PLUS LOIN AVEC LA PROGRAMMATION OBJET :

OOP
ARCHITECTURE N-TIERS[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, à ma contribution sur les classes non visuelles et surtout à la conférence de Toni FELTMANN sur l'architecture n-tiers lors de la conférence ATOUTFOX de Bron en 2005.

Nous allons voir comment découper une application en trois couches :

1. **les données**
2. **la couche métier**
3. **les interfaces**

Ce découpage, qui est intéressant à plus d'un titre, est facilité par l'emploi de classes visuelles ou non et donc parfaitement adapté à VFP.

En se basant sur un exemple extrêmement simple, nous allons donc voir comment on sépare les différents éléments d'une application. J'essaierai de vous donner la logique qui m'a amené à prendre telle ou telle décision pour que vous puissiez facilement adapter cet exemple à vos applications.

L'exemple joint :

> **Dans une application où l'on gère beaucoup de rentrées financières, on m'a demandé d'ajouter la possibilité de saisir la banque émettrice du chèque; d'une part pour faciliter la remise en banque d'autre part pour être capable de retrouver plus facilement un débiteur. C'est cette 'gestion' ultrasimple des banques que nous allons étudier ici. Pour exécuter l'exemple, il suffit de décompresser les fichiers dans un répertoire et de lancer le formulaire MAIN.**

2 accessible sur le site d'Atoutfox

2– la base de données

Nous avons deux possibilités extrêmes pour stocker nos données : une ou plusieurs tables indépendantes ('flat tables') à l'ancienne mode ou une vraie base de données avec des règles d'intégrité, des règles de validation, des triggers, ...

La première solution, si on ne lui ajoute pas une couche supplémentaire, est très simple à mettre en œuvre, permet toutes les fantaisies mais n'offre aucune sécurité : on va pouvoir enregistrer des données non pertinentes, laisser des lignes vides, etc ... sans que VFP proteste le moins du monde.

La deuxième solution permet un contrôle très strict des données avec une garantie de validité des données enregistrées. Il pourra y avoir des erreurs dans l'application, on ne pourra pas véroler la base de données. Mais, à mon humble avis et suite à quelques expériences malheureuses, le déboguage puis la gestion des erreurs est difficile : on a bien l'indication « il y a un problème » mais déterminer quel est exactement le problème est assez lourd. Le code de contrôle des données est dispersé dans de nombreuses méthodes, chacune faisant une petite partie du contrôle et il me paraît très difficile d'avoir une vue d'ensemble.

Je vous propose donc une solution intermédiaire qui allie les avantages des deux solutions précédentes avec un seul inconvénient : il va falloir écrire du code ! Toutefois, la table 'banques' fait bien partie d'une base de données (ici 'demo') pour pouvoir bénéficier de certains avantages (noms longs, explications, ...) mais aucun trigger, aucune règle n'est programmée. Dans la théorie de l'architecture n-tier nous allons créer une classe d'accès aux données; je considère pour ma part qu'elle fait partie de la couche métier mais cela peut se discuter[3] (et cela n'a aucune importance sur l'efficacité du programme !).

J'ai des 'manies' :

1. **J'utilise beaucoup les clefs binaires qui sont très efficaces. Ces clefs sont soit enregistrées dans un champ réellement binaire (sur 4 octets) (pour mes applications VFP6 et suivants) soit dans des champs caractères sur 2 ou 4 caractères (octets) avec utilisation des fonctions BINTOC() et CTOBIN()[4]. La table banques contient donc une clef binaire sur 2 caractères. Ce qui autorise 32767 possibilités. Par contre, je prévois dans l'avenir de pouvoir gérer les agences des banques, c'est pourquoi dans la table des paiements le code est sur 4 caractères : deux pour la banque et deux qui seront utilisés pour discriminer l'agence.**
2. **Je n'aime pas la commande DELETE (qui d'ailleurs est incompatible avec les bdd Oracle ou SQL SERVER). J'emploie donc la méthode suivante : les**

3 Cette classe ne comporte pas beaucoup « d'intelligence » et sa position est discutable. Par contre la classe d'accès à la table paiement gère des choses complexes comme un règlement pour plusieurs factures, un règlement partiel, les avoirs; elle permet des calculs et des recherches complexes, elle est bien plus qu'une simple classe d'accès; elle fait partie indiscutablement de la partie métier.

4 Ces fonctions ne font aucune traduction réelle du contenu (contrairement à STR(), TRANSFORM(), ..); elles se rapprochent plus de la clause CAST de conversion de type.

enregistrements valides ont une clef 'positive', les enregistrements 'marqués pour l'effacement' ont une clef négative. Plus tard, j'essaierai de récupérer la plus ancienne ligne effacée pour y mettre l'enregistrement en cours de création et éviter ainsi d'avoir à 'packer' périodiquement la table dans une procédure de maintenance. Il faut pour cela gérer une file d'attente des lignes vides.

3. **J'utilise beaucoup les vues; pour cette seule petite table, j'en ai 3 ! Par contre il y a une petite particularité : dans la plupart des cas on a besoin de la liste des banques pour récupérer le code de l'une d'entre elles, il n'y a donc pas de raison d'aller chercher les inations de celle qui vient d'être sélectionnée. Par contre, lorsque l'on veut en créer ou en modifier une on doit bien utiliser une vue avec sendupdate = .T. qui ne contienne qu'une seule ligne.**

La table banques a la structure et les index suivants :

```
Champ     Nom de champ      Type         Largeur  Déc Index  Collate  Valeurs nulles
    1     CODE              Caractère          2      Crois.  Machine      Non
*!           contient une valeur binaire (donc BINTOC()).
*!           cette valeur est positive si la ligne est valide. négative sinon.
*!           on n'utilise pas DELETE sur cette table
*!           PAS DE CODE INFERIEUR A 10 !
    2     ABREV             Caractère          4      Crois.  Machine     Non
*!           abréviation de la banque (CA pour Crédit Agricole). Doit être unique.
    3     NOM               Caractère         40      Crois.  Machine    Non
*!           nom de la banque. Doit être unique.
    4     IBAN              Caractère         35      Non
*!           non utilisé pour l'instant
    5     BIC               Caractère          8      Non
*!           non utilisé pour l'instant
    6     FREQUENCE         Entier             4      Décr.   Machine    Non
*!           valeur numérique calculée de temps en temps. Représente le nombre
*!           "d'apparition" de la banque dans la table des paiements. Sert juste
*!           à mettre les banques les plus fréquentes en début de liste dans les
*!           combos.
    7     CREAT             DateHeure          8      Non
*!           datetime de dernière modification

 ** Total **                                  102
Page de code: 1252  (attention la table des paiements est en 850)
TAG:   CODE       Copies assemblées:   Machine    Clé CODE Candidat
TAG:   ABREV      Copies assemblées:   Machine    Clé ABREV
TAG:   NOM        Copies assemblées:   Machine    Clé NOM
TAG:   FREQUENCE  Copies assemblées:   Machine    Clé FREQUENCE      (Décroissant)
```

Cette structure est toute simple. Une 'curiosité' toutefois : l'application réelle doit rester compatible avec la version DOS qui est encore utilisée dans certains lieux. La table des paiements est donc une table DOS en page de code 850. On utilise donc souvent la fonction CPCONVERT().

Une autre curiosité due à l'application : il n'y a aucun code inférieur à la valeur 10. On verra que les codes 1 et 2 sont réservés à certaines situations (pas de banque, la banque n'est pas dans la liste, ...).

Sur cette table banques, j'ai créé les vues suivantes :

```
        CREATE SQL VIEW "BANQUES_LISTE" ;
         AS SELECT code, abrev, nom, iban, bic, frequence ;
          FROM banques WHERE CTOBIN(code) > 0 ORDER BY frequence DESC
*!          avec sendupdate = .F.

        CREATE SQL VIEW "BANQUES_UNE_RW" ;
         AS SELECT * FROM elv!banques WHERE code = ?lccode
*!          avec sendupdate = .T. (c'est la seule). Tous les champs sont modifiables.
```

Banques_liste me donne ... la liste des banques valides dans l'ordre de fréquence décroissant. C'est celle que l'on utilise 'tout le temps'. Banques_une_rw me permet de créer ou de modifier une banque particulière.

Remarquez que les nom des vues commence toutes par le même libelle (ici le nom de la table qui est le même que le nom de la 'fonction générale'). Dans la liste des vues de la base de données, elles sont donc rangées 'ensemble' ce qui facilite grandement leur recherche.

Pour cet exemple, j'ai fait un 'effort' ! Les vues ont été crées visuellement avec la commande MODIFY DATABASE.

Dans l'application réelle, toutes les vues sont créées par code (je me sers de GENDBC pour créer le fond puis je le complète/modifie). Je me pose encore la question de savoir si la définition de ces vues doit être laissée dans la base de données (procédures stockées) ou mises dans la partie métier que nous allons voir dans le chapitre suivant; pour l'instant elles sont dans la base de données. Par contre la procédure de maintenance (correction des erreurs, 'pack', reindexation, ...) est bien placée ici dans la classe d'accès aux données.

Maintenant que nous avons vu la partie 'données' de l'architecture n-tiers, nous allons nous intéresser à la plus grosse partie : la couche métier[5].

5 Je vous rappelle que philosophiquement la classe que je vais vous montrer pourrait faire partie de la couche 'données'.

3– partie métier

Cette partie métier est entièrement construite dans une classe non visuelle dont vous avez le code dans le fichier demo_banques.prg.

Pourquoi l'utilisation d'une classe ? Plusieurs raisons principales :

1. **c'est facile à construire et le code est proche des données (je veux dire que sous un seul nom on accède aux données et aux méthodes)**
2. **on va pouvoir utiliser les avantages des classes en particulier les méthodes init et destroy qui vont nous permettre de simuler un environnement de données 'privé'.**
3. **Le code d'appel aux méthodes est très 'parlant' : nom de la classe (de la fonction générale) puis nom de la fonction; cela facilite la relecture du code de l'application.**
4. **Je suis un 'objeteur de conscience' ;-)**

Il y a beaucoup de commentaires dans demo_banques.prg qui devraient vous suffire pour comprendre le fonctionnement aussi, ici, nous ne verrons que la philosophie générale.

Le premier point très important que je voudrais aborder est l'environnement de données. J'essaie de faire en sorte que chaque table, chaque vue soit ouverte avec un alias particulier (on pourra instancier N objets à partir de la même classe sans qu'il y ait de perturbation). SYS(2015) nous donne un nom unique mais il est 'illisible' en particulier dans la fenêtre de l'environnement de données; c'est pourquoi je le fais précéder de 2 caractères qui m'indique quel est le 'contenu' de l'alias : 'bq' pour la table banques, 'bl' pour la vue banques_liste, etc ... cela facilite grandement la compréhension. La table et les vues (bon ici, il n'y en a qu'une mais on crée un curseur local) sont ouvertes dans la méthodes init donc à l'instanciation de l'objet et elles le restent jusqu'à sa destruction. Je laisse au programmeur la possibilité de laisser la table et/ou une vue ouverte après la destruction; pour cela il lui suffit d'effacer la propriété qui contient l'alias correspondant.

Le deuxième point TRES important est que tous les traitements d'accès et/ou d'utilisation de la table ou des vues se font dans cette partie métier (méthodes ajouter(), modifier(), supprimer(), chercher(), rechercher(), ...). Ces traitements ne sont écrits qu'une seule fois : ici. On verra dans le chapitre suivant que l'interface s'en trouve réduite au minimum. Cette 'unicité' du code d'accès a pour conséquences :

1. **Tous les contrôles de validité, d'intégrité, de relations se font ici et une fois pour toutes. Avec un avantages important par rapport à la mise en place de triggers et de règles : la facilité de programmation (c'est du code lisible et surtout imprimable) et de gestion des erreurs. Une autre de mes manies est le mouchardage; vous trouverez (comme dans toutes mes applis) une table libre VFPMOU.DBF[6]**

6 Il n'est pas rare que cette table soit la plus grosse de toute l'application (le record : 2 ans de mouchard dans 140 Mo). Elle permet d'avoir l'histoire a posteriori de chaque traitement. En cas d'erreur, on peut donc remonter facilement à la source du problème. D'autre part, mais on le verra dans la partie interface, je moucharde aussi chaque saisie de l'utilisateur : fini les « je vous jure, je n'y ai pas touché » !!

qui contient la liste de toutes les opérations réalisées, les bugs, les curiosités. L'utilisation de cette table est très facile ici.

2. **La gestion des erreurs se fait aussi ici, dans l'environnement de l'application; l'usage du débogueur est donc très facile (le code est « procédural » ou plutôt « linéaire »).**
3. **si dans le futur on change la répartition du contrôle d'accès entre cette partie métier et la base de données (en clair si on passe d'une bdd VFP à SQL SERVER par exemple), on n'aura à modifier que la base de données et cette classe mais en aucun cas les nombreux formulaires, classes, programmes de la partie interface. Je ne l'ai pas encore fait mais si cela devait arriver, je conserverais tous les contrôles de cette partie pour ne fournir à la BDD que des enregistrements valides; cela pour conserver la facilité de gestion des erreurs et du déboguage.**
4. **Il est facile d'installer un cryptage des données pour assurer une certaine confidentialité (point faible de VFP).**

La classe banques (qui doit être la plus petite classe d'accès aux données que j'ai écrite .... depuis le début de l'année !) contient donc les procédures d'ajout, modification et suppression d'une banque mais aussi la procédure de maintenance (correction des erreurs, reindexation, vérification des clefs [7]). La procédure de maintenance globale ne fait donc qu'instancier chaque classe d'accès à chaque table et lancer la méthode 'standard' maintenance().

Elle contient aussi d'autres méthodes comme le calcul des fréquences ou les listes. Certaines de ces méthodes peuvent faire appel à d'autres tables et donc à d'autres classes d'accès.

Pour les détails, je vous renvoie aux nombreux commentaires du prg.

Maintenant, nous allons voir un des avantages essentiel de cette méthodologie : la facilité des tests. A partir de la fenêtre de commande, nous allons instancier un objet à partir de la classe banques et à partir de ce moment, nous aurons accès à toutes les méthodes et toutes les données :

```
USE vfpmou && c'est le mouchard !
essaibq = NEWOBJECT('banques', 'demo_banques.prg')
set  && la table et la vue sont ouvertes; le curseur local est créé
essaibq.maintenance()
select vfpmou
go bottom
browse && on voit le résultat de la maintenance de la table
essaibq.image.abrev = 'FOX'
essaibq.image.nom = 'ATOUTFOX'
? essaibq.ajouter()
essai2 = NEWOBJECT('banques', 'demo_banques.prg')
essaibq = NULL && toutes les vues et la table correspondant sont fermés
* mais on a toujours la table et les vues ouvertes pour essai2 !
essai2 = NULL
```

7 Cette procédure est censée corriger tous les problèmes qui peuvent bloquer l'enregistrement d'une ligne dans une table équipée d'un index candidat ou primaire comme la présence d'une ligne vide.

On peut faire des essais de temps de réponse ou de charge (!!) :

```
#DEFINE maxessai 100

DIMENSION ltcodeb( maxessai)
USE vfpmou IN 0 && c'est le mouchard !!!
USE demo!clefs IN 0
bq = NEWOBJECT("banques", "demo_banques.prg")
lndebut = SECONDS()
FOR m.i = 1 TO maxessai
   bq.image.abrev = TRANSFORM( m.i, "@l 999")
   bq.image.nom = "bq"+ bq.image.abrev
   llaccu = bq.ajouter()
   ltcodeb (m.i) = bq.image.code
ENDFOR &&* m.i = 1 TO 10
? "100 ajouts : " + TRANSFORM( SECONDS() - lndebut, "9999.999")
SELECT (bq.alias_liste)
BROWSE
lndebut = SECONDS()
FOR m.i = 1 TO maxessai
   bq.image.code = ltcodeb (m.i)
   llaccu = bq.supprimer()
ENDFOR &&* m.i = 1 TO 10
? "100 suppressions : " + TRANSFORM( SECONDS() - lndebut, "9999.999")
SELECT (bq.alias_liste)
BROWSE
bq.maintenance() && va effacer physiquement les banques ajoutées pour le test
bq = NULL
```

Entraînez-vous ! Vous verrez, l'essayer, c'est l'adopter. On peut vérifier très finement et très facilement chaque méthode. Une fois toutes les méthodes vérifiées on peut alors passer aux interfaces dont le développement sera alors très rapide.

Au risque de me répèter, j'insiste sur le fait que chaque opération n'est programmée qu'une seule fois pour toute l'application. Il n'y a pas d'inconvénient en terme de vitesse ou de complexité.

4– les interfaces

La classe demo.banques et le formulaire saisie_banque joints sont réduits à leur plus simple expression.

ELLES NE CONTIENNENT QUE DU CODE DE GESTION DE LA VISUALISATION ET DES APPELS A LA CLASSE METIER.

Inutile donc de s'y attarder.!!!!!.

Si maintenant vous devez travaillez sur les banques à partir d'une application internet, la page sera facile à faire ...

Revenons un instant sur la fin du chapitre précédent quand je vous donnai du code à exécuter à partir de la fenêtre de commande : nous n'avons fait qu'utiliser notre partie métier avec une autre interface visuelle (l'environnement de développement). Nous pouvons facilement piloter notre classe à partir d'une autre application comme nous l'avons fait pour l'essai des temps de réponse !.

Comme l'a expliqué Toni FELTMANN lors des rencontres de Bron, nous pouvons donc bien avoir plusieurs interfaces homme-machine qui travaillent avec la même partie métier et même une interface machine-machine comme l'automation !.

Imaginez maintenant que dans la form saisie_banque, on enregistre en clair dans un fichier majbanques.prg toutes les opérations faites (comme on pourrait aussi enregistrer les commandes tapées dans la fenêtre de commande pour nos tests) : on serait capable de 'rejouer' ces commandes et de rétablir la dernière version de la table à partir d'une sauvegarde ou de repartir d'une sauvegarde et de ne rejouer qu'une partie des commandes parce que la dernière a tout foutu en l'air. Ou ... ou ....

5– conclusion

Elle sera simple :

Vous commencez à saisir la puissance et la simplicité de cette méthodologie n-tier ?

Jean à Grenoble