Chapitre 7 : Communication homme-machine

7.1. Les interfaces de communication logiciel/utilisateur

- Objets d'E/S
- temps d'attente
- pilotage
- enchaînement
- résistance aux erreurs

7.2. Grammaire pour analyser des phrases

- Un retour sur les grammaires
- Grammaire LL(1) pour analyse déterministe
- Analyser des phrases

7.3. Interface et pilotage en mini-français

- Un peu plus loin avec l'interaction et le pilotage
- Une méthode de construction

7.4. Projet d'IHM : enquête fumeurs

- Construction d'une borne interactive
- Mode saisie et plans d'actions
- Reste du logiciel

7.5. Utilisation des bases de données

- Introduction et généralités
- Le modèle de données relationnel
- Principes fondamentaux d'une algèbre relationnelle
- SQL et algèbre relationnelle
- Exemples de communication de Delphi avec une BD

Chapitre 7.1 interfaces de communication logiciel / utilisateur

Plan du chapitre:

Introduction

Interface homme-machine les concepts :

- **Les objets d'entrée-sortie**
- **Les temps d'attente**
- **Le pilotage de l'utilisateur**
- **Les types d'interaction**
- **L'enchaînement des opérations**
- **La résistance aux erreurs**

Introduction

Nous énumérons quelques principes utiles à l'élaboration d'une interface associée étroitement à la programmation événementielle. Le lecteur qui connaît le sujet peut passer au chapitre suivant.
Notre point de vue reste celui du pédagogue et non pas du spécialiste en ergonomie ou en psychologie cognitive qui sont deux éléments essentiels dans la conception d'une interface homme-machine (**IHM**). Nous nous efforcerons d'utiliser les principes généraux des **IHM** en les mettant à la portée d'un débutant avec un double objectif :

- Faire écrire des programmes interactifs. Ce qui signifie que le programme doit communiquer avec l'utilisateur qui reste l'acteur privilégié de la communication. Les programmes sont Windows-like et nous nous servons du RAD visuel Delphi pour les développer. Une partie de la spécification des programmes s'effectue avec des objets graphiques représentant des classes(programmation objet visuelle).

- Le programmeur peut découpler pendant la conception la programmation de son interface de la programmation des tâches internes de son logiciel (pour nous généralement ce sont des algorithmes ou des scénarios objets).

Nous nous efforçons aussi de ne proposer que des outils pratiques qui sont à notre portée et utilisables rapidement avec un système de développement RAD visuel comme Delphi.

Interface homme-machine, les concepts

Les spécialistes en ergonomie conceptualisent une IHM en six concepts :

- les objets d'entrée-sortie,
- les temps d'attente (temps de réponse aux sollicitations),
- le pilotage de l'utilisateur dans l'interface,
- les types d'interaction (langage,etc..) ,
- l'enchaînement des opérations,
- la résistance aux erreurs (ou robustesse qui est la qualité qu'un logiciel à fonctionner même dans des conditions anormales).

Un principe général provenant des psycho-linguistes guide notre programmeur dans la complexité informationnelle : la mémoire rapide d'un humain ne peut être sollicitée que par un nombre limité de concepts différents en même temps (nombre compris entre sept et neuf).
Développons un peu plus chacun des six concepts composants une interface.

Les objets d'entrée-sortie

Une IHM présente à l'utilisateur un éventail d'informations qui sont de deux ordres : des commandes entraînant des actions internes sur l'IHM et des données présentées totalement ou partiellement selon l'état de l'IHM.

Les commandes participent à la " saisie de l'intention d'action" de l'utilisateur, elles sont matérialisées dans le dialogue par des ***objets d'entrée*** de l'information (boîtes de saisie, boutons, menus etc...).

Voici avec un RAD visuel comme Delphi, des objets visuels associés aux objets d'entrée de l'information , ils sont très proches visuellement des objets que l'on trouve dans d'autres RAD visuels, car en fait ce sont des surcouches logiciels de contrôles de base du système d'exploitation (qui est lui-même fenêtré et se présente sous forme d'une IHM dénommée bureau électronique).

un bouton

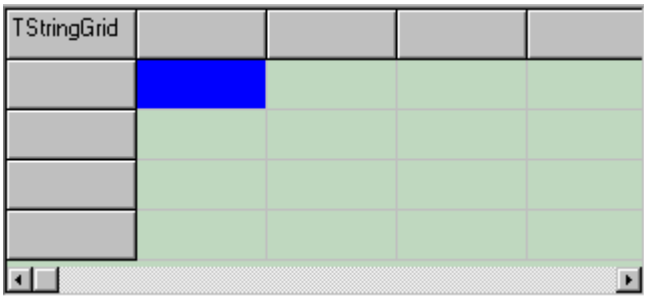
une boîte de saisie mono-ligne

Un menu

une boîte de saisie multi-ligne

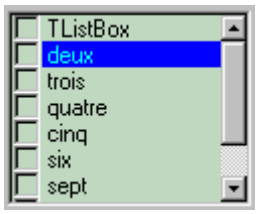
Les données sont présentées à un instant précis du dialogue à travers des ***objets de sortie*** de l'information (boîte d'édition monoligne, multiligne, tableaux, graphiques, images, sons etc...).

Ci-dessous quelques objets visuels associés à des objets de sortie de l'information :

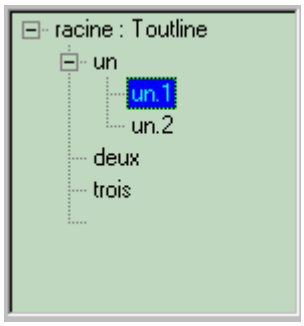


un TstringGrid (tableau)

un Timage(image jpg, png, bm ,ico,...)



un TlistBox (liste)



un Touline (arbre)

Les temps d'attente

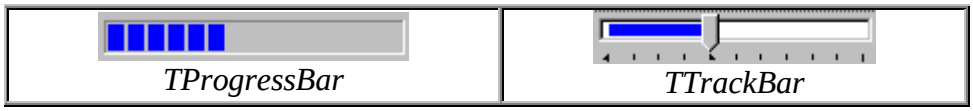
Concept

Sur cette question, une approche psychologique est la seule réponse possible, car l'impression d'attente ne dépend que de celui qui attend selon son degré de patience. Toutefois, puisque

nous avons parlé de la mémoire à court terme (mémoire rapide), nous pouvons nous baser sur les temps de persistance généralement admis (environ 5 secondes).

Nous considérerons qu'en première approximation, si le délai d'attente est :
- inférieur à environ une seconde la réponse est quasi-instantanée,
- compris entre une seconde et cinq secondes il y a attente, toutefois la mémoire rapide de l'utilisateur contient encore la finalité de l'opération en cours.
- lorsque l'on dépasse la capacité de mémorisation rapide de l'utilisateur alors il faut soutenir l'attention de l'utilisateur en lui envoyant des informations sur le déroulement de l'opération en cours (on peut utiliser pour cela par exemple des barres de défilement, des jauges, des boîtes de dialogue, etc...)

Exemples de quelques classes d'objets visuels de gestion du délai d'attente :



TStatusBar (avec deux panneaux ici)

Le pilotage de l'utilisateur

Concept

Nous ne cherchons pas à explorer les différentes méthodes utilisables pour piloter un utilisateur dans sa navigation dans une interface. Nous adoptons plutôt la position du concepteur de logiciel qui admet que le futur utilisateur ne se servira de son logiciel que d'une façon épisodique. Il n'est donc pas question de demander à l'utilisateur de connaître en permanence toutes les fonctionnalités du logiciel.

En outre, il ne faut pas non plus submerger l'utilisateur de conseils de guides et d'aides à profusion, car ils risqueraient de le détourner de la finalité du logiciel. Nous préférons adopter une ligne moyenne qui consiste à fournir de petites aides rapides contextuelles (au moment où l'utilisateur en a besoin) et une aide en ligne générale qu'il pourra consulter s'il le souhaite. Ce qui revient à dire que l'on accepte deux niveaux de navigation dans un logiciel :

- le niveau de surface permettant de réagir aux principales situations,
- le niveau approfondi qui permet l'utilisation plus complète du logiciel.

Il faut admettre que le niveau de surface est celui qui restera le plus employé (l'exemple d'un logiciel de traitement de texte courant du commerce montre qu'au maximum 30% des fonctionnalités du produit sont utilisées par plus de 90% des utilisateurs).

Pour permettre un pilotage plus efficace on peut établir à l'avance un graphe d'actions possibles du futur utilisateur (nous nous servirons du graphe événementiel) et ensuite diriger l'utilisateur dans ce graphe en matérialisant (masquage ou affichage) les actions qui sont

réalisables. En application des notions acquises dans les chapitres précédents, nous utiliserons un pilotage dirigé par la syntaxe comme exemple.

Les types d'interaction

Concept

Le tout premier genre d'interaction entre l'utilisateur et un logiciel est apparu sur les premiers systèmes d'exploitation sous la forme d'un langage de commande. L'utilisateur dispose d'une famille de commandes qu'il est censé connaître, le logiciel étant doté d'une interface interne (l'interpréteur de cette famille de commandes). Dès que l'utilisateur tape textuellement une commande (exemple MS-DOS " **dir c: /w** "), le système l'interprète (dans l'exemple : lister en prenant toutes les colonnes d'écran, les bibliothèques et les fichiers du disque C).

Nous adoptons comme mode d'interaction entre un utilisateur et un logiciel, une extension plus moderne de ce genre de dialogue, en y ajoutant, en privilégiant, la notion d'objets visuels permettant d'effectuer des commandes par actions et non plus seulement par syntaxe textuelle pure.
Nous construisons donc une interface tout d'abord ***essentiellement*** à partir des interactions événementielles, puis lorsque cela est utile ou nécessaire, nous pouvons ajouter un interpréteur de langage (nous pouvons par exemple utiliser des automates d'états finis pour la reconnaissance).

L'enchaînement des opérations

Concept

Nous savons que nous travaillons sur des machines de Von Neumann, donc séquentielles, les opérations internes s'effectuant selon un ordre unique sur lequel l'utilisateur n'a aucune prise. L'utilisateur est censé pouvoir agir d'une manière " aléatoire ". Afin de simuler une certaine liberté d'action de l'utilisateur nous lui ferons parcourir un graphe événementiel prévu par le programmeur. Il y a donc contradiction entre la rigidité séquentielle imposée par la machine et la liberté d'action que l'on souhaite accorder à l'utilisateur. Ce problème est déjà présent dans un système d'exploitation et il relève de la notion de gestion des interruptions.

Nous pouvons trouver un compromis raisonnable dans le fait de découper les tâches internes en tâches séquentielles minimales ininterruptibles et en tâches interruptibles.

Les interruptions consisteront en actions potentielles de l'utilisateur sur la tâche en cours afin de :

- interrompre le travail en cours,
- quitter définitivement le logiciel,
- interroger un objet de sortie,
- lancer une commande exploratoire ...

Il faut donc qu'existe dans le système de développement du logiciel, un mécanisme qui permette de " *demander la main au système* " sans arrêter ni bloquer le reste de l'interface, ceci pendant le déroulement d'une action répétitive et longue. Lorsque l'interface a la main, l'utilisateur peut alors interrompre, quitter, interroger...

Ce mécanisme est disponible dans les RAD visuels pédagogiques (Delphi,Visual Basic, Visual C#), nous verrons comment l'implanter. Terminons ce tour d'horizon, par le dernier concept de base d'une interface : sa capacité à absorber certains dysfonctionnements.

La résistance aux erreurs

Concept

Il faut en effet employer une méthode de programmation défensive afin de protéger le logiciel contre des erreurs comme par exemple des erreurs de manipulation de la part de l'utilisateur. Nous utilisons plusieurs outils qui concourent à la robustesse de notre logiciel. La protection est donc située à plusieurs niveaux.

1°) Une protection est apportée par le graphe événementiel qui n'autorise que certaines actions (activation-désactivation), matérialisé par un objet tel qu'un menu :

l'item " taper un fichier " n'est pas activable

l'item " Charger un fichier " est activable

2°) Une protection est apportée par le filtrage des données (on peut utiliser par exemple des logiciels d'automates de reconnaissance de la syntaxe des données).

3°) Un autre niveau de protection est apporté par les composants visuels utilisés qui sont sécurisés dans le RAD à l'origine. Par exemple la méthode LoadFromfile de Delphi qui permet le chargement d'un fichier dans un composant réagit d'une manière sécuritaire (c'est à dire rien ne se produit)lorsqu'on lui fournit un chemin erroné ou que le fichier n'existe pas.

4°) Un niveau de robustesse est apporté en Delphi par une utilisation des exceptions (semblable à Ada ou à C++) autorisant le détournement du code afin de traiter une situation interne anormale (dépassement de capacité d'un calcul, transtypage de données non conforme etc...). Le programmeur peut donc prévoir les incidents possibles et construire des gestionnaires d'exceptions.

Chapitre 7.2 grammaire pour analyser des phrases

Plan du chapitre:

Bien qu'il ne soit pas dans les objectifs de cet ouvrage de systématiser les méthodes de reconnaissance, nous abordons le sujet sur des cas et des exemples particuliers. L'attitude de systématisation méthodologique et pratique adoptée devrait fournir matière à réflexion et profiter à l'étudiant, nous nous servirons de ces outils pour améliorer ses IHM en particulier dans l'analyse des interactions avec l'utilisateur.

1. Un retour sur les grammaires

L'expérience a montré qu'un des cas particuliers abordables en initiation est celui où une C-grammaire G possède la propriété d'être analysable par analyse LL(1). Ces analyseurs sont plus simples à construire, et surtout il est possible de systématiser leur construction. Il apparaît que beaucoup de grammaires sont LL(1) et qu'un très grand nombre d'exemples du niveau étudiant débutant peuvent être décrits par une grammaire LL(1). C'est pourquoi nous fournirons plus loin un exemple de génération manuelle d'un petit analyseur de mots du genre LL(1) à partir d'un TAD, ainsi qu'une définition d'une grammaire LL(1).

1.1 Dérivation à droite et à gauche

Nous dirons par définition que x se ***dérive le plus à gauche*** en y et l'on écrira :

$x \overset{lpg}{\Rightarrow} y$ *si et seulement si :*	$\exists\ \alpha \in (V_N \cup V_T)^*$ $\exists\ r_i : A \rightarrow$ avec $A \in V_N$ si $x = A\alpha$ alors $y = \beta\alpha$

Nous dirons par définition que x se ***dérive le plus à droite*** en y et l'on écrira :

$x \overset{lpd}{\Rightarrow} y$ *si et seulement si :*	$\exists\ \alpha \in (V_N \cup V_T)^*$ $\exists\ r_i : A \rightarrow$ avec $A \in V_N$ si $x = \alpha A$ alors $y = \alpha\beta$

Comme pour la dérivation, on définit la fermeture transitive de chacune de ces deux relations binaires. Nous les notons :

$x \overset{lpd}{\Rightarrow}{}^{*} y$ **et** $x \overset{lpg}{\Rightarrow}{}^{*} y$.

1.2 Ensembles First et Init

Nous allons définir quelques ensembles de symboles utiles à une reconnaissance des mots dans une grammaire G, $G = (V_N, V_T, S, R)$.

Notations : Les ensembles First et leurs propriétés

Soient $(x,y) \in (V_N \cup V_T)^{*2}$, $A \in V_N$ (on suppose que A possède dans le cas général plusieurs expansions $A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$, le symbole S est l'axiome de la grammaire G. Nous notons :

First(x) $= \{ \mathbf{a} \in V_T \ / \ x \overset{lpg}{\Rightarrow} * \mathbf{a}y \}$

Autrement dit pour l'élément $a \in$ First(x) nous avons :

> **a** est un symbole du vocabulaire terminal V_T qui commence toute chaîne qui se dérive de x (ε inclus), comme dans $x \Rightarrow * \mathbf{a}\beta$.

Enonçons deux propriétés qui vont servir en pratique.

Propriété n°1.2.1 :

si $\alpha \in V_T^*$, **First** $(\alpha x) = \{\alpha\}$

Propriété n°1.2.2 :

$\forall (x,y) \in (V_N \cup V_T)^{*2}$ **First**(xy) = **First**(x),
sauf dans le cas où $x \Rightarrow * \varepsilon$, on a alors :
First(xy) = **First**(x) $\cup$ **First**(y)

Définition des ensembles Initiaux :

Init $(A) = \bigcup_{k=1}^{n}$ **First** (α_k)
où : α_k est l'une des expansions de $A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$
convention : si $A \in V_T$, **Init** $(A) = \{A\}$

Propriété n°1.2.3 :

Dans le cas où il n'y a qu'une expansion pour A :
$A \rightarrow \alpha$
Nous avons : **Init** (A) = **First** (α)

Ces ensembles **Init** (A) permettent de rassembler les symboles terminaux qui se trouvent au début de tout mot dérivé par chacune des expansions de A ($A \rightarrow \alpha_1 \mid ... \mid \alpha_n$). Nous traitons un exemple complet plus loin.

1.3 Ensemble Follow

Seulement dans le cas où $A \in V_N$, on calcule l'ensemble suivant :

Follow(A) = { **a** $\in V_T$ / S $\Rightarrow^* \alpha Ax$ (où $\alpha \in V_T$, et **a** $\in$ **Init** (x)) }

Cet ensemble correspond aux symboles qui suivent les mots dérivés de A dans la grammaire. Ce sont les éléments terminaux **a** $\in V_T$ apparaissant immédiatement à droite de A dans toutes les chaînes contenant A, comme dans $S \Rightarrow \alpha A \mathbf{a} \beta$

Grâce aux ensembles **Init** et **Follow**, nous pouvons dire que nous disposons de deux familles de symboles qui encadrent les mots de la grammaire G. Cette remarque soulève le voile sur une stratégie pratique de reconnaissance des mots engendrés par une grammaire.

Enonçons une propriété calculatoire pratique, des **Follow** qui nous servira:

propriété n°1.3.1 :

si $A \rightarrow \alpha B$, $A \in V_N$, $B \in V_N$, $\alpha \in V_T^*$ **alors**
Follow (A) $\subset$ **Follow** (B)
fsi

1.4 Calcul des Follows à partir des First

Soit G une C-grammaire, $G = (V_N, V_T, S, R)$.

Calcul des Follow(A) où $A \in V_N$:
Les règles contenant A en partie droite peuvent avoir d'une manière générale deux formes :
$\mathbf{B} \rightarrow \alpha A \beta$ ou $\mathbf{B} \rightarrow \alpha A$ ($B \in V_N$).

Pour toute règle de la forme $\mathbf{B} \rightarrow \alpha A \beta$, (si $\varepsilon \notin$ **First (β)**)

Follow (A) = Follow (A) $\cup$ First (β)

Pour toute règle de la forme $\mathbf{B} \rightarrow \alpha A \beta$, (si $\varepsilon \in$ **First** (β))
et Pour toute règle de la forme $\mathbf{B} \rightarrow \alpha A$

Follow (A) = Follow (A) $\cup$ Follow (B)

Exemple de calcul sur une C-grammaire G :

```
VT = {a, b}
VN = {A, S}
axiome: S
Règles :
```

1 : S → aAS
2 : S → b
3 : A → a
4 : A → bSA

On suppose en outre, que tous les mots de G se termineront par le symbole spécial '#' de fin de mot.

Calcul de Init(S)	
Il y a 2 expansions de S (Règle 1 et Règle 2) **Init**(S) = **First** (aAS) ∪ **First** (b) nous avons : **First** (b) = {b} nous avons : **First** (aAS) = {a}	Donc **Init**(S) = {a} ∪ {b}
Calcul de Init(A)	
Il y a 2 expansions de A (Règle 3 et Règle 4) **Init** (A) = **First** (a) ∪ **First** (bSA) nous avons : **First** (a) = {a} nous avons : **First** (bSA) = {b}	Donc : **Init** (A) = {a} ∪ {b}

De ces deux calculs nous concluons que : **Init (S) = Init (A) = {a,b}**

Passons au calcul des Follow à l'aide des **Init** que nous venons d'évaluer.

Calcul de Follow(S)	
S est l'axiome de la grammaire d'après la définition: **Follow** (S) = {#} Dans la règle 4 (**A** → bSA) d'après la définition: **Follow** (S) = **Follow** (S) ∪ **Init** (A)	D'où : **Follow (S) = { # , a , b }**
Calcul de Follow(A)	
Initialisation de **Follow** (A) = ∅ Dans la règle 4 (**S** → aAS) d'après la définition : **Follow** (A) = **Init** (S)	D'où : **Follow (A) = { a , b }**

Résultats obtenus	
VT = {a, b} **VN = {A, S}** **axiome: S** **Règles :** **1 : S → aAS** **2 : S → b** **3 : A → a** **4 : A → bSA**	**Init (S) = Init (A) = {a,b}** **Follow (S) = { # , a , b }** **Follow (A) = { a , b }**

En langage pratique nous pouvons dire que toute chaîne qui dérive de **A** comme de **S** commence soit par un symbole **a** ou par un symbole **b**, toute chaîne qui suit un mot dérivé de **A** commence par un symbole **a** ou par un symbole **b**, de même , enfin toute chaîne qui suit un mot dérivé de **S** commence par un symbole **a** ou par un symbole **b** ou un symbole **#.**

1.5 Calcul pour une grammaire arithmétique

Prenons un exemple plus pratique en utilisant une grammaire G_{exp} ambiguë des expressions arithmétiques, déjà proposée auparavant :

$G_{exp} = (V_N, V_T, \text{Axiome}, \text{Règles})$
$\mathbf{V_T}$ ={ **0**, ..., **9**, **+**, **-**, **/**, *****, **)**, **(** }
$\mathbf{V_N}$ = { ⟨ Expr ⟩, ⟨ Nbr ⟩, ⟨ Cte ⟩, ⟨ Oper ⟩ }
Axiome : ⟨ Expr ⟩
Règles :
1 : ⟨ Expr ⟩ → ⟨ Nbr ⟩ | **(** ⟨ Expr ⟩ **)** | ⟨ Expr ⟩ ⟨ Oper ⟩ ⟨ Expr ⟩
2 : ⟨ Nbr ⟩ → ⟨ Cte ⟩ | ⟨ Cte ⟩ ⟨ Nbr ⟩
3 : ⟨ Cte ⟩ → **0** | **1** |...| **9**
4 : ⟨ Oper ⟩ → **+** | **-** | ***** | **/**

Calcul des ensembles Init

Provenant de la règle n°1

⟨Expr⟩ → ⟨Nbr⟩ | **(** ⟨Expr⟩ **)** | ⟨Expr⟩ ⟨Oper⟩ ⟨Expr⟩

D'après la définition des Init :
Init (Expr) = **First** (Nbr) ∪ **First** ((Expr)) ∪ **First**(Expr Oper Expr)

D'après la propriété 1.2.1 [*A = (Expr) est de la forme A = **aB** où **a** = (*] donc :
First ((Expr))={'**(**'}

Comme ε ne dérive pas de Expr , nous avons :
First (Expr Oper Expr) = **First** (Expr)

Comme Nbr ∈ V_N possède deux expansions, son **First** n'est pas calculable immédiatement, il faut calculer son ensemble **Init** (Nbr).

Premier résultat obtenu

Init (Expr) = Init (Nbr) ∪{ **(** }

Provenant de la règle n°2
<Nbr> → <Cte> | <Cte> <Nbr> D'après la définition des Init : **Init** (Nbr) = **First** (Cte) ∪ **First**(Cte Nbr) D'après la propriété 1.2.2: **First**(Cte Nbr) = **First** (Cte) ∪ **First**(Nbr) Comme Cte ∈ V_N possède plusieurs expansions, son **First** n'est pas calculable immédiatement, il faut calculer son ensemble **Init** (Cte).

second résultat obtenu
Init (Nbr) = Init (Cte)

Nous terminons les calculs car il ne reste que des règles terminales ce qui donne des calculs de **First** immédiats.

Provenant de la règle terminale n°3
<Cte> → 0 | 1… | 9 D'après la définition des Init : **Init** (Cte) = **First** (0) ∪ ... ∪ **First**(9) = { 0,1, …, 9 }
troisième résultat obtenu
Init (Cte) = { 0,1, …, 9 }
Provenant de la règle terminale n°4
⟨Oper⟩ → + | - | * | / D'après la définition des Init : **Init** (Oper) = **First** (+) ∪ **First**(-) ∪ **First**(*) ∪ **First**(/) = { +, - , *, / }
quatrième résultat obtenu
Init (Oper) = { +, - , *, / }
En rassemblant les résultats calculés nous obtenons les **Init** de chaque élément de V_N de la grammaire G_{exp} des expressions arithmétiques

Init (Oper) = { +, - , * , / } **Init** (Cte) = { 0,1,...,9 } **Init** (Nbr) = { 0,1,...,9 } **Init** (Expr) = { 0,1,...,9,(}	Rappellons la signification pratique par exemple de l'ensemble **Init** (Expr) = { 0,1,...,9,(}: Toute expression dérivant de Expr commence par un chiffre de 0 à 9 ou bien commence par une parenthèse ouvrante '(' .

Calcul des ensembles Follow

Calcul de **Follow** (Expr)

Nous devons chercher dans toutes les règles de la grammaire celles qui contiennent **en partie droite le non terminal Expr**. Lorsque dans une règle de la forme " A → α<Expr>β " , Expr est en partie droite, en appliquant la définition, nous obtenons le fait que **Follow** (Expr) contient le First(β). Nous procédons donc ici à un calcul incrémental de l'ensemble **Follow** (Expr) par adjonctions successives des **First** qui le composent.

Provenant de la règle n°1

Les 2 premières expansions :
⟨Expr⟩ → ⟨Nbr⟩
⟨Expr⟩ → **(** ⟨Expr⟩ **)**

Init (‘**)**‘) = **First** (‘**)**’) ⊂ **Follow** (Expr) *d'après la définition*

La troisième expansion :
⟨Expr⟩ → ⟨Expr⟩ ⟨Oper⟩ ⟨Expr⟩

Init (Oper) ⊂ **Follow** (Expr) *d'après la définition*

Comme il n'y a pas dans G_{exp} d'autres règles contenant le symbole Expr en partie droite, donc le calcul du **Follow** (Expr) est terminé , **Follow** (Expr) ne contient que les ensembles **Init** (‘**)**‘) et **Init** (Oper) :

Follow (Expr) = **Init** (‘**)**‘) ∪ **Init** (Oper)

Premier Follow obtenu

Follow (Expr) = { +, - , *, / ,) }

Le calcul est identique pour tous les autres follow

Calcul de **Follow** (Nbr) *(examen de toutes les parties droites contenant Nbr)*

Provenant de la règle n°1

⟨Expr⟩ → ⟨Nbr⟩

D'après la propriété 1.3.1 :
Follow (Expr) ⊂ **Follow** (Nbr)

Provenant de la règle n°2

⟨ Nbr ⟩ →⟨Cte⟩ ⟨Nbr⟩

rien de plus n'est ajouté.

second Follow obtenu

Il n'y a pas d'autres règles contenant le symbole Nbr en partie droite, donc le calcul du **Follow** (Nbr) est terminé :

Follow (Nbr) = **Follow** (Expr) = {+, - , * , / ,) }

Calcul de **Follow** (Cte) *(examen de toutes les parties droites contenant Cte)*

Provenant de la règle n°2

⟨ Nbr ⟩ →⟨Cte⟩ ⟨Nbr⟩

D'après la propriété 1.3.1 :
Follow (Nbr) ⊂ **Follow** (Cte)

D'après la définition :
Init (Nbr) ⊂ **Follow** (Cte)

troisième Follow obtenu

Il n'y a pas d'autres règles contenant le symbole Cte en partie droite, donc le calcul du **Follow** (Cte) est terminé :

Follow (Cte) = **Follow** (Nbr) ∪ **Init** (Nbr) = {+, - , * , / ,) , 0 , 1 , … , 9 }

Calcul de **Follow** (Oper) *(examen de toutes les parties droites contenant Oper)*

Provenant de la règle n°1

⟨ Nbr ⟩ →⟨Cte⟩ ⟨Nbr⟩

D'après la définition :
Init (Expr) ⊂ **Follow** (Oper)

quatrième Follow obtenu

Il n'y a pas d'autres règles contenant le symbole Oper en partie droite, donc le calcul du **Follow** (Oper) est terminé : `Follow(Oper)= Init(Expr)= { 0,1,...,9,( }`

Follow (Oper) = **Init** (`Expr`) = { 0 , 1 , … , 9 , (}

Récapitulons les **Init et les Follow** de chaque élément de V_N de la grammaire G_{exp}

Init (Expr) = { 0,1,...,9,(} **Init** (Nbr) = { 0,1,...,9 } **Init** (Cte) = { 0,1,...,9 } **Init** (Oper) = { +, - , * , / } **Follow** (Expr) = { +, - , *, / ,) } **Follow** (Nbr) = {+, - , * , / ,) } **Follow** (Cte) = {+, -, *, /,), 0, 1, … ,9 } **Follow** (Oper) = { 0 , 1 , … , 9 , (}	Rappellons la signification pratique par exemple des ensembles **Follow.** **Follow** (Expr): Tout mot suivant une expression dérivant de Expr commence par +, - , *, / ,). **Follow** (Cte): Tout mot suivant une constante dérivant de Cte commence par +, -, *, /,), 0, 1, … ,9 . **Follow** (Oper): Tout mot suivant un opérateur dérivant de Oper commence par 0 , 1 , … , 9 , (.

Nous pouvons savoir ainsi en consultant ces ensembles si les expressions suivantes sont correctes ou incorrectes dans G_{exp} :

> **12(5+3)** est incorrecte car la parenthèse ouvrante qui est le First de (5+3) n'est pas dans le Follow du nombre 12 (il n'y a pas de parenthèse ouvrante après un nombre).
>
> **12*)+2** est incorrecte car la parenthèse fermante qui est le First de)+2 n'est pas dans le Follow de l'opérateur * (il n'y a pas de parenthèse fermante après un opérateur).

1.6 Calcul pour une grammaire du mini-français

Soit G_{F1} une grammaire déjà étudiée au chapitre 6 d'un mini-français.

G = (V_N,V_T,S,R)
V_T ={**le, un, chat, chien, aime, poursuit, malicieusement, joyeusement, gentil, noir, blanc, beau, '.'**}
V_N = { ⟨*phrase*⟩, ⟨*GN*⟩, ⟨*GV*⟩, ⟨*Art*⟩, ⟨*Nom*⟩, ⟨*Adj*⟩, ⟨*Adv*⟩, ⟨*verbe*⟩ }
Axiome : ⟨ *phrase* ⟩
Règles :

```
1 : ⟨ phrase ⟩→⟨ GN ⟩⟨ GV ⟩⟨ GN ⟩ .
2 : ⟨ GN ⟩→⟨ Art ⟩⟨ Adj ⟩⟨ Nom ⟩
3 : ⟨ GN ⟩→⟨ Art ⟩⟨ Nom ⟩⟨ Adj ⟩
4 : ⟨ GV ⟩→⟨ verbe ⟩ | ⟨ verbe ⟩⟨ Adv ⟩
5 : ⟨ Art ⟩→ le | un
6 : ⟨ Nom ⟩→ chien | chat
7 : ⟨ verbe ⟩→ aime | poursuit
8 : ⟨ Adj ⟩→ blanc | noir | gentil | beau
9 : ⟨ Adv ⟩→ malicieusement | joyeusement
```

Nous livrons ci-après le calcul des ensembles **Init** et **Follow** sans le détailler car il s'agit de l'application à cet exemple de la même stratégie utilisée dans le cas de la grammaire précédente. Il est conseillé au lecteur de refaire le calcul lui-même à titre d'entraînement.

Indications sur le calcul des ensembles Init :

Init (Phrase) = **Init** (GN)
Init (GN) = **Init** (Art)
Init (GV) = **Init** (Verbe)
Init (Art) = {le,un}
Init (Nom) = {chat,chien}
Init (Verbe) = { aime, poursuit }
Init (Adj) = { gentil, noir, blanc, beau }
Init (Adv) = { malicieusement, joyeusement }

Récapitulatif :

Init (Phrase) = {le,un}
Init (GN) = {le,un}
Init (GV) = { aime, poursuit }
Init (Art) = {le,un}
Init (Nom) = {chat,chien}
Init (Verbe) = { aime, poursuit }
Init (Adj) = { gentil, noir, blanc, beau }
Init (Adv) = { malicieusement, joyeusement }

Indications sur le calcul des ensembles Follow :

Follow(Phrase) = {#} *fin de chaîne*
Follow(GN) = Init(GV) ∪ Init('.') *à partir de règle.1*
Follow(GV) = Init(GN) *à partir de règle.1*
Follow(Art) = Init(Adj) ∪ Init(Nom) *à partir de règles.2 et 3*
Follow(Nom) = Follow(GN) ∪ Init(Adj) *à partir de règles.2 et 3*
Follow(Verbe) = Follow(GV) ∪ Init(Adv) *à partir de règle 4*
Follow(Adj) = Follow(GN) ∪ Init(Nom) *à partir de règles.2 et 3*
Follow(Adv) = Follow(GV) ∪ Init(Adj) *à partir de règle 4*

Récapitulatif

Follow (Phrase) = {#}
Follow (GN) = { aime, poursuit, '.' }
Follow (GV) = {le,un}
Follow (Art) = { gentil, noir, blanc, beau, chat, chien }
Follow (Nom) = { gentil, noir, blanc, beau, aime, poursuit }
Follow (Verbe) = { le, un, malicieusement, joyeusement }
Follow (Adj) = { aime, poursuit, chat, chien, '.' }
Follow (Adv) = {le,un}

Voyons maintenant comment nous pouvons utiliser pratiquement ces ensembles Follow et Init (ou First) et dans quel cadre s'en servir. Le support de stratégie se dénomme l'analyse LL(1).

Nous décortiquons dans le paragraphe suivant un exemple pratique complet en soulignant les aspects méthodologiques généraux sous-jacents.

2. Grammaire LL(1) pour analyse déterministe

Les ensembles Init et Follow précédemment étudiés présentent un intérêt pratique dans l'analyse de mots d'une C-grammaire d'un " bon type ". C'est en vue d'une construction ultérieure systématique du filtrage du dialogue homme-machine que nous les avons présentés. En outre ils pourront aussi, comme nous le verrons lorsque nous aborderons ce thème, diriger notre programmation par plans d'action dans le guidage d'une saisie sur une interface. Dans le cas de dialogue avec l'utilisateur, c'est le concepteur du logiciel qui prévoit le " genre " de dialogue. Donc s'il dispose d'un outil rapide d'analyse du dialogue, il pourra dans la majorité des cas essayer d'élaborer une grammaire analysable rapidement sans alourdir sa tâche de programmation.

Nous avons choisi de présenter la technique la plus simple pour reconnaître les mots d'un langage dans le cas où la C-grammaire est de type LL(1).

La stratégie générale que nous adoptons est la suivante :

A chaque analyse du symbole en cours, il nous suffira de " regarder " le symbole suivant. Si la grammaire s'y prête (et nous allons donner les propriétés de telles grammaires), nous pourrons connaître à l'avance l'ensemble de tous les symboles (tous différents)pouvant se trouver après le symbole analysé. Chacun des symboles conduira à une branche d'analyse différente, le procédé est donc déterministe.

Une bonne C-grammaire (donc de type-2)qui se prête à ce genre d'analyse est dite analysable par la technique LL(1) ou plus brièvement appelée grammaire LL(1).

Nous possédons un mécanisme de construction d'analyseur de mots avec les automates d'états finis pour les grammaire de type-3. Le procédé LL(1) est un outil simple mais suffisamment intéressant pour fournir un cadre méthodique et introductif à d'autres développements. Avec cet outil de nombreux exemples efficaces et non triviaux peuvent être construits et programmés au niveau de l'initiation.

2.1 Une condition pratique d'analysabilité LL(1)

Nous donnons une condition nécessaire et suffisante posée comme définition pour qu'une C-grammaire soit LL(1). Cette CNS est un énoncé constructif qui va nous servir à vérifier rapidement si nous avons une grammaire LL(1) ou non.

CNS-LL(1):

$\forall A \in V_N$, $A \rightarrow \alpha_1 | \alpha_2 | ..... | \alpha_n$
$\forall (i,j) / i \neq j$: **Init**$(\alpha_i) \cap$ **Init**$(\alpha_j) = \varnothing$
si $\alpha_k \Rightarrow^* \varepsilon$ **alors**
$\forall i / i \neq k$, **Init**$(\alpha_i) \cap$ **Follow**$(A) = \varnothing$
fsi

2.2 Méthode pratique de vérification

Afin de savoir si une C-grammaire donnée est LL(1) et pour appliquer la CNS précédente, nous construisons une table regroupant toutes les informations sur les Init et les Follow de chaque élément du vocabulaire auxiliaire de la grammaire.
Soient **A** → X1 |...| **X**n toutes les expansions de A, A ∈ V_N. Nous construisons le tableau suivant pour chaque élément A ∈ V_N :

Sym ∈ V_N	Init (Sym)				Follow(Sym)
	X_1	X_2	.....	X_n	
A	...	...		...	...

Où chaque colonne identifiée par $\mathbf{X}_k$ contient tous les symboles de l'ensemble **First**($\mathbf{X}_k$).

Puis, à l'aide de ce tableau, nous appliquons la CNS-LL1 :

- Pour un A, A ∈ V_N fixé, les contenus de chacune des colonnes **X**k ne doivent pas avoir d'éléments communs.
- Si une expansion **X**k de A se dérive en ε (**X**$_k \Rightarrow^* \varepsilon$), les contenus de toutes les autres colonnes **X**p (p ≠ k) ne doivent pas avoir d'éléments commun avec **Follow**(A).

Ci-après, les tables construites à partir des deux calculs précédents sur la grammaire des expressions arithmétiques et celle du mini-français :

	Init			**Follow**
Expr	**0,...,9**	**(**	**0,...,9**	**+ , * , -, /,)**
Nbr	**0,...,9**			**+ , * , -, /,)**
Cte	**0,...,9**			**+, *, -, /,), 0,...,9**
Oper	**+ , * , -, /**			**0,..., 9, (**

tableau pour la grammaire G_{exp}

Nous observons que G_{exp} n'est pas LL(1), car le premier **First** et le second **First** du symbole Expr ont une intersection non vide : **0,...,9.**

Remarque :

Ceci est en particulier dû au fait que la récursivité gauche pose un problème pour ce genre d'analyse. Il y a deux **First** identiques.

Décrivons maintenant le tableau associé à la grammaire du mini-français notée plus haut G_{F1}:

	Init		**Follow**
Phrase	**le, un**		**#**
GN	**le, un**	**le, un**	**aime, poursuit, '.'**
GV	**aime, poursuit**	**aime, poursuit**	**le, un**
Art	**le, un**		**blanc, noir, gentil, beau, chat, chien**
Nom	**chien, chat**		**blanc, noir, gentil, beau, aime, poursuit**
Verbe	**aime, poursuit**		**le,un,malicieusement, joyeusement**
Adj	**blanc,noir,gentil,beau**		**aime, poursuit, chat, chien**
Adv	**malicieusement, joyeusement**		**le, un**

tableau pour la grammaire G_{F1}

Cette grammaire G_{F1} n'est pas LL(1) à cause de l'**Init** de GN ou de celui de GV, qui ont une intersection non vide de leurs **First**. Le problème est dû à la présence dans cette grammaire de règles non déterministes de la forme : $\mathbf{A} \rightarrow \alpha_1 B \mid \alpha_1 C$, (où $B \in V_N$ et $C \in V_N$).

De telles règles $\mathbf{A} \rightarrow \alpha_1 B \mid \alpha_1 C$ ne peuventt pas être analysées avec la stratégie d'observation du prochain symbole, puisque les deux expansions $\alpha_1 B$ et $\alpha_1 C$ débutent chacune par les mêmes symboles (ceux de **First** (α_1)).

Nous pouvons éliminer ce problème en construisant une autre grammaire en ajoutant un élément A' au vocabulaire auxiliaire de G_{F1} et en remplaçant les règles $\mathbf{A} \rightarrow \alpha_1 B \mid \alpha_1 C$ par les deux règles suivantes (procédé classique en compilation appelé **factorisation**):

$\mathbf{A} \rightarrow \alpha_1 A'$
$\mathbf{A'} \rightarrow B \mid C$

(en espérant que **First** (A) et **First** (B) n'aient pas d'éléments communs)

Voyons enfin comment les outils que nous venons de considérer peuvent servir en programmation de l'analyse de phrases.

3.Analyser des phrases

3.1 Une grammaire LL(1) du mini-français

Voici G_{F2} une grammaire LL(1) obtenue à partir de G_{F1} par changement des règles selon l'idée de transformation précédente. Nous avons ajouté deux nouveaux symboles dans V_N: <LeNom> et <Suite>

$G_{F2} = (V_N, V_T, S, R)$
$\mathbf{V_T}$ ={le, un, chat, chien, aime, poursuit, malicieusement, joyeusement, gentil, noir, blanc, beau}
$\mathbf{V_N}$ = {⟨phrase⟩, ⟨GN⟩, ⟨GV⟩, ⟨Art⟩, ⟨Nom⟩, ⟨Adj⟩, ⟨Adv⟩, ⟨verbe⟩,⟨LeNom⟩, ⟨Suite⟩}
Axiome : ⟨phrase⟩
Règles :

```
1 :⟨ phrase ⟩→⟨ GN ⟩⟨ GV ⟩⟨ GN ⟩.
2 :⟨ GN ⟩→⟨ Art ⟩⟨ LeNom ⟩
3 :⟨ LeNom ⟩ →⟨ Adj ⟩⟨ Nom ⟩ | ⟨ Nom ⟩⟨ Adj ⟩
4 :⟨ GV ⟩→⟨ verbe ⟩⟨ Suite ⟩
5 :⟨ Suite ⟩ →⟨ GN ⟩ | ⟨ Adv ⟩⟨ GN ⟩
6 :⟨ Art ⟩→ le | un
7 :⟨ Nom ⟩→ chien | chat
8 :⟨ verbe ⟩→ aime | poursuit
9 :⟨ Adj ⟩→ blanc | noir | gentil | beau
10 :⟨ Adv ⟩→ malicieusement | joyeusement
```

En calculant comme précédemment les Init et les Follow, nous obtenons le tableau récapitulatif suivant:

	Init	Follow
Phrase	**le, un**	**#**
GN	**le, un**	**aime, poursuit, '.'**
LeNom	**blanc, noir, gentil, beau** / **chien, chat**	**aime, poursuit, '.'**
GV	**aime, poursuit**	**'.'**
Suite	**le, un** / **malicieusement, joyeusement**	**'.'**
Art	**le, un**	**blanc, noir, gentil, beau, chat, chien**
Nom	**chien, chat**	**blanc, noir, gentil, beau, aime, poursuit, '.'**
Verbe	**aime, poursuit**	**le,un,malicieusement, joyeusement**
Adj	**blanc, noir, gentil, beau**	**aime, poursuit, chat, chien, '.'**
Adv	**malicieusement, joyeusement**	**le, un**

ci-dessus un tableau pour la grammaire G_{F2}

Elle vérifie la CNS-LL1, cette grammaire G_{F2} est LL(1).
Nous indiquons ensuite un moyen destiné à effectuer la reconnaissance manuelle tout d'abord, puis par programme en Delphi par exemple, uniquement dans le cas de la grammaire LL(1) G_{F2} du mini-français. Cela pourra inciter l'étudiant à aller chercher dans les ouvrages spécialisés les méthodes générales mettant en œuvre ces techniques de reconnaissance.

3.2 Schémas d'algorithmes associés à G_{F2}

Nous supposons disposer d'un moyen d'extraire dans la phrase le symbole à analyser. Il sera mis dans la variable notée " **SymLu** ". Nous découpons notre travail d'analyse en blocs comme nous l'avions découpé lors de l'utilisation d'une grammaire en mode génération de phrases. Chaque bloc syntaxique est associé à un élément du vocabulaire auxiliaire V_N dans G_{F2}.
La démarche descendante reste semblable à la " programmation par la syntaxe" déjà vue et assure une cohérence pédagogique sur la méthode de travail adoptée.

Nous supposons disposer de deux outils supplémentaires pour effectuer notre analyse :

- un outil nommé " **Symsuivant** " qui met le prochain symbole de la phrase dans " **SymLu** ",

- un outil " **Erreur** " de signalement d'erreur dès qu'une erreur d'analyse est détectée. L'outil " Erreur " arrête en même temps tout le processus de reconnaissance.

Voici une écriture algorithmique des différents blocs d'analyse associés aux éléments de V_N dans G_{F2}. Cette écriture permet en l'appliquant à une phrase de G_{F2}, de valider ou non son analyse (de la reconnaître). Notre stratégie est basée sur les **Init** de chaque symbole de V_N utilisés comme ensembles de prédiction sur l'unique " bon chemin " à prendre dans la suite des règles.

Bloc Analyser Phrase :

```
si SymLu ∈ Init(GN) alors
 Analyser GN ;
 si SymLu ∈ Init(GV) alors
  Analyser GV ;
  si SymLu ≠ '.' Alors Erreur fsi
 sinon Erreur
 fsi
sinon Erreur
fsi
```

Bloc Analyser GN :

```
si SymLu ∈ Init(Art) alors
 Analyser Art;
 si SymLu ∈ Init(LeNom) alors
  Analyser LeNom;
 sinon Erreur
 fsi
sinon Erreur
fsi
```

Bloc Analyser GV :

```
si SymLu ∈ Init(Verbe) alors
 Analyser Verbe;
 si SymLu ∈ Init(Suite) alors
  Analyser Suite;
 sinon Erreur
 fsi
sinon Erreur
fsi
```

Bloc Analyser Suite :

```
si SymLu ∈ Init(GN) alors
 Analyser GN;
sinon
 si SymLu ∈ Init(Adv) alors
  Analyser Adv;
  si SymLu ∈ Init(GN) alors
   Analyser GN;
  sinon Erreur
  fsi
 sinon Erreur
 fsi
fsi
```

Bloc Analyser LeNom :

```
si SymLu ∈ Init(Adj) alors
 Analyser Adj;
 si SymLu ∈ Init(Nom) alors
  Analyser Nom
 sinon Erreur
 fsi
sinon
 si SymLu ∈ Init(Nom) alors
  Analyser Nom;
  si SymLu ∈ Init(Adj) alors
   Analyser Adj;
  sinon Erreur
  fsi
 sinon Erreur
 fsi
fsi
```

Bloc Analyser Art :

```
si SymLu ∈ {le,un } alors
  Symsuivant
sinon Erreur
fsi
```

Bloc Analyser Nom :

```
si SymLu ∈ {chat,chien } alors
 Symsuivant
sinon Erreur
fsi
```

Bloc Analyser Verbe :

```
si SymLu ∈ {aime, poursuit }
  alors   Symsuivant
sinon Erreur
fsi
```

Bloc Analyser Adj :

```
si SymLu ∈ {beau, blanc,
        noir, gentil } alors
   Symsuivant
sinon Erreur
fsi
```

Bloc Analyser Adv :

```
si SymLu ∈ {malicieusement,
joyeusement } alors
   Symsuivant
sinon Erreur
fsi
```

3.3 Procédures Pascal-Delphi associées aux blocs dans G_{F2}

Afin de clore cette ouverture sur la reconnaissance, nous traduisons en pascal les spécifications précédentes sur les blocs d'analyse. A chaque bloc d'analyse nous faisons correspondre une procédure pascal-Delphi en nous inspirant des choix effectués lors de l'écriture d'un générateur de phrase du mini-français.

Les structures de données :
Nous disposons d'un type **liste** obtenu à partir d'une unité de liste linéaire de chaînes.
Le type **liste** sert à stocker des éléments de V_T qui sont tous des **string**.

```
var
tnom,tadjectif,tarticle,tverbe,tadverbe:liste;
{toutes ces listes contiennent les symboles de VT }

Init_Grp_Nom, Init_LeNom, Init_Grp_Verbal,Init_Suite:liste;
{toutes ces listes contiennent les symboles des Init}
Symlu:string; {le symbole lu}
PhraseLue,Copiephrase:string; {la phrase à analyser et sa copie}

Les outils de base :
procedure Symsuiv(var Sym:string);
{l'outil Symsuivant, le paramètre Sym
contient le symbole extrait}
begin
 if Copiephrase<>'' then
  if Copiephrase='.' then Sym:='.'
  else
  begin
   Sym:=copy(Copiephrase,1,pos(' ',Copiephrase)-1);
   delete(Copiephrase,pos(Sym,Copiephrase),length(sym)+1)
  end;
end;

procedure Erreur; {outil Erreur}
begin
 writeln('Erreur');
 readln ;
 halt
end;

function AppartientA(Sym:string;Ensemble:liste):boolean;
{teste l'appartenance à un Init donné du symbole Sym }
```

```
begin
 AppartientA:=False;
 if Test (Ensemble,Sym) then AppartientA:=True
end;
```

Traduction de chacun des blocs d'analyse

```
procedure Nom; {Bloc Analyser Nom}
begin
 if AppartientA(Symlu,tnom) then
  symsuiv(Symlu);
end;
```

```
procedure Adjectif; {Bloc Analyser Adj}
begin
 if AppartientA(Symlu,tadjectif) then
  symsuiv(Symlu);
end;
```

```
procedure Adverbe; {Bloc Analyser Adv}
begin
 if AppartientA(Symlu,tadverbe) then
  symsuiv(Symlu);
end;
```

```
procedure Verbe; {Bloc Analyser Verbe}
begin
 if AppartientA(Symlu,tverbe) then
  symsuiv(Symlu);
end;
```

```
procedure LeNom; {Bloc Analyser LeNom }
begin
 if AppartientA(Symlu,tadjectif) then
 begin
  Adjectif;
  if AppartientA(Symlu,tnom) then
   Nom
  else erreur
 end
 else
 if AppartientA(Symlu,tnom) then
 begin
  Nom;
  if AppartientA(Symlu,tadjectif) then
   Adjectif
  else erreur
 end
 else erreur
end;
```

```
procedure Grp_Nom; {Bloc Analyser GN}
begin
 if AppartientA(Symlu,tarticle) then
 begin
  Article;
  if AppartientA(Symlu, Init_LeNom) then
```

```
   LeNom
  else erreur
 end
 else erreur;
end;
```

```
procedure Suite; {Bloc Analyser Suite}
begin
 if AppartientA(Symlu,Init_Grp_Nom) then
  Grp_Nom
 else
 if AppartientA(Symlu,tadverbe) then
 begin
  Adverbe;
  if AppartientA(Symlu,Init_Grp_Nom) then
   Grp_Nom
  else erreur
 end
 else erreur
end;
```

```
procedure Grp_Verbal; { Bloc Analyser GV }
begin
 if AppartientA(Symlu,tverbe) then
 begin
  Verbe;
  if AppartientA(Symlu,Init_Suite) then
   Suite
  else erreur
 end
 else erreur
end;
```

```
procedure phrase; {Bloc Analyser Phrase }
begin
 if AppartientA(Symlu,Init_Grp_Nom) then
 begin
  Grp_Nom;
  if AppartientA(Symlu,Init_Grp_Verbal) then
  begin
   Grp_Verbal;
   if Symlu <>'.' then erreur
   else writeln('Phrase reconnue !')
  end
  else erreur
 end
 else erreur;
end;
```

Chapitre 7.3 interaction et pilotage

interface en mini français

Plan du chapitre:

1. Un peu plus loin avec l'interaction et le pilotage

Nous allons nous servir dans ce paragraphe, de ce que nous connaissons sur la programmation par les grammaires pour piloter les actions de dialogue avec l'utilisateur.

1.1 Reconnaissance syntaxique du dialogue

Nous supposons que le dialogue peut être spécifié par une grammaire (ceci est d'autant plus vrai que c'est le programmeur qui décide du genre de dialogue à instaurer dans son interface, il lui est donc loisible de définir une " bonne " grammaire pour le dialogue entre son logiciel et l'utilisateur).

Nous recensons deux catégories de dialogue :

> Soit l'on guide pas à pas l'utilisateur dans la saisie et il ne peut entrer que de l'information syntaxiquement correcte. Nous dénommerons cette catégorie sous le vocable " ***saisie dirigée par la syntaxe*** ".
>
> Soit l'utilisateur est libre de saisir de l'information et l'on lance une vérification de la syntaxe dès la fin de la saisie (ce qui se passe lorsque vous utilisez un compilateur qui vérifie votre programme source après que vous l'avez tapé). Cette deuxième façon de faire nécessite la construction d'un ***analyseur syntaxique*** (plus ou moins complexe selon que la grammaire est de type 2 ou 3 et selon sa classe d'analyse).

Le deuxième mode de guidage est classique et ressort de ce qui a été vu dans les chapitres précédents. Nous nous intéressons plutôt au premier mode de saisie qui est en fait plus aisé à programmer.

1.2 Saisie dirigée par la syntaxe : principe

Ce genre de saisie est le plus simple à mettre en œuvre en initiation, et donne des résultats immédiats ; en particulier, il se prête bien au prototypage avec un RAD visuel. Le programmeur peut élaborer un prototype (squelette d'IHM) de son interface, le faire fonctionner et le tester d'une manière autonome sans avoir besoin d'écrire le code interne complet. La partie visuelle et événementielle du RAD permet de construire, de tester et de modifier rapidement l'interface.

Le principe général de conception est le suivant :

L'interface présente à l'utilisateur et pour chaque nouveau plan d'action, tous les choix admissibles pour le passage au plan d'action suivant. Ceci se traduira pour la saisie d'un texte par la présentation à l'utilisateur des symboles terminaux actuellement possibles pour construire son dialogue et au masquage de tous les autres. Nous introduisons ici la notion de **plan d'action** matérialisé par un ensemble d'états des objets de l'interface combiné avec des actions sur ces objets.

Nous nous servons d'une grammaire décrite par ses diagrammes syntaxiques pour spécifier les étapes de passage d'un plan d'action à l'autre (la grammaire peut être de type 3 ou de type 2 mais LL(1)).

1.3 Utilisation du TAD grammaire graphique pour la saisie

Nous rappelons le **TAD** générique ***Diag*** de grammaire graphique déjà défini pour spécifier de façon abstraite et graphique les règles d'une C-grammaire.

TAD : Diag
utilise : **VT**, **VN** *// les vocabulaires de la grammaire*
opérations :
t_1 : → Diag
t_2 : VT ∪ VN → Diag
t_3 : VN → Diag
t_4 : VT ∪ VN → Diag
t_5 : (VT ∪ VN)**n** → Diag
t_6 : (VT ∪ VN)**2** → Diag
t_7 : (VT ∪ VN)**2** → Diag
• : Diag x Diag → Diag

Axiomes :
la loi • est associative *(concaténation de diagrammes)*
$\forall t_i \in$ Diag (i ≥1) / $t_1 \bullet t_i = t_i \bullet t_1$ (t_1 *élément neutre)*
t_i(A) • t_k(B) = t_i(A)t_k(B) *(méthode de construction des diagrammes)*

Nous proposons au lecteur une spécification opérationnelle (plus concrète) de chaque diagramme de règle du **TAD *Diag*** à l'aide de diagrammes événementiels (les conventions utilisées sont celles qui ont été indiquées lors de la définition du graphe événementiel). Cette spécification a pour but de fournir un outil méthodique de construction d'une série de plans d'action (activation-désactivation) afin d'implanter une saisie dirigée par la syntaxe.

Les opérateurs t_k représentent pour notre interface des **plans d'action élémentaires**. Un plan d'action général est constitué d'une **combinaison de plans d'action élémentaires**.

Comme notre souci est de rester pratique, chaque diagramme événementiel associé à un opérateur $\mathbf{t_i}$, sera traduit par un exemple en Delphi.

> Nous avons choisi d'implanter les objets événementiels par des boutons de la classe des TButton.
>
> L'action d'activation ou de désactivation est implantée par la variation de la propriété booléenne de masquage " enabled " de l'objet TButton.

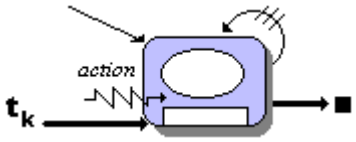
Remarque :

Nous aurions pu aussi utiliser une autre propriété booléenne de masquage des TButton, la propriété

" visible ".

Notes d'implantation en Delphi

A partir d'un diagramme événementiel général comme ci-dessous :



où :

t_k ⟶ = entrée dans le plan d'action t_k

⟶■ = passer au plan d'action suivant

Nous implanterons en Delphi:

action "↯" = " événement clic du bouton"
activation "⟶" = " NomduBouton.enabled := true "
désactivation "⟶" = " NomduBouton.enabled :=false "
gestionnaire d'action = gestionnaire d'événement Clic du bouton.

Avec comme apparence visuelle pour l'activation/désactivation :

Propriété enabled :=true (activable)

Propriété enabled :=false(inactivable)

L'exécution de plans d'action de saisie correspond dans ce cas essentiellement à masquer / démasquer des boutons utilisables.

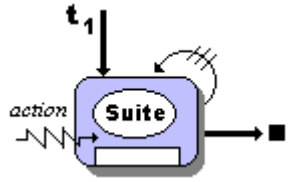
2. Une méthode pratique de construction

Nous proposons une méthode de construction de plans d'action en suivant **la syntaxe d'une grammaire LL(1)**. Nous associons des diagrammes événementiels et des schémas de plan d'action algorithmique aux diagrammes syntaxiques de la grammaire. L'implantation des plans d'action sera exécutée en Delphi.

2.1 Tableau de traduction de Vt en diagrammes événementiels

Cette traduction est valide pour des éléments A $\in V_T$et B $\in V_T$. L'élément " Suite " (associé à la règle vide) représente un événement permettant de passer d'un plan d'action au suivant sans avoir de choix terminal à proposer à l'utilisateur.

Opérateur t_1 :

Diagramme de règle	*Diagramme événementiel associé*
$t_1 = \xrightarrow{A:}$	

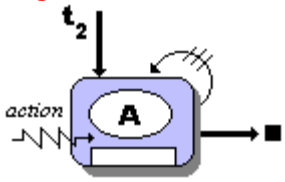
Implantation en Delphi du diagramme t_1 :

Au moment où le plan d'action t_1 est exécuté, le bouton Suite est activé.
Une action clic appelle le gestionnaire de l'événement clic du bouton suite.

Le code du gestionnaire du bouton **suite** est :

```
Suite.enabled :=false ; {le bouton se désactive}
AfficherPlanSuivant ; {exécution du plan suivant}
```

Opérateur t_2 :

Diagramme de règle	*Diagramme événementiel associé*
$t_2(A) = \xrightarrow{B:} \text{—A—}\rightarrow$	

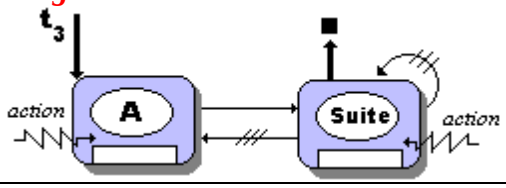
Implantation en Delphi du diagramme t_2 :

Au moment où le plan d'action t_2 est exécuté, le bouton A est activé.
Une action clic appelle le gestionnaire de l'événement clic du bouton A.

Le code du gestionnaire du bouton A est :

```
A.enabled :=false ; {le bouton se désactive}
Saisie(A) ; {saisie de l'information}
AfficherPlanSuivant ; {exécution du plan suivant}
```

Opérateur t_3 :

Diagramme de règle	*Diagramme événementiel associé*
$t_3(A) = \xrightarrow{B:} \text{—A—}\rightarrow$	

Implantation en Delphi du diagramme t_3 :

Au moment où le plan d'action t_3 est exécuté, le bouton A est activé, le bouton Suite n'est pas activé.
Une action clic sur le bouton **A** appelle le gestionnaire de l'événement clic du bouton **A**.

Le code du gestionnaire du bouton **suite** est :

```
Suite.enabled :=false ; {le bouton se désactive}
A.enabled :=false ; {bouton A désactivé}
AfficherPlanSuivant ; {exécution du plan suivant}
```

Le code du gestionnaire du bouton **A** est :

```
Suite.enabled :=true ; {bouton suite activé}
Saisie(A) ; {saisie de l'information}
```

Opérateur t_4 :

Diagramme de règle

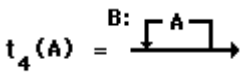
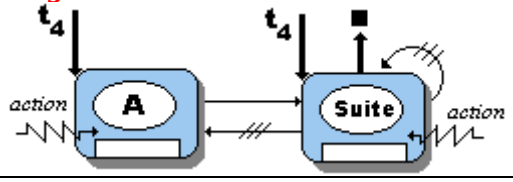


Diagramme événementiel associé



Implantation en Delphi du diagramme t_4 :

Au moment où le plan d'action t_4 est exécuté, le bouton A est activé, le bouton Suite est activé aussi.

Une action clic sur le bouton **suite** appelle le gestionnaire de l'événement clic du bouton suite. Une action clic sur le bouton **A** appelle le gestionnaire de l'événement clic du bouton **A**.

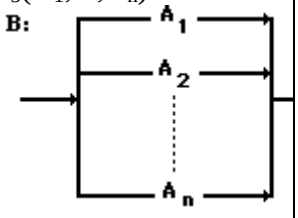
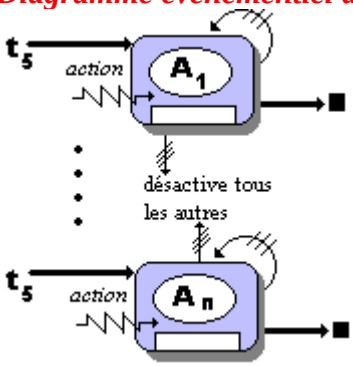
Le code du gestionnaire du bouton **suite** est :

```
Suite.enabled :=false ; {le bouton se désactive}
A.enabled :=false ; {bouton A désactivé}
AfficherPlanSuivant ; {exécution du plan suivant}
```

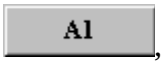
Le code du gestionnaire du bouton **A** est :

```
Suite.enabled :=true ; {bouton suite activé}
Saisie(A) ; {saisie de l'information}
```

Opérateur t_5 :

Diagramme de règle	*Diagramme événementiel associé*
$t_5(A_1,...,A_n)$ = 	

Implantation en Delphi du diagramme t_5 :

Au moment où le plan d'action t_5 est exécuté, les boutons  ,...,  sont activés.

Une action clic sur un des boutons $\mathbf{A_k}$ appelle le gestionnaire de l'événement clic du bouton $\mathbf{A_k}$.

Le code du gestionnaire de chaque bouton $\mathbf{A_k}$ est :

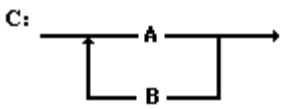
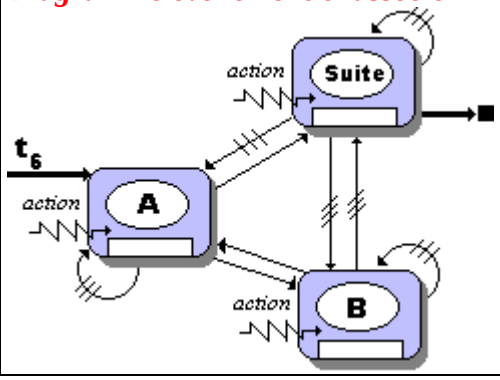
DésactiverTous; {*tous les boutons sont désactivés*} 

...etc... 

Saisie(Ak) ; {*saisie de l'information de A_k*}

AfficherPlanSuivant ; {*exécution du plan suivant*}

Opérateur t_6 :

Diagramme de règle	*Diagramme événementiel associé*
$t_6(A,B)$= 	

Implantation en Delphi du diagramme t_6 :

Au moment où le plan d'action t_6 est exécuté, le bouton 

est activé, les boutons 

et 

ne sont pas activés.
Une action clic sur le bouton **A** appelle le gestionnaire de l'événement clic du bouton **A** qui agit sur **suite** et **B**. Une action clic sur le bouton **B** appelle le gestionnaire de l'événement clic du bouton **B** qui agit sur **A** et **suite**. Une action clic sur le bouton **suite** appelle le gestionnaire de l'événement clic du bouton **suite** qui agit sur **A** et **B**.

Le code du gestionnaire du bouton **suite** est :

Suite.enabled :=false ; {*le bouton se désactive*} 

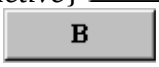
A.enabled :=false ; {*bouton A désactivé*} 

B.enabled :=false ; {*bouton B désactivé*} 

AfficherPlanSuivant ; {*exécution du plan suivant*}

Le code du gestionnaire du bouton **A** est :

Suite.enabled :=true ; {*bouton suite activé*} 

B.enabled :=true ; {*bouton B activé*} 

A.enabled :=false ; {*bouton A désactivé*} 

Saisie(A) ; {*saisie de l'information de A*}

Le code du gestionnaire du bouton **B** est :

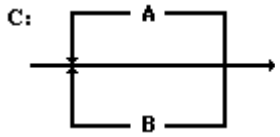
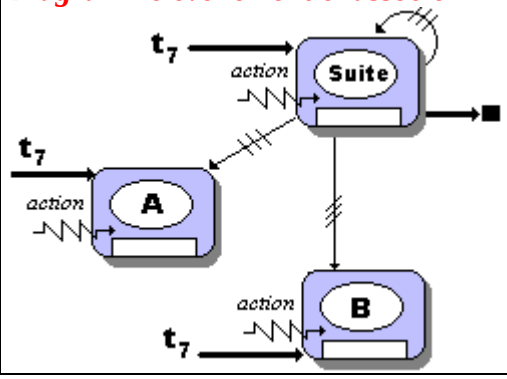
Suite.enabled :=false ; {*bouton suite activé*} 

B.enabled :=false ; {*bouton B activé*} 

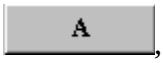
A.enabled :=true ; {*bouton A désactivé*} 

Saisie(B) ; {*saisie de l'information de B*}

Opérateur t_7 :

Diagramme de règle	***Diagramme événementiel associé***
$t_7(A,B)=$ 	

Implantation en Delphi du diagramme t_7 :

Au moment où le plan d'action t_7 est exécuté, les boutons 

, 

et 

sont activés.

Une action clic sur le bouton **A** appelle le gestionnaire de l'événement clic du bouton **A**. Une action clic sur le bouton **B** appelle le gestionnaire de l'événement clic du bouton **B**. Une action clic sur le bouton **suite** appelle le gestionnaire de l'événement clic du bouton **suite** qui agit sur **A** et **B**.

Le code du gestionnaire du bouton **suite** est :

```
Suite.enabled :=false ; {le bouton se désactive}
A.enabled :=false ; {bouton A désactivé}
B.enabled :=false ; {bouton B désactivé}
AfficherPlanSuivant ; {exécution du plan suivant}
```

Le code du gestionnaire du bouton **A** est :

```
Saisie(A) ; {saisie de l'information de A}
```

Le code du gestionnaire du bouton **B** est :

```
Saisie(B) ; {saisie de l'information de B}
```

2.2 Tableau de traduction de V_N en schémas LDFA

Nous nous préoccupons maintenant des plans associés à des éléments du vocabulaire non terminal V_N de notre grammaire supposée être LL(1).

Le passage d'un plan d'action à un autre est conditionné par les symboles de V_t qui doivent être présentés à l'utilisateur dans le nouveau plan d'action. Or nous savons que l'ensemble Init(A) de l'élément A de V_N d'une grammaire nous fournit tous les symboles possibles d'un plan d'action associé à l'élément A. Puis nous ferons fonctionner notre grammaire en mode générateur. Au lieu d'engendrer aléatoirement des phrases du langage nous engendrons des phrases correctes où les choix des éléments terminaux ont été effectués par l'utilisateur. En partant de cette remarque, nous pouvons construire des schémas généraux écrits en langage d'algorithme (LDFA) décrivant le fonctionnement d'un plan d'action associé à un élément A, $A \in V_N$.

Le tableau avec ses objets et outils de base :

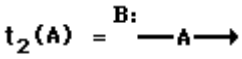
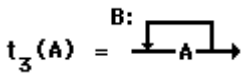
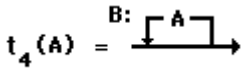
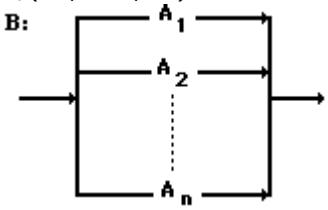
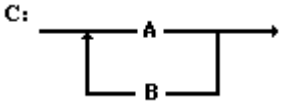
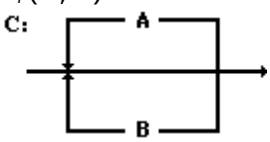
Nous supposons disposer d'un moyen d'activer (présenter à l'utilisateur) tous les symboles de Init(A) afin de ne lui laisser que ces choix possibles : **Activer(Init(A)).**

Nous supposons disposer d'un moyen de désactiver tous les symboles de Init(A) dès que l'utilisateur a fait son choix : **Désactiver(Init(A)).**

Le symbole **suite** a la même signification qu'au paragraphe précédent, il sert à passer au plan d'action suivant.

Le symbole **action** indique quelle est l'action que vient d'effectuer l'utilisateur sur l'interface.

Nous en déduisons le tableau de traduction de V_N en schéma Algorithmique LDFA :

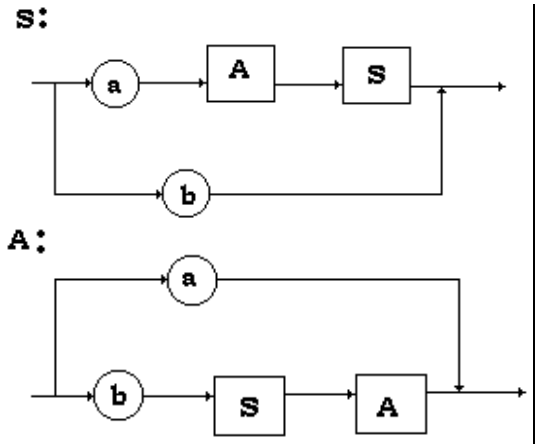
Diagramme de règle	Schéma LDFA associé
Opérateur t_2 	Activer (Init(A)) Plan A ; Désactiver (Init(A))
Opérateur t_3 	Activer (Init(A)) **Répéter** Plan A ; Activer (Init(Suite)) **jusquà** action = Suite ; Désactiver (Init(A))
Opérateur t_4 	Activer (Init(A)) ; Activer (Suite) ; **Tantque** action ≠ Suite **faire** Plan A ; Activer (Suite) **Ftant** ; Désactiver (Init(A))
Opérateur t_5 $t_5(A_1,...,A_n)$ = 	Activer (Init(A_1)); Activer (Init(A_2)) ; .... **si** action ∈ Init(A_1) **alors** Plan A_1 **sinon si** action ∈ Init(A_2) **alors** Plan A_2 **sinon si** .... **fsi** ; Désactiver (Init(A_1)) ; ....
Opérateur t_6 $t_6(A,B)$= 	**répéter** Activer (Init(A)) ; Plan A ; Activer (Init(B)) ; Activer (Init(Suite)) ; **si** action ∈ Init(B) **alors** Plan B **fsi** **jusquà** action = Suite
Opérateur t_7 $t_7(A,B)$= 	Activer (Init(A)) ; Activer (Init(B)) ; Activer (Suite) ; **Tantque** action ∈ Init(B) ∪ Init(A) **faire** **si** action ∈ Init(B) **alors** Plan B **sinon** Plan A **fsi** ; Activer (Init(A)) ; Activer (Init(B)) ; Activer (Suite) ; **Ftant** ;

Application à deux exemples

Exemple-1

Soit la grammaire G déjà vue lors de l'étude des First et des Follow :

G:
VT = {a, b}
VN = {A, S}
axiome: S



Règles :

1 : S → aAS

2 : S → b

3 : A → a

4 : A → bSA

Cette grammaire est LL(1) car nous avons déjà calculé les Init de A et de S :

InitA) = Init(S) = {a,b}

En nous servant du tableau de traduction précédent écrivons les plans d'action associés aux deux éléments de V_N soient A et S.

Plan A :	**Plan S :**
Activer(a) ; **Activer(b) ;** **Saisie(SymLu) ;** **si action = a alors** **désactiver(a)** **sinon** **désactiver(b);** **Plan S;** **Plan A** **fsi** **désactiver(a) ;** **désactiver(b) ;**	**Activer(a) ;** **Activer(b) ;** **Saisie(SymLu) ;** **si action = a alors** **désactiver(a) ;** **Plan A ;** **Plan S** **sinon** **désactiver(b);** **fsi ;** **désactiver(a) ;** **désactiver(b) ;**

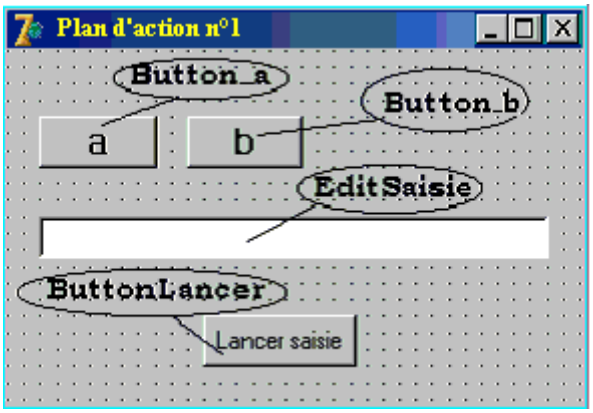
Implantation avec Delphi

- Nous choisissons d'utiliser deux TButton pour les éléments terminaux " a "(Button_a) et " b "(Button_b), la saisie a lieu dans un TEdit (EditSaisie).

- Les actions activer-désactiver sont implantées par la propriété enabled des Tbutton :

procedure Activer(Elt:TButton); **begin** Elt.enabled:=true; **end**;	**procedure** Desactiver(Elt:TButton); **begin** Elt.enabled:=false; **end**;

L'interface retenue est la suivante :



- Nous n'utilisons pas l'élément **suite**.
- Le symbole **action** est implanté par un drapeau booléen " ActionFaite " indiquant si l'utilisateur a effectué une action oui ou non.
- Nous avons conservé le mode séquentiel avec interruption afin de pouvoir bénéficier de la méthode de programmation par syntaxe en mode génération. La saisie du caractère dans **SymLu** est donc programmée selon une boucle d'attente infinie qui ne se libère que lorsque l'utilisateur a clické sur l'un des choix possibles.

```
procedure Attendre;
begin
  if not ActionFaite then
  begin
   repeat
      Application.ProcessMessages
   until ActionFaite ;
   ActionFaite:=false
  end
end{Attendre};

procedure saisie(ch:string);
begin
 if ActionFaite=false then
  Form1.Edit_saisie.text:= Form1.Edit_saisie.text+' '+ch
 else ActionFaite:=false
end{saisie};

procedure AttenteSaisie;
begin
 Attendre;
 saisie(SymLu);
end{AttenteSaisie};
```

Chacun des plans est implanté selon une procédure:

```
procedure Plan_A;
begin
 with Form1 do
 begin
  Activer(Button_a);
  Activer(Button_b);
  AttenteSaisie;
  if  SymLu='a' then
   Desactiver(Button_a)
  else
  begin
    Desactiver(Button_b);
    Plan_S;
    Plan_A
  end;
  Desactiver(Button_a);
  Desactiver(Button_b)
 end
end;
```

```
procedure Plan_S;
begin
 with Form1 do
 begin
  Activer(Button_a);
  Activer(Button_b);
  AttenteSaisie;
  if  SymLu='a' then
  begin
   Desactiver(Button_a);
   Plan_A;
   Plan_S
  end
  else
     Desactiver(Button_a);
  Desactiver(Button_b)
end
end;
```

Lorsque l'utilisateur sélectionne un bouton par un clic, le symbole SymLu contient la valeur du choix effectué.

```
procedure TForm1.Button_aetbClic(Sender: TObject);
begin
 ActionFaite:=true;
 Symlu:=TButton(Sender).caption
end;
```

Le bouton lancer la saisie appelle le plan associé à l'axiome S.

```
procedure TForm1.ButtonLancerClic(Sender: TObject);
begin
 RAZTout;
 ButtonLancer.enabled:=false;
 Plan_S;
 PlanSuivant;
end;
```

Il appelle aussi à la fin la procédure PlanSuivant qui sert à passer au plan d'action suivant (dans l'exemple le plan suivant est l'état initial, il suffit de réactiver le bouton lancer).

```
procedure PlanSuivant;
begin
 Form1.ButtonLancer.enabled:=true;
end;
```

Exemple-2 *Interface de saisie du mini-français*

La grammaire choisie est la grammaire LL(1) GF2 du mini-français déjà étudiée.

$G_{F2} = (V_N, V_T, S, R)$
$\mathbf{V_T}$ ={le, un, chat, chien, aime, poursuit, malicieusement, joyeusement, gentil, noir, blanc, beau, '.'}
$\mathbf{V_N}$ = { ⟨ phrase⟩, ⟨ GN ⟩, ⟨GV⟩, ⟨ Art⟩, ⟨ Nom ⟩, ⟨ Adj⟩, ⟨ Adv ⟩, ⟨verbe⟩, ⟨ LeNom ⟩, ⟨ Suite ⟩ }

Axiome : <phrase>
Règles :
1 : ⟨ phrase ⟩→⟨ GN⟩⟨ GV ⟩.
2 : ⟨ GN ⟩→ ⟨ Art ⟩⟨ LeNom⟩
3 : ⟨ LeNom⟩ →⟨ Adj ⟩⟨ Nom ⟩ | ⟨ Nom ⟩⟨ Adj ⟩
4 : ⟨ GV ⟩→ ⟨ verbe⟩⟨ Suite⟩
5 : ⟨ Suite⟩→ ⟨ GN⟩ | ⟨ Adv ⟩⟨ GN⟩
6 : ⟨ Art ⟩→ le | un
7 : ⟨ Nom ⟩→ chien | chat
8 : ⟨ verbe⟩→ aime | poursuit
9 : ⟨ Adj ⟩→ blanc | noir | gentil | beau
10: ⟨ Adv ⟩→ malicieusement | joyeusement

Nous laissons le lecteur écrire les diagrammes syntaxiques obtenus à partir des règles précédentes.

Afin que le lecteur puisse bien se pénétrer de la similitude des démarches entre les blocs d'analyse avec la saisie par plan d'action, nous lui livrons ci-dessous deux plans et les blocs correspondants, il continuera à écrire de la même façon ceux des autres éléments de V_N.

Premier plan d'action celui de l'axiome < phrase>

Bloc Analyser Phrase : **si SymLu** ∈ Init(GN) **alors** Analyser GN ; **si** SymLu ∈ Init(GV) **alors** Analyser GV ; **si** SymLu ≠ '.'**alors** Erreur **fsi** **sinon** Erreur **fsi** **sinon** Erreur **fsi**	**Plan phrase** : **Activer(Init(Phrase));** **Plan GN ;** **Plan GV ;** **Plan point** **Désactiver(Init(Phrase)) ;**

L'implantation du plan d'action Phrase en Delphi est immédiate :

```
procedure Plan_phrase;
begin
  Plan_GN;
  Plan_GV;
  Plan_point;
end;
```

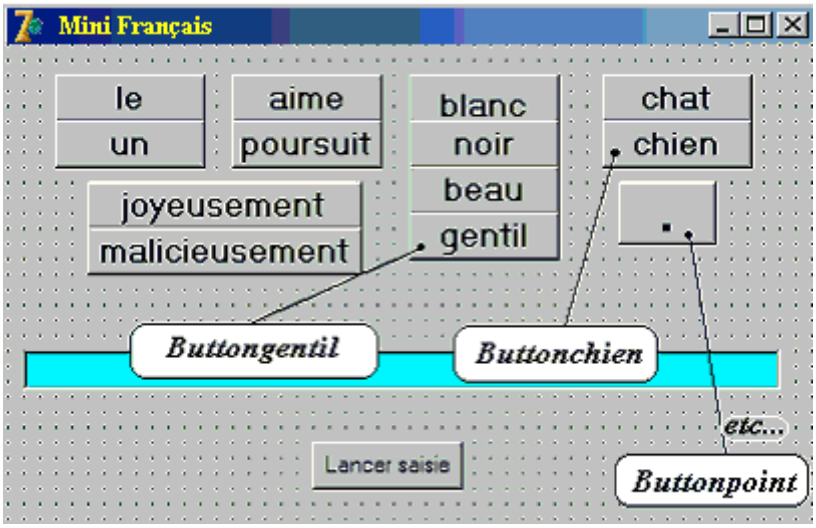
Deuxième plan d'action celui du symbole < LeNom >

Bloc Analyser LeNom : **si SymLu** ∈ Init(Adj) **alors** Analyser Adj; **si** SymLu ∈ Init(Nom) **alors** Analyser Nom **sinon** Erreur **fsi** **sinon** **si** SymLu ∈ Init(Nom) **alors** Analyser Nom; **si** SymLu ∈ Init(Adj) **alors** Analyser Adj; **sinon** Erreur **fsi** **sinon** Erreur **fsi** **fsi**	**Plan LeNom** **Activer(Init(LeNom));** **Saisie;** **Désactiver(Init(LeNom));** **si action ∈Init(Nom) alors** **Plan Nom ;** **Plan Adj** **sinon** **Plan Adj;** **Plan Nom** **fsi**

L'implantation du plan d'action LeNom en Delphi est immédiate :

```
procedure Plan_LeNom;
begin
  …. etc
end;
```

Code Delphi7 complet de l'exemple-2 *Interface de saisie du mini-français*




```
unit UFplanGF2;

 interface

 uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Buttons;
```

```
type
 TFPPlanGF2 = class(TForm)
 ButtonLe: TButton;
 ButtonUn: TButton;
 Buttonblanc: TButton;
 Buttonnoir: TButton;
 Buttonbeau: TButton;
 Buttongentil: TButton;
 Buttonaime: TButton;
 Buttonpoursuit: TButton;
 Buttonchat: TButton;
 Buttonmalicieux: TButton;
 Buttonjoyeus: TButton;
 ButtonLancer: TButton;
 Edit_saisie: TEdit;
 Buttonchien: TButton;
 Buttonpoint: TButton;
 BitBtnFermer: TBitBtn;
 procedure ButtonLancerClick(Sender: TObject);
 procedure ButtonsClick(Sender: TObject);
 procedure FormCreate(Sender: TObject);
 procedure BitBtnFermerClick(Sender: TObject);
private
{ Déclarations privées }
public
{ Déclarations publiques }
 SymLu:string;
 ActionFaite , stop : boolean;
 procedure ActiverDesactiver(Elt:TButton; const etat:boolean);
 procedure InitGN(active:boolean);
 procedure InitLeNom(active:boolean);
 procedure InitGV(active:boolean);
 procedure InitSuite(active:boolean);
 procedure InitArt(active:boolean);
 procedure InitNom(active:boolean);
 procedure InitVerbe(active:boolean);
 procedure InitAdj(active:boolean);
 procedure InitAdv(active:boolean);
 procedure RAZTout;
 procedure AttenteSaisie;
 procedure PlanSuivant;
 procedure Plan_Art;
 procedure Plan_Nom;
 procedure Plan_Verbe;
 procedure Plan_Adj;
 procedure Plan_Adv;
 procedure Plan_LeNom;
 procedure Plan_GN;
 procedure Plan_Suite;
 procedure Plan_GV;
 procedure Plan_point;
 procedure Plan_phrase;
end;

 var
  FPPlanGF2: TFPPlanGF2;

implementation
```

```
{$R *.dfm }

procedure TFPPlanGF2.ActiverDesactiver(Elt:TButton;const etat:boolean);
begin
 Elt.enabled:=etat;
end;
////////////////////  LES INIT ////////////////////////////
procedure TFPPlanGF2.InitGN(active:boolean);
begin
 ActiverDesactiver(FPPlanGF2.ButtonLe,active);
 ActiverDesactiver(FPPlanGF2.ButtonUn,active)
end;

procedure TFPPlanGF2.InitLeNom(active:boolean);
begin
 ActiverDesactiver(FPPlanGF2.Buttonblanc,active);
 ActiverDesactiver(FPPlanGF2.Buttonnoir,active);
 ActiverDesactiver(FPPlanGF2.Buttonbeau,active);
 ActiverDesactiver(FPPlanGF2.Buttongentil,active);
 ActiverDesactiver(FPPlanGF2.Buttonchat,active);
 ActiverDesactiver(FPPlanGF2.Buttonchien,active)
end;

procedure TFPPlanGF2.InitGV(active:boolean);
begin
 ActiverDesactiver(FPPlanGF2.Buttonaime,active);
 ActiverDesactiver(FPPlanGF2.Buttonpoursuit,active)
end;

procedure TFPPlanGF2.InitSuite(active:boolean);
begin
 ActiverDesactiver(FPPlanGF2.ButtonLe,active);
 ActiverDesactiver(FPPlanGF2.ButtonUn,active);
 ActiverDesactiver(FPPlanGF2.Buttonjoyeus,active);
 ActiverDesactiver(FPPlanGF2.Buttonmalicieus,active)
end;

procedure InitArt(active:boolean);
begin
 InitGN(active)
end;

procedure TFPPlanGF2.InitNom(active:boolean);
begin
 ActiverDesactiver(FPPlanGF2.Buttonchat,active);
 ActiverDesactiver(FPPlanGF2.Buttonchien,active)
end;

procedure TFPPlanGF2.InitVerbe(active:boolean);
begin
 ActiverDesactiver(FPPlanGF2.Buttonaime,active);
 ActiverDesactiver(FPPlanGF2.Buttonpoursuit,active)
end;

procedure TFPPlanGF2.InitAdj(active:boolean);
begin
 ActiverDesactiver(FPPlanGF2.Buttonblanc,active);
 ActiverDesactiver(FPPlanGF2.Buttonnoir,active);
 ActiverDesactiver(FPPlanGF2.Buttonbeau,active);
 ActiverDesactiver(FPPlanGF2.Buttongentil,active)
```

```
end;

procedure TFPPlanGF2.InitAdv(active:boolean);
begin
 ActiverDesactiver(FPPlanGF2.Buttonjoyeus,active);
 ActiverDesactiver(FPPlanGF2.Buttonmalicieus,active)
end;
////////////////////// fin des INIT ////////////////////////
procedure TFPPlanGF2.RAZTout;
begin
 with FPPlanGF2 do
 begin
  InitArt(false);
  InitNom(false);
  InitVerbe(false);
  InitAdj(false);
  InitAdv(false);
  Buttonpoint.Enabled:=false;
  SymLu:='';
  ActionFaite:=false;
  Edit_saisie.text:='';
  stop:=false
 end
end;

procedure TFPPlanGF2.AttenteSaisie;
procedure Attendre;
begin
 if not ActionFaite then
 begin
  repeat
   Application.ProcessMessages
  until
  actionfaite ;
  ActionFaite:=false
 end
end{ Attendre};

procedure saisie(ch:string);
begin
 if ActionFaite=false then
  FPPlanGF2.Edit_saisie.text:= FPPlanGF2.Edit_saisie.text+' '+ch
 else
  actionfaite:=false
end{saisie};

begin
 Attendre;
 saisie(SymLu);
end{AttenteSaisie};

procedure TFPPlanGF2.PlanSuivant;
begin
 FPPlanGF2.ButtonLancer.enabled:=true;
end;
{---------- Les procédures des plans d'actions ----------}
procedure TFPPlanGF2.Plan_Art;
begin
 InitArt(true);
 AttenteSaisie;
```

```
 InitArt(false);
end;

procedure TFPPlanGF2.Plan_Nom;
begin
 InitNom(true);
 AttenteSaisie;
 InitNom(false);
end;

procedure TFPPlanGF2.Plan_Verbe;
begin
 InitVerbe(true);
 AttenteSaisie;
 InitVerbe(false);
end;

procedure TFPPlanGF2.Plan_Adj;
begin
 InitAdj(true);
 AttenteSaisie;
 InitAdj(false);
end;

procedure TFPPlanGF2.Plan_Adv;
begin
 InitAdv(true);
 AttenteSaisie;
 InitAdv(false);
end;

procedure TFPPlanGF2.Plan_LeNom;
begin
 InitLeNom(true);
 AttenteSaisie;
 InitLeNom(false);
 ActionFaite:=true; // action déjà saisie
 if (SymLu='chat') or (SymLu='chien') then
 begin
  Plan_Nom;
  Plan_Adj
 end
 else  // adjectif
 begin
  Plan_Adj;
  Plan_Nom
 end;
end;

procedure TFPPlanGF2.Plan_GN;
begin
 Plan_Art;
 Plan_LeNom;
end;

procedure TFPPlanGF2.Plan_Suite;
begin
 InitSuite(true);
 AttenteSaisie;
 InitSuite(false);
```

```
 ActionFaite:=true; // action déjà saisie
 if (SymLu='le') or (SymLu='un') then
 begin
  Plan_GN;
 end
 else  // adverbe
 begin
  Plan_Adv;
  Plan_GN
 end;
end;

procedure TFPPlanGF2.Plan_GV;
begin
 Plan_Verbe;
 Plan_Suite;
end;

procedure TFPPlanGF2.Plan_point;
begin
 FPPlanGF2.Buttonpoint.Enabled:=true;
 AttenteSaisie;
 FPPlanGF2.Buttonpoint.Enabled:=false;
end;

procedure TFPPlanGF2.Plan_phrase;
begin
 Plan_GN;
 Plan_GV;
 Plan_point;
end;

procedure TFPPlanGF2.ButtonLancerClick(Sender: TObject);
begin
 RAZTout;
 ButtonLancer.enabled:=false;
 Plan_phrase;
 PlanSuivant;
end;

procedure TFPPlanGF2.ButtonsClick(Sender: TObject);
begin
 ActionFaite:=true;
 if stop then
  halt;//l'utilisateur a demandé d'arrêter
 Symlu:=TButton(Sender).caption
end;

procedure TFPPlanGF2.FormCreate(Sender: TObject);
begin
 RAZTout;
end;

procedure TFPPlanGF2.BitBtnFermerClick(Sender: TObject);
begin
 stop:=true
end;

end.
```

Chapitre 7.4 Une projet d'IHM

Enquête fumeurs 

Plan du chapitre: 

1. Le projet de construction d'une borne interactive

Programmons un exemple en utilisant les principes d'élaboration d'une interface par plans d'action.
Nous sommes l'I.N.S., un institut d'enquêtes et de sondages au service de clients qui nous commandent des enquêtes de données chiffrées que nous exploiterons ultérieurement. Le travail ci-dessous peut être réalisé en environ 500 lignes de Delphi. Il peut être réalisé par un seul programmeur (temps de programmation 50h environ) ou avec une équipe de trois étudiants à qui l'on confiera des activités différentes.

1.1 Le marché avec le client

Le client, une organisation de lutte contre le tabagisme, nous a commandé une enquête sur le public des fumeurs de cigarettes dont l'âge est compris entre 10 ans et 120 ans. Il désire obtenir un fichier de données recueillies auprès du public des deux sexes. Pour chaque personne interrogée nous devons stocker dans notre fichier :

- le sexe,
- l'âge actuel,
- depuis combien d'années la personne fume,
- le nombre de cigarettes fumées par jour.

Le client est conscient qu'il est difficile de demander à un individu le nombre exact de cigarettes fumées par jour. Il admet que nous proposions aux personnes interrogées une série de plages de consommations plutôt qu'un nombre précis.

Le client veut en même temps faire œuvre de pédagogie auprès des individus fumeurs sondés. Il souhaite que le sondé puisse se situer dans une fourchette de pourcentages de consommation sur toute la population des personnes déjà sondées. D'autre part le client dispose de tables de mortalité dont il a extrait des formules permettant d'évaluer le risque de cancer du poumon ou du larynx en fonction du sexe, de l'âge, de la durée du tabagisme et du nombre de cigarettes fumées par jour. Le client espère ainsi faire prendre conscience du risque accru d'apparition d'un cancer en cas de tabagisme prolongé.

1.2 Aspect général d'un prototype

Notre équipe de développement a réfléchi au problème et a décidé que nous construirions un prototype de borne interactive :

- Nous allons mettre en place un **logiciel ouvert** qui fonctionnera 24h sur 24, qui attendra le client et lui permettra de remplir son questionnaire " à la volée ".
- Le logiciel doit être verrouillé contre toute fausse manipulation de la part de l'utilisateur. L'équipe de développement a décidé d'employer la méthode de saisie dirigée par la syntaxe par plans d'action.
- L'utilisateur pourra revenir, pour correction, sur l'une quelconque des données qu'il aura entrées. Avant son stockage définitif dans le fichier, chaque transaction est mémorisée sur disque dès qu'elle a lieu.

- Le principe d'une sauvegarde générale journalière effectuée par une personne de l'INS a été retenu. Les actions de maintenance éventuelles effectuées par du personnel de l'INS s'effectueront à partir d'un menu spécial protégé par un mot de passe.

Voici le prototype d'interface proposé dans sa totalité :

1.3 Partition de l'interface en zones d'action

Actuellement tous les plans d'action sont visibles. Nous proposons de diviser l'interface en plusieurs zones d'action différentes.

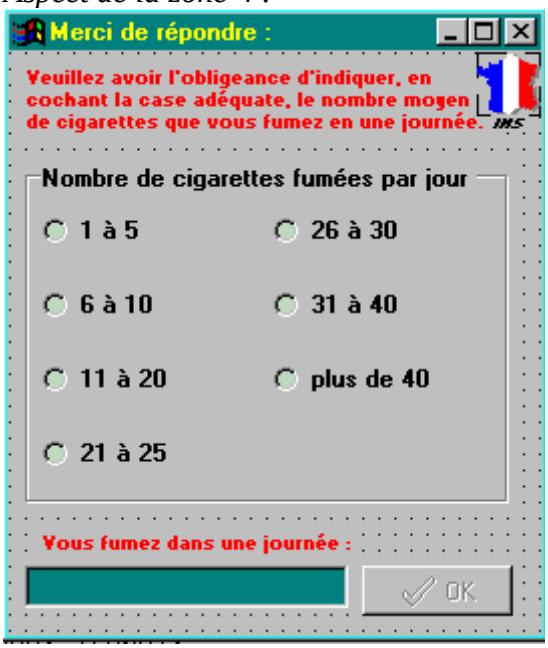
Cinq zones de saisies de l'information

- zone-1 d'indication du sexe du sondé,
- zone-2 de saisie de l'âge actuel,
- zone-3 de saisie de la durée du tabagisme,
- zone-4 de saisie du nombre de cigarettes par jour,
- zone-5 de correction des données entrées.

Aspect de la zone-1 :

Aspect de la zone-2 : *Aspect de la zone-3 :*

Aspect de la zone-4 :

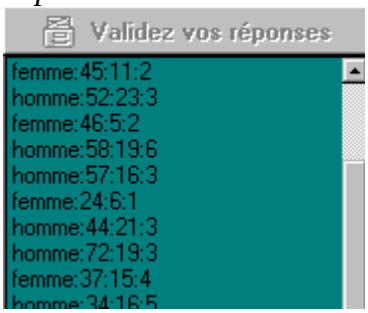


Aspect de la zone-5 :

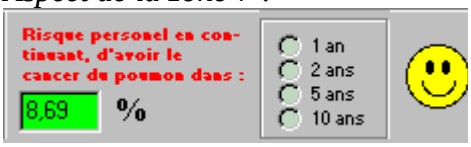
Deux zones de résultats consultables

- zone-6 de liste des données déjà entrées,
- zone-7 de consultation du risque de cancer

Aspect de la zone-6 :



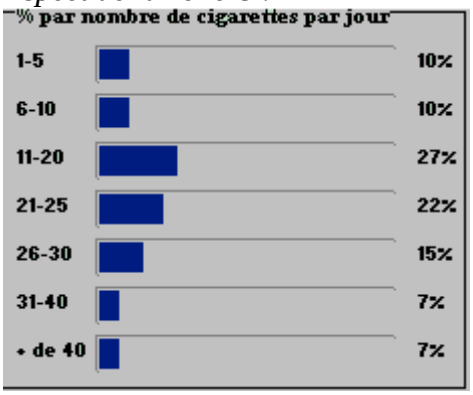
Aspect de la zone-7 :



Une zone d'affichage de résultats

zone-8 d'affichage des pourcentages de répartition

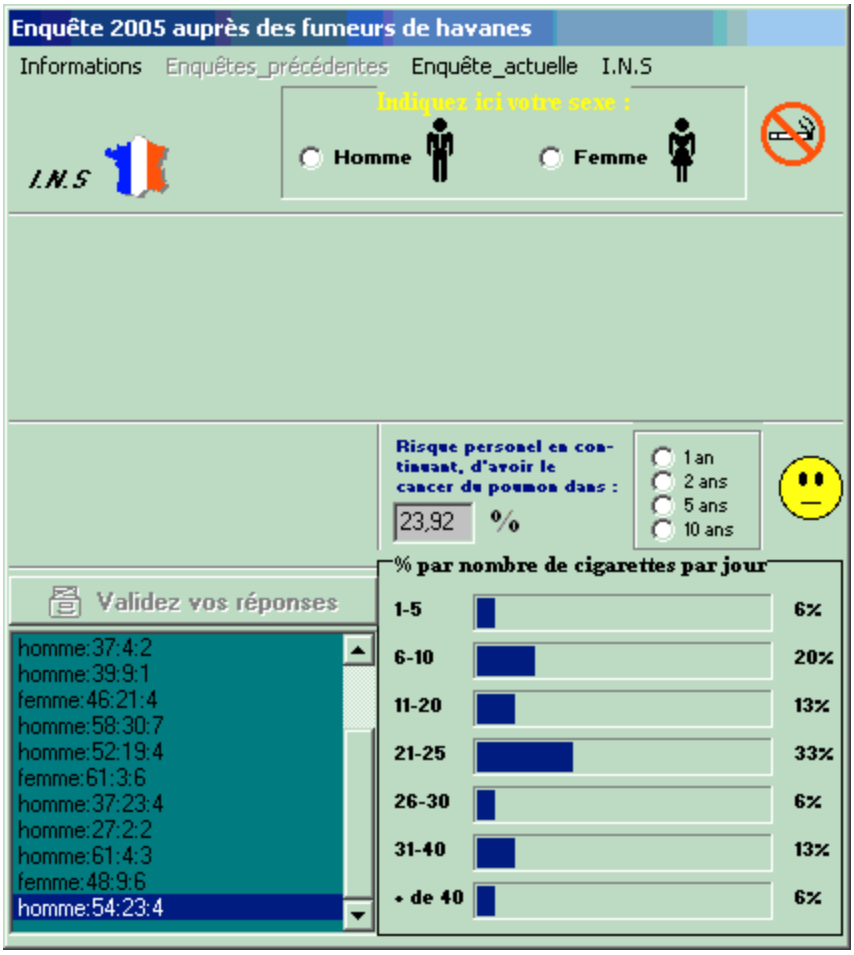
Aspect de la zone-8 :



1.4 Le mode attente utilisateur

Au lancement et après le passage de chaque sondé, l'interface est dans un état initial standard (écran d'accueil en mode attente). Elle peut prendre deux chemins d'action : soit le mode saisie (par séquencement : sondage), soit le mode consultation du fichier (visualisation de statistiques…).

Aspect visuel du mode de départ après passage de plusieurs sondés :



Remarque
un certain nombre de personnes ont déjà répondu à l'enquête. Les pourcentages apparaissent ainsi que la liste des réponses.
A ce stade, le sondé peut soit entrer ses données personnelles, et il entre dans le séquencement des plans d'action que nous allons voir ensuite, soit consulter les zones 6 et 7.

1.5 Le mode consultation

Le sondé peut clicker sur une donnée de la liste comme ci dessous. Les deux zones 5 et 7 sont immédiatement informées par les données sélectionnées dans la liste.

Graphe événementiel de la zone-6

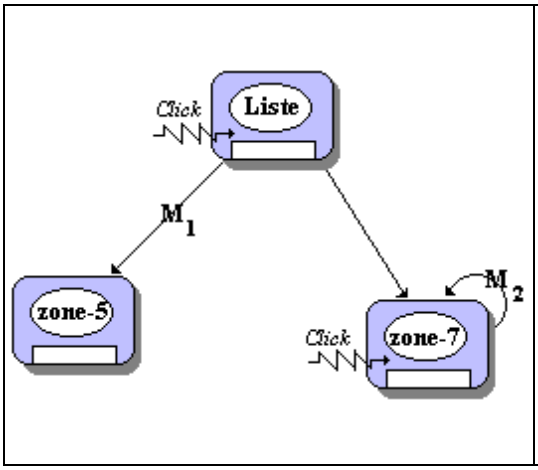
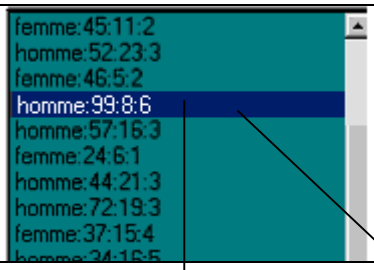


Table des états initiaux :

zone-5 désactivé
Liste activé
zone-7 activé
Homme désactivé
Femme désactivé
M1 = affichage dans zone désactivée (lecture seulement)
M2 = affichage des informations.

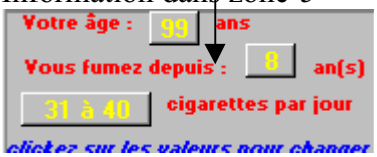
L'objet " liste " est implanté par l'objet visuel ListBoxReponses de la classe des TListBox.

Aspect visuel d'une consultation :

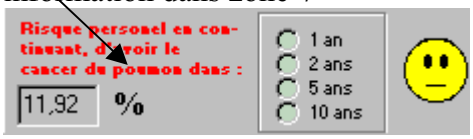


L'utilisateur sélectionne dans la liste une configuration (homme de 99 ans ayant fumé pendant 8 ans de 31 à 40 cigarettes par jour)

Information dans zone-5



information dans zone-7



L'utilisateur peut alors consulter les différents pourcentages de risques associés à cette configuration en sélectionnant dans la zone-7 le risque à un an, à deux ans etc... L'affichage se fait visuellement d'une façon chiffrée et d'une façon imagée.

Risque à 2 ans :

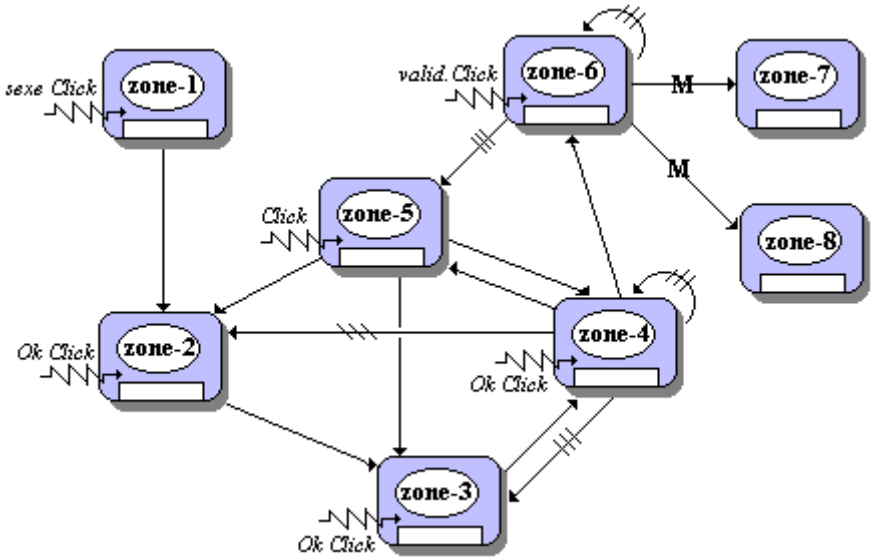
Risque à 10 ans :

2. Le mode saisie et les plans d'action

Ce mode est dans le graphe événementiel le chemin produisant le plus grand nombre de lignes de code. C'est pourquoi il fait l'objet d'une étude à part.

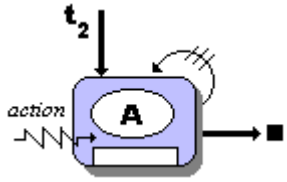
2.1 Graphe général des plans d'action

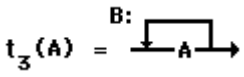
Nous présentons le graphe événementiel assurant le séquencement des plans d'action associés chacun à une des 8 zones décrites plus haut.



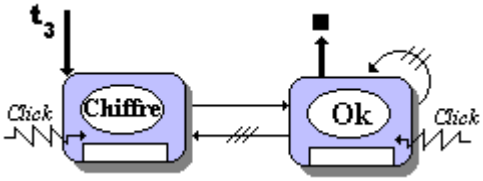
Nous avons choisi, pour la plupart des zones de représenter l'activation - désactivation par la propriété " visible ".
Pour les zones de saisie 2 et 3 où nous demandons à l'utilisateur d'entrer un nombre, nous avons choisi la saisie dirigée par la syntaxe. Un nombre est décrit par les diagrammes syntaxiques suivants :

Nombre



de la forme : 

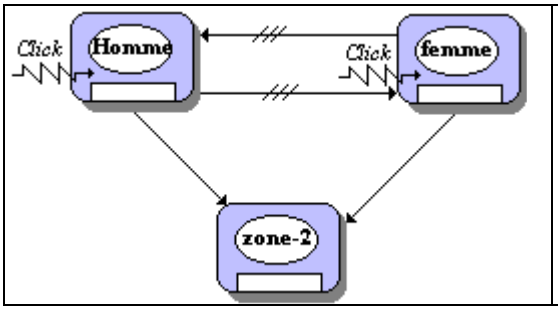
Opérateur t_3 dont nous avons déjà spécifié une implantation :



Chiffre sera implanté par un ensemble de boutons clickables associés chacun à un seul chiffre de 0 à 9.
Le bouton " Ok " permet de passer au plan d'action suivant.

2.2 Graphe événementiel de la zone-1

Table des états initiaux :



zone-1 activé
Homme désactivé
Femme désactivé
Zone-2 désactivé

Les objets " homme " et " femme " sont implantés par les objets visuels de RadioButtonHomme et RadioButtonFemme de la classe des TRadioButton.

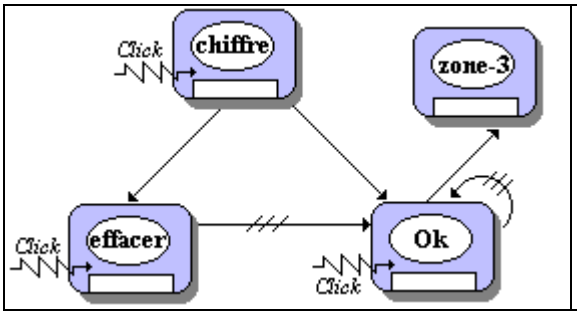
L'activation de la zone-2 (plan suivant)consiste à la rendre visible.

L'action du click sur l'une des deux cases dans zone-1

fait passer au plan d'action suivant.

2.3 Graphe événementiel de la zone-2

Table des états initiaux :



zone-2 activé
chiffre activé
effacer désactivé
Ok désactivé
Zone-3 désactivé

Les 10 objets " chiffre " sont implantés par les objets visuels de SpeedButtonAge1 à SpeedButtonAge10, les objets " effacer " et " ok " sont des objets visuels de type boutons poussoirs. L'activation de la zone-3 (plan suivant)consiste à la rendre visible.

Aspect visuel du deuxième plan d'action :

zone-2 avant saisie

à

zone-2 pendant saisie

le click sur le bouton Ok fait passer au plan d'action suivant

2.4 Graphe événementiel de la zone-3

Strictement identique au précédent avec comme seule différence le fait que l'objet (bouton poussoir) " ok " active la zone-4.

Aspect visuel du troisième plan d'action :

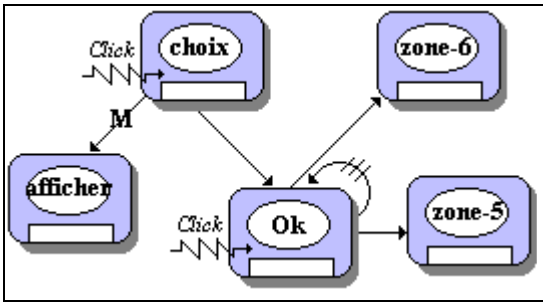
zone-3 avant saisie

zone-3 pendant saisie

à le click sur le bouton Ok fait passer au plan d'action suivant.

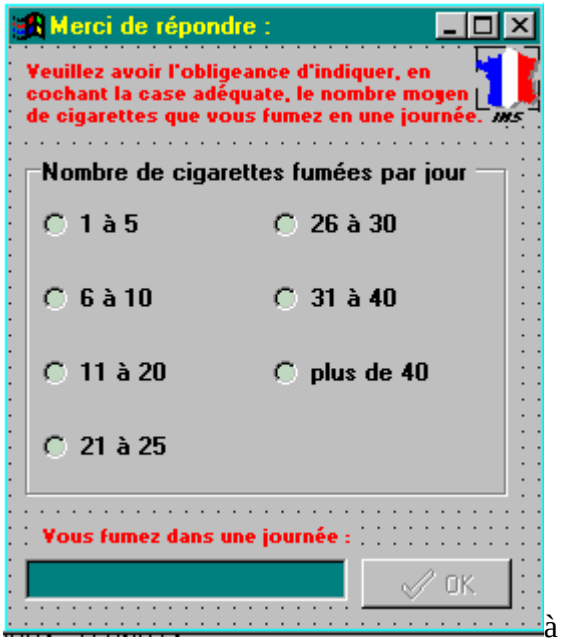
2.5 Graphe événementiel de la zone-4

Table des états initiaux :

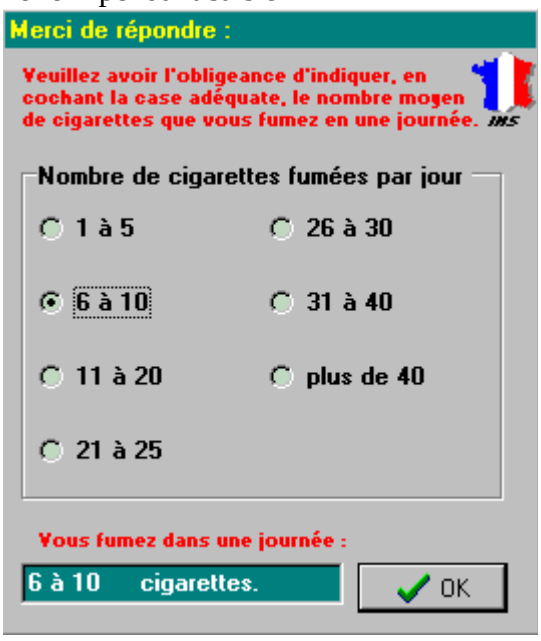
	zone-4 activé Ok désactivé afficher désactivé Zone-5 désactivé Zone-6 désactivé

Aspect visuel du quatrième plan d'action :

zone-4 avant saisie



zone-4 pendant saisie

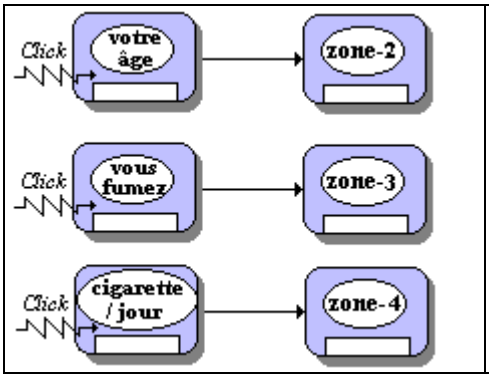


à

le click sur le bouton Ok fait passer au plan d'action suivant. **¯**

2.6 Graphe événementiel de la zone-5

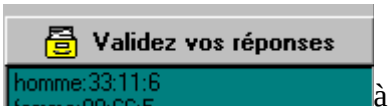
Table des états initiaux :



zone-6 activé
votre âge activé
vous fumez activé
cigarette/j activé
Zone-2 désactivé
Zone-3 désactivé
Zone-4 désactivé

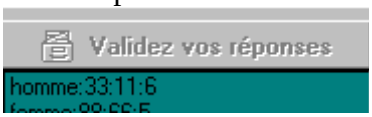
Aspect visuel du cinquième plan d'action :
Le bouton de validation de la zone-6 est activé.

zone-6 avant saisie



à

zone-6 après saisie

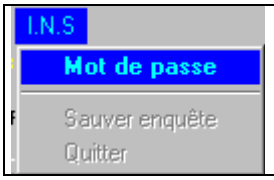


Fin du séquencement des plans d'action : l'interface est remise à l'état initial.

3. Le reste du logiciel

Ce qui reste à décrire dans notre logiciel figure à des fins pédagogiques afin que le lecteur puisse voir que la notion de séquencement de plan d'action ne se limite pas à l'utilisation des seuls objets visuels que sont les boutons. La liaison est faite entre une autre fiche de dialogue permettant de saisir un mot de passe et l'activation-désactivation de zones de menu.

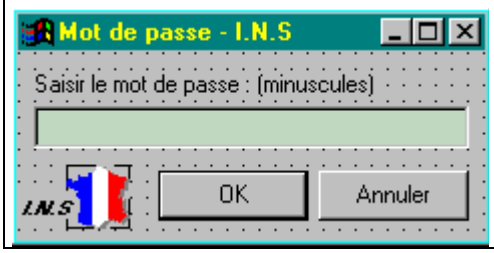
3.1 Le menu lance la saisie du mot de passe



Remarque
L'utilisateur clique sur la zone surlignée.

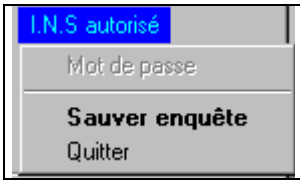
Le clic appelle le gestionnaire suivant :

```
procedure TForm1.Motdepasse1Click(Sender: TObject);
{le menu affiche une fiche de saisie du mot de passe dans UFmotPasse}
begin
  MotdePass.Showmodal;
end;
```



La fiche ci-contre (name = MotdePass) qui est dans l'unité *UFmotPasse*, est affichée en catégorie modale (impossible de cliquer ailleurs).

Si le mot de passe est valide, la fiche " MotdePass " se ferme et informe le menu " INS " en désactivant une zone (la zone mot de passe) et en activant les deux qui étaient inactivées au départ.



Remarque
le séquencement d'action est assuré aussi de cette façon.

Chapitre 7.5 utilisation des bases de données

Plan du chapitre:

1. Introduction et Généralités

1.1 Notion de système d'information

L'informatique est une science du traitement de l'information, laquelle est représentée par des données Aussi, très tôt, on s'est intéressé aux diverses manières de pouvoir stocker des données dans des mémoires auxiliaires autres que la mémoire centrale. Les données sont stockées dans des périphériques dont les supports physiques ont évolué dans le temps : entre autres, d'abord des cartes perforées, des bandes magnétiques, des cartes magnétiques, des mémoires à bulles magnétiques, puis aujourd'hui des disques magnétiques, ou des CD-ROM ou des DVD.

La notion de fichier est apparue en premier : le fichier regroupe tout d'abord des objets de même nature , des enregistrements. Pour rendre facilement exploitables les données d'un fichier, on a pensé à différentes méthodes d'accès (accès séquentiel, direct, indexé).

Toute application qui gère des systèmes physiques doit disposer de paramètres sémantiques décrivant ces systèmes afin de pouvoir en faire des traitements. Dans des systèmes de gestion de clients les paramètres sont très nombreux (noms, prénoms, adresse, n°Sécu, sport favori, est satisfait ou pas,..) et divers (alphabétiques, numériques, booléens, ...).

Dès que la quantité de données est très importante, les fichiers montrent leurs limites et il a fallu trouver un moyen de stocker ces données et de les organiser d'une manière qui soit facilement accessible.

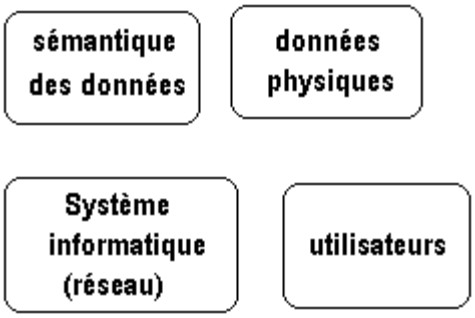
Base de données (BD)

> Une BD est composée de données stockées dans des mémoires de masse sous une forme structurée, et accessibles par des applications différentes et des utilisateurs différents. Une BD doit pouvoir être utilisée par plusieurs utilisateurs en "même temps".

Une base de données est structurée par définition, mais sa structuration doit avoir un caractère universel : il ne faut pas que cette structure soit adaptée à une application particulière, mais qu'elle puisse être utilisable par plusieurs applications distinctes. En effet, un même ensemble de données peut être commun à plusieurs systèmes de traitement dans un problème physique (par exemple la liste des passagers d'un avion, stockée dans une base de données, peut aussi servir au service de police à vérifier l'identité des personnes interdites de séjour, et au service des douanes pour associer des bagages aux personnes….).

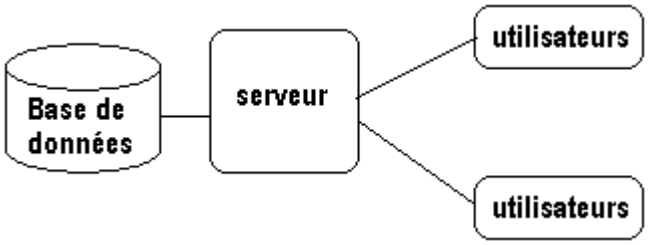
Système d'information

> Dans une entreprise ou une administration, la structure sémantique des données, leur organisation logique et physique, le partage et l'accès à de grandes quantités de données grâce à un système informatique, se nomme un système d'information.



Les entités qui composent un système d'information

L'organisation d'un SI relève plus de la gestion que de l'informatique et n'a pas exactement sa place dans un document sur la programmation. En revanche la cheville ouvrière d'un système d'information est un outil informatique appelé un **SGBD** (système de gestion de base de données) qui repose essentiellement sur un système informatique composé traditionnellement d'une **BD** et d'un réseau de postes de travail consultant ou mettant à jour les informations contenues dans la base de données, elle-même généralement située sur un ordinateur-serveur.



Système de Gestion de Base de Données (SGBD)

Un SGBD est un ensemble de logiciels chargés d'assurer les fonctions minimales suivantes :

- Le maintien de la cohérence des données entre elles,
- le contrôle d'intégrité des données accédées,
- les autorisations d'accès aux données,
- les opérations classiques sur les données (consultation, insertion , modification, suppression)

La cohérence des données est subordonnée à la définition de contraintes d'intégrité qui sont des règles que doivent satisfaire les données pour être acceptées dans la base. Les contraintes d'intégrité sont contrôlées par le moteur du SGBD :

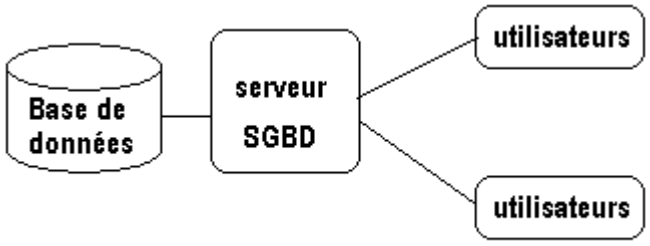
- au niveau de chaque champ, par exemple le : prix est un nombre positif, la date de naissance est obligatoire.
- Au niveau de chaque table - voir plus loin la notion de clef primaire : deux personnes ne doivent pas avoir à la fois le même nom et le même prénom.
- Au niveau des relations entre les tables : contraintes d'intégrité référentielles.

Par contre la redondance des données (formes normales) **n'est absolument pas vérifiée automatiquement** par les SGBD, il faut faire des requêtes spécifiques de recherche

d'anomalies (dites post-mortem) **à postériori**, ce qui semble être une grosse lacune de ces systèmes puisque les erreurs sont déjà présentes dans la base !

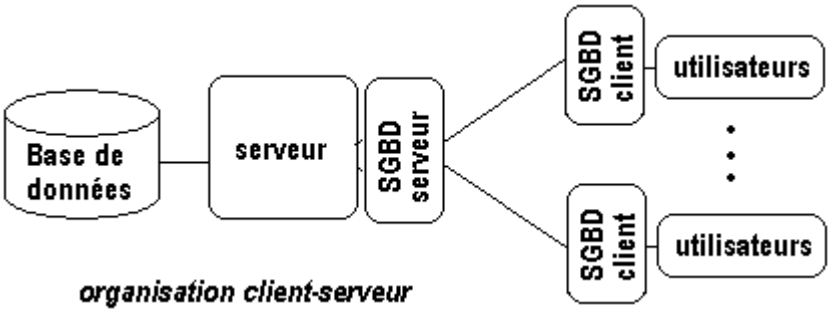
On organise actuellement les SGBD selon deux modes :

L'organisation locale selon laquelle le SGBD réside sur la machine où se trouve la base de données :



organisation locale

L'organisation client-serveur selon laquelle sur le SGBD est réparti entre la machine serveur locale supportant la BD (partie SGBD serveur) et les machines des utilisateurs (partie SGBD client). Ce sont ces deux parties du SGBD qui communiquent entre elles pour assurer les transactions de données :



organisation client-serveur

Le caractère généraliste de la structuration des données induit une description abstraite de l'objet BD (Base de données). Les applications étant indépendantes des données, ces dernières peuvent donc être manipulées et changées indépendamment du programme qui y accédera en implantant les méthodes générales d'accès aux données de la base, conformément à sa structuration abstraite.

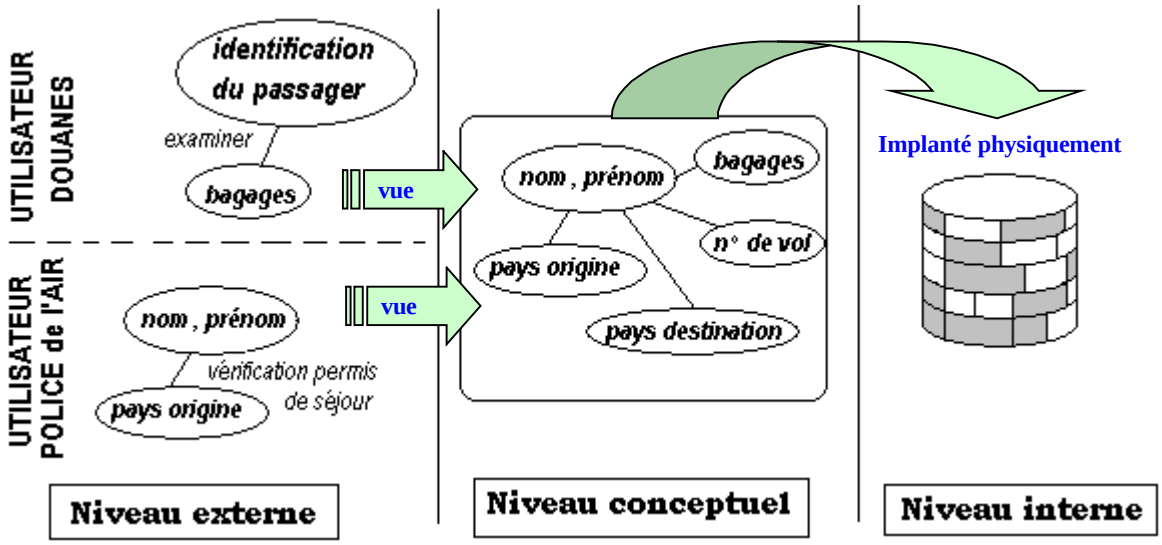
Une Base de Données peut être décrite de plusieurs points de vue, selon que l'on se place du côté de l'utilisateur ou bien du côté du stockage dans le disque dur du serveur ou encore du concepteur de la base.

Il est admis de nos jours qu'une **BD** est décrite en trois niveaux d'abstraction : un seul niveau a une existence matérielle physique et les deux autres niveaux sont une explication abstraite de ce niveau matériel.

Les 3 niveaux d'abstraction définis par l'ANSI depuis 1975

- **Niveau externe** : correspond à ce que l'on appelle une vue de la BD ou la façon dont sont perçues au niveau de l'utilisateur les données manipulées par une certaine application (vue abstraite sous forme de schémas)
- **Niveau conceptuel** : correspond à la description abstraite des composants et des processus entrant dans la mise en œuvre de la BD. Le niveau conceptuel est le plus important car il est le résultat de la traduction de la description du monde réel à l'aide d'expressions et de schémas conformes à un modèle de définition des données.
- **Niveau interne** : correspond à la description informatique du stockage physique des données (fichiers séquentiels, indexages, tables de hachage,…) sur le disque dur.

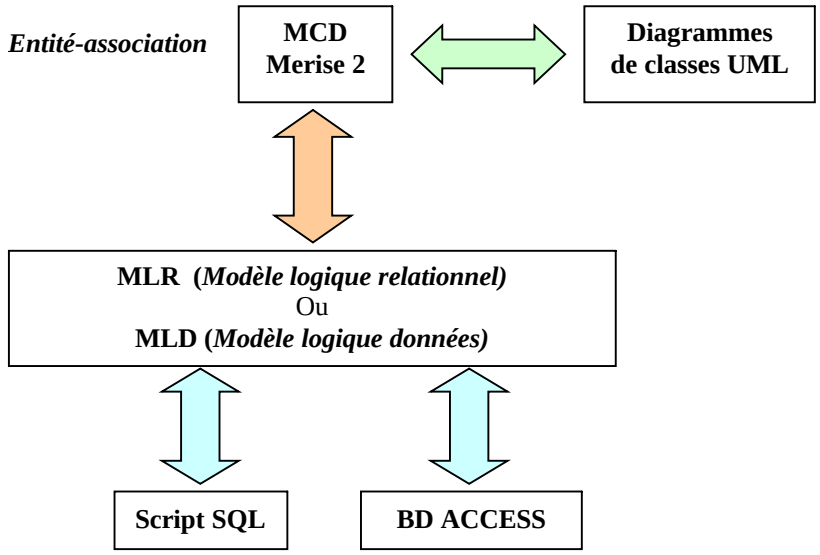
Figurons pour l'exemple des passagers d'un avion , stockés dans une base de données de la compagnie aérienne, sachant qu'en plus du personnel de la compagnie qui a une vue externe commerciale sur les passagers, le service des douanes peut accéder à un passager et à ses bagages et la police de l'air peut accéder à un passager et à son pays d''embarquement.



Le niveau conceptuel forme l'élément essentiel d'une BD et donc d'un SGBD chargé de gérer une BD, il est décrit avec un modèle de conception de données MCD avec la méthode française Merise qui est très largement répandu, ou bien par le formalisme des diagrammes de classes UML qui prend une part de plus en plus grande dans le formalisme de description conceptuelle des données (rappelons qu'UML est un langage de modélisation formelle, orienté objet et graphique ; Merise2 a intégré dans Merise ces concepts mais ne semble pas beaucoup être utilisé). Nous renvoyons le lecteur intéressé par cette partie aux très nombreux ouvrages écrits sur Merise ou sur UML.

Dans la pratique actuelle les logiciels de conception de BD intègrent à la fois la méthode Merise 2 et les diagrammes de classes UML. Ceci leur permet surtout la génération automatique et semi-automatique (paramétrable) de la BD à partir du modèle conceptuel sous forme de scripts (programmes simples) SQL adaptés aux différents SGBD du marché (ORACLE, SYBASE, MS-SQLSERVER,…) et les différentes versions de la BD ACCESS. Les logiciels de conception actuels permettent aussi la rétro-génération (ou reverse engeneering) du modèle à partir d'une BD existante, cette fonctionnalité est très utile pour reprendre un travail mal documenté.

En résumé pratique :



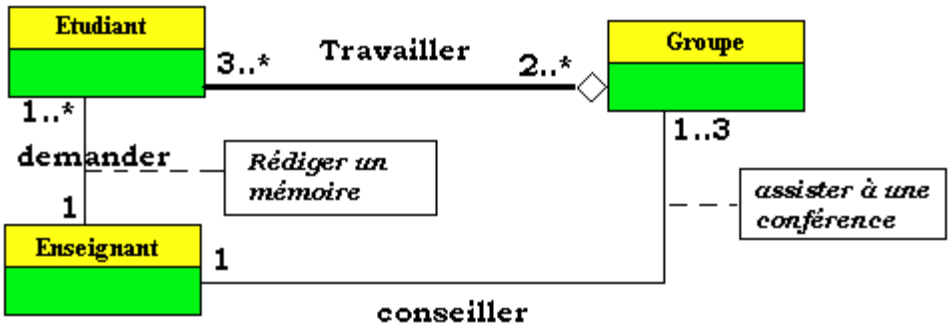
C'est en particulier le cas du logiciel français WIN-DESIGN dont une version démo est disponible à www.win-design.com et de son rival POWER-AMC (ex AMC-DESIGNOR).

Signalons enfin un petit logiciel plus modeste, très intéressant pour débuter avec version limitée seulement par la taille de l'exemple : CASE-STUDIO chez CHARONWARE. Les logiciels basés uniquement sur UML sont , à ce jour, essentiellement destinés à la génération de code source (Java, Delphi, VB, C++,…), les versions **Community** (versions logicielles libres) de ces logiciels ne permettent pas la génération de BD ni celle de scripts SQL.

Les quelques schémas qui illustreront ce chapitre seront décrits avec le langage UML.

L'exemple ci-après schématise en UML le mini-monde universitaire réel suivant :

- un enseignant pilote entre 1 et 3 groupes d'étudiants,
- un enseignant demande à 1 ou plusieurs étudiants de rédiger un mémoire,
- un enseignant peut conseiller aux groupes qu'il pilote d'aller assister à une conférence,
- un groupe est constitué d'au moins 3 étudiants,
- un étudiant doit s'inscrire à au moins 2 groupes.



Si le niveau conceptuel d'une BD est assis sur un modèle de conceptualisation de haut niveau (Merise, UML) des données, il est ensuite fondamentalement traduit dans le Modèle Logique de représentation des Données (MLD). Ce dernier s'implémentera selon un modèle physique des données.

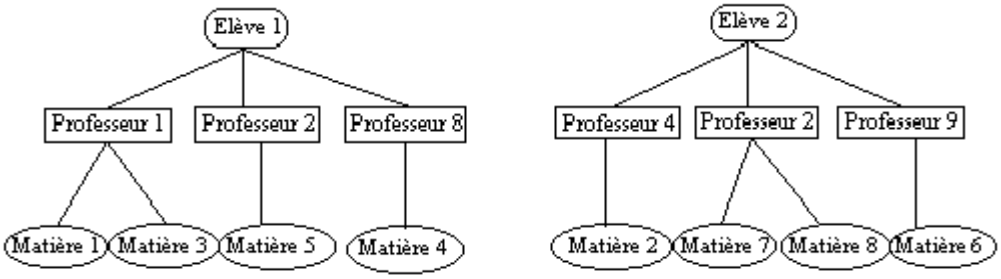
Il existe plusieurs MLD **M**odèles **L**ogiques de **D**onnées et plusieurs modèles physiques, et pour un même MLD, on peut choisir entre plusieurs modèles physiques différents.

Il existe 5 grands modèles logiques pour décrire les bases de données.

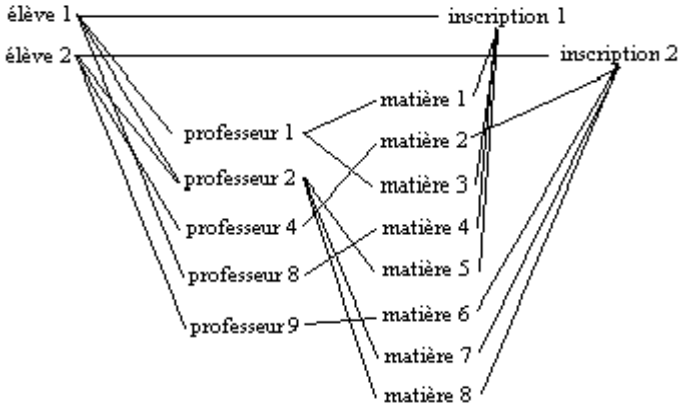
Les modèles de données historiques

(Prenons un exemple comparatif où des élèves ont des cours donnés par des professeurs leur enseignant certaines matières (les enseignants étant pluridisciplinaires)

- **Le modèle hiérarchique:** l'information est organisée de manière arborescente, accessible uniquement à partir de la racine de l'arbre hiérarchique. Le problème est que les points d'accès à l'information sont trop restreints.



- **Le modèle réseau:** toutes les informations peuvent être associées les unes aux autres et servir de point d'accès. Le problème est la trop grande complexité d'une telle organisation.



- **Le modèle relationnel**: toutes les relations entre les objets contenant les informations sont décrites et représentées sous la forme de tableaux à 2 dimensions.

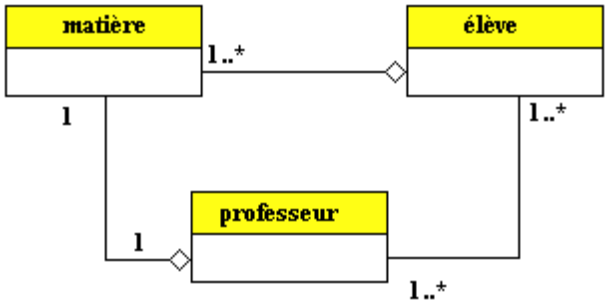
élève 1	matière 1
élève 1	matière 3
élève 1	matière 4
élève 1	matière 5
élève 2	matière 2
élève 2	matière 6
élève 2	matière 7
élève 2	matière 8

professeur 1	matière 1
professeur 1	matière 3
professeur 2	matière 5
professeur 2	matière 7
professeur 2	matière 8
professeur 4	matière 2
professeur 8	matière 4
professeur 9	matière 6

Dans ce modèle, la gestion des données (insertion, extraction,…) fonctionne selon la théorie mathématique de l'algèbre relationnelle. C'est le modèle qui allie une grande indépendance vis à vis des données à une simplicité de description.

- **Le modèle par déduction :** comme dans le modèle relationnel les données sont décrites et représentées sous la forme de tableaux à 2 dimensions. La gestion des données (insertion, extraction,…) fonctionne selon la théorie mathématique du calcul dans la logique des prédicats. Il ne semble exister de SGBD commercial directement basé sur ce concept. Mais il est possible de considérer un programme Prolog (programmation en logique) comme une base de données car il intègre une description des données. Ce sont plutôt les logiciels de réseaux sémantiques qui sont concernés par cette approche (cf. logiciel AXON).

- **Le modèle objet :** les données sont décrites comme des classes et représentées sous forme d'objets, un modèle **relationnel-objet** devrait à terme devenir le modèle de base.



L'expérience montre que le modèle relationnel s'est imposé parce qu'il était le plus simple en terme d'indépendance des données par rapport aux applications et de facilité de représenter les données dans notre esprit. C'est celui que nous décrirons succinctement dans la suite de ce chapitre.

2.Le modèle de données relationnel

Défini par EF Codd de la société IBM dès 1970, ce modèle a été amélioré et rendu opérationnel dans les années 80 sous la forme de SBGD-R (SGBD Relationnels). Ci-dessous une liste non exhaustive de tels SGBD-R :

Access de Microsoft,
Oracle,
DB2 d'IBM,
Interbase de Borland,
SQL server de microsoft,
Informix,
Sybase,
MySQL,
PostgreSQL, ….

Nous avons déjà vu dans un précédent chapitre, la notion de relation binaire : une relation binaire R est un sous-ensemble d'un produit cartésien de deux ensembles finis E et F que nous nommerons domaines de la relation R :

$$R \subseteq E \times F$$

Cette définition est généralisable à n domaines, nous dirons que **R** est une relation n-aire sur les domaines E_1 , E_2 , … , E_n si et seulement si :

$$R \subseteq E_1 \times E_2 \ldots \times E_n$$

Les ensembles E_k peuvent être définis comme en mathématiques : en extension ou en compréhension :

E_k = { 12 , 58 , 36 , 47 } *en extension*

E_k = { x / (x est entier) **et(** x ∈ [1, 20]) } *en compréhension*

Notation

si nous avons: $R = \{ (v_1 , v_2 \ldots , v_n) \}$,
Au lieu d'écrire : $(v_1 , v_2 \ldots , v_n) \in R$, on écrira $R(v_1 , v_2 , \ldots , v_n)$

Exemple de déclarations de relations :

Passager (nom, prénom, n° de vol, nombre de bagages) , cette relation contient les informations utiles sur un passager d'une ligne aérienne.

Personne (nom, prénom) , cette relation caractérise une personne avec deux attributs

Enseignement (professeur, matière) , cette relation caractérise un enseignement avec le nom de la matière et le professeur qui l'enseigne.

Schéma d'une relation

On appelle schéma de la relation R : $R(a_1 : E_1, a_2 : E_2 ,\ldots, a_n : E_n)$
Où $(a_1 , a_2 \ldots , a_n)$ sont appelés les **attributs**, chaque attribut a_k indique comment est utilisé dans la relation R le domaine E_k , chaque attribut prend sa valeur dans le domaine qu'il définit, nous notons $val(a_k)= v_k$ où v_k est un élément (une valeur) quelconque de l'ensemble E_k (domaine de l'attribut a_k).

Convention : lorsqu'il n'y a pas de valeur associée à un attribut dans un n-uplet, on convient de lui mettre une valeur spéciale notée **null**, indiquant l'absence de valeur de l'attribut dans ce n-uplet.

Degré d'une relation

On appelle degré d'une relation, le nombre d'attributs de la relation.

Exemple de schémas de relations :

Passager (nom : chaîne, prénom : chaîne, n° de vol : entier, nombre de bagages : entier) relation de degré 4.

Personne (nom : chaîne, prénom : chaîne) relation de degré 2.

Enseignement (professeur : ListeProf, matière : ListeMat) relation de degré 2.

*Attributs : prenons le schéma de la relation **Enseignement***

Enseignement (professeur : ListeProf, matière : ListeMat). C'est une relation binaire (degré 2) sur les deux domaines ListeProf et ListeMat. L'attribut professeur joue le rôle d'un paramètre formel et le domaine ListeProf celui du type du paramètre.

Supposons que :

ListeProf ={ Poincaré, Einstein, Lavoisier, Raimbault , Planck }
ListeMat = { mathématiques, poésie , chimie , physique }

L'attribut professeur peut prendre toutes valeurs de l'ensemble ListeProf :

Val(professeur) = Poincaré, …. , Val(professeur) = Raimbault

Si l'on veut dire que le poste d'enseignant de chimie n'est pas pourvu on écrira :

Le couple (**null** , chimie) est un couple de la relation **Enseignement.**

Enregistrement dans une relation

Un n-uplet $(val(a_1), val(a_2) \ldots , val(a_n)) \in R$ est appelé un enregistrement de la relation R. Un enregistrement est donc constitué de valeurs d'attributs.

Dans l'exemple précédent (Poincaré , mathématiques), (Raimbault , poésie) , (**null** , chimie) sont trois enregistrements de la relation **Enseignement**.

Clef d'une relation

Si l'on peut caractériser d'une façon **bijective** tout n-uplet d'attributs $(a_1 , a_2 \ldots , a_n)$ avec seulement un sous-ensemble restreint $(a_{k1} , a_{k2} \ldots , a_{kp})$ avec $p < n$, de ces attributs, alors ce sous-ensemble est appelé une **clef** de la relation. Une relation peut avoir plusieurs clefs, nous choisissons l'une d'elle en la désignant comme **clef primaire de la relation**.

Clef minimale d'une relation

On a intérêt à ce que la clef **primaire soit minimale** en nombre d'attributs, car il est clair que si un sous-ensemble à p attributs $(a_{k1} , a_{k2} \ldots , a_{kp})$ est une clef, tout sous-ensemble à p+m attributs dont les p premiers sont les $(a_{k1} , a_{k2} \ldots , a_{kp})$ est aussi une clef :
$(a_{k1} , a_{k2} \ldots , a_{kp} , a_0 , a_1)$
$(a_{k1} , a_{k2} \ldots , a_{kp} , a_{10} , a_5 , a_9 , a_2)$ sont aussi des clefs etc…

Il n'existe aucun moyen méthodique formel général pour trouver une clef primaire d'une relation, il faut observer attentivement. Par exemple :

- Le code Insee est une clef primaire permettant d'identifier les personnes.
- Si le couple (nom, prénom) peut suffire pour identifier un élève dans une classe d'élèves, et pourrait être choisi comme clef primaire, il est insuffisant au niveau d'un lyçée où il est possible que l'on trouve plusieurs élèves portant le même nom et le même premier prénom ex: (martin, jean).

Convention : on souligne dans l'écriture d'une relation dont on a déterminé une clef primaire, les attributs faisant partie de la clef.

Clef secondaire d'une relation

Tout autre clef de la relation qu'une clef primaire (minimale) , exemple :
Si $(a_{k1} , a_{k2} \ldots , a_{kp})$ est un clef primaire de R
$(a_{k1} , a_{k2} \ldots , a_{kp} , a_0 , a_1)$ et $(a_{k1} , a_{k2} \ldots , a_{kp} , a_{10} , a_5 , a_9 , a_2)$ sont des clefs secondaires.

Clef étrangère d'une relation

Soit $(a_{k1} , a_{k2} \ldots , a_{kp})$ un p-uplet d'attributs d'une relation R de degré n. [$R(a_1 : E_1, a_2 : E_2 ,\ldots, a_n : E_n)$]

Si $(a_{k1} , a_{k2} \ldots , a_{kp})$ est une clef primaire d'une autre relation Q on dira que $(a_{k1} , a_{k2} \ldots , a_{kp})$ est une clef étrangère de R.

Convention : on met un # après chaque attribut d'une clef étrangère.

Exemple de clef secondaire et clef étrangère :

Passager (nom# : chaîne, prénom# : chaîne , n° de vol : entier, nombre de bagages : entier, n° client : entier) relation de degré 5.

Personne (nom : chaîne, prénom : chaîne , âge : entier, civilité : Etatcivil) relation de degré 4.

n° client est une clef primaire de la relation **Passager**.
(nom, n° client) est une clef secondaire de la relation **Passager**.
(nom, n° client , n° de vol) est une clef secondaire de la relation **Passager**….etc

(nom , prénom) est une clef primaire de la relation **Personne**, comme (nom# , prénom#) est aussi un couple d'attributs de la relation **Passager**, c'est une clef étrangère de la relation **Passager**.

On dit aussi que dans la relation **Passager**, le couple d'attributs (nom# , prénom#) réfère à la relation **Personne**.

Règle d'intégrité référentielle

> Toutes les valeurs d'une clef étrangère (v_{k1} , v_{k2} … , v_{kp}) se retrouvent comme valeur de la clef primaire de la relation référée (ensemble des valeurs de la clef étrangère est **inclus** au sens large dans l'ensemble des valeurs de la clef primaire)

Reprenons l'exemple précédent

(nom , prénom) est une clef étrangère de la relation **Passager**, c'est donc une clef primaire de la relation **Personne** : les domaines (liste des noms et liste des prénoms associés au nom doivent être strictement les mêmes dans **Passager** et dans **Personne**, nous dirons que la clef étrangère respecte la contrainte d'intégrité référentielle.

Règle d'intégrité d'entité

> Aucun des attributs participant à une clef primaire ne peut avoir la valeur **null**.

Nous définirons la valeur **null**, comme étant une valeur spéciale n'appartenant pas à un domaine spécifique mais ajoutée par convention à tous les domaines pour indiquer qu'un champ n'est pas renseigné.

Représentation sous forme tabulaire

Reprenons les relations Passager et Personne et figurons un exemple pratique de valeurs des relations.

Passager (nom# : chaîne, prénom# : chaîne, n° de vol : entier, nombre de bagages : entier, n° client : entier).

Personne (nom : chaîne, prénom : chaîne, âge : entier, civilité : Etatcivil) relation de degré 4.

Nous figurons les tables de valeurs des deux relations

Passager :

Clef étrangère

nom#	prénom#	N°vol	Nbr bagages	N° client
Einstein	Albert	45622	2	154565
Lavoisier	Antoine	45622	1	785154
Raimbault	Arthur	12896	2	544552
Poincaré	Henri	45644	3	781201
Einstein	Albert	75906	**null**	858547
Lavoisier	Antoine	45644	4	235002

Personne:

Clef primaire

Clef primaire

nom	prénom	âge	Civilité
Einstein	Albert	45	marié
Lavoisier	Antoine	41	marié
Planck	Max	52	veuf
Raimbault	Arthur	25	célibataire
Poincaré	Henri	45	marié

Nous remarquons que la compagnie aérienne attribue un numéro de client unique à chaque personne, c'est donc un bon choix pour une clef primaire.

Les deux tables (relations) ont deux colonnes qui portent les mêmes noms colonne **nom** et colonne **prénom**, ces deux colonnes forment une clef primaire de la table **Personne**, c'est donc une clef étrangère de **Passager** qui réfère **Personne.**

En outre, cette clef étrangère respecte la contrainte d'intégrité référentielle : la liste des valeurs de la clef étrangère dans **Passager** est incluse dans la liste des valeurs de la clef primaire associée dans **Personne**.

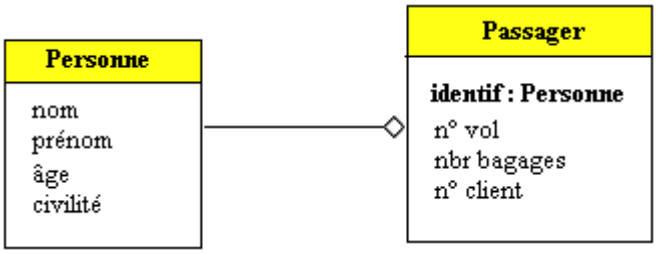


Diagramme UML modélisant la liaison Passager-Personne

Ne pas penser qu'il en est toujours ainsi, par exemple voici une autre relation **Passager2** dont la clef étrangère ne respecte pas la contrainte d'intégrité référentielle :

Clef étrangère , réfère Personne

Passager2 :

nom	prénom	N°vol	Nbr bagages	N° client
Einstein	Albert	45622	2	154565
Lavoisier	Antoine	45644	1	785154
Raimbault	Arthur	12896	2	544552
Poincaré	Henri	45644	3	781201
Einstein	Albert	75906	**null**	858547
Picasso	Pablo	12896	5	458023

En effet, le couple (Picasso , Pablo) n'est pas une valeur de la clef primaire dans la table **Personne**.

Principales règles de normalisation d'une relation

1ère forme normale :

Une relation est dite en première forme normale si, chaque attribut est représenté par un identifiant unique (les valeurs ne sont pas des ensembles, des listes,…) .Ci-dessous une relation qui n'est pas en 1ère forme normale car l'attribut **n° vol** est multivalué (il peut prendre 2 valeurs) :

nom	prénom	N°vol	Nbr bagage	N° client
Einstein	Albert	45622 , 75906	2	154565
Lavoisier	Antoine	45644 , 45622	1	785154
Raimbault	Arthur	12896	2	544552
Poincaré	Henri	45644	3	781201
Picasso	Pablo	12896	5	458023

En pratique, il est très difficile de faire vérifier automatiquement cette règle, dans l'exemple proposé on pourrait imposer de passer par un masque de saisie afin que le N°vol ne comporte que 5 chiffres.

2ème forme normale :

Une relation est dite en deuxième forme normale si, elle est **en première forme normale** et si chaque attribut qui n'est pas une clef, dépend entièrement de tous les attributs qui composent la clef primaire. La relation **Personne** (nom : chaîne, prénom : chaîne , age : entier , civilité : Etatcivil) est en deuxième forme normale :

nom	**prénom**	**âge**	**Civilité**
Einstein	Albert	45	marié
Lavoisier	Antoine	41	marié
Planck	Max	52	veuf
Raimbault	Arthur	25	célibataire
Poincaré	Henri	45	marié

Car l'attribut âge ne dépend que du nom et du prénom, de même pour l'attribut civilité.

La relation **Personne3** (nom : chaîne, prénom : chaîne , age : entier , civilité : Etatcivil) qui a pour clef primaire (nom , âge) n'est pas en deuxième forme normale :

nom	**prénom**	**âge**	**Civilité**
Einstein	Albert	45	marié
Lavoisier	Antoine	41	marié
Planck	Max	52	veuf
Raimbault	Arthur	25	célibataire
Poincaré	Henri	45	marié

Car l'attribut Civilité ne dépend que du nom et non pas de l'âge ! Il en est de même pour le prénom, soit il faut changer de clef primaire et prendre (nom, prénom) soit si l'on conserve la clef primaire (nom , âge) , il faut décomposer la relation **Personne3** en deux autres relations **Personne31** et **Personne32** :

nom	**âge**	**Civilité**
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf
Raimbault	25	célibataire
Poincaré	45	marié

Personne31(nom : chaîne, age : entier , civilité : Etatcivil) : **2ème forme normale**

nom	**âge**	**prénom**
Einstein	45	Albert
Lavoisier	41	Antoine
Planck	52	Max
Raimbault	25	Arthur
Poincaré	45	Henri

Personne32(nom : chaîne, age : entier , prénom : chaîne) : **2ème forme normale**

En pratique, il est aussi très difficile de faire vérifier automatiquement la mise en deuxième forme normale. Il faut trouver un jeu de données représentatif.

3ème forme normale :

Une relation est dite en troisième forme normale si chaque attribut qui ne compose pas la clef primaire, dépend directement de son identifiant et à travers une dépendance fonctionnelle. Les relations précédentes sont toutes en forme normale. Montrons un exemple de relation qui n'est pas en forme normale. Soit la relation **Personne4** (nom : chaîne, age : entier, civilité : Etatcivil, salaire : monétaire) où par exemple le salaire dépend de la clef primaire et que l'attribut civilité ne fait pas partie de la clef primaire (nom , âge) **:**

nom	âge	Civilité	salaire
Einstein	45	marié	1000
Lavoisier	41	marié	1000
Planck	52	veuf	1800
Raimbault	25	célibataire	1200
Poincaré	45	marié	1000

salaire = f (Civilité) =>
Pas 3ème forme normale

L'attribut salaire dépend de l'attribut civilité, ce que nous écrivons salaire = f(civilité), mais l'attribut civilité ne fait pas partie de la clef primaire clef = (nom , âge), donc **Personne4** n'est pas en 3ème forme normale **:**

Il faut alors décomposer la relation **Personne4** en deux relations **Personne41** et **Personne42** chacune en troisième forme normale:

Personne41 :

nom	âge	Civilité#
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf
Raimbault	25	célibataire
Poincaré	45	marié

Personne41 en 3ème forme normale, civilité clef étrangère

Personne42 :

Civilité	salaire
marié	1000
veuf	1800
célibataire	1200

Personne42 en 3ème forme normale, civilité clef primaire

En pratique, il est également très difficile de faire contrôler automatiquement la mise en troisième forme normale.

Remarques pratiques importantes pour le débutant :

- Les spécialistes connaissent deux autres formes normales. Dans ce cas le lecteur intéressé par l'approfondissement du sujet, trouvera dans la littérature, de solides références sur la question.

- Si la clef primaire d'une relation n'est composée que d'un seul attribut (choix conseillé lorsque cela est possible, d'ailleurs on trouve souvent des clefs primaires sous forme de numéro d'identification client, Insee,…) automatiquement, la relation est en 2ème forme normale, car chaque autre attribut non clef étrangère, ne dépend alors que de la valeur unique de la clef primaire.

- Penser dès qu'un attribut est fonctionnellement dépendant d'un autre attribut qui n'est pas la clef elle-même à décomposer la relation (créer deux nouvelles tables).

- En l'absence d'outil spécialisé, il faut de la pratique et être très systématique pour contrôler la normalisation.

Base de données relationnelles BD-R:

Ce sont des données structurées à travers :

- Une famille de domaines de valeurs,
- Une famille de relations n-aires,
- Les contraintes d'intégrité sont respectées par toute clef étrangère et par toute clef primaire.
- Les relations sont en $3^{ème}$ forme normale. (à minima en $2^{ème}$ forme normale)

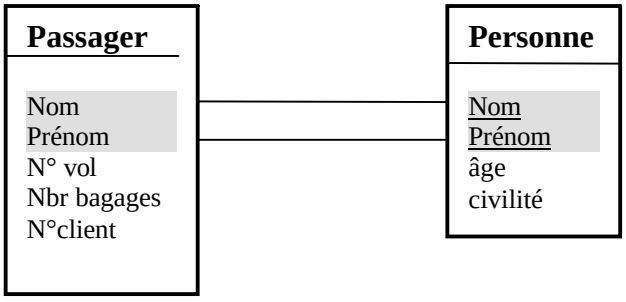
Les données sont accédées et manipulées grâce à un langage appelé **langage d'interrogation** ou **langage relationnel** ou **langage de requêtes**

Système de Gestion de Base de Données relationnel :

C'est une famille de logiciels comprenant :

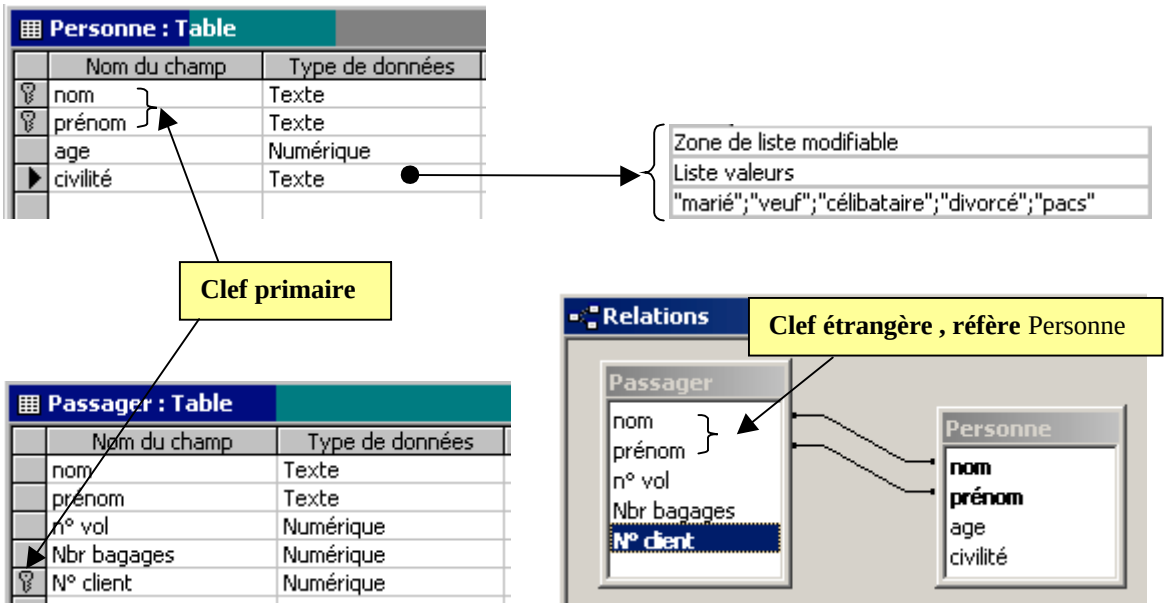
- Une BD-R.
- Un langage d'interrogation.
- Une gestion en interne des fichiers contenant les données et de l'ordonnancement de ces données.
- Une gestion de l'interface de communication avec les utilisateurs.
- La gestion de la sécurité des accès aux informations contenues dans la BD-R.

Le schéma relation d'une relation dans une BD relationnelle est noté graphiquement comme ci-dessous :

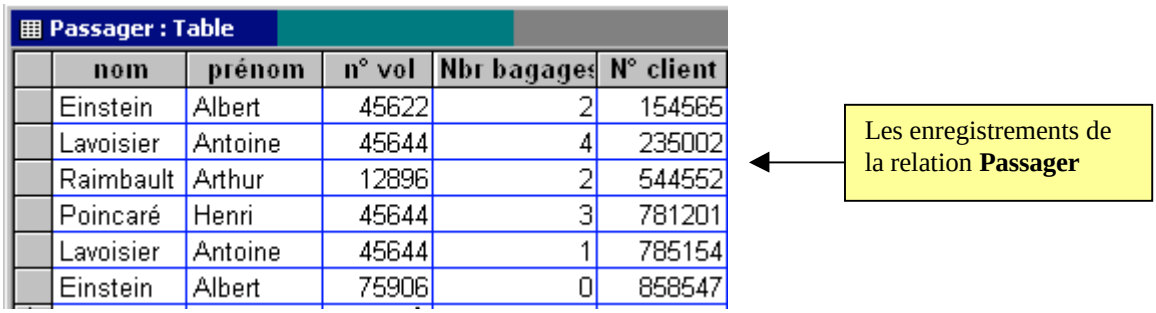


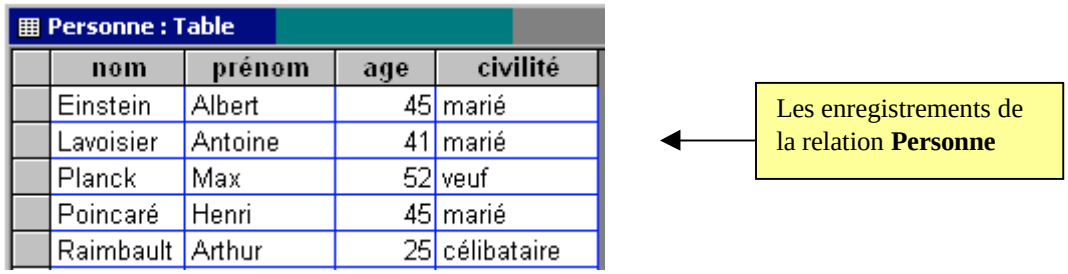
Les attributs reliés entre ces deux tables indiquent une liaison entre les deux relations.

Voici dans le **SGBD-R Access**, la représentation des schémas de relation ainsi que la liaison sans intégrité des deux relations précédentes **Passager** et **Personne** :



Access et la représentation des enregistrements de chaque table :





Les besoins d'un utilisateur d'une base de données sont classiquement ceux que l'on trouve dans tout ensemble de données structurées : insertion, suppression, modification, recherche avec ou sans critère de sélection. Dans une BD-R, ces besoins sont exprimés à travers un langage d'interrogation. Historiquement deux classes de langages relationnels équivalentes en puissance d'utilisation et de fonctionnement ont été inventées : les langages **algébriques** et les langages des **prédicats**.

Un langage relationnel n'est pas un langage de programmation : il ne possède pas les structures de contrôle de base d'un langage de programmation (condition, itération, …). **Très souvent il doit être utilisé comme complément à l'intérieur de programmes Delphi, Java,...**

Les langages d'interrogation prédicatifs sont des langages fondés sur la logique des prédicats du 1er ordre, le plus ancien s'appelle **Q**uery **B**y **E**xample QBE.

Ce sont les langages algébriques qui sont de loin les plus utilisés dans les SGBD-R du commerce, le plus connu et le plus utilisé dans le monde se dénomme le **S**tructured **Q**uery **L**anguage ou SQL. Un tel langage n'est qu'une implémentation en anglais d'opérations définies dans une algèbre relationnelle servant de modèle mathématique à tous les langages relationnels.

3. Principes fondamentaux d'une l'algèbre relationnelle

Une algèbre relationnelle est une famille d'opérateurs binaires ou unaires dont les opérandes sont des **relations**. Nous avons vu que l'on pouvait faire l'union, l'intersection , le produit cartésien de relations binaires dans un chapitre précédent, comme les relations n-aires sont des ensembles, il est possible de définir sur elle une algèbre opératoire utilisant les opérateurs classiques ensemblistes, à laquelle on ajoute quelques opérateurs spécifiques à la manipulation des données.

Remarque pratique :

> La phrase "**tous les n-uples sont distincts, puisqu'éléments d'un même ensemble nommé relation**" se transpose en pratique en la phrase **" toutes les lignes d'une même table nommée relation, sont distinctes** (même en l'absence de clef primaire explicite)".

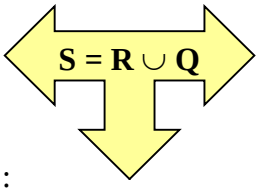
Nous exhibons les opérateurs principaux d'une algèbre relationnelle et nous montrerons pour chaque opération, un exemple sous forme.

Union de 2 relations

> Soient R et Q deux relations de même domaine et de même degré on peut calculer la nouvelle relation S = R ∪ Q de même degré et de même domaine contenant les enregistrements différents des deux relations R et Q :

R :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf



Q :

nom	âge	Civilité
Lupin	45	célibataire
Planck	52	veuf
Mozart	34	veuf
Gandhi	64	célibataire

S :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf

Lupin	45	célibataire
Mozart	34	veuf
Gandhi	64	célibataire

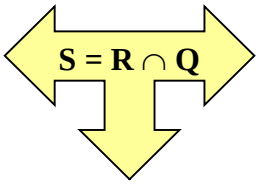
Remarque : (Planck, 52, veuf) ne figure qu'une seule fois dans la table R ∪ Q.

Intersection de 2 relations

Soient R et Q deux relations de même domaine et de même degré on peut calculer la nouvelle relation S = R ∩ Q de même degré et de même domaine contenant les enregistrements communs aux deux relations R et Q :

R :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf



Q :

nom	âge	Civilité
Einstein	45	marié
Planck	52	veuf
Mozart	34	veuf
Gandhi	64	célibataire

S :

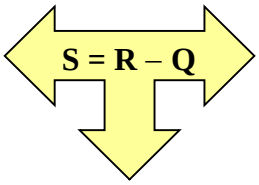
nom	âge	Civilité
Einstein	45	marié
Planck	52	veuf

Différence de 2 relations

Soient R et Q deux relations de même domaine et de même degré on peut calculer la nouvelle relation S = R – Q de même degré et de même domaine contenant les enregistrements qui sont présents dans R mais qui ne sont pas dans Q (on exclut de R les enregistrements qui appartiennent à R ∩ Q) :

R :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf



Q :

nom	âge	Civilité
Einstein	45	marié
Planck	52	veuf
Mozart	34	veuf
Gandhi	64	célibataire

S :

nom	âge	Civilité
Lavoisier	41	marié

Produit cartésien de 2 relations

Soient R et Q deux relations de domaine et de degré quelconques (degré(R)=n, degré(Q)=p), avec Domaine(R) ∩ Domaine(Q) = ∅ (pas d'attributs en communs).

On peut calculer la nouvelle relation S = R x Q de degré n + p et de domaine égal à l'union des domaines de R et de Q contenant tous les couples d'enregistrements à partir d'enregistrements présents dans R et d'enregistrements présents dans Q :

R :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf

S = R x Q

Q :

ville	km
Paris	874
Rome	920

S :

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Einstein	45	marié	Rome	920
Lavoisier	41	marié	Paris	874
Lavoisier	41	marié	Rome	920
Planck	52	Veuf	Paris	874
Planck	52	Veuf	Rome	920

Selection ou Restriction d'une relation

Soit R une relation, soit $R(a_1 : E_1, a_2 : E_2 ,\ldots, a_n : E_n)$ le schéma de cette relation. Soit Cond($a_1, a_2 , \ldots , a_n$) une expression booléenne classique (expression construite sur les attributs avec les connecteurs de l'algèbre de Boole et les opérateurs de comparaison < , > , = , >= , <= , <>)

On note S = **select** (Cond($a_1, a_2 , \ldots , a_n$) , R), la nouvelle relation S construite ayant le même schéma que R soit $S(a_1 : E_1, a_2 : E_2 ,\ldots, a_n : E_n)$, qui ne contient que les enregistrements de R qui satisfont à la condition booléenne Cond($a_1, a_2 , \ldots , a_n$).

R :

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Mozart	32	marié	Rome	587
Gandhi	64	célibataire	Paris	258
Lavoisier	41	marié	Rome	124
Lupin	42	Veuf	Paris	608
Planck	52	Veuf	Rome	405

Cond($a_1, a_2 , \ldots , a_n$) = { âge > 42 **et** ville=Paris }

S :

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Gandhi	64	célibataire	Paris	258

Select ({ âge > 42 et ville=Paris }, R) signifie que l'on ne recopie dans S que les enregistrements de R constitués des personnes ayant séjourné à Paris et plus âgées que 42 ans.

Projection d'une relation

Soit R une relation, soit $R(a_1 : E_1, a_2 : E_2 ,\ldots, a_n : E_n)$ le schéma de cette relation. On appelle S = **proj**($a_{k1} , a_{k2} \ldots , a_{kp}$) la projection de R sur un sous-ensemble restreint ($a_{k1} , a_{k2} \ldots , a_{kp}$) avec p < n , de ses attributs, la relation S ayant pour

schéma le sous-ensemble des attributs $S(a_{k1} : E_{k1}, a_{k2} : E_{k2}, \ldots, a_{kp} : E_{kp})$ et contenant les enregistrements différents obtenus en ne considérant que les attributs $(a_{k1}, a_{k2} \ldots, a_{kp})$.

Exemple

R :

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Mozart	32	marié	Rome	587
Lupin	42	Veuf	Paris	464
Einstein	45	marié	Venise	981
Gandhi	64	célibataire	Paris	258
Lavoisier	41	marié	Rome	124
Lupin	42	Veuf	Paris	608
Planck	52	Veuf	Rome	405

S1 = **proj**(nom , civilité)

nom	Civilité
Einstein	marié
Mozart	marié
Lupin	Veuf
Gandhi	célibataire
Lavoisier	marié
Planck	Veuf

S2 = **proj**(nom , âge)

nom	âge
Einstein	45
Mozart	32
Lupin	42
Gandhi	64
Lavoisier	41
Planck	52

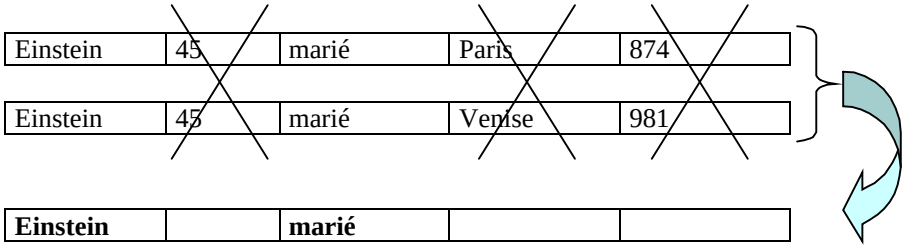
S3 = **proj**(nom , ville)

nom	ville
Einstein	Paris
Mozart	Rome
Lupin	Paris
Einstein	Venise
Gandhi	Paris
Lavoisier	Rome
Planck	Rome

S4 = **proj**(ville)

ville
Paris
Rome
Venise

Que s'est-il passé pour Mr Einstein dans S2 ?



Lors de la recopie des enregistrements de R dans S2 on a ignoré les attributs âge, ville et km, le couple (Einstein , marié) ne doit se retrouver qu'une seule fois car une relation est un ensemble et ses éléments sont tous distincts.

Jointure de deux relations

Soit Soient R et Q deux relations de domaine et de degré quelconques (degré(R) = n, degré(Q) = p), avec Domaine(R) ∩ Domaine(Q) = ∅ (pas d'attributs en communs).

soit R x Q leur produit cartésien de degré n + p et de domaine D union des domaines de R et de Q.
Soit un ensemble $(a_1, a_2, \ldots, a_{n+p})$ d'attributs du domaine D de R x Q.

La relation **joint**(R,Q) = **select** (Cond(a_1, a_2 , … , a_{n+p}) , R x Q), est appelée jointure de R et de Q (c'est donc une sélection de certains attributs sur le produit cartésien).

Une jointure couramment utilisée en pratique, est celle qui consiste en la sélection selon une condition d'égalité entre deux attributs, les personnes de "l'art relationnel" la dénomment alors l'**équi-jointure**.

Exemples de 2 jointures :

R :

nom	âge	Civilité
Einstein	45	marié
Lavoisier	41	marié
Planck	52	veuf

joint(R,Q)

Q :

ville	km
Paris	874
Rome	920

1°) S = **select** (km < 900 , R x Q)

nom	âge	Civilité	ville	km
Einstein	45	marié	Paris	874
Lavoisier	41	marié	Paris	874
Planck	52	veuf	Paris	874

2°) S = **select** (km < 900 **et** Civilité = veuf, R x Q)

nom	âge	Civilité	ville	km
Planck	52	veuf	Paris	874

Nous nous plaçons maintenant du point de vue pratique, non pas de l'administrateur de BD mais de l'utilisateur uniquement concerné par l'extraction des informations contenues dans une BD-R.

Un SGBD permet de gérer une base de données. A ce titre, il offre de nombreuses fonctionnalités supplémentaires à la gestion d'accès simultanés à la base et à un simple interfaçage entre le modèle logique et le modèle physique : il sécurise les données (en cas de coupure de courant ou autre défaillance matérielle), il permet d'accéder aux données de manière confidentielle (en assurant que seuls certains utilisateurs ayant des mots de passe appropriés, peuvent accéder à certaines données), il ne permet de mémoriser des données que si elles sont du type abstrait demandé : on dit qu'il vérifie leur intégrité (des données alphabétiques ne doivent pas être enregistrées dans des emplacements pour des données numériques,...)

Actuellement, une base de données n'a pas de raison d'être sans son SGBD. Aussi, on ne manipule que des bases de données correspondant aux SGBD qui les gèrent : il vous appartient de choisir le SGBD-R qui vous convient (il faut l'acheter auprès de vendeurs qui généralement, vous le fournissent avec une application de manipulation visuelle, ou bien utiliser les SGBD-R qui vous sont livrés gratuitement avec certains environnements de développement comme Delphi ou Visual Studio ou encore utiliser les produits gratuits comme mySql).

Lorsque l'on parle d'utilisateur, nous entendons l'application utilisateur, car l'utilisateur final n'a pas besoin de connaître quoique ce soit à l'algèbre relationnelle, il suffit que l'application utilisateur communique avec lui et interagisse avec le SGBD.

Une application doit pouvoir "parler" au SGBD : elle le fait par le moyen d'un langage de manipulation des données. Nous avons déjà précisé que la majorité des SGBD-R utilisait un langage relationnel ou de requêtes nommé SQL pour manipuler les données.

4. SQL et Algèbre relationnelle

Requête

Les requêtes sont des questions posées au SGBD, concernant une recherche de données contenues dans une ou plusieurs tables de la base.

Par exemple, on peut disposer d'une table définissant des clients (noms, prénoms, adresses, n°de client) et d'une autre table associant des numéros de clients avec des numéros de commande d'articles, et vouloir poser la question : quels sont les noms des clients ayant passé des commandes ?

Une requête est en fait, une instruction de type langage de programmation, respectant la norme SQL, permettant de réaliser un tel questionnement. L'exécution d'une requête permet d'extraire des données en provenance de tables de la base de données : ces données réalisent ce que l'on appelle une **projection** de champs (en provenance de plusieurs tables). Le résultat d'exécution d'une requête est une table constituée par les réponses à la requête.

Le SQL permet à l'aide d'instructions spécifiques de manipuler des données à l'intérieur des tables :

Instruction SQL	Actions dans la (les) table(s)
INSERT INTO <…>	Ajout de lignes
DELETE FROM <…>	Suppression de lignes
TRUNCATE TABLE <…>	Suppression de lignes
UPDATE <…>	Modification de lignes
SELECT <…>**FROM** <…>	Extraction de données

Ajout, suppression et modification sont les trois opérations typiques de la **mise à jour** d'une BD. L'extraction concerne la **consultation** de la BD.

Il existe de nombreuses autres instructions de création, de modification, de suppression de tables, de création de clefs, de contraintes d'intégrités référentielles, création d'index, etc… Nous nous attacherons à donner la traduction en SQL des opérateurs principaux de l'algèbre relationnelle que nous venons de citer.

Traduction en SQL des opérateurs relationnels

C'est l'instruction SQL "**SELECT** <…>**FROM** <…>" qui implante tous ces opérateurs. Tous les exemples utiliseront la relation R = TableComplete suivante et l'interpréteur SQL d'**Access** :

TableComplete : Table

nom	âge	civilité	ville	km
Einstein	45	marié	Paris	874
Mozart	32	marié	Rome	587
Lupin	42	Veuf	Paris	464
Einstein	45	marié	Venise	981
Gandhi	64	célibataire	Paris	258
Lavoisier	41	marié	Rome	124
Lupin	42	Veuf	Paris	608
Planck	52	Veuf	Rome	405

La relation initiale : R=TableComplete

Projection d'une relation R

S = **proj**(a_{k1} , a_{k2} … , a_{kp})

SQL : **SELECT DISTINCT** a_{k1} , a_{k2} … , a_{kp} **FROM** R

Instruction SQL version opérateur algèbre :
SELECT DISTINCT nom , civilité **FROM** Tablecomplete

Lancement de la requête SQL :

Requête51 : Requête Sélection

```
SELECT DISTINCT [nom], [civilité]
FROM TableComplete;
```

Le mot DISTINCT assure que l'on obtient bien une nouvelle relation.

Table obtenue après requête :

Requête51 : Requête Sélection

nom	civilité
Einstein	marié
Gandhi	célibataire
Lavoisier	marié
Lupin	Veuf
Mozart	marié
Planck	Veuf

Instruction SQL non relationnelle (tout même redondant) :
SELECT nom , civilité **FROM** Tablecomplete

Lancement de la requête SQL :

Requête51 : Requête Sélection

```
SELECT  [nom], [civilité]
FROM TableComplete;
```

Table obtenue après requête :

Requête51 : Requête Séle

nom	civilité
Einstein	marié
Mozart	marié
Lupin	Veuf
Einstein	marié
Gandhi	célibataire
Lavoisier	marié
Lupin	Veuf
Planck	Veuf

Remarquons dans le dernier cas **SELECT** nom , civilité **FROM** Tablecomplete que la table obtenue n'est qu'une extraction de données, mais qu'en aucun cas elle ne constitue une relation puisqu'une relation est un ensemble et que les enregistrements sont tous distincts!

Une autre projection sur la même table :

Instruction SQL version opérateur algèbre :
SELECT DISTINCT nom , ville **FROM** Tablecomplete

Lancement de la requête SQL :

Requête52 : Requête Sélection

```
SELECT DISTINCT [nom], [ville]
FROM TableComplete;
```

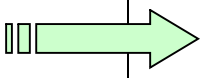


Table obtenue après requête :

Requête52 : Requête Sél

	nom	ville
	Einstein	Paris
	Einstein	Venise
	Gandhi	Paris
	Lavoisier	Rome
	Lupin	Paris
	Mozart	Rome
▶	Planck	Rome

Sélection-Restriction

S = **select** (Cond(a_1, a_2 , … , a_n) , R)
SQL : **SELECT** * **FROM** R **WHERE** Cond(a_1, a_2 , … , a_n)
*Le symbole * signifie toutes les colonnes de la table (tous les attributs du schéma)*

Instruction SQL version opérateur algèbre :
SELECT * FROM Tablecomplete **WHERE** âge > 42 **AND** ville = Paris

Lancement de la requête SQL :

TableComplete Requête : Requête Sélection

```
SELECT * FROM TableComplete
WHERE [âge]>42 AND [ville]="Paris";
```

Table obtenue après requête :

TableComplete Requête : Requête Sélection

	nom	âge	civilité	ville	km
	Einstein	45	marié	Paris	874
	Gandhi	64	célibataire	Paris	258

On a sélectionné toutes les personnes de plus de 42 ans ayant séjourné à Paris.

Combinaison d'opérateur projection-restriction

Projection distincte et sélection :
SELECT DISTINCT nom , civilité, âge **FROM** Tablecomplete **WHERE** âge >= 45

Lancement de la requête SQL :

TableComplete Requête : Requête Sélection

```
SELECT DISTINCT [nom], [civilité], [âge]
FROM TableComplete
WHERE [âge]>=45 ;
```

Table obtenue après requête :

TableComplete Requête : Requête Sé

	nom	civilité	âge
	Einstein	marié	45
	Gandhi	célibataire	64
▶	Planck	Veuf	52

On a sélectionné toutes les personnes d'au moins 45 ans et l'on ne conserve que leur nom, leur civilité et leur âge.

Produit cartésien

S = **R x Q**
SQL : **SELECT** * **FROM** R , Q

Afin de ne pas présenter un exemple de table produit trop volumineuse, nous prendrons comme opérandes du produit cartésien, deux tables contenant peu d'enregistrements :

Personne : Table

nom	prénom	age	civilité
Einstein	Albert	45	marié
Lavoisier	Antoine	41	marié

Employeur : Table

nom	n°Insee
Université	12112472
Collège	25478550
Institut	54578559

Produit cartésien :
SELECT * FROM Employeur , Personne

Employeur Requête : Requête Sélection

Personne.nom	prénom	age	civilité	Employeur.nom	n°Insee
Einstein	Albert	45	marié	Université	12112472
Lavoisier	Antoine	41	marié	Université	12112472
Einstein	Albert	45	marié	Collège	25478550
Lavoisier	Antoine	41	marié	Collège	25478550
Einstein	Albert	45	marié	Institut	54578559
Lavoisier	Antoine	41	marié	Institut	54578559

Nous remarquons qu'en apparence l'attribut **nom** se retrouve dans le domaine des deux relations ce qui semble contradictoire avec l'hypothèse "Domaine(R) ∩ Domaine(Q) = ∅ (pas d'attributs en communs). En fait ce n'est pas un attribut commun puisque les valeurs sont différentes, il s'agit plutôt de deux attributs différents qui ont la même identification. Il suffit de préfixer l'identificateur par le nom de la relation (Personne.nom et Employeur.nom).

Combinaison d'opérateurs : projection - produit cartésien

SELECT Personne.nom , prénom, civilité, n°Insee **FROM** Employeur , Personne

On extrait de la table produit cartésien uniquement 4 colonnes :

Employeur Requête : Requête Sélection

nom	prénom	civilité	n°Insee
Einstein	Albert	marié	12112472
Lavoisier	Antoine	marié	12112472
Einstein	Albert	marié	25478550
Lavoisier	Antoine	marié	25478550
Einstein	Albert	marié	54578559
Lavoisier	Antoine	marié	54578559

Intersection, union, différence,

S = **R** ∩ **Q**
SQL : **SELECT * FROM** R **INTERSECT SELECT * FROM** Q

S = **R** ∪ **Q**
SQL : **SELECT * FROM** R **UNION SELECT * FROM** Q

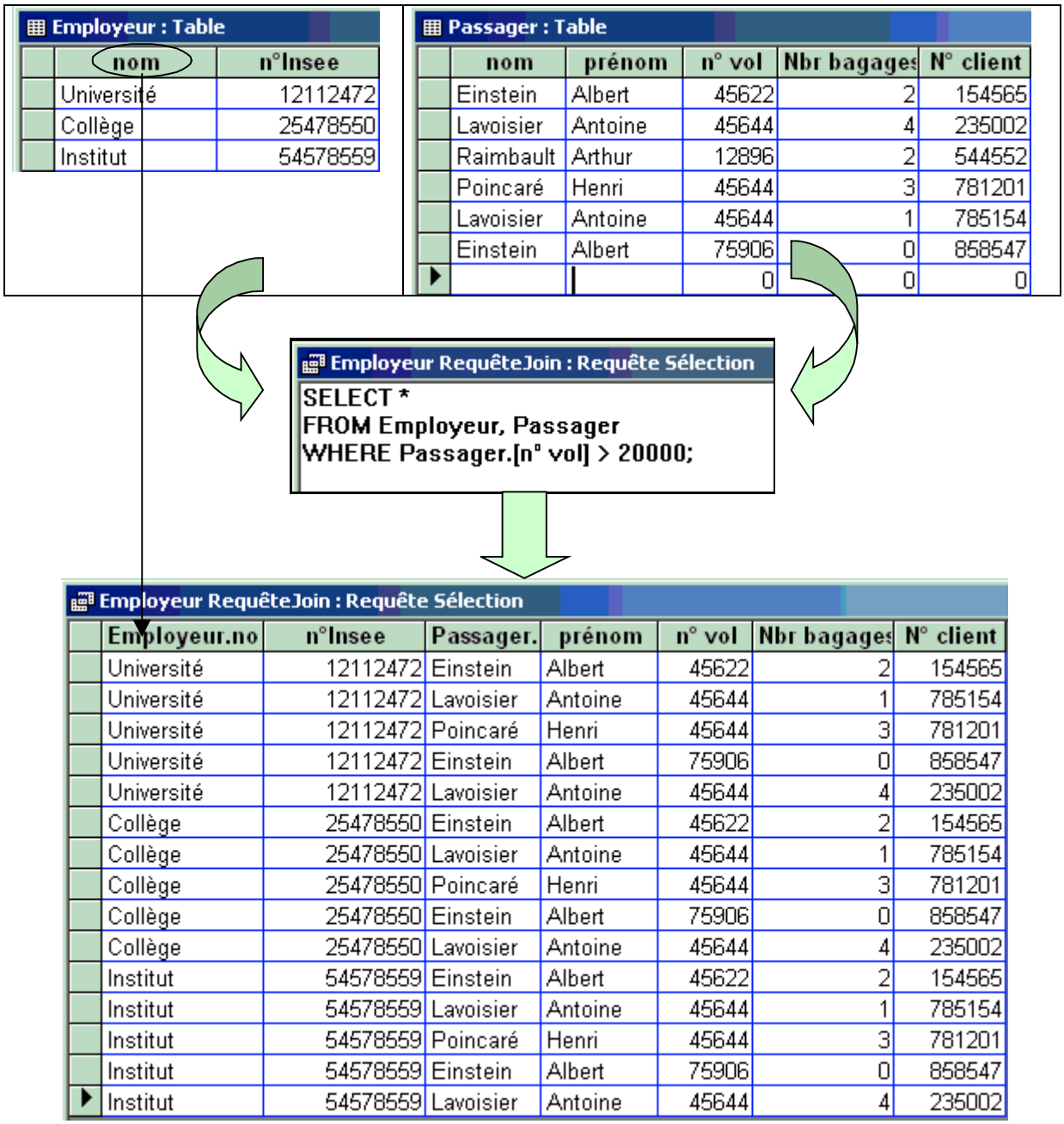
S = **R** – **Q**
SQL : **SELECT * FROM** R **MINUS SELECT * FROM** Q

Jointure de deux relations

Soient R et Q deux relations de domaine et de degré quelconques (degré(R) = n, degré(Q) = p), avec Domaine(R) ∩ Domaine(Q) = ∅ (pas d'attributs en communs).

La jointure **joint**(R,Q) = **select** (Cond(a_1, a_2 , … , a_{n+p}) , R x Q).

SQL : **SELECT** * **FROM** R ,Q **WHERE** Cond(a_1, a_2 , … , a_{n+p})



Remarque pratique importante

Le langage SQL est plus riche en fonctionnalités que l'algèbre relationnelle. En effet SQL intègre des possibilités de calcul (numériques et de dates en particulier).

Soit une table de tarifs de produit avec des prix hors taxe:

Tarifs : Table

Article	PrixHT
chaise	10,00 €
table	100,00 €
Téléviseur	1 000,00 €

1°) Usage d'un opérateur multiplicatif : calcul de la nouvelle table des tarifs TTC abondés de la TVA à 20% sur le prix hors taxe.

Tarifs : Table

Article	PrixHT
chaise	10,00 €
table	100,00 €
Téléviseur	1 000,00 €

Requête SQL : Tarifs TTc

Tarifs TTc : Requête Sélection

```
SELECT Article , PrixHT*1.20 AS PrixTTC
FROM Tarifs;
```

Tarifs TTc : Requête Sélection

Article	PrixTTC
chaise	12
table	120
Téléviseur	1200

2°) Usage de la fonction intégrée SUM : calcul du total des prix HT.

Tarifs : Table

Article	PrixHT
chaise	10,00 €
table	100,00 €
Téléviseur	1 000,00 €

Requête SQL : Total HT

Total HT : Requête Sélection

```
SELECT SUM(PrixHT) AS Total
FROM Tarifs;
```

Total HT : Requête Sélection

Total
1 110,00 €

5. Exemple de communication entre Delphi et les BD

Chaque langage permet d'une façon plus ou moins simple, d'accéder à une BD. Ce qui revient à dire que chaque constructeur de langage met en place sa propre solution pour qu'une application s'interface avec un SGBD à travers des commandes SQL (on appelle cela une solution propriétaire). Du côté de l'utilisateur, l'application rend le plus transparent possible la navigation sous SQL.

Delphi de Borland propose et a proposé des solutions propriétaires qui évoluent au cours du temps et des versions. En outre, les SGBD évoluent eux aussi, ce qui rend quasi impossible un choix simple et portable d'une solution standard. Enfin, il y a une grande différence de fonctionnalités entre un SGBD comme Access et Oracle ce qui ne va pas dans le sens de l'interopérabilité.

Principe général à adopter lorsque l'on veut écrire une application qui accède à une base déjà existante. Voir dans la documentation du langage si la version du SGBD est supportée par la version du langage Delphi que vous comptez utiliser. Ensuite, la documentation propose une stratégie de communication (utilisation de certaines classes) plus adaptée au SGBD.

Il existe parmi les choix proposés par Delphi, un moyen d'accès qui se nomme le BDE (Borland Data Engine) qui, bien qu'il soit en fin de vie est présent dans les versions 5, 6 et 7 professionnelle ou architecte, entreprise client-serveur de Delphi. Les versions perso ne contiennent rien qui permet d'accéder aux bases de données. O.Dahan et P.Thot dans leur excellent ouvrage "Applications professionnelles Delphi 7 studio" parlent du BDE en tant qu'outil pour le monde professionnel : *"..le BDE n'est plus une solution d'avenir. Toutefois enrichi par des années d'évolution, ce produit sait rendre des services qui ne peuvent être satisfaits par les autres solutions disponibles. Connaître ces avantages peut vous sauver la mise dans certaines situations.*"

Nous supposerons que vous disposez d'Access97 (contenu dans Office pro97), au minimum une base Access déjà existante et une version de Delphi contenant le BDE.

L'organisation physique d'une base de données dépend du SGBD qui la gère : certaines bases sont physiquement représentées par un seul fichier dans lequel sont stockées toutes les tables, requêtes,... (Microsoft Access fait cela par exemple), d'autres sont physiquement représentées par un dossier sur disque, et les tables, requêtes,... sont stockées sous la forme de fichiers séparés dans ce dossier.

5.1 Principe du BDE

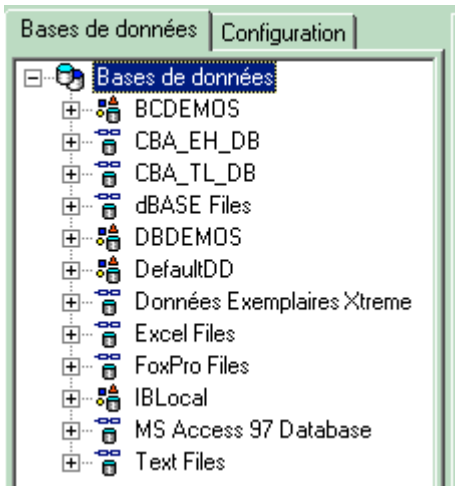
Vous pouvez accéder au BDE par le Panneau de Configuration en lançant "Administrateur BDE". Le BDE permet la communication de Delphi avec les Bases de données. Borland utilise un ensemble de DLL dans lesquelles sont codées les SGBD.

Pour pouvoir gérer des SGBD d'autres types (par exemple Microsoft Access) vous devez avoir acheté les DLL nécessaires (par exemple si vous avez acheté Microsoft Access, la DLL

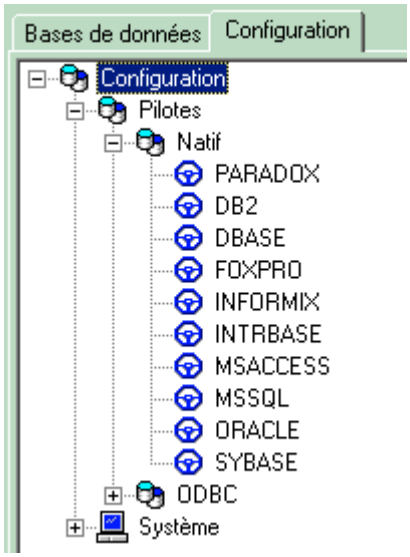
nécessaire est IDDA3532.DLL, qui est automatiquement installée sous Windows lorsque vous installez Access, et le BDE pourra l'utiliser).

Il y a 2 sections dans le BDE :

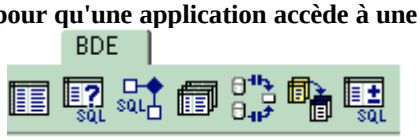
- Base de données : dans cette section, on déclare des alias de bases de données (c'est à dire des noms fictifs pour une base de données physique), et on indique sa position physique sur disque, et le type de SGBD qui peut la gérer (le type de pilote dans notre cas, qui est associé à un SGBD).



- Configuration : c'est là que les noms des pilotes sont déclarés, et associés physiquement aux DLL qui permettent de gérer le SGBD associé à un type donné.

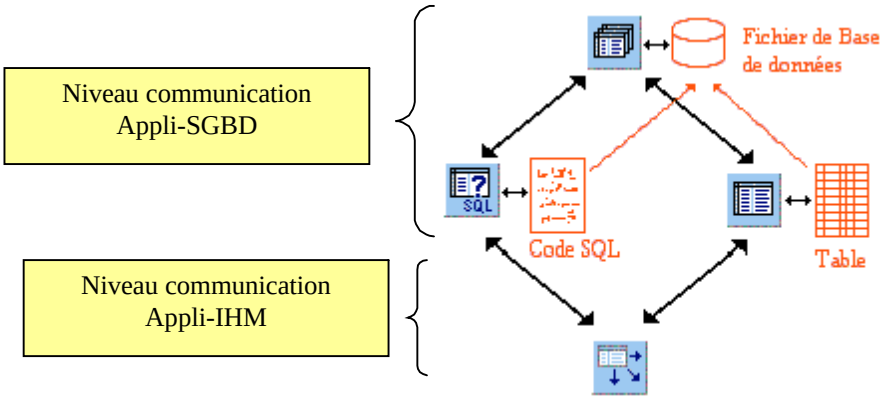


5.2. Les classes Delphi de communication avec les bases de données

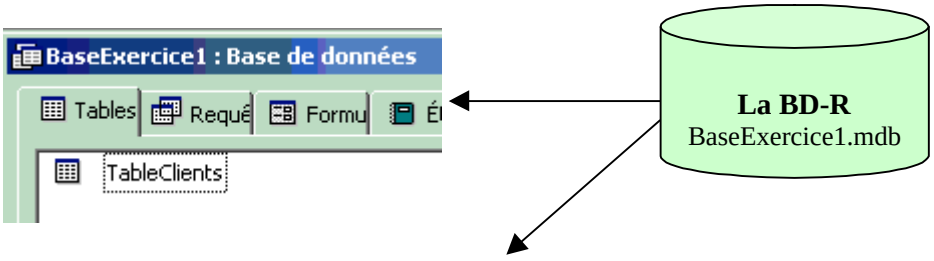
Trois classes fondamentales pour qu'une application accède à une BD *onglet BDE de la palette :* 	
TQuery	*Permet d'effectuer des requêtes dans une BD*
TDataBase	*Permet de connecter physiquement une application Delphi à un fichier de BD*
TTable	*Permet d'accéder à une table de la BD*

Une classe pour communiquer avec les classes précédentes et l'utilisateur *onglet ControlesDB de la palette :* 	
TDataSource	*Permet aux composants de navigations (ceux de l'onglet "ControlesDB" de la palette de Delphi, d'accéder aux objets fournis par un Ttable ou un TQuery*

Schéma de communication dans une application accédant à une BD :



Exemple : soit une BD Access nommée BaseExercice1.mdb, contenant une table nommée TableClients avec 4 enregistrements :

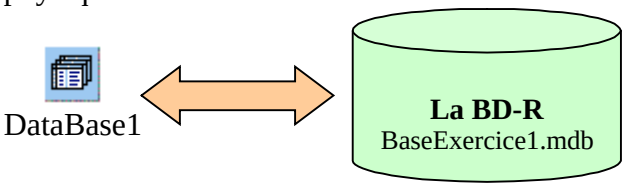


La TableClients est composée de 4 enregistrements :

TableClients : Table

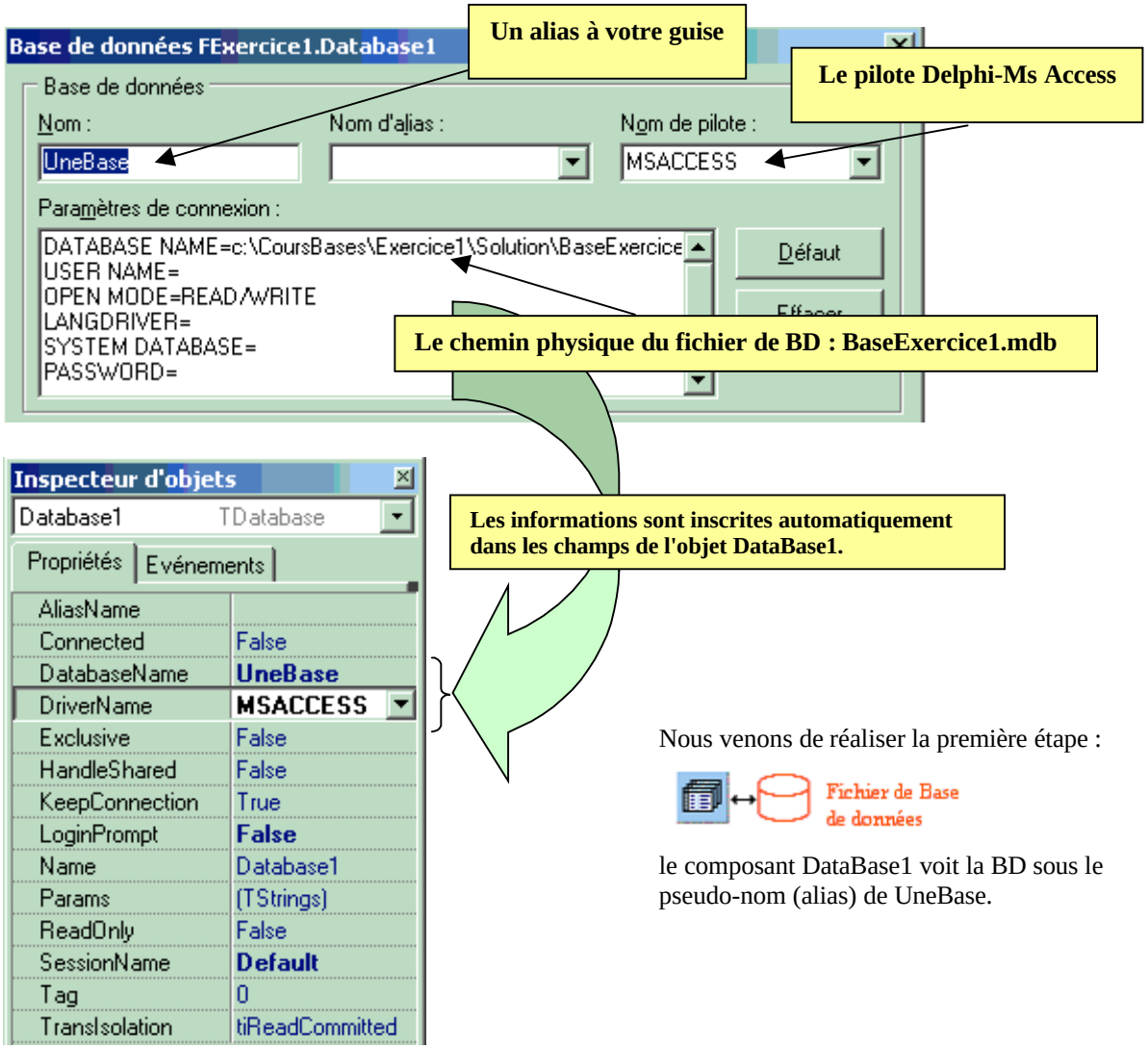
Nom	Prénom	Adresse	CP	Ville
ALKARTI	Joseph	3 route des Merles	54023	Cartes
BOULAIS	Sylvie	5 allée des Pins	05123	Teluies
GANZ	Karl	27 avenue St Laurent	78129	Isyfes
PROUDON	Amélie	105 rue des Platanes	45123	Mentisse

Dans l'application Delphi, il faut déposer un composant TDataBase que nous relions à la BD physique :



On double clique sur le composant DataBase1et on obtient une fenêtre de dialogue permettant la connexion physique.

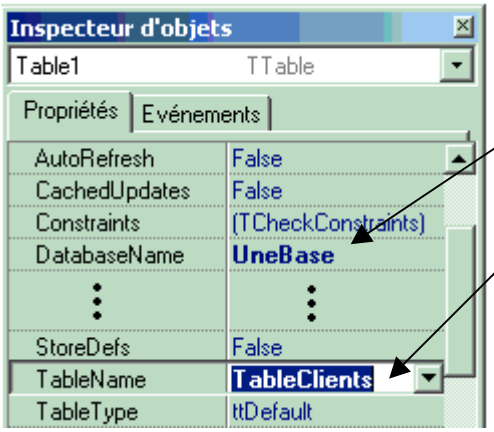
La fenêtre de dialogue obtenue :



Nous venons de réaliser la première étape :

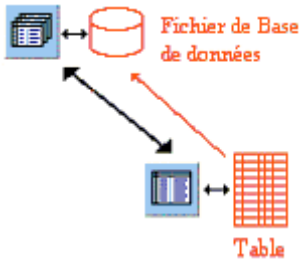
le composant DataBase1 voit la BD sous le pseudo-nom (alias) de UneBase.

Dans la seconde étape du travail, nous devons donner la possibilité à notre application de travailler avec une table de la BD, en l'occurrence ici, avec la table TableClients de **BaseExercice1.mdb**, nous déposons un composant **Table1** de type TTable pour cette opération :



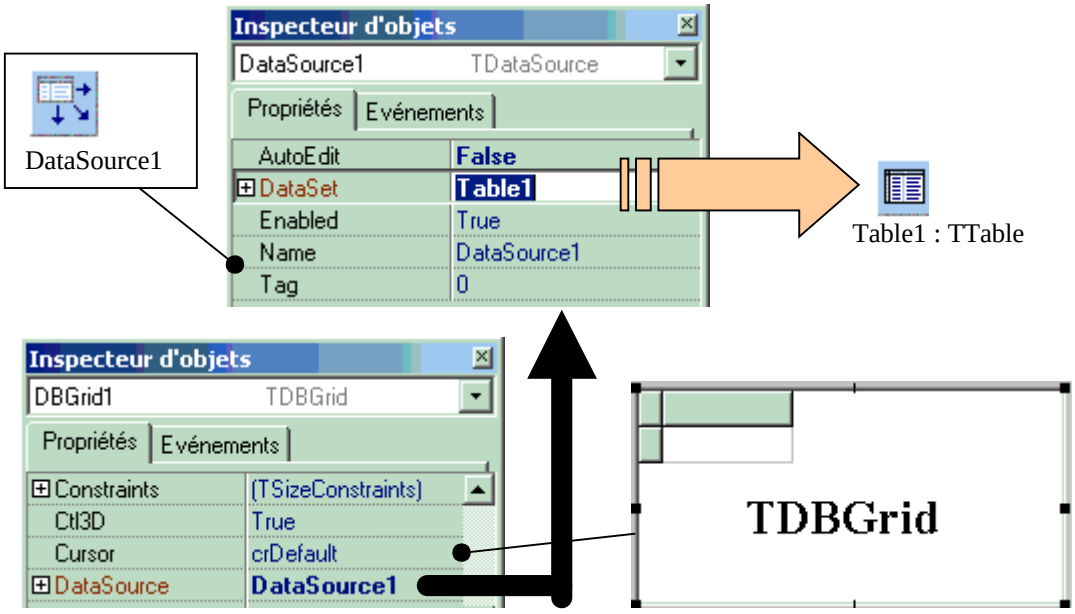
Nous indiquons ici que la BD utilisée est BaseExercice1.mdb **à travers son alias** UneBase **et que la table à utiliser dans la BD se nomme** TableClients**.**

Nous venons de réaliser la seconde étape :

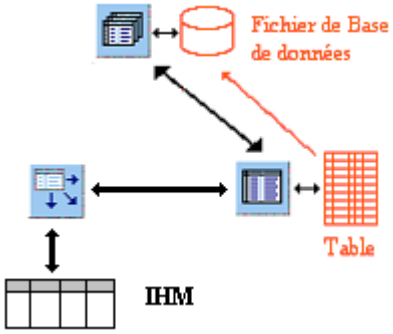


le composant **Table1** accède à la table TableClients de la BD.

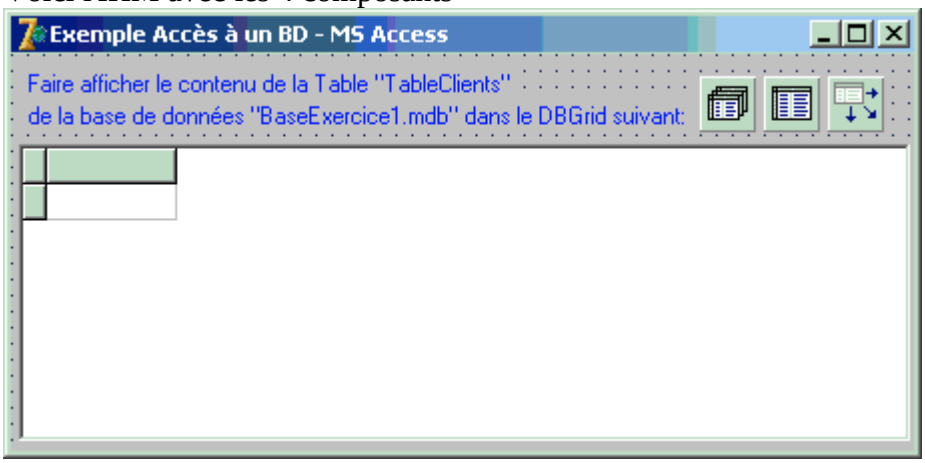
Nous terminons par la partie IHM de l'application : nous utilisons un composant de navigation TDBGrid qui affiche et manipule les enregistrements d'une table dans une grille tabulaire. Il n'est pas directement connecté au composant Table1, nous avons vu que c'est la classe TDataSource qui fait la liaison entre le composant d'IHM (ici nous avons choisi TDBGrid) et le TTable connecté à la table TableClients dans la BD :



Nous avons fini le processus de mise en place de la connexion de l'application et de la navigation dans la table TableClients de la BD :

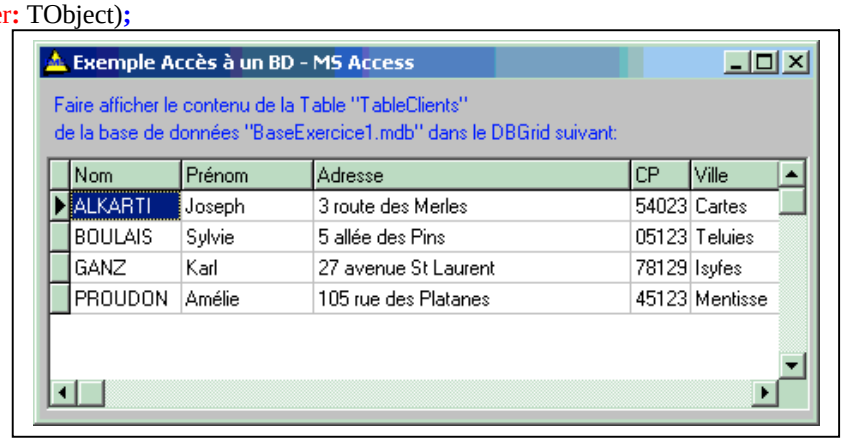


Voici l'IHM avec les 4 composants



Code source Delphi 5 , 6, 7

```
unit uFExercice1;

 interface

 uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, DBCtrls, Grids, DBGrids, StdCtrls, Mask, DBTables, Db;

 type
  TFExercice1 = class(TForm)
  DBGrid1: TDBGrid;
  Table1: TTable;
  DataSource1: TDataSource;
  Database1: TDatabase;
  Label1: TLabel;
  Label2: TLabel;
  procedure FormCreate(Sender: TObject);
 private
 { Déclarations privées }
 public
 { Déclarations publiques }
end;

 var
  FExercice1: TFExercice1;
  App_Path:string;

implementation

 {$R *.DFM}
```




```
 procedure TFExercice1.FormCreate(Sender: TObject);
 begin
  App_Path:=extractfilepath(application.exename);
  DataBase1.params[0]:='DATABASE NAME='+App_Path+'BaseExercice1.mdb';
  //paramètre dynamiquement le chemin de la base
  //de données (les lignes vides de paramètre doivent
  //déjà exister)
  //ou alternativement à la ligne précédente:
  {DataBase1.params.values['DATABASE NAME']:=App_Path+'BaseExercice1.mdb';}
```

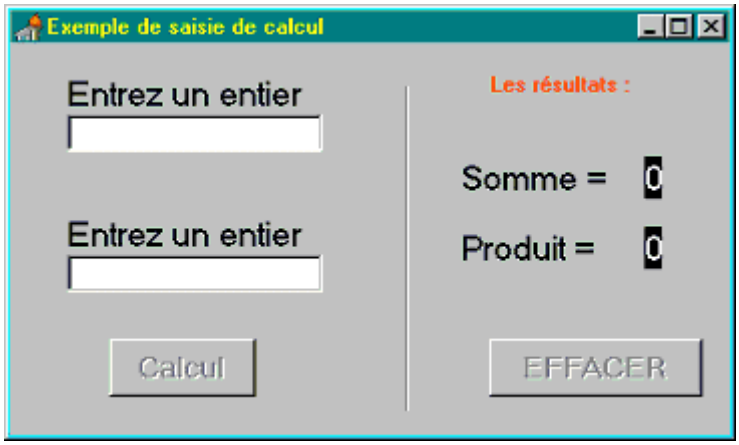
```
 DataBase1.connected:=true;  //connexion sur la base de données
 //(évidemment il faut d'abord paramétrer
 //correctement le composant DataBase1)
 Table1.open; //Ouverture de la table attachée au composant Table1
              //(il faut avoir bien paramétré le composant Table1auparavant
end;

end.
```

Exercices chapitre 7

Ex-1 : pilotage récapitulatif

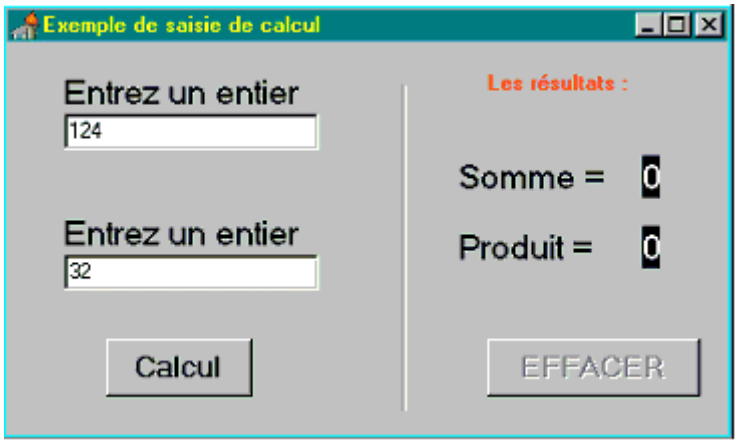
Nous voulons construire une interface permettant la saisie par l'utilisateur de deux entiers et l'autorisant à effectuer leur somme et leur produit uniquement lorsque les entiers sont entrés tous les deux. Aucune sécurité n'est apportée **pour l'instant** sur les données.

Voici l'état visuel de l'interface au lancement du programme :

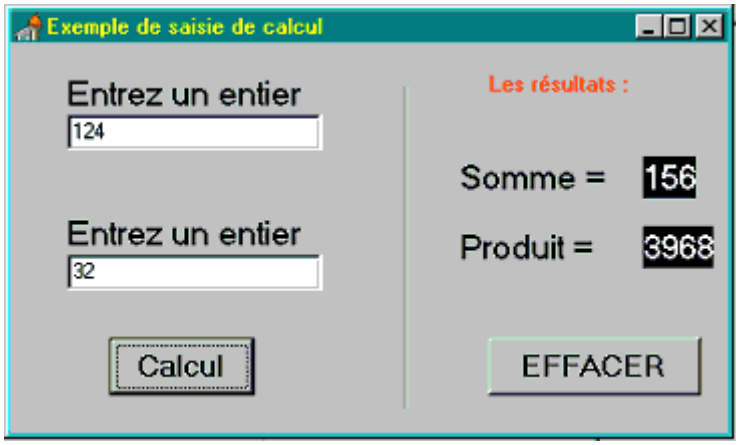


2 zones de saisie
2 boutons d'actions
2 zones d'affichages des résultats

Dès que les deux entiers sont entrés, le bouton **Calcul** est activé:



Lorsque l'on clique sur le bouton **Calcul**, le bouton **EFFACER** est activé et les résultats s'affichent dans leurs zones respectives :



Un clic sur le bouton **EFFACER** ramène l'interface à l'état initial.

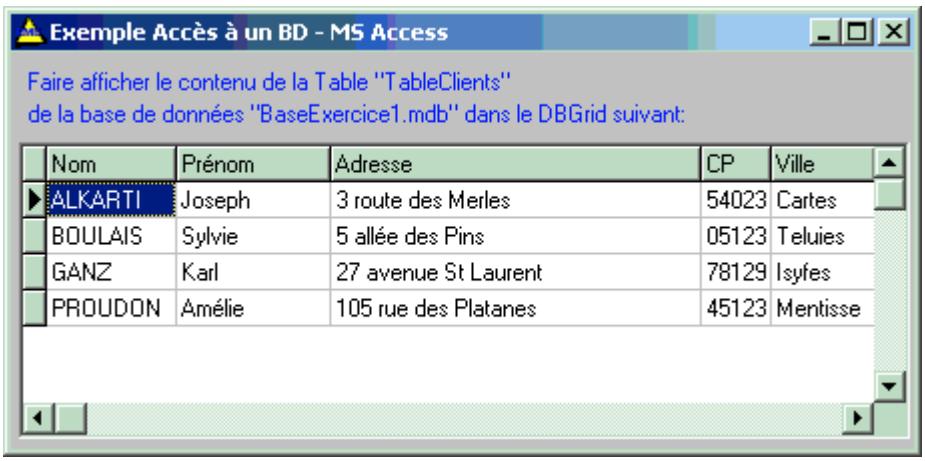
Exercices BD et Delphi

On donne la BD nommée BaseExercice.mdb et la relation TableClients ci-dessous :

TableClients : Table

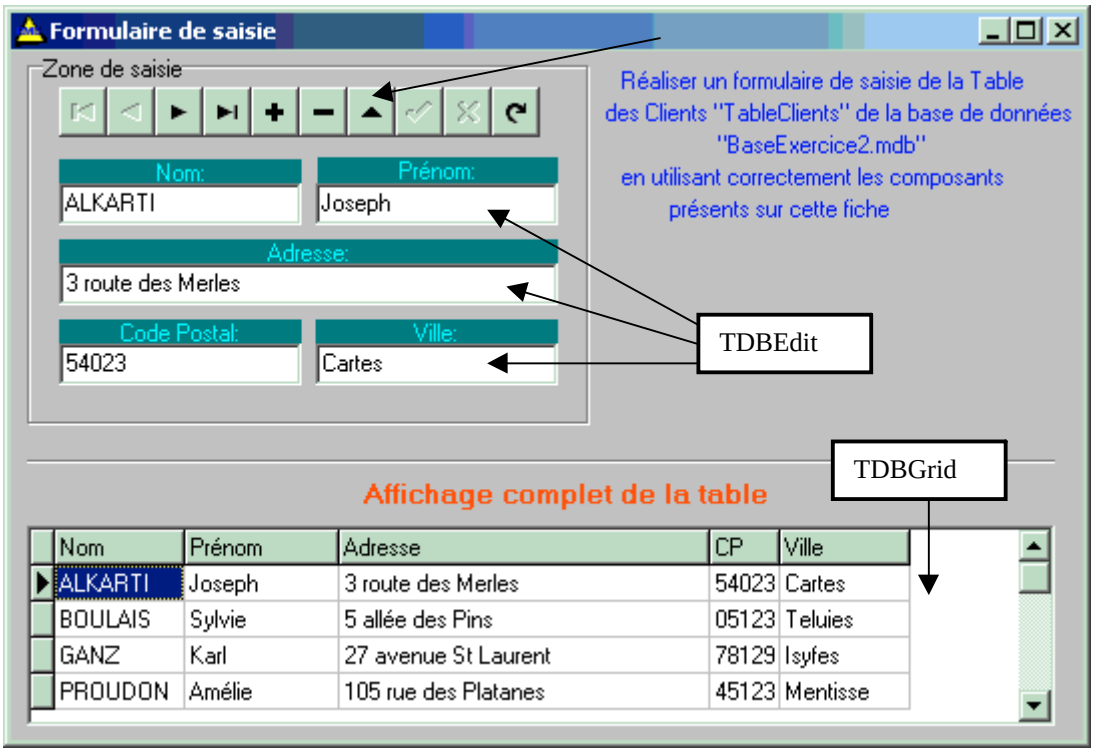
Nom	Prénom	Adresse	CP	Ville
ALKARTI	Joseph	3 route des Merles	54023	Cartes
BOULAIS	Sylvie	5 allée des Pins	05123	Teluies
GANZ	Karl	27 avenue St Laurent	78129	Isyfes
PROUDON	Amélie	105 rue des Platanes	45123	Mentisse

Ex-2 : construire une IHM Delphi permettant d'afficher la table TableClients dans un TDBGrid , comme ci-desssous :



Ex-3 : construire une IHM Delphi de formulaire de saisie la table TableClients en utilisant les composant TDBEdit, TDBNavigator, TDBGrid , comme ci-desssous ::

TDBNavigator

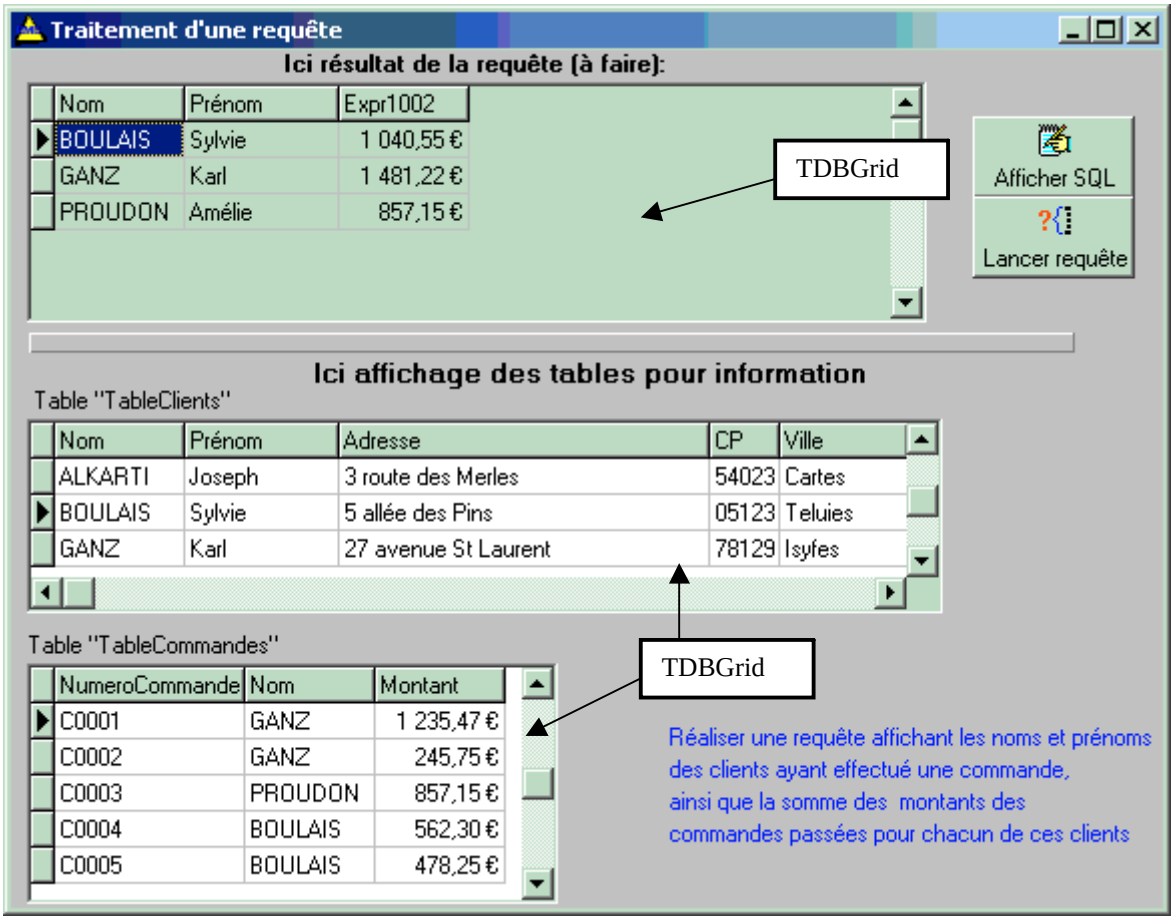


Ex-4 : construire une IHM Delphi de lancement d'une requête sur la table TableClients et une autre TableCommandes en utilisant le composant Tquery.

La commande SQL à lancer est la suivante :

```
SELECT TableClients.Nom, TableClients.Prénom, Sum(TableCommandes.Montant)
FROM TableClients INNER JOIN TableCommandes ON TableClients.Nom = TableCommandes.Nom
GROUP BY TableClients.Nom, TableClients.Prénom
```

IHM à programmer :



Ex-5 : Soit une BD contenant des informations de produits vendus dans un commerce. La BD contient deux tables, une table magasin :

Magasin : Table

CodeArticle	DésignArticle	QuantitéStock	SeuilAlerte
A0001	Bottes	157	20
A0002	Brosse poil dur	56	60
A0003	Savon doux	78	50
A0004	Trousse secours	6	10
A0005	Cirage noir	46	15
A0006	Ampoule 40W	13	20

Et une table PrixArticles:

PrixArticles : Table

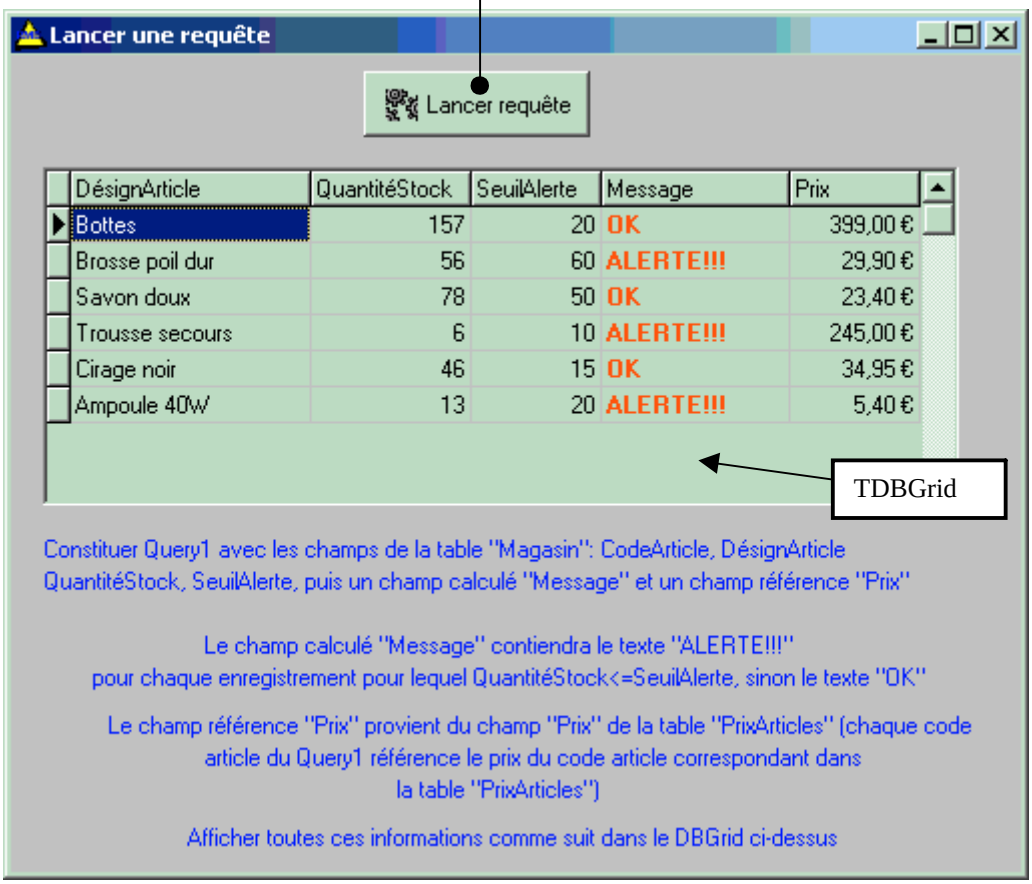
CodeArticle	Prix
A0001	399,00 F
A0002	29,90 F
A0003	23,40 F
A0004	245,00 F
A0005	34,95 F
A0006	5,40 F

Le champ CodeArticle est une clef primaire de chaque table

La commande SQL à lancer est la suivante :

```
SELECT Magasin.CodeArticle, Magasin.DésignArticle, Magasin.QuantitéStock, Magasin.SeuilAlerte
FROM Magasin  ORDER BY Magasin.CodeArticle
```

Construire une IHM Delphi de lancement de la requête ci-dessus sur la table Magasin en utilisant le composant TQuery.



Ex-6 : Expressions arithmétiques et arbre binaire : nous détaillons dans cet exemple, la démarche de création d'un programme ayant pour but de construire et de parcourir un arbre de syntaxe abstrait des expressions arithmétiques fondées sur une C-grammaire.

Nous procédons par étapes séparées afin de bien faire comprendre le processus de conception du programme. La première étape consiste à écrire un programme d'analyse des expressions permettant de repérer les variables représentées par des lettres et les opérateurs. Il ne s'agit pas ici d'un processus lexical, mais d'un processus syntaxique déjà abordé dans le cours.

Une fois le programme en version analyse syntaxique élaboré, nous passons à l'étape de génération de l'arbre binaire abstrait représentant l'expression en cours d'analyse.

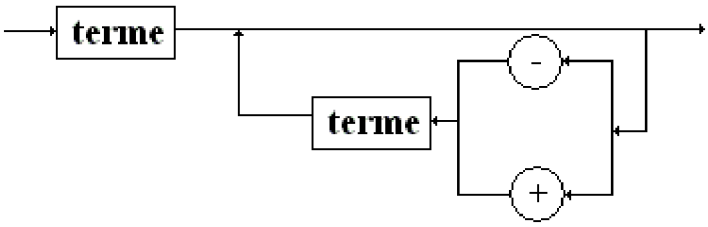
Grammaire des expressions utilisée :

Vn = {<expr>, <terme>, <facteur>, <caractère> }
Vt = { **+** ,**-** , **(** , **)** ,***** , **/** , **a** ,..., **z**}

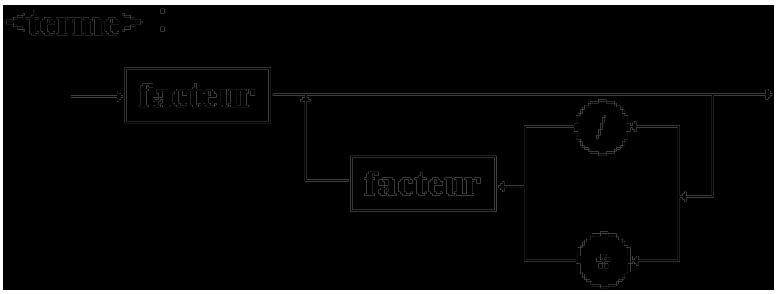
Axiome : <expr>

Règles :

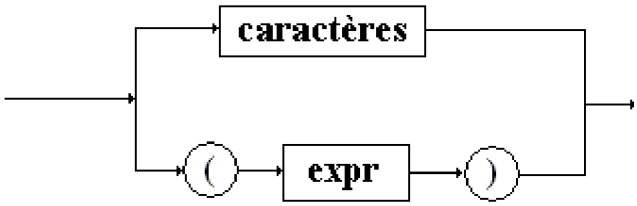
<expr> :



<terme> :



<facteur> :



<caractères> ::= a| b | c | ..... | z

Solution des exercices

Ex-1 : pilotage récapitulatif- solution complète détaillée.

Graphe événementiel complet de l'interface

Avec comme conventions sur les messages :

X_1 = exécuter les calculs sur les valeurs entrées.

X_2 = exécuter l'effacement des résultats.

M_1 = message à l'edit1 de tout effacer.

M_2 = message à l'edit2 de tout effacer.

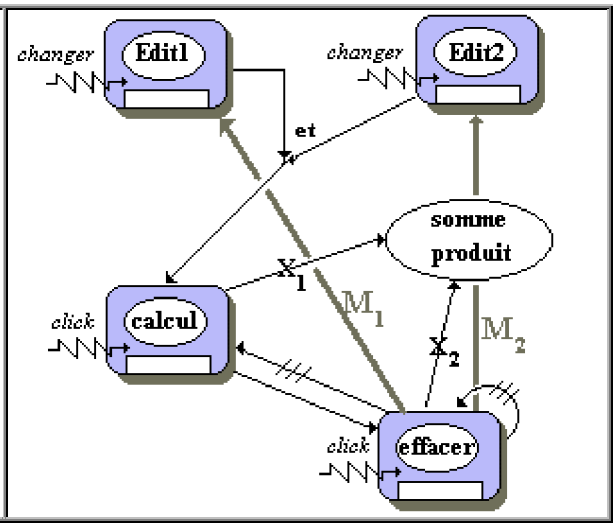


Table des actions événementielles associées au graphe

Changer Edit1	Calcul activable (si changer Edit2 a eu lieu)
Changer EDIT2	Calcul activable (si changer Edit1 a eu lieu)
Clic CALCUL	Exécuter X1 EFFACER activable Afficher les résultats
Clic EFFACER	EFFACER désactivé CALCUL désactivé Message M1 Message M2

Table des états initiaux des objets sensibles aux événements

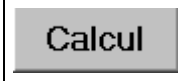
Edit1	***activé***
Edit2	***activé***
Buttoncalcul	***désactivé***
Buttoneffacer	***désactivé***

- Nous voulons disposer d'un bouton permettant de lancer le calcul lorsque c'est possible et d'un bouton permettant de tout effacer. Les deux boutons seront désactivés au départ.
- Nous choisissons 6 objets visuels : deux TEdit, **Edit1** et **Edit2** pour la saisie ; deux Tbutton, **ButtonCalcul** et **Buttoneffacer** pour les changements de plans d'action ; deux Tlabel **LabelSomme** et **LabelProduit** pour les résultats

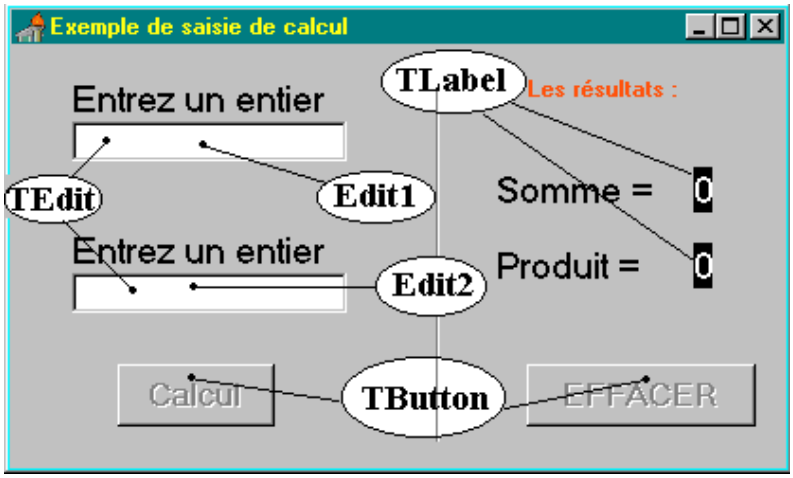
Les objets visuels Delphi choisis

Edit1: TEdit;
Edit2: TEdit;
LabelSomme: TLabel;
LabelProduit: TLabel;

ButtonCalcul: TButton;



Buttoneffacer: TButton;



Construction progressive du gestionnaire d'événement Onchange

Nous devons réaliser une synchronisation des entrées dans Edit1 et Edit2 : le déverrouillage (activation) du bouton " ButtonCalcul " ne doit avoir lieu que lorsque Edit1 et Edit2 contiennent des valeurs. Ces deux objets de classe TEdit, sont sensibles à **l'événement OnChange** qui indique que le contenu du champ **text** a été modifié, l'action est indiquée dans le graphe événementiel sous le vocable " *changer* ".

La synchronisation se fera à l'aide de deux drapeaux binaires (des booléens) qui seront levés chacun séparément par les TEdit lors du changement de leur contenu. Le drapeau Som_ok est levé par l'Edit1, le drapeau Prod_ok est levé par l'Edit2.

Implantation : deux champs privés booléens

```
  Som_ok , Prod_ok : boolean;
```

Le drapeau Som_ok est levé par l'Edit1 lors du changement du contenu de son champ text, sur l'apparition de l'événement OnChange, il en est de même pour le drapeau Prod_ok et l'Edit2 :

Implantation n°1 du gestionnaire de OnChange :

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  Som_ok:=true; // drapeau de Edit1 levé
end;
```

```
procedure TForm1.Edit2Change(Sender: TObject);
begin
  Prod_ok:=true; // drapeau de Edit2 levé
end;
```

Une méthode privée de test vérifiera que les deux drapeaux ont été levés et lancera l'activation du **ButtonCalcul**.

```
procedure TForm1.TestEntrees;
{les drapeaux sont-ils levés tous les deux ?}
begin
 if  Prod_ok and Som_ok then
    ButtonCalcul.Enabled:=true  //  si oui: le bouton calcul est activé
end;
```

Nous devons maintenant tester lorsque nous levons un drapeau si l'autre n'est pas déjà levé. Cette opération s'effectue dans les gestionnaires de l'événement OnChange de chacun des Edit1 et Edit2 :

Implantation n°2 du gestionnaire de OnChange :

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  Som_ok:=true; // drapeau de Edit1 levé
  TestEntrees;
end;
```

```
procedure TForm1.Edit2Change(Sender: TObject);
begin
  Prod_ok:=true; // drapeau de Edit2 levé
  TestEntrees;
end;
```

Construction des gestionnaires d'événement Onclick

Nous devons gérer l'événement clic sur le bouton calcul qui doit calculer la somme et le produit, placer les résultats dans les deux Tlabels et activer le Buttoneffacer.

Implantation du gestionnaire de OnClick du ButtonCalcul :

```
procedure TForm1.ButtonCalculClick(Sender: TObject);
var S , P : integer;
begin
    S:=strtoint(Edit1.text); // transtypage : string à integer
    P:=strtoint(Edit2.text); // transtypage : string à integer
    LabelSomme.caption:=inttostr(P+S);
    LabelProduit.caption:=inttostr(P*S);
    Buttoneffacer.Enabled:=true // le bouton effacer est activé
end;
```

Nous codons une méthode privée dont le rôle est de réinitialiser l'interface à son état de départ indiqué dans la table des états initiaux :

Edit1	***Activé***
Edit2	***Activé***
Buttoncalcul	***Désactivé***
Buttoneffacer	***désactivé***

```
procedure TForm1.RAZTout;
begin
   Buttoneffacer.Enabled:=false; //le bouton effacer se désactive
   ButtonCalcul.Enabled:=false; //le bouton calcul se désactive
   LabelSomme.caption:='0'; // RAZ valeur somme affichée
   LabelProduit.caption:='0'; // RAZ valeur produit affichée
   Edit1.clear; // message M1
   Edit2.clear; // message M2
   Prod_ok:=false; // RAZ drapeau Edit2
   Som_ok:=false; // RAZ drapeau Edit1
end;
```

Nous devons gérer l'événement click sur le Buttoneffacer qui doit remettre l'interface à son état initial (par appel à la procédure RAZTout) :

Implantation du gestionnaire de OnClick du Buttoneffacer:

```
procedure TForm1.ButtoneffacerClick(Sender: TObject);
begin
  RAZTout;
end;
```

Au final, lorsque l'application se lance et que l'interface est céée nous devons positionner tous les objets à l'état initial (nous choisissons de lancer cette initialisation sur l'événement **OnCreate** de création de la fiche):

Implantation du gestionnaire de OnCreate de la fiche Form1:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  RAZTout;
end;
```

Si nous essayons notre interface nous constatons que nous avons un problème de sécurité à deux niveaux dans notre saisie :

- Le premier niveau est celui des corrections apportées par l'utilisateur (effacement de chiffres déjà entrés) et la synchronisation avec l'éventuelle vacuité d'au moins un des champs text après une telle modification : en effet l'utilisateur peut effacer tout le contenu d'un " Edit " et lancer alors le calcul, provoquant ainsi des erreurs.

- Le deuxième niveau classique est celui du transtypage incorrect, dans le cas où l'utilisateur commet des fautes de frappe et rentre des données autres que des chiffres (des lettres ou d'autres caractères du clavier).

Améliorations de sécurité de premier niveau par plan d'action

Nous protégeons les calculs dans le logiciel, par un test sur la vacuité du champ text de chaque objet de saisie TEdit. Dans le cas favorable où le champ n'est pas vide, on autorise la saisie ; dans l'autre cas on désactive systématiquement le ButtonCalcul et l'on abaisse le drapeau qui avait été levé lors de l'entrée du premier chiffre, ce qui empêchera toute erreur ultérieure. L'utilisateur comprendra de lui-même que tant qu'il n'y a pas de valeur dans les entrées, le logiciel ne fera rien et on ne passera donc pas au plan d'action suivant (calcul et affichage). Cette amélioration s'effectue dans les gestionnaires d'événement OnChange des deux TEdit.

Implantation n°2 du gestionnaire de OnChange :

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
 if Edit1.text<' ' then // champ text non vide ok !
 begin
   Som_ok:=true;
   TestEntrees;
 end
 else
 begin
   ButtonCalcul.enabled:=false; // bouton désactivé
   Som_ok:=false; // drapeau de Edit1 baissé
 end
```

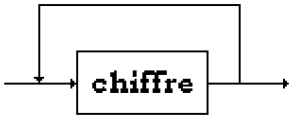
```
procedure TForm1.Edit1Change(Sender: TObject);
begin
 if Edit2.text<' ' then // champ text non vide ok !
 begin
   Prod_ok:=true;
   TestEntrees;
 end
 else
 begin
   ButtonCalcul.enabled:=false; //bouton désactivé
   Prod_ok:=false; // drapeau de Edit2 baissé
 end
```

end;	**end**;

On remarque que les deux codes précédents sont très proches, ils diffèrent par leTEdit et le drapeau auxquels ils s'appliquent. Il est possible de réduire le code redondant en construisant par exemple une méthode privée avec deux paramètres.

Améliorations de sécurité de second niveau par automate de filtrage

Nous pouvons améliorer cet état de la saisie des caractères chiffres en construisant un **analyseur de filtrage** qui ne conserve que les caractères valides tapés dans chaque TEdit. La syntaxe de l'entrée est fournie par le diagramme suivant :



Nous construisons un AEFD (**automate d'états finis**) reconnaissant ces chiffres et il nous servira de système de filtrage des caractères entrés.

L'AEFD de filtrage :

```
procedure Filtrage(entree :string;var resultat:string);
var i:integer;
   saisie :string;
   CarValides:set of char;
begin
 CarValides:=['0'..'9']; // les chiffres seulement
 saisie:=entree;
 if length(saisie)<0 then
 begin
  i:=1;
  while saisie[i] in CarValides do i:=i+1; // le premier caractère non juste
  if not(saisie[i] in CarValides) then delete(saisie,i,1);
 end;
 resultat:=saisie
end;
```

Implantation n°3 du gestionnaire de OnChange :

```
procedure TForm1.Edit1Change(Sender: TObject);
var sortie :string ;
begin
   Filtrage(Edit1.text,sortie);
   Edit1.text:=sortie;
   Som_ok:=true; // drapeau de Edit1 levé
   TestEntrees;
end;
```

```
procedure TForm1.Edit2Change(Sender: TObject);
var sortie :string ;
begin
   Filtrage(Edit2.text,sortie);
   Edit2.text:=sortie;
   Prod_ok:=true; // drapeau de Edit2 levé
   TestEntrees;
end;
```

Combinons le niveau 1 et le filtrage effectué lors de la saisie des entrées dans les gestionnaires d'événement OnChange des deux Tedit. Chaque changement du contenu du champ text d'un Edit (comme l'entrée d'un nouveau caractère) provoque le déclenchement de l'événement OnChange qui rejette alors les caractères non valides.

Implantation n°4 du gestionnaire de OnChange :

```
procedure TForm1.Edit1Change(Sender: TObject);
var sortie :string ;
begin
  Filtrage(Edit1.text,sortie);
  Edit1.text:=sortie;
  if Edit1.text<'' then// champs text non vide ok !
  begin
    Som_ok:=true;
    TestEntrees;
  end
  else
  begin
    ButtonCalcul.enabled:=false; // sinon bouton désactivé
    Som_ok:=false; // drapeau de Edit1 baissé
  end
end;
```

```
procedure TForm1.Edit2Change(Sender: TObject);
var sortie :string ;
begin
  Filtrage(Edit2.text,sortie);
  Edit2.text:=sortie;
  if Edit2.text<'' then // champs text non vide ok !
  begin
    Prod_ok:=true;
    TestEntrees;
  end
  else
  begin
    ButtonCalcul.enabled:=false; // sinon bouton désactivé
    Prod_ok:=false; // drapeau de Edit2 baissé
  end
end;
```

En combinant la sécurité du premier et du second niveau dans le gestionnaire d'événement OnChange nous obtenons un niveau acceptable de sécurisation de notre logiciel grâce à son interface. Il nous manque encore une protection sur les débordements de calcul (dépassement de l'intervalle sur les integer qui ne provoquent pas d'incidents mais induisent des résultats erronés) pour assurer une sécurité optimale de fiabilité.
Pour terminer la réalisation de cet exemple, nous pouvions aussi procéder à un autre genre de protection sans avoir à écrire un analyseur de filtrage (qui était dans ce cas un AEFD), en utilisant directement les facilités de protection fournies par les exceptions.

Améliorations de sécurité de second niveau par exceptions

Dans cette éventualité, l'on déporte le problème de la protection non plus sur le séquencement des plans d'actions mais au cœur même de l'action en protégeant directement le code où l'erreur se produit par un gestionnaire d'exception. Afin de présenter un traitement complet de l'exemple nous anticipons légèrement sur le chapitre sur la programmation défensive ; le lecteur pourra donc en première lecture sauter ce paragraphe s'il le désire et y revenir plus tard.

On fait exécuter le programme en provoquant volontairement l'erreur (on entre des lettres au lieu de chiffres) ; le logiciel signale la levée de l'exception EconvertError et nous programmons le gestionnaire associé à cette levée d'exception. Elle apparaît lorsque la fonction StrtoInt essaye de transtyper le champ text de l'Edit et échoue ; c'est donc cette ligne de code qui doit être protégée :

```
try
    S:= StrtoInt(edit1.text);
except on EconvertError do
begin
   Edit1.text:='0';
   S:=0
end
end;
```

Nous avons décidé ici de mettre 0 dans le champ text de l'Edit1 et de forcer la valeur de la variable de calcul S à 0. Ce traitement est identique pour la variable P à partir du champ text de l'Edit2. Ces deux traitements de protection sont effectués lors du click sur ButtonCalcul (traitement de l'erreur après saisie).

```
procedure TForm1.ButtonCalculClick(Sender: TObject);
var S,P:integer;
begin
 try
    S:= StrtoInt(edit1.text);
 except on EconvertError do
 begin
    Edit1.text:='0';
    S:=0
 end
end;
 try
    P:= StrtoInt(edit2.text);
 except on EconvertError do
 begin
    Edit2.text:='0';
    P:=0
 end
end;
 LabelSomme.caption:=inttostr(P+S);
 LabelProduit.caption:=inttostr(P*S);
 Buttoneffacer.enabled:=true // le bouton effacer est activé
end;{ButtonCalculClick}
```

Regroupement du code

Reprenons le code du premier niveau par plan d'action en le regroupant dans la méthode privée Autorise possédant deux paramètres, le premier est la référence d'un des deux TEdit, le second le drapeau booléen associé :

```
procedure TForm1.Autorise ( Ed : TEdit ; var flag : boolean );
begin
 if Ed.text<' ' then // champ text non vide ok !
 begin
   flag :=true;
   TestEntrees;
 end
 else
 begin
   ButtonCalcul.enabled:=false; //bouton désactivé
    flag:=false; // drapeau de Ed baissé
 end
end;
```

Nouvelle implantation n°2 du gestionnaire de OnChange :

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
 Autorise ( Edit1 , Som_ok );
end;
```

```
procedure TForm1.Edit2Change(Sender: TObject);
begin
 Autorise ( Edit2 , Prod_ok );
end;
```

Code final où tout est regroupé dans la classe TForm1

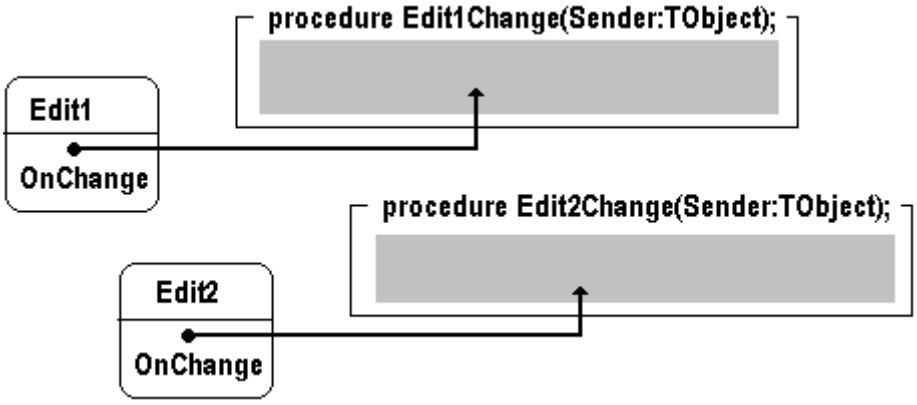
```
unit UFcalcul;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,   StdCtrls, ExtCtrls;

type
 TForm1= class ( TForm )
    Edit1: TEdit;
    Edit2: TEdit;
    ButtonCalcul: TButton;
    LabelSomme: TLabel;
    LabelProduit: TLabel;
    Buttoneffacer: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Edit1Change(Sender: TObject);
    procedure Edit2Change(Sender: TObject);
    procedure ButtonCalculClick(Sender: TObject);
    procedure ButtoneffacerClick(Sender: TObject);
  private   { Déclarations privées }
      Som_ok , Prod_ok : boolean;
      procedure TestEntrees;
      procedure RAZTout;
      procedure Autorise ( Ed : TEdit ; var flag : boolean );
  public   { Déclarations publiques }
  end;

implementation
{ ---------- Méthodes privées ----------}
```

```
procedure TForm1.TestEntrees;
{les drapeaux sont-ils levés tous les deux ?}
begin
  if  Prod_ok and Som_ok then
     ButtonCalcul.Enabled:=true  //  si oui: le bouton calcul est activé
end;

procedure TForm1.RAZTout;
begin
    Buttoneffacer.Enabled:=false; //le bouton effacer se désactive
    ButtonCalcul.Enabled:=false; //le bouton calcul se désactive
    LabelSomme.caption:='0'; // RAZ valeur somme affichée
    LabelProduit.caption:='0'; // RAZ valeur produit affichée
    Edit1.clear; // message M1
    Edit2.clear; // message M2
    Prod_ok:=false; // RAZ drapeau Edit2
    Som_ok:=false; // RAZ drapeau Edit1
end;

procedure TForm1.Autorise ( Ed : TEdit ; var flag : boolean );
begin
  if Ed.text<>' ' then // champ text non vide ok !
  begin
    flag :=true;
    TestEntrees;
  end
  else
  begin
    ButtonCalcul.enabled:=false; //bouton désactivé
     flag:=false; // drapeau de Ed baissé
  end
end;

{ ------------------- Gestionnaires d'événements --------------------}
procedure TForm1.FormCreate(Sender: TObject);
begin
   RAZTout;
end;

procedure TForm1.Edit1Change(Sender: TObject);
begin
   Autorise ( Edit1 , Som_ok );
end;

procedure TForm1.Edit2Change(Sender: TObject);
begin
   Autorise ( Edit2 , Prod_ok );
end;

procedure TForm1.ButtonCalculClick(Sender: TObject);
var S , P : integer;
begin
     S:=strtoint(Edit1.text); // transtypage : string à integer
     P:=strtoint(Edit2.text); // transtypage : string à integer
     LabelSomme.caption:=inttostr(P+S);
     LabelProduit.caption:=inttostr(P*S);
     Buttoneffacer.Enabled:=true // le bouton effacer est activé
end;
```

```
procedure TForm1.ButtoneffacerClick(Sender: TObject);
begin
  RAZTout;
end;

end.
```

Un gestionnaire d'événement centralisé grâce au :
- **polymorphisme d'objet**
- **pointeur de méthode**

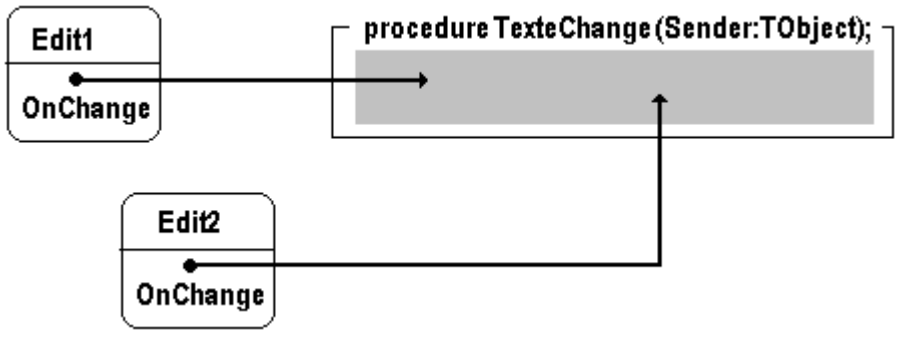
Dans le code précédent, chaque Edit possède son propre gestionnaire de l'événement OnChange :




```
procedure TForm1.Edit1Change(Sender: TObject);
begin
 Autorise ( Edit1 , Som_ok );
end;
```

```
procedure TForm1.Edit2Change(Sender: TObject);
begin
 Autorise ( Edit2 , Prod_ok );
end;
```

On peut aussi mettre en place un gestionnaire unique de l'événement OnChange, puis de le lier à chaque zone de saisie Edit1 et Edit2 (souvenons-nous qu'un champ d'événement est un pointeur de méthode) :



Nous définissons d'abord une méthode public par exemple, qui jouera le rôle de gestionnaire centralisé d'événement, nous la nommons TexteChange.

Cette méthode pour être considérée comme un gestionnaire de l'événement OnChange, doit obligatoirement être compatible avec le type de l'événement Onchange. Une consultation de la documentation Delphi nous indique que :

property OnChange: TNotifyEvent;

L'événement OnChange est du même type que l'événement OnClick (TnotifyEvent = **procedure**(Sender:Tobject) **of object**), donc notre gestionnaire centralisé TexteChange doit avoir l'en-tête suivante :

procedure TexteChange (Sender : Tobject);

Le paramètre Sender est la référence de l'objet qui appelle la méthode qui est passé automatiquement lors de l'exécution, en l'occurrence ici lorsque Edit1 appellera ce gestionnaire c'est la référence de Edit1 qui sera passée comme paramètre, de même lorsqu'il s'agira d'un appel de Edit2.

Implantation du gestionnaire centralisé

```
procedure TForm1.TexteChange(Sender: TObject);
begin
  if  Sender is TEdit then
  begin
     if (Sender as TEdit )=Edit1 then
        Autorise ( (Sender as TEdit ) , Som_ok )
     else
        Autorise ( (Sender as TEdit ) , Prod_ok );
   end
end;
```

On teste si l'émetteur Sender est bien un TEdit,

polymorphisme d'objet :

Si l'émetteur est Edit1 on le **transtype** en TEdit pour pouvoir le passer en premier paramètre à la méthode Autorise **sinon** c'est Edit2 et l'on fait la même opération.

On lie maintenant ce gestionnaire à chacun des champs OnChange de chaque TEdit dès la création de la fiche :

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   RAZTout;
   Edit1.OnChange := TexteChange ;
   Edit2.OnChange := TexteChange ;
end;
```

pointeur de méthode :

chaque champ Onchange pointe vers le même gestionnaire (méthode)

```
TForm1= class ( TForm )
   Edit1: TEdit;       …
   procedure FormCreate(Sender: TObject);
   procedure ButtonCalculClick(Sender: TObject);
   procedure ButtoneffacerClick(Sender: TObject);
 private    { Déclarations privées }
    Som_ok , Prod_ok :
    procedure TestEntrees;
    procedure RAZTout;
    procedure Autorise ( Ed : TEdit ; var flag : boolean );
 public    { Déclarations publiques }
   procedure TexteChange(Sender: TObject);
 end;
```

Le gestionnaire centralisé est déclaré ici, puis il est implémenté à la section **implementation** de la **unit**.

Ce logiciel d'interface simplifiée a permis de mettre en œuvre tous les concepts de base d'une interface à l'exception des temps d'attente qui ne sont pas pertinents dans cet exemple :

> les objets d'entrée-sortie sont fournis par le RAD,
> le pilotage a été réalisé à l'aide du graphe événementiel,
> l'enchaînement a été implanté à travers la synchronisation dans le graphe événementiel,
> une partie de la sécurité a été obtenue en décomposant l'interaction homme-logiciel en deux plans d'action : le plan de saisie et le plan de calcul-affichage.

Corrections des exercices sur les BD (2,3,4,5)

Rappelons que vous pourrez développer avec delphi des logiciels d'accès à une BD Access uniquement si vous possédez les deux logiciels suivants :

- Le SGBD-R Microsoft Access
- Une version de Delphi contenant le BDE (version professionnelle, version développeur, version entreprise, version….) Les versions standard et gratuites dites personnelles, ne contiennent pas de composants de Base de Données et vous ne pourrez pas développer ces exercices.

Le minimum requis pour faire fonctionner les exercices ci-après est :

- Delphi professionnel 5
- Access 97
- Windows Xp

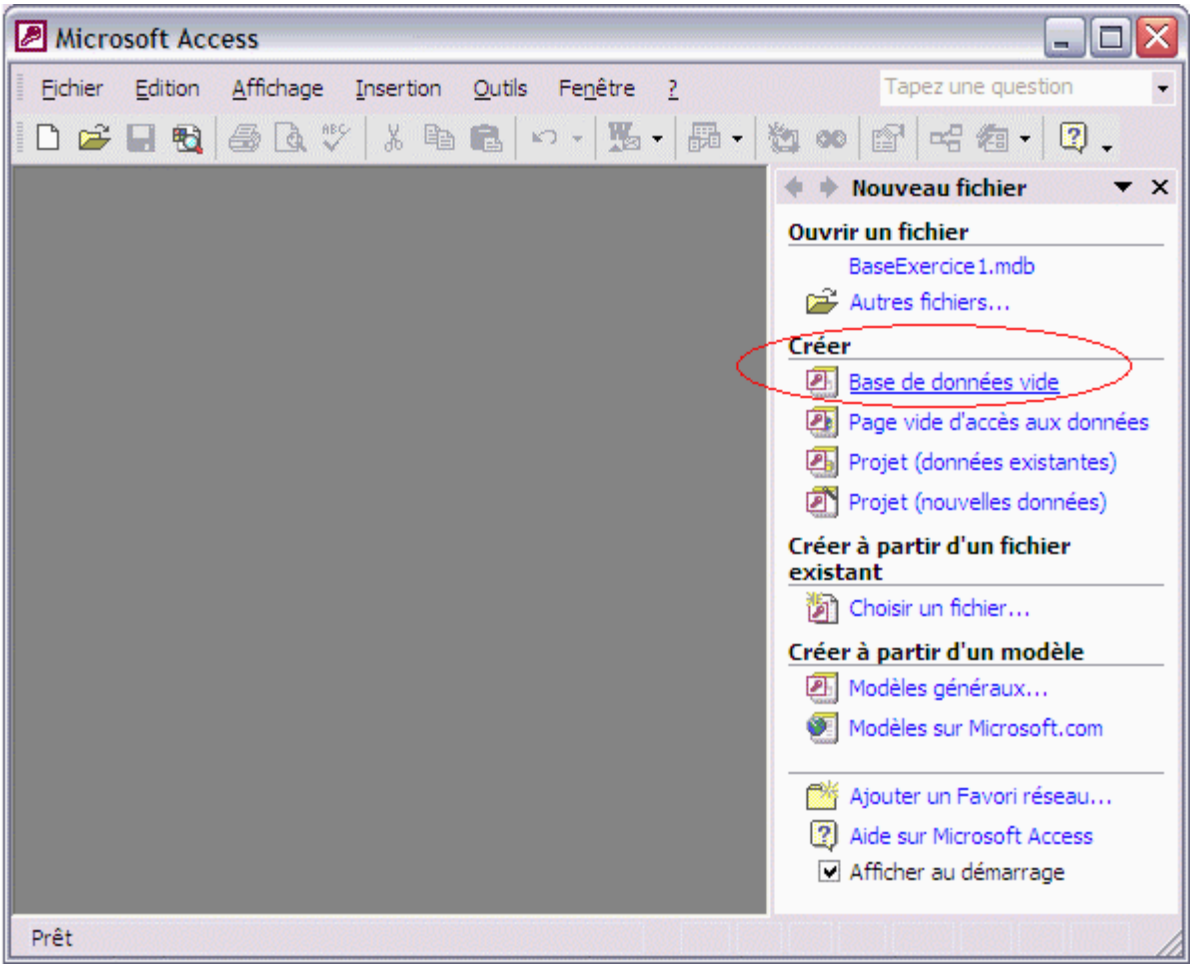
(ils ont aussi été testés avec Delphi 7 professionnel et Access 2002 sous Xp)

Démarche de présentation des solutions des exercices :

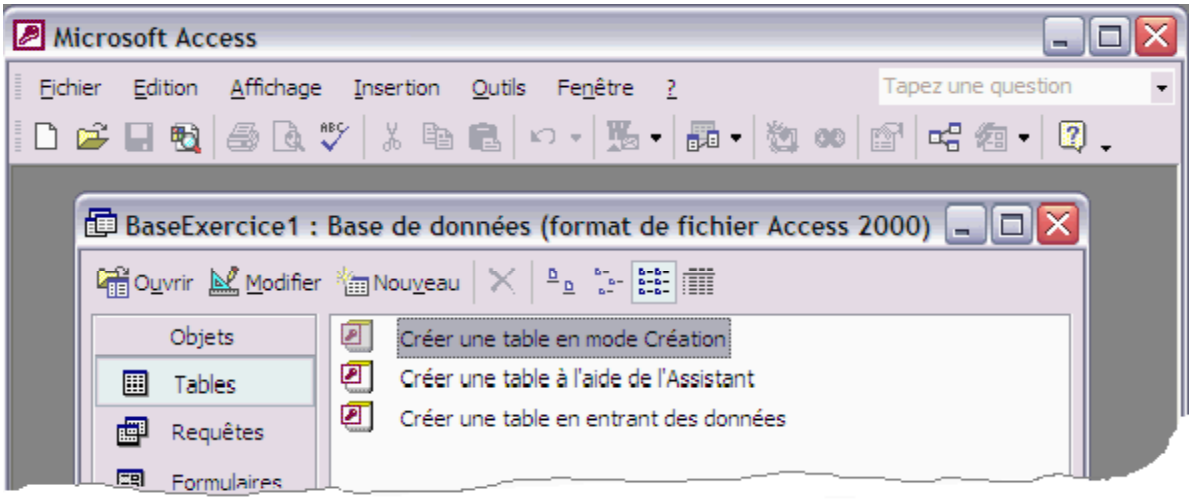
- Nous décrivons sous forme synoptique comment créer la base de données BD.
- Nous montrons ensuite comment gérer des alias de BD à travers le BDE de Borland
- Puis pour chacun des 4 exercices :
  - Nous donnons la construction de l'interface de l'exercice avec Delphi.
  - Nous donnons le code source de l'exercice en Delphi.

Décrivons d'abord pas à pas la création de la BD avec le SGBD-R Access (les images présentées correspondent ici à la version Access 2002).

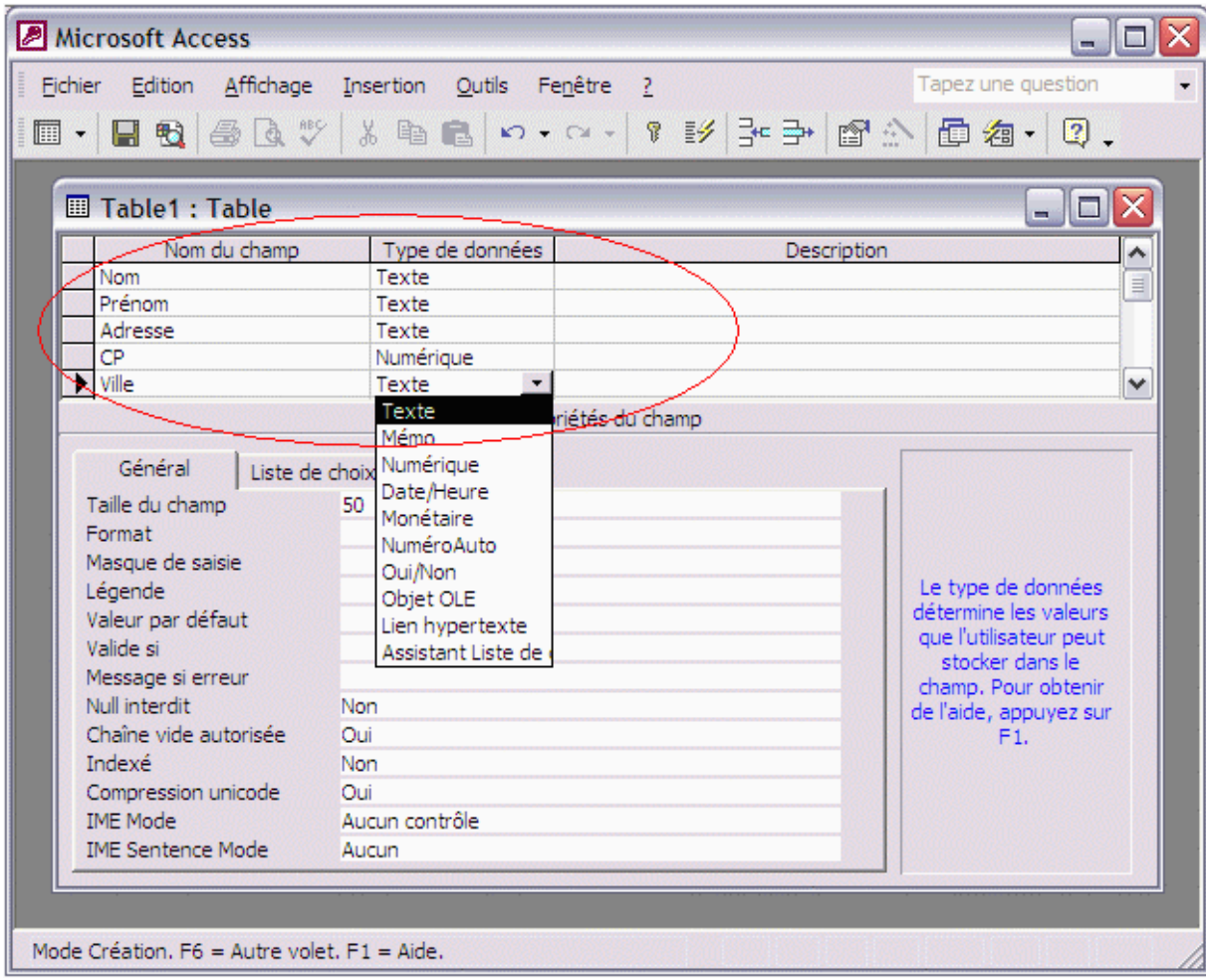
1°) créer une base de données (vide au départ) :



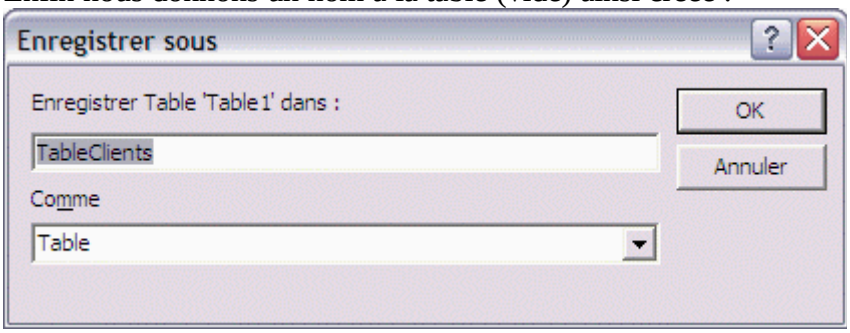
Il faut créer la structure de la table TableClients :



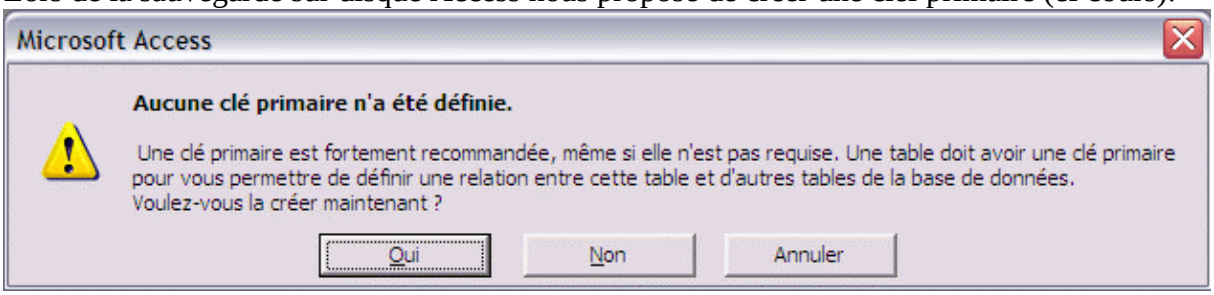
Nous devons nommer et typer les champs de la table :



Enfin nous donnons un nom à la table (vide) ainsi créée :

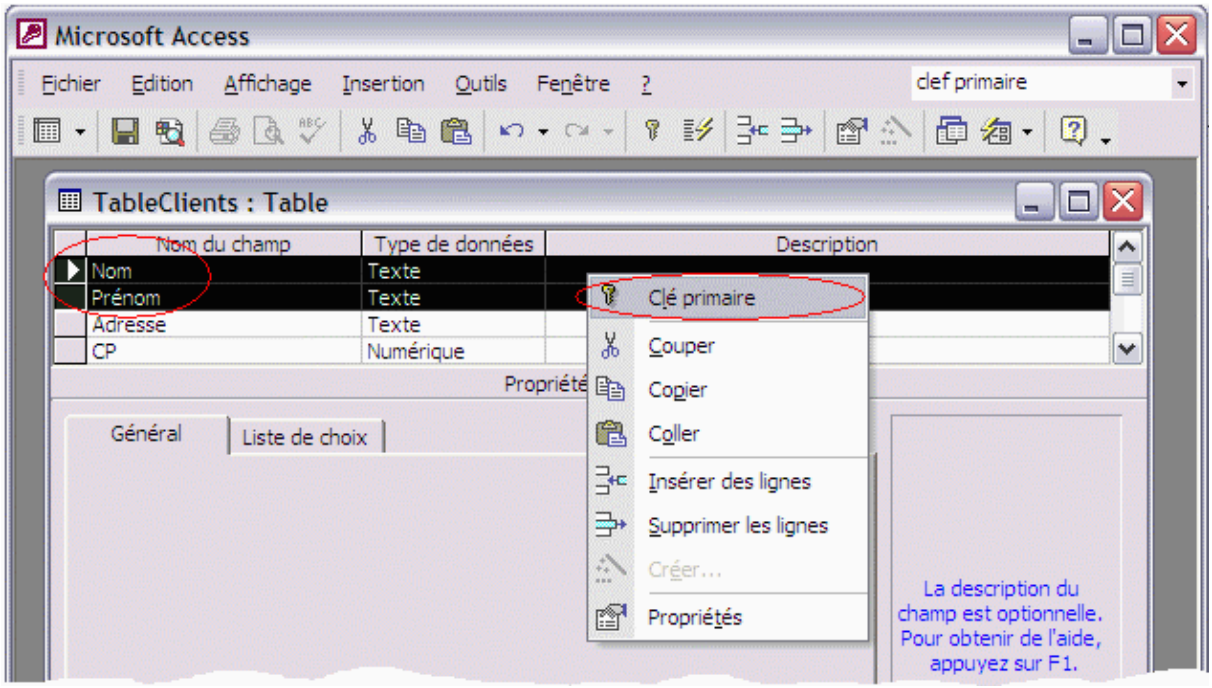


Lors de la sauvegarde sur disque Access nous propose de créer une clef primaire (cf cours):

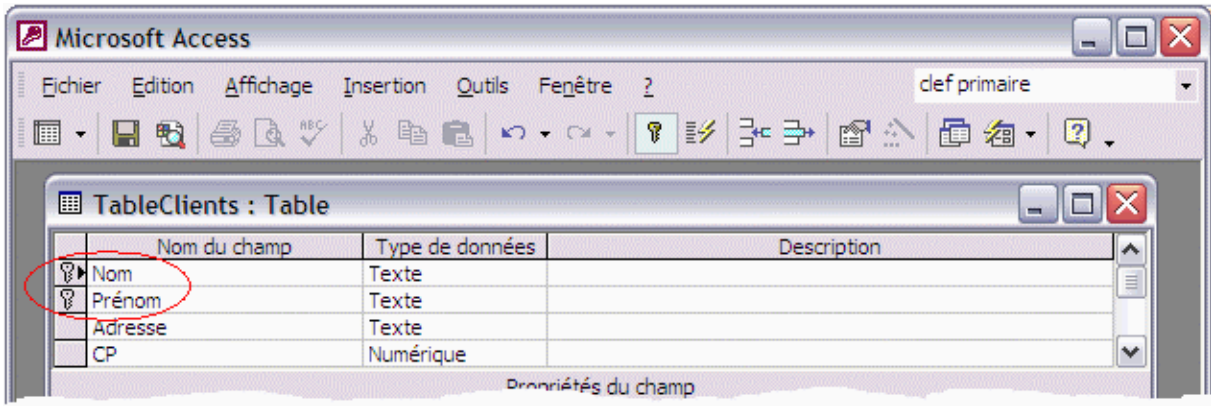


Nous acceptons de définir notre clef primaire en répondant oui à cette invite. Nous définissons une clef primaire possédant 2 champs : le **nom** et le **prénom**. Nous sélectionnons les deux

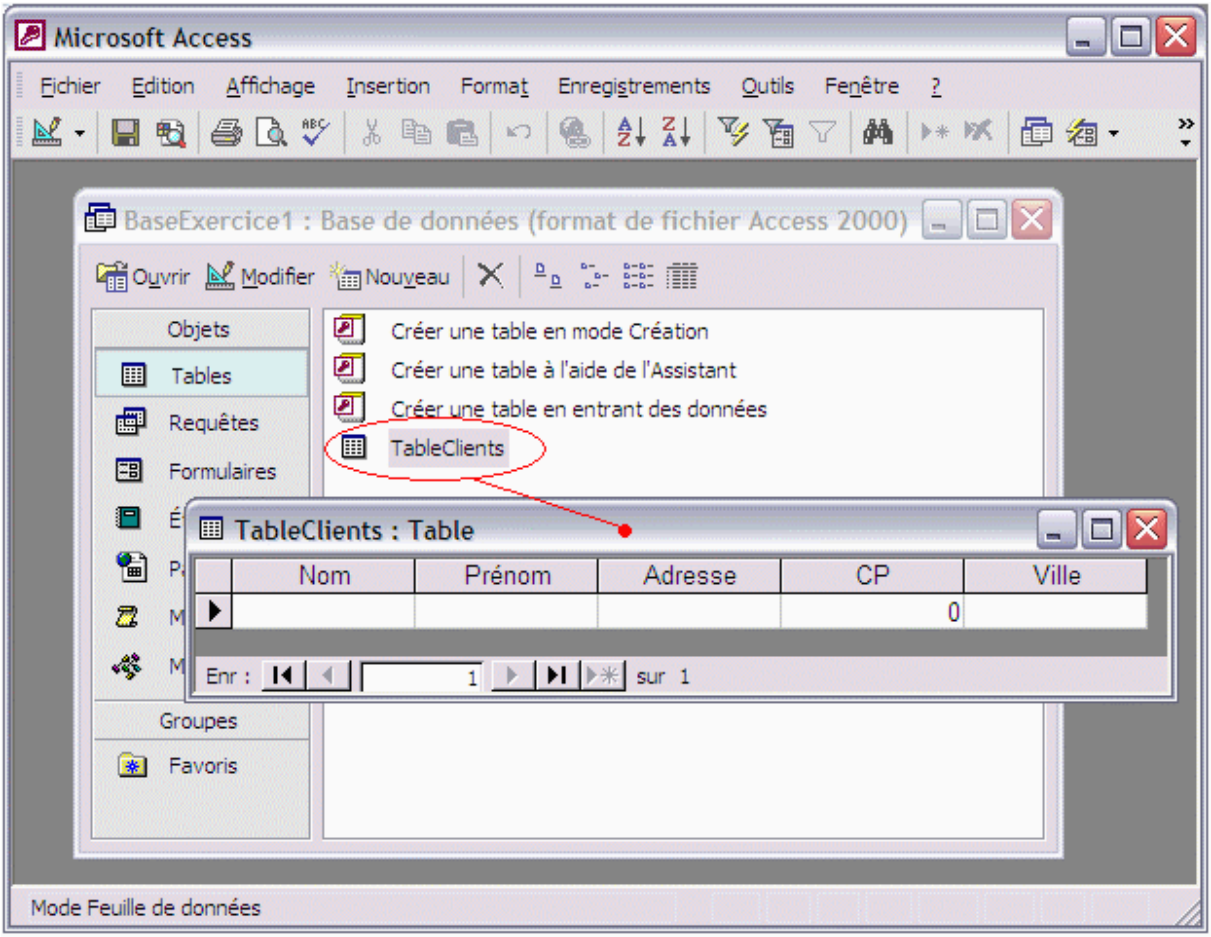
lignes nom et prénom dans l'assistant de création de la table et avec le bouton droit de souris, nous indiquons que ces deux champs (les 2 lignes sélectionnées) constituent la clef primaire de notre relation :



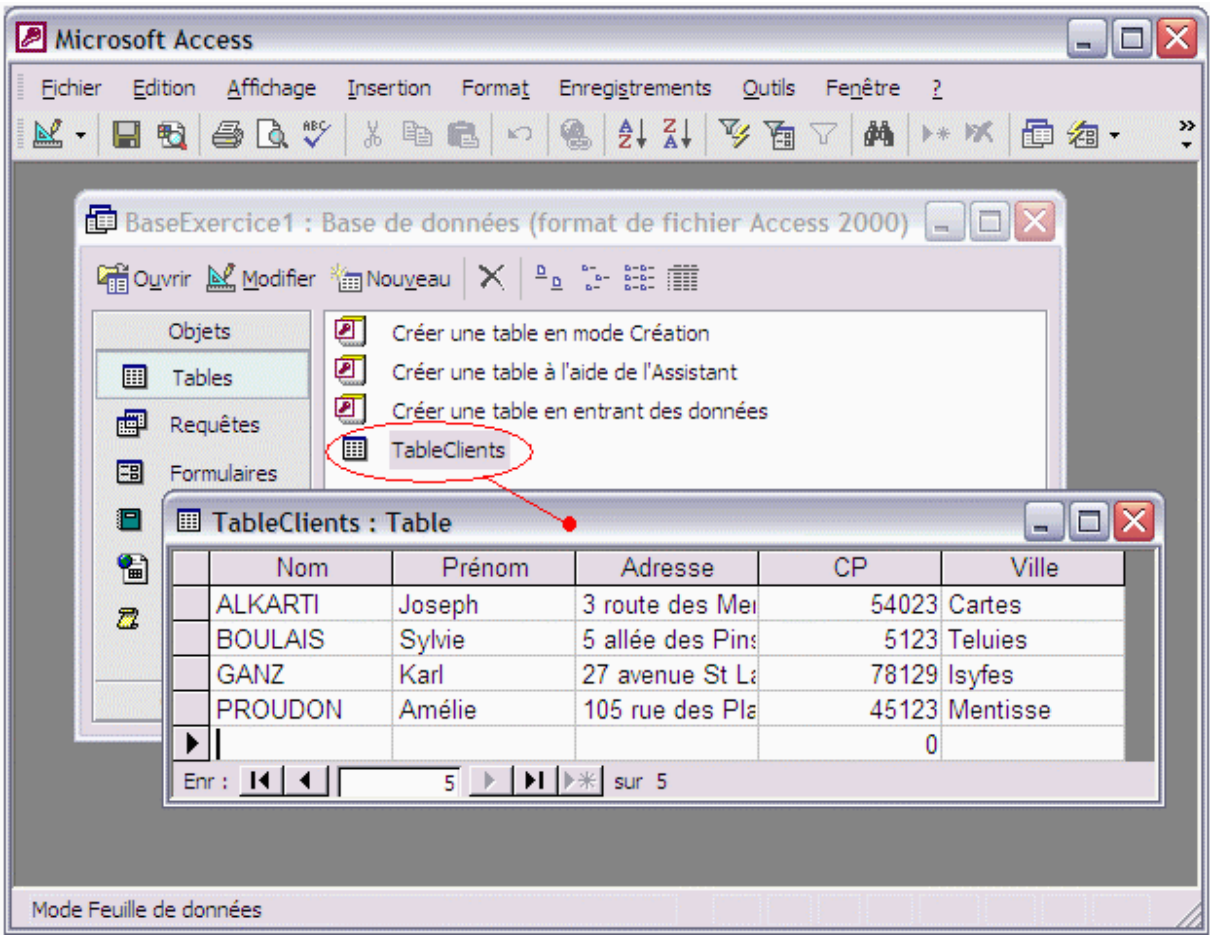
Access nous signale par une petite icône de clef les champs composant la clef primaire de la relation :



La structuration de la table TableClients est enfin terminée et l'assistant du SGBD Access nous fournit la table vide prête à être saisie :



Voici affiché par l'interface du SGBD Access, notre table une fois remplie manuellement :



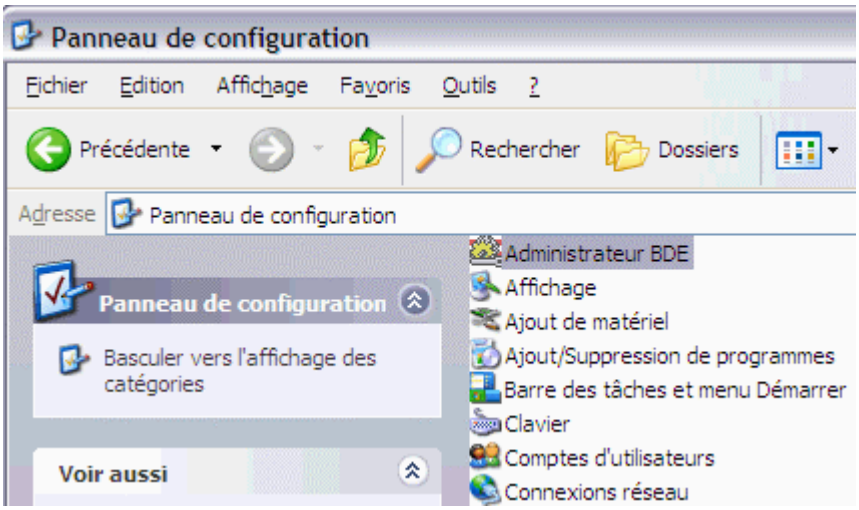
Cette BD est sauvegardée sous le nom : **BaseExercice1.mdb**

TableClients : Table

Nom	Prénom	Adresse	CP	Ville
ALKARTI	Joseph	3 route des Merles	54023	Cartes
BOULAIS	Sylvie	5 allée des Pins	05123	Teluies
GANZ	Karl	27 avenue St Laurent	78129	Isyfes
PROUDON	Amélie	105 rue des Platanes	45123	Mentisse

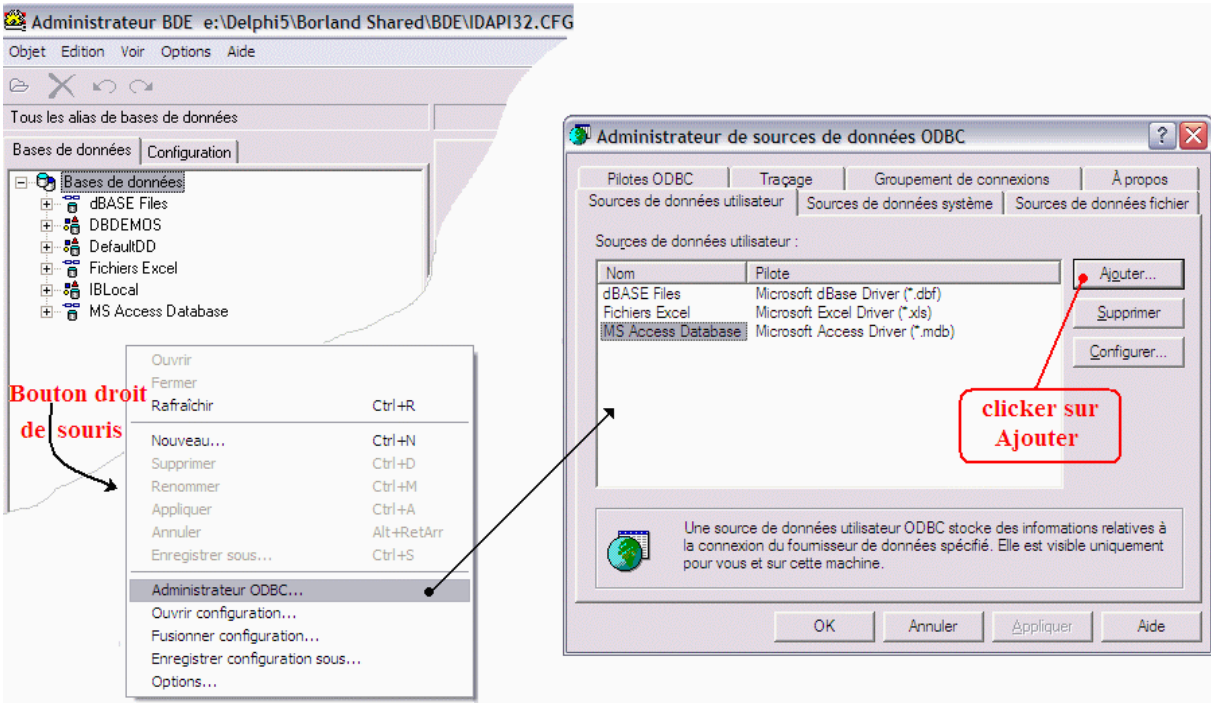
2°) Gestion des alias de BD à travers le BDE

Nous nous servons du moteur de base de données appelé le BDE de Borland pour accéder à notre base Access à partir d'un programme Delphi. Nous utilisons l'outil d'administration du BDE nommé « Administrateur BDE » mis à disposition par Delphi dans le dossier « Panneau de configuration ». Il est alors possible de paramétrer (établir la connexion physique avec la BD) le BDE directement grâce à l'administrateur du BDE fournit avec Delphi :



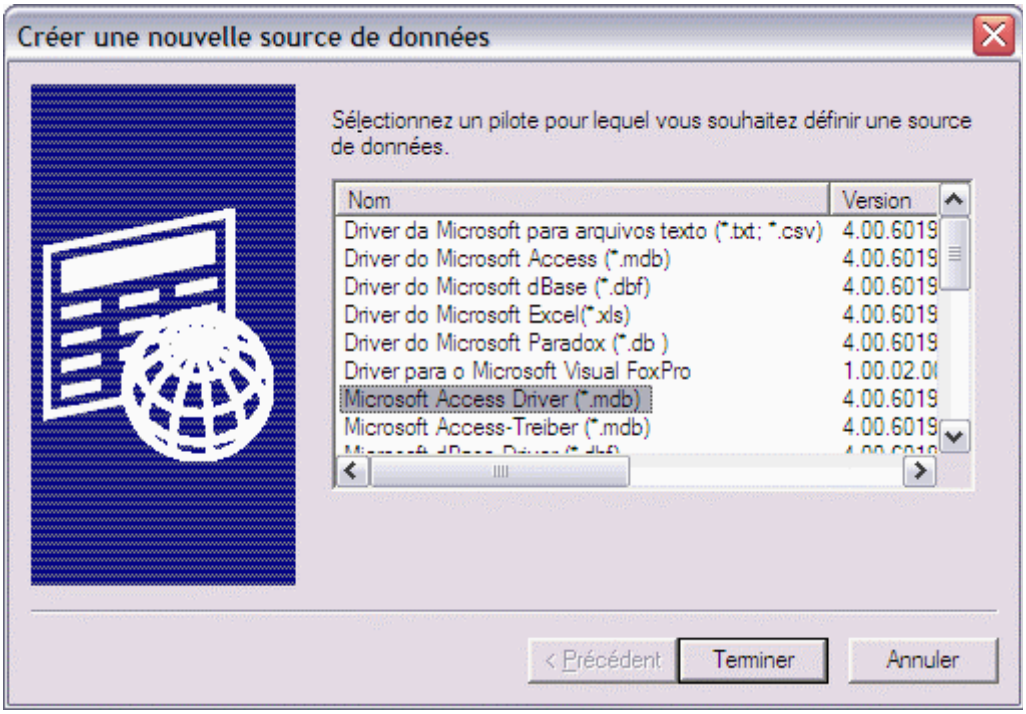
Dans le cours les exemples ont utilisé un pilote dit natif "MSACCESS" fourni par Borland et paramétrant le BDE à partir du composant TDatabase. Dans les exercices, nous allons utiliser un autre genre de pilote de type ODBC (l'Open Data Base Connectivity est un standard d'accès aux BD) de Microsoft à la place du pilote natif fourni.

Nous appelons l'administrateur du BDE à partir du panneau de configuration (cf. chap 7.5) , puis nous demandons à créer un nouvel alias de connexion physique par pilote ODBC :

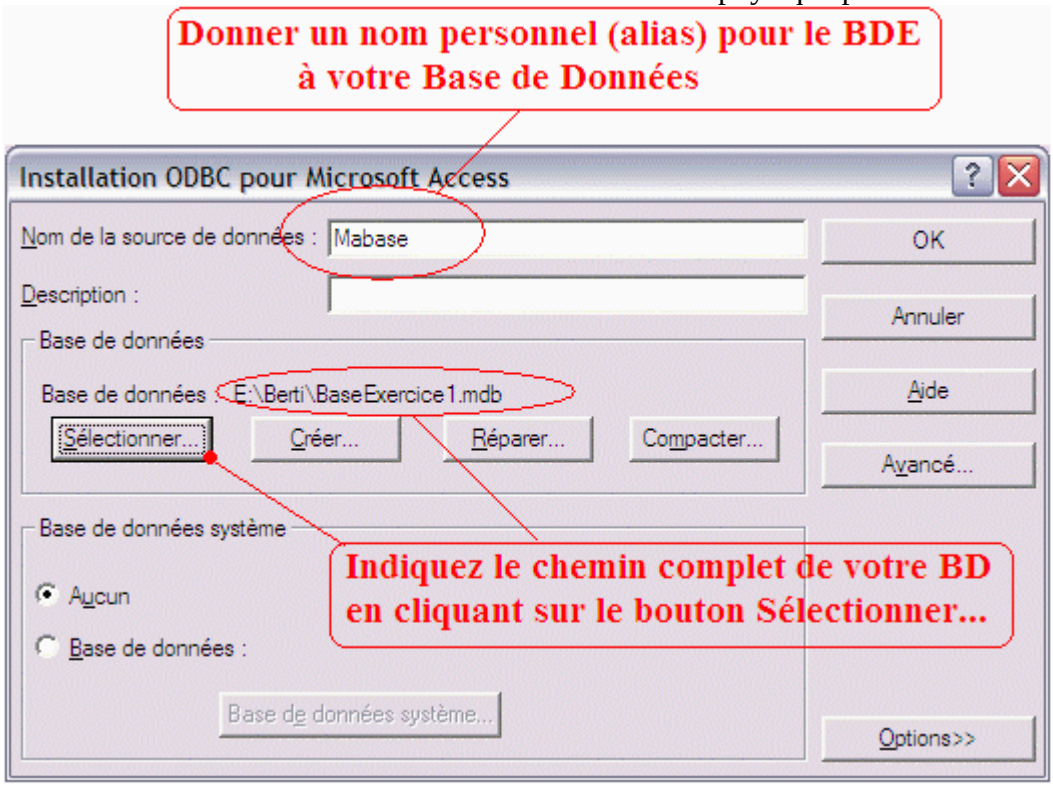


Nous ajoutons dans l'onglet « Sources de données utilisateur » notre BD « BaseExercice1.mdb »

a) en sélectionnant le pilote ODBC fourni par Microsoft dans Access, dénommé « **Microsoft Access Driver** » :

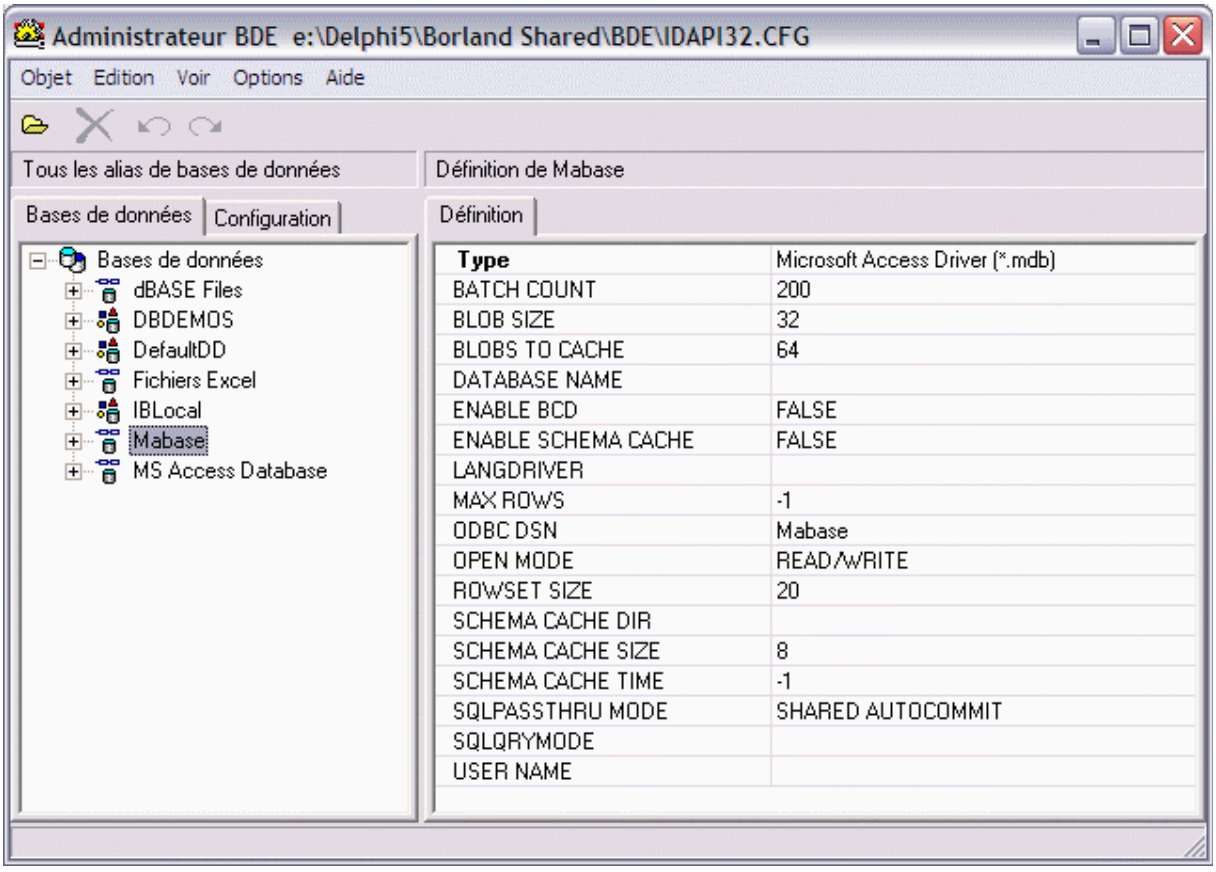


b) en reliant ce pilote avec un nom d'alias que nous choisissons (ici : **MaBase**) et notre BD « **BaseExercice1.mdb** » dont nous donnons le chemin physique précis :



La fenêtre d'administration de sources de données ODBC a enregistré notre nouvelle Base avec son nom « MaBase » :

L'administrateur du BDE contient et affiche maintenant une BD nouvellement accessible qui se dénomme « MaBase » :

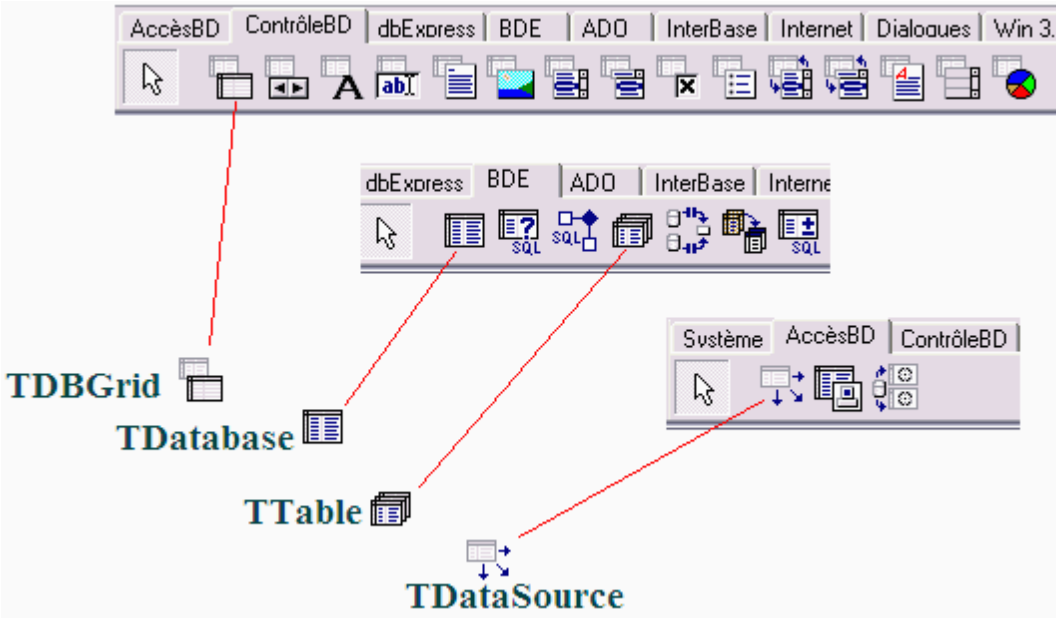


Ces opérations terminées, la BD « **BaseExercice1.mdb** » est dès lors utilisable sous le nom **MaBase** par des programmes Delphi en consultation ou en mise à jour.

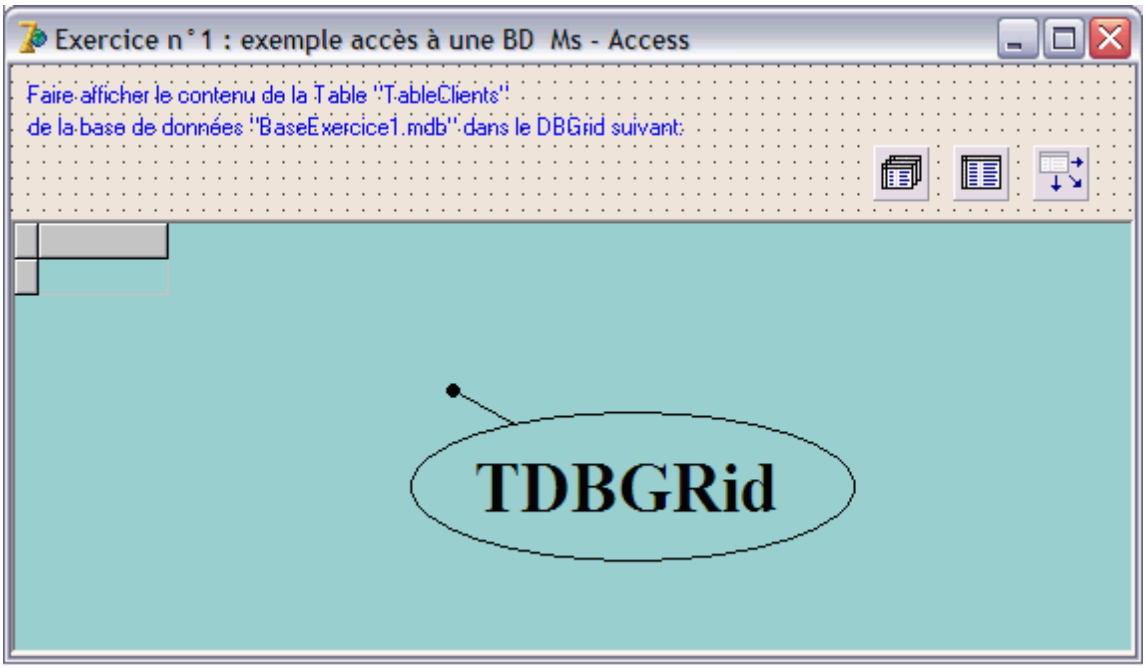
Ex-2 : construire une IHM Delphi - consultation

3°) construction de l'interface de consultation de la BD en Delphi

La démarche complète de création de l'interface avec Delphi se trouve au paragraphe 5.3.2 du chapitre 7.5 sur l'utilisation des bases de données.

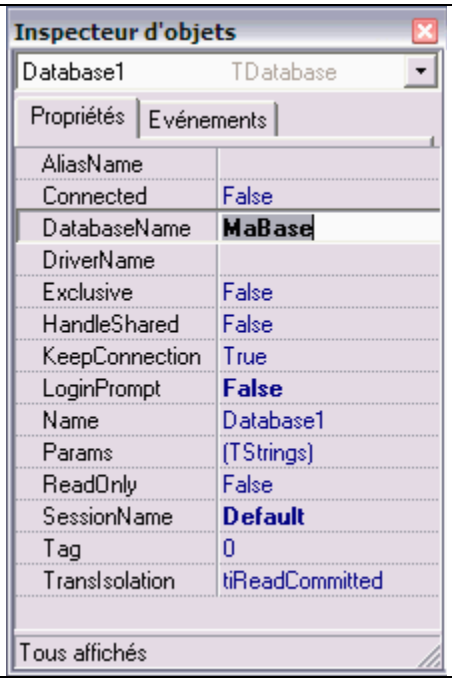


Parmi les 4 composants précédents un seul est un composant visuel le **TDBGrid** :



Pour préparer le programme, il nous faut relier le **TDataBase** à la base de données physique (cela est fait à travers le BDE) :

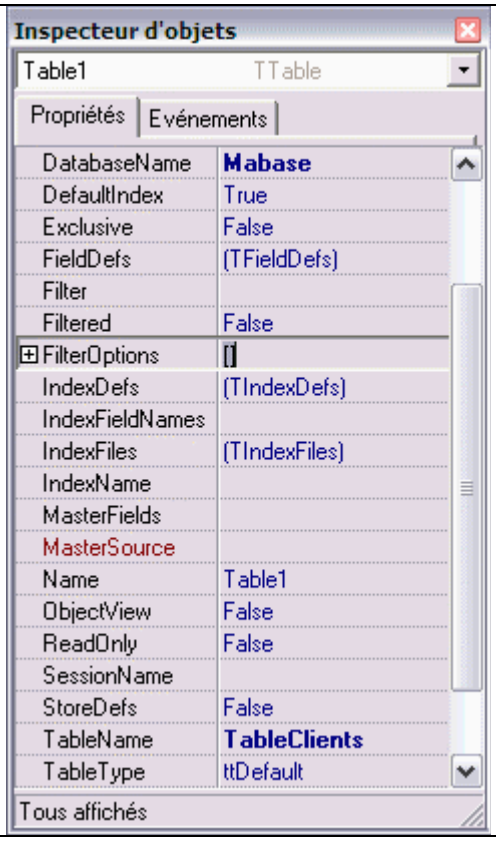
Pour effectuer cette liaison, il nous suffit de donner dans le champ ***DataBaseName*** du TDataBase, le nom de l'alias que nous avons utilisé dans le BDE : « **MaBase** »



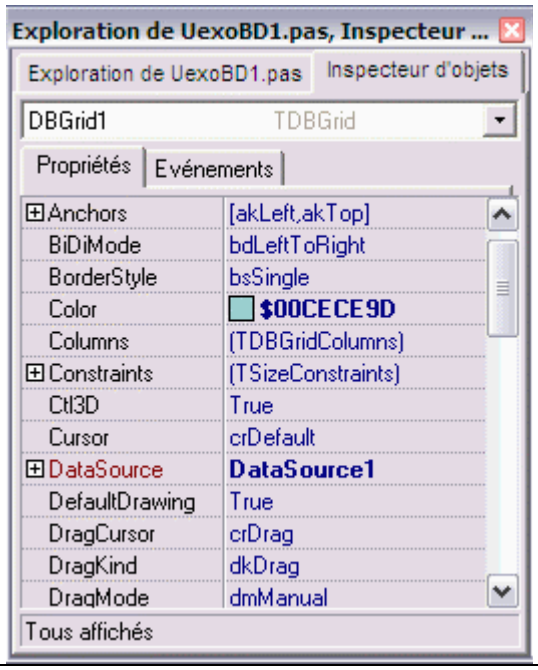
Comme nous voulons afficher la table « TableClients », il nous faut utiliser le composant **TTable** qui permet d'accéder aux données d'une table de base de données en utilisant le BDE ; Il faut indiquer au **TTable** quelle est le nom de la BD à laquelle il doit être connecté et la table de cette base à laquelle il doit être relié :

Le champ ***DataBaseName*** du TTable contient le nom (alias) de la BD à laquelle il est connecté.

Le champ ***TableName*** du TTable contient le nom de la table à laquelle il est relié



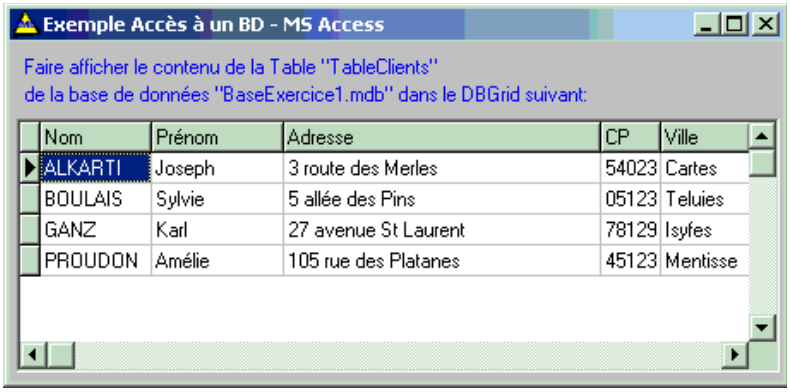
Le **TDataSource** est relié à la fois à la table à afficher à travers le **TTable** et au composant visuel d'affichage le **TDBGrid**:

Le TDataSource	Le TDBGrid
	

4°) Code source delphi pour l'affichage du contenu de la table de la BD

```
unit uFExercice1;

 interface

 uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, DBCtrls, Grids, DBGrids, StdCtrls, Mask, DBTables, Db;

 type
  TFExercice1 = class(TForm)
  DBGrid1: TDBGrid;
  Table1: TTable;
  DataSource1: TDataSource;
  Database1: TDatabase;
  Label1: TLabel;
  Label2: TLabel;
  procedure FormCreate(Sender: TObject);
 private
 { Déclarations privées }
 public
 { Déclarations publiques }
end;

 var
  FExercice1: TFExercice1;
  App_Path:string;

implementation
```

```
{$R *.DFM}
```




```
procedure TFExercice1.FormCreate(Sender: TObject);
 begin
   DataBase1.connected:=true;  //connexion sur la base de données
  //(évidemment il faut d'abord paramétrer
  //correctement le composant DataBase1)
   Table1.open; //Ouverture de la table attachée au composant Table1
                //(il faut avoir bien paramétré le composants Table1auparavant
 end;

end.
```

Nous remarquons que le code est très court; ce sont les composant qui font tout le travail parce qu'ils ont été paramétrés pendant la conception.

Autre solution de code : le paramétrage des composants lors de la création de la fiche

Dans cette deuxième éventualité, le **TDataBase** et le **TTable** ne sont pas paramétrés lors du dépôt visuel comme dans les schémas présentés ci-haut lors de la conception grâce à l'inspecteur d'objets.

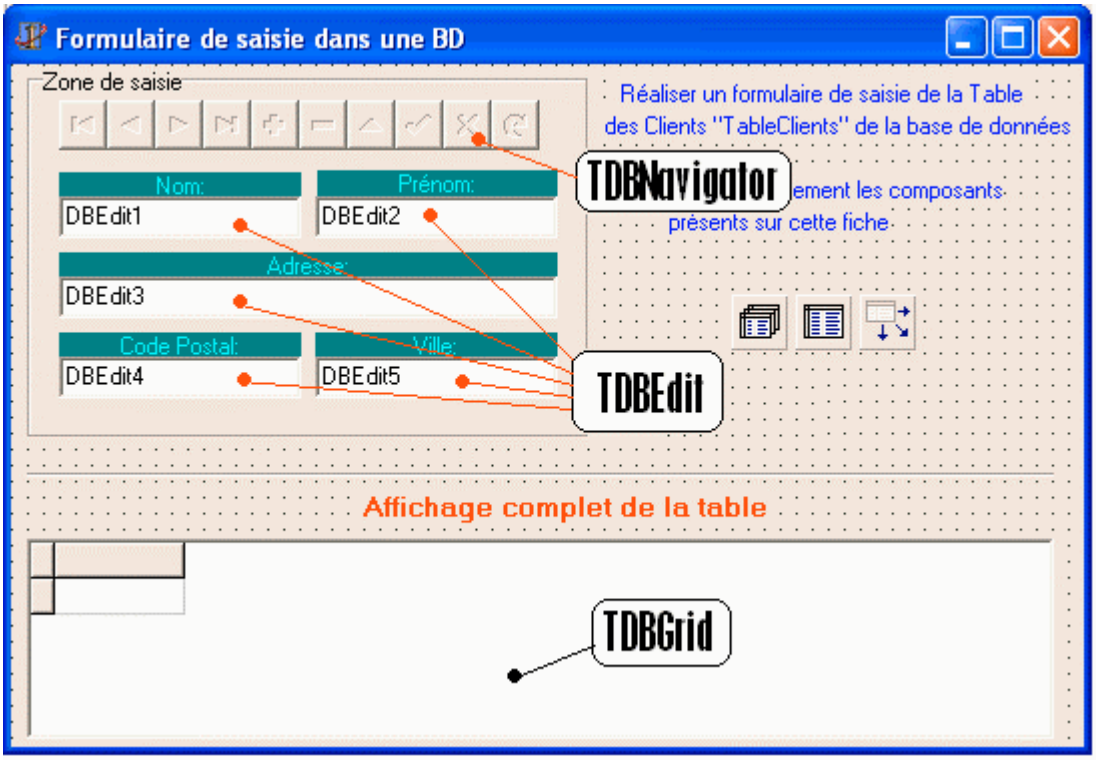
Ils sont paramétrés directement par programmation, pendant l'exécution et lors de la survenue de l'événement de création de la fiche :

```
procedure  TFExercice1.FormCreate( Sender:  TObject) ;
  begin
    DataBase1.DatabaseName := 'MaBase';
    DataBase1.connected := true ;  //connexion sur la base de données

    Table1.DatabaseName := 'MaBase';
    Table1.TableName := 'TableClients';
    Table1.open ;  //Ouverture de la table attachée au composant Table1
   end;
```

Ceci montre que ces composants peuvent, pendant l'exécution être paramétrés à nouveau sur une autre BD existante et jouer leur rôle d'accès à cette nouvelle base.

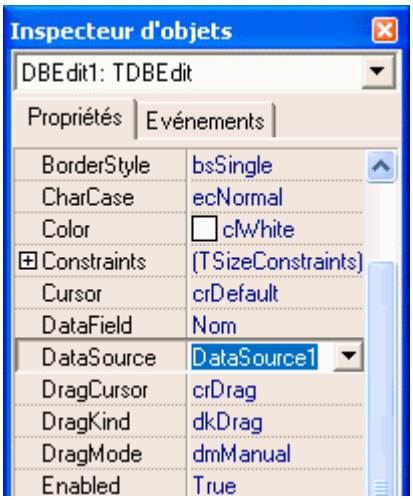
Pour les exercices suivants, nous ne paramétrerons pas le **TDataBase** et le **TTable** lors de la conception, mais par programme comme indiqué dans ce paragraphe.

3°) construction de l'interface de navigation dans la BD en Delphi



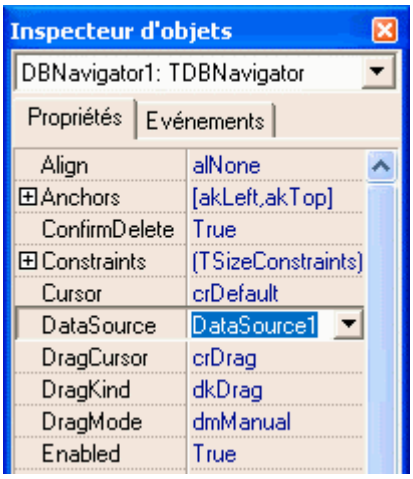
Le **TDataSource** est relié à la fois à la table à afficher à travers le **TTable** et au composant visuel d'affichage le **TDBGrid.** Le **TDataBase** est relié à la base de données physique comme dans l'exercice précédent.

Les quatre **TDBEdit** représentent des contrôles de saisie pouvant afficher et modifier les valeurs d'un champ de la BD. Chacun d'eux est relié par sa propriété DataSource au composant **TDataSource** (nommé DataSource1) :



etc …

Le **TDBNavigator** est un contrôle utilisé pour se déplacer dans une BD afin d'effectuer des actions sur les données(suppression, modification, insertion,…), il est relié par sa propriété DataSource au même composant **TDataSource** (nommé DataSource1) que les **TDBEdit** et le **TDBGrid** :



4°) Code source delphi pour la navigation dans la table de la BD

```
unit  UBDExo2 ;

 interface

 uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Grids, DBGrids, ExtCtrls, Mask, DBCtrls, Db, DBTables ;

 type
   TForm1  =  class (TForm)
  DBGrid1 :  TDBGrid ;
   Label1 :  TLabel ;
   Label2 :  TLabel ;
   Label4 :  TLabel ;
   Label5 :  TLabel ;
   Label6 :  TLabel ;
   GroupBox1 :  TGroupBox ;
   DBNavigator1 :  TDBNavigator ;
   DBEdit1 :  TDBEdit ;
   DBEdit2 :  TDBEdit ;
   DBEdit3 :  TDBEdit ;
   DBEdit4 :  TDBEdit ;
   DBEdit5 :  TDBEdit ;
   Label7 :  TLabel ;
   Label8 :  TLabel ;
   Label9 :  TLabel ;
   Label10 :  TLabel ;
   Label11 :  TLabel ;
   Bevel1 :  TBevel ;
   Database1 :  TDatabase ;
   Table1 :  TTable ;
   DataSource1 :  TDataSource ;
  procedure  FormCreate( Sender:  TObject) ;
```

```
 private
  { Déclarations privées }
  public
  { Déclarations publiques }
  App_Path :string ;
end;

 var
   Form1 :  TForm1 ;

implementation

  {$R *.DFM}

 procedure  TForm1.FormCreate( Sender:  TObject) ;
 begin
    DataBase1.DatabaseName := 'MaBase';
    DataBase1.connected := true ;  //connexion sur la base de données

    Table1.DatabaseName := 'MaBase';
    Table1.TableName := 'TableClients';
    Table1.open ;  //Ouverture de la table attachée au composant Table1
 end;

end.
```

De la même manière que dans l'exercice précédent, le **TDBEdit** et le **TDBNavigator** ont été paramétrés pendant la conception. Ils peuvent aussi être paramétrés par programme pendant l'exécution.

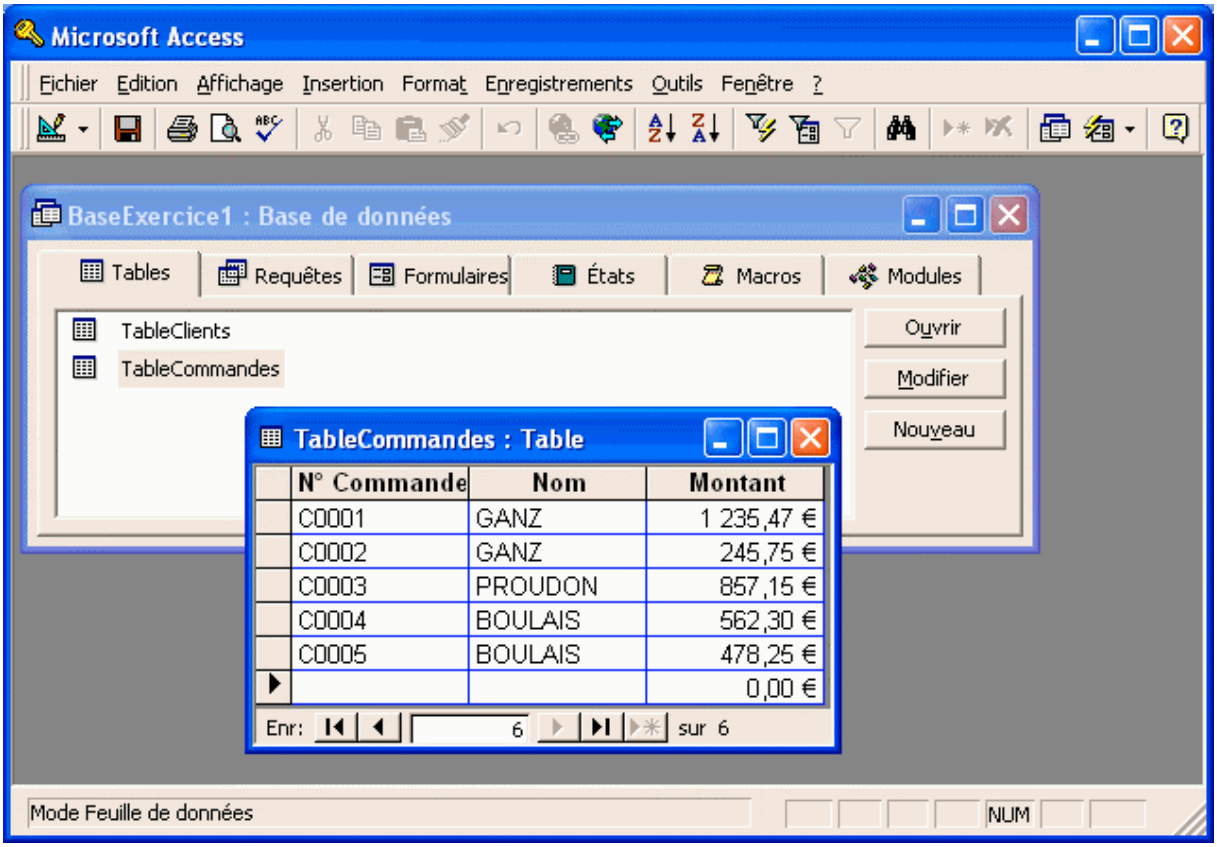
Autre solution de code : le paramétrage des composants lors de la création de la fiche

```
procedure  TForm1.FormCreate( Sender:  TObject) ;
begin
    DataBase1.DatabaseName := 'MaBase';
    DataBase1.connected := true ;  //connexion sur la base de données
    DBNavigator1.DataSource := DataSource1 ;
    DBEdit1.DataSource := DataSource1 ;
    DBEdit2.DataSource := DataSource1 ;
    DBEdit3.DataSource := DataSource1 ;
    DBEdit4.DataSource := DataSource1 ;
    DBEdit5.DataSource := DataSource1 ;
    Table1.DatabaseName := 'MaBase';
    Table1.TableName := 'TableClients';
    Table1.open ;  //Ouverture de la table attachée au composant Table1
end;
```

Ex-4 : Affichage des résultats d'une requête

3°) Requête affichant nom, prénom et montant des achats

Avec Access, dans notre BD nous ajoutons une seconde table : **TableCommandes**, qui contient les achats effectués par les personnes rangées dans la base (considérées comme des acheteurs) :

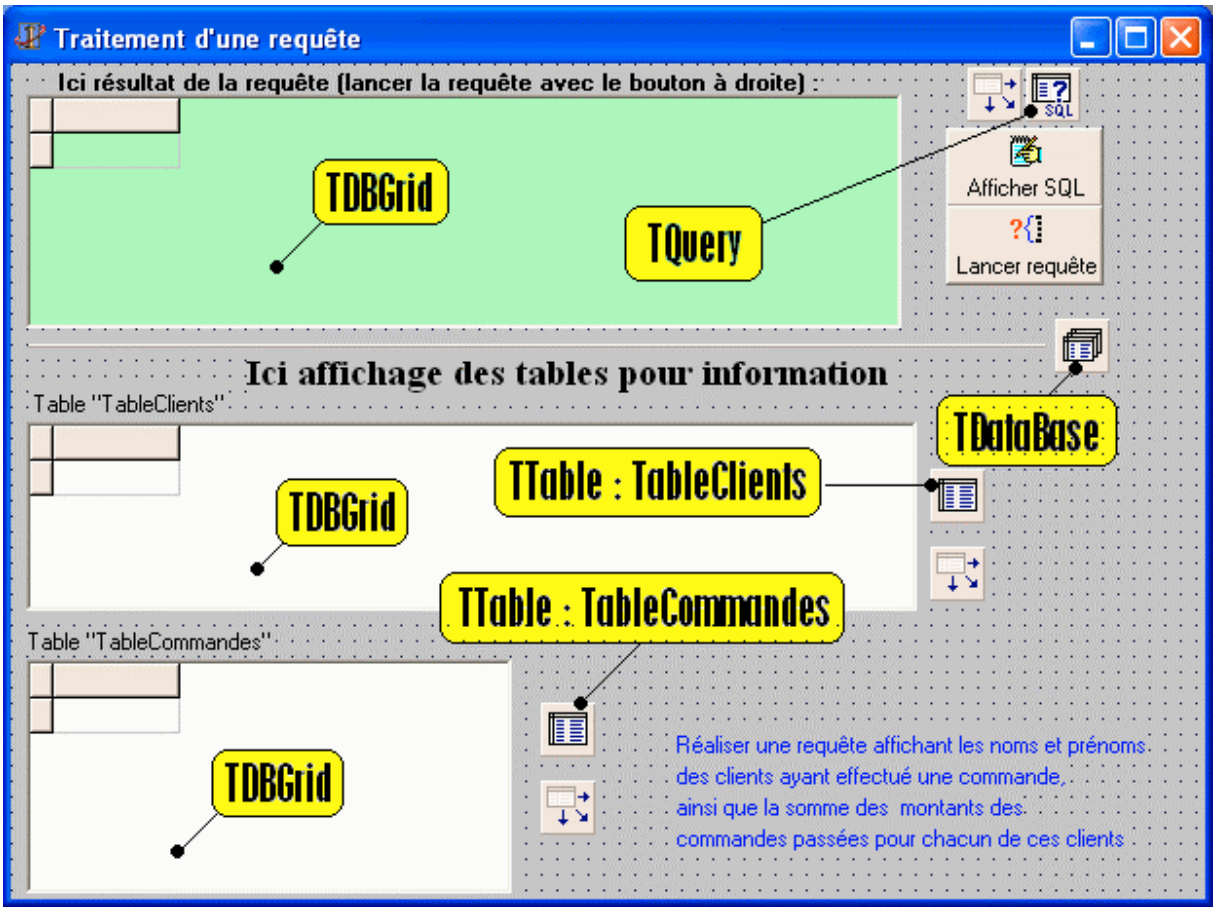


L'interface de l'exercice, dans laquelle aucun des composants déposés n'est paramétré lors de la conception, mais programmé pendant l'exécution afin que le lecteur puisse réutiliser le code source.

4°) Code source delphi de l'exercice

```
unit  UBDExo3 ;

 interface

 uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  DBTables, StdCtrls, ExtCtrls, Db, Buttons, Grids, DBGrids ;

 type
   TForm1  =  class (TForm)
  DBGrid3 :  TDBGrid ;
   DBGrid2 :  TDBGrid ;
   DBGrid1 :  TDBGrid ;
   SpeedButton2 :  TSpeedButton ;
   SpeedButton1 :  TSpeedButton ;
   Database1 :  TDatabase ;
   TableClients :  TTable ;
   DSTableClients :  TDataSource ;
   TableCommandes :  TTable ;
   DSTableCommandes :  TDataSource ;
```

```
  Bevel1 : TBevel ;
  DataSource1 : TDataSource ;
  Label4 : TLabel ;
  Label5 : TLabel ;
  Label3 : TLabel ;
  Label1 : TLabel ;
  Label2 : TLabel ;
  Label6 : TLabel ;
  Query1 : TQuery ;
  Label7 : TLabel ;
  Label8 : TLabel ;
```

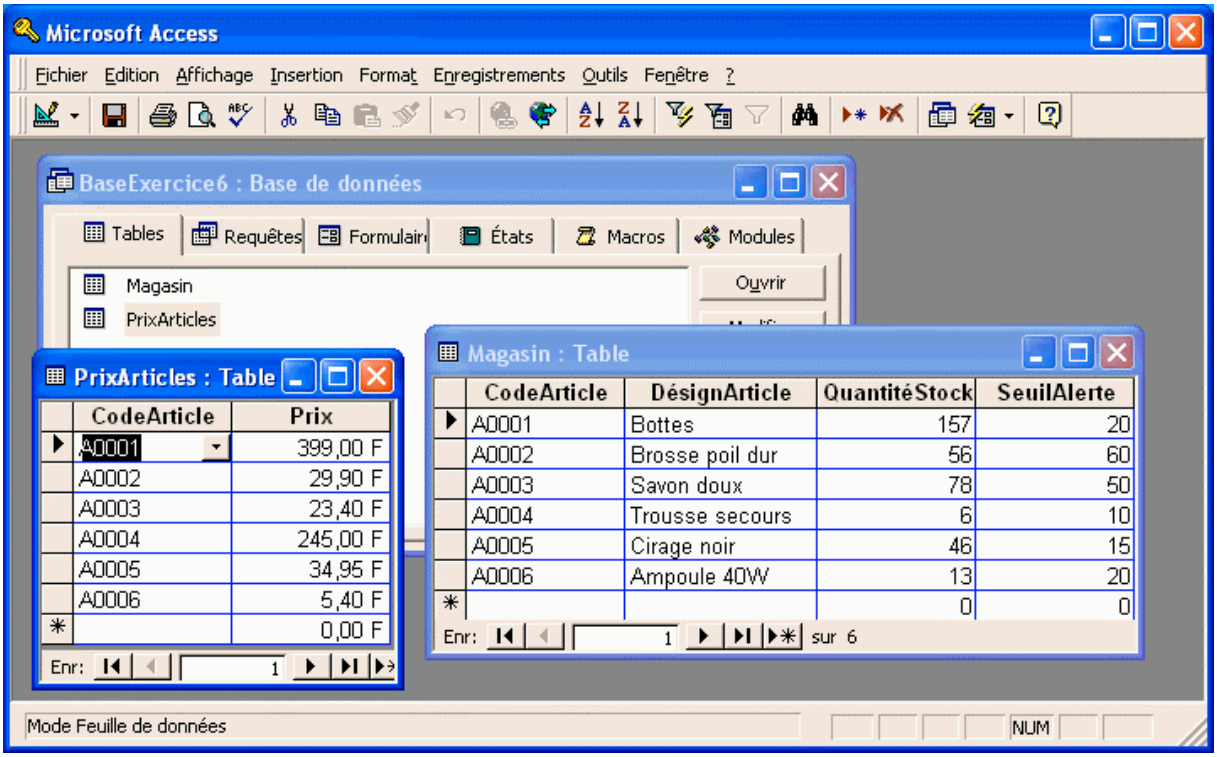



```
  procedure SpeedButton2Click( Sender: TObject) ;
  procedure SpeedButton1Click( Sender: TObject) ;
  procedure FormCreate( Sender: TObject) ;
 private
  { Déclarations privées }
  public
  { Déclarations publiques }
  App_Path :string ;
end;

 var
  Form1 : TForm1 ;

implementation

uses uFAffichage3 ;

  {$R *.DFM}
```

```
 // Affiche dans un Tmemo le contenu du Tstrings Query.SQL (le texte de la requête)
procedure  TForm1.SpeedButton2Click( Sender:  TObject) ;
begin
   FAffichage3.memo1.lines.assign(Query1.SQL) ;
   FAffichage3.showmodal ;
end;

  procedure  TForm1.SpeedButton1Click( Sender:  TObject) ;
  begin
    Query1.open ;  // lance la requête incluse dans le champ SQL du TQuery
   end;

  procedure  TForm1.FormCreate( Sender:  TObject) ;
  begin
    DataBase1.DatabaseName  :=  'MaBase';
    DataBase1.connected  :=  true  ;   //connexion sur la base de données

    TableClients.DatabaseName  :=  'MaBase';
    TableClients.TableName  :=  'TableClients';
    TableClients.open  ;   //Ouverture de la table attachée au composant : TableClients
    DSTableClients.DataSet := TableClients ; // liaison TDBGrid <--> TTable
    DBGrid2.DataSource := DSTableClients ;

    TableCommandes.DatabaseName  :=  'MaBase';
    TableCommandes.TableName  :=  'TableCommandes';
    TableCommandes.open  ;   //Ouverture de la table attachée au composant : TableCommandes
    DSTableCommandes.DataSet := TableCommandes ;  // liaison TDBGrid <--> TTable
    DBGrid1.DataSource := DSTableCommandes ;

    Query1.DatabaseName := 'MaBase';
    Query1.SQL.Clear ;
    Query1.SQL.Append( 'SELECT TableClients.Nom, TableClients.Prénom, ' +
                                  'Sum(TableCommandes.Montant)' ) ;
    Query1.SQL.Append('FROM TableClients INNER JOIN TableCommandes' +
                                 'ON TableClients.Nom = TableCommandes.Nom' ) ;
    Query1.SQL.Append( 'GROUP BY TableClients.Nom, TableClients.Prénom' ) ;
    DataSource1.DataSet := Query1 ;  // liaison TDBGrid <--> TQuery
   end;

end.
```

Ex-5 : Affichage des résultats calculés d'une requête

3°) Requête et affichage de résultats calculés



Si l'on lance la requête SQL demandée :

```
SELECT Magasin.CodeArticle, Magasin.DésignArticle, Magasin.QuantitéStock, Magasin.SeuilAlerte
FROM Magasin ORDER BY Magasin.CodeArticle
```

vers un TDBGrid comme dans l'exemple précédent, voici ce qui sera affiché :

CodeArticle	DésignArticle	QuantitéStock	SeuilAlerte
A0001	Bottes	157	20
A0002	Brosse poil dur	56	60
A0003	Savon doux	78	50
A0004	Trousse secours	6	10
A0005	Cirage noir	46	15
A0006	Ampoule 40W	13	20

L'affichage de la table TablePrix se fait par liaison avec un TTable et un TDataSource :

CodeArticle	Prix
A0001	399
A0002	29,9
A0003	23,4
A0004	245
A0005	34,95
A0006	5,4

En fait l'énoncé nous demande à partir de la requête précédente de supprimer à l'affichage la colonne "CodeArticle" et d'ajouter deux colonnes en plus, l'une appelée "Message" est

obtenue par calcul sur le niveau du seuil d'alerte, l'autre nommée "Prix" correspond au prix de chaque "Article" contenu dans la table "TablePrix" :

DésignArticle	QuantitéStock	SeuilAlerte	Message	Prix
Bottes	157	20	OK	399,00 €
Brosse poil dur	56	60	ALERTE!!!	29,90 €
Savon doux	78	50	OK	23,40 €
Trousse secours	6	10	ALERTE!!!	245,00 €
Cirage noir	46	15	OK	34,95 €
Ampoule 40W	13	20	ALERTE!!!	5,40 €

Il faut donc créer des données intermédiaires permettant ce dernier affichage sur 5 colonnes.

Au lieu d'envoyer directement dans un TDBGrid le résultat de la requête << **SELECT** Magasin.CodeArticle, Magasin.DésignArticle, Magasin.QuantitéStock, Magasin.SeuilAlerte **FROM** Magasin **ORDER BY** Magasin.CodeArticle >>, nous l'envoyons dans des objets de champ persistant Delphi (cf. doc Delphi).

Un objet de champ persistant est en première approximation un moyen souple de stocker des informations de données, il est équivalent à une sorte de variable locale pour les données de la BD.

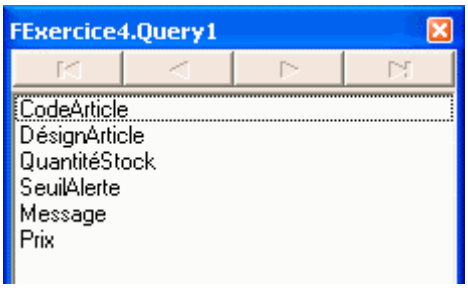
Il nous faut créer 6 objets de champ persistant : 4 pour le résultat de la requête + 1 pour le champ Prix de la TablePrix + 1 nouveau pour le message d'alerte. Chaque champ a un type correspondant exactement au type de données du champ de la BD associée (TstringField pour du texte, TintegerField pour du numérique, TcurrencyField pour du monétaire).

```
Query1CodeArticle: TStringField;
Query1DsignArticle: TStringField;
Query1QuantitStock: TIntegerField;
Query1SeuilAlerte: TIntegerField;
Query1Message: TStringField;
Query1Prix: TCurrencyField;
```

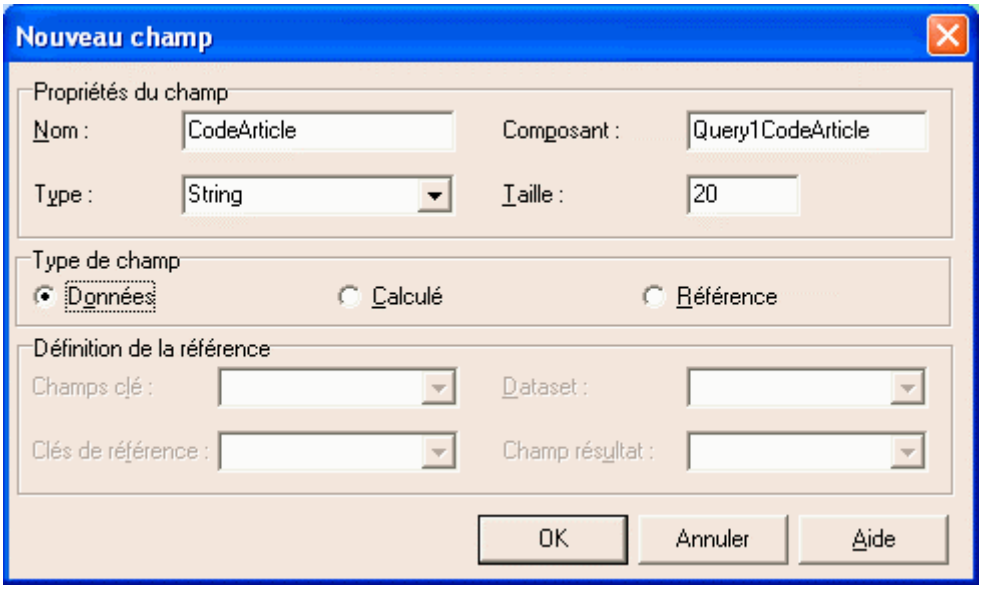
La documentation Delphi indique comment créer par programme de tels champ, par exemple ci-dessous la création de l'objet Query1CodeArticle: TStringField; lié au Tquery : Query1 :

```
Query1.Close;
Query1DsignArticle:= TStringField.Create(Self); // self représente la Tform de dépôt du Query1
Query1DsignArticle.FieldName := 'DesignArticle';
Query1DsignArticle.Index := Query1.FieldCount;
Query1DsignArticle.DataSet := Query1;
 Query1.FieldDefs.UpDate;
 Query1.Open;
```

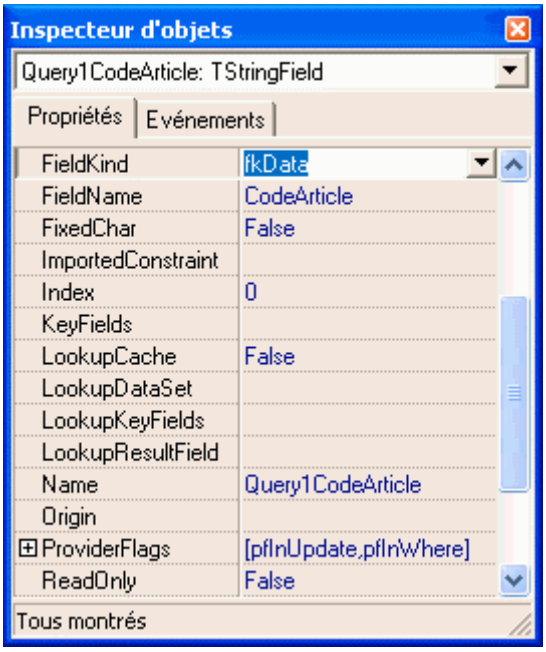
Il en est de même pour les 5 autres objets de champ. Delphi permet une création directement pendant la conception par double click sur le composant Query1, une interface de saisie d'une liste d'objets de champ apparaît. Ci dessous la liste obtenue après saisie des 6 objets :



Le premier objet de champ est saisi comme suit :

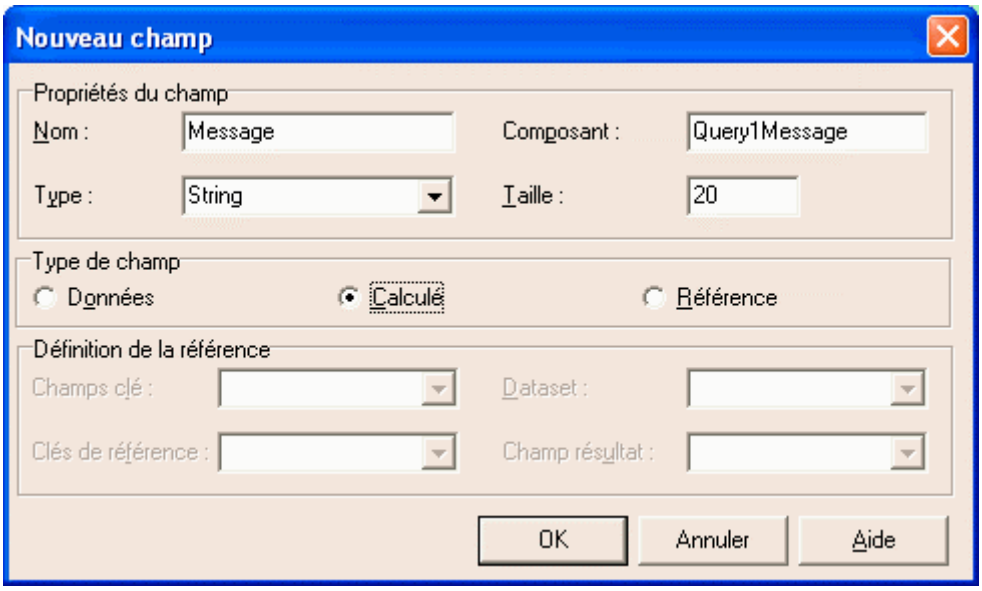


Voici sa vue dans l'inspecteur d'objet :

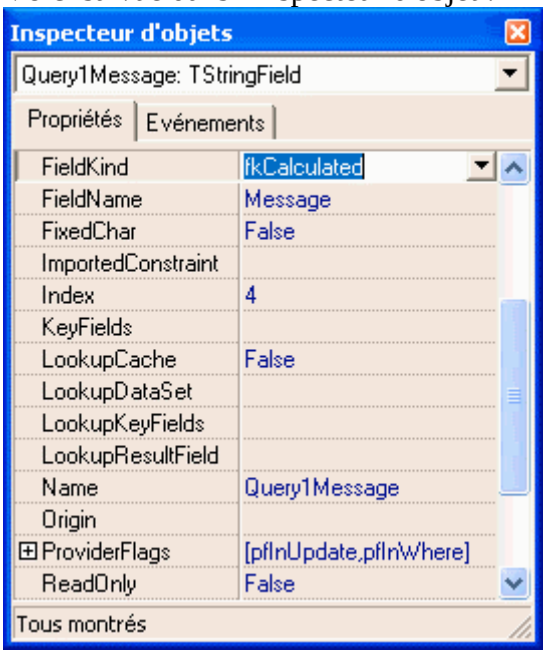


Les trois objets de champ suivants sont sur le même modèle (type de champ : données).

Le cinquième champ de Message est calculé :



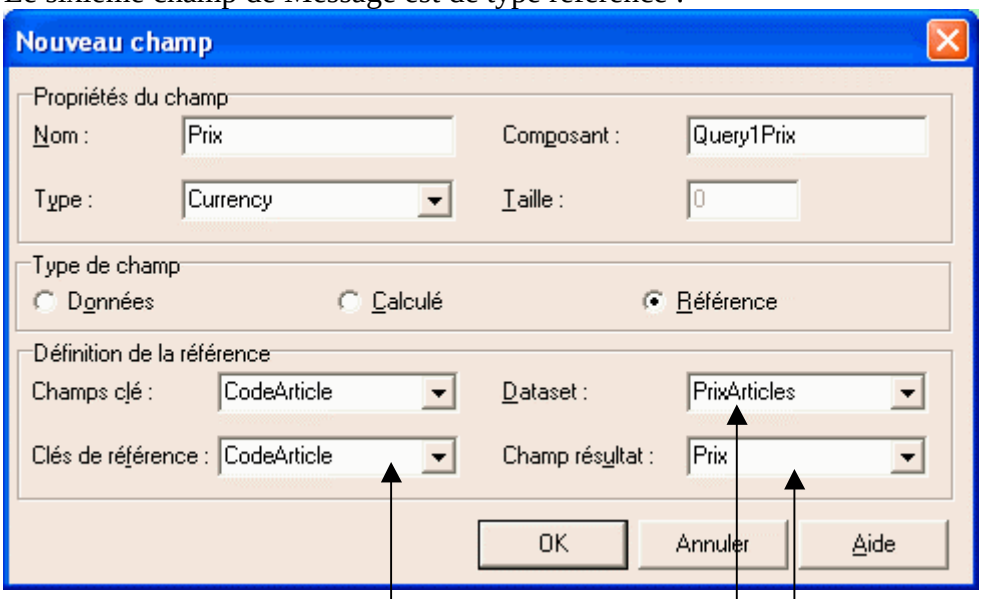
Voici sa vue dans l'inspecteur d'objet :



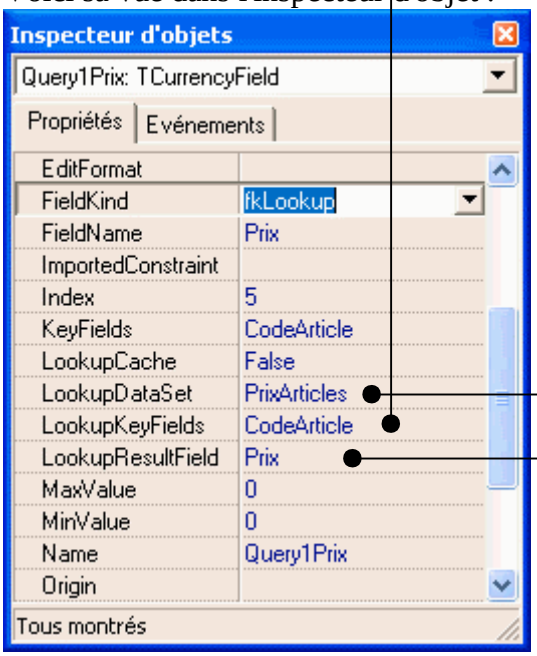
Le sixième et dernier champ de Prix est une référence à un champ existant dans la TablePrix, il est donc nécessaire de fournir :

- le nom de la table,
- le champ de clef primaire,
- le nom du champ où l'on va chercher la valeur correspondant à la clef

Le sixième champ de Message est de type référence :



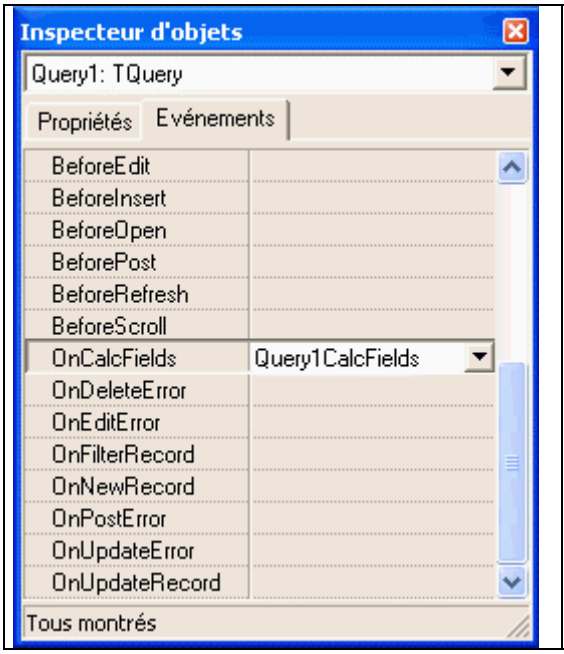
Voici sa vue dans l'inspecteur d'objet :



Voici le code de calcul d'alerte, permettant de mettre la valeur du message d'alerte en fonction du seuil d'alerte prévu :

```
if  Query1QuantitStock.value <= Query1SeuilAlerte.value  then
    Query1Message.value := 'ALERTE!!!'
else
    Query1Message.value := 'OK'
```

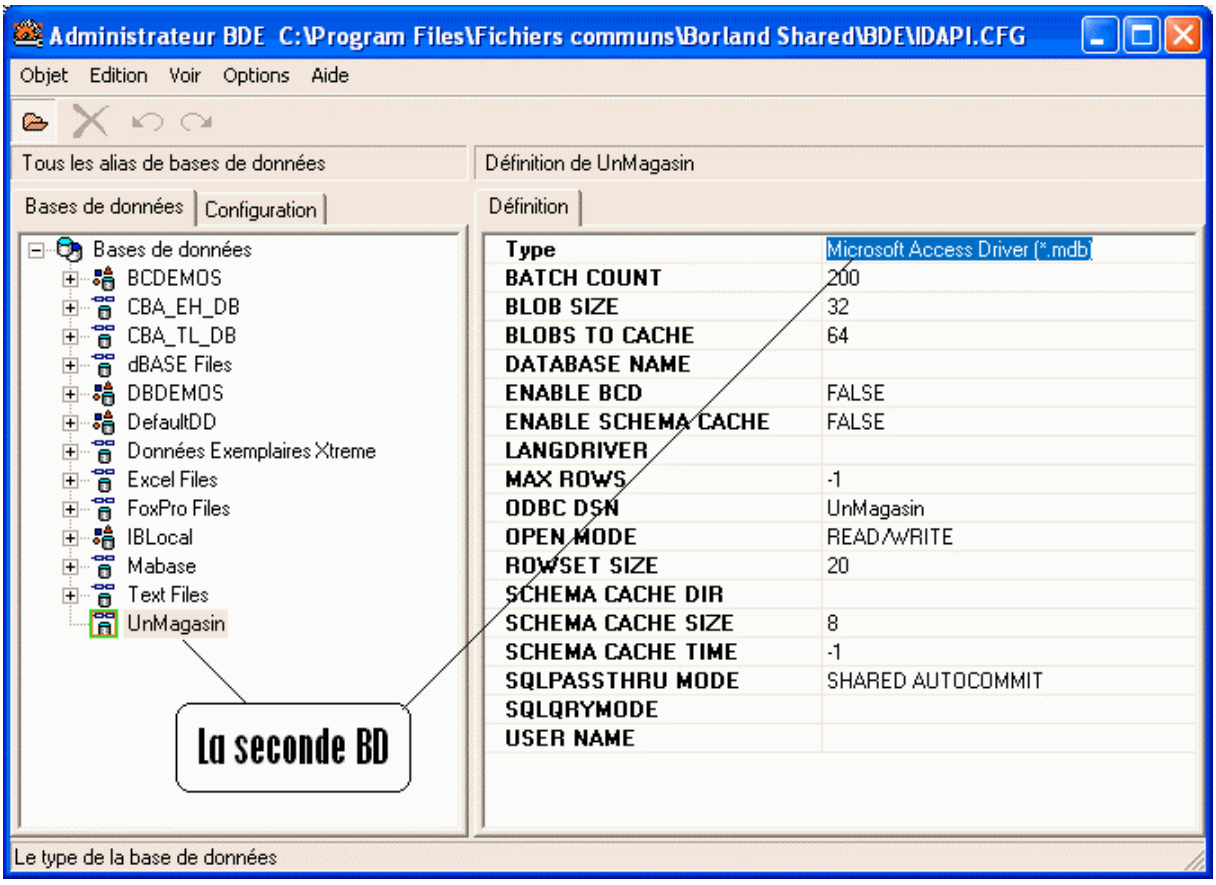
Le composant Query1 de type TQuery possède une événement OnCalcFields qui se produit lorsque l' application évalue les champs calculés. Le champ Query1Message est un champ calculé donc le code de calcul d'alerte peu être opportunément exécuté à chaque fois que la requête est lancée :



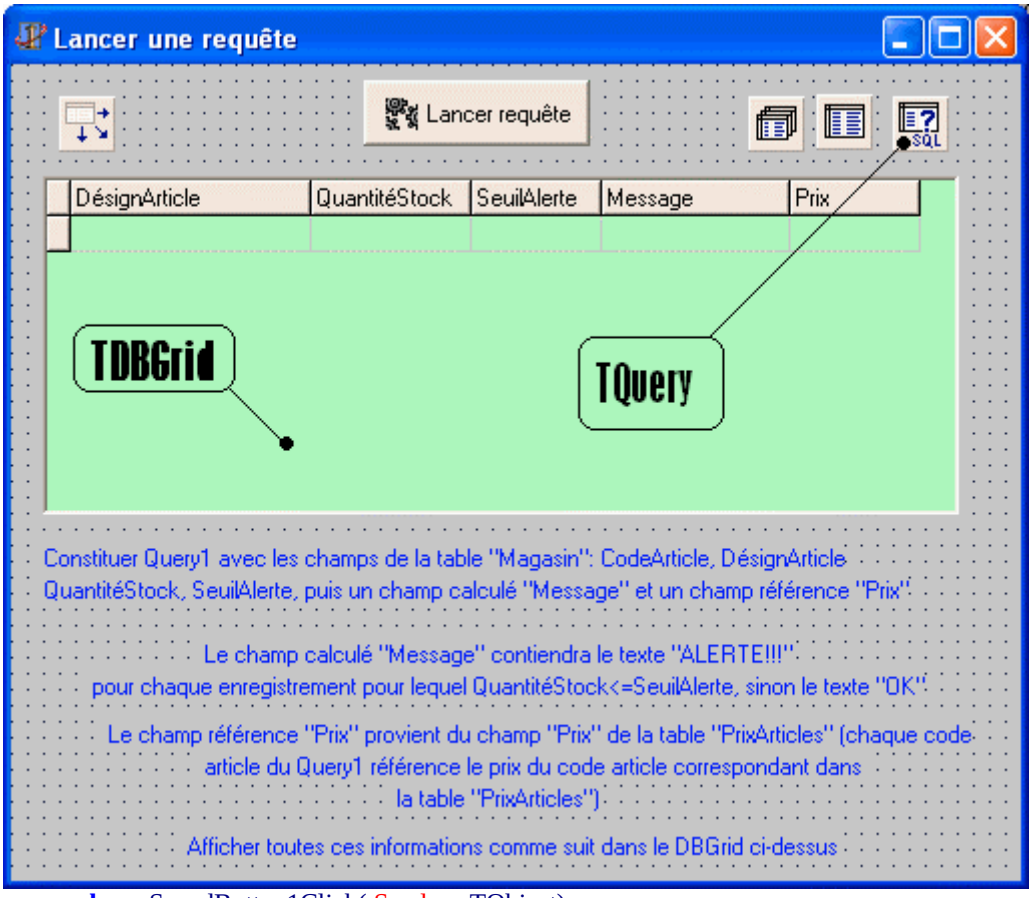

```
procedure Tform1.Query1CalcFields (DataSet: TDataSet);
begin
    // le code de calcul d'alerte
end;
```

4°) Code source delphi de l'exercice

Nous avons créé un nouvel alias de connexion physique par pilote ODBC dénommé "UnMagasin", pour une autre BD associée au fichier physique « **BaseExercice6.mdb** » :

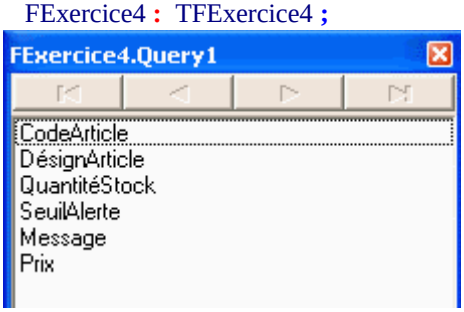


```
unit UBDExo4 ;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
 Buttons, Grids, DBGrids, Db, DBTables, StdCtrls ;

type
  TFExercice4 = class (TForm)
 Database1 : TDatabase ;
  Query1 : TQuery ;
  DataSource1 : TDataSource ;
  DBGrid1 : TDBGrid ;
  SpeedButton1 : TSpeedButton ;
  Query1DsignArticle : TStringField ;
  Query1QuantitStock : TIntegerField ;
  Query1SeuilAlerte : TIntegerField ;
  Query1Message : TStringField ;
  Label1 : TLabel ;
  Label2 : TLabel ;
  Label3 : TLabel ;
  Label4 : TLabel ;
  Label5 : TLabel ;
  Label6 : TLabel ;
  Label7 : TLabel ;
  Query1CodeArticle : TStringField ;
  PrixArticles : TTable ;
  Query1Prix : TCurrencyField ;
  Label8 : TLabel ;
```




```
  procedure SpeedButton1Click( Sender: TObject) ;
```

```
  procedure  FormCreate( Sender:  TObject) ;
  procedure  Query1CalcFields(DataSet :  TDataSet) ;
 private
  { Déclarations privées }
  public
  { Déclarations publiques }
end;

 var
   FExercice4 :  TFExercice4 ;
```




```
{

 On a créé comme indiqué plus haut, les 6 champs
 d'objets persistants pendant la conception .

}
```

```
implementation
var
  rep_appli :string ;

  {$R *.DFM}

 procedure  TFExercice4.SpeedButton1Click( Sender:  TObject) ;
 begin
   Query1.Open ;
 end;

 procedure  TFExercice4.FormCreate( Sender:  TObject) ;
 begin
    DataBase1.DatabaseName  :=  'UnMagasin';
    DataBase1.connected  :=  true  ;   //connexion sur la base de données

    PrixArticles.DatabaseName  :=  'UnMagasin';  //TTable connecté à la BD
    PrixArticles.TableName := 'PrixArticles';  // TTable connecté à  la table PrixArticles de la BD

    DBGrid1.DataSource := DataSource1 ;  // liaison TDBGrid <--> TDataSource

    Query1.DatabaseName  :=  'UnMagasin';
    Query1.SQL.Clear  ;
    Query1.SQL.Append(  'SELECT Magasin.CodeArticle, Magasin.DésignArticle,'  +
                                    'Magasin.QuantitéStock, Magasin.SeuilAlerte'  ) ;
    Query1.SQL.Append( 'FROM Magasin ORDER BY Magasin.CodeArticle' ) ;
    DataSource1.DataSet  :=  Query1  ;  // liaison TDataSource <--> TQuery
 end;

 procedure  TFExercice4.Query1CalcFields(DataSet :  TDataSet) ;
 begin
  if  Query1QuantitStock.value <= Query1SeuilAlerte.value  then
    Query1Message.value := 'ALERTE!!!'
  else
    Query1Message.value := 'OK'
 end;
end.
```

Etape n°1 le processus d'analyse

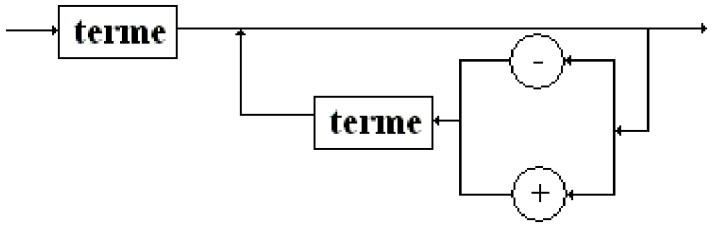
Nous suivons la stratégie d'analyse évoquée au chapitre 7.2.

- **Afin d'améliorer le traitement des données, nous ajoutons une sentinelle à l'expression, soit le caractère '.' choisi comme sentinelle.**
- **Afin de simplifier le code (l'objectif principal étant la construction de l'arbre abstrait) nous n'analysons que des expressions correctes, donc nous ne nous préoccuperons pas des tests d'erreurs.**

Nous procédons donc règles par règles et nous donnons au lecteur l'algorithme et l'implantation en Delphi de chaque bloc associé à une règle.

étude de la Règle-1 :

<expr> :



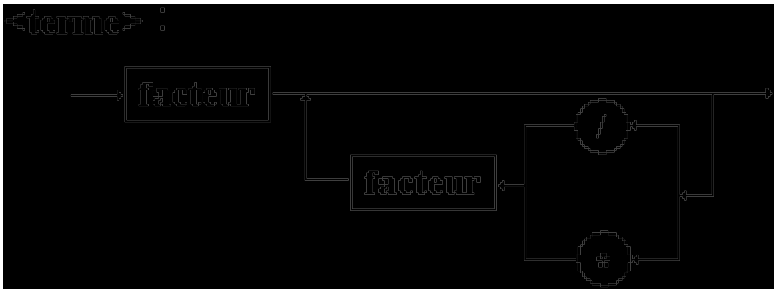
Bloc Analyser expr :

```
si SymLu ∈ Init(terme) alors
 Analyser terme ;
 tantque SymLu ∈ Init( {+,-} )faire
   Symsuivant;
   Analyser terme ;
 ftant;
 //si SymLu ∉ Init( {'.'} )Alors Erreur fsi
sinon Erreur
fsi
```

Implantation en Delphi :

```
procedure expr;
begin
   terme;
   while SymLu in ['+', '-'] do
     begin
       Symsuivant;
       terme;
     end
end;{expr}
```

étude de la Règle-2 :



Bloc Analyser terme :

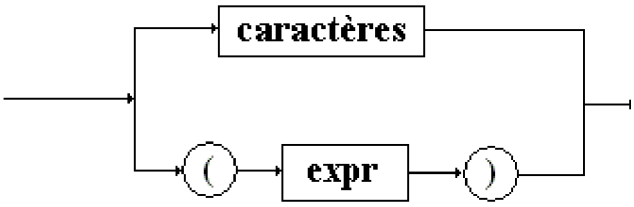
```
si SymLu ∈ Init(facteur) alors
 Analyser facteur ;
 tantque SymLu ∈ Init( { *, / } )faire
   Symsuivant;
   Analyser facteur ;
 ftant;
 //si SymLu ∉ Init( {+,-,*,/,'.'} )Alors Erreur fsi
fsi
```

Implantation en Delphi :

```
procedure terme;
begin
     facteur;
     while SymLu in ['*', '/'] do
       begin
         Symsuivant;
         facteur;
       end
 end;{terme}
```

étude de la Règle-3 :

<facteur> :



Bloc Analyser facteur :

```
si SymLu ∈ Init(caractères) alors
 Symsuivant
sinon
 si SymLu ∈ Init({'('}) alors
   Symsuivant;
   Analyser expr ;
   si SymLu ∈ Init({')'}) alors
     Symsuivant;
   fsi
 sinon Erreur
 fsi
fsi
```

Implantation en Delphi :

```
procedure facteur ;
begin
     if SymLu = '(' then
       begin
         Symsuivant;
         expr;
         if SymLu = ')' then
           Symsuivant
       end
     else // on est dans caractères car: expression correcte
       begin
         Symsuivant
       end
end;{facteur}
```

programme complet en Delphi :

Voici le programme Delphi obtenu pour analyser une expression correctement écrite, pour l'instant si l'on exécute ce programme tel quel, il ne produira rien, car nous sommes encore à la première étape du travail.

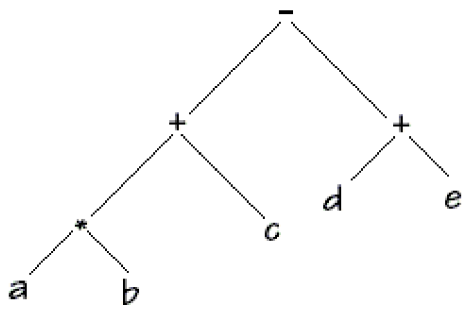
```
program expression;
var
    SymLu: char;
    numcar: integer;
    chaine: string;

 procedure init;
 begin
   numcar := 0;
 end;

 procedure Symsuivant;
 begin
    numcar := numcar + 1;
    SymLu := chaine[numcar]
 end;

procedure expr;
begin
    terme;
    while SymLu in ['+', '-'] do
      begin
        Symsuivant;
        terme;
      end
end;{expr}

procedure terme;
begin
      facteur;
      while SymLu in ['*', '/'] do
        begin
          Symsuivant;
          facteur;
        end
 end;{terme}
```

```
procedure facteur ;
begin
     if SymLu = '(' then
       begin
         Symsuivant;
         expr;
         if SymLu = ')' then
           Symsuivant
       end
     else // on est dans caractères car: expression correcte
       begin
         Symsuivant
       end
end;{facteur}

begin
  init;
  writeln('entrez une expression:');
  readln(chaine);
  chaine := concat(chaine, '.');
  Symsuivant;
  expr(x);
end.
```

Etape n°2 la construction de l'arbre abstrait

Nous allons stocker les différents symboles de l'expression au fur et à mesure de son analyse, dans un arbre binaire représentant l'arbre abstrait de l'expression comme nous l'avons déjà vu au chapitre 4.2 .

Pour l'exemple à partir de l'expression a*b + c-(d+e) nous construirons l'arbre abstrait figuré ci-dessous :



Implantation de la structure en Delphi avec variables dynamiques :

```
type
   parbre = ^arbre;
   arbre = record
               val: char;
               g, d: parbre
         end;
```

Nous proposons d'écrire une fonction de construction d'un arbre (un constructeur d'arbre)

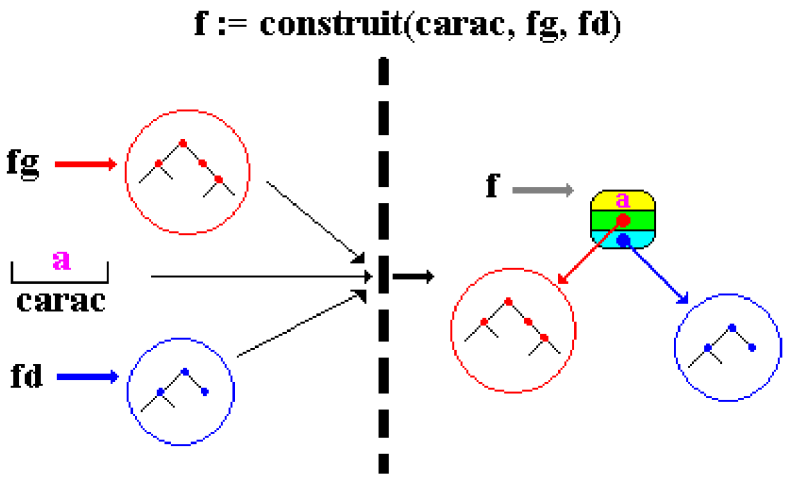
- La fonction reçoit en entrée 3 paramètres correspondant au contenu d'un noeud :
  - la valeur (un char ici),
  - le pointeur (ou référence) vers le fils gauche de ce noeud,
  - et le pointeur (ou référence) vers le fils droit de ce noeud.
- La fonction renvoie un fois le noeud construit, un pointeur (ou référence) sur le noeud nouvellement construit.

Voici le code de la fonction 'Construit' proposée selon le type parbre défini plus haut :

```
function Construit (c: char; fg, fd: parbre): parbre;
var  floc: parbre;
begin
   new(floc);
   with floc^ do
   begin
      val := c;
      g := fg;
      d := fd
   end;
   construit := floc
end;{construit}
```

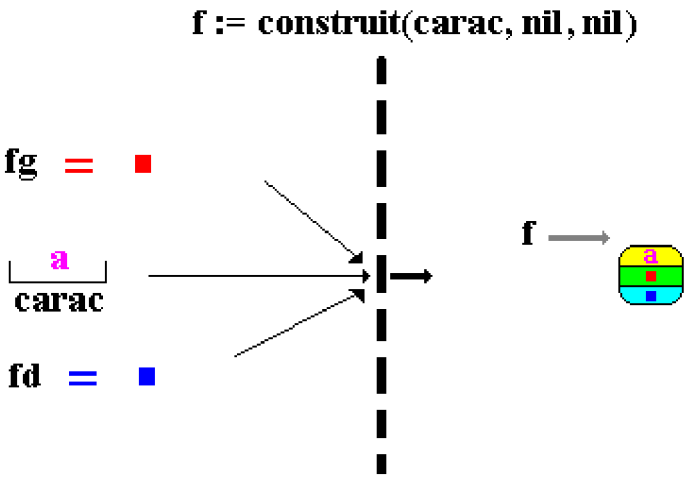
Ci-dessous sur deux exemples montrons ce que produit la **function** Construit(..).

Exemple -1 : la construction d'un noeud ayant deux descendants
(un sous-arbre gauche nommé **fg**, et un sous-arbre droit nommé **fd**)



A la fin de la construction la fonction construit a relié l'arbre gauche rouge avec l'arbre droit bleu dans le noeud pointé par **f**, respectivement comme sous-arbre gauche rouge et comme sous-arbre droit bleu, le champ valeur du noeud **f** contient le caractère '**a**'.

Exemple -2 : la construction d'une feuille



génération pour la Règle-1 :

Bloc générer arbre - expr :

```
si SymLu ∈ Init(terme) alors
 Analyser terme ;
 tantque SymLu ∈ Init( {+,-} )faire
   Symsuivant;
   Analyser terme ;
   construire arbre
 ftant;
 //si SymLu ∉ Init( {'.'} )Alors Erreur fsi
sinon Erreur
fsi
```

Implantation en Delphi :

```
procedure expr (var f: parbre) ;
var
      carloc: char;
      ft: parbre;
 begin
    terme(f);
    while SymLu in ['+', '-'] do
      begin
        carloc := SymLu;
        Symsuivant;
        terme(ft);
        f := construit(carloc, f, ft){construction de l'arbre}
      end
end;{expr}
```

génération pour la Règle-2 :

Bloc générer arbre - terme :

```
si SymLu ∈ Init(facteur) alors
 Analyser facteur ;
 tantque SymLu ∈ Init( { *, / } )faire
   Symsuivant;
   Analyser facteur ;
```

```
   construire arbre
 ftant;
 //si SymLu ∉ Init( {+,-,*,/,'.'} )Alors Erreur fsi
fsi
```

Implantation en Delphi :

```
procedure terme (var f: parbre);
    var
      carloc: char;
      floc: parbre;
  begin
    facteur(f);
    while SymLu in ['*', '/'] do
      begin
        carloc := SymLu;
        Symsuivant;
        facteur(floc);
        f := construit(carloc, f, floc){construction de l'arbre}
      end
  end;{terme}
```

génération pour la Règle-3 :

Bloc générer arbre - facteur :

```
si SymLu ∈ Init(caractères) alors
 Symsuivant
sinon
 si SymLu ∈ Init({'('}) alors
   Symsuivant;
   Analyser expr ;
   si SymLu ∈ Init({')'}) alors
     Symsuivant;
   fsi
  sinon Erreur
 fsi
fsi
```

Implantation en Delphi :

```
procedure facteur (var f: parbre);
begin
    if SymLu = '(' then
      begin
        Symsuivant;
        expr(f);
        if SymLu = ')' then
          Symsuivant
      end
    else
      begin
        f := construit(SymLu, nil, nil); {construction de l'arbre }
        Symsuivant
      end
end;{facteur}
```

programme complet en Delphi :

Voici un programme Delphi obtenu pour engendrer un arbre abstrait d'une expression correctement écrite; ici les procédures terme et facteur ont été incluses dans la partie déclaration de la procédure expr, mais elles peuvent être dissociées et se retrouver au même niveau de déclaration sans rien changer au fonctionnement du programme.

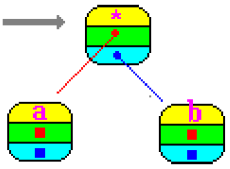
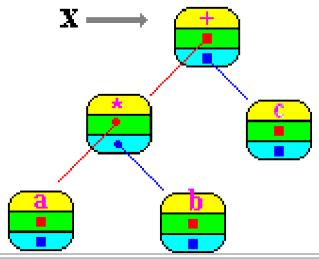
```
program expression;
{les expressions entrées sont correctes }
 type
   parbre = ^arbre;
   arbre = record
             val: char;
             g, d: parbre
           end;
 var
   x: parbre;
   SymLu: char;
   numcar: integer;
   chaine: string;

 procedure init;
 begin
   numcar := 0;
   x := nil;
 end;

 function construit (c: char; fg, fd: parbre): parbre;
   var
     floc: parbre;
 begin
   new(floc);
   with floc^ do
     begin
       val := c;
       g := fg;
       d := fd
     end;
   construit := floc
 end;{construit}

 procedure Symsuivant;
 begin
   numcar := numcar + 1;
   SymLu := chaine[numcar]
 end;

{construction d'un arbre d'expressions à partir de la grammaire :}
 procedure expr (var f: parbre);
   var
     carloc: char;
     ft: parbre;
   procedure facteur (var f: parbre);
   begin
     if SymLu = '(' then
       begin
         Symsuivant;
         expr(f);
         if SymLu = ')' then
           Symsuivant
       end
     else
```

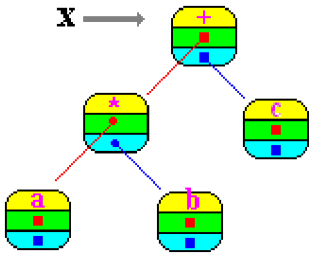
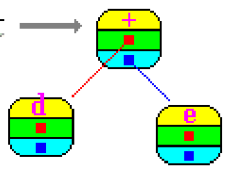
```
        begin
          f := construit(SymLu, nil, nil); {construction de l'arbre }
          Symsuivant
        end
    end;{facteur}

    procedure terme (var f: parbre);
      var
        carloc: char;
        floc: parbre;
    begin
      facteur(f);
      while SymLu in ['*', '/'] do
        begin
          carloc := SymLu;
          Symsuivant;
          facteur(floc);
          f := construit(carloc, f, floc){construction de l'arbre}
        end
    end;{terme}

  begin{expr}
    terme(f);
    while SymLu in ['+', '-'] do
      begin
        carloc := SymLu;
        Symsuivant;
        terme(ft);
        f := construit(carloc, f, ft){construction de l'arbre}
      end
  end;{expr}

begin
  init;
  writeln('entrez une expression:');
  readln(chaine);
  chaine := concat(chaine, '.');
  Symsuivant;
  expr(x);
end.
```

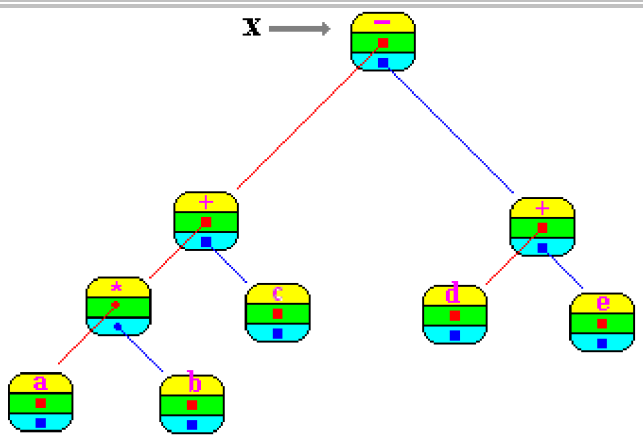
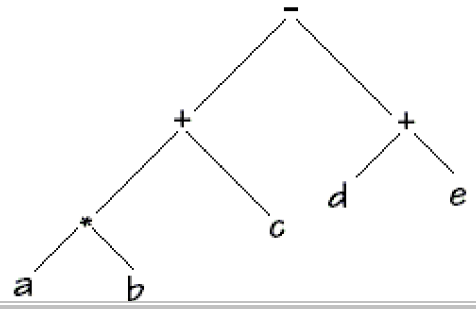
Suivons à titre d'exemple la construction par le programme précédent, de l'arbre abstrait de l'expression déjà citée plus haut soit : **a*b + c-(d+e)**.

Nous effectuons une trace pas à pas du début de l'exécution sur l'expression :

Procédures appelées	Résultat produit
readln(chaine);	chaine = **a*b + c-(d+e)**.
init;	x = **nil**
Symsuivant	SymLu = **a** , dans **a*b + c-(d+e)**.
expr(x) |terme(x) | |facteur(x) | | |x := construit(SymLu, **nil**, **nil**); | | |Symsuivant	**x** →  SymLu = * , dans **a*b + c-(d+e)**.
expr(x)	carloc = *

|terme(x) | |facteur(x) | |SymLu **in** ['*', '/'] = true | carloc := SymLu; | Symsuivant; | facteur(floc);	SymLu = **b** , dans **a*b + c-(d+e)**. floc = **nil**
expr(x) |terme(x) | |facteur(floc); | | |floc := construit(SymLu, **nil**, **nil**); | | |Symsuivant	*carloc* = * **floc** →  SymLu = + , dans **a*b + c-(d+e)**.
expr(x) |terme(x) | |facteur(floc); | |x := construit(carloc, x, floc)	*carloc* = * **x** → 
expr(x) |terme(x); |SymLu **in** ['+', '-'] = true | carloc := SymLu; | Symsuivant; | terme(ft);	carloc = + SymLu = **c** , dans **a*b + c-(d+e)**. ft = **nil**
expr(x) |terme(x) | |terme(ft) | | |facteur(ft); | | | |ft := construit(SymLu, **nil**, **nil**); | | | |Symsuivant	*carloc* = + **ft** →  SymLu = - , dans **a*b + c - (d+e)**.
expr(x) |terme(x) | |terme(ft) | | |facteur(ft); | | |x := construit(carloc, x, ft)	*carloc* = + **x** → 
expr(x) |terme(x) | |terme(ft) | |facteur(ft) | | |SymLu **in** ['*', '/'] = false |SymLu **in** ['+', '-'] = true | carloc := SymLu; | Symsuivant; | |terme(ft);	carloc = - SymLu = **(** , dans **a*b + c-(d+e)**. ft = **nil**
expr(x) |terme(x) | |facteur(ft); | | |SymLu = '(' = true | | Symsuivant; | | |expr(ft); | | |terme(ft); | | | |facteur(ft);	carloc = - SymLu = **d** , dans **a*b + c-(d+e)**. ft = **nil**
expr(x) |terme(x)	*carloc* = -

| |facteur(ft) | | |expr(ft) | | | |terme(ft) | | | | |facteur(ft) | | | | |ft := construit(SymLu, **nil**, **nil**); | | | | |Symsuivant;	**ft** →  SymLu = **+** , dans **a*b + c-(d+e)**.
expr(x) |terme(x) | |facteur(ft) | | |expr(ft) | | | |terme(ft) | | | | |facteur(ft) | | | | |SymLu **in** ['*', '/'] = false | | | |SymLu **in** ['+', '-'] = true | | | | carloc := SymLu; | | | | Symsuivant; | | | | |terme(ft); | | | | | |facteur(ft);	*dans la pile :* --- *carloc* = - **ft** →  --- carloc = + SymLu = **e** , dans **a*b + c-(d+e)**. ft = **nil**
expr(x) |terme(x) | |facteur(ft) | | |expr(ft) | | | |terme(ft) | | | | |terme(ft) | | | | | |facteur(ft) | | | | | | |ft := construit(SymLu, **nil**, **nil**); | | | | | | |Symsuivant;	*dans la pile :* --- *carloc* = - **ft** →  --- carloc = + **ft** →  SymLu = **)** , dans **a*b + c-(d+e)**.
expr(x) |terme(x) | |facteur(ft) | | |expr(ft) | | | |terme(ft) | | | | |terme(ft) | | | | | |facteur(ft) | | | | | | |ft := construit(carloc , *ft* , ft) | | | | | |SymLu **in** ['*', '/'] = false	*dans la pile :* --- *carloc* = - **x** →  --- **ft** → 
expr(x) |terme(x) | |facteur(ft)	

		|expr(ft) |		|terme(ft) |		|SymLu **in** ['+', '-'] = false | |x := construit(carloc, x, ft) |SymLu **in** ['*', '/'] = false Fin	
Arbre abstrait obtenu après exécution :	

Si le lecteur souhaite visualiser les contenus des arbres abstraits, il peut utiliser les 3 parcours en profondeur vus au chapitre 4.3, nous listons ci-dessous les 3 procédures associées au type parbre de l'exercice ainsi que le corps du programme principal appellant ces procédures :

```
program expression;
{les expressions sont correctes et elles sont lues sous les 3 formes
in,post et pré-fixées }
  type
    parbre = ^arbre;
    arbre = record
                  val: char;
                  g, d: parbre
            end;
  var
    x: parbre;
    SymLu: char;
    numcar: integer;
    chaine: string;

  procedure init;
  begin
    numcar := 0;
    x := nil;
  end;

  function construit (c: char; fg, fd: parbre): parbre;
    var
      floc: parbre;
  begin
    new(floc);
    with floc^ do
      begin
```

```
        val := c;
        g := fg;
        d := fd
      end;
    construit := floc
  end;{construit}

  procedure Symsuivant;
  begin
    numcar := numcar + 1;
    SymLu := chaine[numcar]
  end;

 {construction d'un arbre d'expressions à partir de la grammaire :}
  procedure expr (var f: parbre);
    var
      carloc: char;
      ft: parbre;
    procedure facteur (var f: parbre);
    begin
      if SymLu = '(' then
        begin
          Symsuivant;
          expr(f);
          if SymLu = ')' then
            Symsuivant
        end
      else
        begin
          f := construit(SymLu, nil, nil); {construction de l'arbre }
          Symsuivant
        end
    end;{facteur}

    procedure terme (var f: parbre);
      var
        carloc: char;
        floc: parbre;
    begin
      facteur(f);
      while SymLu in ['*', '/'] do
        begin
          carloc := SymLu;
          Symsuivant;
          facteur(floc);
          f := construit(carloc, f, floc){construction de l'arbre}
        end
    end;{terme}

  begin{expr}
    terme(f);
    while SymLu in ['+', '-'] do
      begin
        carloc := SymLu;
        Symsuivant;
        terme(ft);
        f := construit(carloc, f, ft){construction de l'arbre}
      end
  end;{expr}

 procedure edite (f: parbre);
```

```
  {infixe parenthesee}
  begin
    if f <> nil then
      with f^ do
        begin
          write('(');
          edite(g);
          write(val);
          edite(d);
          write(')')
        end
  end;

  procedure postfixe (f: parbre);
  {sert dans les machine a piles a la compilation}
  begin
    if f <> nil then
      with f^ do
        begin
          postfixe(g);
          postfixe(d);
          write(val);
        end
  end;

  procedure prefixe (f: parbre);
  begin
    if f <> nil then
      with f^ do
        begin
          write(val);
          prefixe(g);
          prefixe(d)
        end
  end;

  procedure infixe (f: parbre);
  begin
    if f <> nil then
      with f^ do
        begin
          infixe(g);
          write(val);
          infixe(d);
        end
  end;

begin
  init;
  writeln('entrez une expression:');
  readln(chaine);
  chaine := concat(chaine, '.');
  Symsuivant;
  expr(x);
  writeln('Expression parenthésée:');
  edite(x);
  writeln;
  writeln('Expression notation infixée:');
  infixe(x);
  writeln;
```

```
    writeln('Expression notation postfixée:');
    postfixe(x);
    writeln;
    writeln('Expression notation préfixée:');
    prefixe(x);
    writeln
  end.
```