PLUS LOIN AVEC LA PROGRAMMATION OBJET :

OOP+[1]

Sommaire

1 Ce document a été créé avec Star Office.

1– Généralités

Cet article fait suite au compte rendu de l'atelier C6[2] des réunions ATOUTFOX de La Défense en 2004. Dans cet atelier, Richard FLOURIOT et moi-même vous montrions les bases de la programmation objet et la création de classes soit en mode visuel soit par programmation. Ce document complète son chapitre 8. En particulier avec les réflexions menées par la société BULL sur l'avenir de la programmation dans un monde où l'internet est en train de prendre de plus en plus de place et où les PC deviennent d'une puissance phénoménale.

Ces réflexions ont mené à des réalisations qui ont largement dépassé le monde informatique pour atteindre le grand public. C'est le cas, par exemple, du site du gouvernement permettant de changer d'adresse en une seule opération; l'application principale se chargeant de transmettre les informations collectées à d'autres applications disparates (EDF, Poste, impôts, Banque, ...) ceci étant du à l'organisation des applications beaucoup plus qu'à des techniques sofistiquées.

Cette manière de programmer se rapproche beaucoup de l'architecture n-tiers (avec n>=3) que nous a présentée Toni FELTMANN[3] lors des rencontres de Lyon. Architecture largement adoptée dans le monde pour sa facilité d'adaptation aux évolutions technologiques.

Nous verrons aussi comment la technique des 'motifs' ('patterns'[4]) de Steven BLACK peut facilement s'y intégrer.

Mais commençons par reprendre les bases de la création des classes non visuelles (l'ex chapitre 8 du C6) en les complétant puis nous philosopherons un peu ....

2 accessible sur le site d'Atoutfox
3 www.f1technologies.com société éditrice de la 'charpente' Visual FoxExpress.
4 Cette technique permet de modifier (dynamiquement si nécessaire) le fonctionnement d'une clase sans la sous-classer.

2– Définition des classes par programmation

De la même manière que nous savons construire des classes visuelles avec l'aide du générateur de classes, nous pouvons construire des classes dites 'non visuelles' qui reprennent exactement les mêmes principes, qui permettent l'encapsulation, la hiérarchisation, etc. mais malheureusement sans l'aide de l'interface graphique. Il faudra donc, oh horreur !, écrire du code.

En clair, nous allons créer des classes permettant de gérer des structures de données ('structure' au sens du langage C) plus ou moins complexes (de presque rien à presque toute l'application) et le code y afférent. Ces classes vont d'une part entrer en 'concurrence' avec la bufferisation habituelle fournie par VFP et permettre de créer une couche logicielle efficace autour des données, d'autre part permettre de structurer l'application d'une manière tout à fait inhabituelle mais en définitive très efficace.

Isoler les traitements propres aux données dans une couche particulière ('Data Layer' de l'architecture n-tiers) permet d'en espérer les avantages suivants :

1. **élaboration découplée de celle de l'application;**
2. **tests aisés y compris à partir de la fenêtre de commande;**
3. **élaborer des règles de validation et des règles d'intégrité référentielle dans des applications ne faisant pas appel à un conteneur base de donnée;**
4. **modification aisée des structures des données sans avoir à réécrire toute l'application;**
5. **Passage aisé de tables libres à une base de donnée ou d'une BDD VFP vers une BDD externe (SQL Server);**
6. **traitement physiquement isolable dans une DLL par exemple ou même sur une autre machine;**
7. **enfin, application aisée des principes évoqués par Steven Black (un 'décorateur' pouvant ajouter, par exemple, un cryptage/décryptage des données ou une traduction dans une autre langue).**

Préliminaires :

1. **tous les exemples qui suivent ont pour classe de base 'Custom'. Il existe depuis VFP8 une classe encore plus vide : « Empty » (mais j'ai commencé avec VFP6 !); Si vous consultez les exemples de S. Black, ses classes sont issues de 'relation'.**
2. **toutes les classes sont définies dans un .PRG (pas de .scx) a priori inclut dans la liste d'un SET PROCEDURE TO (ne pas mettre SET CLASSLIB !). Mais on peut aussi créer directement des objets utilisables au moment où on en a besoin : on instancie un objet vide (monobjet = CREATEOBJECT('custom')) puis on le complète avec sa méthode addproperty (il y a un exemple plus bas).**
3. **nous allons faire un grand usage des méthodes _assign : rappelons que si une propriété d'un objet s'appelle 'momo', VFP lancera automatiquement l'exécution de la méthode 'momo_assign' chaque fois que l'on tentera de modifier la valeur de momo. De même il existe une méthode access exécutée chaque fois que l'on tente de lire la valeur de la propriété correspondante. La compréhension de ces deux méthodes est importante pour la compréhension de la suite de ce document. Après VFP8, il existe**

même la possibilité d'écrire This_Access qui sera déclenchée à chaque appel de méthode (attention ceci est très différent du fonctionnement normal sur les propriétés) ou de propriété de l'objet concerné.

4. tous les exemples ci_dessous sont fournis dans un fichier classes_non_visuelles.prg à votre disposition. Globalement, nous allons gérer une famille humaine dans une école : les parents et un ou plusieurs enfants. Nous avons 2 tables PARENTS et ENFANTS et 3 vues : 'une_ligne_parents', 'un_enfant' et 'les_enfants_une_famille' dont les noms sont parfaitement explicites.

Commençons par le plus simple :

```
SELECT enfants
SCATTER TO gt_menfants
```

gt_menfants est un tableau qui est l'image de l'enregistrement courant. On accède à chacun des éléments par son indice (? gt_menfants[3]).

```
SCATTER NAME oenfants
```

nous donne ce qu'en langage C on appelle une structure : c'est une seule 'variable' mais chaque champ est accessible par son nom : par exemple : oenfants.prenom ou oenfants.datenai. En VFP oenfants est un objet mais il ne possède aucune méthode; il est donc impossible d'y faire un 'addproperty' pour lui ajouter des propriétés qui ne seraient pas un champ de la table sous-jacente.

Allons plus loin : la famille de code 97040 a 3 enfants : on suppose qu'on les a dans un curseur ENF97040. On peut écrire :

```
SELECT enf97040
DIMENSION tenf[ RECCOUNT()]
GO 1
SCATTER NAME tenf[1]
GO 2
SCATTER NAME tenf[2]
GO 3
SCATTER NAME tenf[3]
```

On peut accéder ainsi aux 3 enfants par tenf[1].prenom ou tenf[2].classe, … On peut écrire du code comme :

```
For lnindice = 1 TO 3
      ? tenf[ m.lnindice].prenom + ' '+ tenf[ m.lnindice].classe
Next
```

qui peut s'écrire (pour une meilleure efficacité et une meilleure lisibilité) :

```
For lnindice = 1 TO 3
   WITH tenf[ m.lnindice]
      ? .prenom + ' '+ .classe   && attention aux points initiaux
   ENDWITH && tenf[ m.lnindice]
Next
```

Toute la suite de ce discours étant basée sur le principe ci-dessus, sa bonne compréhension est indispensable : on est en présence d'un tableau de 3 éléments dont chaque élément est un objet reproduisant la structure du curseur sous-jacent. Plus tard on fera des tableaux de structures ayant des tableaux comme éléments, chaque élément de ces tableaux étant eux mêmes des structures (VFP s'y retrouve bien mieux que nous!) ..... Le tableau tenf[ ] ci-dessus (et ceux bien plus compliqués que l'on va faire) sont facilement transférable comme paramètre soit à un formulaire soit à une fonction :
DO FORM afficher_enfant WITH tenf ce qui va nous permettre de manipuler très facilement de grosses quantités de données.

Allons encore plus loin : on aimerait avoir une 'structure' nous permettant de garder les valeurs originales de l'enregistrement d'un enfant, de travailler sur une image de l'enregistrement et de conserver quelques informations supplémentaires comme par exemple un indicateur de modification. Un tableau ou un 'objet' comme ci-dessus ne suffisent plus, il faut écrire une classe d'où on instanciera autant d'objets que nécessaire. Soit la classe :

```
DEFINE CLASS un_enfant_demo AS CUSTOM && ceci est une démonstration
    code_famille = '00000'
    code_enfant = '00'
    enfant = NULL           && copie de l'enregistrement
    enfant_initial = NULL   && idem sert à détecter les modifications
    * on pourra détecter précisément les modifications en comparant les deux
    * objets 'enfant' (sur lequel on travaille) et 'enfant_initial' qui mémorise
    * l'état initial. Avec enfant_initial, on peut gérer facilement un bouton du style
    * "Reprise des valeurs initiales" ou "abandon des modifications faites".
    erreur = -1         && -1 non initialisé, 0 non trouvé, 1 ok
    modifie = .F.       && .T. si l'élève a été modifié
ENDDEFINE && CLASS un_enfant_demo AS custom
```

On peut instancier 'un enfant_demo' par :

```
loenf = CREATEOBJECT('un_enfant_demo')
```

et écrire :

```
loenf.code_famille = enf97040.codfam
loenf.code_enfant = enf97040.codenf
SCATTER NAME loenf.enfant
SCATTER NAME loenf.enfant_initial
loenf.erreur = 1
```

Petite digression : nous avons définit ci-dessus une classe et nous avons instancié un objet à partir d'elle. Ce même objet peut être créé d'une autre manière qui sera utile dans certains cas lorsque l'on n'aura pas besoin de l'héritage :

```
loenf = CREATEOBJECT('custom')
loenf.addproperty('code_famille') && il y a 2 moyens d'initialiser une propriété
loenf.code_famille = ''
loenf.addproperty('code_enfant', '')
loenf.addproperty('enfant', NULL)
loenf.addproperty('enfant_initial', NULL)
loenf.addproperty('erreur', -1)
loenf.addproperty('modifie', .F.)
loenf.addproperty('demo_tableau[5]) && démo pour la création d'un tableau
```

Rappel: ce principe n'est pas applicable aux objets résultats d'un SCATTER NAME;
Note : un autre avantage de cette manière de faire sera vu plus loin (« addmethod »)

On peut maintenant définir une famille, c'est à dire une classe contenant des informations sur les parents et autant d'objet 'enfants' que nécessaire. Par exemple :

```
DEFINE CLASS une_famille_demo AS CUSTOM
    code_famille = "00000"
    parents = NULL          && copie de l'enregistrement de PARENTS
    parents_initial = NULL && idem sert à détecter les modifications
    nombre_enfants = 0      && d'après vous ?
    DIMENSION enfants[1]   && contiendra l'ensemble des enfants
    erreur = -1             && -1 non initialisé, 0 non trouvé, 1 ok
    modifie = .F.           && .T. si les parents ont été modifiés
ENDDEFINE && CLASS une_famille_demo AS custom
```

et écrire :

```
SELECT parents
LOCATE FOR codfam= '97040'
famille = CREATEOBJECT('une_famille_demo')
WITH famille  && attention les points initiaux ne sont pas très visibles !
    .code_famille = parents.codfam
    SCATTER NAME .parents
    SCATTER NAME .parents_initial
    SELECT enf97040
    .nombre_enfants = RECCOUNT()
    DIMENSION .enfants[ .nombre_enfants]
    SCAN ALL
      lnaccu = RECNO()
      .enfants[ m.lnaccu ] = CREATEOBJECT('un_enfant_demo')
      WITH .enfants[ lnaccu]
          .code_famille = famille.code_famille
          .code_enfant = enf97040.codenf
          SCATTER NAME .enfant
          SCATTER NAME .enfant_initial
          .erreur = 1
      ENDWITH && .enfants[ lnaccu]
    ENDSCAN
ENDWITH  && famille
```

Après avoir instancié la classe (lofamille = CREATEOBJECT('une_famille_demo')), on peut écrire :

- **le nom de famille : lofamille.nomfam**
- **le nombre d'enfants : lofamille.nombre_enfants**
- **le prénom du premier enfant : lofamille.enfants[1]. prenom**
- **ses notes de calcul du 1er trimestre : #DEFINE calcul 3 puis lofamille.enfants[1].notes[calcul].trimestre[1]**
  ....

On a ainsi une seule 'variable' qui contient l'ensemble des données de la famille. Cette variable peut être passée en paramètre à un formulaire, une procédure, une fonction ….. comme on l'a vu un peu plus haut. Cette variable étant complexe, elle doit être passée par référence et alors toute modification de cette variable dans le formulaire ou la procédure sera 'reportée' dans la procédure ou le formulaire appelant (à un bug près concernant

SCATTER NAME[5]). La quantité de données mémorisée sous un seul nom peut être très grande, rassembler les valeurs de plusieurs vues ou tables et surtout réunir des valeurs qui ont toutes un point commun. Contrairement aux vues ou aux tables bufferisées qui peuvent, elles aussi, mettre en mémoire une quantité importante de données mais dans une architecture 'table par table', cette méthode permet de faire la même chose mais sous une architecture logique propre à l'application (dans l'application actuelle, on mémorise des choses comme : toutes les nouvelles familles, tous les élèves en CM2, ...).

Un découpage 'hiérarchique' judicieux (ici on a séparé les enfants des parents)[6] va permettre de créer facilement des classes fonctionnelles différentes à partir des mêmes briques de base, soit par sous-classement (nous sommes en orienté objet) soit en ajoutant ou en remplissant des propriétés au moment de l'exécution avec CREATEOBJECT() soit en créant ce que Steven Black appelle un décorateur (et que nous verrons plus loin dans ce document).

Les exemples de ce document sont tirés d'une application de gestion d'école primaire où l'on doit soit traiter les enfants famille par famille par exemple pour les inscriptions, la facturation, …. soit par classe (au sens classe d'école ! !) avec le nom de leur instituteur pour gérer les notes, la cantine, la garderie, les voyages scolaires; d'autres groupements sont réalisés par exemple les élèves de CM2 pour traiter leur départ vers un collège etc …

Dans la plupart des cas, on n'aura besoin que de certaines informations sur la famille mais dans d'autres, par exemple la facturation, on aura besoin de toutes les données familiales. On a deux solutions pour résoudre ce problème (historiquement j'ai commencé par la première avant de m'apercevoir que la deuxième était tout aussi valable) :

1. **on va créer une classe 'famille' contenant les données courantes et définir une sous-classe 'famille_complète' qui contiendra, en plus, les données moins utilisées et on instancie l'une ou l'autre en fonction des besoins;**
2. **on crée une seule classe qui contient tous les 'pointeurs' vers les données et on ne met à jour que les propriétés utiles. Cette classe contiendra par exemple un tableau d'un seul élément 'factures' qui ne sera dimensionné et mis à jour qu'en cas de besoin.**

on va aussi créer une classe 'classe' contenant les coordonnées du maître puis la liste des élèves de cette classe, une classe 'cm2' contenant les élèves des différents CM2 ....

Mais, on travaille sur des classes, on peut donc créer des méthodes !

c'est à dire mettre dans la classe définissant des données ayant toutes un point commun (ici un enfant, une famille ou une classe d'école) tout le code (en fait toutes les méthodes) concernant ces données (lecture, modification, vérification, enregistrement, calculs, impressions, …). On va pouvoir introduire un 'constructeur' (lancé à la création de l'objet) et un 'destructeur' (lancé à la destruction de l'objet) et des méthodes propres à la classe. Ceci est fondamental bien que découlant tout simplement de la définition de 'classes'. J'insiste sur ce point : *nous allons mettre le code là où il est le plus judicieux, c'est à dire dans la même classe que les données qu'il traite*.

Reprenons notre classe 'enfant' et ajoutons lui un peu de code (note : init et destroy sont des mots réservés à employer obligatoirement) :

5 Avant VFP9, il ne faut pas mettre à jour un élément par SCATTER NAME dans une procédure appelée, cette mise à jour ne sera pas reportée dans le programme appelant. Dans VFP9, il faut ajouter la clause ADDITIVEpour que le programme appelant puisse voir la mise à jour.

6 'dans le temps', on avait la programmation modulaire, on va faire la même chose pour les données.

```
DEFINE CLASS un_enfant_demo2 AS CUSTOM
    code_famille = "00000"
    code_enfant = "00"
    enfant = NULL          && copie de l'enregistrement
    enfant_initial = NULL  && idem sert à détecter les modifications
    erreur = -1            && -1 non initialisé, 0 non trouvé, 1 ok
    modifie = .F.          && .T. si l'élève a été modifié

    PROCEDURE init
      * mettre ici le code qui doit être exécuté à l'instanciation de l'objet
      WAIT WINDOW "Instanciation à partir de un_enfant_demo2" NOWAIT
    ENDPROC

    PROCEDURE destroy
      * mettre ici le code qui doit être exécuté à la destruction de l'objet
      WAIT WINDOW "destruction de l'objet " NOWAIT
    ENDPROC
ENDDEFINE && CLASS un_enfant_demo2 AS custom
```

Dans la fenêtre de commande, écrivons : enf = CREATEOBJECT('un_enfant_demo2') le message adéquat s'affiche. Écrivons : enf = NULL le message correspondant s'affiche.

Allons plus loin[7], avec la méthode _assign (une introduction aux méthodes _assign et _access a été faite au début de ce chapitre**), nous allons pouvoir lancer la méthode de recherche de l'enfant dans la table lorsque l'on met à jour la propriété 'code_enfant' (à condition que la propriété code_famille soit correcte). On va en profiter pour écrire une méthode qui imprime les données de l'enfant. Cela donne[8] :**

```
DEFINE CLASS un_enfant AS CUSTOM
    code_famille = "00000"
    code_enfant = "00"
    enfant = NULL            && copie de l'enregistrement
    enfant_initial = NULL  && idem sert à détecter les modifications
    * on pourra détecter précisément les modifications en comparant les deux objets 'enfant' ( sur
    * lequel on travaille) et 'enfant_initial' qui mémorise l'état initial. Avec enfant_initial, on peut
    * gérer facilement un bouton du style "Reprise des valeurs initiales" ou "abandon des
    * modifications faites"
    erreur = -1     && -2 erreur, -1 non initialisé, 0 non trouvé, 1 ok, 2 anomalie
    modifie = .F. && .T. si l'élève a été modifié et que, donc, les données doivent être enregistrées
    creation_vue = .F.     && .T. si la vue utilisée pour la lecture de l'enfant était déjà active

    * partie 'programmes' (méthodes) de l'objet (de la classe) un_enfant
    PROCEDURE init  && init est un mot réservé
       * mettre ici le code qui doit être exécuté à l'instanciation de l'objet
       DODEFAULT()
       * dans ce cas précis, DODEFAULT() ne sert à rien parcequ'il n'y a pas de méthode init() dans
       * la classe de base custom. Mais je le met systématiquement de peur de l'oublier dans une
       *  sous_classe ! et aussi à cause de la présence de ces commentaires.
    ENDPROC && init
```

7 Je répeterai souvent 'allons plus loin' : c'est effectivement le but de ce document !

8 j'ai une tendance à mettre les méthodes _assign et _access entre les constructeurs (init, destroy) et les méthodes normales. Il ya peut-être une meilleure manière de ranger ces méthodes.

```
PROCEDURE code_enfant_assign  && lecture des données de l'enfant
    * il suffira de mettre le code de l'enfant (après avoir mis à jour le code de la
    * famille) pour que la lecture de l'enregistrement correspondant se fasse
    * automatiquement : c'est magique !
    * NOTE : une procédure _assign est forcément 'hidden' (non accessible hors de la classe)
    LPARAMETERS punewval
    * la nouvelle valeur est dans punewval, elle n'est pas encore assignée à code_enfant,
    * c'est à cette méthode de le faire (si nécessaire)

    IF punewval == This.code_enfant
        * on a rien à faire si on met la même valeur qu'avant
    ELSE
        * avant de lire les données de l'enfant, on vérifie que toutes les conditions soient bonnes
        IF VARTYPE(This.code_famille) == "C" AND LEN(This.code_famille)=5 ;
          AND VARTYPE(punewval)  = "C" AND LEN(punewval) = 2
            * il faudrait vérifier si on avait pas déjà un autre enfant et si oui il faudrait l'enregistrer
            * avant de lire celui-ci !
            * Les classes non visuelles vous permettent d'écrire une seule fois toutes les vérifications
            * nécessaires aux données. Cela peut remplacer et/ou compléter les règles de validité ou
            * d'intégrité référentielle des bases de données (en particulier donc dans les applis
            * n'utilisant pas les BDD).
            STORE punewval TO lcenfant, This.code_enfant
            This.lecture_enfant()
            * maintenant this.code_enfant contient la valeur qu'on lui a mis
        ELSE
            WAIT WINDOW "Erreur paramètre" NOWAIT
            This.erreur = -2
        ENDIF && VARTYPE(punewval)  = "C" AND LEN(punewval) = 2
    ENDIF && punewval == This.code_enfant
ENDPROC && code_enfant_assign && lecture des données de l'enfant

PROCEDURE destroy && mettre ici le code à exécuter à la destruction de l'objet
    LOCAL lcvue

    * au cas où on aurait oublié d'enregistrer les données de l'enfant, le dernier
    * événement où l'on peut agir est cet événement DESTROY.
    * Remarquez la facilité !
    IF This.modifie = .T.
        This.enregistrement_enfant()
    ENDIF && This.modifie = .T.
    * j'utilise beaucoup les vues et les classes non visuelles, il faut donc bien gérer l'ouverture et
    * la fermeture de chaque vue, plusieurs objets pouvant utiliser la même vue.
    IF This.creation_vue = .T. && Seul l'objet qui a créé la vue peut la fermer
        USE IN ("v1e"+ This.code_famille + This.code_enfant)
    ENDIF && creation_vue = .T.
    DODEFAULT() && même remarque que pour init
ENDPROC && destroy
```

```
PROCEDURE lecture_enfant && recherche de l'enfant à partir de son code
    * Dans cette méthode, vous allez pouvoir faire toutes les conversions de données nécessaires
    * après la lecture (décryptage, décodage des données, ..)
    * Tous ces traitements ne seront écrits qu'une seule fois : ici !
    LOCAL lcvue, lccodcou, lcenfant
    * lccodcou, lcenfant sont des variables 'standardisées' pour toutes les vues paramétrées

    lccodcou = This.code_famille
    lcvue = "v1e"+ lccodcou+ lcenfant  && NDLR : je suis un fanatique des vues !
    * la base de données 'classes_non_visuelles' est fournie par ailleurs.
    * ATTENTION la même vue peut être utilisée par plusieurs objets identiques
    * il faut donc faire attention au problème suivant :
    * on ne peut pas avoir un environnement de données privé avec une classe
    * non visuelle et donc une même vue peut être ouverte plusieurs fois
    * (seul l'objet qui l'a ouverte peut la fermer).
    IF ! USED(lcvue)
       SELECT 0
       * NOTE : il est TRES dangereux de mettre une clause AGAIN dans l'ouverture d'une vue
       USE classes_non_visuelles!un_enfant ALIAS (lcvue) SHARED
       This.creation_vue = .T.
    ELSE
       SELECT (lcvue)
       REQUERY()
    ENDIF && ! USED(lcvue)
    IF RECCOUNT() = 0
       This.erreur = 0 && non trouvé
    ELSE
       This.erreur = IIF( RECCOUNT() = 1, 1, 2) && si un seul enreg : OK
    ENDIF && RECCOUNT() = 0
    IF This.erreur = 1 && on a une ligne et une seule, c'est parfait
       SCATTER NAME This.enfant
       SCATTER NAME This.enfant_initial
       * surtout ne pas écrire this.enfant_initial = this.enfant ! ! ! ! !
       * les deux propriétés pointeraient sur les mêmes données
       WAIT WINDOW "Lecture de " + ALLTRIM(This.enfant.nomenf)+" "+ ;
         ALLTRIM(This.enfant.prenom)+ " en classe de "+ ;
         This.enfant.classe NOWAIT
       * vous allez pouvoir mettre ici toutes les conversions de données nécessaires au
       * traitement de l'enregistrement.
       * Ces conversions ne seront écrites qu'une seule fois : ici !
    ELSE && il y a un problème
       SCATTER NAME This.enfant BLANK
       SCATTER NAME This.enfant_initial BLANK
       WAIT WINDOW " Problème lecture de " + lccodcou+"."+lcenfant NOWAIT
    ENDIF && This.erreur = 1
    This.modifie = .F.
    * Au cas où vous auriez à changer de méthode d'enregistrement (vous passez de tables libres
    * en BDD VFP avec vues locales puis plus tard en BDD extérieure avec vues distantes,
    * vous n'aurez qu'à modifier cette méthode sans toucher au reste !
ENDPROC && lecture_enfant
```

```
    PROCEDURE enregistrement_enfant
        * Tableupdate de la vue + traitement des erreurs
        * reste à écrire (je n'ai pas voulu alourdir la démonstration)
        * Dans cette méthode, vous allez pouvoir faire toutes les conversions de données nécessaires
        * avant l'enregistrement, compléter et/ou remplacer les règles de validité ou d'intégrité
        * référentielle, etc etc ... Tous ces traitements ne seront écrits qu'une seule fois : ici !
        WAIT WINDOW "Normalement : enregistrement des données de l'enfant" ;
          + CHR(13)+ This.code_famille+"."+This.code_enfant+ " "+ ;
          ALLTRIM(This.enfant.nomenf)+" "+ ALLTRIM(This.enfant.prenom) NOWAIT
        This.modifie = .F.
        * Au cas où vous auriez à changer de méthode d'enregistrement (vous passez de tables libres
        * en BDD VFP avec vues locales puis plus tard en BDD extérieure avec vues distantes,
        * vous n'aurez qu'à modifier cette méthode sans toucher au reste !
    ENDPROC && enregistrement_enfant

    PROCEDURE impression && cette méthode est obsolète ;-))
        WITH This.enfant
            ? .codfam+ "."+ .enfant+ " "+.nomenf+ " "+ .prenom + ;
              IIF(.sexe == "F", "née", "né") + " le "+ DTOC(.datenai)+ ;
              " classe: "+ .classe
        ENDWITH  && This.enfant
    ENDPROC

    * NOTE : finalement, vous allez écrire dans cette classe toutes les méthodes
    * (procédures, codes) qui concernent un enfant. Ces méthodes ne seront écrites
    * qu'une seule fois : ici. D'une manière générale, on va 'sortir' les traitements des
    * formulaires pour les rapprocher des données. Nous verrons plus loin l'intérêt de
    * cette façon de faire[9].
ENDDEFINE && CLASS un_enfant AS CUSTOM
```

et l'on va pouvoir écrire (même dans la fenêtre de commande ce qui est très intéressant pour le déboguage) :

```
loenf = CREATEOBJECT( 'un_enfant')
loenf.code_famille = '97040'
loenf.code_enfant = '01'
```

cette dernière instruction provoquant la lecture des données de l'enfant comme on peut le vérifier par exemple avec le débogueur (SET STEP ON ou DEBUG puis fenêtre 'local' ou 'espion'). Pour imprimer les données de l'enfant, il suffit d'écrire :

```
loenf.impression()
```

Pour enregistrer les modifications, il suffira d'écrire (et cela fonctionnera quand la méthode sera écrite ! !) :

```
loenf.enregistrement_enfant()
```

Et on détruira l'objet par : `loenf = NULL`

Il n'y a pas plus simple !

9 Je me pose la question de savoir si la procédure de création des vues relatives à une table doit être faite dans une telle classe ou si les créations de vues doivent être regroupées dans une procédure indépendante en particulier dans une procédure stockée. Pour l'instant et seulement pour une raison historique elles sont, dans mes applications, dans une procédure stockée. Ceci ne concerne, bien sûr, que ceux qui, comme moi, préfèrent créer les vues par code !

Je reprend un commentaire fait plus haut : vous allez mettre dans la classe qui contient un certain type de données (dans cet exemple un enfant) TOUS les traitements qui concernent ces données. Il est évident que

1. **le code n'est écrit qu'une seule fois;**
2. **les tests sont très faciles à faire car chaque commande (chaque méthode) peut être lancée de la fenêtre de commande;**
3. **la maintenance est facile surtout si vous mettez beaucoup de commentaires;**
4. **dans le cas d'un gros développement, chaque programmeur travaille sur une partie bien isolée de l'application**

Dans l'architecture n-tier (voir l'exposé de Toni Feltmann à Bron), on va compiler dans une (ou plusieurs) DLL toutes les classes qui servent à l'accès aux données. C'est ce que Toni appelle la 'séparation logique'. Mais ces DLLs peuvent aussi être sur une autre machine (au 'pire', elles peuvent constituer un web-service !) : c'est la séparation physique prônée dans cette architecture n-tier. Les avantages lors du développement, de la mise au point et plus tard lors des modifications de structures sont évidents.

Pendant que nous sommes dans la technique, je voudrai vous montrer trois classes qui sont intéressantes car elles introduisent des notions inhabituelles.

D'abord, voici une classe qui ne contient que des méthodes et aucune propriété : c'est une manière de ranger des fonctions et des procédures 'standards' que, d'habitude, on range dans un fichier de fonctions :

```
DEFINE CLASS fonc_dates AS custom[10]
    * toutes les méthodes concernant les dates; cette classe n'a pas de propriété !

    PROCEDURE init && rien à faire mais je le met quand même.
    ENDPROC

    FUNCTION moicivsco && mois civil vers mois scolaire : septembre (9) devient 2
        LPARAMETERS m
    RETURN IIF( BETWEEN(m,1, 12), IIF(BETWEEN(m,1,7),m+5,m-7), 0)

    FUNCTION moiscociv  && mois scolaire vers mois civil
        LPARAMETERS m  && mois scolaire 1 = aout
    RETURN IIF(BETWEEN(m,1,12),IIF(BETWEEN(m,1,5), m+7, m-5),0)

    FUNCTION datdebmoi  && date du premier jour du mois passé en paramètre
        LPARAMETERS m && m est un numéro de mois civil 1-> janvier
    RETURN CTOD("01/"+STR(m,2)+"/"+STR(IIF(m<8, annscola+1, annscola),4))

    FUNCTION datfinmoi  && date fin du mois passé en paramètre
        LPARAMETERS m && m est un numéro de mois civil 1-> janvier
    RETURN GOMONTH(CTOD("01/"+STR(m,2)+"/"+ ;
            STR(IIF(m<8, annscola+1, annscola),4)),1)-1

ENDDEFINE && CLASS fonc_dates AS custom
```

10 c'est la première que j'ai faite : cela se voit !

une fois cette classe instanciée, on peut appeler chaque méthode comme une fonction standard :

```
fonction_dates = CREATEOBJECT('fonc_dates')
toto = fonction_dates.datdebmoi( 3)
```

C'est donc un moyen de ranger les innombrables fonctions dont on a besoin dans une application par groupes fonctionnels. Dans vos propres développements, vous penserez à mettre des noms de fonctions plus explicites !

La deuxième est plus intéressante. Comme pour les exemples précédents, nous allons faire une classe qui contient des données (ici toutes les dates utilisées dans l'application de gestion d'école) et les méthodes associées. Mais nous allons chercher (évaluer) chaque donnée au moment où on en a besoin et non à l'initialisation. Par exemple, on aura besoin des dates des trimestres et des factures que lors du calcul de ces factures c'est à dire 3 fois par an ... On n'ira donc pas les chercher inutilement. Pour parler de choses plus gaies, nous nous intéresserons aux dates des vacances ! (la classe complète est visible dans le fichier demo_gestion_dates.prg ci-joint).

Vous y noterez les points suivants :

1. **l'usage de la clause HIDDEN pour éviter que certaines méthodes soient accessibles « de l'extérieur ».**
2. **La facilité avec laquelle on a pu modifier une donnée structurelle importante : dans la version DOS de cette application (qui fonctionne toujours) l'année scolaire est numérique et dans la version VFP, c'est une lettre code ('A' pour 1992) qui la remplace (n'en chercher pas les raisons profondes, c'est « historique »)**

```
DEFINE CLASS dates AS custom && toute la gestion des dates de babazou
    * ensemble des dates principales utilisées dans l'application
    * les principales méthodes sont :
    *   lecture_dates_principales  lecture rentree et sortie
    *   enregistrement_dates_principales
    *   lecture_annee_scolaire annee_scolaire, lettre_annee, semaine1, rentree, sortie
    *   moicivsco  mois civil vers mois scolaire
    *   moiscociv  mois scolaire vers mois civil
    *   datdebmoi  date du premier du mois passé en paramètre
    *   datfinmoi  date fin du mois passé en paramètre
    *   anneenum_to_lettreannee  1992 -> A 2005-> N
    *   lecture_vacances
    *   ajout_vacances
    *   modification_vacances
    *   suppression_vacances
    *   enregistrement_vacances

    annee_scolaire = 0  && année scolaire numérique
        * cette propriété a 2 méthodes _assign et _access
        * _assign va permettre de mettre à jour les propriétés annexes et
        * _access va permettre la mise à jour de annee_scolaire au moment de la première utilisation
        * Il est important que annee_scolaire soit initialisé à 0.
    lettre_annee = ""  && lettre code année scolaire
        * cette propriété a 2 méthodes _assign et _access
    texte_annee = ""    && ex '2004'
    texte_annee_scolaire = "" && ex '2004-2005'
    rentree = {}        && date de la rentrée
    julrentree = 0      && date julienne correspondant à la rentrée
    sortie  = {}        && date de la sortie
```

```
lundi_semaine1 = {} && premier lundi qui suit le 1er août qui précède la rentrée
    * Ce lundi est la base de tous les calculs de date pour l'année scolaire
    * en particulier lorsqu'il s'agit de retrouver le bit correspondant à une date
    * donnée dans une chaîne de 53 caractères.
julsem1 = 0        && numéro du jour julien correspondant à lundi_semaine1
nbre_trimestre = 0  && nombre de trimestre
DIMENSION trimestres[1,2] && date début/fin de chaque trimestre
nbre_facture = 0    && nombre de factures pour l'année scolaire
DIMENSION factures[1, 2]  && dates et codes des factures
nbre_vacances = 0   && nombre de périodes de vacances ( + _access)
DIMENSION vacances[1,3] && dates des vacances 1 début  2 fin  3 libellé ( + _access)
yamodif_vacances = .F.  && .T. si les vacances ont été modifiées

PROCEDURE init
   * je préfère mettre la méthode init même si elle ne sert pas.
ENDPROC && init

PROCEDURE destroy && jme 18/08/2005
    WITH This
        * si on ne veut pas enregistrer les vacances en détruisant l'objet,
        * il faut penser à remettre à .F. yamodif_vacances
        * (l'OOP permet des astuces qui sont difficilement réalisable autrement !)
        IF .yamodif_vacances
           .enregistrement_vacances()
        ENDIF && yamodif_vacances
    ENDWITH  && This
    DODEFAULT()
ENDPROC && destroy

* l'application est tirée d'une appli DOS qui fonctionne encore mais entretemps on
* a modifié la gestion des années scolaires ...
* les évènements _ASSIGN nous permettent de gérer ce problème facilement
* (avant on n'utilisait que des années numériques, maintenant que des lettres code)
HIDDEN PROCEDURE annee_scolaire_ASSIGN
    * chaque fois que l'on met une valeur dans annee_scolaire on passe ici
    * cette méthode n'est pas accessible de l'extérieur
    LPARAMETERS pnannee

    WITH This
        .annee_scolaire = pnannee
        * je met toujours beaucoup de sécurité dans mes applis ...
        * NOTE : remarquez qur ci-dessous, j'utilise pnannee au lieu de .annee_scolaire
        *          c'est 'important' sinon chaque lecture de .annee_scolaire va
        *          déclencher la méthode annee_scolaire_access !!
        IF pnannee > 1991 && date mise en place de l'appli !
            * le simple fait de mettre à jour l'année scolaire va mettre à jour toutes
            * les données annexes. Lorsque le programmeur travaillera sur le 'niveau
            * supérieur', il n'aura pas à s'en soucier.)
            .lettre_annee = CHR(65 + pnannee - 1992)
            .texte_annee = STR( m.pnannee,4)
            .texte_annee_scolaire = .texte_annee+"-"+STR( m.pnannee+1,4)
            .lundi_semaine1 = CTOD("01/08/" + .texte_annee)
```

```
            DO WHILE DOW( .lundi_semaine1,1) <> 2
                .lundi_semaine1 = .lundi_semaine1 + 1
            ENDDO
            .julsem1 = VAL( SYS(11, .lundi_semaine1))
        ELSE
            STORE "" TO .lettre_annee, .texte_annee, .texte_annee_scolaire
            .lundi_semaine1 = {}
            .julsem1 = 0
        ENDIF && pnannee > 1992
        * quand on change d'année scolaire, il faut RAZ les vacances, les factures, ...
        * on le fait automatiquement ici, le programmeur n'a plus à s'en soucier
        .raz_dates_annee()
        * je moucharde beaucoup; c'est à dire que j'enregistre dans une table toutes les opérations
        * faites par l'opérateur ou les zones franchies par le programme surtout si c'est inhabituel.
        IF USED("vfpmou")
            INSERT INTO vfpmou VALUE (datefox, SECONDS(), 70100, ;
            "annee_scolaire_assign", "elvclass.dates", "", "", "", pnannee)
        ENDIF && USED("vfpmou")
        * et mettre à jour les dates principales (il ne faut pas lire l'année scolaire
        * car cela remettrait .annee_scolaire à la valeur initiale !)
        .lecture_dates_principales()
    ENDWITH  && This
ENDPROC && annee_scolaire_ASSIGN

HIDDEN PROCEDURE annee_scolaire_ACCESS
    * on veut lire l'année scolaire, on vérifie qu'elle a bien été mise à jour
    * avec cette simple méthode on laisse la liberté au développeur de mettre
    * à jour annee_scolaire au début de l'application ou de ne rien faire
    * et dans ce cas la propriété est mise à jour à la première utilisation. Je trouve
    * cela merveilleux !
    IF This.annee_scolaire = 0
        * l'année scolaire n'a pas été mise à jour, on le fait
        This.lecture_annee_scolaire()
        * j'ai vérifié : malgré que l'on relise this.annee_scolaire dans
        * This.lecture_annee_scolaire, on ne relance pas cette méthode _access.
        * c'est important car sinon on arriverait dans une boucle infernale
    ENDIF && This.annee_scolaire = 0
    RETURN This.annee_scolaire
ENDPROC && annee_scolaire_ACCESS

HIDDEN PROCEDURE lettre_annee_ASSIGN
    LPARAMETERS pclettre
    * VFP est intelligent ! dans cette méthode on met à jour This.annee_scolaire sur  laquelle il y a
    * un _assign qui remet à jour lettre_annee. Il pourrait donc y avoir une boucle infinie, mais non !
    * par contre annee_scolaire_assign est bien lancée

    This.lettre_annee = pclettre
    This.annee_scolaire = 1992 - 65 + ASC(pclettre)
ENDPROC && lettre_annee_ASSIGN
```

```
HIDDEN PROCEDURE lettre_annee_ACCESS
    * on veut lire la lettre code de l'annee, on vérifie qu'elle a bien été mise à jour
    * pour les commentaires voir annee_scolaire_ACCESS ci-dessus.
    IF This.annee_scolaire = 0 && c'est bien annee_scolaire que l'on teste !
        * l'année scolaire n'a pas été mise à jour, on le fait
        This.lecture_annee_scolaire()
    ENDIF && This.annee_scolaire = 0
    RETURN This.lettre_annee
ENDPROC && lettre_annee_ACCESS

HIDDEN PROCEDURE raz_dates_annee
    WITH This
       STORE 0 TO .nbre_trimestre, .nbre_facture, .nbre_vacances
       DIMENSION .trimestres[1,2], .factures[1,2], .vacances[1,3]
       STORE {} TO .vacances[1,1], .vacances[1,2], .factures[1,1], ;
          .trimestres[1,1], .trimestres[1,2]
       STORE "" TO .vacances[1,3], .factures[1,2]
    ENDWITH  && This
ENDPROC && raz_dates_annee

* maintenant on attaque les vacances. On a deux propriétés qui les concernent :
* nbre_vacances donnent le nombre de vacances de l'année scolaire et le tableau
* vacances[ ] contient les caractéristiques de chaque période de vacances.
* comme pour l'année scolaire, on laisse le choix au programmeur de mettre à
* jour ces deux propriétés quand il le souhaite, sinon (j'allais dire 'au plus tard') elles
* sont mises à jour au moment où on en a besoin.
HIDDEN PROCEDURE nbre_vacances_access
    WITH This
       IF .nbre_vacances = 0
          * a priori les vacances ne sont pas encore à jour, on le fait ici
          IF .annee_scolaire > 0
             * il se peut que l'année scolaire n'ait pas été mise à jour. Mais
             * on vient de la lire (le simple fait de faire le test provoque bien la
             * lecture de l'année scolaire d'accord ?) donc de la mettre à jour.
             * Par contre, si elle n'est pas définie dans parametr, elle est toujours
             * nulle et ce n'est pas la peine de chercher les vacances !
             * en résumé : si on lit nbre_vacances alors que rien n'a encore été
             * initialisé, on initialise tout !!!! C'est automagique !
             .lecture_vacances()
          ELSE
             .nbre_vacances = 0
             DIMENSION .vacances[1,3]
             STORE {} TO .vacances[1,1], .vacances[1,2]
             STORE "" TO .vacances[1,3]
          ENDIF && .annee_scolaire > 0
       ENDIF && .nbre_vacances = 0
       RETURN .nbre_vacances
    ENDWITH  && This
ENDPROC && nbre_vacances_access
```

```
HIDDEN PROCEDURE vacances_access && attention vacances est un tableau !
    * idem nbre_vacances_access par contre vous pouvez voir comment on traite
    * une méthode access sur un tableau.
    LPARAMETERS lnligne, lncolonne

    WITH This
       IF .nbre_vacances = 0
          IF .annee_scolaire > 0 && il y a un _access sur annee_scolaire
               .lecture_vacances()
          ELSE
               .nbre_vacances = 0
               DIMENSION .vacances[1,3]
               STORE {} TO .vacances[1,1], .vacances[1,2]
               STORE "" TO .vacances[1,3]
          ENDIF && This.annee_scolaire > 0
       ENDIF && This.nbre_vacances = 0
       RETURN .vacances[ m.lnligne, m.lncolonne) && un élément du tableau
    ENDWITH  && This
ENDPROC && nbre_vacances_access

PROCEDURE lecture_dates_principales
        * voir le fichier annexe
ENDPROC &&  lecture_dates_principales

PROCEDURE lecture_annee_scolaire
    * on lit l'année scolaire courante dans la table des paramètres puis
    * on met à jour les dates principales (rentrée, sortie, ...)
    * je laisse cette méthode, car il y a une astuce un peu plus bas : je met la propriété
    * annee_scolaire dans une variable locale parce que j'utilise le contenu dans une
    * boucle. En gardant annee_scolaire, je déclencherai la méthode access à chaque
    * fois alors qu'en la copiant dans une variable locale je ne déclenche cette
    * méthode access qu'une seule fois.
    LOCAL lnseldep, lnannee

    WITH This
       lnseldep = SELECT(0)
       SELECT 0
       USE (repfich+"parametr") ALIAS prmtlas NOUPDATE SHARED AGAIN
       LOCATE FOR prmtlas->code = "AS "  && année scolaire
       IF FOUND()
          STORE prmtlas->num TO .annee_scolaire, lnannee
          * le fait de mettre à jour This.annee_scolaire va déclencher _assign
          * et donc la mise à jour des autres valeurs liées ... C'est magique !
          * et cela RAZ les données des trimestres, factures, vacances, ....
          * on met la propriété annee_scolaire dans lnannee pour
          * éviter de déclencher la méthode access à chaque LOCATE
          LOCATE FOR prmtlas->code = "DRT" AND prmtlas->num = m.lnannee
          STORE IIF( FOUND(), prmtlas->date, .lundi_semaine1) TO .rentree
          STORE VAL( SYS(11, .rentree)) TO .julrentree
          LOCATE FOR prmtlas->code = "DSO" AND prmtlas->num = m.lnannee
```

```
            STORE IIF( FOUND(), prmtlas->date, daterentr +320) TO .sortie
          ENDIF
          USE
          SELECT (m.lnseldep)
       ENDWITH  && This
    ENDPROC &&  lecture_annee_scolaire

    PROCEDURE lecture_vacances
       * on ne met à jour .nbre_vacances qu'à la fin de cette méthode pour éviter de
       * déclencher les méthodes assign et access à chaque itération.
       * voir le fichier annexe
    ENDPROC && lecture_vacances

    PROCEDURE enregistrement_dates_principales && jme 19/08/2005
       * voir le fichier annexe
       * note : toutes les modifications sont « horodatées » !
    ENDPROC && enregistrement_dates_principales

    PROCEDURE enregistrement_vacances && jme 04/08/2005
       * voir le fichier annexe
    ENDPROC && enregistrement_vacances && jme 04/08/2005

    PROCEDURE ajout_vacances  && jme 18/08/2005
       * voir le fichier annexe
    ENDPROC && ajout_vacances

    PROCEDURE modification_vacances &&  jme 18/08/2005
       * voir le fichier annexe
    ENDPROC && modification_vacances

    PROCEDURE suppression_vacances &&  jme 18/08/2005
       * voir le fichier annexe
    ENDPROC && suppression_vacances

    FUNCTION moicivsco && mois civil vers mois scolaire
       * septembre (9) devient 2
       LPARAMETERS m
    RETURN IIF( BETWEEN(m,1, 12), IIF(BETWEEN(m,1,7),m+5,m-7), 0)

    FUNCTION anneenum_to_lettreannee
       LPARAMETERS pnanneenum
       *!  RETURN CHR(65+ IIF(PCOUNT()=0, This.annee_scolaire, pnanneenum) - 1992)
    RETURN CHR(IIF(PCOUNT()=0, This.annee_scolaire, pnanneenum) - 1927)

ENDDEFINE && CLASS dates AS custom
```

Dans le fichier CLASSES_NON_VISUELLES.PRG vous trouverez une classe papa_maman qui est basée sur le même principe que la classe un_enfant explicitée ci-dessus. Et surtout, vous trouverez une classe une_famille, toujours basée sur le même principe mais qui récupère les données des parents en instanciant la classe papa_maman et les données des enfants en instanciant autant de fois que nécessaire la classe un_enfant (les instanciations ayant lieu au moment de l'exécution). En écrivant simplement :

```
famille = CREATEOBJECT('une_famille')
famille.code_famille= '97040'
```

on obtient l'ensemble des données de la famille !.
En écrivant : famille.impression() on imprime les données de la famille (et regardez comme cette méthode est simple !)

En exercice, je vous propose d'ajouter la possibilité de passer le code famille en paramètre du CREATEOBJECT() …

Vous trouverez aussi, dans le fichier CLASSES_NON_VISUELLES.PRG une classe UNE_FAMILLE_COMPLETE qui est une sous-classe de UNE FAMILLE mais dans laquelle on a ajouté quelques propriétés et méthodes[11]. Vous regarderez avec intérêt l'utilisation des clauses HIDDEN et PROTECTED.

Lors du calcul des factures dans mon application de gestion d'école, un objet instancié à partir de une_famille_complete contient :

1. **les données de la famille : coordonnées, barème, réduction...**
2. **la liste des règlements déjà effectués pour l'année scolaire avec les anomalies éventuelles**
3. **les valeurs 'familiales' de la facture en cours d'élaboration**
4. **la liste des enfants de cette famille inscrits pour l'année scolaire avec, pour chaque enfant :**
   - **4.1ses coordonnées (classe, rentrée effective, radiation éventuelle, ...)**
   - **4.2ses jours de cantine**
   - **4.3ses présences à l'étude ou à la garderie**
   - **4.4ses activités**
   - **4.5les valeurs en cours de calcul pour cette facture**

Le tout 'accessible' par une seule 'adresse' !

On peut lancer des procédures ou fonctions :

- **calcul du nombre de repas des enfants pour le troisième trimestre:**

```
FOR m.linindice = 1 TO lofamille.nombre_enfants
    lofamille.enfants[ m.lnindice].calcul_nbre_repas( ;
     lofamille.enfants[ m.lnindice].nbre_repas, ;
       facture_courante.debut_periode, facture_courante.debut_periode)
NEXT
```

ou mieux :

```
lddebut = facture_courante.debut_periode
ldfin = facture_courante.fin_periode
FOR m.linindice = 1 TO lofamille.nombre_enfants
    WITH lofamille.enfants[ m.lnindice]
        .calcul_nbre_repas( .nbre_repas, m.lddebut, m.ldfin)
    ENDWITH
NEXT
```

11 Pour des raisons de simplicité, c'est une version 'ancienne' que je vous montre. La version actuelle est ... complète !

Steven BLACK[12] nous a montré à Bron les 'design patterns' (que l'on pourrait traduire par 'motifs' ou 'modèles'). Dans son exposé, qui pousse le langage VFP (et non la partie gestion des données) à un point incroyable, il a introduit ce qu'il appelle un 'décorateur' : une classe qui modifie le fonctionnement d'une autre classe sans être une de ses sous-classes ! C'est utile quand on veut modifier le fonctionnement d'une classe alors qu'on n'a pas ses sources. En voici le principe.

Soit la classe :

```
DEFINE CLASS classe_fonctionnelle AS custom
   propriete_un = ''
   propriete_deux = 0

   PROCEDURE methode_un
   ENDPROC

   PROCEDURE methode_deux
   ENDPROC
ENDDEFINE && classe_fonctionnelle AS custom
```

On peut créer une classe qui change le fonctionnement de la classe ci-dessus :

```
DEFINE CLASS decorateur AS custom
   cf = NULL  && pointeur sur la classe fonctionnelle

   PROCEDURE init
      * on pourrait passer le nom de la classe fonctionnelle en paramètre du createobject et
      * on la récupérerait ici avec un LPARAMETERS
      This.cf = CREATEOBJECT('classe_fonctionnelle')
   ENDPROC

   PROCEDURE destroy
      * Pour que VFP puisse détruire la classe fonctionnelle quand tous ses décorateurs
      * sont détruits, il faut penser à détruire le 'lien de subordination' entre le décorateur
      * et sa classe fonctionnelle
      This.cf = NULL
   ENDPROC

   PROCEDURE methode_un
      ici on fait les traitements à faire avant d'appeler la méthode fonctionnelle
      IF traitement_normal_a_faire = .T.
         * on n'est même pas obligé d'appeler la méthode fonctionnelle !
         This.cf.methode_un()
      ENDIF
      ici on fait les traitements à faire après l'appel de la méthode fonctionnelle
   ENDPROC

   PROCEDURE methode_deux
      idem
   ENDPROC

ENDDEFINE && decorateur AS custom
```

12 www.stevenblack.com

A partir de VFP8, on peut même utiliser This_access pour ajouter des méthodes au décorateur sans avoir à manipuler les méthodes déjà existantes. This_access est executé à chaque accès à une méthode ou une propriété de This. La méthode doit retourner un pointeur vers un objet :

```
DEFINE CLASS decorateur_universel AS custom
   cf = NULL  && pointeur sur la classe fonctionnelle

   PROCEDURE init
      * on pourrait passer le nom de la classe fonctionnelle en paramètre du createobject et
      * on la récupérerait ici avec un LPARAMETERS
      This.cf = CREATEOBJECT('classe_fonctionnelle')
   ENDPROC

   PROCEDURE destroy
      * Pour que VFP puisse détruire la classe fonctionnelle quand tous ses décorateurs
      * sont détruits, il faut penser à détruire le 'lien de subordination' entre le décorateur
      * et sa classe fonctionnelle
      This.cf = NULL
   ENDPROC

   FUNCTION This_Access( tc_nom_methode)
      * avec cette méthode, on n'est pas obligé de créer une méthode dans le décorateur
      * pour chaque méthode de la classe fonctionnelle. Ce décorateur peut donc être utilisé
      * avec des classes sur lesquels on a très peu d'information (pas de sources, ...°)
      * Steven note cependant une certaine dégradation des performances qui n'est pas
      * sensible si l'appel au décorateur est peu fréquent mais très lourd si l'appel a lieu
      * dans une boucle (SCAN d'une table par exemple).
      IF PEMSTATUS( This, UPPER( m.tc_nom_methode), 5) = .T.
         * la méthode appelée existe dans le décorateur (la présente classe)
         * c'est celle qu'on appelle
         RETURN This
      ELSE
         * la méthode appelée n'existe pas dans le décorateur
         * on appelle la méthode dans la classe fonctionnelle !
         RETURN This.cf
      ENDIF
   ENDFUNC && This_Access

   PROCEDURE methode_a_modifier
      ici on fait les traitements à faire avant d'appeler la méthode fonctionnelle
      IF traitement_normal_a_faire = .T.
         * on n'est même pas obligé d'appeler la méthode fonctionnelle !
         This.cf.methode_a_modifier()
      ENDIF
      ici on fait les traitements à faire après l'appel de la méthode fonctionnelle
   ENDPROC

ENDDEFINE && decorateur_universel AS custom
```

Un autre moyen d'autoriser la modification du fonctionnement d'une classe est d'utiliser ce que les anglo-saxons appellent 'hook' (comme dans project_hook) et ce que j'appelle dans mes classes personnelles les méthodes ADP : A Disposition du Programmeur. Si je reprend l'exemple de la méthode enregistrement_enfant de la classe un_enfant (voir page 11), on peut écrire :

```
PROCEDURE enregistrement_enfant
    This.ADP_avant_enregistrement_enfant()
    .... traitement avant l'enregistrement
    IF TABLEUPDATE( .....) && par exemple !
        .... traitement enregistrement ok
        This.ADP_enregistrement_enfant_ok()
    ELSE
        .... traitement erreur tableupdate
        This.ADP_erreur_enregistrement_enfant()
    ENDIF
    ....traitements après l'enregistrement
    This.ADP_apres_enregistrement_enfant()
ENDPROC && enregistrement_enfant

PROCEDURE   ADP_avant_enregistrement_enfant()
    * cette procédure ne contient aucun code et pourra être surchargée sans problème
    * par le programmeur dans une sous-classe
ENDPROC

PROCEDURE   ADP_apres_enregistrement_enfant()
    * cette procédure ne contient aucun code et pourra être surchargée sans problème
    * par le programmeur dans une sous-classe
ENDPROC

PROCEDURE   ADP_enregistrement_enfant_ok()
    * cette procédure ne contient aucun code et pourra être surchargée sans problème
    * par le programmeur dans une sous-classe
ENDPROC

PROCEDURE   ADP_erreur_enregistrement_enfant()
    * cette procédure ne contient aucun code et pourra être surchargée sans problème
    * par le programmeur dans une sous-classe
ENDPROC
```

Vous autorisez donc le programmeur à intervenir (mettre du code complémentaire) lors de 4 'événements' déclenchés par l'enregistrement d'un enfant : au tout début de la méthode, si l'enregistrement s'est bien passé, si il y a une erreur et en toute fin de la procédure quelque soit le résultat de l'enregistrement.
Si vous ne faisiez pas cela, le programmeur pourrait bien surcharger la méthode enregistrement mais il ne pourrait appeler le code de la classe qu'à un seul endroit : là où il met le DODEFAULT().

Un autre usage de ces classes non visuelles est l'ajout 'dynamique' (au moment de l'execution) de méthodes à des objets déjà instanciés pour lesquels on ne peut, normalement, pas ajouter de méthodes. En clair, certains objets ont une méthode « addproperty' qui permet de leur ajouter des propriétés lors de l'exécution, nous allons voir comment simuler la méthode 'addmethod' ! Par exemple, chaque page d'un pageframe a une méthode addproperty et vous pouvez bien lui ajouter des propriétés pour personnaliser chaque page en fonction de l'environnement. Pour lui ajouter des méthodes, on va créer une classe non visuelle qui va contenir toutes les méthodes utiles, faire un 'addproperty' d'un pointeur sur la classe ... et c'est tout ! Un exemple sera plus parlant : créons une classe non visuelle bidon :

```
DEFINE CLASS page_bidon AS custom
  tata = 1
  toto = ''
  tutu = .F.

  PROCEDURE methode_un
  ENDPROC

  PROCEDURE methode_deux
  ENDPROC

  PROCEDURE methode_trois
  ENDPROC

ENDDEFINE && CLASS page_bidon AS custom
```

Dans la méthode activate d'une page d'un pageframe[13] on pourra écrire[14] :

```
This.addproperty( 'funcs', .F.)
This.funcs = newobjects(page_bidon, 'toto.prg')
This.funcs.tata = 2
This.funcs.methode_un()
```

En fait, les propriétés tata, toto, tutu ainsi que les méthodes methode_un, methode_deux et methode_trois seront accessibles de 'partout' par une adresse du style :

. . . . .pageframe.page3.funcs.toto ou . . . . .pageframe.page3.funcs.methode_un()

13 C'est un exemple !

14 En faisant en sorte que l'instanciation n'ait lieu qu'une seule fois ...

Hierarchisation des classes non visuelles.
Business Objects

En partant des données, on va être capable de créer des classes et des sous_classes de plus en plus complexes (une 'première couche' très proche des données (une classe par table par exemple), une deuxième couche un peu plus abstraite, etc …avec à coté des classes 'utilitaires') mais toutes ces classes ne font que l'interface entre l'application et les données. Vous avez vu aussi que d'autres classes pouvaient gérer des données non liées à des tables (l'exemple des dates ci-dessus). Une fois toutes ces briques construites, l'application devient beaucoup plus simple à élaborer sans ajout de lourdeur excessive (le code n'est toujours écrit qu'une seule fois). Dans les grosses entreprises, le partage du développement est grandement facilité.
Ces classes non visuelles sont faciles à créer (il n'y a pas de grandes nouveautés ni de 'philosophie' difficilement compréhensible ; on ne fait que pousser dans un sens peu courant des connaissances déjà acquises par ailleurs) ; elles sont aussi très faciles à documenter, … Les essayer c'est les adopter !

Je disais au début de ce chapitre qu'utiliser ce type de classe entrait en concurrence avec la bufferisation des données, je pense que vous le comprenez mieux maintenant. Il faut tout de même se rappeler (et vous l'avez vu dans les exemples) que l'on ne peut pas avoir d'environnement de données privé pour une tel classe.

> **Dans l'application de gestion des élèves, toutes les formulaires sont en environnement de données privé et il est fait un grand usage des vues. Je me suis trouvé devant le problème suivant : si, au cours du traitement d'une famille, on modifiait les jours de cantine ou d'étude, les modifications étaient gardées dans le buffer de la table correspondante tant que l'on ne quittait pas la famille et si, dans la même session, on demande un état financier, les modifications de cantine ou d'étude ne peuvent pas y être prises en compte. Avec une classe 'famille' complète, on passera l'objet 'famille' en paramètre au formulaire de traitement de l'état financier et toutes les informations (factures, règlements, jours de cantine, jours d'étude, barème, réductions, ...), y compris celles que l'on vient de modifier, y seront disponibles. ….**

Maintenant, si vous entendez parler de BUSINESS OBJECTS, vous saurez ce que cela veut dire !

Lors des rencontres de Bron (ATOUTFOX 2005), Toni FELTMAN a réservé le terme 'business object' à la partie métier (middleware) de l'architecture n-tier. Il faudra donc inventer un nom pour toutes ces classes qui contiennent des données et tout le code qui va avec ![15]

15 Je propose le terme de clanovi (pour CLAsse Non Visuelle). Qui dit mieux ?

OOP+

La société BULL, après avoir fabriqué des ordinateurs de moyen et haut de gamme (mais Français !) s'est tournée (nécessité oblige) vers le 'software'. Elle dispose à Grenoble d'un centre de recherche qui est dans le peloton de tête mondial.

Elle s'est beaucoup intéressée à l'Orienté Objet et à l'évolution de l'informatique dans la période récente et actuelle où l'internet arrive partout, où les ordinateurs deviennent d'une puissance incroyable et où les utilisateurs deviennent de plus en plus exigeants.

Je ne vais vous exposer, brièvement, qu'un seul point qui, comme je vous l'ai déjà dit plus haut, ne fait pas appel à des réflexions philosophiques extraordinaires ou à des techniques incompréhensibles par le simple programmeur, mais ne fait que pousser un peu plus loin les techniques vues ci-dessus.

Vous avez remarqué que tout le code fonctionnel (la partie « métier » de l'application, le 'middleware') était mis dans des classes non visuelles. Les formulaires de votre application (la partie « interface homme-machine », le GUI) est réduite à cette fonction et ne contient plus aucun élément de calcul.

Qu'est-ce qui empêche, alors, d'utiliser les mêmes classes non visuelles (la même partie métier) avec plusieurs interfaces différentes ? Rien ! C'est ainsi qu'une application développée pour du client-serveur standard, pourra être facilement mise 'en ligne' en créant une autre application dont 80% du travail est déjà fait. Mais cette application pourra aussi être transformée en un 'serveur d'automation'[16].

C'est ainsi qu'un site gouvernemental (www.changement-adresse.gouv.fr) **permet de réaliser en une seule opération un changement d'adresse. L'application 'en ligne' collecte les informations nécessaire auprès du connecté puis va envoyer des ordres à des applications (a priori une par administration concernée) qui ont été construites selon les principes de l'OOP+ et qui vont les exécuter en même temps que des opérateurs vont manipuler les mêmes classes sur leur écran ou que d'autres applications feront des statistiques en utilisant les mêmes méthodes. Et c'est parce que les principes de l'OOP+ ont été utilisés dans les administrations française depuis longtemps que ce changement d'adresse a pu être automatisé aussi facilement. Ce n'est certainement pas à cause d'une 'révolution technologique'.**

Jean à Grenoble

16 Je n'ai pas retrouvé le nom adéquat pour définir ces applications (comme Word ou Excel), qui peuvent être pilotées par une autre application (Serveur d'automation ?).