Les bonnes pratiques de la programmation des macros de données

Par Christophe WARIN

Date de publication : 28 septembre 2013

Toujours à propos des évènements de table d'Access 2010, je vous propose un nouveau document consacré cette fois-ci à la méthodologie de cette nouvelle approche.

Il est destiné à un public d'un niveau intermédiaire connaissant déjà les concepts mis en jeu dans une base de données et dont la curiosité l'amène à rechercher les solutions les plus optimales. Vous y découvrirez les quelques pièges à éviter et les optimisations possibles, tout en gardant à l'esprit qu'il s'agit là d'une nouvelle fonctionnalité peu documentée puisqu'issue d'une version en cours de développement (Access 2010 Beta)

Commentez

I - Introduction....3
II - La structure du code....4
III - Modélisation....5
IV - Validation des données....8
V - Identification des données....11
VI - La Récursivité....12
VI-A - Récursivité non désirée....12
VI-B - Récursivité désirée....14
VII - Parcours d'enregistrements....18
VIII - Code VBA....21
IX - Conclusion....23

I - Introduction

A plusieurs reprises, nous avons illustré les bienfaits des évènements de table au sein d'une base de données Access. Pourtant, leur utilisation excessive ou inadaptée peut conduire à l'apparition de problèmes graves mettant en jeu la cohérence des données. Etant donné qu'il serait fastidieux de lister un à un les exemples où les évènements de table peuvent être utilisés, je vous propose un guide résumant les points essentiels devant attirer toute votre attention.

II - La structure du code

Avant de rentrer dans le vif du sujet et de se concentrer sur les données de l'application, arrêtons-nous quelques instants sur l'exemple de code VBA suivant :

```
Private Sub btnBlue_Click()
    txtNom.ForeColor = vbBlue
    txtPrenom.ForeColor = vbBlue
    txtAdresse.ForeColor = vbBlue
    txtVille.ForeColor = vbBlue
    txtCP.ForeColor = vbBlue
End Sub

Private Sub btnRed_Click()
    txtNom.ForeColor = vbRed
    txtPrenom.ForeColor = vbRed
    txtAdresse.ForeColor = vbRed
    txtVille.ForeColor = vbRed
    txtCP.ForeColor = vbRed
End Sub

Private Sub btnGreen_Click()
    txtNom.ForeColor = vbGreen
    txtPrenom.ForeColor = vbGreen
    txtAdresse.ForeColor = vbGreen
    txtVille.ForeColor = vbGreen
    txtCP.ForeColor = vbGreen
End Sub
```

Qui n'a jamais hurlé devant un tel code lorsqu'il était nécessaire de rajouter une nouvelle zone de texte ? Il aurait pourtant été si facile et préférable d'écrire :

```
Private Sub btnBlue_Click()
    Call sub_txtColor(vbBlue)
End Sub

Private Sub btnRed_Click()
    Call sub_txtColor(vbRed)
End Sub

Private Sub btnGreen_Click()
    Call sub_txtColor(vbGreen)
End Sub


Sub sub_txtColor(vColor As Integer)
    txtNom.ForeColor = vColor
    txtPrenom.ForeColor = vColor
    txtAdresse.ForeColor = vColor
    txtVille.ForeColor = vColor
    txtCP.ForeColor = vColor
End Sub
```

Le même principe peut être appliqué aux macros de données. Ainsi, si les évènements **Après Insertion** et **Après MAJ** doivent produire les mêmes traitements, inutile de pratiquer du copier/coller. Créez simplement une macro de données nommée **AprèsInsertetMaj** et lancez la dans chacun des évènements grâce à la méthode **ExécuterMacroDonnées**. De plus, comme sous VBA, il est possible de définir des paramètres pour les sous-macros.

III - Modélisation

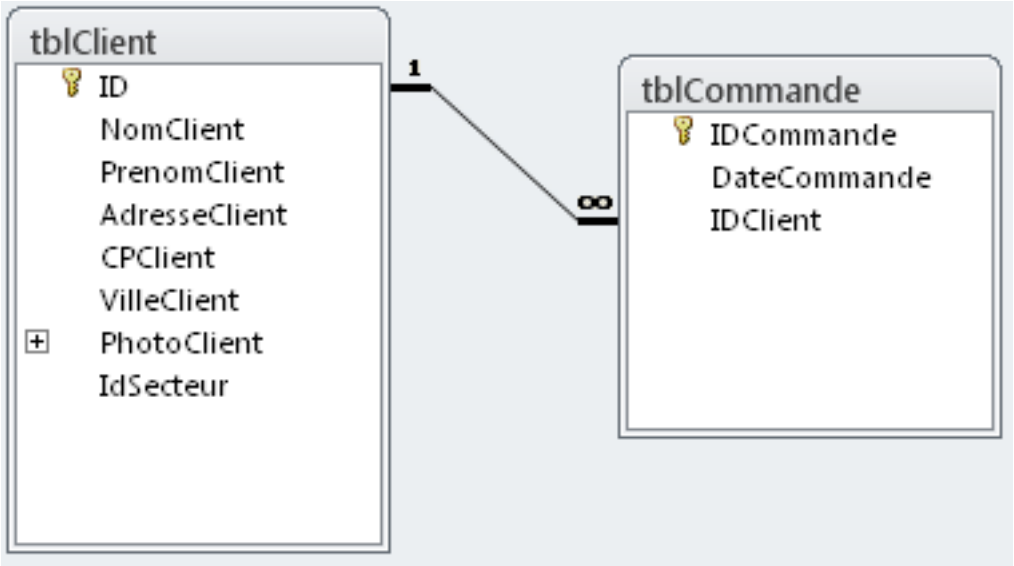
Les évènements de table introduits avec Microsoft Access 2010 permettent de mettre en place la quasi-totalité des traitements qu'il est possible de faire subir à un jeu de données. Ils peuvent être utilisés pour valider la saisie de l'utilisateur (nous verrons cette partie au chapitre suivant) ou bien pour transposer la saisie dans une autre structure en y appliquant des règles de gestion jusque-là confiées à VBA via un formulaire. Comme pour chaque nouveauté Access, le public utilisateur peut être divisé en 3 catégories :

- Les récalcitrants : ils n'utiliseront jamais les évènements de table, persuadés qu'ils sont peu fiables ou inutiles bien qu'ils fassent partie de ceux qui les ont réclamés pendant longtemps.
- Les utilisateurs avertis : ils connaissent la fonctionnalité et savent en tirer profit.
- Les novices : ils viennent de découvrir la nouveauté et ont besoin de parfaire leurs connaissances avant de la mettre en place dans un cadre professionnel.

C'est justement à cette troisième catégorie que ce document, et plus particulièrement ce chapitre, est adressé. En effet, si tous les autres chapitres tiennent plus de l'optimisation et du perfectionnement d'une base, celui-ci et dans une moindre mesure le suivant, seront les garants d'une application stable, robuste, évolutive et maintenable.

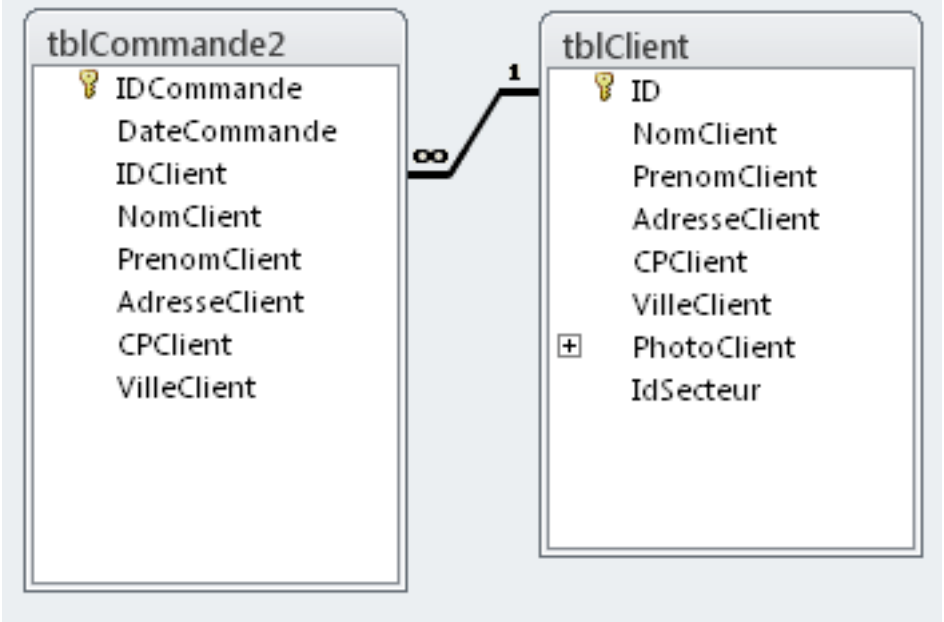
Le principal risque réside dans une dénormalisation de la structure de la base. Celle-ci pourra être « volontaire » (le concepteur a sciemment ignoré les règles) ou « involontaire » (les maintenances successives et le recours systématique aux évènements de table ont conduit à des duplications de structure faisant du projet une véritable usine à gaz)

Prenons l'exemple d'une gestion de commandes :



Cet exemple est le cas d'école le plus courant : des commandes, des clients, les informations du client sont disponibles via la relation des deux tables, etc. Oui mais... Cet exemple n'est pas juste, que va-t-il se passer lorsque le client va changer d'adresse ? **Réponse** : la nouvelle adresse va être utilisée par toutes les commandes y compris celles datant de l'ancienne domiciliation. **Résultat** : des statistiques géographiques fausses, des documents comptables faux, etc.

Partant de ce constat, par simplicité, le rapatriement des données du client dans la table des commandes est réalisé aveuglément et mène au schéma et à la macro de données ci-dessous :




```
/* Recherche des informations du client
  Rechercher un enregistrement dans    tblClient
                   Condition Where    ID=[tblCommande2].[IDClient]
                             Alias    recClient
    DefinirVarLocale   (vNom;[NomClient])
    DefinirVarLocale   (vPrenom;[PrenomClient])
    DefinirVarLocale   (vAdresse;[AdresseClient])
    DefinirVarLocale   (vVille;[VilleClient])
    DefinirVarLocale   (vCP;[CPClient])

/* Edition de la commande
  ModifierEnregistrement
                             Alias
    DefinirChamp   (NomClient;vNom)
    DefinirChamp   (PrenomClient;vPrenom)
    DefinirChamp   (AdresseClient;vAdresse)
    DefinirChamp   (VilleClient;vVille)
    DefinirChamp   (CPClient;vCP)
  Terminer ModifierEnregistrement
```

D'une part, la redondance des champs **NomClient** et **PrenomClient** consomme inutilement de l'espace de stockage et d'autre part, un oeil extérieur pourrait rapidement conclure que la table **tblClient** est finalement inutile, menant un jour à sa suppression puis un peu plus tard à sa restauration mais avec une conception inversée :

Les commandes sont saisies dans la structure **tblCommande**, puis si le client n'existe pas, il sera ajouté à la table **tblClient**.

```
/*  Vérifie si le client existe dans la table tblClient

  DefinirVarLocale   (vTrouve;False)
  Rechercher un enregistrement dans   tblClient
                   Condition Where   [NomClient]=[tblCommande3].[NomClient]
                             Alias   recClient

  DefinirVarLocale   (vTrouve;Not IsNull([NomClient])

  Si Not [vTrouve]  Alors

     Creer un enregistrement dans     tblClient
                             Alias   recNewClient

          DefinirChamp   (NomClient;[tblCommande3].[NomClient])
          DefinirChamp   (PrenomClient;[tblCommande3].[PrenomClient])
          DefinirChamp   (AdresseClient;[tblCommande3].[AdresseClient])
```

```
            DefinirChamp   (VilleClient;[tblCommande3].[VilleClient])
            DefinirChamp   (CPClient;[tblCommande3].[CPClient])
    Fin Si
```

Comme indiqué ci-dessus, quelques lignes seulement de code très basiques ont permis de dénormaliser complètement la structure de la base de données. A de rares expressions près, il est important de considérer que les évènements de table doivent intervenir au sein d'un modèle de données normalisé. Les autres cas consisteront en une solution de remplacement de règles de gestion complexes. Un exemple parmi tant d'autres : la gestion d'un stock FIFO (Premier entré, premier sorti). Il est plus facile de décrémenter à chaque sortie les entrées correspondantes plutôt que d'établir une formule récursive lorsque l'utilisateur souhaite valoriser son stock.

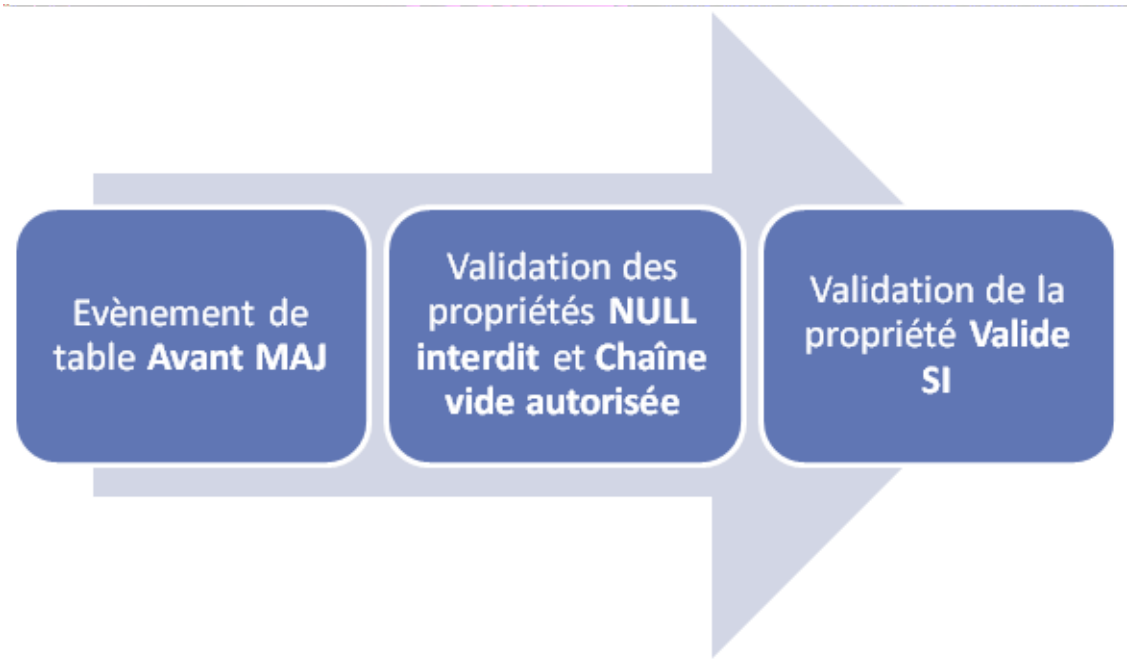
IV - Validation des données

Les évènements de table **Before *** (**Avant Suppression** et **Avant Modification**) permettent de vérifier que les nouvelles données insérées dans la table respectent des critères établis. Leur architecture se déroule autour d'une gestion d'erreur : dès qu'une règle de gestion n'est pas respectée, l'action **DéclencheErreur** permet d'avertir l'utilisateur et de stopper la mise à jour des données.

Avant toute utilisation des évènements de table afin de valider des données, il est primordial d'une part de connaître l'ensemble des propriétés de contrôles déjà disponibles et d'autre part de comprendre comment est effectuée la validation lors de mise à jour en masse.

L'ordre des différents tests effectués par le moteur de base de données est important afin de déterminer si certains d'entre-eux sont superflus. Par exemple, si la vérification par évènement de table intervient après la vérification de la règle **Null Interdit**, il sera possible de considérer dans toute la macro que la valeur du champ est renseignée et donc, par conséquent, qu'un recours à **Nz** ou **IsNull** sera inutile.

Malheureusement, ce n'est pas le cas, les évènements de table interviennent en amont de tout autre traitement. Dans le cas d'une mise à jour, le processus de vérification peut être schématisé de la façon suivante :



A la grande question : "Faut-il gérer les règles **Valide Si**, **Null interdit** et **Chaîne vide autorisée** dans la macro d'évènement de table ?", difficile de répondre. Au regard du peu de ressources nécessaires, j'aurais tendance à "doublonner" les vérifications. (Par exemple : écarter les **NULL** dans la macro tout en fixant la propriété **Null Interdit** à **Oui**) Toutefois, quelle que soit votre décision, il est important que celle-ci reste la même au sein de tout le projet et que ces règles de validation soient suffisamment détaillées dans la documentation afin d'éviter deux versions différentes de tests au sein de deux propriétés d'une même table.

Passé cette question, il est nécessaire de garder à l'esprit, qu'entreront dans le processus de validation, des données que l'on aurait pu croire refoulées par les autres propriétés (qui interviennent hélas trop tard).

Autre point important : la portée de l'erreur. Celle-ci n'a pas vraiment d'incidence lorsque les mises à jour sont réalisées ligne à ligne. Toutefois, lorsque c'est une requête **action** qui est à l'origine du processus, cette notion devient capitale. Pour mettre en évidence nos propos, prenons l'exemple d'une table **tblHistorique** dont le contenu ne doit pas dépasser 10 lignes.

Structure de la table **tblHistorique**:

Nom du champ	Type	Extra
IDLigne	Numéro Auto	Clé primaire
IDEvenement	Numérique	
DateEvenement	Date	

Code de la macro de données **Avant Modification** :

```
Si [Insertion] Alors
  Rechercher un enregistrement dans  qryCountHistorique
                    Condition Where
                              Alias  recHisto
  Si [NB]>=10 Alors
        DeclencherErreur
                  Numero de l'erreur  10001
           Description de l'erreur  Taille maximale atteinte
  Fin Si
Fin Si
```

Requête qryCountHistorique:

```
SELECT Count(*) AS NB
FROM tblHistorique;
```

Constituons un jeu d'essai :

IDLigne	IDEvenement	DateEvenement
1	1	21/11/2009 16:14:40
2	3	21/11/2009 16:14:44
3	2	21/11/2009 16:14:46
4	1	21/11/2009 16:14:47
5	1	21/11/2009 16:14:40
6	3	21/11/2009 16:14:44
7	2	21/11/2009 16:14:46
8	1	21/11/2009 16:14:47
9	3	22/11/2009 16:14:00
10	3	22/11/2009 16:14:04

L'insertion d'une nouvelle ligne directement depuis la table provoque l'affichage du message suivant :

Supprimons maintenant les deux derniers enregistrements et tentons d'exécuter la requête ci-dessous :

```
INSERT INTO tblHistorique ( IDEvenement, DateEvenement )
SELECT IDEvenement, DateEvenement
FROM tblHistorique;
```

Il s'agit simplement de dupliquer les enregistrements de la table **tblHistorique** déjà présents. Trois hypothèses peuvent être envisagées :

- Seulement deux enregistrements vont être copiés, les autres ne seront pas validés, la table aura atteint sa taille maximale.
- Tous les enregistrements seront copiés, la règle de validation n'ayant lieu qu'au début de la transaction.
- Aucun enregistrement ne sera copié, la règle de validation ayant lieu au début de la transaction.

C'est cette troisième hypothèse qui est utilisée par le moteur de base de données. On parle alors de **contrainte au niveau de la transaction**. Si la règle n'est pas respectée par un seul des échantillons de la transaction, l'ensemble des actions de celle-ci est invalidé. Ajoutons que si l'ordre d'exécution est lancé depuis la méthode **Execute** d'un objet DAO, l'erreur levée ne possède pas le numéro qui a été défini dans la macro mais un numéro standard pour toutes les erreurs de validation : **3939**. (La description de l'erreur est quant à elle conservée).

Difficile de parler de véritable limitation. Cela en devient véritablement une si le développeur méconnaît ces mécanismes mais s'il adapte ses projets en conséquence, il ne devrait pas rencontrer de problèmes majeurs.

V - Identification des données

Dans un précédent article (**Création d'une numérotation personnalisée**), nous avons vu comment tirer avantage des méthodes des macros de données afin de réaliser une séquence pour la numérotation de factures. La question principale que se posent plusieurs lecteurs : que se passe-t-il en cas d'accès concurrent ?

Pour illustrer les mécanismes mis en jeu, plaçons-nous dans un environnement multi-utilisateurs avec les éléments suivants :

- Deux utilisateurs : A et B
- Une table tblFacture(NumFacture,IDClient,DateFacture)
- NumFacture est clé primaire et calculée par un évènement de table **Après Insertion** que nous ne développerons pas ici.

Voici le contenu de la table **tblFacture** à t=0

NumFacture	IDClient	DateFacture
FA20091001	1	12/10/2009
FA20091002	2	13/10/2009

A t=1, les deux utilisateurs créent une nouvelle facture. En partant du principe que l'utilisateur A est à l'origine du document daté du 14/10/2009 et l'utilisateur B à l'origine du document daté du 15/10/2009, voici le contenu de la table **tblFacture** disponible pour chaque utilisateur,

Utilisateur A :

NumFacture	IDClient	DateFacture
FA20091001	1	12/10/2009
FA20091002	2	13/10/2009
A calculer	*2*	*14/10/2009*

Utilisateur B :

NumFacture	IDClient	DateFacture
FA20091001	1	12/10/2009
FA20091002	2	13/10/2009
A calculer	*1*	*15/10/2009*

A cet instant, chacun des deux utilisateurs pense créer la facture FA20091003. Cette erreur présente-elle un problème ? Oui, si le champ **NumFacture** n'est pas indexé sans doublon car deux factures portant le même identifiant comptable seront créées, ce qui, je ne vous le cache pas est un cauchemar pour les comptables qui devront justifier de cet incident. En revanche, si le champ est correctement indexé, une erreur sera levée pour l'utilisateur appliquant ses modifications en dernier. Bien évidemment, le risque est minime mais il doit toutefois alerter sur les dangers du temps de traitement des évènements de table. Plus celui-ci augmente pour un utilisateur, plus le risque que d'autres exécutent des actions sur le même jeu de données est important.

VI - La Récursivité

La récursivité est la notion qui intervient lorsqu'un évènement de table exécute une action conduisant à un nouveau déclenchement de cet évènement, que celui-ci s'exécute sur le même enregistrement ou sur un autre.

VI-A - Récursivité non désirée

Pour illustrer ce terme, prenons l'exemple d'un professeur paranoïaque souhaitant stocker la date de mise à jour de ses notes dans sa base de données afin de détecter la moindre tentative de fraude.

Voici la structure de sa table **tblResultat** :

Nom du champ	Type	Extra
IDNote	Numéro Auto	Clé primaire
IDEleve	Numérique	Relié à la table tblElève
IDDevoir	Numérique	Relié à la table tblDevoir
Note	Numérique	
DateMaj	Date	

La valeur du champ **DateMaj** est modifiée via une macro de données **MacroMaj** exécutée sur les évènements **Après Insertion** et **Après MAJ.**

```
ModifierEnregistrement
  DefinirChamp(DateMaj;Now())
Terminer ModifierEnregistrement
```

A l'utilisation, notre professeur ne constate aucun dysfonctionnement. La table est mise à jour correctement à chaque insertion et à chaque mise à jour.

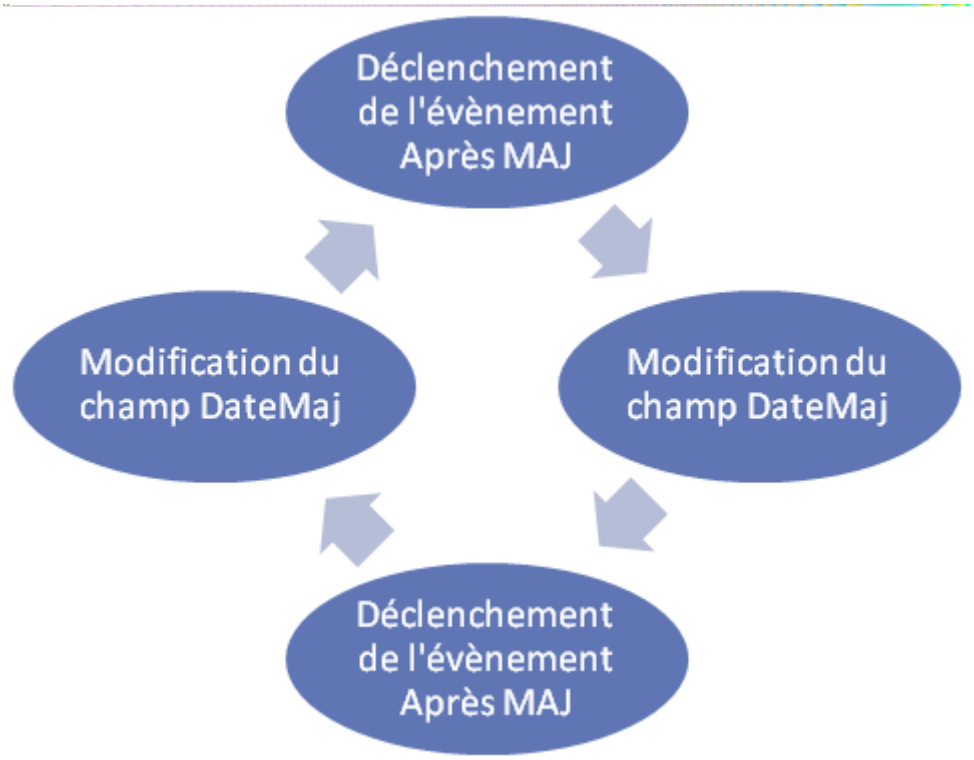
IDNote	IDEleve	IDDevoir	Note	DateMAJ
1	12	1	10	19/11/2009 14:11:50
2	13	1	11	19/11/2009 14:12:13
3	14	1	0	19/11/2009 14:12:24
4	15	1	20	19/11/2009 14:12:27

Cependant, la visualisation de la table **USysApplicationLog**permet de mettre en évidenceun défaut majeur :

Category	Context	Created	Description	ErrorNumber
Execution	EditRecord	19/11/2009 14:12:14	Une limite de ressources a été atteinte pour les macros de données.	-20341
Execution	EditRecord	19/11/2009 14:12:24	Une limite de ressources a été atteinte pour les	-20341

			macros de données.	
Execution	EditRecord	19/11/2009 14:12:28	Une limite de ressources a été atteinte pour les macros de données.	-20341

Ceci est dû à une récursivité dans l'évènement **Après MAJ :** la modification d'un enregistrement entraîne la modification d'un de ses champs (DateMAJ). Ceci constitue une nouvelle modification de l'enregistrement, ce qui entraine une nouvelle modification d'un de ses champs (DateMAJ), ce qui ...



Cette erreur a deux répercussions néfastes :

- La table **USysApplicationLog** grossit inutilement
- Les opérations d'écritures sont plus lentes

Bien entendu, il est possible d'inhiber la boucle ainsi générée. Pour cela, il suffit de réaliser un test sur l'évènement **Après MAJ** afin de savoir si le champ modifié nécessite ou non de d'exécuter le reste de la macro. Le code devient alors :

```
Si Updated("Note")
  ExécuterMacroDonnées(tblResultat.MacroMaj)
Fin Si
```

La différence en termes de délai d'écriture est assez impressionnante puisque la récursivité illustrée plus haut est deux fois plus longue que l'opération « sécurisée ».

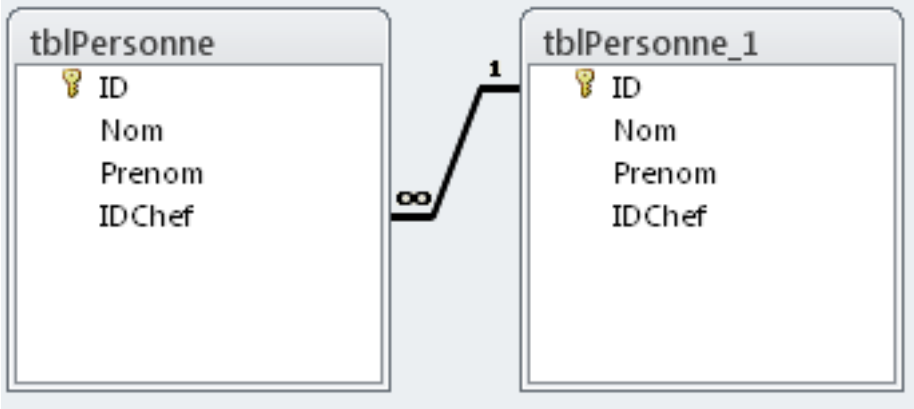
Pour information, voici la moyenne des résultats obtenus pour 1000 insertions via un **recordset** :

- Avec récursivité inutile : 0,84 secondes

- Sans récursivité : 0,43 secondes

VI-B - Récursivité désirée

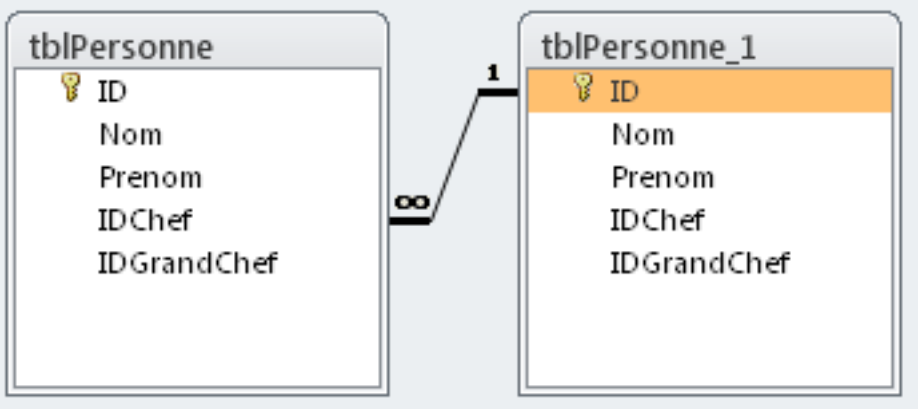
Partant du constat que le moteur semble capable à lui seul de gérer la récursivité des traitements, le développeur peut alors tenter d'essayer d'en tirer profit et plus particulièrement dans la gestion de données hiérarchiques. En effet, sous Access, il est très difficile de mettre au point des requêtes permettant de reconstituer une arborescence telle que répondant au schéma suivant :



Suivant les besoins, il est alors parfois nécessaire de dénormaliser le modèle de façon à simplifier l'expression des requêtes. Dans une structure comme celle ci-dessus, il est par exemple assez périlleux de connaître le supérieur hiérarchique le plus gradé d'un employé étant donné qu'il s'agit de :

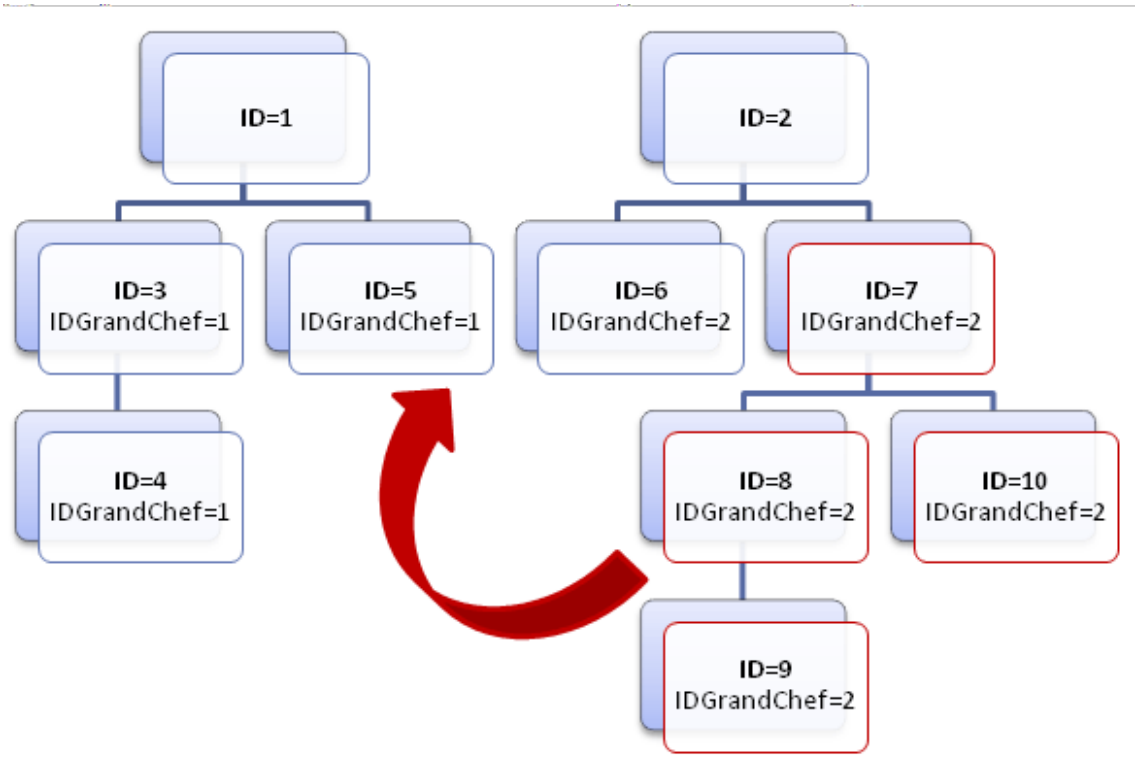
ID->IDCHEF->IDCHEF->IDCHEF ... ->IDCHEF

En revanche avec le modèle ci-dessous, la tâche est grandement facilitée :

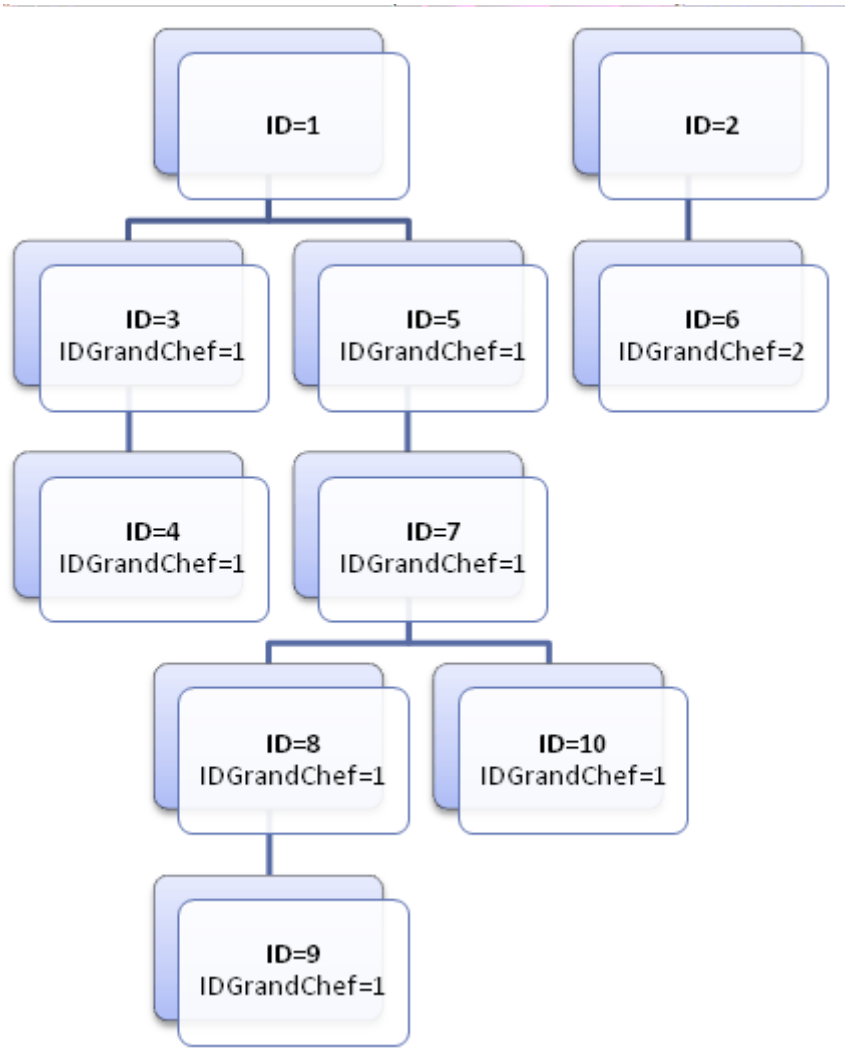


La seule difficulté sera de répercuter la modification d'un noeud dans les noeuds enfants de telle sorte que ceux-ci possèdent bien le même **IDGrandChef** que leur supérieur direct.

Avant :

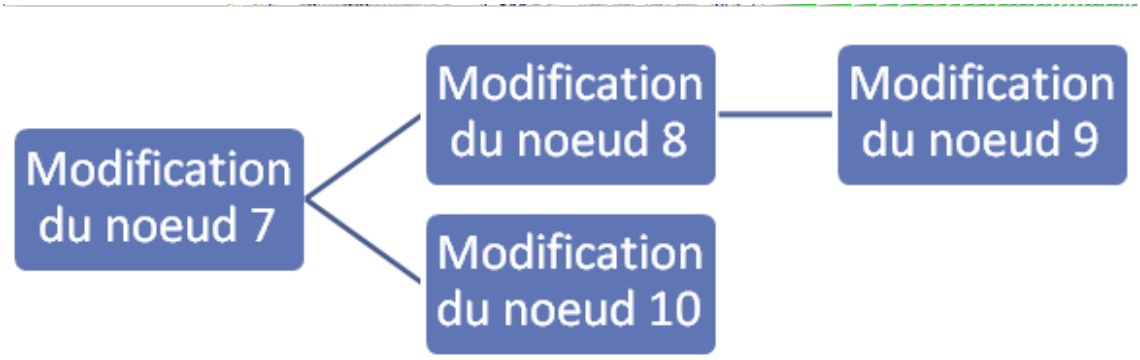


Après :



La tentation est grande de confier un tel traitement aux évènements de table via une opération récursive de mise à jour consistant en la phrase suivante : **Pour chaque noeud enfant du noeud en cours de modification, mettre le noeud à jour.**

Soit ici :



Dans la pratique, ce n'est pas si simple. En effet, pour éviter une boucle infinie, l'équipe de développement de Microsoft Access a restreint la récursivité à un nombre de saut maximal au-delà duquel une erreur est déclenchée et conservée dans la table **USysApplicationLog :**

Une limite de ressources a été atteinte pour les macros de données. Cela peut être dû à une macro de données faisant appel à elle-même de façon récursive. Vous pouvez utiliser la fonction Mise à jour() pour détecter les champs mis à jour dans un enregistrement afin d'éviter les appels récursifs.

Cette limite était de 9 sauts dans la version **Technical Preview** de Microsoft Access 2010 et semble avoir été rabaissée pour des raisons de performances dans la version Beta. Actuellement, seuls 5 nouveaux déclenchements d'un même évènement sont autorisés. Si vous souhaitez mettre en évidence ce comportement, le plus simple consiste à créer une table **matable** possédant un champ **Num** de type numérique et dont la macro de données **Après Insertion** se chargera d'ajouter un nouvel enregistrement :

```
Creer un enregistrement dans MaTable
                Alias RecMaTable
  DefinirChamp(Num;[MaTable].[Num]+1)
```

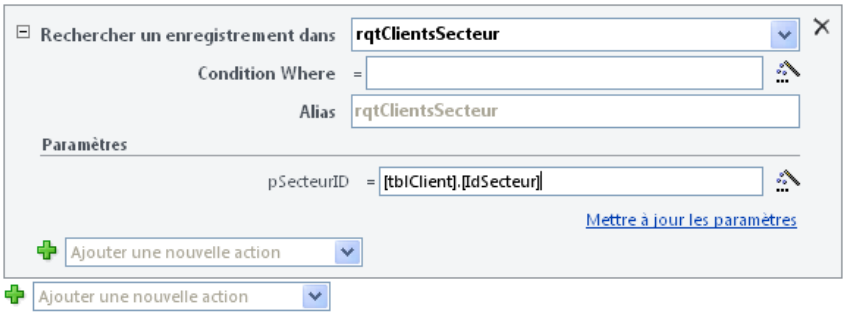
L'insertion d'un nouvel enregistrement dans la table **MaTable** conduira à la création automatique d'un maximum de 5 enregistrements.

VII - Parcours d'enregistrements

Plusieurs méthodes sont disponibles pour rechercher, compter, parcourir des enregistrements. A ce sujet, les développeurs se posent sans cesse cette question essentielle : "Est-il préférable de restreindre le jeu d'enregistrements par une requête ou bien faut-il utiliser uniquement les actions du catalogue des macros de données ?"
Sur des calculs simples, difficile de voir une réelle différence, qui plus est quand on sait que pour une dizaine de milliers de lignes, le parcours d'une table via la méthode **PourChaqueEnregistrement** est quasi instantané. Il faudra vraiment que la table atteigne une taille assez conséquente pour constater un changement sensible.

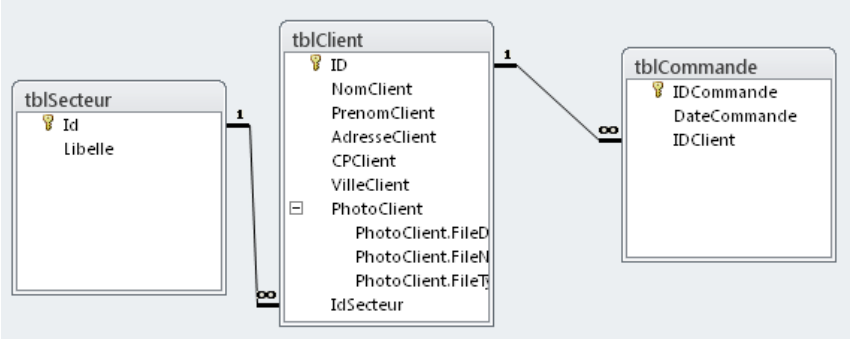
Prenons l'exemple d'un comptage. Si vous êtes certains que le jeu d'enregistrements concerné sera de taille modérée, le passage par une requête peut être superflu. Il m'a fallu un peu plus de 0,12 secondes pour compter 10000 lignes. En revanche, si le jeu est plus conséquent, la boucle devient complètement inefficace avec, par exemple, un traitement de 8,12 secondes pour parcourir 100000 lignes alors que l'appel d'une requête SELECT Count est instantané.

Ajoutons au passage que la programmabilité des requêtes paramétrées offre un avantage indéniable à l'utilisation systématique de requêtes dans les évènements **Après X**(*cette fonctionnalité n'est pas disponible pour les évènements* ***Avant X***)




```
PARAMETERS pSecteurID Long;
SELECT Count(*) AS NBCLIENT
FROM tblClient
WHERE IDSecteur=pSecteurID;
```

Malheureusement, il n'est pas possible d'être si catégorique et de recommander une façon systématique de procéder. Il s'agit en fait bien souvent de comparer un mode de programmation ensembliste (SQL) avec un mode de programmation procédural (parcours de recordset). Tout comme parfois en VBA l'utilisation d'un recordset est préférable, il en est de même dans les macros de données avec une boucle **PourChaqueEnregistrement**. Il est en effet inutile de mettre en relation les milliers de commandes avec le millier de clients d'une centaine de secteurs si seulement les clients d'un secteur particulier sont concernés.



Il est plus intéressant, à priori, d'isoler dans un premier temps les clients du secteur et de parcourir pour chacun d'eux leurs commandes.

Comme il n'existe pas de règle stricte, seule l'application de tests vous permettra de vous familiariser rapidement avec l'optimisation de l'accès aux données via les macros de données. La façon la plus simple de chronométrer une exécution réside dans le module suivant (que vous pouvez adapter et enrichir, bien entendu) :

```
Public debut
Function setdebut() As Boolean
debut = Timer
End Function
Function setfin() As Boolean
MsgBox Timer - debut
End Function
```

Il suffit alors de programmer la macro autour du squelette ci-dessous :

```
DEBUT DE MACRO
  DefinirVarLocale(vTemp;setdebut())
     ...Actions à chronométrer...
  DefinirVarLocale(vTemp;setfin())
FIN DE MACRO
```

Les résultats peuvent être parfois assez déstabilisants, tant au profit d'une grosse jointure que d'une boucle complexe. Nul doute que cela constitue un bon enseignement.

En ce qui concerne les tests de non-existence d'une donnée, plusieurs algorithmes peuvent être utilisés :

- Utiliser une requête de comptage (SELECT Count(*)) possédant une clause PARAMETERS. Les paramètres de recherche sont alors définis dans la macro de données (incompatible avec les évènements **Before**)
- Utiliser une variable locale booléenne.

Cette deuxième méthode ayant le mérite de fonctionner avec n'importe quel évènement, il s'agit de celle que nous allons détailler. Elle répond globalement à l'algorithme suivant :

```
Variable Trouve=Faux
Recherche de l'enregistrement
   Variable Trouve=Vrai
Fin Recherche
Si Trouve=Faux Alors
   --Actions à exécuter si l'enregistrement n'est pas trouvé--
Fin Si
```

Soit, par exemple, en langage macro de données :

```
DefinirVarLocale    (vTrouve;False)
  Rechercher un enregistrement dans    tblClient
                   Condition Where     [NomClient]=[tblCommande3].[NomClient]
                             Alias     recClient

  DefinirVarLocale    (vTrouve;Not IsNull([NomClient])

  Si Not [vTrouve]  Alors

     Creer un enregistrement dans      tblClient
                             Alias     recNewClient

          DefinirChamp   (NomClient;[tblCommande3].[NomClient])
          DefinirChamp   (PrenomClient;[tblCommande3].[PrenomClient])
          DefinirChamp   (AdresseClient;[tblCommande3].[AdresseClient])
          DefinirChamp   (VilleClient;[tblCommande3].[VilleClient])
          DefinirChamp   (CPClient;[tblCommande3].[CPClient])
  Fin Si
```

Bien évidemment, les actions à exécuter si l'enregistrement existe n'utiliseront pas cette structure mais seront directement « posées » dans le bloc **Recherche**.

```
DefinirVarLocale    (vTrouve;False)
  Rechercher un enregistrement dans   tblClient
                    Condition Where   [NomClient]=[tblCommande3].[NomClient]
                              Alias   recClient

     DeclencherErreur(10002;Le client existe déjà)

  DefinirVarLocale    (vTrouve;Not IsNull([NomClient])

  Si Not [vTrouve]  Alors
    ...
```

VIII - Code VBA

Contrairement à ce que certains ont pu écrire, il est tout à fait possible d'utiliser du code VBA dans une macro de données. Il suffit pour cela d'utiliser l'action **DéfinirVarLocal** en définissant sa valeur comme étant égale au retour d'une fonction VBA publique.

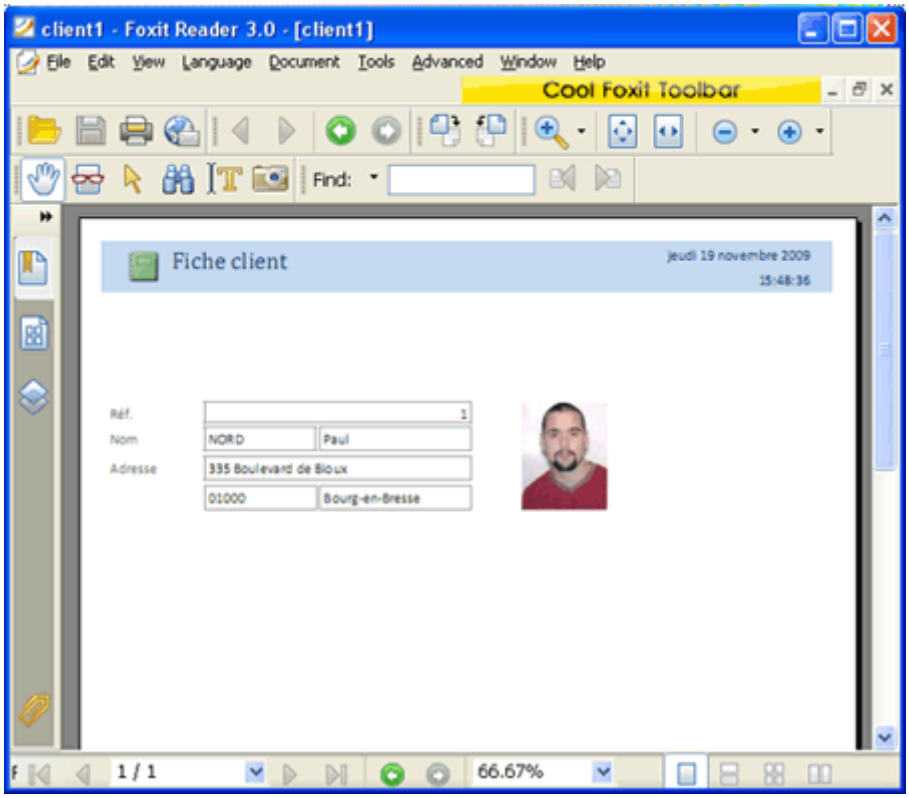
L'exemple ci-dessous permet de sauvegarder les nouvelles fiches client au format PDF en les ouvrant dans un état.

Code VBA :

```
Function GenerationPDF(IDClient As Integer) As Boolean
On Error GoTo err
    DoCmd.OpenReport "rptClient", acViewPreview, , "[ID]=" & IDClient, acHidden
    DoCmd.OutputTo acOutputReport, "rptclient", "PDF", "c:\client" & IDClient & ".pdf"
    DoCmd.Close acReport, "rptclient"
    GenerationPDF = True
err:
End Function
```

Macro Après Insertion :

```
DefinirVarLocale(vImpression ;GenerationPDF([ID]))
```



Toutefois, ce n'est pas parce que l'utilisation du VBA est rendue possible qu'elle est souhaitable et recommandable. Premièrement, j'entends déjà les plaintes de personnes ayant développé des fonctions VBA dans la base de données dorsale et ne comprenant pas pourquoi elles ne s'exécutent pas. Ce n'est pas parce qu'Access intègre une nouvelle fonctionnalité jusque-là réservée aux SGBD client/serveur qu'il en devient un SGBD client/serveur. Pour vous en convaincre, placez votre fichier sur un serveur dépourvu d'Access (par exemple Linux), liez-y l'application frontale et testez une macro de données d'envoi d'email. Aucun doute, cela fonctionne. Les macros de données sont donc exécutées par le poste client, c'est donc sur ce poste que doivent se trouver les fonctions VBA appelées par les macros de données et tout appel à une fonction VBA présente dans la base de données dorsale échouera.

Deuxièmement, il ne faut pas oublier que les évènements de table sont là pour garantir l'intégrité des données lorsque celles-ci sont saisies en dehors de l'applicatif que vous avez développé. Cela peut être via une saisie directe dans les tables, via une autre base Access voire même depuis une application totalement différente de l'univers Access (un script VBS par exemple). Or dans ce dernier cas, les fonctions VBA présentes dans votre fichier accdb ne seront pas chargeables, par conséquent, les traitements censés garantir la cohérence de vos données seront donc inutilisables. La cohérence ne sera donc pas garantie et on peut donc se demander l'intérêt de mettre en place une telle stratégie.

Enfin, si pour une quelconque raison vous persistez à vouloir utiliser des fonctions VBA dans vos macros de données, prenez garde à l'état de vos données aux cours des évènements, comme indiqué dans le chapitre concernant les clés étrangères. Tant que l'ensemble des macros de données affectant une opération ne sont pas terminées, la transaction de données n'est pas finie. Cela signifie, en d'autres termes, qu'en mode multi-utilisateurs, plus les évènements prendront de temps, plus le risque qu'un utilisateur voit une donnée « périmée » est accru. VBA permettant une forte interaction homme-machine, le temps d'exécution a tendance à augmenter considérablement du fait que certaines actions attendent parfois la saisie de l'utilisateur ou la réaction du système.

Pour toutes ces raisons, il est fortement déconseillé d'utiliser du code VBA dans les évènements de tables hormis pendant la phase de développement où le recours à certaines fonctions (msgbox, debug, timer) permet de déboguer et d'optimiser certaines macros.

IX - Conclusion

Certains des points abordés ci-dessus vous paraissent peut être inutiles dans le cadre de l'application que vous développez actuellement. Toutefois, il est nécessaire de garder à l'esprit que les macros de données introduisent une nouvelle façon de programmer sous Microsoft Access, à mi-chemin entre les requêtes SQL et le code VBA. Bien que disposant d'une syntaxe totalement différente et bien plus limité, ce nouveau langage est assez proche au niveau du concept de PL SQL sous Oracle : un langage procédural autour d'une programmation ensembliste.