Programmation, Volume VII :

Style Objets et Langages [à/orientés] Objets

Gilles Hunault

Université d'Angers, 1999

1. Concepts et vocabulaire de la P.O.

En théorie des graphes, on manipule des éléments et des liaisons, nommés respectivement noeuds et arcs[1] par certain(e)s dans le cas orienté, sommets et arêtes par d'autres, dans le cas non-orienté. On trouve en P.O. (Programmation Objets) le même flou dans les désignations. La "grande tradition" parle de Classe avec des champs et des méthodes, bien qu'on trouve souvent les termes de *modèles*, avec leurs *attributs* et *propriétés* ou encore d'*entités* avec leurs *données* et *comportements*. La première difficulté est donc de se retrouver dans le vocabulaire.

L'enjeu de la P.O. est pourtant fondamental : permettre de bien programmer pour les grandes applications. Il ne s'agit pas *a priori* d'un style d'écriture mais d'une volonté d'être efficace, fiable, réutilisable. Si une programmation classique peut suffire pour un système de facturation, il n'est pas possible de se permettre la moindre erreur pour un programme de surveillance aérienne, par exemple. La complexité inhérente aux grandes applications et à la qualité industrielle des logiciels oblige à un fort niveau d'abstraction, à une grande rigueur, à une coopération (forcée) entre programmes et programmeurs.

[1] on reconnait bien là un certain écart de langage qu'affectionnent les mathématiques : tout être humain normalement constitué nommerait *flèche* et non pas *arc* une liaison orientée.

Apprendre un nouveau langage traditionnel (Pascal, Rexx, C...) se fait surtout par la découverte de la nouvelle syntaxe. Par exemple la boucle POUR connue comme `for (i=1;i<=n;i++) { ... }` doit être réapprise en `do i=1 to n ... end` ou en `for i := 1 to n do begin ... end`. Par contre, apprendre le style objets se fait plus difficilement : il faut d'abord comprendre les enjeux, repenser sa façon d'analyser et de construire les programmes, puis se défaire des habitudes acquises. La deuxième difficulté est donc de maitriser les concepts.

Une des premières difficultés à lever est la duplication de code, l'utilisation de mêmes fichiers de sous-programmes car ces comportements se révélent à la longue dangereux, voire causes d'incohérences graves. Pour pallier à de tels inconvénients, et malgré la compilation séparée, en réponse au développement anarchique de programmes, sous-programmes, modules, unités, librairies, bibliothèques, la P.O. propose une structuration des variables et actions en classes (noms) et méthodes (verbes) et une utilisation de classes standards (tableaux, fenêtres, boutons, flots...).

Programmer en P.O., c'est donc d'abord réfléchir pour oraniser les données et les actions correspondantes. Ensuite, on masque les détails d'implémentation et on fait "hériter" les objets d'actions définies dans des classes plus générales. La notion de masquage a fini par aboutir à celle d'encapsulation, qui revient à dire "qu'on ne doit voir que ce qu'il faut", c'est à dire les grandes lignes mais pas les détails. Ainsi, pour réaliser une analyse statistique sur un fichier de données, la lecture d'une ligne de données doit ajouter des valeurs aux variables statistiques. Avec une vision naïve, ce que nous pouvons nommer "objet-fichier" viendra envoyer le "message" `Ajouter des Valeurs` aux "objets-variables statistiques".

Par exemple si la ligne numéro 3 du fichier est

```
M003    31    1     4
```

alors il faut ajouter la valeur 31 à la variable numéro 2 (qui est ici une variable d'age). L'action fondamentale `Ajouter des Valeurs` ne doit pas faire apparaitre à ce niveau, le numéro de la valeur à ajouter. Elle ne doit pas non plus montrer explicitement l'effet annexe qui consiste à dire que la taille de la variable (utile pour calculer la moyenne) est de 3. Là ou la programmation classique viendrait écrire quelquechose comme

```
ajouterValeur(2,31)
```

la P.O. vient écrire

```
VariableSt[2].AjouterValeur(31)
```

ce qui montre la volonté de "localiser" les programmes aux "objets". L'effet annexe dont nous parlions vient alors s'écrire

```
VariableSt[2].taille <-- VariableSt[2].taille  + 1
```

et c'est à la variable statistique qu'il revient d'effectuer cette incrémentation.

Où trouver ce programme ? En programmation classique, on "enfourne", chacun essayant de gérer au mieux découpage logique et découpage physique. En P.O., il faut toute une organisation...

Prenons un exemple simple, celui du tri de valeurs pour impression. Nommons A un tableau d'entiers et B un tableau de réels et disons qu'avant d'imprimer ces tableaux, nous avons besoin de les trier. Intellectuellement, on conçoit bien qu'en nommant triT le module de tri et impT le module d'impression. la suite d'instructions

```
triT(A)  ;  impT(A)  ;  triT(B)  ;  impT(B)
```

résoud notre problème. Les ennuis commencent pourtant déjà : il n'est pas possible dans un langage typé classique (style *Pascal*) d'avoir un même nom pour deux procédures différentes. Le Pascal oblige à écrire

```
procedure triTe(var tA : tabEntiers)
procedure triTr(var tB : tabReels)
```

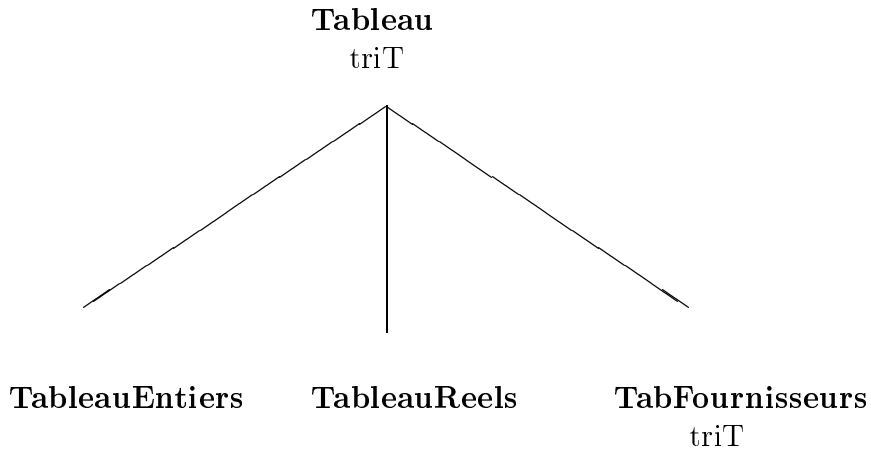
alors que la seule différence entre les procédures résulte du type des données. L'homonymie, interdite ici serait pourtant de mise. De plus, rien de permet de prédire *a priori* où ces procédures seront écrites.

En P.O., on viendra inventer une "classe" générale Tableau qui aura deux "sous-classes" TableauEntiers et TableauReels, on viendra écrire la "méthode" triT au niveau général et les deux sous-classes viendront "hériter" de cette méthode. L'héritage est donc une façon de contourner l'homonymie, permettant de partager des variables, des procédures à un niveau général. De plus, les procédures sont "liées" aux classes dont on sait où aller les chercher.

Mieux, si besoin est, on pourra adjoindre d'autre sous-classes fonctionnant sur le même principe qu'un tableau (par exemple une liste de fournisseurs) et "raffiner" la méthode triT en redéfinissant la procéure correspondante pour la classe considérée.

Résumons-nous : au lieu d'inventer pour des tableaux similaires pour lesquels seul change le type des données, on invente un modèle ou moule ou classe (le mot est lancé) et on lui associe une procédure de tri. Tant que celle-ci suffit, toutes les sous-classes qui en héritent peuvent l'utiliser. Si besoin est, on invente une autre procédure mais on a le droit de lui donner le même nom. On nomme alors "instance de classe" toute variable (on dit plutot objet) créée à partir de cette classe. Ainsi A est une instance de TableauEntiers, B une instance de TableauReels. A l'écriture traditionnelle `triT(A)` on préfére la notation `A.triT()` qui montre mieux la dépendance entre méthode et objet.[2]

Afin de savoir d'où vient la définition de la méthode, il est d'usage de tracer ce qu'on nomme graphe d'héritage. On ne remet que les méthodes "raffinées". Ainsi, pour nos trois tableaux, le graphe d'héritage ressemblerait à



[2] de plus, s'il y avait des paramètres, `A.triT(x,y)` est "conceptuellement" plus parlant que `triT(A,x,y)`.

Au découpage des procédures se superpose un découpage des données. C'est de là que vient la richesse de la P.O. et c'est aussi ce qui fait sa difficulté. Le programmeur classique qui avait l'habitude de penser "d'abord on fait ceci, puis on fait cela" est souvent déboussolé car il ne sait plus par où commencer.

La démarche est bien la même qu'en programmation classique, sauf qu'au lieu de chercher les actions (les verbes) pour en déduire les modules, on vient ici chercher les objets (les noms) et les actions. On doit ensuite hiérarchiser ces objets, trouver des modèles généraux communs que l'on nomme classes et y faire rentrer les modules.

Le gain de productivité est gagné dans la réutilisation du code. SI la première ligne de code coute cher en P.O., la dix-millième est peu chère, en comparaison de la programmation classique. De plus comme les objets sont réutilisables, le programmeur enrichit régulièrement ses classes et de nouveaux logiciels sur des données similairaires sont infiniment plus faciles à réécrire.

En particulier, l'attachement (ou plutot la localisation) des méthodes aux classes facilite la recherche des sous-programmes déjà écrits, évite la redondance. Les différents niveaux d'organisation des classes avec l'héritage et le raffinement permettent de partager ce qui est commun et de définir ce qui est particulier, d'où une grande clarté de relecture.

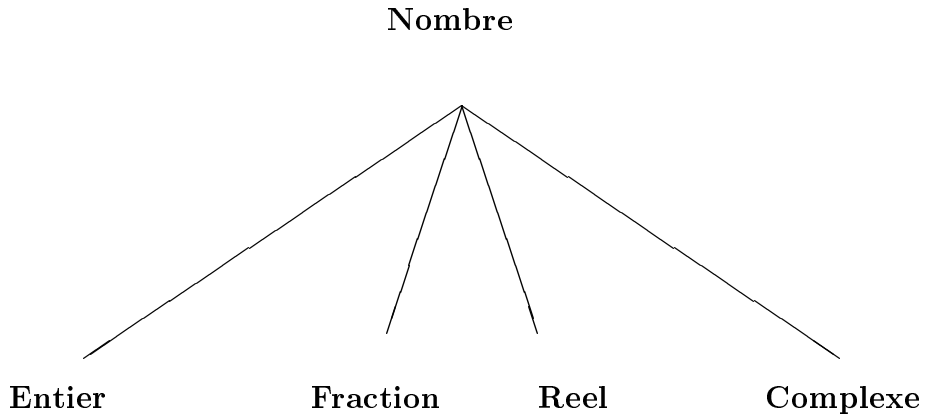
Le prix à payer pour une telle programmation est cependant élevé. D'abord en conceptualisation parce qu'il faut se défaire d'une vision trop "linéaire" des actions. Ensuite en investissement car pour éviter de réinventer les objets, il faut apprendre les classes existantes, leurs méthodes. Par exemple *Visual Dbase* compte 27 classes, dont la classe "Form" (ou formulaire ou fenêtre générale). *Visual Dbase* définit 40 champs pour la classe Form avec 24 méthodes liées directement aux évènements et 14 méthodes non liées directement aux évènements. Dans le même genre d'idées, *SmallTalk* compte plus d'une centaine de classes et aux environs de 2000 méthodes. On comprend bien qu'alors le programmeur se trouve déchargé de tout réinventer mais que par contre il lui incombe de tout apprendre...

2. Exemple : Nombres en P.O.

Imaginons que l'on doive, pour un logiciel d'édition scientifique, gérer des nombres de diverses sortes (entier, réels, complexes, fractionnaires). L'inclusion traditionnelle des mathématiciens

$$\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C} \subset \mathbb{H} \subset \mathbb{O}$$

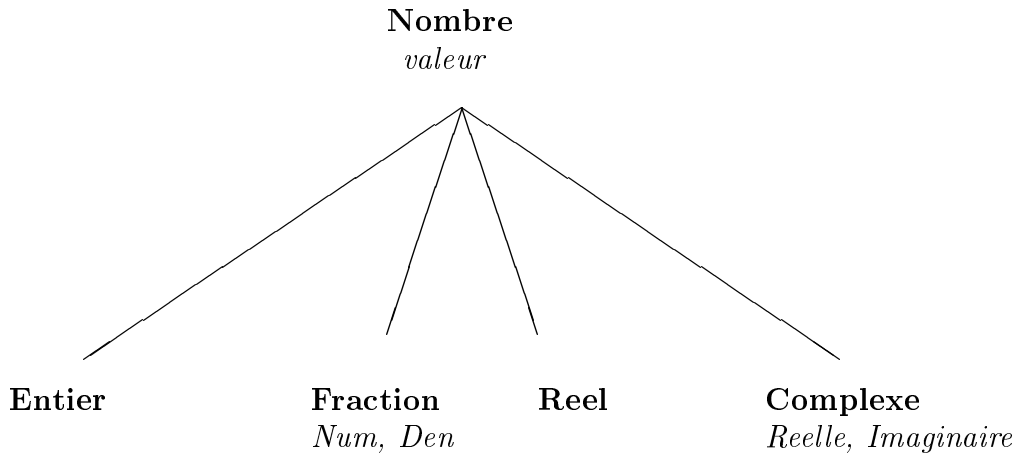
est de peu d'intérêt ici puisqu'elle ne s'intéresse pas à la représentation interne de ces nombres. Pour la P.O., on verra ces nombres comme différents, soit le graphe des données



mais dérivants tous d'une classe commune (artificielle) Nombre. On nomme Entier, Fraction, Reel, Complexe les classes correspondantes et suivant le point de vue elles sont "sous-classes" de Nombre ou Nombre est leur "super-classe".

A un nombre est en général associée une valeur (en général seulement, puisque pour une fraction et un nombre complexe, ce sont deux valeurs qui son associées). On nomme champ d'une classe, d'un objet, toute variable définie dans la classe. Ici, on viendra donc logiquement définir un champ valeur comme variable de base. Les classes Fraction et Complexe se verront dotées de champs supplémentaires, respectivement Num, Den et Reelle, Imaginaire.

Le graphe précédent peut alors être complété en



où, pour des raisons de lisibilité, nous n'avons pas répété le mot *valeur* pour les sous-classes alors que ce champ est hérité par toutes ces sous-classes. Concrétement, en Pascal Objet, de telles constructions s'écrivent simplement via les instructions

```
TYPE
   Nombre = Object
    Valeur : Real ;
   end ;

   Entier = Object(Nombre)
   end ;

   Reel = Object(Nombre)
   end ;

   Fraction = Object(Nombre)
     Num, Den : real ;
   end ;

   Complexe = Object(Nombre)
     Reelle, Imaginaire : real ;
   end ;
```

Les objets apparaissent donc comme des types particuliers de variables et un programme principal qui veut utiliser ces objets peut se contenter de comporter les lignes

```
var N : Nombre ;
    E : Entier ;
    R : Fraction ;
    X : Reel ;
    Z : Complexe ;
```

Passons maintenant à des méthodes élémentaires comme lire et écrire. Définissons lire comme `readln` au niveau le plus général et venons raffiner seulement la lecture des fractions et des nombres complexes. On viendra donc implémenter les méthodes dont les noms complets sont Nombre.Lire, Fraction.Lire et Complexe.Lire ; par contre Entier.Lire et Reel.Lire ne sont pas à écrire puisque Nombre.Lire est suffissant. Le texte du programme Pascal doit alors être modifié en conséquence :

```
TYPE
   Nombre = Object
    Valeur : Real ; procedure Lire ;
   end ;

   Entier = Object(Nombre) end ;

   Reel = Object(Nombre) end ;

   Fraction = Object(Nombre)
     Num, Den : real ; procedure Lire ;
   end ;

   Complexe = Object(Nombre)
     Reelle, Imaginaire : real ; procedure Lire ;
   end ;

   procedure Nombre.Lire ; begin
     write(' Entrez un nombre ') ;
     readln(Valeur)
   end ;
```

```
procedure Fraction.Lire ; begin
  write(' Entrez le numérateur ') ;
  readln(Num) ;
  write(' Entrez le dénominateur ') ;
  readln(Den) ;
  Valeur := Num / Den
end ;

procedure Complexe.Lire ; begin
  write(' Partie réelle     ? ' ) ;  readln(Reelle)  ;
  write(' Partie imaginaire ? ' ) ;  readln(Imaginaire)  ;
end ;
```

On doit commencer à entrevoir ici la richesse et la difficulté de la P.O.. La richesse, c'est de pouvoir réutiliser les mêmes instructions d'une classe à l'autre pour ce qui est commun ; la difficulté, c'est de bien structurer au départ. Ainsi, on aurait pu définir un nombre complexe comme composé de deux nombres réels avec les déclarations

```
complexe = Object(Nombre)
  Rl,Im : Reel ; procedure Lire ;
end ;
```

au lieu de

```
complexe = Object(Nombre)
  Rl,Im : real ; procedure Lire ;
end ;
```

Rien n'interdit même de "fricoter" avec des fractions en partie réelle et imaginaire avec les déclarations

```
complexe = Object(Nombre)
  Rl,Im : Fraction ; procedure Lire ;
end ;
```

3. Exemple : Analyse Statistique en P.O.

Passons maintenant à un programme plus conséquent. Le but du programme est ici de réaliser l'analyse statistique élémentaire d'un fichier de données, comme par exemple le fichier Elf.dat dont voici un extrait

```
#
# Id    Age       Sexe  Etud
#
  M001   62          1    2
  M002   60          0    3
  M003   31          1    4
  M004   27          1    4
  M005   22          0    4
  M006   70          1    1
  M007   19          1    4
  M008   53          1    2
  M009   62          0    4
  M010   63          1    0
...
```

à l'aide d'un fichier de description des variables statistiques (colonnes). Ce fichier de description aura le même nom que celui des données mais sera de type DFV. Par exemple, pour les données précédentes, il sera nommé Elf.dfv. Voici son contenu

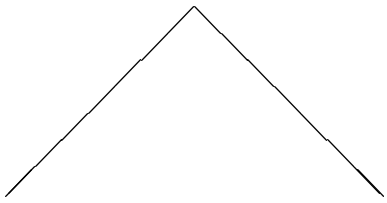
```
#
QT 1 Age   Ans
QL 2 Sexe  1 0 homme 1 femme
QL 3 Etud  4 0 sans  1 sixieme 2 bepc 3 bac 4 sup
```

Un programme en programmation classique viendrait récupérer le nom de l'étude, lire le fichier de description associé, puis le fichier des données pour ensuite calculer les caratéristiques de chaque variable statistique : moyenne, écart-type, coefficient de variation pour une QT (variable quantitative) à unité, comptage et pourcentage pour une QL (variable qualitative).

Selon le gout du programmeur, la lecture des deux fichiers serait effectuée par la même procédure (avec des tests pour savoir si on est dans le .dat ou dans le .dfv) ou dans deux procédures différentes, ce qui obligerait à dupliquer le code.

En P.O., on définit la classe Fichier qui correspondra aux fichiers-textes généraux. Cette classe contiendra deux sous-classes FichierDecritVariables et FichierDeDonnees.

Les champs généraux sont le nom du fichier (nomfich), son nombre de lignes (nblig). Les méthodes de base sont Associer, Ouvrir, Fermer, Lire, Ecrire. On raffine lire dans les sous-classes FichierDecritVariables et FichierDeDonnees de façon à utiliser respectivement la méthode LitLigneDeVariable et la méthode LitLigneDeDonnees . Le graphe d'héritage et d'appartenance propre peut donc être représenté par :



Fichier

Associer / Ouvrir / Fermer
Lire / Ecrire
nomfich
nblig

FichierDecritVariables

Lire
LitLigneDeVariable

FichierDeDonnees

Lire
LitLigneDeDonnees

Le programme d'analyse statistique utilise trois fichiers : celui de description Fv, celui des données Fd , celui des résultats Fr. Fv est un objet de la (sous)classe FichierDecritVariables, Fd est un objet de la (sous)classe FichierDeDonnees et Fr est un objet de la classe Fichier. L'analyse statistique utilisera deux tableaux, tabQT dont les éléments sont des objets de la (sous)classe VariableQT et tabQL dont les éléments sont des objets de la (sous)classe VariableQL.

Le traitement du fichier de description se fait par l'envoi des messages

Fv.associer('elf.dfv') et Fv.lire

celui du fichier de données par

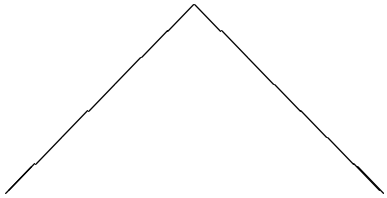
Fd.associer('elf.dat') et Fd.lire.

Pour représenter les variables statistiques, nous définissons la classe générale nommée VariableSTAT avec les champs nomVar, typVar, taille et les méthodes Init, Analyse. Elle admet les deux sous-classes VariableQT et VariableQL.

La classe VariableQT ajoute les champs valeursQT, unite ainsi que les méthodes AjouteValeurQT, Somme, SommeCarre et on y redéfinit les méthodes Init et Analyse.

La classe VariableQL ajoute les champs maxMod, codes, labels, effectifs, frequences, valeursQL ainsi que les méthodes AjouteValeurQL Compte, et on y redéfinit les méthodes Init et Analyse.

Le graphe d'héritage correspondant est



VariableSTAT

Init
Analyse
nomVar
typVar
taille

VariableQT

valeursQT
Init
Analyse
AjouteValeurQL
Somme
SommeCarre
unité

VariableQL

valeursQL
Init
Analyse
AjouteValeurQL
Compte
codes
effectifs
fréquences
maxMod

Pour notre exemple, la première variable est de type QT. Elle se nomme age et ses unités sont en années. La lecture de la ligne 1 du fichier Fv doit donc envoyer le message

```
tabQT[1].Init(' AGE ',1,' an(s) ').
```

La deuxième variable étant de type QL, nommé SEXE avec comme code maximum 1, la lecture de la ligne 2 du fichier Fv doit donc envoyer le message

tabQL[1].Init(' SEXE ',2,1).

De même, la lecture de la ligne 3 du fichier Fv doit donc envoyer le message

tabQL[2].Init(' Etudes ',3,4).

Après avoir envoyé les messages

Fv.associer('elf.dfv') et Fv.ouvrir(Ecriture)

pour écrire une ligne dans le fichier de résultats, il suffit de mettre la chaine de caractère dans une variable quelconque, disons LigTxt et d'envoyer le message

Fr.Ecrire(LigTxt)

et bien sûr, en fin de programme, on ferme tous les fichiers grâce aux trois messages

Fv.fermer, Fd.fermer et Fr.fermer.

Montrons comment réaliser ces programmes en *Turbo Pascal Objet* et en *Visual Dbase*. La déclaration des classes VariableStat et Fichier se fait en *Turbo Pascal Objet* à l'aide du type *Object* et on inclut dès le départ (comme pour les unités) les en-têtes des sous-programmes qui seront explicitement définis. Le corps de ces sous-programmes peut alors être mis plus loin, au gré du programmeur.

Par contre, en *Visual Dbase*, le texte des sous-programmes est mis directement "en ligne" dans la définition des objets. Dans les deux cas, chaque classe d'objets doit être déclarée/définie avant qu'on ne puisse en utiliser des instances (c'est à dire avant de pouvoir créer des objets du type considéré).

Le texte *Pascal* de définition des objets est ici

```
TYPE

 (*******************************************************)
 (**                                                   *)
 (**                     Classe LIGNE                  *)
 (**                                                   *)
 (*******************************************************)

 Ligne   = Object
  texte    : string ;
  function  longueur : integer ;
  function  nbmots : integer ;
  function  mot(numMot:integer ) : string ;
 End ; (* fin de déclaration de Ligne *)


 (*******************************************************)
 (**                                                   *)
 (**                     Classe FICHIER                *)
 (**                                                   *)
 (*******************************************************)

 Fichier =  Object

  nomfich : string ;
  fid     : text   ;
  nature  : string ;
  nblig   : integer ;

  procedure associer( nom : string) ;
  procedure lire ;
  procedure ouvrir ;
  procedure ouvrir_en_Eriture ;
  procedure fermer ;
  procedure litLigne(var ligL : Ligne) ;
  procedure ecrire(ligT : string ) ;

 End ; (* fin de déclaration de la classe Fichier *)
```

```
FichierDeDonnees = Object( Fichier )
 procedure lire ;
 procedure litLigneDeDonnees(var ligL : Ligne) ;
End ;

FichierDecritVariables = Object( Fichier )
 procedure lire ;
 procedure LitLigneDeVariable(var ligL : Ligne ) ;
End ;

(*********************************************************)
(**                                                     *)
(**                    Classe VARIABLE STAT             *)
(**                                                     *)
(*********************************************************)

VariableStat = Object

 numero  : integer ;
 nomVar  : string  ;
 nature  : integer ; (* 1 pour QT, 2 pour QL *)
 nomtype : string  ;
 taille  : integer ;

 procedure init ;
 procedure analyse ;

End ; (* fin de déclaration de VariableStat *)

VariableQT = Object(VariableStat)

 valeursQT : array[1..MaxTaille] of real ;
 unite     : string  ;

 procedure ajoutevaleurQT(nouvelleVal:real) ;
 procedure Init(nomQt:string ; numQT:integer  ; uniQT : string ) ;
 function somme      : real ;
 function sommecarre : real ;
 procedure analyse ;

End ; (* fin de déclaration de VariableQT *)
```

```
 VariableQL = Object(VariableStat)

  maxMod     : integer ;
  codes      : array[0..maxModalite]  of  integer    ;
  labels     : array[0..maxModalite]  of  string  ;
  effectifs  : array[0..maxModalite]  of  integer ;
  frequences : array[0..maxModalite]  of  real    ;
  valeursQL  : array[1..MaxTaille] of real ;

  procedure ajoutevaleurQL(nouvelleVal:real) ;
  procedure init(nomQL:string ; numQL:integer ;  nbmodQL : integer ) ;
  procedure compte ;
  procedure analyse ;

 End ; (* fin de déclaration de VariableQL *)

 (*********************************************)
 (**                                         *)
 (**        Variables de l'application       *)
 (**                                         *)
 (*********************************************)

Var   nbQT : integer ; { nombre de variables QuanTitatives }
      nbQL : integer ; { nombre de variables QuaLTitatives }

      tabQT : array[1..maxVarQT] of variableQT ;
      tabQL : array[1..maxVarQL] of variableQL ;
      typV  : array[1..maxVar]   of integer    ; { >0 pour QT, <0 pour QL }

      Na   : string ; (* nom de l'étude *)

      Fv   : FichierDecritVariables ;
      Fd   : FichierDeDonnees ;
      Fr   : Fichier          ;
```

Là encore, en Visual Dbase, le texte des sous-programmes est mis directement "en ligne" dans la définition des objets, soit

```
*****************************************************)
**                                                   *)
**                    Classe VARIABLESTAT            *)
**                                                   *)
*****************************************************)

CLASS VariableSTAT OF OBJECT

 Protect nomvar, typvar

 Procedure init(nom,typ)
   this.nomvar = nom
   this.typvar = typ
 * fin procedure init

 Procedure analyse
   ? " variable ", this.nomvar
 * fin procedure analyse

ENDCLASS
* fin de classe VariableSTAT

*****************************************************)
**                                                   *)
**                sousClasse VariableQT              *)
**                                                   *)
*****************************************************)


CLASS VariableQT OF VARIABLESTAT

 Protect  unite

 Procedure init(nom,typ,unit)
   super::init(nom,typ)
   this.unite  = unit
 * procedure init
```

```
Function somme
    store upper(this.nomvar) to ndv
    sum &ndv to s
    return  s
* fin de Function somme

Function sommeCarre
    store upper(this.nomvar) to ndv
    sum &ndv*&ndv to s
    return  s
* fin de Function sommeCarre

Procedure analyse

   ? " Analyse de la Variable ", this.nomvar," de type ",this.typvar

   * cherchons si la variable existe

   store 1 to i
   store fldcount() to nbvar
   store -1 to idv
   do while i <= nbvar
      if field(i)=upper(this.nomvar)
         store i to idv
      endif
      store i+1 to i
   enddo
   if  idv < 0
       ? " variable ",this.nomvar," non trouvée "
   endif

   * si taille = 0, pas d'analyse
   * comme pour idv < 0

   count to taille
   if taille = 0
        ? ' problème avec la variable : taille = 0 '
        store -1 to idv
   endif

   * arrivé ici, si idv>0, on peut faire les calculs
```

```
    if idv > 0

         store this.somme()      to vs
         store vs/taille    to vm
         store this.sommecarre() to vs
         store (vs/taille) - vm*vm to vv

         if vs > 0
              store sqrt(vv) to ve
         else
              store -1 to ve
         endif

         store 100.0*(ve/vm)  to vc

         ? '   taille      ',str(taille,7),' individu(s) '
         ? '   moyenne     ',str(vm,7,2)  ,this.unite

         if ve >= 0
            ? '   écart-type  ',str(ve,7,2)  ,this.unite
            ? '   cdv         ',str(vc,7,2)  ,' % '
         else
            ? '   écart-type  ??? '
            ? '   cdv         ??? '
         endif

    endif

 * fin de procedure analyse

ENDCLASS
* fin de classe VariableQL

++++++++++++++++++++++++++++++++++++++++
+                                      +
+  fin de définition des classes       +
+  pour Visual Dbase                   +
+                                      +
++++++++++++++++++++++++++++++++++++++++
```

L'utilisation des objets dans les programmes ressemble à celle des variables classiques. Ainsi le programme *Pascal* qui effectue l'analyse statistique ressemble à

```
PROGRAM ASDE ;

 uses winCrt,strings,wintypes,winprocs ;

 (*$i asde.inc *)

 BEGIN

  Entete ;

  if paramcount = 0 then aideSyntaxe else begin

    writeln(' 1. Dfinition des Variables ') ;  gereVar ;

    writeln(' 2. Lecture des Donnes ')      ;  gereDat ;

    writeln(' 3. Analyse des Variables ')    ;  AnalyseStat ;

  end ; { fin de si paramcount = 0 }

  FinAsde ;

END.
```

La procédure *analyseStat* par exemple est définie par

```
procedure AnalyseStat ;

  var iv  : integer ; (* indice de variable *)
      Nfr : string  ; (* nom du fichier des variables *)

begin (* procedure AnalyseStat  *)

 Nfr := na+'.ras' ; Fr.associer(Nfr) ; Fr.ouvrir_en_Eriture ;
 Fr.Ecrire(' asde (gh) 1999 : ',
     'Analyse Statistique Descriptive  Elémentaire ') ;
```

```
 Fr.Ecrire('') ;
 Fr.Ecrire(' -- Etude du dossier '+Na) ;
 Fr.Ecrire('') ;

 if nbQT>0 then for iv := 1 to nbQT do
    if iv <= maxVarQT then tabQT[iv].Analyse ;
 if nbQL>0 then for iv := 1 to nbQL do
        if iv <= maxVarQL then tabQL[iv].Analyse ;

 Fr.Ecrire('') ;
 Fr.Ecrire(' -- Fin de l''étude pour le dossier '+Na) ;
 Fr.Ecrire('') ;
 Fr.Ecrire(' Copyright (gH) 1999 :') ;
 Fr.Ecrire('  gilles.hunault@univ-angers.fr  ') ;
 Fr.Ecrire('  http://www.info.univ-angers.fr/pub/gh/index.html ') ;
 Fr.Ecrire('') ;

 writeln('') ;
 writeln(' Résultats dans le fichier ',Nfr) ;
 writeln('') ;

 Fr.Fermer ;
```

Les méthodes utilisées ont été définies dans des fichiers inclus. Les méthodes nommées Analyse, par exemple, sont les suivantes

```
procedure VariableStat.Analyse  ;

 var pr : string ; (* phrase de résultats *)
     sr : string ; (* résultat de conversion *)

begin (* procedure Analyse   *)

 Fr.Ecrire('') ;
 pr := ' Etude de la variable numéro ' ;
 str(numero:3,sr) ; pr := pr + sr + '   nommée ' + nomVar ;
 Fr.Ecrire(pr) ;
 writeln('      variable numéro ',numero,' nommée ',nomVar) ;

end ; (* procedure Analyse   *)
```

```
procedure VariableQT.Analyse ;

   var vs : real ;
       vm : real ;
       ve : real ;
       vv : real ;
       vc : real ;
       pr : string ; (* phrase de résultat *)

begin (* procedure VariableQT.Analyse  *)

  VariableStat.Analyse ;

  if taille = 0 then begin
     pr := ' problème avec la variable : taille = 0 ' ; Fr.Ecrire(pr) ;
  end else begin
      vs := somme ; vm := vs / taille ;
      vs := sommecarre ;
      vv := (vs/taille) - (vm*1.0)*vm ;
      if vv >= 0 then ve := sqrt(vv) else ve := -1 ;
      vc := 100.0*(ve/vm) ;
      str(taille:7,pr) ;
      pr := '   taille      ' + pr + ' individu(s) ' ;
      Fr.Ecrire(pr) ;
      str(vm:7:2,pr)   ;
      pr := '   moyenne     ' + pr + ' ' + unite ;
      Fr.Ecrire(pr) ;

      if ve>0 then begin
         str(ve:7:2,pr); pr := 'écart-type  ' + pr + ' ' + unite ;
         str(vc:7:2,pr); pr := 'cdv         ' + pr + ' % '       ;
       end else begin
         str(ve:7:2,pr); pr := 'écart-type  ??? ' ;
         str(vc:7:2,pr); pr := 'cdv         ??? ' ;
       end ;
  end ;

end ; (* procedure VariableQT.Analyse  *)
```

```
procedure VariableQL.Analyse ;

 var j : integer ; { numéro de modalité }
     pr : string ; (* phrase de résultat *)
     sj : string ; (* chaine pour j      *)

begin (* procedure VariableQL.Analyse  *)

  VariableStat.Analyse ;

  if taille = 0 then begin

     pr := ' problème avec la variable : taille = 0 ' ; Fr.Ecrire(pr) ;

  end else begin

       compte ;

       str(taille:7,pr) ;
       pr := '      taille     ' + pr + ' individu(s) ' ;
       Fr.Ecrire(pr) ;

       pr := '' ; { labels }
       for j := 0 to maxMod do begin
           while length(labels[j]) < 12 do labels[j] := ' ' + labels[j] ;
           pr := pr + labels[j] ;
       end ; (* fin pour j *)
       Fr.Ecrire(pr) ;

       pr := '' ; { numéros/codes }
       for j := 0 to maxMod do begin
           str(j:12,sj) ; pr := pr + sj ;
       end ; (* fin pour j *)
       Fr.Ecrire(pr) ;

       pr := '' ; { effectifs }
       for j := 0 to maxMod do begin
           str(effectifs[j]:12,sj) ; pr := pr + sj ;
       end ; (* fin pour j *)
       Fr.Ecrire(pr) ;
```

```
        pr := '' ; { pourcentages }
        for j := 0 to maxMod do begin
            str(frequences[j]:12:2,sj) ; pr := pr + sj ;
        end ; (* fin pour j *)
        Fr.Ecrire(pr) ;

  end ;

end ; (* procedure VariableQL.Analyse  *)
```

Ces quelques extraits de programmes doivent rassurer le lecteur : les programmes en P.O. ressemblent aux programmes traditionnels. On y trouve seulement des écritures à la fois plus compliquées et plus simples. Plus compliquées parce qu'au lieu de paramètres multiples, comme

```
XXX(a,b,c,d,e,f)
```

on trouve des paramètres et des objets, comme

```
UU[a,b].VVV(c,WW[d,e],f)
```

Mais le gain n'est pas là : il est dans le temps qu'on met à retrouver les procédures, à réutiliser les objets sans avoir soit recopier des anciennes lignes de programme, soit à essayer de trouver comment cela fonctionne.

A cet égard, la méthode de lecture des données pour l'analyse statistique illustre bien ce qui se passe : la chaine de caractères ligL est lue et chacune de ses valeurs vr est ajoutée au tableau des valeurs de la variable correspondante.

On trouvera ce texte sur la page suivante. Nous viendrons ensuite commenter les avantages d'une telle écriture.

Dans un programme classique, pour ajouter la valeur `vr` a la variable `nov` en tant que valeur numero `nbval`, on se contenterait d'écrire

```
tabV[ nov, nbVal ] <-- vr
```

mais en objets, on laisse ce soin à la procédure ajouteValeurQT ou à la procédure ajouteValeurQL, ce qui donne les textes

```
  tabQT[nov].ajouteValeurQT(vr)
...
  tabQL[nov].ajouteValeurQL(vr)
```

Si demain on doit écrire un autre programme, et si on ne doit utiliser que deux ou trois variables statistiques, alors, une fois `AGE` définie comme `variableQT`, le texte

```
AGE.ajouteValeurQT(donnee)
```

s'appliquera directement, sans aucune modification alors que précédemment, `tabV[ numVar, nbVal ] <-- Vr` ne s'appliquait qu'à un tableau...

A force de séparer les "choses", que ce soit les objets ou les méthodes, les programmes finissent par se lire comme des dialogues, des communications, des transmissions de messages entre objets.

Nous allons y revenir avec la section suivante ou les messages seront directement visibles sous formes de couleurs dans diverses fenêtres sur l'écran de l'application.

4. P.O, P.E. et Interfaces

Il est deux autres domaines de la programmation profesionnelle où la P.O. se justifie : ce sont la programmation par évènements et la programmation d'interfaces, modes obligatoires pour les logiciels de ce XXIeme siècle. Le déroulement séquentiel classique lecture de fichiers puis calculs puis impression etc. ne correspond à l'utilisation d'un logiciel avec des menus, des fenêtres. Dans celui-ci de nombreux actions sont possibles sans qu'il n'y ait de hiérarchier forte (il a toujours une hiérarchie faible, ne serait-ce parce qu'il faut sélectionner un fichier avant de l'imprimer). La programmation par événènments est prévue pour ce genre de situation : les évènements sont les clicks de souris, la frappe au clavier ou encore le changement de fichiers, de variables... La programmation d'interface avec les boutons, formulaires, listes, panneaux... rejoint la programmation par événènments en lui donnant des objets graphiques pour déclencher les évènements, en plus d'assurer un meilleur confort à l'utilisateur.

Pour ces deux modes de programmation, la P.O. est obligatoire : les logiciels doivent utiliser les mêmes standards de fenêtre (comme la C.U.A avec Fichier comme première rubrique de menu et Quitter comme dernière option de cette rubrique), les fenêtres doivent se comporter de la même façon etc. La volonté de réutiliser le code est ici une obligation de façon à garantir une cohérence entre les différentes fenêtres de l'application. On ne s'étonnera donc pas si le développement de la P.O. coincide avec les développements sous *Windows*...

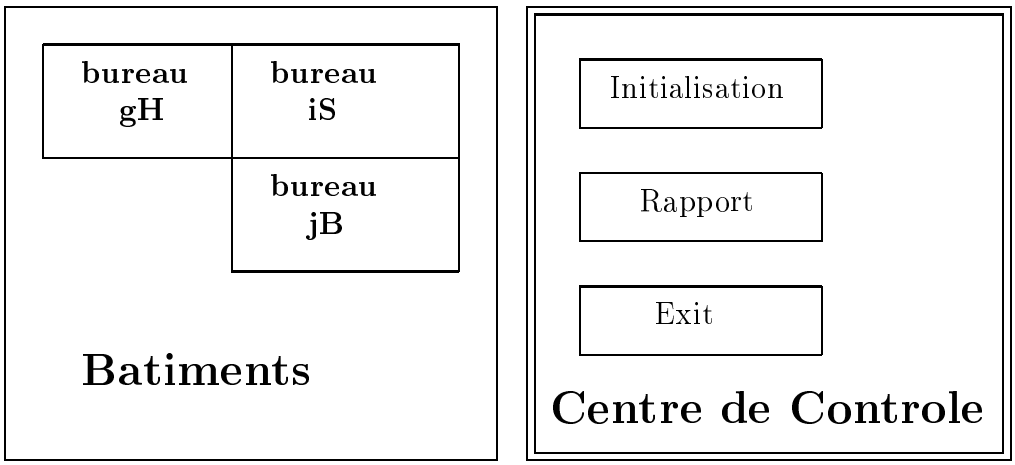
Les fenêtres, les boutons sont des objets prédéfinis, définis par de nombreux champs et de de nombreuses méthodes que le programmeur a juste à utiliser. Prenons le cas de Visual Dbase. Ce logiciel, suite "logique" pour Windows du mythique Dbase sert au départ à gérer des bases de données. Il disposait au départ d'un langage de programmation adapté aux bases de données. Ainsi `sum Prix for article=127 to totVentes` et `store totVentes/VentesPrec to RatioVentes` étaient des actions classiques.

Devenant langage à objets, *Dbase* s'enrichit de classes. Il suffit de citer les classes

```
array          assocArray       browse
checkBox       comboBox         ddeLink
ddeTopic       editor           entryField
form           image            line
listBox        menu             menuBar
ole            oleAutoClient    paintBox
popUp          pushButton       radioButton
rectangle      scrollBar        shape
spinBox        tabBox           text
```

pour comprendre que *Dbase* devient alors un langage à menus, à fenêtres, capable d'affichages graphiques, bref tout ce qui lui manquait pour être non plus seulement un logiciel de bases de données mais un outil logiciel de développement d'applications industrielles.

Montrons par exemple comment écrire un petit tableau de bord pour réaliser un tableau de présence. Admettons que nous ayons n personnes qui occupent des bureaux dans un batiment et qu'on veuille afficher si ces personnes sont là, si elles téléphonent etc. Le logiciel devrait afficher deux fenêtres, une pour représenter le batiment et les bureaux, l'autre étant un "centre de controle" qui permet d'initialiser, d'imprimer un rapport, de quitter... Nous donnons ci-dessous un aperçu de telles fenêtres.



La définition de la fenêtre qui représente le batiment et les bureaux se fait avec les deux seules lignes

```
batiment = new form()
batiment.open()
```

En mode interactif (ce qui est sans doute la façon la plus rapide de se familiariser avec les objets), Visual Dbase afiche dès la frappe de ces deux lignes, une fenêtre vide, nommée Formulaire pour la version française, Form pour la version anglaise.

Les dimensions de la fenêtre sont gérés par les champs width et height. Ainsi `batiment.width = 300` et `batiment.height = 200` suffisent à faire de batiment une fenêtre de `300x200` pixels. Mieux, l'inspecteur d'objets, accessible par `inspect(batiment)` et le concepteur de formes accessible par `modify form...` permettent à l'utilisateur de gérer en direct, en temps réel et en interactif tous les champs et les méthodes de la fenêtre. Ainsi l'inspecteur affiche trois onglets nommé Propriétés, Evénements et Méhodes car Visual Dbase distingue les méthodes liées aux champs et les méthodes évènementielles. Le terme propriétés correspond à ce que nous nommons champs. Les propriétés d'une fenêtre (Formulaire pour la terminologie Dbase) sont affichées dans un ordre alphabétique, sous la forme de départ suivante

```
+Accs
.ActiveControl
+Aide
+Aspect
.DesignView
+Fentre
.First
+Identification
.MenuFile
.NextObj
.PopupMenu
+Position
.RefreshAlsways
.Text
.View
```

Les points indiquent les champs, les plus sont des bascules pour faire apparaitre les champs de la catégorie considérée (au clavier, la frappe de la touche + ouvre la catégorie). Une fois toutes les catégories ouvertes, on a une vue globale de tous les champs de la fenêtre considérée. Nous avons laissé les valeurs par défaut fournies par *Dbase*. Elles sont modifiables sauf les valeurs entre parenthèses, imposées par *Dbase*.

```
-Accs
 .Enabled              .T.
 .Visible              .T.
.ActiveControl
-Aide
 .HelpFile
 .HelpId
 .ShowSpeedTip         .T.
 .StatusMessage
-Aspect
 .ColorNormal          BtnFace
 .Icon
 .MousePointer         0 - Défaut
 .PageNo               1
 .ScrollBar            0 - Inactif
.DesignView
-Fentre
 .AutoCenter           .F.
 .AutoSize             .F.
 .EscExit              .T.
 .Maximize             .T.
 .Mdi                  .T.
 .Minimize             .T.
 .Moveable             .T.
 .ScaleFontBold        .T.
 .ScaleFontName        Ms Sans Serif
 .ScaleFontSize        8,00
 .Sizeable             .T.
 .SysMenu              .T.
 .TopMost              .F.
 .WindowState          0 - Normal
.First                 .F.
```

```
-Identification
 .ClassName          (FORM)
 .hWnd               (558)
.MenuFile
.NextObj             .F.
.PopupMenu           .F.
-Position
 .Height             18,82
 .Left               45,33
 .Top                00,00
 .Width              80,00
.RefreshAlsways      .T.
.Text
.View                Formulaire
```

Un novice en P.O. s'affolera à une telle liste de champs, un programmeur classique s'en trouvera rassuré : il saura grâce à cette liste ce qu'il peut modifier. Ainsi le champ `ScaleFontName` contient le nom de la police de caractères. Si on écrit `batiment.ScaleFontName="Arial"` alors, le texte contenu dans la fenêtre `batiment` utilisera aussitot cette police (l'affichage sera aussitot instantanément modifié car le champ `RefreshAlways` contient `.T.`.

Passons maintenant aux méthodes évènementielles. *Dbase* nous fournit la liste

```
CanClose                CanNavigate
OnAppend                OnChange
OnClose                 OnDesignOpen
OnGotFocus              OnHelp
OnLeftDblClick          OnLeftMouseDown
OnLeftMouseUp           OnLostFocus
OnMiddleDblClick        OnMiddleMouseDown
OnMiddleMouseUp         OnMouseMove
OnNavigate              OnOpen
OnRightDblClick         OnRightMouseDown
OnRightMouseUp          OnSelection
OnSize
```

Ces noms parlent preque tous d'eux-mêmes (pour un angliciste). Il est donc assez facile d'imaginer où on peut mettre du code pour l'initialisation, pour la fermeture, pour l'aide...

Les méthodes non liées (en tous cas vue par *Dbase* comme non directement liées) aux évènements sont

```
AbandonRecord      (FORM::ABANDONRECORD   )
BeginAppend        (FORM::BEGINAPPEND     )
Close              (FORM::CLOSE           )
IsRecordChanged    (FORM::ISRECORDCHANGED )
Move               (FORM::MOVE            )
NextCol            (FORM::NEXTCOL         )
NextRow            (FORM::NEXTROW         )
Open               (FORM::OPEN            )
PageCount          (FORM::PAGECOUNT       )
Print              (FORM::PRINT           )
ReadModal          (FORM::READMODAL       )
Refresh            (FORM::REFRESH         )
Release            (FORM::RELEASE         )
SaveRecord         (FORM::SAVERECORD      )
SetFocus           (FORM::SETFOCUS        )
```

et on y reconnait bien la marque d'une gestionnaire d'enregistrements de bases de données ("records" en anglais).

Revenons à notre application de tableau de bord. Notre application se réduit aux lignes

```
public batiment,bureaux,personnes,gh,is,jb,cc

batiment = new form()
batiment.text   = " les bureaux de la Fac ! "
batiment.width  = 102
batiment.height = 008
batiment.left   = 002
batiment.top    = 002
batiment.showspeedtip  = .t.
batiment.open()

bureaux   = new assocarray()
personnes = new assocarray()

set procedure to demobj.cla
```

```
cc = new controle()
cc.text = " Gestion "
cc.width  = 043
cc.height = 014
cc.left   = 122
cc.top    = 001
cc.open()
```

Après avoir défini la fenêtre batiment pour tracer les bureaux, on y trouve la création de deux objets de type Tableaux Associatifs[3] pour gérer les bureaux et les personnes. Ensuite, il ne reste qu'à définir et gérer l'objet cc (Centre de Controle).

La ligne `set procedure to...` est la ligne qui renvoie au fichier de définition des classes et procédures. On y trouve notamment

```
CLASS Bureau of pushbutton(Batiment)

  this.colornormal      = "N+/W"  ; this.width            = 22
  this.height           = 05      ; this.fontsize         = 14
  this.top              = 01      ; this.showspeedtip  = .t.

procedure init(nom,rang)
  this.text = nom  ; this.left = 03 + 25*(rang-1)
* fin procedure init

procedure estLA
  this.colornormal      = "RG+/G"
* fin procedure estLA

procedure nestpasLA
  this.colornormal      = "N+/W"
* fin procedure estLA

Procedure telephone
  this.colornormal      = "RG+/R+"
* fin procedure estLA

ENDCLASS * fin de classe Bureau
```

[3] rappelons que ce sont des tableaux dont les indices sont des chaines de caractères

qui définit les objets graphiques qui représentent les bureaux du batiment. Les méthodes de la classe viennent seulement changer la couleur du cadre pour montrer si la personne est là et disponible, ou là et non disponible (car déjà au téléphone) ou si elle n'est ps là.

La gestion des personnes, quant à elle, se trouve contenue dans les lignes

```
CLASS Personne of object

procedure init(nomP,presP)
  this.nom      = nomP ; this.presence = presP
* fin procedure init

procedure arrive
  this.presence = .t.
  store this.nom to lap
  if bureaux.iskey(lap)
     bureaux[lap].estla()
  endif
* fin procedure arrive

procedure part
  this.presence = .f.
  store this.nom to lap
  if bureaux.iskey(lap)
     bureaux[lap].nestpasla()
  endif
* fin procedure part

procedure telephone
  if this.presence
     store this.nom to lap
     if bureaux.iskey(lap)
        bureaux[lap].telephone()
     endif
  else
     ? " IMPOSSIBLE : "+this.nom+" n'est pas la "
  endif
* fin procedure telephone

ENDCLASS * fin de classe Personne
```

et la définition du centre de controle se réduit à

```
CLASS controle of form

define pushbutton b1 of this ;
property ;
onclick matin,;
text "Initialisation"     ,;
top           1           ,;
left          4           ,;
width         35          ,;
height        3           ,;
colornormal   "B+/RG+"    ,;
fontsize      14

define pushbutton b2 of this ;
property ;
onclick rapport,;
text "Rapport"           ,;
top           6           ,;
left          4           ,;
width         35          ,;
height        3           ,;
colornormal   "B+/RG+"    ,;
fontsize      14

define pushbutton fin of this ;
property ;
onclick fin,;
text "Exit"               ,;
top           11          ,;
left          4           ,;
width         35          ,;
height        3           ,;
colornormal   "B+/RG+"    ,;
fontsize      14

ENDCLASS
```

Il reste enfin à donner deux procédures importantes qui montrent bien l'intérêt des objets, la procédure qui remplit les bureaux et celle qui affiche un rapport sur l'occupation des bureaux.

```
++++++++++++++++++++++++++++++++
+                              +
+   function remplitBureaux    +
+                              +
++++++++++++++++++++++++++++++++


* on remplit les tableaux à partir de la base

use demobj
goto 1
do while .not. eof()

   * d'abord une personne de plus
   store trim(nc) to nomp
   personnes[nomp] = new personne()
   personnes[nomp].init(nomp,.f.)

   * donc un bureau de plus
   store recno() to nump
   bureaux[nomp] = new bureau()
   bureaux[nomp].init(nomp,nump)
   bureaux[nomp].speedtip = nomprof

   skip
enddo

**** pour la démo interactive

gh = personnes["gH"]
jb = personnes["jB"]
is = personnes["iS"]

return 1
*fin de function remplitBureaux
```

```
++++++++++++++++++++++++++++++++
+                              +
+   function rapport           +
+                              +
++++++++++++++++++++++++++++++++

function rapport

 ?
 ? " Présence par bureaux "
 ?

  use demobj
  goto 1
  do while .not. eof()

    store trim(nc) to nomp
    if bureaux.iskey(nomp)
        store  nomp+" ("+nomprof+") " to msg
        if personnes.iskey(nomp)
           if personnes[nomp].presence
              store msg + "est la " to msg
           else
              store msg + " n'est pas la " to msg
           endif
        endif
        ? msg
    endif
    skip
  enddo

return 1
*fin de function rapport
```

Un lecteur attentif pourra se demander à quoi servent les lignes qui suivent le commentaire `**** pour la demo interactive`. En fait, puisque *Dbase* est interactif, il est possible de démarrer le programme, de taper des instructions et de voir la réponse de *Dbase* au fur et à mesure et pour éviter d'écrire `bureau["gH"]` nous avons mis le synonyme `gH`. Ainsi `gH.arrive()` colorie le bureau de "gH" en vert, `jB.telephone()` affiche le message "Impossible, il n'est pa là", ce qui donne une démonstration directe sans avoir à écrire

un programme pour gérer ces actions. C'est l'une des raisons pour laquelle nous conseillons souvent d'utiliser des langages interactifs : la découverte des concepts y est souvent la plus rapide. Signalons pourtant que la définition des classes doit quand même se faire dans un fichier car les instructions de définition ne sont pas exécutées mais mémorisées pour la création d'instances.

5. Smalltalk, ou le "tout Objet"

Nous avons présenté des exemples en *Turbo Pascal Objet* et en *Visual Dbase*. Pour des puristes, ces langages ne sont pas "langages objets" mais des "langages à objets", relevant d'une P.O.O. (Programme Orientée Objets) plutôt que d'une P.O. (Programme Objets) pure et dure. La différence est plus fondamentale qu'il n'y parait. Dans un langage complétement objet, tout est objet, y compris le langage lui-même. Le langage *SmallTalk* est à cet égard le plus remarquable.

On y retrouve les quatre notions fondamentales que sont l'encapsulation, l'héritage, le polymorphisme et l'envoi de messages. *SmallTalk* fait de l'encapsulation "une intégration des différentes parties de l'objet (données et programmes, structure et opérations, champs et méthodes, caractéristiques et opérations) en un même ensemble tout en masquant le détail de l'écriture". Un objet *SmallTalk* devient alors une "boite noire avec qui on ne communique que par message". L'héritage en *SmallTalk* "permet de transférer des propriétés ou des actions d'une classe à une sous-classe spécialisée". Ainsi, la sous-classe Ecrans issue de la classe Périphériques de Sorties hérite de l'action Remise à zéro. Le polymorphisme y est "le regroupement sous le même nom d'actions différentes liées aux objets". Ainsi les sous-objets de la classe Polygones (tels triangles, rectangles etc.) ont la même méthode Périmètre dont la définition est différente suivant l'objet. Associé à l'héritage, le polymorphisme assure une homogénéité quant aux actions, un masquage de la définition. Le polymorphisme dont un exemple simple est la surcharge des opérateurs classiques (comme + pour à la fois l'addition des nombres et la concaténation des chaînes de caractères) permet une plus grande lisibilité et un meilleur niveau d'abstraction.[4]

[4] signalons que cette surcharge des opérateurs a toujours été présente dans de nombreux langages. Ainsi en *Turbo Pascal* + désigne à la fois l'addition de nombres et la concaténation de chaines, *write* et *read* lisent aussi bien des entiers que des réesl etc..

L'envoi de messages se substitue à l'écriture d'instructions et à l'appel de sous programmes. Les instructions traditionnelles

```
Argument_gauche          Operateur   Argument_droite
Appel_du_sousprogramme   X           liste d'arguments
```

deviennent respectivement l'envoi du message "opérateurO argument_droite" à l'objet "argument_gauche" et l'envoi du message "X" à l'objet "A". Bien qu'artificiel parfois (comment penser que dans 2 + 3 les nombres 2 et 3 ne jouent pas un rôle symétrique mais que le message + 3 envoyé à 2 ?) ce mécanisme de transmission de message est pourtant très soupl, très général et donne un seul schéma d'instruction :

```
Objet       Message
```

où le message contient à la fois l'objet receveur et les parametres, souvent en nombre variable. Le tableau suivant illustre les différences de syntaxe en *Pascal* et en *SmallTalk* quant à l'utilisation des fonctions, le passage de paramètres. On notera au passage ques les méthodes sont repérées par le symbole : en fin de mot, que les instructions en *SmallTalk* sont séparées par un point.

Pascal	*Smalltalk*
`u := f(x)      ;`	`u := f: x          .`
`v := g(x,y)    ;`	`v := x  g: y       .`
`w := h(x,y,z)  ;`	`w := x  h: and: z  .`

Mais cela va beaucoup plus loin : puisque tout est objet en *SmallTalk*, toute instruction doit être un envoi de message et les actions élémentaires, comme les structures deviennent des appels de méthodes. D'où la correspondance pour les tableaux

Pascal	*Smalltalk*
`j    := t[i] ;`	`j := t at : i      .`
`t[i] := j    ;`	`(t at: i) put: j   .`

et pour les tests et boucles :

Pascal

```
if a < b then begin
                ...
              end ;
if eof(stream) then
                  ...
               else
                  ...
while i<10 do begin
                ...
              end  ;
for i := 1to 5 do
    begin ... end  ;
```

Smalltalk

```
a<b ifTrue: [
              ...
            ].
stream atEnd ifTrue:
                        ...
                ifFalse:
                        ...
[i<10] whileTrue: [
                     ...
                   ] .
1 to: 5 do:
  [i: |... ] .
```

De même de la programation modulaire, le découpage en fichiers logiques et physiques (par fonctions, procédures,sous programmes, fichiers inclus etc.) représentent un progrès par rapport à la programmation monobloc, la programmation par objets permet un meilleur niveau d'abstraction, de découpage, de relecture et de mise à jour.

L'abstraction provient tout d'abord d'une simplification ou unification : au lieu de manipuler des données et des programmes, on ne traite qu'un seul type d'entités, les objets. Ensuite, il y a un seul moyen de communication : les messages. Enfin, le raffinement progressif dans l'élaboration des solutions se fait par l'écriture de sous-classes ou de sur-classes. La conception d'une application orientée objet repose sur la construction d'une hiérarchie d'objets exprimant la "généalogie" des objets de l'application. On se méfiera, toutefois de la dépendance du langage sous-jacent : en PO, la gestion des objets diffère beaucoup de celle de la POO. Ainsi, un objet pour Turbo Pascal Objet ressemble à une structure de type *record*, ce qui n'est jamais qu'une façon parmi tant d'autres d'implémenter les objets, alors qu'en Smalltalk un objet est une structure élémentaire.

Le schéma de développement en programmation classique passe une phase d'analyse où les actions sont privilégiées. Les données et les structures de données ne sont intégrées qu'en tout dernier. Ainsi, l'algorithme focalise la lecture d'un nombre alors que l'implémentation et la traduction dans un langage typé viendra spécifier s'il s'agit d'un entier, d'un réel etc. même s'il est difficile de justifier à priori le choix (comment choisir entre `real`, `single` et

`double` en Turbo Pascal ?). Ecrire le programme consistera d'abord à tester la validité de la séquence d'instructions puis à intégrer les types de données aux instructions. En PO, on se concentre d'abord sur les classes d'objets, puis sur leurs caractéristiques et enfin sur les actions. Cette programmation permet de mélanger plus intimement algorithme et données, d'examiner les rapports entre code et valeur.

Le vocabulaire traditionnel de la P.O. comprend les termes de Classes, d'Objets, de Champs, de Méthodes, de Messages et d'Instances. Ces différentes notions peuvent être présentées dans la résolution en P.O. du problème de *Comar*. Rappelons que le problème de *Comar* pour une phrase consiste à "noéliser" cette phrase c'est à direr à l'afficher en arbre de Noel en retournant les most sur eux-mêmes. Par exemple, la phrase suivante "La guerre de Troie n'aura pas lieu." est retranscrite selon le Problème de *Comar* comme suit :

```
                    eiorT
               ed        arua'n
   erreug                       sap
al                                   .ueil
```

Une classe est la définition "théorique" de l'objet, son moule, sa "déclaration" au sens d'une énumération de ses possibilités, que ce soit en termes de variables ou en termes de sous-programmes. Ainsi la classe *Phrase* est caractérisée par sa longueur, son texte. Les "opérations" sur *Phrase* sont la lecture, la noelisation... Les champs d'un objet sont les données associées, les méthodes sont les programmes associés. *Longueur* et *texte* sont ainsi des champs de *Phrase*, *Lecture* et *noelisation* des méthodes de *Phrase*. *Lecture* pourra donc être envoyée à un objet de type *Phrase*. Une instance de la classe *Phrase* est la réalisation effective d'un objet de type *Phrase*. La création disons de *maPhrase* comme objet (ou instance ou réalisation) de la classe *Phrase*, la lecture de cette phrase et l'affection à *lonp* de la longueur de *maPhrase* s'écrit en *SmallTalk* (le cadrage choisi, objet sous objet, méthode sous méthode est purement pédagogique et n'est en aucun cas obligatoire)

```
maPhrase  := Phrase    new       .
             maPhrase  lecture:  .
lonp      := maPhrase  longueur: .
```

Le remplissage des champs d'un objet peut être effectué soit par la création de l'instance, que ce directement ou par héritage soit par les méthodes (au travers de l'envoi de messages appropriés) soit par manipulation directe des champs. Ainsi, le remplissage des champs *longueur* et *texte* d'un objet *Phrase* sera certainement réalisé par la méthode *Lecture* : `Phrase new` doit "raisonnablement mettre" le champ *longueur* à zéro, `maPhrase lecture:` doit certainement (re)calculer ce champ, `maPhrase.longueur := 0.` doit le remettre à zéro.

Dans le problème de *Comar*, la donnée de départ est une phrase composée d'un ensemble de Mots qu'il faut Denombrer, Inverser et qui doit ensuite être Retranscrite en arbre de nol avec pour sommet son Milieu. A priori, deux objets émergent immédiatement : l'objet Phrase et l'objet Mot qui héritera des champs et méthodes de Phrase. L'objet Phrase aura notamment pour CHAMPS sa longueur (nombre de caractères), sa cardinalité (nombre de mots) et pour METHODES la lecture, la recherche du milieu, la nolisation (écriture en arbre de noel), l'inversion (retournement). L'objet Mot héritera de longueur, de lecture et de l'inversion, de la recherche du milieu. Si on préferre, on peut ne créer qu'un objet (Chaine) dont Phrase et Mot sont des instances, car finalement, un mot peut être considérée comme une phrase.

On peut aussi généraliser le problème de Comar en demandant d'afficher en arbre de Nol soit une phrase, soit un mot, soit un vecteur (liste de nombres) ou une liste de symboles. Dans tous ces cas, on doit décomposer une collection en éléments et la réécrire en arbre de noel. On commencera donc par définir une sur-classe de Phrase nommée Collection dont les champs sont sa cardinalité (nombre d'éléments) et ses éléments, dont des méthodes sont lecture, noelisation, inversion.

C'est à ce niveau que la P.O. prend tout son sens : le temps de développement passé à rajouter une nouvelle collection est très réduit puisqu'on n'aura pas à rechercher les sources, à réécrire ou à recopier en modifiant légèrement les entrées, les sorties. De plus, et la cohérence (ou seulement la liaison de par le même traitement) entre Phrase, Vecteur, Liste se retrouve grâce à la clase Collection. On pourra dans la foulée noéliser un vecteur, une liste, une phrase puisqu'on aura écrit une méthode "noéliser" qui traite une collection et qui sait donc traiter toutes les sous-classes de collection.

Le graphe d'héritage complet de la hiérarchie des objets doit alors ressembler à

```
Objet Collection          Champ   Cardinalit
+                         Champ   Elments
|                         Mthode Lecture
|                         Mthode Noelisation
|                         Mthode Inversion
+-----+--Objet Phrase     Champ   Cardinalit     (hrit)
|     |                   Champ   Elments        (hrit)
|     |                   Champ   Longueur        (propre)
|     |                   Mthode Lecture         (hrite)
|     |                   Mthode Noelisation     (hrite)
|     |                   Mthode Inversion       (hrite)
|     +-----Objet Mots    Champ   Longueur        (hrit)
|                         Champ   Contexte        (propre)
|                         Mthode Lecture         (hrit)
+-----+--Objet Vecteur    Champ   Cardinalit     (hrit)
|     |                   Champ   Elments        (hrit)
|     |                   Mthode Lecture         (hrite)
|     |                   Mthode Noelisation
|     |                   Mthode Inversion
|     +-----Objet Nombre  Champ   Valeur          (propre)
|                         Champ   Signe           (propre)
|                         Mthode Lecture         (hrite)
+-----+--Objet Liste      Champ   Cardinalit     (hrit)
|     |                   Champ   Elments        (hrit)
|     |                   Mthode Lecture         (hrite)
|     |                   Mthode Noelisation
|     +-----Objet Symbole Champ   Valeur          (propre)
|                         Mthode Lecture         (hrite)
```

Une présentation simplifiée de la hiérarchie des objets correspond à

```
Collection
    Phrase
       Mot...
    Vecteur
       Nombre...
    Liste...
```

Le degré d'abstraction et de raffinement introduit par la P.O. est clair : au lieu de définir de nombreuses structures de données similaires (comme en Pascal) et de réécrire les procédures associées, la P.O. permet de partir d'un type de structure et de ses procédures pour ensuite dériver d'autres structures, que ce soit des sous-structures ou des sur-structures. Concrètement, cela rejoint la technique de "refinement" de certains types de programmation. Au fur et à mesure que l'on programme, on crée ainsi des hiérarchies d'objets.

Intérêts et difficultés de la PO

Ce type de programmation n'est pas pour autant plus facile ou plus naturel que la programmation traditionnelle. Tout d'abord il faut être capable de structurer la hiérarchie, savoir faire dériver, hériter correctement. La qualité du logiciel associé est alors liée à une non-redondance du code, un masquage des procédures les plus internes. Avant d'inventer un objet, il faut donc vérifier qu'il n'a pas déjà été inventé, comme en programmation classique. Avec un langage comme Smalltalk V sur PC, il y a en standard environ 100 objets et 2000 méthodes. S'y retrouver représente un gros travail. Par exemple, il serait incohérent de réinventer le calcul de la moyenne d'une liste de nombres si elle existe déjà. Parcourir l'encyclopédie des classes (partie 4 du manuel de Smalltalk) où utiliser en interactif le Class Hierarchy Browser prend beaucoup plus de temps que d'écrire la méthode nommée moyenne pour l'objet liste de nombre.

Le masquage des procédures donne une programmation plus aisée mais aussi plus astreignante : on travaille comme avec une bibliothèque de sous programmes dont on ne connait que la syntaxe d'appel, les conditions d'utilisations, mais pas le texte source. Cela ne garantit pas pour autant de la "bonne" écriture de ces procédures. Un mauvais programmeur fera un piètre programmes en P.O. Prenons un exemple concret : supposons que nous redéfinissions la conversion en majuscule d'une chaine de caractères par programme, sachant qu'on dispose d'une conversion caractère par caractère seulement. Nous voulons utiliser pour cela une boucle REPETER ... JUSQU'A. Notre idée (fausse) de départ est qu'on doit traiter les caractères 1 à n où n désigne la longueur de la chaine.

Pour une chaine nommée CH l'algorithme itératif associé est

```
N <-- longueur(CH)
{ on parcourt la chaine caractre par caractre
  not K et repr par son indice IC }
IC <-- 1
rpter
   K <-- SousElment(CH,IC)
   ... { conversion de K dans CH }
   IC <-- IC + 1
jusqu' IC = N
```

L'erreur de notre méthode est de ne pas prendre en compte la chaîne vide. Faire de notre algorithme une fonction ou une procédure ne résoudra pas le problème, pas plus que d'en faire une méthode pour les objets associés aux chaines de caractères. En d'autres termes, rien ne sert de le traduire en *Pascal* par

```
N := length(CH) ;
{ on parcourt la chaine caractre par caractre
  not K et repr par son indice IC }
IC := 1 ;
repeat
   K := CH[IC] ;
   ... { conversion de K dans CH }
   IC := IC + 1
until IC = N
```

ou en *SmallTalk* par

```
 N := CH length: .
{ on parcourt la chaine caractre par caractre
  not K et repr par son indice IC }
  IC := 1 .
[...]
 whileTrue:[
   K := CH at: IC.
   ... { conversion de K dans CH }
   IC := IC + 1 .
 ]
```

Une autre difficulté de la PO qui ne la rend pas immédiatement accessible est qu'elle ne prend tout son sens que pour des "gros" programmes. Pour un programme court et jetable (et il y en a beaucoup), la PO n'est pas forcément rentable. Par contre, des environnements complets (tels Smalltalk) présentent un "plus" par rapport à des langages orientés objets : la diminution du nombre de fichiers, par exemple, est flagrante. Smalltalk contient tout en une seule "image". On est loin des fichiers sources, des fichiers inclus, des fichiers de données, des fichiers de code relogeable, des fichiers de code exécutable etc. L'inconvénient immédiat est que tout regrouper en un seul fichier occupe une taille importante, de même que de garder une trace de tout ce qui a été fait, que ce soit en lecture, en modification, en test, etc. Il n'est pas rare d'avoir une image de Smalltalk et un "log" (copie des interactions via le Transcript) de chacun 1 ou 2 méga octets.

Dans un langage à objets, il est conseillé au programmeur de réutiliser le code précédemment développé. En effet dans un système comme SMALLTALK/V, l'environnement de programmation contient déjà une multitude de classes et de méthodes prédéfinies. C'est une sorte de boite à outils 'logicielle' dans laquelle l'utilisateur puise pour ses propres besoins. Il est même conseillé qu'un programme écrit sous SMALLTALK/V utilise 90% de codes existant. Dans ce tel environnement, on programme par adaptation de l'existant plutôt que par redéfinition complète. Cette manière de développer un logiciel comporte néanmoins un désavantage : toute l' information associé à une classe n'est pas forcément localisée dans celle-ci. Pour comprendre le comportements des instances d'une classe donnée, il faut connaître ses superclasses. Tout ce qui n'est pas dit explicitement dans une classe est implicite dans la hiérarchie de celle-ci. On comprend donc que cette tâche pour un programmeur débutant sous SMALLTALK/V peut représenter un effort important et même décourageant. On comprend donc aussi aisément que ce langage trouve sa pleine justification lors de réalisations très importantes. Réutilisation du code et héritage Dans la réutilisation du code, le principe de base est de rechercher dans la hiérarchie des classes si une méthode ne correspond pas à la tâche voulue. Ainsi dans notre programme, nous avons réutilisé la méthode 'asArrayOfSubstrings' de la classe 'String' pour obtenir directement un tableau de mot par rapport à la phrase traitée. En conséquence pour réutiliser une classe, il faut connaître les messages que les instances de celle-ci envoient aux autres objets, ou en d'autres termes comprendre les interaction existantes entres les différentes classes de l'environnement. Pour l'héritage, il s'agit, lorsque la création d'une nouvelle classe s'impose, de savoir profiter de tout ce que peut faire sa superclasse Comparé à SMALLTALK 80, SMALLTALK/V peut sembler plus aisé dans son utilisation de par son héritage simple.

Pour illustrer la difficulté de trouver "la" méthode, prenons l'exemple du manuel de SmallTalk. Soit à programmer le comptage du nombre de lettres d'un texte (après conversion éventuelle des majuscules en minuscules). Le texte *Pascal* suivant s'en acquitte, nommant f le tableau des fréquences, s le texte de départ, c le caractère courant et k la position de c dans f.

```
var s   : string  ;
    c   : char    ;
    i,k : integer ;
    f   : array[1..26] of integer ;
(* initialisation *)
for i := 1 to 26 do f[i] := 0 ;
(* conversion et comptage *)
for i := 1 to length(s) do begin
    c    := lowcase( s[i] ) ;
    if isLetter(c) then begin
       k    := ord(c) - ord('a') + 1 ;
       f[k] := f[k] + 1
    end ;
end ;
```

Une traduction au pied de la lettre en *SmallTalk* aboutit à

```
| s, c, i,k, f |
" initialisation "
f := Array new: 26.
[1 to: 26] do: [:i | f at: i put: 0].
" conversion et comptage "
1 to: (s size:) do: [:i |
    c    := (s at: i) asLowercase.
    c isLetter
    ifTrue:[
       k    := (c asciiValue) - ($a asciiValue) + 1
       f at: k put: (f at: k) + 1
    ]
].
```

Mais ce serait mal connaitre *SmallTalk*. Dans la classe *Bag*, il y a une méthode *add* : qui fait ce qu'il faut ! Et donc, une solution propre en *SmallTalk* s'écrit avec les les seules lignes

```
| s f |
f := Bag new: . " initialisation "
" conversion et comptage "
s  do: [:c | c isLetter ifTrue: [ f add: c  asLowercase ] ].
```

encore fallait-il savoir où chercher ! Car *Bag* n'est pas directement avec *String* ou *Array*. La construction `Collection do:` est typique de la programmation objet : toute structure contient des éléments (un tableau contient des valeurs, une phrase contient des caractères etc.) et le `do:` considéré vient parcourir la structure. Mieux : la variable de parcours (*c* dans l'exemple, déclarée par : *c* est forcément locale au parcours : c'est vraiment propre !

Pour terminer, montrons comment implémenter nos méthodes pour le problème de *Comar*. Les classes de *SmallTalk* sont suffisantes pour résoudre notre problème et nous ne créons pas de nouvelles classes, utilisant la classe *String* pour les phrases. Commençons par les "petites méthodes faciles"

```
mot: unEntier
   " renvoie le mot numero unEntier"
   | reponse taille |
   reponse  := String new.
   taille   := self nbmots.
   ( unEntier = 0 or: [ unEntier > taille ] )
       ifTrue:  [reponse := ''. ]
       ifFalse: [reponse := self asArrayOfSubstrings at: unEntier.].
    ^reponse

motmilieu
   " renvoie le mot du milieu"
   | reponse indice |
   reponse := String new.
   indice := self nbmots // 2 +1.
   reponse := self mot: indice.
   ^reponse
```

```
nbmots
   " compte le nombre de mots "
   | reponse  asTableau|
   asTableau := Array new.
   asTableau := self asArrayOfSubstrings.
   reponse := asTableau size.
   ^reponse
```

Puis essayons de faire l'interface, la saisie de la phrase et l'appel de la noelisation :

```
noelgraph
   | phrase motmilieu indicemotmilieu motgauche
     motdroite pointdroit pointgauche ordonnee origine |
   phrase := String new.
   Display fill: rectangle
        rule: Form over
        mask: (Display compatibleMask color: 14).
   phrase := Prompter
             prompt:  ' Entrer votre phrase'
             default: ' Ceci est un exemple instructif.'.
   indicemotmilieu := phrase nbmots // 2 + 1.
   motmilieu := phrase motmilieu.
   origine := 300 - ( motmilieu size * 8 ).
   pointgauche := origine + (motmilieu size * 8).
   pointdroit := origine.
   ordonnee:= 50.
   motmilieu displayAt: origine @ ordonnee.
    1 to: indicemotmilieu - 1 do:
          [ :imc|
            motdroite := phrase mot: indicemotmilieu - imc.
            motgauche := phrase mot: indicemotmilieu + imc.
            pointdroit := pointdroit - (motdroite size * 8).
            ordonnee:= ordonnee + 14.
            ( motdroite reversed ) displayAt: pointdroit @ ordonnee.
            ( motgauche reversed ) displayAt: pointgauche @ ordonnee.
            pointgauche := pointgauche + (motgauche size * 8).
          ].
  Menu message: 'continue'.
```

Et voici maintenant l'affichage en sapin :

```
noellit
    | phrase motmilieu espaceadroite espaceentremot
      motgauche motdroite indicemotmilieu fenetre |
    phrase := String new.
    Display fill: rectangle
        rule: Form over
        mask: (Display compatibleMask color: 14).
    phrase := Prompter
        prompt: '              Entrer votre phrase            '
        default: ' Ceci est un exemple instructif.'.
    fenetre := TextEditor
        windowLabeled: 'Voici votre phrase selon Comar'
        frame: ( 45 @ 70 extent: 550 @ 400 ).
    indicemotmilieu := phrase nbmots // 2 + 1.
    motmilieu := phrase mot: indicemotmilieu .
    espaceadroite := 30.
    espaceentremot := motmilieu size.
    fenetre nextPutAll: ( ' ' chaine: espaceadroite ),motmilieu reversed;
                        cr.
    1 to: indicemotmilieu - 1 do:
          [ :index|
            motdroite := phrase mot: indicemotmilieu - index.
            motgauche := phrase mot: indicemotmilieu + index.
            espaceadroite := espaceadroite - ( motdroite size).
            fenetre nextPutAll:
                 ( ' ' chaine: espaceadroite ),
                 motdroite reversed,
                 ( ' ' chaine: espaceentremot ),
                 motgauche reversed;
                 cr.
             espaceentremot := espaceentremot + motdroite size
                           + motgauche size.
          ].
   Menu message: 'Continue'.
   fenetre closeWindow.
```

Nous laissons le soin au lecteur et à la lectrice de lire attentivement les lignes de code *SmallTalk*. On y appréciera la valeur par défaut (avec la méthode `default:`) dans la lecture d'une variable au clavier (grâce au Prompter), l'affichage dans une fenêtre dont le titre est le paramètre passé par la méthode `windowLabeled:` et toutes ces bonnes choses "évidentes" qui sont standard en P.O..

6. En guise de conclusion

Pour en finir avec le style objets, nous retiendrons les points suivants

- un logiciel qui dialogue avec l'utilisateur via une interface graphique utilise naturellement des objets comme les fenêtres
- un logiciel conséquent, qui doit être efficace et fiable doit réutiliser du code testé, éviter de dupliquer du code d'où la notion de hiérarchie de classe et d 'héritage
- les objets structurent les données et les actions, d'où une plus grande concision et une plus grande clarté
- les mêmes actions ou les actions similaires doivent porter le même nom d'où polymorphisme et surchage des opérateurs
- la cohérence du tout doit primer sur la visualisation de perfection du détail d'où l'encapsulation
- le mode de communication, la programmation repose alors sur un seul mode de fonctionnement : l'envoi de messages.

BIBLIOGRAPHIE

G. Booch
Analyse & conception orientées objets
2ème édition, Addison-Wesley, 1994.

J. Ferber
Conception et programmation par objets
2ème édition, Hermés, 1991.

G. Masini, A. Napoli, D.Colnet, D. Léonard, K. Tombre
Les langages à objets
InterEditions, 1991.

B. Meyer
Introduction à la théorie des langages de programmation
InterEditions, 1992.

Digitalk
Smalltalk V Tutorial and Programming Handbook
Digitalk Inc., 1992.

Borland
Visual Dbase, Guide de référence
Borland International, 1995.

Borland
Turbo Pascal, Programmation Orientée Objets
Borland International, 1989.

Table des matières