organisation des informations au sein du fichier, découlent différents modes d'accès possibles.

1.1. Types d'organisation

1.1.1. Les fichiers texte

Ce sont ceux que vous pouvez manipuler avec un éditeur ou un traitement de texte. Dans de tels fichiers, les informations sont stockées sous forme de chaînes de caractères ; leur contenu est donc lisible.

Ces fichiers sont structurés par lignes (en général délimitées par le caractère « Retour chariot » suivi du caractère « fin de ligne »).

Cette organisation empêche de demander au système, par exemple, de lire la 3^{ème} ligne. En effet, les lignes n'ont a priori pas de taille fixe, on ne peut donc présupposer de l'emplacement du début d'une ligne particulière – sauf à parcourir le fichier ligne à ligne, ce qui est peu performant.

Remarque : le clavier et l'écran sont considérés comme des fichiers texte (Entrée/Sortie standard) que le système ouvre automatiquement lors du démarrage d'un programme – respectivement en lecture et écriture – et ferme à la fin.

1.1.2. Fichiers séquentiels avec enregistrements

Ces fichiers contiennent des données ayant toutes le même type, le plus souvent, regroupées en **enregistrements** (records).

Les données de ces fichiers sont enregistrées physiquement dans **l'ordre chronologique** de leur écriture et seront restituées, en lecture, dans le même ordre.

Lorsqu'un fichier est organisé en enregistrements, il faudra, à l'utilisation, en indiquer la structure (le « masque »).

Si ces enregistrements sont de format fixe (de même longueur), il sera possible d'accéder directement à un élément par son numéro d'ordre dans le fichier.

1.1.3. Fichiers séquentiels indexés

Cette fois, on associe un fichier « **index** » au fichier séquentiel typé, qui permettra de retrouver rapidement un élément dans le fichier séquentiel :

On enregistre dans ce fichier index uniquement les valeurs de la (ou des) donnée(s) servant de critère de recherche (la « clé »), accompagnées de l'adresse physique du ou des enregistrements correspondants dans le fichier séquentiel.

Un index est en général un arbre binaire, structure permettant une recherche dichotomique (bien plus rapide qu'une recherche séquentielle).

1.1.4. Les bases de données

Un système de gestion de bases de données est un logiciel qui assure notamment l'interface entre utilisateurs (ou développeurs) et fichiers physiques.

Ceux-ci n'ont donc plus à connaître la structure et l'emplacement des fichiers utilisés (seul le DBA^(*) s'en préoccupe) et peuvent se concentrer sur une vision purement logique des données.

(*) Data Base Administrator

1.2. Utilisation

Quelle que soit l'organisation du fichier, la procédure générale sera la même :

1. **Déclaration du nom du fichier** : celui qu'on utilisera au sein du programme pour le référencer.

C'est lors de cette première étape qu'on indiquera l'**organisation physique** du fichier (texte, séquentiel, accès direct...) et le **mode d'accès** choisi (séquentiel, indexé...).

Remarque : Il restera à le mettre en correspondance avec le nom réel du fichier sur le disque (et son emplacement exact).

2. **Ouverture du fichier** qui doit préciser la méthode d'accès choisie. Sont possibles pour les fichiers en organisation séquentielle (Sequential Access Method ou « SAM ») :

En création Le fichier n'existe pas, son « ouverture » provoque l'écriture du *label de début* ("BOF" ou "TOF" pour Bottom, ou Top Of File). Le traitement consistera à remplir le fichier, c'est donc une ouverture en écriture.

En extension Le fichier existe, on souhaite y ajouter des informations *à la suite* de celles qui y sont déjà.

En lecture Le fichier existe, on veut juste le lire.

3. **Opérations de lecture / écriture**

Un ordre de lecture - ou d'écriture - dans un fichier, lit - ou écrit - un élément dans le fichier et déplace un pointeur (ou « curseur ») sur l'élément suivant.

4. **Fermeture du fichier**

Provoque l'écriture du *label de fin de fichier* ("EOF" pour End Of File) s'il avait été ouvert en écriture ; déconnecte le fichier du programme.

Exemples d'utilisation :**→ Façon Cobol****MODULE manip_fichier**

```
(* * * Recopier dans un fichier les employés habitant à Paris * * *)

(* Déclarations *)

FIC  fic_employes          (* Nom interne du fichier *)
    Séquentiel            ACCES séquentiel  (* Organisation et mode d'accès
*)

FIC  fic_parisiens
    Séquentiel            ACCES séquentiel

(* Il faut une variable par fichier : le "masque" *)

VARIABLE employé
    Matricule  caractères  5
    Nom        caractères 20
    Adresse    caractères 80
    Codpost    numérique   5
    Ville      caractères 20
    Salaire    numérique  7,2

VARIABLE parisien
    Matricule  caractères  5
    Nom        caractères 20
    Adresse    caractères 80
    Codpost    numérique   5
    Ville      caractères 20
    Salaire    numérique  7,2

VARIABLE fin_fic_employes (* Et un booléen pour gérer la fin du fichier
*)
```

```
(* Partie exécutable *)

OUVRIR fic_employes      EN LECTURE
OUVRIR fic_parisiens     EN CREATION      (* Ecriture du label de début de
parisiens *)

(* Parcours séquentiel du fichier employes *)
LIRE  fic_employes        (* Lecture du 1er employé *)
  A LA FIN  fin_fic_employes ← vrai

TANT QUE  NON fin_fic_employes
|
|  SI emp.codpost ≥ 75000 ET emp.codpost ≤ 75999
|  |
|  |  parisien ← employé
|  |  ECRIRE fic_parisiens      (* Ajout d'un parisien *)
|  |
|  FIN-SI
|  LIRE  fic_employes          (* Lecture du prochain employé *)
FIN-TANT-QUE

FERMER fic_employes
FERMER fic_parisiens      (* Ecriture du label de fin de
parisiens *)

FIN-MODULE manip_fichier
```

→ Façon micro

MODULE manip_fichier

```
(* * * Recopier dans un fichier les employés habitant à Paris * * *)

(* Déclarations *)
FIC  fic_employes          (* Nom interne du fichier *)
    Séquentiel            ACCES séquentiel  (* Organisation et mode d'accès *)

FIC  fic_parisiens
    Séquentiel            ACCES séquentiel

TYPE ENREG                (* Les 2 fichiers auront la même structure *)
Matricule  caractères      5
Nom        caractères     20
Adresse    caractères     80
Codpost    numérique       5
Ville      caractères     20
Salaire    numérique      7,2

VARIABLE Emp  ENREG        (* Une seule variable suffit *)

(* Partie exécutable *)
OUVRIR fic_employes  EN LECTURE
OUVRIR fic_parisiens EN CREATION      (* Ecriture du label de début de
parisiens *)

(* Parcours séquentiel du fichier employes *)
LIRE  ( fic_employes, emp )           (* Lecture du 1er employé *)
TANT QUE  NON  FIN_DE_FICHER (fic_employes)
|
|   SI emp.codpost ≥ 75000 ET emp.codpost ≤ 75999
|   |   ECRIRE ( fic_parisiens, emp ) (* Ajout d'un parisien *)
|   FIN-SI
|   LIRE  ( fic_employes, emp )       (* Lecture du prochain employé *)
FIN-TANT-QUE

FERMER fic_employes
FERMER fic_parisiens      (* Ecriture du label de fin de
parisiens *)
```

FIN-MODULE manip_fichier

2. LA PERSISTANCE

2.1. Définition

La persistance des données et des états d'un programme font référence aux mécanismes de sauvegarde et de restauration des données.

Ces mécanismes font en sorte qu'un programme puisse se terminer sans que ses données et son état d'exécution ne soient perdus.

Ces informations de reprise peuvent être enregistrées sur disque, éventuellement sur un serveur distant quand il s'agit d'un système de gestion de base de données (SGBD).

2.2. Le modèle des SGBDR

Les systèmes de bases de données relationnelles (SGBDR) apparues dès la fin des années 1970 ont comme particularité une gestion performante des sauvegardes, sans perte d'informations après reprise lors d'un plantage. C'est ce qui a permis à ce modèle de s'implanter dans les entreprises pendant plus de 30 ans. Implantation due aussi au fait qu'il existait un langage standard de conception de base et manipulation des données **nommé SQL** (Strutured Query Language).

2.2.1. Présentation du modèle relationnel

Les bases de données relationnelles permettent de faire évoluer le schéma des données sans modifier les programmes existants.

Elles sont entièrement intégrées aux architectures distribuées et proposent de nombreuses fonctionnalités liées au développement et à l'exploitation d'applications client-serveur.

Elles sont conçues pour aller vite en mise à jour et assurent un mode transactionnel, qui permet un retour en arrière s'il n'y a pas de validation.

L'élément principal d'une base relationnelle est la **table**. La table est un ensemble de colonnes fixes (de type alphanumérique, numérique, ...) contenant des lignes représentant les données stockées par l'entreprise. Chaque ligne est identifiée de manière unique par une **clé primaire**. Il est possible d'accéder à ces données via différents index.

2.2.2. Exemples de tables du modèle relationnel

Exemple d'accès aux données d'un SGBDR (SGBD Relationnel)

- Soit les tables : EMPLOYE (MLE, NOM, PRENOM, SEXE)
- AFFECTATION (MLE, CODPRO, DATDEB)
- DEPT (CODEPT, DESIGN, DG)

MLE [DECIMAL(10 , 0)]	NOM [CHAR(20)]	PRENOM [CHAR(20)]	SEXE [CHAR(1)]
25	LORENT	CATHERINE	F
53	BLASQUEZ ...	NICOLAS	M
54	HERNANDEZ ...	ANTOINE	M
55	MARCIER	JACQUES	M
56	GALLET	BRUNO	M
100	DUCHE	SYLVIE	F
103	SOUPINACIO ...	JESUS	M
218	FONTAINE ...	JEAN-PIERRE	M
273	CHICHE	GLADYS	F
286	ROUVRAIS ...	PHILIPPE	M
303	SALAMI	JUSTIN	M
304	ALPHANDERY ...	ALPHONSE	M
333	PONCHEL	VINCENT	M
633	BALLARD	GASTON	M
652	BIBER	ALBERT	M
672	DUVAL	CHRISTINE	F

MLE [DECIMAL(10 , 0)]	CODPRO [DECIMAL(10 , 0)]	DATDEB [DATE]
25	6	01/04/96
53	120	02/02/97
54	120	02/02/97
55	120	02/02/97
56	6	02/02/97
100	12	01/04/96

CODEPT [DECIMAL(10 , 0)]	DESIGN [CHAR(20)]	DG [CHAR(20)]
2	CONSEIL	DUPONT
3	INDUSTRIE	PEREZ
6	ETHNOS	ROQUE ...
12	FORMATION	MARTIN
120	COMPTABILITE	MIRAN

Exemple de SQL : (* * * Rechercher les employés habitant à Paris * * *)

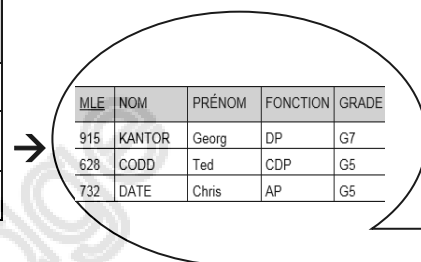
SELECT mle, nom, prenom, sexe FROM ingénieur WHERE ville = 'PARIS'

2.3.1. Principes de la journalisation

Les modifications apportées dans les bases peuvent être conservées dans un journal. Les informations sauvegardées dans le journal sont les valeurs des lignes des tables avant et après la modification. Cela permet de maintenir **l'intégrité des données** et de ne pas perdre d'informations en cas de plantage important avec endommagement des disques et obligation de repartir des sauvegardes journalières.

Exemple : La table Salarié contient 3 lignes, ces informations ont été sauvegardées, le soir, sur un support externe.

SALARIÉ	<u>MLE</u>	NOM	PRÉNOM	FONCTION	GRADE
	915	KANTOR	Georg	DP	G7
	628	CODD	Ted	CDP	G5
	732	DATE	Chris	AP	G5



MLE	NOM	PRÉNOM	FONCTION	GRADE
915	KANTOR	Georg	DP	G7
628	CODD	Ted	CDP	G5
732	DATE	Chris	AP	G5

Dans la journée, l'utilisateur effectue des modifications sur la base.

Il ajoute le salarié 10 Dupont Pierre DP G7 et modifie la fonction du salarié 732 avec la valeur CDP.

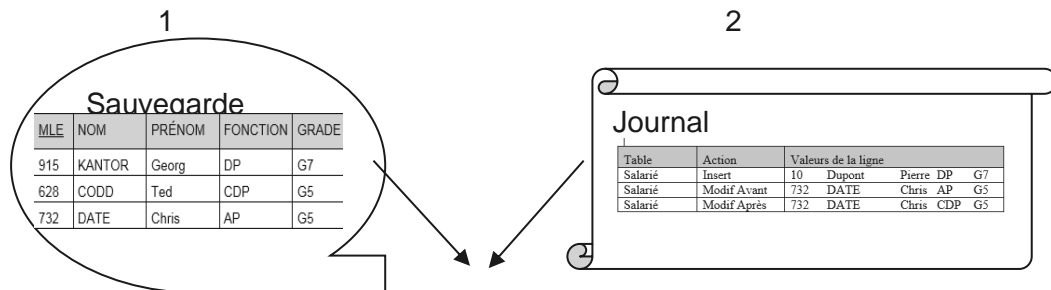
SALARIÉ	<u>MLE</u>	NOM	PRÉNOM	FONCTION	GRADE
	915	KANTOR	Georg	DP	G7
	628	CODD	Ted	CDP	G5
	732	DATE	Chris	AP CDP	G5
	10	Dupont	Pierre	DP	G7

Le journal (qui n'est pas une table) contiendra :

Table	Action	Valeurs de la ligne				
Salarié	Insert	10	Dupont	Pierre	DP	G7
Salarié	Modif Avant	732	DATE	Chris	AP	G5
Salarié	Modif Après	732	DATE	Chris	CDP	G5

Ainsi, nous sommes capables à partir du journal de refaire la modification (lecture avant), et/ou **de défaire la modification** (lecture arrière).

En cas de plantage, on **restaure la table** Salarié à partir du support externe, (elle ne contient que les 3 lignes du départ) et à **partir du journal**, on refait les mises à jour (Insertion et Modification Après). Il n'y a pas de perte de données entre la sauvegarde de la veille et le travail effectué mais perdu suite au plantage !



Récupération de la table des salariés et des mises à jour de la journée

SALARIÉ	MLE	NOM	PRÉNOM	FONCTION	GRADE
	915	KANTOR	Georg	DP	G7
	628	CODD	Ted	CDP	G5
	732	DATE	Chris	CDP	G5
	10	Dupont	Pierre	DP	G7

La journalisation n'est pas obligatoire mais fortement conseillée !

Des points de validation (dits COMMIT) et d'invalidation (ROLLBACK) peuvent être placés dans le journal pour délimiter des transactions (détail des transactions dans le point suivant). (Mises à jour de plusieurs lignes de plusieurs tables). Dans ce cas, on dit que la base est **sous contrôle de validation**.

Pour les Commit et Rollback et pour la gestion des contraintes d'intégrités référentielles, la journalisation est obligatoire.

2.4. Les transactions

2.4.1. Définition

Une transaction est une unité logique de travail qui comprend un ensemble d'ordres SQL indissociables, exécutés par un seul utilisateur. Elle fait passer la base de données **d'un état cohérent à un autre état cohérent**.

Si le moindre problème intervient au cours de l'exécution d'une transaction, toutes les mises à jour qui ont été faites à partir du début de la transaction peuvent être annulées.

2.4.2. Propriétés d'une transaction

On utilise l'acronyme ACID pour désigner les propriétés d'une transaction. Une transaction assure :

- L'**atomicité** des instructions, qui sont considérées comme indissociables les unes des autres. Soit elles arrivent toutes à terme, soit elles sont toutes annulées.
- La **cohérence** de la base de données, qui passe d'un état cohérent à un autre état cohérent.
- L'**isolation** des opérations ayant lieu au cours d'une transaction, par le mécanisme des verrous
- La **durabilité** des mises à jour, une fois qu'une validation est faite à l'issue d'une transaction, il n'est plus possible de demander l'annulation des mises à jour.

2.4.3. Début et fin d'une transaction

Certains SGBD (comme SQL Server), ont des ordres spécifiques pour démarrer une transaction. Sinon, Le début d'une transaction est toujours implicite. Une transaction **débute** :

- Dès la première commande de mise à jour rencontrée ; INSERT, UPDATE ou DELETE,
- Dès la fin de la transaction précédente

Une transaction se **termine** :

- sur un ordre de validation de transaction : COMMIT,
- sur un ordre d'annulation de transaction : ROLLBACK,
- lors de la fin du processus qui a débuté la transaction : fin normale de session utilisateur avec déconnexion, ou fin anormale d'une session utilisateur (sans déconnexion)
- sur utilisation de commandes SQL de DDL (Data Definition Language) ou de DCL (Data Control Language).

Les modifications apportées aux données au cours d'une transaction ne seront visibles dans les transactions concurrentes que lorsque celle-ci sera terminée avec validation des changements.

2775cbe5ce
Global Knowledge

2.5. Algorithme d'accès au monde relationnel

(* * * Rechercher les employés habitant à Paris * * *)

SELECT mle, nom, prenom, sexe FROM ingénieur WHERE ville = 'PARIS'

MODULE accès à la base en Lecture

(* Déclarations du serveur contenant le schéma de la base *)

(* Charger le pilote d'accès à la base (= les définitions du fournisseur du SGBDR permettant de travailler avec le SGBDR *)

URL	caractères	256
NomBase	numérique	10
USER	caractères	10

VARIABLE Resultat (* Une seule variable suffit *)

(* Partie exécutable *)

SE CONNECTER à la base

CREER UNE REQUETE SQL DE LECTURE

EXECUTER LA REQUETE SUR LE SERVEUR

RECUPERER LE RESULTAT DU SERVEUR (Resultat) (* Tous les employés *)

TANT QUE NON FIN_DE_FICHIER (Resultat)

| LIRE (1 ligne du Resultat) (* Lecture du prochain employé *)

TRAITER la ligne

FIN-TANT-QUE

FERMER la connexion

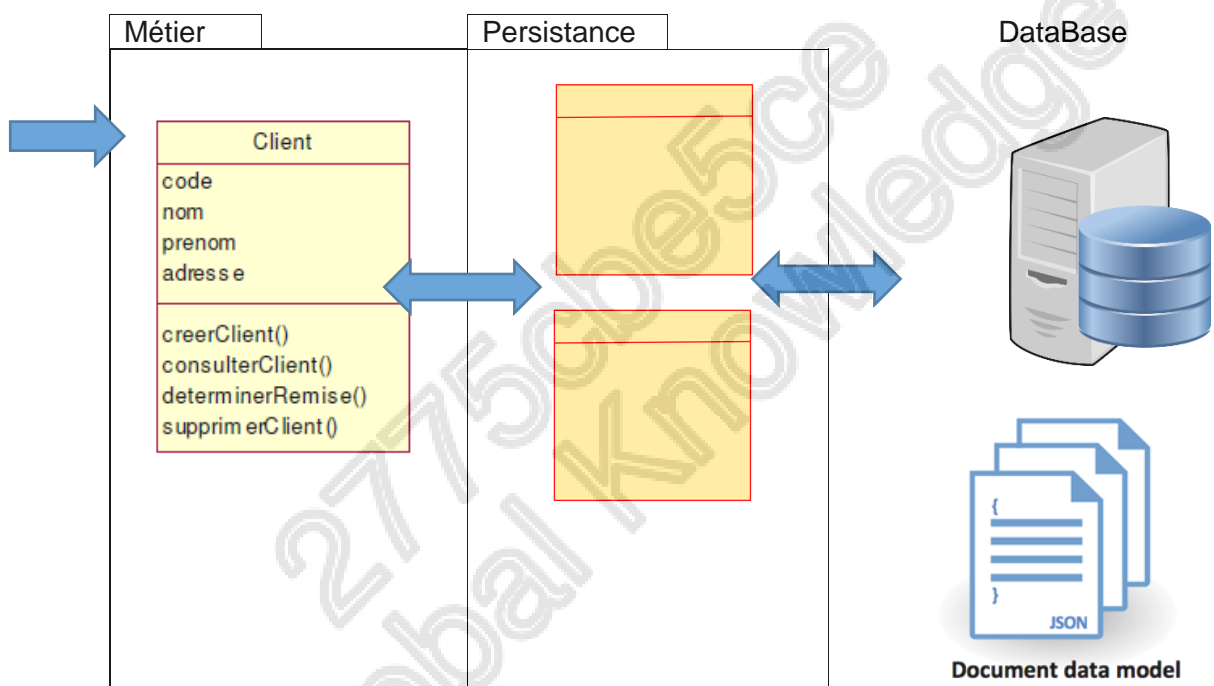
FIN-MODULE accès base

Remarque : Pour un traitement de mise à jour après la connexion, il faudra indiquer le mode transactionnel choisi pour le traitement et définir l'ordre COMMIT pour valider la mise à jour et l'ordre ROLLBACK pour invalider.

2.6. Impact de la persistance pour le concepteur-développeur

Le concepteur-développeur doit être au fait de la gestion de la persistance mise en œuvre sur le projet car l'algorithme à définir peut être simplifié, si la gestion des retours en arrière (suite à un abandon) est gérée par le système lui-même. Il suffira d'indiquer si on abandonne la transaction (Rollback) ou si on la valide (Commit).

En technologie objet, on s'affranchit même du système de gestion de la base de données. On indique uniquement que l'objet « métier » est persistant. Il y a la plupart du temps un système de persistance et de mapping entre les deux mondes : Relationnel/Objet.



CHAPITRE VII MODULARITE OU « COMMENT DIVISER POUR MIEUX REGNER »

1. PRESENTATION

Souvent des tâches analogues sont nécessaires à plusieurs endroits de votre développement : même traitement exécuté, mais avec des données différentes (les « paramètres » ou « arguments »).

Un module est une partie d'un algorithme (ou d'un programme) à laquelle on donne un nom, qui servira à appeler son exécution.

Cette technique a plusieurs intérêts : elle fait correspondre la structure de votre algorithme avec votre analyse, permettant ainsi une lecture « descendante » - ce qui facilite tant la mise au point que l'évolution de votre logiciel. De plus, lorsque les traitements sont semblables, vous n'écrirez qu'une seule fois le module correspondant.

Vous pouvez même ainsi vous constituer une bibliothèque d'outils généralistes (modules d'accès aux fichiers, de contrôle et de mise en forme des dates...), qui accélérera vos développements ultérieurs.

Enfin, la réalisation des modules peut parfaitement être confiée à plusieurs personnes.

Pour définir un module - procédure, fonction ou méthode(*) - il est donc important de **délimiter** logiquement **l'action** encore plus ou moins complexe qu'il représente (le rôle, la fonction qu'on lui attribue) et d'**indiquer les données** nécessairement connues au début, les données obtenues en résultat.

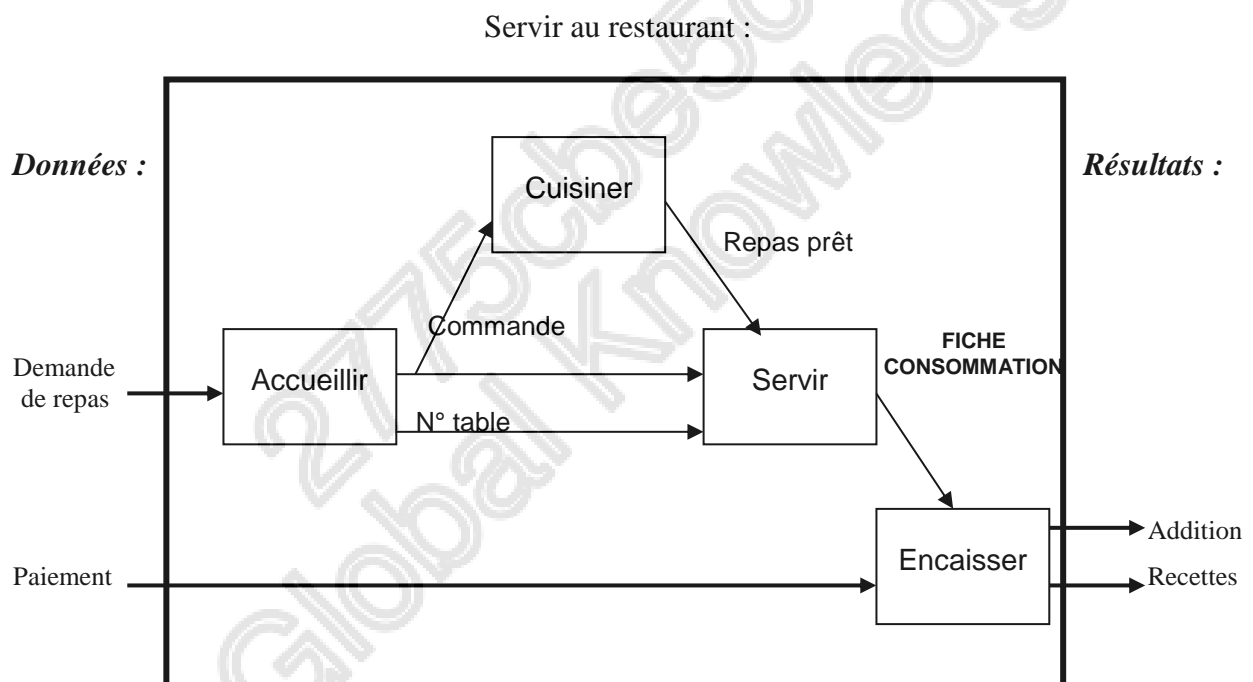
(*) Une méthode est un module relatif à une classe d'objets.

2. FONDEMENTS SYSTEMIQUES

Mais ces modules, qui seront ensuite examinés isolément, ne sont pas indépendants de tout contexte : ils y puisent des données, transmettent des résultats. Bref, à chaque étape de l'analyse, se pose également la question du **partage des informations** :

Des données du problème sont inutiles pour la résolution de certains modules. Inversement, certaines données ne seront utiles que pour un module et encombreraient la vue d'ensemble.

Evidemment, plus le nombre d'informations échangées entre les différents modules est important, plus il est difficile à contrôler : mieux vaut chercher à le minimiser.



3. TYPES DE MODULES

Il existe deux formes traditionnelles de modules :

- Une **fonction** nomme une **valeur** résultant d'un calcul (arithmétique, booléen...). Elle renvoie donc toujours une valeur unique et équivaut à une opération : son appel peut figurer dans une expression.

Exemples :

Variable_entière \leftarrow arrondi(*v_réelle*)

SI *premiers_caractères*(mot, 3) = 'ABC' ...

- Une **procédure** désigne une **action** complexe et peut ne retourner aucune, une ou plusieurs valeurs.

Son appel équivaut à une nouvelle instruction.

Exemples :

Controler_saisie

Saisir(nombre, msg_question, val_mini, val_maxi)

Ainsi, il est toujours possible d'écrire une fonction sous forme de procédure (attention : pas l'inverse). C'est une affaire de style.

Attention : Certains langages ne connaissent pas ces notions de fonctions ou procédures, mais différencient simplement celles de « modules » (internes au programme) et de « sous-programmes » (externes au programme).

4. TRANSMISSION DE PARAMETRES

Il existe différents modes de transmission des arguments :

- Les **paramètres en entrée** : le module n'utilise ces données qu'en tant que renseignements et ne les modifie pas.

Exemples :

la fiche consommation (exemple p.4) est en entrée du module « encaisser » ;
le mot dont on isole les 1^{ers} caractères, et le nombre de caractères à isoler ;
msg_question, val_mini, val_maxi de la procédure « Saisir »

Remarque : Traditionnellement, les paramètres des fonctions sont toujours des paramètres d'entrée : le seul résultat attendu étant émis par la donnée renvoyée par la fonction.

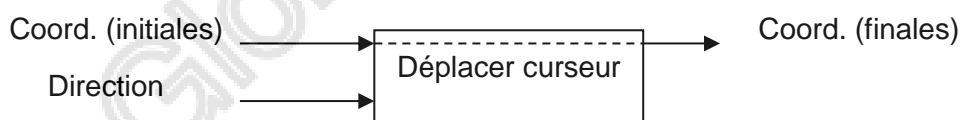
- Les **paramètres en sortie** sont entièrement « calculés » par le module – y compris leur valeur initiale.

Par exemple : la commande émise par le module « accueillir » ; le nombre saisi

Dans ces deux cas, la communication est à sens unique.

- Les **paramètres en entrée/sortie** ou « **entrées modifiées** » : cette fois la communication est bilatérale.

Par exemple, les coordonnées d'un curseur à déplacer :



5. DECLARATIONS / APPELS DE MODULES

Comme d'habitude, il faudra déclarer ce nouvel objet (le module) avant de pouvoir l'appeler.

La déclaration d'une fonction a le même aspect que celle d'une procédure, mais il faut en plus :

- Dans son entête, déclarer le type de la donnée qu'elle représente (la valeur retournée)
- Dans sa partie exécutable, placer au moins une instruction « RENVOIE(valeur) »

Dans tous les cas, au moment de la définition d'un module, les paramètres sont « *formels* » puisqu'on ignore encore sur quelles données le module va réellement opérer ; ces paramètres ne peuvent qu'en être des modèles.

Au moment de l'appel, il s'agit de donner les paramètres « *effectifs* » qui se substitueront aux paramètres formels lors de l'exécution du module.

Les deux listes doivent correspondre :

- même nombre d'arguments,
- mêmes types,
- cités dans le même ordre.

Attention : Certains langages ne mettent pas en œuvre ces techniques de transmission de paramètres (voir l'exemple plus loin).

Exemples :**Déclaration d'une procédure**

```
PROCEDURE P_Saisir(en sortie :   nombre      entier,  
                   en entrée :   msg_question caractères,  
                                val_mini     entier,  
                                val_maxi     entier)  
  
    ECRIRE msg_question  
  
    LIRE nombre  
  
    TANT QUE nombre < val_mini OU nombre > val_maxi  
    |   ECRIRE « Attention, erreur de saisie. Recommencez »  
    |   ECRIRE msg_question  
    |   LIRE nombre  
    FIN-TANT-QUE  
  
FIN-PROCEDURE
```

Appel de la procédure

```
MODULE xxx  
  
    CONSTANCE      MAX  NUMERIQUE 2    = 30  
    VARIABLE      nb   NUMERIQUE 3  
  
    .../...  
  
    P_Saisir (nb, « Donnez le nombre d'articles », 1, MAX )  
  
FIN-MODULE xxx
```

Déclaration d'une fonction

```
FONCTION F_Saisir( msg_question caractères,  
                  val_mini entier,  
                  val_maxi entier)  
  
RENVOIE un entier  
    Nombre : entier  
    ECRIRE msg_question  
  
    LIRE nombre  
  
    TANT QUE nombre < val_mini OU nombre > val_maxi  
    |   ECRIRE "Attention, erreur de saisie. Recommencez"  
    |   ECRIRE msg_question  
    |   LIRE nombre  
    FIN-TANT-QUE  
  
    RENVOIE (nombre)  
  
FIN-FONCTION
```

Appel de la fonction

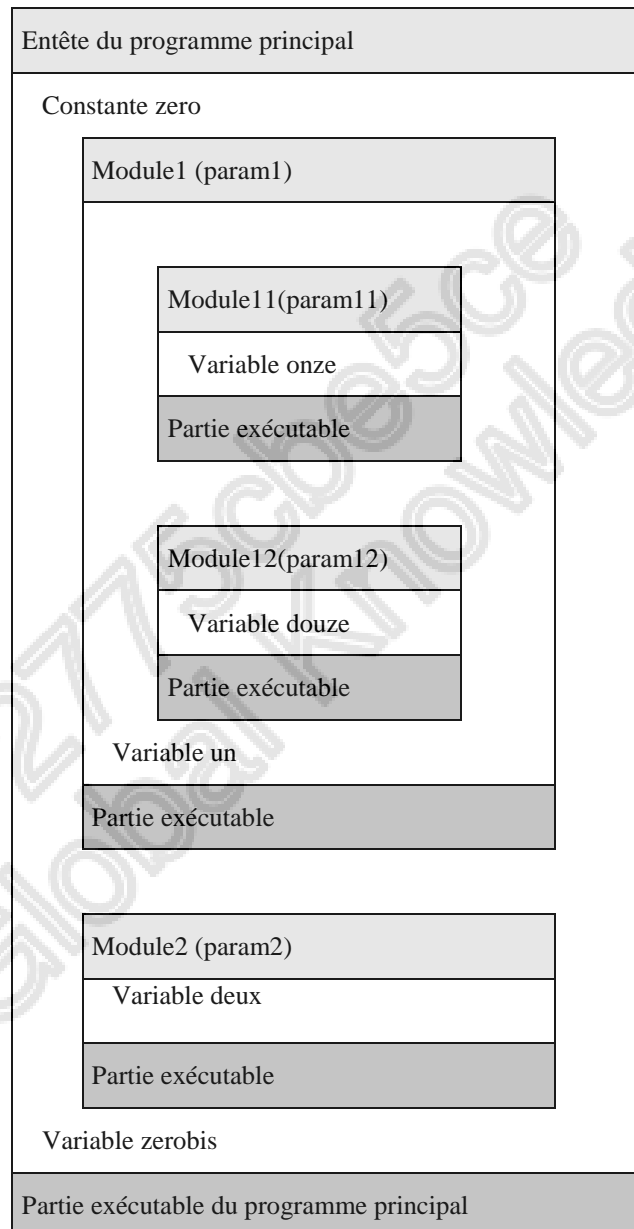
```
MODULE xxx  
  
    CONSTANTE MAX = 30  
    VARIABLE  nb  NUMERIQUE 3  
    .../...  
    nb ← F_Saisir (« Donnez le nombre d'articles », 1, MAX )  
  
FIN-MODULE xxx
```

ou encore :

```
MODULE xxx  
  
    CONSTANTE MAX = 30  
    .../...  
    SI F_Saisir (« Donnez le nombre d'articles », 1, MAX ) =  
    10  
    .../...  
    FIN SI  
  
FIN-MODULE xxx
```

6. IMBRICATION DES MODULES ET PORTEE DES IDENTIFICATEURS

Chaque module pouvant avoir besoin d'objets qui lui sont propres (constantes, types, variables, mais aussi procédures et fonctions), on aboutit à une construction imbriquée qui détermine différents *niveaux de visibilité* des identificateurs^(*) :



(*) Ce paragraphe ne s'applique pas à certains langages pour lesquels tous les modules sont de même niveau et toutes les variables globales.

Les identificateurs déclarés dans un module (paramètres formels compris) sont « **locaux** » : ils ne sont connus qu'à l'intérieur de celui-ci – pas des modules de même niveau, ni de niveau supérieur.

Par contre, un module connaît les identificateurs « **globaux** », déclarés au(x) niveau(x) supérieur(s).

Lorsqu'il y a homonymie d'identificateurs dans deux blocs de niveaux différents, le dernier masque les premiers (la référence concerne l'objet déclaré dans le bloc le plus proche en remontant l'imbrication).

Ainsi, dans le schéma précédent :

- Le module principal connaît zero, module1, module2 et zerobis
- Le module1 connaît zero, module1, param1, module11, module12, un
- Le module11 connaît zero, module1, param1, module11, param11, onze
- Le module12 connaît zero, module1, param1, module12, param12, douze
- Le module2 connaît zero, module2, param2, deux

Puisqu'un module peut accéder aux variables globales, il peut paraître redondant de les définir comme des arguments. Mais comment pourrez-vous utiliser ce module dans un autre contexte ?

De plus, en paramétrant systématiquement les données partagées, vous obtiendrez une véritable description fonctionnelle du module, rien qu'en consultant son entête.

Certains langages n'admettent ni procédure, ni fonction, ni passage de paramètres. Dans ce cas, vous devrez déclarer toutes les données dans le module appelant :

Déclaration d'un module

```
MODULE Saisir

    ECRIRE msg_question
    LIRE nombre
    TANT QUE nombre < MAX OU nombre > MIN
        | ECRIRE "Attention, erreur de saisie. Recommencez"
        | ECRIRE msg_question
        | LIRE nombre
    FIN-TANT-QUE

FIN-MODULE
```

Appel du module

```
MODULE xxx

    CONSTANTE    MAX = 30
    CONSTANTE    MIN = 30
    VARIABLE      msg_question      CARACTERES  30
    VARIABLE      nombre             NUMERIQUE   3
    VARIABLE      nb_articles        NUMERIQUE   3
    VARIABLE      nb_fournisseurs    NUMERIQUE   3

    .../...
    msg_question  ← "Donnez le nombre d'articles"
    APPEL Saisir
    nb_articles   ← nombre
    .../...
    msg_question  ← "Donnez le nombre de fournisseurs"
    APPEL Saisir
    nb_fournisseurs ← nombre

FIN-MODULE xxx
```

Dans tous les cas, mieux vous aurez défini des sous-ensembles *modulaires*, tant en ce qui concerne les *actions* effectuées (découpage en blocs logiques) que les *paramètres*, plus ils seront *réutilisables*.

C'est le **b.a.-ba** des technologies objet.

CHAPITRE VIII

LES TESTS

1. TYPES DE TESTS

Nous ne distinguerons ici que deux **types de tests** :

- Les tests « **unitaires** », ou tests de module, qui ont pour objet de contrôler que le module est réalisé conformément à ses spécifications.
- Les tests « **d'intégration** », l'assemblage des composants du logiciel pouvant lui-même être source d'erreurs si les liens entre les différents modules n'ont pas été préalablement correctement définis.
Ceux-ci sont particulièrement importants lorsque la programmation est évènementielle (intégration, enchaînement de fenêtres).

2. TECHNIQUES DE CONCEPTION

Deux familles de **techniques de conception** de tests existent :

- L'approche « **boîte noire** » permettant de réaliser des tests fonctionnels : on s'intéresse uniquement au comportement externe du module (description de ses entrées / sorties).
- L'approche « **boîte blanche** » pour des tests structurels : cette fois, on veut contrôler non plus ce que fait le module mais la manière dont il le fait, ce qui implique de connaître sa logique interne (instructions, débranchements, conditions...).

3. ETAPES DE CONCEPTION DES TESTS

La phase de tests peut être décomposée en **plusieurs étapes** :

1. Préparation des tests :

- Définir les situations à tester. En théorie, elles doivent représenter l'exhaustivité des cas possibles pour le programme.
- Déterminer les valeurs des données d'entrée permettant de contrôler ces situations.
- Décrire explicitement et entièrement les résultats attendus à l'issue de l'exécution.

2. Exécution des tests et comparaison des résultats obtenus avec ceux attendus

3. Correction :

- Localisation et définition de la nature exacte des défauts.
- Correction des défauts. Et dans ce cas, retour au ② : refaire tous les tests pour s'assurer qu'une « correction » n'a pas tout cassé ! « **non régression** » jusqu'à obtenir le résultat attendu.

Il faut évidemment recommencer ces 2 dernières étapes jusqu'à ce qu'il n'y ait plus d'erreur !

Dans les **technologies objet**, il est défini de mettre en œuvre **des cas d'utilisation ou Use Case**. Les cas d'utilisation sont des descriptions textuelles, utilisées pour détecter et consigner les besoins. Ils montrent les interactions fonctionnelles entre les acteurs et le système à l'étude. Lors de la production de UC, il est nécessaire d'écrire des scénarii.

Le scénario est un ensemble d'actions qui permet d'atteindre l'objectif du cas d'utilisation.

Il existe plusieurs scénarii pour un cas d'utilisation et on identifie :

- Le scénario normal ou principal : séquence d'actions la plus fréquente ou le cas où tout se passe bien.
- Les scénarii alternatifs : séquences d'actions qui permettent d'aller du début à la fin normale du cas d'utilisation et différentes du cas normal.
- Les scénarii d'exceptions : séquences d'actions qui ne permettent pas de terminer normalement le cas d'utilisation.

Dans sa description au niveau le plus fin et en faisant intervenir les instances des objets impliqués, les scénarii obtenus à ce niveau sont les scénarii utilisés pour la **recette utilisateur**. **Les utilisateurs détiennent la connaissance métier et peuvent donc fournir des jeux d'essais** qui seront utilisés le concepteur développeur. Les cas d'utilisation sont intégrés aux outils de développements. Et sont réutilisés pour les tests de non régression.

Exemple de scénarii et de test d'un cas d'utilisation pour un client d'une banque souhaitant retirer de l'argent au DAB (Distributeur Automatique de Billet).



Cas d'utilisation	Retrait d'argent à un DAB
Début du cas d'utilisation	Le client insère sa carte de crédit dans le DAB
Fin du cas d'utilisation	Le client obtient l'argent demandé
Acteurs	Le client, le système interbancaire
Scénarii	
Scénario normal	« tout est normal »
	1 Le client insère sa carte
	2 Le client tape son code confidentiel et valide
	3 Le système accepte
	4 Le client tape son montant et valide
	5 Le système autorise la somme demandée
	6 Le client retire sa carte, son argent et son reçu
Scénario alternatif 1	« le client se trompe une fois dans la saisie de son code »
	1.. 2 idem
	A 1 Le système refuse
	A 2 Le client tape à nouveau son code et valide.
	3.. 6 Idem.
Scénario alternatif 2	« le crédit disponible autorisé est inférieur à la somme demandée »
	1.. 4 idem
	B 1 Le système avertit le client qu'il ne peut pas lui donner la somme désirée
	B 2 Le client tape un nouveau montant et valide
	5.. 6 idem
.../...	
Scénario d'exception 1	« le client se trompe 3 fois dans la saisie de son code »
	.../...

Test fourni par l'utilisateur :

- Le client Dupont de la banque insert sa carte n°14571254.
- Il tape son code 1025.
- Le système valide.
- Il tape le montant demandé : 100 Euros.

Exemple de fiche de tests, (non Use Case) Chemin GCF : Suivre les Commissions/ Résultat des PMI en Commission

N° de cas	Description du cas	Type du cas	Résultat attendu	Données utilisées	Auteur	Résultat constaté
1	Positionnement du curseur sur la ligne qui permet l'ajout, dans la liste, des PMI passées en commission. F06 et sélection d'une PMI F12 pour revenir sur l'écran JI41	P	Arrivée sur la map MJ141A affichage d la PMI sélectionnée sur la ligne qui permet l'ajout, dans la liste, de la PMI avec le résultat de la commission.	aucune	IM	OK 03/02/016
2	Positionnement du curseur sur une des lignes de la liste proposée, se placer au niveau du résultat de la commission F7 et sélection d'un résultat F12 pour revenir sur l'écran JI41	P	Arrivée sur la map MJ141A affichage d la réponse sélectionnée dans la colonne résultat de la ligne en cours.	aucune	im	ok 03/02/2016
3	Saisie du code contractant dans la zone 'CN DOSSIER'	P	Positionnement de la liste des pmj passées en commission sur la première occurrence à partir du code contractant saisi, affichage jusqu'à la fin de la liste	aucune	im	ok 02/03/92016
4	MAJ d'un résultat de PMI à partir de la zone basse de l'écran 'CREATION ET MISE A JOUR'	P	Affichage du message 'MAJ BIEN EFFECTUEE'. L'abend 4038 ne se produit plus	PROG : A5 CODE_CM : STEPH NO_CM : 01	SB	ok 10/03/92015
5	MAJ d'un résultat d'IDEI en liste avec une valeur autre que IA, IP, RF, AJ, AN.	NP	Affichage du message 'VALEUR non autorisée pour une IDEI'	IDEI 16000/CH-R1/LE-R0 COMMISSION CCB 24	SL	OK 14/10/2015
6	MAJ d'un résultat d'IDEI en zone basse de l'écran avec une valeur autre que IA, IP, RF, AJ, AN.	NP	Affichage du message 'VALEUR non autorisée pour une IDEI'	IDEI 16000/CH-R1/LE-R0 COMMISSION CCB 24	SL	OK 14/10/2015
7	Saisie d'une réponse AC sur une PMI de classe 1A	NR	Affichage du message 'MAJ bien effectuée' L'avancement de la PMI reste à 9. Le bilan du dossier n'est pas renseigné.	PMI 33333/CC-R1/CC-R0 COMMISSION : CMECC N° CM : 01	YD	OK 25/10/15
8	Saisie d'une réponse NO sur une PMI de classe 1A	NR	Affichage du message 'MAJ bien effectuée' L'avancement de la PMI passe à 4. Le bilan du dossier est renseigné (par NOCC01), ainsi que la date_da et rev-da.	PMI 33333/CC-R1/HH-R0 COMMISSION : CMECC N° CM : 01	YD	OK 25/10/15

ANNEXES

1. REPRESENTATION INTERNE DES DONNEES

1.1. ORGANISATION DE LA MEMOIRE

La mémoire est constituée de Binary digiTs (ou bits) qui ne peuvent avoir que les valeurs :
0 ou 1.

La lecture d'une zone mémoire en base 2 devient donc rapidement fastidieuse. Afin de faciliter l'exploration de cette mémoire, on utilise des techniques de vidage utilisant soit la base 8, c'est-à-dire l'octal, soit la base 16, c'est-à-dire l'hexadécimal.

1.1.1. OCTAL

En octal, les bits sont regroupés par paquets de 3 ; à chaque combinaison de 3 bits, on associe un chiffre en base 8 (donc de 0 à 7).

Base deux	Base huit
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

9 bits = 1 byte

4 bytes = 1 mot = 36 bits

Ainsi 100 111 000 101 010 110 en binaire
peut être représenté par 470 526 en octal.

Le vidage en Octal est utilisé sur les DPS 8 de Bull – Des machines qui n'existent plus en 2000.

1.1.2. HEXADECIMAL

Les bits sont regroupés par paquets de 4 ; à chaque combinaison de 4 bits, on associe un chiffre en base 16 (de 0 à 9 et de A à F).

Base deux	Base seize
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

8 bits = 1 byte (ou octet)
4 octets = 1 mot = 32 bits

Ainsi 0001 0100 0100 1111 1101 1101 en binaire
 équivaut à : 1 4 4 F D D en hexadécimal
 L'hexadécimal est la technique la plus utilisée.

1.2. LE CODAGE

Nous avons vu dans le paragraphe précédent les différents types de vidage de la mémoire. Nous allons maintenant appliquer à ces différents types de représentation des critères d'interprétation.

On va **coder** la mémoire.

Les codages les plus communément utilisés pour représenter les données sont :

- l'EBCDIC (Extended Binary Coded Decimal Interchange Code).
- l'ASCII (American Standard Code for Information Interchange).

Exemple 1 : Bull DPS 8
Codage ASCII en octal.

Valeur décimale	Vidage octal	Codage ASCII	Binaire
32	040	Space	000 100 000
40	050	(000 101 000
41	051)	000 101 001
42	052	*	000 101 010
43	053	+	000 101 011
45	055	-	000 101 101
47	057	/	000 101 111
48	060	0	000 110 000
49	061	1	000 110 001
50	062	2	000 110 010
51	063	3	000 110 011
52	064	4	000 110 100
53	065	5	000 110 101
54	066	6	000 110 110
55	067	7	000 110 111
56	070	8	000 111 000
57	071	9	000 111 001
60	074	<	000 111 100
61	075	=	000 111 101
62	076	>	000 111 110
65	101	A	001 000 001
66	102	B	001 000 010
67	103	C	001 000 011
68	104	D	001 000 100
69	105	E	001 000 101
97	141	a	001 100 001
98	142	b	001 100 010
99	143	c	001 100 011
100	144	d	001 100 100
101	145	e	001 100 101
123	174	{	001 111 100

Remarque : dans ce cas
blanc < chiffres < majuscules < minuscules

Exemple 2 : IBM Grands systèmes
Codage EBCDIC hexadécimal

Valeur décimale	Vidage hexadécimal	Codage EBCDIC	Binaire
64	40	Space	0100 0000
76	4C	<	0100 1100
78	4E	+	0100 1110
92	5C	*	0101 1100
96	60	-	0110 0000
97	61	/	0110 0001
110	6E	>	0110 1110
129	81	a	1000 0001
130	82	b	1000 0010
131	83	c	1000 0011
132	84	d	1000 0100
133	85	e	1000 0101
193	C1	A	1100 0001
194	C2	B	1100 0010
195	C3	C	1100 0011
201	C9	I	1100 1001
209	D1	J	1101 0001
211	D3	L	1101 0011
217	D9	R	1101 1001
226	E2	S	1110 0010
227	E3	T	1110 0011
240	F0	0	1111 0000
241	F1	1	1111 0001
242	F2	2	1111 0010
243	F3	3	1111 0011
244	F4	4	1111 0100
245	F5	5	1111 0101
246	F6	6	1111 0110
247	F7	7	1111 0111
248	F8	8	1111 1000
249	F9	9	1111 1001

Remarque : Ici
« espace » < minuscules < majuscules < chiffres

Sur les micro-ordinateurs, on utilise le vidage hexadécimal et le codage ASCII.

1.3. REPRESENTATION INTERNE : DONNEES ALPHANUMERIQUES

Pour les données alphanumériques, chaque caractère est représenté par un byte.

Exemple : ZONA-AN-7

'BONJOUR' → ZONA

Représentations internes

1°) Octal ASCII (Bull DPS8)

001 000 010	001 001 111	001 001 110	001 001 010	001 001 111	001 010 101	001 010 010
-------------	-------------	-------------	-------------	-------------	-------------	-------------

102	117	116	112	117	125	122
-----	-----	-----	-----	-----	-----	-----

BONJOUR

2°) Hexadécimal EBCDIC (IBM Minis et grands systèmes)

1100 0010	1101 0110	1101 0101	1101 0001	1101 0110	1110 0100	1101 1001
-----------	-----------	-----------	-----------	-----------	-----------	-----------

C2	D6	D5	D1	D6	E4	D9
----	----	----	----	----	----	----

BONJOUR

3°) Hexadécimal ASCII (Micro-ordinateurs)

0100 0010	0100 1111	0100 1110	0100 1010	0100 1111	0101 0101	0101 0010
-----------	-----------	-----------	-----------	-----------	-----------	-----------

42	4F	4E	4A	4F	55	52
----	----	----	----	----	----	----

BONJOUR

1.4. REPRESENTATION INTERNE : DONNEES NUMERIQUES

1.1.3. LE MODE ETENDU

Le premier mode de représentation des données numériques est identique à celui utilisé pour les données alphanumériques.

Chaque chiffre sera représenté sur un octet.

Exemple 1 : Hexadécimal / EBCDIC

NB1-N-5V0

12345 → NB1

Ecrivez en hexadécimal le contenu de la zone mémoire NB1.

--	--	--	--	--

NB1

Exemple 2 : Octal / ASCII

NB2-N-5V0

12345 → NB2

Ecrivez en octal le contenu de la zone mémoire NB2.

--	--	--	--	--

NB2

1.1.4. LE MODE BINAIRE

Le nombre est représenté en mémoire par sa valeur binaire.

5 → 0101
(10) (2)

Un nombre négatif s'obtient en complémentant tous les bits à 1 puis en ajoutant 1 au nombre ainsi obtenu.

Le bit de gauche vaut 0 si le nombre est positif.

Le bit de gauche vaut 1 si le nombre est négatif.

A-N-4V0 (BINAIRE)

Codage de +5 en base 2 : 0000 0000 0000 0101

Complémentation : 1111 1111 1111 1010

Ajout de 1 : 1111 1111 1111 1011

Soit

FF

FB

Quel est le plus grand nombre positif que l'on peut écrire en mode binaire sur 2 octets ?

2 octets = 16 bits soit 1 bit de signe et 15 bits significatifs.

$2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 + 2^8 + 2^9 + 2^{10} + 2^{11} + 2^{12} + 2^{13} + 2^{14} = \mathbf{32767}$
soit en base deux : 0111 1111 1111 1111 ($2^{15} - 1$)

Le plus petit nombre négatif est

1000 0000 0000 0000 en binaire
donc -32 768 (-2^{15})

Remarque importante :

Quel que soit le mode de représentation choisi, le calculateur effectue les calculs en binaire. Il convertit tous les autres usages si cela est nécessaire. Donc l'utilisation de ce mode de représentation interne des données permet de réaliser les calculs plus rapidement. Cependant la lecture d'un vidage de mémoire (DUMP) en binaire n'est pas très claire.

Repère

1 Kilo-octet (Ko)	= 2^{10} caractères	= 1 024 caractères	= ¼ de page
1 Mega-octet (Mo)	= 2^{20} caractères	= 1 048 576 caractères	= 1 livre de 260 pages
1 Giga-octet (Go)	= 2^{30} caractères	= 1 073 741 824 caractères	= 1000 livres
1 Tera-octet (To)	= 2^{40} caractères	= 1 099 511 627 776 caractères	= 1 million de livres
1 Peta-octet (Po)	= 2^{50} caractères	= 1 125 899 906 842 624 caractères	= 1 milliard de livres

Ensuite on a, 1 zettaoctet (Zo) 2^{70} ou 10^{21} caractères, puis le yottaoctet (Yo) 2^{80} ou 10^{24} caractères.

2. LES FICHIERS SEQUENTIELS

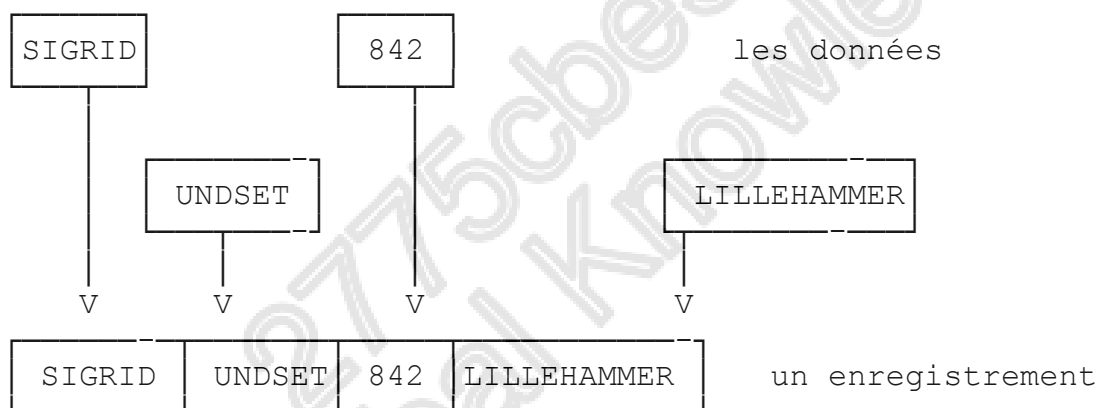
2.1. GENERALITES

Nous avons déjà écrit quelques programmes simples qui traitaient les données en provenance du clavier du terminal.

La donnée en entrée, est immédiatement traitée et la donnée en sortie est affichée à l'écran du même terminal. Quelques rares programmes, en particulier ceux écrits en BASIC, peuvent travailler de cette manière interactive si les données en entrée ne sont pas trop volumineuses et si les données en sortie ne doivent pas être conservées.

Dans l'immense majorité des cas, les applications utilisent en entrée un important volume de données (INPUT) et obtiennent en sortie des traces permanentes (OUTPUT).

Pour ce type d'application, les données sont regroupées en **enregistrements** (RECORD) et l'ensemble de ces enregistrements constitue un **fichier** (FILE).



Remarques :

- Un fichier est donc une série d'enregistrements ;
- Les enregistrements d'un même fichier peuvent être de même longueur (format fixe) ou de longueur variable (format variable) ;
- Un fichier peut aussi bien être constitué de données que de programmes.

2.2. ORGANISATION DES DONNEES

2.2.1. FORMAT DES ENREGISTREMENTS

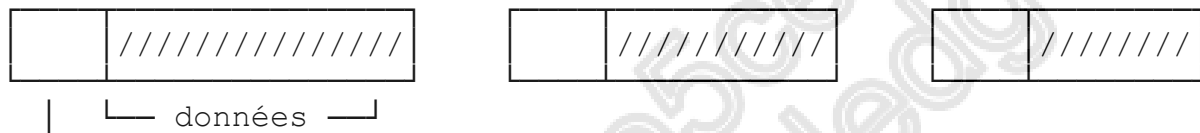
Les enregistrements d'un fichier peuvent avoir différents formats. On distingue principalement trois types de format :

Format fixe :

Tous les enregistrements ont le même nombre d'octets. Les enregistrements sont donc de taille fixe ou de format fixe.

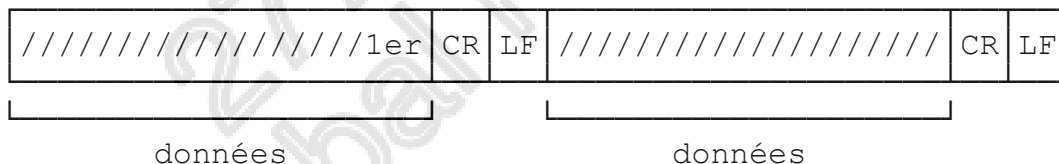
Format variable :

Les enregistrements n'ont pas tous la même longueur, la taille de chacun d'eux est donnée dans un mot de contrôle (4 octets) placé en tête de chaque enregistrement.



4 octets de contrôle

Il existe sous MS-DOS, un format variable particulier : le LINE SEQUENTIAL. Les enregistrements y sont délimités par la séquence de caractères CARIAGE-RETURN et LINE-FEED :



Format indéfini :

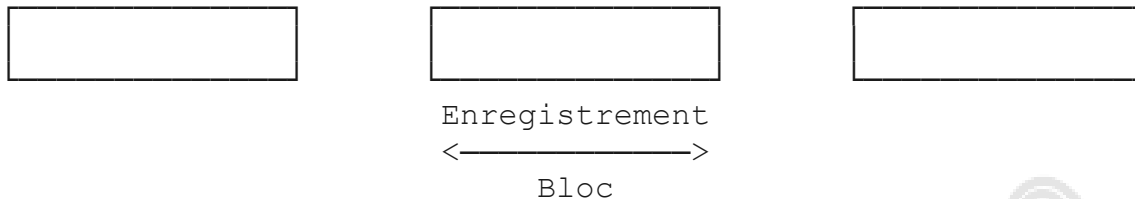
Les enregistrements n'ont pas tous la même longueur mais la taille n'est pas indiquée (peu utilisé pour des fichiers de données).

2.2.2. BLOCAGE DES ENREGISTREMENTS

Un bloc (ou enregistrement physique) est un regroupement de plusieurs enregistrements d'un même fichier (aussi appelés enregistrements logiques). Le bloc correspond à l'unité élémentaire d'entrée/sortie.

Enregistrements non bloqués

On dit que les enregistrements sont non bloqués lorsque le bloc ne contient qu'un seul enregistrement (1 enregistrement physique = 1 enregistrement logique).

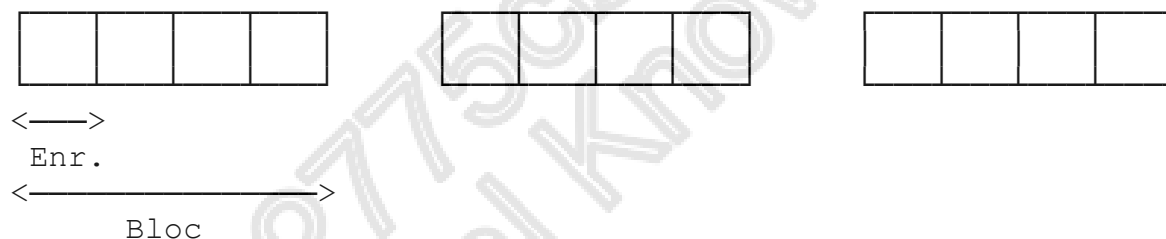


Chaque ordre de lecture ou d'écriture émis par un programme donne lieu à une entrée/sortie physique entre le support de stockage (disque dur, bande magnétique, ...) et la zone de mémoire centrale utilisée par le programme.

Les opérations d'entrée/sortie étant les plus coûteuses en temps d'exécution, le non-blocage des enregistrements est une solution à bannir pour le traitement de fichiers de données.

Enregistrements bloqués

Les enregistrements sont bloqués lorsqu'un bloc contient n enregistrements logiques ($n > 1$). n est appelé le facteur de blocage et est choisi en fonction de contraintes système (type de disque utilisé).

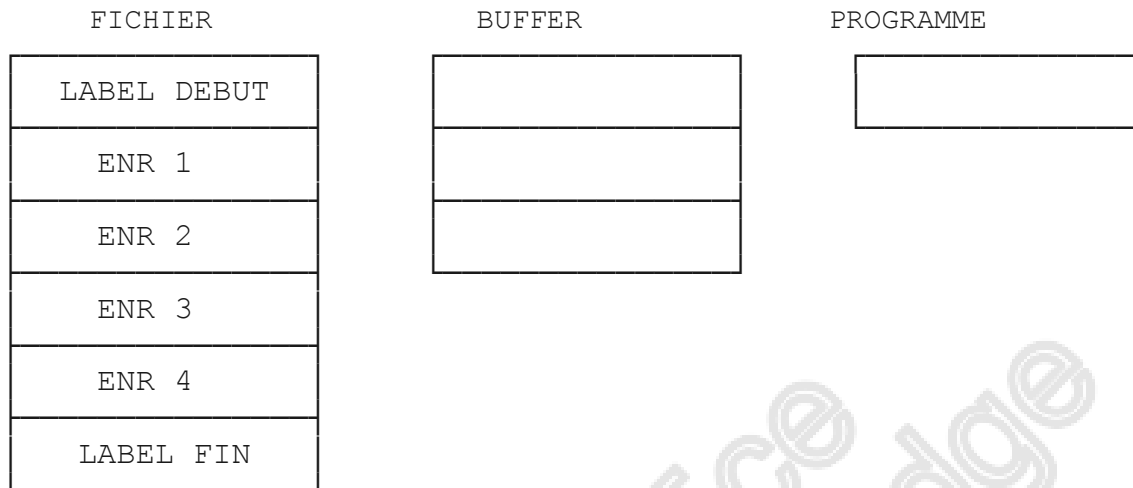


Le facteur de blocage dans le schéma représenté ci-dessus est quatre. Dans ce cas, le premier ordre de lecture émis par un programme provoque le transfert des quatre premiers enregistrements logiques du fichier dans une zone de mémoire centrale appelée buffer ou mémoire tampon. Le premier de ces quatre enregistrements est alors accessible par le programme. Le deuxième enregistrement est déjà en mémoire centrale (dans le buffer). Ainsi quatre enregistrements peuvent être lus par le programme en n'effectuant qu'un seul transfert entre le disque dur et la mémoire centrale ce qui équivaut à un gain de temps considérable le transfert de n enregistrements prenant sensiblement le même temps que le transfert d'un seul enregistrement.

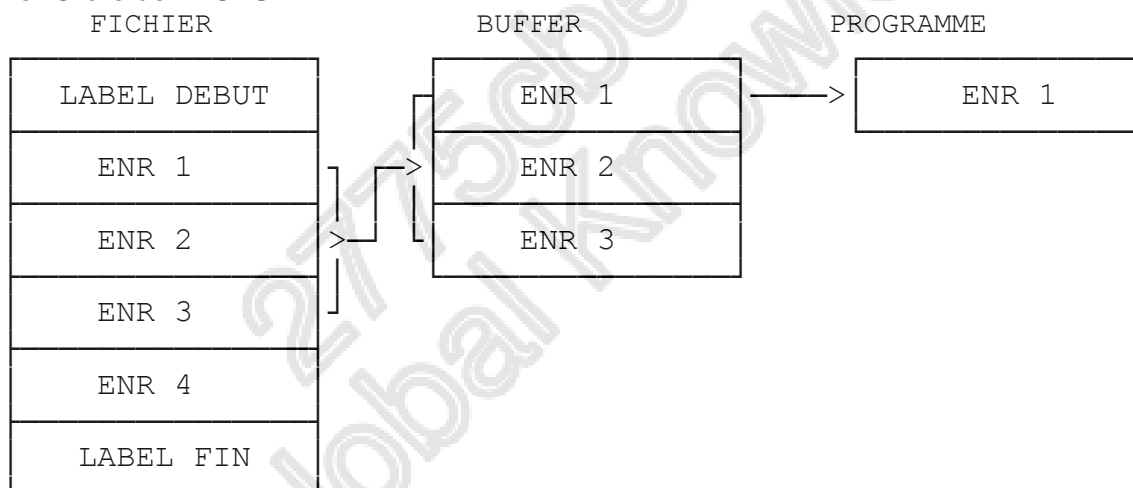
Exemple :

Lecture d'un fichier dont les enregistrements sont bloqués par 3

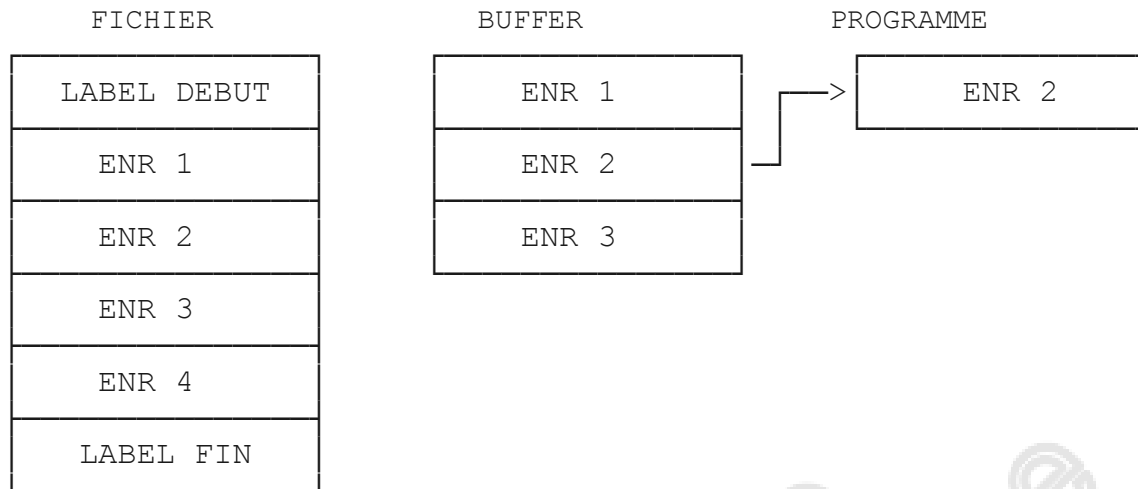
Ouverture du fichier :



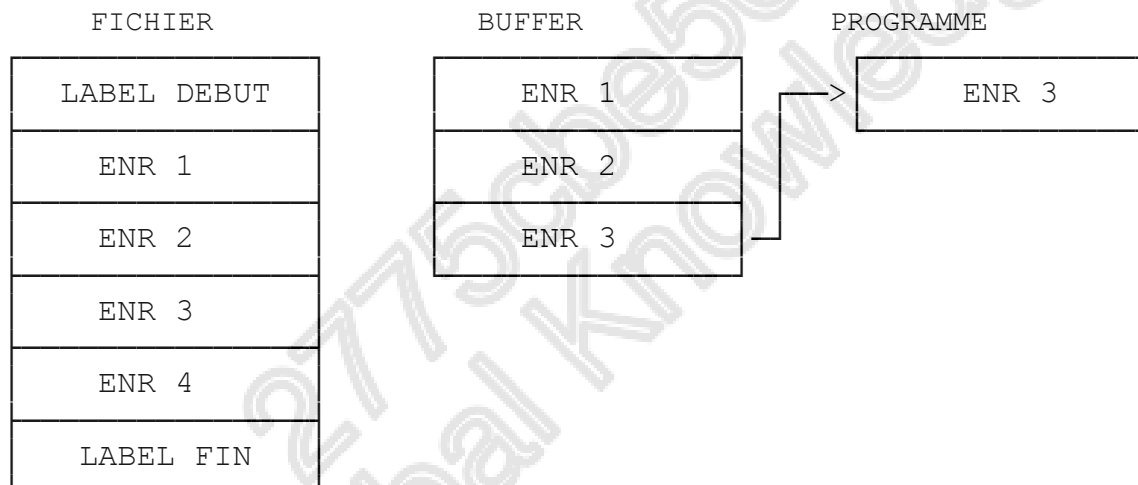
1er Ordre de LECTURE



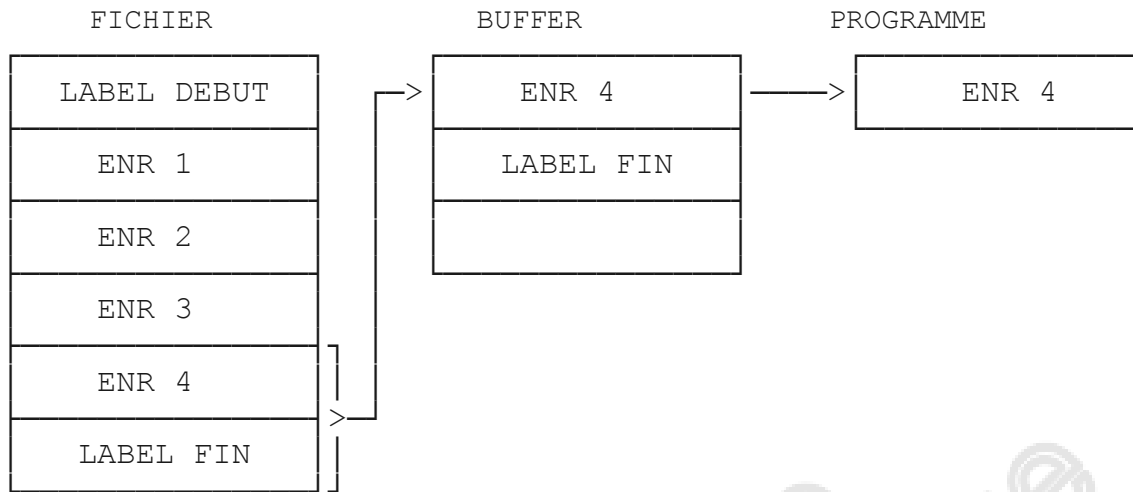
2ème Ordre de LECTURE :



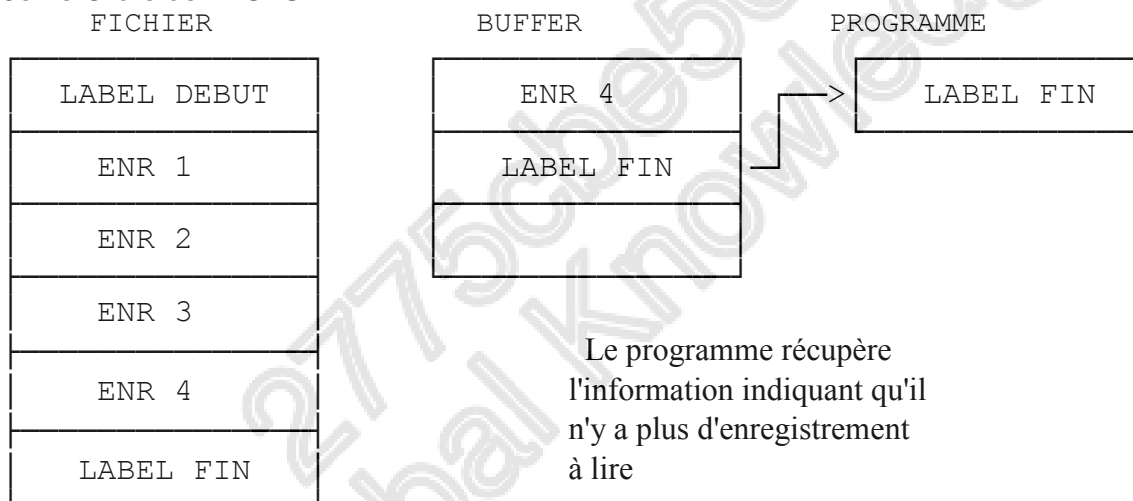
3ème Ordre de LECTURE :



4ème Ordre de LECTURE :



5ème Ordre de LECTURE :



Ordre de FERMETURE du fichier :

FICHER

LABEL DEBUT
ENR 1
ENR 2
ENR 3
ENR 4
LABEL FIN

BUFFER

PROGRAMME

--

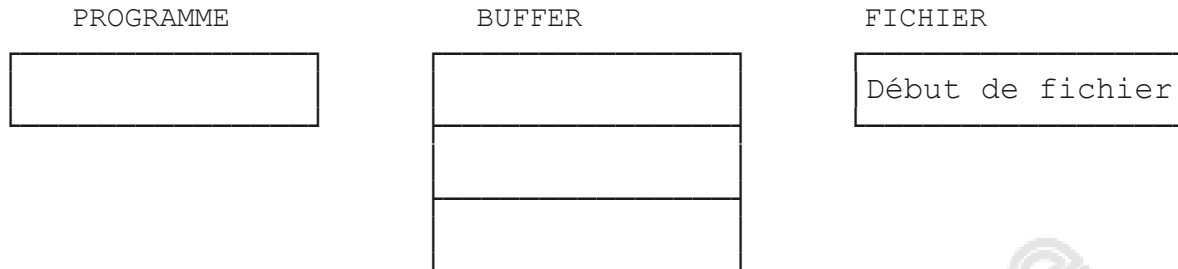
2775cbe5ce
Global Knowledge

Le mécanisme d'écriture d'un fichier dont les enregistrements sont bloqués est similaire au mécanisme de lecture.

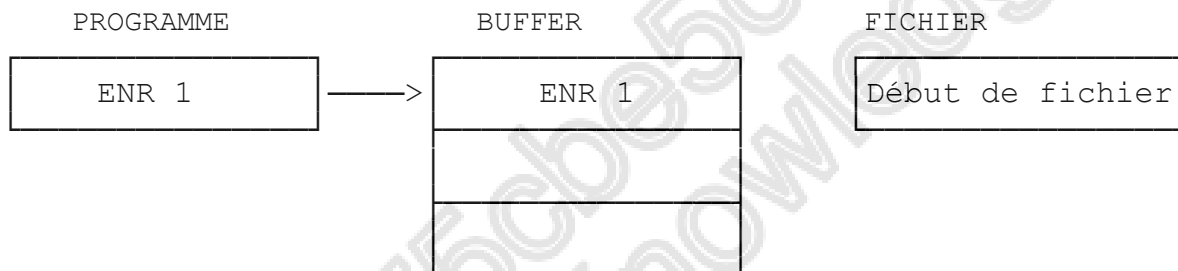
Exemple :

Un bloc contient trois enregistrements logiques; il y a quatre enregistrements à écrire.

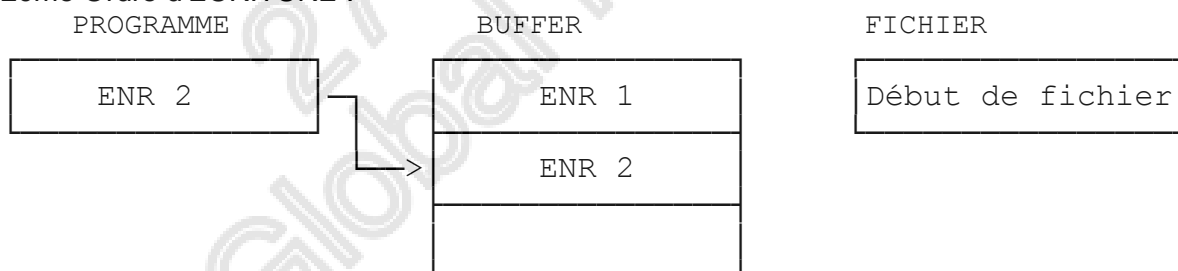
OUVERTURE du fichier :



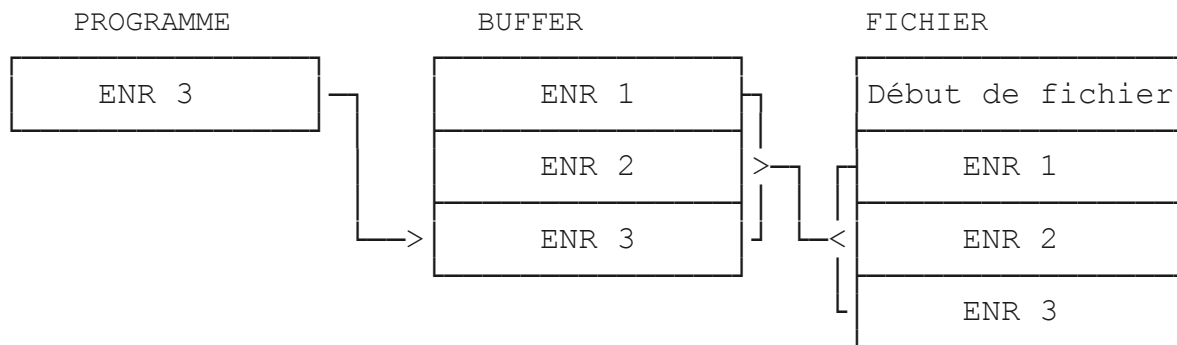
1er Ordre d'ECRITURE :



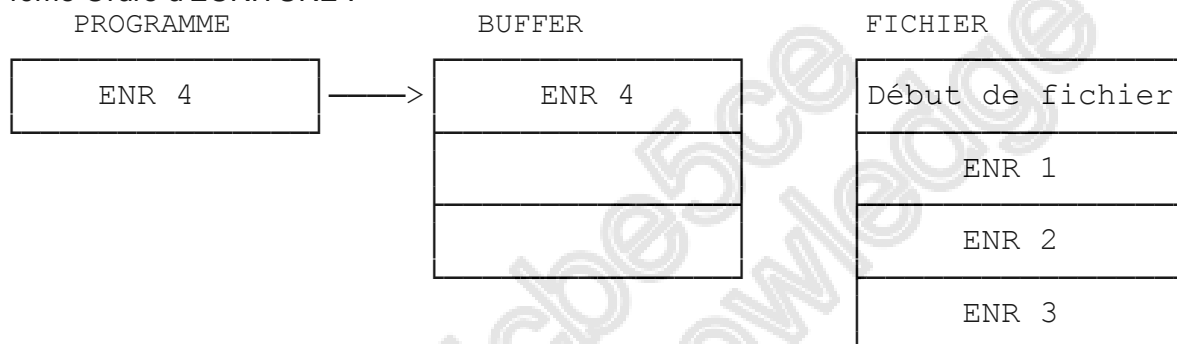
2ème Ordre d'ECRITURE :



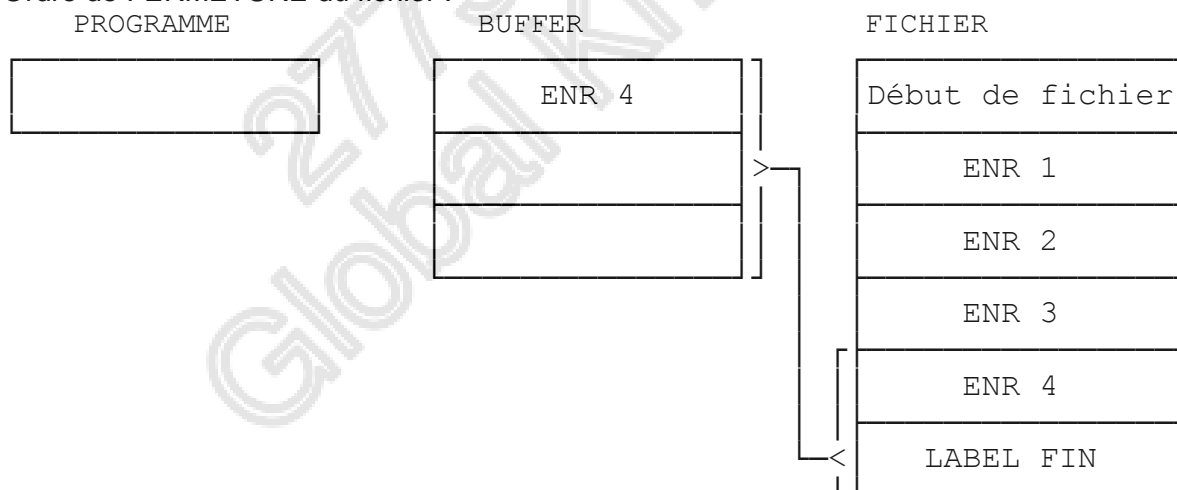
3ème Ordre d'ECRITURE :



4ème Ordre d'ECRITURE :



Ordre de FERMETURE du fichier :



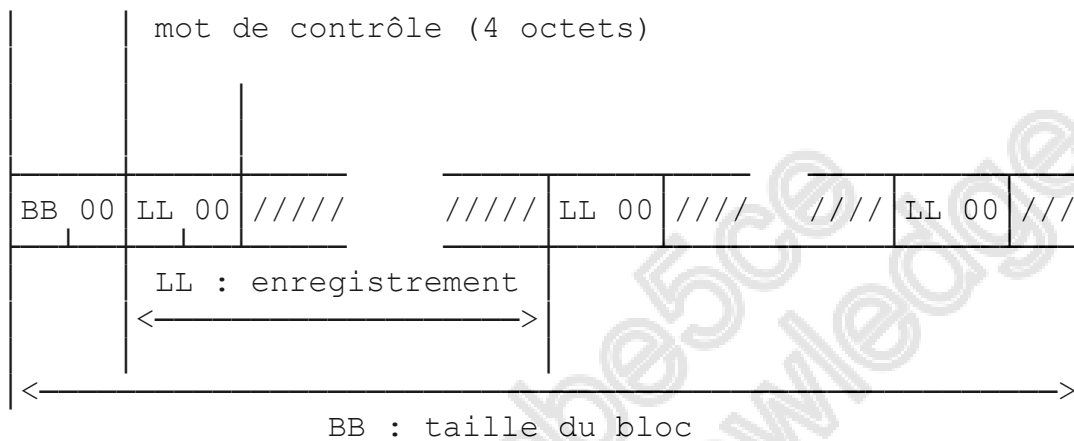
Taille d'un bloc

Un bloc est défini par sa taille.

Lorsque les enregistrements sont de format fixe, la taille d'un bloc est égale à la taille d'un enregistrement multipliée par le facteur de blocage.

Pour des enregistrements de format variable, la taille d'un bloc est égale à la longueur maximale des enregistrements (sans oublier les quatre octets de contrôle en tête de chaque enregistrement) multiplié par le facteur de blocage + quatre octets de contrôle du bloc. Un bloc présente alors la structure suivante :

mot de contrôle (4 octets)



BB : Taille du bloc y incluant les quatre octets de contrôle (en binaire).

00 : Zéros binaires.

Cas particulier : Enregistrements spannés

Un enregistrement spanné est un enregistrement qui peut chevaucher plusieurs blocs.




2.3. LES SUPPORTS DES DONNEES

2.3.1. BANDES MAGNETIQUES

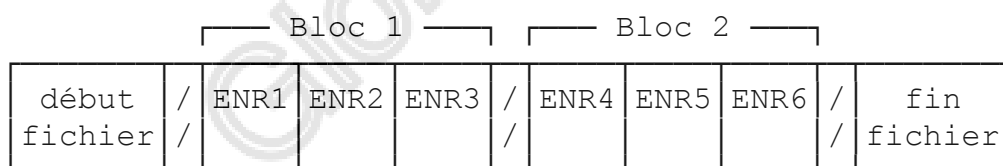
Une bande magnétique est divisée dans le sens de la longueur en pistes.

	0		piste 1
	1		piste 2
	0		piste 3
	1		piste 4
	1		piste 5
	0		piste 6
	1		piste 7
	0		piste 8
	0		piste de parité


 > 1 octet : 8 bits de données
 + 1 bit de parité

Les informations sont enregistrées sur la bande avec une certaine **densité**. La densité est indiquée par le nombre d'octets par pouce (byte per inch : bpi).

Un fichier est délimité sur une bande par un label de début de fichier et un label de fin de fichier.



2.3.2. DISQUES MAGNETIQUES

Les disques sont composés de plateaux d'aluminium recouverts d'une pellicule magnétique. L'ensemble de ces disques superposés sous forme de pile, tournent ensemble sur un axe commun. Chaque disque est utilisé sur les 2 faces.

Sur chaque face les données sont disposées en **pistes** concentriques.

Positionnée à quelques microns de la piste se trouve une tête de lecture-écriture par face ; l'ensemble de ces têtes de lecture/écriture sont fixées à un bras rigide qui se déplace.

A un instant donné, toutes les têtes de lecture/écriture se trouvent dans une même verticale, ainsi l'ensemble des pistes accessibles pour une position donnée du bras de lecture forme un **cylindre**.

Il existe aussi des unités de disques à tête fixe : il y a alors autant de têtes de lecture/écriture que de pistes.

Ainsi chaque bloc peut être repéré par :

- numéro de cylindre CC
- numéro de piste HH
- numéro de bloc R

CCHHR = adresse physique

Inventé en 1956, le disque dur a fait l'objet d'évolutions de capacité et de performances considérables, tout en voyant son coût diminuer, ce qui a contribué à la généralisation de son utilisation.

2.3.3. CONCURRENTS DES DISQUES DURS

SSD

Un SSD (pour *Solid State Drive*) peut avoir extérieurement l'apparence d'un disque dur classique, y compris l'interface, ou avoir un format plus réduit (mSATA, mSATA half-size, autrement dit demi-format) mais est dans tous les cas constitué de plusieurs puces de mémoire flash et ne contient **aucun élément mécanique**.

Par rapport à un disque dur, les temps d'accès sont très rapides pour une consommation généralement inférieure, mais lors de leur lancement, leur capacité était encore limitée à 512 Mo et leur prix très élevé.

Depuis 2008, la commercialisation par la plupart des grands constructeurs (Apple, Acer, Asus, Sony, Dell, Fujitsu, Toshiba,...), d'ordinateurs portables (généralement des ultra portables) équipés de SSD arrive sur le marché. Comme toute nouvelle technologie les caractéristiques évoluent très rapidement (source wikipédia) :

- En 2009, on trouve des modèles de 128 Go à des prix d'environ 350 \$ ce qui reste nettement plus cher qu'un disque dur ;
- En 2011, on trouve des SSD de 128 Go à moins de 200 euros, et la capacité des SSD disponibles dépasse désormais 1 To ;
- Fin 2012, on trouve des SSD de 128 Go aux alentours de 75 euros ;
- Fin 2012, on trouve des SSD de 240 Go aux alentours de 80 euros ;
- en 2016, on trouve des SSD de 1 To aux alentours de 300 euros.

SSHD Disque dur hybride

À mi-chemin entre le disque dur et le SSD, les disques durs hybrides (SSHD) sont des disques magnétiques classiques accompagnés d'un petit module de mémoire Flash (8 à 64 Go selon le fabricant) et d'une mémoire cache (8 à 64 Mo selon le fabricant).

Développé en priorité pour les portables, l'avantage de ces disques réside dans le fait de réduire la consommation d'énergie, d'augmenter la vitesse de démarrage et d'augmenter, enfin, la durée de vie du disque dur.

Lorsqu'un ordinateur portable équipé d'un disque hybride a besoin de stocker des données, il les range temporairement dans la mémoire Flash, ce qui évite aux pièces mécaniques de se mettre en route.

L'utilisation de la mémoire Flash devrait permettre d'améliorer de 20 % les chargements et le temps de démarrage des PC. Les PC portables devraient quant à eux profiter d'une augmentation d'autonomie de 5 à 15 %, ce qui pourrait se traduire par un gain de 30 minutes sur les dernières générations de PC portables.

2.4. L'ORGANISATION SEQUENTIELLE

Les enregistrements d'un fichier peuvent être écrits physiquement dans l'ordre **chronologique** de leur écriture. En lecture, ils seront obtenus dans le même ordre. Ils peuvent être bloqués ou non.

Ces caractéristiques sont celles de **l'organisation séquentielle**.

Il existe d'autres types d'organisations telles que :

- organisation directe,
- organisation relative,
- organisation de liste,
- organisation séquentielle indexée,
- organisation partitionnée.

Pour chaque type d'organisation, il existe une technique particulière pour accéder aux enregistrements.

Ces techniques sont appelées méthodes d'accès.

Méthodes d'accès pour les fichiers en organisation séquentielle : SAM (Sequential Access Method).

Types d'accès :

- Ecriture en création (à partir du début du fichier).
- Ecriture en extension (à partir de la fin du fichier).
- Lecture.
- Les mises à jour directes sont impossibles.

Avantages :

- C'est l'organisation la plus performante en traitement séquentiel.
- Bonne occupation du support (enregistrements consécutifs).
- Support économique (bandes).

Inconvénients :

- Pas d'accès direct.
- Mise à jour parfois complexe (plusieurs fichiers avec parfois plusieurs niveaux de rupture).

2.5. INTEGRITE DES DONNEES

Lors des traitements informatiques, l'un des risques les plus importants est la perte ou la corruption de l'information.

Nous verrons par la suite les moyens logiques de limiter ce risque (contrôle des données ...). Il arrive cependant que des fichiers soient perdus ou endommagés, par exemple à la suite de pannes matérielles. Pour pallier ces impondérables, le service exploitation ou service production, procède à des sauvegardes régulières des données. Ces sauvegardes consistent généralement à copier physiquement (sans contrôle de contenu) les fichiers disques sur cartouches ou sur d'autres serveurs.

2775cbe5ce
Global Knowledge

Index

A

actions	10, 14
Algèbre de Boole	22
algorithme	2
algorithmes	6
algorithmique	2
Alternative	18
alternatives <i>imbriquées</i>	37
analyse	4
analyse ascendante	5
analyse descendante	4
architectures distribuées	48

B

boîte blanche	69
boîte noire	69
Bottom-up	5
branchement inconditionnel	39

C

cas d'utilisation	70
cas d'utilisation	71
choix multiple	37
classe	36
clé primaire	48
client-serveur	48
Commit	54
COMMIT	51, 53
conception	5, 70
contraintes d'intégrité	51

D

DBA	43
déclaration d'une fonction	60
données	10
Données	13

E

ET conjonction	23
----------------	----

F

fichier	42
fichier séquentiel	43
fichier séquentiel indexé	43
fichier texte	42
fonction	58

G

global	64
--------	----

I

identificateur	13
imbrication	63
imbrication de structures	12
index	43, 48
Itération	18

J

journal	50
journalisation	50, 51

L

langage objet	36
local	64
Loi de Morgan	27

M

méthode	22
methodologie	2, 3
mode transactionnel	48
module	56
modules	4, 57
modules réutilisables	5

N

NON conjonction	25
-----------------	----

O

objet	36
ordinogramme	16, 39
organigramme	16
orientés objets	36
OU conjonction	24
OU inclusif	24

P

paramètre	59
paramètres	56, 60
persistance	48, 54
procédure	58
programmation structurée	2, 3
pseudo-code	17

R

réalisation	6
recette	7, 70
répétitif	10, 12
restauration	48
Rollback	54
ROLLBACK	51, 53

S

sauvegarde	48
scénarii de tests	70
schémas	16
sélectif	10, 11
séquentiel	10
SGBDR	48, 49
source	3
SQL	48, 49
structure de données	34
structures d'enchaînements	10

T

table	48
table de vérité	23
tableau	32
technologie objet	54
technologies objet	65

test	70
test d'intégration	7
test opérationnel	7
test système	7
test unitaire	7
tests	69
Top-down	4
transactionnel	53
transmission de paramètres	59
transmission des arguments	59
type	13
type alphanumérique	48
type booléen	31
type caractère	31
type numérique	30

V

valeur d'une donnée	13
---------------------	----