PLAN QUALITE
pour les applications écrites avec DELPHI
Release 1.24

© 1997 - Olivier Dahan
Email : odahan@cybercable.fr
Web : http://perso.cybercable.fr/stargate/

> ***NOTE 7-2000*** *Bien que nécessitant une mise à jour globale, cet article présente de nombreux concepts toujours valides. Les remarques positives des lecteurs me font penser qu'il peut être encore diffuser sous cette forme. Mais n'oubliez pas en lisant que tout cela s'inscrit dans un context qui a 3 ans et plus pour certains passages et que certains anachronismes sont possibles… merci par avance de votre indulgence.*

Présentation

Le présent document à pour but de collecter une série de « conseils » de développement. Il s'adresse donc en priorité à une population professionnelle ayant déjà de solides notions de programmation.
Je n'ai pas eu ici la prétention d'édicter une « norme » au sens académique du terme, ce qui serait à la fois présomptueux et hors de toute réalité. Ce document n'est qu'une suite d'avis et conseils, parfois commentés longuement car ce qui prime est la justification des choix, et, souvent, des digressions se mêleront aux informations techniques. Une application professionnelle ne se conçoit pas à coup de « recettes », c'est avant tout un esprit, une rigueur, une attention de chaque instant, un attachement au détail, une certaine noblesse. La même que celle de l'artisan qui corrige les moindres erreurs pourtant invisibles à l'oeil du profane. Comme pour un bel objet façonné, l'utilisateur d'un logiciel ne sait pas dire ce qui a été fait techniquement et qui créé son attachement à ce dernier, il l'aime et l'utilise. C'est tout. ... Mais pour en arriver à un tel niveau de satisfaction il faut souvent beaucoup de travail dont les détails échappent et échapperont toujours à l'oeil de celui qui n'est pas initié. Un artisan est un magicien, il transforme l'inanimé en objet de plaisir et de satisfaction, soyez-en un aussi ...

Les conseils proposés ici sont les fruits de réflexions longues, de nombreuses expériences, ou du simple bon sens. Ils contiendront parfois des interdictions voire des obligations. Pour d'autres rubriques il s'agira plutôt de faire des choix entre diverses possibilités. Comme il faut bien se décider j'y propose alors une solution qui a l'avantage, si elle est suivie, d'offrir cohérence et homogénéité aux développements.

Tout ce que le lecteur trouvera ici doit être vu sous l'angle d'un travail en perpétuelle évolution. Cela implique qu'il a le pouvoir (le devoir ?) de proposer ses critiques, ses solutions. Toutefois, comme un tel document tire une partie de sa valeur de son « unicité », il conviendra de me faire directement ces propositions afin que je puisse maintenir à jour le « Plan Qualité » et qu'à une date donnée il n'y ait toujours qu'une seule version « officielle » en circulation.

Les conseils ici prodigués le seront sous l'angle du développement DELPHI. L'existence de disparités entre la version 16 bits et la version 32 bits imposent aussi un nivellement sur une base commune. Ne seront donc pas abordés ici les Threads, le Repository, les DataModules, l'héritage de Tform et autres techniques propres à

l'environnement 32 bits. Une notice complémentaire sera conçu pour les développements utilisant DELPHI 32 (2, 3, 4, 5) et les techniques qui lui sont propres. L'ouverture sous Linux imposera une refonte…

I. Ergonomie Générale

Il est en fait difficile de trancher sur ce qui est du pur ressort de l'ergonomie et de ce qui ne l'est pas. Un bon logiciel n'est qu'une suite de bonnes décisions qui toutes influencent, directement ou non, l'ergonomie générale du produit. Toutefois nous nous limiterons dans ce chapitre à ce qui concerne directement l'interface.

A. Les messages d'aide

DELPHI propose une gestion assez souple des messages d'aide notamment avec les bulles d'aide (HINTS). Il est très important d'utiliser ce système qui peut souvent éviter la création d'un système d'aide contextuel plus lourd à gérer. Il convient dès lors de bien connaître le fonctionnement de ces HINTS et des astuces qui y sont relatives.

1. Les bulles d'aide

La norme est assez floue à propos des bulles qui ne font pas partie du fonctionnement de base de Windows. Toutefois on peut affirmer que les textes doivent être très courts : une bulle d'aide ne doit pas être un pavé d'aide !
Le mot « bulle », lui-même, implique une certaine légèreté, ne l'oubliez pas...
On préférera des bulles ne contenant qu'un seul mot ou expression avec le minimum d'articles et de mots de liaison : QUITTER, MISE A JOUR, et non « Quitter l'application », « Lancer la mise à jour ». Ces formes plus littéraires ralentissent le travail de l'œil…

Si la traduction française de HINT (bulle) insiste sur la légèreté, le sens original de HINT insiste lui sur le caractère fugace de l'information donnée et sur sa fonction première qui est de donner une indication et non un cours complet.
Pour information :
« HINT (1) Allusion, To drop a hint = faire une allusion *(à noter « drop », action rapide en général, nda)*,
(2) Conseil (piece of advise : morceau de conseil...), (3) Soupçon... »

Une bulle, un soupçon, un morceau de conseil... voici bien résumé ce que doit être une bulle d'aide.
Proscrivez l'aspect amateur et antinomique des bulles multi-lignes (sauf exceptions que nous aborderons plus loin).

Court et précis ne signifie pas utiliser le style télégraphique ou une sibylline notation hiéroglyphique. L'ergonomie est un métier à part entière, des gens y consacrent leur profession, des chercheurs leur vie. Si tout un chacun peut aujourd'hui développer sous Windows, cette démocratisation s'accompagne (comme toujours) d'un nivellement par le bas et d'un affaiblissement de la compréhension des concepts sous-jacents.
IBM et Microsoft ont publié leurs normes, il faut les lire et s'en inspirer, ou au moins en connaître l'existence et le sens. Et ce, quoi qu'on pense de ces sociétés et de l'admiration ou de l'inimitié qu'on ressent pour leurs dirigeants. Un professionnel sert un but, la qualité, ses sentiments personnels, même justifiés, ne sont que secondaires.

Le choix d'un texte pour une bulle est une démarche de même nature que celle du choix et du dessin d'une icône, d'une musique d'accompagnement, d'une séquence vidéo. Le caractère multimédia de Windows impose des compétences artistiques à tout concepteur.
Ainsi, choisir un mot ou une expression qui résumera bien une fonction est une tâche parfois difficile qui réclame des qualités qui ne sont pas celles d'un informaticien. L'humilité, en partie, consiste à connaître ses limites. Si vous ne trouvez pas, faites dans la simplicité et l'évidence. S'il faut être un informaticien ou un nouveau Champollion pour décrypter vos bulles, mieux vaux ne pas en mettre du tout...

Dans certains cas rares, il se peut que vous ne trouviez pas quelque chose de court et de sensé. Ce n'est pas trop grave, dans ce cas veillez à ce que la bulle ne soit pas aussi large que l'écran. Pour y remédier on peut alors admettre (et uniquement dans ce cas) de placer le texte sur deux lignes, pour « ramasser » l'allure générale de la bulle. L'éditeur de DELPHI ne permet pas de placer des changements de ligne dans le texte d'une bulle. Ce dernier est considéré comme une propriété STRING, le retour chariot met simplement fin à la saisie en la validant.
Cela ne doit pas vous décourager, la fenêtre affichant les bulles utilise correctement les capacités de Windows et prévoit l'affichage sur plusieurs lignes. Pour faire la jonction entre l'éditeur de propriété qui n'autorise pas la saisie du caractère #13 et la classe de fenêtre des bulles qui, elle, le gère, vous avez deux méthodes : Saisir le texte par programmation (`MonObject.Hint := '1ère ligne'+#13+'seconde ligne'`), méthode qui vous poussera à ne pas abuser de la chose, ou bien utiliser un éditeur de propriété HINT spécialisé. Il existe sur le Forum DELPHI un outil Freeware du nom de HINTEDIT qui assure cette fonction à merveille.

Il est important de prévoir, par un paramètre persistant de l'application (donc stocké dans un fichier INI -ou en REGISTRY- en général), la possibilité de déconnecter les bulles d'aide.
Chaque objet TFORM dispose d'une propriété SHOWHINT pour ce faire et tous les objets possédant une propriété HINT ont aussi une propriété PARENTSHOWHINT qu'on laisse à TRUE (sauf cas particuliers).La visibilité globale pour la fiche est contrôlée par TFORM.SHOWHINT.
Pour gérer de façon habile cette option sans intervenir sur chaque fiche, programmez un gestionnaire de l'événement ONACTIVEFORMCHANGE de l'objet SCREEN. Systématiquement vous placerez le SHOWHINT de la nouvelle fiche active à la valeur de la fiche maître ou à celle d'une constante globale contenant le choix de l'utilisateur à propos de la visibilité des bulles.

2. La ligne de statut (status line)

Si cette technique est plus ou moins évidente à mettre en place dans des applications suivant le modèle SDI (à moins qu'il n'y ait qu'une seule fiche), le modèle MDI se prédispose plus facilement à son utilisation. Voyons tout d'abord le cas le plus simple.

a) Ligne de statut en mode MDI

La ligne de statut est un espace généralement placé en bas de la fenêtre mère dans lequel sont affichées des informations complémentaires sur l'objet en cours de sélection (contrôle ou élément de menu). En ce sens, la ligne de statut est une technique proche de la bulle d'aide. Plus simple techniquement que les HINTS (DELPHI vous offre une gestion des bulles d'aide déjà programmée mais cela n'est pas trivial à faire en partant de zéro...), cette information contextuelle précède historiquement ces dernières. Les bulles ne sont d'ailleurs qu'une amélioration de la ligne de statut visant à disposer à l'écran l'information près du contrôle auquel elle se rattache pour renforcer l'ergonomie du procédé.

Une application suivant le modèle MDI possède une fenêtre mère toujours visible, il est donc naturel de placer la ligne de statut en bas de celle-ci. Un TPANEL fera parfaitement l'affaire (aligné ALBOTTOM). Il est possible de mettre en œuvre ce procédé dans le modèle SDI mais comme il n'y a pas de fenêtre mère assurément visible et non masquée, il convient de placer la ligne de statut en bas de chaque fenêtre. La gestion en est plus technique comme nous allons le voir.

La ligne de statut, par la place plus grande qu'elle offre au texte, est l'endroit idéal pour compléter les explications des bulles d'aide. Une application bien conçue possède ainsi 4 niveaux d'aide que sont les labels précédant chaque contrôle, les bulles d'aide, la ligne de statut et enfin l'aide contextuelle.
DELPHI met à votre disposition une gestion presque automatique de la ligne de statut, au prix d'une petite « ruse » qu'il faut connaître.

La propriété HINT de chaque contrôle contient le texte de la bulle d'aide. C'est une chose. Nous avons vu que cette ligne peut être segmentée par l'insertion du caractère #13, c'en est une autre. Mais en ajoutant, à la fin du texte de la bulle, un caractère « pipe » ALT-124, on peut placer, derrière celui-ci, un morceau de texte qui sera celui de la ligne de statut. Il ne doit pas contenir de retour chariot, bien que cela soit possible si le texte est affiché par l'intermédiaire d'un TLABEL en mode WORDWRAP. Toutefois, éternel débat, si cette ligne peut contenir plus de texte qu'une bulle, elle ne doit pas remplacer l'aide contextuelle, plus fournie. Son texte doit rester cours, même si, ici, on peut utiliser une phrase complète.
D'ailleurs, il est bon de noter la granularité sémantique conseillée pour chaque niveau d'aide :

- Label précédant les contrôles et texte des bulles ..... mot ou expression
- Ligne de statut ..... phrase
- Aide contextuelle ..... page

Comme DELPHI ne peut pas deviner le nom de l'objet qui contiendra la ligne de statut (alors qu'il fourni en interne un objet bien identifié pour afficher les bulles) ni même le nom de la propriété de cet objet qui contiendra le texte lui-même (CAPTION, TEXT, ...) et encore moins le procédé que vous retiendrez pour l'affichage (un simple conteneur de type TPANEL ou TLABEL, ou une gestion plus sophistiquée) il est nécessaire, pour afficher une ligne de statut de répondre à un événement précis : ONHINT de la variable APPLICATION.

C'est au sein de ce gestionnaire que vous récupérerez le texte à afficher, fourni par DELPHI dans la propriété HINT de la variable APPLICATION, et que vous le dispatcherez où bon vous semble.
La version la plus simple d'un tel gestionnaire se limite à une affectation du type « `MonPanel.Caption := Application.Hint ;` ».
Vous comprenez maintenant pourquoi le modèle MDI se prête mieux à une telle gestion. L'événement ONHINT étant unique pour toute l'application, il est plus difficile, en SDI, de router le texte vers plusieurs lignes différentes selon la fiche qui est active.

b) Ligne de statut en mode SDI

Pour gérer une ligne de statut sur plusieurs fiches (donc dans le modèle SDI) il existe plusieurs méthodes, l'une d'entre elles a été décrite par Borland dans le TI 2796. Elle consiste à coder une réponse personnalisée, dans chaque TFORM, à l'événement APPLICATION.ONHINT. Chaque fiche aura ainsi son propre gestionnaire qui saura où et comment afficher le texte. Ces gestionnaires sont souvent très simples et identiques (affectation de la variable APPLICATON.HINT au CAPTION d'un TPANEL).

Pour avertir DELPHI qu'une nouvelle fenêtre prend la focalisation et que c'est celle-ci qui doit afficher la ligne de statut, il suffit, dans chaque fiche, de gérer l'événement ONACTIVATE du TFORM qui se limitera à affecter, à la propriété APPLICATION.ONHINT, le nom du gestionnaire local. Ainsi, lorsqu'une fiche devient active, elle modifie le gestionnaire centralisé afin de le faire pointer vers sa propre routine d'affichage.

L'un des nombreux avantages de la ligne de statut est que DELPHI génère toujours l'événement APPLICATON.ONHINT, même si les bulles sont déconnectées (TFORM.SHOWHINT :=FALSE). L'utilisateur aura ainsi le choix de voir ou non les bulles sans que la ligne de statut ne soit affectée par ce paramètre. Cela implique, en retour, que le texte de la bulle et celui de la ligne soient totalement indépendants. La ligne de statut ne peut pas faire référence, en le complétant, au texte de la bulle puisqu'il n'y a aucune assurance que celle-ci soit affichée.

Vous noterez, pour terminer, que le HINT d'un objet peut fort bien ne contenir qu'un message pour la ligne de statut, sans message de bulle. Il suffit de ne mettre aucun texte devant le symbole « pipe », donc de commencer la phrase par celui-ci (pratique pour les éléments de menu n'affichant jamais de bulle)...
Comme nous avons pu le voir ici, DELPHI propose de nombreuses solutions pour aider l'utilisateur, certaines se mettent en œuvre automatiquement, d'autres réclament un peu de code. Dans tous les cas vous devez porter une grande attention à cet aspect de votre application, c'est l'un des points qui différencie une application professionnelle d'un simple assemblage de boutons et de contrôles.

3. Les autres aides

Il est bien entendu possible de compléter tous les systèmes précédemment étudiés par des informations ponctuelles de nature différente et ce, selon le contexte de l'application.
Par exemple, il peut être important pour l'utilisateur, dans une application donnée, d'avoir un indicateur des touches spéciales de type « Verrouillage numérique » ou « Majuscules ». Dans d'autres cas il faudra pouvoir afficher le résultat ou la progression d'actions en tâche de fond, etc...
Il n'y a pas d'autre emplacement (sauf à le justifier) pour toutes ces données que dans l'espace de la ligne de statut. La fin de celle-ci est alors segmentées en « sous-panels » contenant chacun un type d'information précis.
N'oubliez pas, enfin, que l'aide à l'utilisateur n'est complète que si votre application intègre un fichier HLP. La conception de ce dernier dépend grandement de l'outil de conception et il ne nous est pas possible ici de faire le tour de toutes les possibilités. Ce silence apparent ne doit pas vous faire penser que nous n'y attachons pas une grande importance : une application Windows n'est pas terminée tant qu'elle ne dispose pas d'un fichier d'aide contextuel intelligemment intégré et structuré.

B. Dialogues internes

Il est fréquent qu'au sein d'un projet il soit nécessaire de demander à l'utilisateur une chaîne de caractères, éventuellement initialisée. En général le développeur créé une fiche appelée en mode MODAL pour assurer cette fonction. N'oubliez pas que DELPHI met à votre disposition des instructions répondant déjà à ce besoin. Souvent ignorées, elles ont pourtant l'avantage d'être simples et de ne pas réclamer le moindre octet de plus dans votre exécutable. Qui plus est, d'où la présence de cette rubrique dans le chapitre traitant d'ergonomie, cela assurera une homogénéité salutaire à votre application.

Il s'agit des fonctions INPUTBOX et INPUQUERY. La première réclamant une chaîne, la seconde prenant en compte la possibilité d'annulation (boutons OK / ANNULER). Parallèlement, nous insisterons sur le fait qu'il faut toujours utiliser les dialogues standard de Windows partout où cela peut se faire. DELPHI propose des « wrappers » permettant d'accéder à ces dialogues (OpenDialog, SaveDialog, etc).

C. Traitement longs

il est indispensable que l'utilisateur soit averti de la progression d'un traitement qui risque d'être long. Le fait d'inscrire un message du type « Traitement en cours » ne sert à rien du tout. Un traitement en cours peut durer 5 secondes ou 5 heures. Dans ce dernier cas l'utilisateur aura rebooté sa machine bien avant, la croyant bloquée !

Si votre application possède de tels traitements (dépassant une dizaine de secondes ou pouvant, en charge réelle, les dépasser), créez une fiche simple possédant un message d'attente ainsi qu'un barre de progression. Utilisez la technique de la procédure appelante pour cette fiche, technique que vous perfectionnerez pour l'occasion. Nous vous conseillons d'appeler la procédure d'ouverture de cette fiche OPENWAIT(CONST MSG :STRING), MSG étant le message qui sera inscrit. Cette procédure aura pour tâche de créer une instance de la fiche d'attente, d'affecter le message et mettre la barre de progression à zéro puis d'appeler la fiche en mode SHOW non modal (sinon le traitement long en question serait bloqué jusqu'au retour de la fiche ...). La fiche d'attente doit être de type ALWAYSONTOP.
Créez une procédure CLOSEWAIT dont la tâche sera de détruire la fiche d'attente.
Pour mettre à jour la barre de progression créez enfin une procédure
`SENDWAIT(CONST NEWMSG :STRING ; GAUGEVALUE :INTEGER).`
Si le paramètre NEWMSG est vide, le message courant est conservé, sinon il est modifié pour prendre la valeur de NEWMSG. Le paramètre GAUGEVALUE sera une valeur entre 0 et 100 représentant le pourcentage d'avance du traitement.
Cette procédure, après affectation des différents paramètres, se terminera par un appel à APPLICATION.PROCESSMESSAGES afin d'assurer le rafraîchissement de l'image.

D'ailleurs, au sein de votre traitement long, placez au moins un appel à cette procédure afin d'assurer la lecture fréquente de la pile des messages de Windows. Cela est indispensable sous Windows 3.1 pour simuler le multitâche coopératif (justement). Certains développeurs pensent que le multitâche des environnements 32 bits (W95 et NT) rend cet appel caduque. Ils se trompent. Si la commutation de tâche n'est plus assurée par la coopération des tâches entre elles, la lecture de la pile des messages Windows par l'application reste conditionnée à l'appel de PROCESSMESSAGES qui reprend alors tout son sens premier (traiter les messages).

Certains traitement réellement longs doivent toujours pouvoir être interrompus par l'utilisateur. Il est indispensable de prévoir cette possibilité. Cela se fera en ajoutant un bouton « ANNULER » dans la fiche d'attente. L'appui sur ce bouton positionnera une propriété de la fiche d'attente, ANNULATION_DEMANDEE :=TRUE, propriété initialisée à FALSE par la procédure OPENWAIT.
Le traitement long, en plus de faire des appels réguliers à SENDWAIT, testera après ces derniers la valeur de la propriété ANNULATION_DEMANDEE qui sera lisible à travers une fonction ISCANCELREQUESTED :BOOLEAN qui, toujours dans l'esprit d'une isolation totale entre les fiches sera la seule à savoir lire la véritable propriété de la fiche d'attente.
Si aucun appel à PROCESSMESSAGES n'est effectué dans la boucle du traitement long, cette variable ne sera jamais positionnée correctement.
Charge à votre application d'annuler le traitement en cours proprement en rétablissement la situation précédente ou en interdisant certaines opérations tant que le dit traitement n'a pas été relancé.

Si vous devez lancer un traitement réellement long qui ne peut vraiment pas être annulé, en plus de la barre de progression nous vous conseillons d'afficher une estimation du temps restant, à la manière des copies de fichiers de Windows 95 (mais en mieux...). Ce temps s'analyse simplement en fonction du pourcentage d'avance, et du nombre de pas réellement réalisés et de ceux restants. L'affichage devra être effectué sous la forme Heure, Minute, Seconde.

D. La gestion du Presse-papiers

Le presse-papiers est un outil essentiel pour l'utilisateur, il est le mécanisme le plus simple qui illustre la coopération entre diverses applications et reste le symbole même des environnements graphiques comme celui du Mac d'Apple ou Windows. Avant même l'invention du DDE ou de l'OLE, la seule réelle coopération qui faisait la différence entre ces environnements et les autres était justement de pouvoir copier des informations d'une « fenêtre » vers une autre. Les opérations sur le presse-papiers sont ainsi essentielles dans une application professionnelle et elles doivent faire l'objet d'une attention toute particulière.

1. Quand « copier » n'est pas suffisant

DELPHI, pour les contrôles de sa VCL, automatise la gestion du Couper/Copier/Coller. Mais le sens de ces opérations est orienté « contrôle » alors qu'un humain travaille plutôt sur des concepts, ce que, bien entendu, le meilleur des environnements de développement ne sait absolument pas gérer...
Prenons l'exemple d'une fiche individuelle client au sein d'une gestion commerciale. Elle contiendra certainement l'adresse du client. Techniquement celle-ci sera matérialisée selon toute vraisemblance par les contrôles : adresse 1 et 2, code postal et ville.
Les fonctions presse-papiers automatiques de DELPHI concerneront chaque contrôle mais pas la totalité du groupe de contrôles formant l'adresse qui n'est qu'un concept. Si l'utilisateur souhaite « copier » l'adresse pour écrire un courrier avec WORD, la gestion de base sera très lourde (plusieurs va et vient entre WORD et l'application pour copier/coller chacun des contrôles).
Pour offrir un plus grand confort, il est nécessaire de proposer au minimum une fonction « copier » qui copiera toute l'adresse en une seule opération. Ce traitement peut être automatique (selon le contexte de votre application) ou bien être matérialisé par un bouton ou une entrée dans le menu EDITION.
Techniquement, vous devrez piloter l'opération de copie par code en utilisant l'unité CLIPBRD. Pour grouper plusieurs informations dans un même « copier », il suffit de les mettre bout à bout, séparées par un caractère 13_{10} (retour chariot) dans un PCHAR puis de copier ce dernier dans le presse-papiers avec le format texte (voir les identifiants de type de données du presse-papiers).
Cette technique peut largement être exploitée dans vos applications qui n'en seront que plus conviviales.

Parallèlement, il est bon de constater que nous n'avons traité ici qu'un seul sens : le copiage. Pour le collage l'opération est plus complexe, notamment lorsque celle-ci est réalisée entre plusieurs applications différentes.

Par exemple, pour récupérer une suite de ligne copiées depuis WORD dans la série de contrôle formant l'adresse dans votre application, il faudrait découper le buffer du presse-papiers puis appliquer tous les contrôles de saisie avant d'accepter ou refuser le collage (par exemple tester la numéricité de la zone qui devra remplir le contrôle gérant le code postal).

Ces difficultés, assorties de la nécessité de détourner les messages Windows entrant de type « coller » pour les traiter de façon spécifique, font que les applications ne gèrent presque jamais ce type de collage multi-contrôle.

Par contre, il est tout à fait possible d'exploiter le copier/coller multi-élément au sein d'une même application qui, elle, peut reconnaître les constituants du groupe à coller. DELPHI, pour les besoins de son IDE, enregistre un format spécial dans le presse-papiers pour les objets. C'est ce qui permet de copier/coller des composants sur une fiche en mode conception.
Ainsi, votre application peut définir des descendants de TPERSISTENT qui contiennent des informations fortement typées (des propriétés de l'objet) servant aux opérations de presse-papiers complexes entre les divers modules du logiciel (structures dynamiques, objets, ...). Généralement, les opérations de collage acceptant des formats non habituels (texte ou image) font l'objet d'un menu spécifique. Ce dernier est souvent appelé « coller / spécial ».
La possibilité de pouvoir manipuler des objets persistants dans le presse-papiers est une formidable ouverture qui peut simplifier le codage de vos applications tout en améliorant l'ergonomie (par exemple copier une fiche client entière qui peut être récupéré, même partiellement, par tous les modules de l'application). L'utilisateur reste maître des « instants » et des « lieux » et le code, objet, peut être centralisé.

2. La gestion centralisée du presse-papiers

La gestion de des fonctions de base de Windows est assurée, dans DELPHI, par chaque contrôle. Il s'agit des fonctions Couper, Copier et Coller. Toutefois, il est courant d'avoir l'envie de proposer des boutons pour les commander, voire un menu « Edition » afin de se conformer à l'ergonomie standard de Windows.
Si tout bon développeur sait que les boutons doivent être de type TSPEEDBUTTON car ils ne prennent pas le focus, il est fréquent de voir des morceaux de code qui sont soit indigents, et ne fonctionnant pas dans tous les cas, soit trop complexes.
On peut effectivement être tenté de tester le type de contrôle actif (ACTIVECONTROL) pour assurer un traitement personnalisé. Le plus courant est le test « `IF ACTIVECONTROL IS TCUSTOMEDIT THEN TCUSTOMEDIT(ACTIVECONTOL).COPYTOCLIPBOARD` » répété pour les trois commandes à l'identique en dehors de l'ordre final. Cette solution, bien que valide, ne gère qu'une toute petite partie des possibilités des contrôles de DELPHI en ne s'intéressant qu'aux type dérivés de TCUSTOMEDIT (TEDIT et TMEMO). Que se passe-t-il si le contrôle est un TDBIMAGE ? Rien ! Et c'est dommage...
La direction généralement prise par ceux qui se sont aperçus de la faiblesse de la technique ci-dessus évoquée est celle d'une sophistication accrue de la séquence de code gérant chacune des fonctions. Cette voie n'est pas la bonne.
En fait, la gestion du presse-papiers nous donne ici l'occasion de vous conseiller de vous intéresser à l'OS sous lequel tournent vos applications : Windows. En ignorer les concepts fondamentaux, en nier les API est un comportement d'amateur. Sans connaître les moindre méandres de Windows, faites l'effort de mieux l'appréhender car le code que vous écrirez sera à la fois plus court, plus rapide, moins buggé, plus élégant et moins « amateur »...
Ainsi, en s'appuyant sur une connaissance somme toute fort modeste de Windows, comment gérer le problème du presse-papiers ici posé ? En utilisant les messages appropriés, simplement :

```
SendMessage(ActiveControl.Handle, WM_CUT, 0, 0);
```

C'est tout ! La constante du message est la seule chose variable selon la fonction (WM_PASTE et WM_COPY pour les 2 autres possibilités à la place de WM_CUT).
Si votre application est de type MDI, vous utiliserez ACTIVEMDICHILD.ACTIVECONTROL.HANDLE et les trois boutons seront placés dans une « speedbar » en haut de la fiche mère, en copie des commandes du menu

EDITION. Vous compléterez l'instruction par un test contrôlant si la forme MDI active est différente de NIL afin d'éviter tout problème (`IF ACTIVEMDICHILD<>NIL THEN ...`). En fait ce test est redondant si vous avez décidé d'être encore plus soucieux de l'ergonomie de votre application. Dans ce cas, vous désactiverez les boutons concernés sur l'événement SCREEN.ONACTIVECHANGE où vous testerez le MIDCHILDCOUNT pour savoir s'il est supérieur à zéro. Dans la négative les entrées du menu et les boutons gérant les fonctions du presse-papiers sont déconnectées en modifiant leur propriété ENABLED à FALSE.
La technique ici utilisée, celle du simple message Windows, consiste à donner l'ordre au contrôle actif de gérer la fonction désirée. S'il n'en a pas la capacité, le message sera ignoré. Mais dans le cas contraire, il n'y aucun besoin de contrôler le type de l'objet concerné, il saura comment traiter le message.
Répétons une fois encore le conseil de cette rubrique, il est trop important : apprenez à connaître la philosophie de Windows afin d'écrire des applications qui la respecte. Votre code sera plus performant et plus compact.

E. Ergonomie des fiches

Il y a beaucoup de choses à dire sur l'ergonomie des fiches, et pour cause, ce sont, in fine, les parties les plus présentes de l'interface d'une application.
Nous aborderons ici les différents thèmes qui semblent, le plus souvent, être oubliés, mal compris ou dont les mises en œuvre sont trop disparates. Le reste sera laissé à l'appréciation de chacun...

1. Le focus d'entrée

Il n'est pas concevable qu'à son premier affichage une fiche ne positionne pas correctement le focus sur le premier champ ayant le plus d'intérêt. DELPHI met à la disposition du développeur, dans l'objet TFORM, la propriété ACTIVECONTROL. Cette propriété permet, durant l'exécution, de connaître le contrôle ayant le focus et, au premier affichage d'indiquer quel contrôle recevra le focus. C'est ici dans ce dernier sens qu'on l'exploitera systématiquement. N'oubliez pas, non plus, de repositionner le focus dans certains cas. Par exemple, sur une fiche de type Données, lorsque l'utilisateur déclenche (par quelque moyen) la création d'une nouvelle fiche, le focus doit être transféré au premier contrôle de celle-ci automatiquement...

2. La dynamique du focus

Il n'est pas normal que le curseur puisse se « balader » un peu partout et, le pire, dans n'importe quel ordre. Il est essentiel de veiller à l'ordre des tabulations, ce qui peut se faire simplement depuis l'EDI en mode conception.
De même, si certains boutons n'ont pas un intérêt vital dans l'optique d'une saisie manuelle, il n'est pas nécessaire de les inclure dans la chaîne de tabulation (propriété TABSTOP) ce qui est le cas par défaut. Et même dans le cas où leur activation depuis le clavier a un sens, il sera préférable de programmer un accélérateur plutôt que de voir le curseur, lors d'une saisie, être obligé de faire le tour de tous les boutons pour revenir au premier contrôle (un accélérateur de bouton se programme en ajoutant le symbole « & » devant une lettre de son texte). Bien entendu, dans certains cas, il sera préférable de laisser les boutons dans le circuit de tabulation. Répétons-le encore une fois : ce document s'adresse à des professionnels qui doivent toujours utiliser leur bon sens, seule instance capable, *in fine* et pour une application donnée, de trancher...

3. Les grilles et le focus

Si une grille (DBGRID, STRINGGRID ou autre) est placée sur une fiche et si elle ne sert pas directement à la saisie, il est indispensable de positionner sa propriété

DGTABS à FALSE. Ainsi, la touche de tabulation permet de sortir de la grille au lieu que le curseur reste coincé à l'intérieur de celle-ci passant de champ en champ.

4. Elaborer un circuit de saisie

Toute fiche se doit de proposer un circuit de tabulation rapide suivant la logique et l'ordre de saisie des informations. Les contrôles n'entrant pas directement dans le processus de saisie ne doit appartenir au circuit défini.

a) Simulation de la touche Tabulation

Il est souvent nécessaire, lorsque les utilisateurs migrent depuis des environnements « console »vers Windows, de proposer une équivalence entre la touche ENTREE et la touche TABULATION afin de mimer la logique du mode console. De plus, les claviers ont été conçu pour que la touche ENTREE soit naturellement à porté des doigts, sa grande taille en atteste. Ce n'est pas le cas de la touche de tabulation.
Pour ces diverses raisons (et d'autres) on peut admettre, dans la majorité des applications, de proposer une équivalence ENTREE = TABULATION. N'oubliez toutefois pas que cela ne respecte pas du tout l'ergonomie de Windows et que, dans certains cas, cette équivalence risque de poser des problèmes cornéliens.
La mise en œuvre de ce procédé se fait en basculant la propriété KEYPREVIEW de l'objet TFORM à TRUE et en programmant un gestionnaire pour l'événement ONKEYPRESS :

```
if Key = #13 then
  begin
    Key := #0;
    PostMessage(Handle, WM_NEXTDLGCTL, 0, 0);
  end;
```

Vous noterez, ici encore, que nous n'utilisons pas d'instruction VCL pour accomplir cette tâche, mais, à la place, une API Windows directe. Lorsque cela est possible, rappelez-vous que poster un message est toujours une solution de meilleure qualité que tout autre bricolage plus ou moins savant avec la VCL. Ici nous postons le message WM_NEXTDLGCTL (Next Dialog Control = Prochain contrôle du dialogue).

b) Mise à jour automatique des données

Si une fiche utilise des données passant par le BDE, il est important de prévoir un mécanisme clair et évident pour leur validation, notamment lors de la fermeture de la fiche. Le problème s'étend souvent aux changements de page lorsque la fiche propose un étalement des données sur plusieurs pages d'un NOTEBOOK (à onglet ou non). Selon si la validation des données entraîne ou non des conséquences d'importance, vous opterez pour l'une ou l'autre des méthodes suivantes :

(1) Validation automatique

Normalement, la fermeture de tous les DATASET est regroupée dans l'événement ONDESTROY des objets TFORM. Juste avant la fermeture de chacun de ces DATASET vous contrôlerez leur propriété STATE (`IF STATE IN [DSEDIT,DSINSERT]`) et effectuerez un POST (ou un CANCEL suivant la stratégie adoptée).

(2) Validation semi-automatique

Lorsque la validation de certaines données est lourde de conséquences il est normal de s'assurer que l'utilisateur souhaite réellement abandonner la fiche (ou l'application) sans validation explicite de sa part.

Il existe un événement de TFORM pour gérer de telles situations : ONCLOSEQUERY dont le but est de vous donner la main avant que la demande de fermeture de l'utilisateur n'aboutisse réellement. Dans le gestionnaire de cet événement vous utiliserez un dialogue (par exemple la fonction MESSAGEBOX de l'objet APPLICATION) pour demander une confirmation d'abandon. Dans l'affirme vous n'avez rien à faire. Dans la négative vous avez l'option soit de faire un POST sur chaque table en cours de modification, soit d'interdire la sortie de l'application (ou de la FORM). Les deux méthodes se valent et seul le contexte vous permettra de trancher (sans oublier votre bon sens).
La méthode totalement silencieuse consiste à refuser la sortie tant que des tables sont en modifications et ce, sans message particulier. Cela ne peut s'appliquer qu'à des fiches sur lesquelles la visualisation de l'état des tables et évident.

F. Grisé ou invisible ?

L'une des questions qui rongent le développeur est celle du choix Grisé ou Invisible pour les boutons, items de menus et autres éléments d'interface. Rappelons quelques règles de bon sens en accord avec l'esprit de Windows.
Un élément d'interface est grisé (ENABLED=FALSE) lorsqu'il est indisponible mais reste tout de même visible. Lorsqu'il est invisible, l'utilisateur ne peut pas se poser la question de savoir s'il est disponible ou non.
Qu'il est bon d'enfoncer les portes ouvertes...
On en conclue que :

- L'invisibilité (VISIBLE=FALSE) supprime une fonction que l'utilisateur n'a pas même à connaître. L'invisibilité ne donne aucune information, au contraire elle soustrait du champ visuel une fonction inutile.
- L'indisponibilité (ENABLED=FALSE) déconnecte une fonction de façon temporaire car le contexte l'impose. Cette indisponibilité **visible** renseigne l'utilisateur sur l'état de l'application.

De là, on peut dire que :

- Si une fonction n'est pas disponible pour des **raisons externes** (liées à l'utilisateur, à l'état d'un périphérique, etc), les éléments d'interface la représentant doivent être **grisés** (ENABLED=FALSE). C'est le cas d'un bouton « Calculer » si aucune formule n'a été saisie, d'une entrée de menu « Coller » si le presse papier est vide, d'un bouton de fermeture d'un écran de détail si celui-ci n'a pas été ouvert, etc.
- Si une fonction n'est pas disponible pour des **raisons internes** (fonction non autorisée pour le profil de l'utilisateur, fonction déconnectées car n'ayant pas de sens dans le contexte, etc), les éléments d'interface la représentant doivent être **invisibles** (VISIBLE=FALSE). C'est le cas d'un bouton « Création » sur une fiche permettant la création mais utilisée dans un contexte où cette création n'est pas autorisée, d'un item de menu non accessible à l'utilisateur car celui-ci n'a pas les droits d'accès nécessaires, etc...

II. La gestion des données

Nous pourrions utiliser ici la même introduction que celle du chapitre sur l'ergonomie. En effet, il est très difficile de séparer le traitement des données, leur stockage, leur utilisation du reste d'une application. Rappelons la formule de Wirth : Algorithme + Structure de données = Programme. Toutefois nous nous limiterons dans le présent chapitre à tout ce qui concerne les données stockées sur support permanent, qu'il s'agisse de fichiers « texte » ou de tables d'une base de données SQL.

A. Améliorer les performances du BDE

Afin que vos applications fonctionnent au plus vite il est nécessaire de bien configurer le BDE (si vous utilisez des tables, bien entendu). Vous noterez que certains des conseils ci-dessous sont génériques alors que d'autres peuvent variés selon la base cible choisie. Le plus important est ici de porter l'attention sur certains paramètres plutôt que d'offrir un « plan de réglage » universel.
Voici quelques conseils importants :

1. Maintenez toujours un nombre d'index secondaires très bas et rappelez-vous qu'il est souvent plus rapide de créer un index à l'exécution le temps de son utilisation puis de le détruire plutôt que de surcharger en permanence l'application avec des index surnuméraires.
2. Si possible, augmentez la taille du buffer du BDE ainsi que le nombre de HANDLES qui lui sont attribués (programme de configuration du BDE). N'oubliez, surtout sous Windows 3.1 n'augmenter aussi le nombre de HANDLES de votre applications (API Windows SETHANDLECOUNT).
3. Quand cela est possible, ouvrez les tables en mode exclusif (non partagé).
4. Regroupez, si possible, le maximum d'opérations, notamment par l'utilisation de la fonction BDE DBIBATCHMOVE (un composant BATCHMOVE existe dans la palette de DELPHI). Si vous traitez directement les données, pensez à utilisez les API du BDE telles que DBIWRITEBLOCK et DBIREADBLOCK et essayez de travailler sur des buffers de taille multiple du buffer du BDE (c'est à dire 2 ou 4ko en général).
5. Si vous êtes amené à ouvrir et fermer de façon répétitive une ou plusieurs tables, songez à utiliser l'API du BDE DBIACQPERSISTTABLELOCK en l'appliquant, une première fois, sur un fichier non existant. Cela forcera le BDE à créer le fichier LCK qui ne sera pas créé puis détruit à chaque fermeture de table. Cela ne s'applique bien entendu qu'aux données de type PARADOX. N'oubliez pas d'appeler en fin de traitement l'ordre de relâchement du fichier LCK par l'API BDE DBIRELPERSISTTABLELOCK.

B. Maintenance des tables

Le menu Administration (voir la rubrique concernant cet aspect) de votre application devra contenir une option de régénération des index et de compactage des données si elle travaille sur des tables dBase ou Paradox.
Pour Rafraîchir les index d'une table, placez les unités DBITYPES, DBIPROCS et DBIERRS (l'unité BDE des versions récentes remplace ces trois unités) dans la clause USES du module puis appelez l'API du BDE `DBIRegenIndexes(Table1.Handle);` La table doit être ouverte en mode exclusif et les index doivent effectivement exister.
Pour compacter les données, utilisez l'appel suivant (dans les mêmes conditions) :

```
Check(DbiPackTable(Table1.DbHandle, Table1.Handle, Nil,
szPARADOX, TRUE));
```

C. Connaître le nom réseau de l'utilisateur

Plusieurs méthodes existent, selon qu'on souhaite obtenir ce nom depuis l'OS ou le BDE. L'accès à l'OS est très spécifique et nécessite des séquences de code éventuellement différentes suivant la version de Windows et, éventuellement, selon le driver réseau.
Pour utiliser la fonction interne du BDE, et après avoir placé DBIPROCS, DBIERRS, DBITYPES dans la clause USES de votre module :

```
function ID: String ;
var
```

```
 rslt: DBIResult ;
 szErrMsg: DBIMSG ;
 pszUserName: PChar ;
begin
 try
   Result := '' ;
   pszUserName := nil ;
   GetMem(pszUserName, SizeOf(Char) * DBIMAXXBUSERNAMELEN) ;
   rslt := DbiGetNetUserName(pszUserName);
   if rslt = DBIERR_NONE then
    Result := StrPas(pszUserName)
   else
    begin
     DbiGetErrorString(rslt, szErrMsg);
     raise Exception.Create(StrPas(szErrMsg));
    end ;
   FreeMem(pszUserName, SizeOf(Char) * DBIMAXXBUSERNAMELEN) ;
   pszUserName := nil ;
 except
   on E: EOutOfMemory do ShowMessage('ID failed. ' + E.Message);
   on E: Exception do ShowMessage(E.Message);
 end ;
 if pszUserName <> nil then FreeMem(pszUserName, SizeOf(Char) *
DBIMAXXBUSERNAMELEN) ;
end ;
```

D. La gestion des tables

Toute application gérant des données via le BDE se doit d'avoir un objet TDATABASE pour chaque base de données ouverte. Le TDATABASE doit être relié à l'ALIAS BDE et propose à l'application un ALIAS local auquel seront connectés tous les TDATASET de l'application. Le TDATABASE est placé sur la fiche mère (modèle MDI) ou sur la fiche principale (modèle SDI). C'est dans le gestionnaire d'événement ONCREATE de cette fiche que la connexion est généralement ouverte. En tout cas, il ne faut jamais laisser la connexion ouverte depuis le mode design de l'IDE.

Parallèlement, toute fiche manipulant des données doit effectuer les ouvertures soit au moment du besoin (mais attention au temps de connexion), soit en répondant à l'événement ONCREATE de l'objet fiche TFORM. La fermeture doit être effectuée en respectant l'ordre des dépendances (généralement l'inverse des ouvertures pour éviter des resynchronisations inutiles) en répondant à l'événement ONDESTROY de l'objet fiche. En aucun cas les tables ne doivent être ouvertes en mode design et laissées dans cet état.

E. Gérer simplement des sous-requêtes SQL

Il est parfois nécessaire d'extraire ou de traiter des données à partir du résultat d'une autre requête. Si vous utilisez des SGBD-R puissants (voire même Local Interbase), la syntaxe de ces moteurs supporte les sous-requêtes. Par contre, le BDE ne supporte pas ces dernières sur les données locales. La méthode la plus efficace et la plus simple consiste en l'appel d'une procédure de l'API du BDE qui rend persistante une requête et ce, sous un nom de fichier choisi par vous.
L'exemple suivant vous éclairera :

```
with GeneralQuery do
```

```
begin
  SQL.Clear;
  SQL.Add(.... inner SQL);
  SQL.Open;
  DbiMakePermanent(handle, 'temp.db',true);
  SQL.Clear;
  SQL.Add(SELECT  ... From 'temp.db'....)
  SQL.Open;
end;
```

Le seul problème à gérer est celui de la collision des noms si votre application fonctionne en mode réseau. Il vous faudra aussi détruire le fichier temporaire une fois le traitement terminé. Pour cette tâche utilisez un TQUERY avec un ordre DELETE TABLE, le BDE se débrouillant pour faire le ménage notamment lorsque les données sont de type PARADOX (où le fichier principale est souvent accompagné de très nombreux fichiers auxiliaires - index, checks...). Cette solution a aussi l'avantage d'être portable sans modification.

III. Mise en œuvre

La mise en œuvre (traduction de IMPLEMENTATION) c'est le « passage à l'acte », la transposition dans le monde de l'action d'une élaboration de la pensée.
Pour être fidèle à cette pensée, à son esprit, la mise en œuvre doit elle-même être intégrée à un processus de développement de pensée plus vaste. La cohérence est à ce prix. Trop souvent on s'imagine qu'il y a d'un côté des penseurs (en informatique on dirait des analystes) et de l'autres des ouvriers spécialisés (des programmeurs).

C'est oublier que mettre en œuvre c'est réaliser et que Réalisateur est un métier noble à part entière qui ne peut exister sans une philosophie, sans une méthodologie propre.

D'ailleurs, ce n'est pas pour rien si, dans le milieu du cinéma, la promotion est faite presque exclusivement avec le nom du Réalisateur, celui de l'auteur étant en tout petit quelque part en bas de l'affiche.

Il n'y a pas d'OS (ouvrier spécialisé…) dans le cinéma, ou bien pour les petits bouleaux annexes (aller chercher les sandwiches ou perchman) tous les métiers sont uniques et réclament un haut niveau de savoir-faire : du maquilleur aux effets spéciaux, il n'y a pas d'ouvriers spécialisés mais des spécialistes (nuance...).

L'informatique, dans sa mise en œuvre, est bien plus proche des métiers de l'esprit que de l'industrie. D'ailleurs, les logiciels sont protégés en France par les mêmes lois que les oeuvres des auteurs et compositeurs et non pas celles des concepteurs de moteurs ou d'immeubles.

L'erreur de beaucoup de sociétés est de vouloir industrialiser l'informatique en la traitant comme on traite la construction d'un stade ou la fabrication en série de savonnettes parfumées.

Si on veut industrialiser le logiciel il faut le faire dans le respect de sa spécificité qui est celle d'une œuvre de l'esprit et s'inspirer des méthodes industrielles appliquées à la production musicale ou cinématographique.

Résoudre une analyse informatique au plan d'un architecte, résoudre la programmation au métier de maçon, et aussi nobles que soient ces métiers qui méritent respect, est un fantasme d'argentier, de spéculateur et de boursicoteur n'ayant rien compris à l'essence même de l'informatique.

Il est donc essentiel de comprendre qu'il ne peut y avoir de recettes ni de « pensée unique » lorsqu'on aborde la mise en œuvre du code d'une application.

C'est un acte tout aussi pensé, tout aussi sophistiqué que celui de la conception. Il se place sur un autre registre, c'est tout. Pour terminer avec les métaphores cinématographiques, disons que conception et réalisation sont deux caméras qui prennent la même scène (l'application) sous deux angles de vues différents avec des objectifs différents.

Cette longue introduction n'a pas forcément grand rapport avec les rubriques du présent chapitre qui sont, par force ici, très orientées vers les aspects bas niveau. Mais justement, elle était nécessaire pour bien replacer le code qui suivra dans son vrai contexte qui n'est pas technique mais philosophique.

A. La lisibilité du code

Le fait de pouvoir facilement lire (et comprendre !) le code source qu'on a écrit il y a longtemps ou écrit par d'autres est essentiel.
Il est tellement important que nous avons même déjà vu des appels d'offres pour des développements en C accompagnés de restrictions d'écriture qui transformaient presque le C en Pascal !

Aujourd'hui, Object Pascal comporte des astuces de notation et des ajouts importants qui l'éloignent de sa pureté originelle. Certain mots clés sont même ambigus, ce qui est en contradiction avec l'esprit même du Pascal (DEFAULT, par exemple, défini soit la valeur par défaut d'une PROPERY, soit une PROPERTY de type ARRAY par défaut pour l'objet, tout dépend du « point virgule » qui le précède ou non...).

De fait, il convient d'adopter une conduite claire face à une syntaxe qui devient permissive et (trop ?) subtile.

Le premier conseil sera ici ne pas tomber dans le piège que les psychologues nomment « la pensée magique ».
Pour éviter un débat psychanalytique qui n'aurait pas sa place ici donnons immédiatement un exemple en rapport avec le sujet : Le C (par exemple et sans « lutte de classe » - jeu de mot très fin vous en conviendrez) est souvent utilisé par des gens victimes du syndrome de la pensée magique qui consiste à croire que puisque le code qu'on écrit est « hyper » compact et « ultra » sibyllin, le programme qui sera généré par le compilateur et l'éditeur de lien sera, par force, tout aussi efficace et compact.

Grave erreur... L'efficacité en programmation n'a rien à voir avec l'aspect de ce qu'on tape mais avec l'influence des mots qu'on utilise et de l'ordre de ceux-ci. Ces points peuvent même varier d'une version à l'autre d'un même compilateur et sont loin d'être toujours commentés. Ainsi, remplacer le très clair « BEGIN END » par « { } « en C, n'influence en rien le code généré.

Prenons l'instruction CASE comme exemple. Jusqu'au BP7 l'ordre le meilleur des constantes de cas pour un code plus efficace devait être celui de l'ordre décroissant de la fréquence d'utilisation. Le cas le plus fréquent étant en premier, le moins fréquent en dernier.
Cela était lié au code machine généré par le PASCAL (une suite de comparaisons et de « JUMP » de cas en cas, du premier au dernier).
Le PASCAL de DELPHI génère un code très différent, et, ici, le moyen d'optimiser l'efficacité est de placer les constantes de cas dans l'ordre numérique croissant (qui

est celui de l'ordre de la déclaration dans le type si le sélecteur de CASE est de type énuméré).

Donc, inutile d'écrire du code ésotérique et difficilement lisible uniquement dans le vain espoir que le code généré sera meilleur. Il ne faut pas confondre exercice de style et programmation efficace.
Toute subtilité d'écriture devra se justifier par une argumentation technique sans faille (voir le cas du WITH étudié plus haut). En dehors de ces cas parfaitement identifiés, aucune « astuce » ne sera acceptable dans un logiciel professionnel qui doit être maintenu, donc relu et compris.

Dans le même sens, on pourra étendre encore plus le champ du premier conseil : entre efficacité de code et lisibilité, et sauf cas critique ou la vitesse de traitement est réellement cruciale, on choisira toujours la lisibilité du code.

De cet conseil étendu on pourra par exemple affirmer que :
Bien que jugé avec mépris par les « Pascaliens », l'ordre GOTO (et LABEL) peut et doit être utilisé en place et lieu de structure IF THEN ELSE imbriquées si le niveau d'imbrication rend la compréhension de l'algorithme difficile.

Bien que nous ayons démontré la puissance de l'instruction WITH, il sera parfois plus clair de répéter un nom de variable si les sous-champs de celle-ci ont trop d'homonymes avec les variables de la portée et / ou que ces dernières doivent être exploitées à l'intérieur du WITH.
Concernant les objets, il est préférable d'écrire : « MonObjet.Var1 := Var1 » que « WITH MonObjet do Var1 :=Self.Var1 ».
Bien entendu il faut éviter, tant que faire se peut, de déclarer des variables portant des noms entraînant ce type de problème.

Par exemple, on tentera de ne jamais utiliser de variables ayant le même nom que certaines propriétés d'objet. Une variable FONT entrera immédiatement en conflit au sein d'un WITH portant sur un objet visible de la VCL (possédant une propriété FONT en général). On choisira quelque chose comme FONTLocale ou TmpFONT, ou tout autre préfixe ou suffixe permettant de créer une différence.

B. Le choix du modèle, la stratégie mono-instance.

Windows offre deux stratégies : le modèle MDI[1] et le modèle SDI[2]. La norme édictée par Microsoft est très claire quant au choix à faire. Si votre application manipule des fenêtres ayant une structure différente (données de structures différentes), le modèle SDI s'impose.
Si votre application utilise plusieurs instances du même type de fenêtre (même « cadre » avec des données différentes mais de même structure) alors il faut choisir le modèle MDI.

Pour rappel voici quelques exemples : Le gestionnaire de fichier de Windows 3.1 (WinFile encore présent sous W95) est de type MDI, on peut ouvrir plusieurs vues sur différents lecteurs mais la structure de chaque fenêtre est rigoureusement identique. L'EDI de DELPHI est un exemple du modèle SDI : une fenêtre mère (le bandeau supérieur avec la palette) est accompagnée de plusieurs fenêtres filles de nature très différente (Inspecteur d'objets, éditeur de source, ...).

[1] MDI : Multiple Document Interface
[2] SDI : Single Document Interface

Dans le modèle MDI, les fiches filles sont posées sur l'espace de la fenêtre mère qui simule ainsi un « bureau local ». Dans le modèle SDI, les fenêtres filles sont posées sur le bureau de Windows.

En dehors des différences cosmétiques entre ces deux modèles il est important de noter que le choix n'est pas innocent.
En effet, lorsqu'on utilise le modèle MDI, seule la fenêtre mère peut créer des filles. Ainsi, si une fille doit déboucher sur l'affichage d'une autre fille, elle sera obligée d'appeler une commande de la fenêtre mère pour le faire, la nouvelle fiche étant alors une « sœur ».
Le modèle SDI autorise lui la filiation multiple et toute fiche peut être la mère (propriétaire) de toute autre fiche. Cette simple nuance entre les deux modèles impose le choix du SDI lorsque des fenêtres filles doivent être reliées entre elles par un lien de subordination fonctionnelle (de type synchronisation, par exemple). La synchronisation d'une fenêtre « liste » et d'une fenêtre « détail » est très délicat à gérer avec le modèle MDI absolument pas étudié pour ce type de développement.

Malgré tout, et bien que l'industrie informatique ait payé fort cher la pression des utilisateurs l'ayant obligé à adopter le standard Windows, il convient de constater que la très grande majorité de ces mêmes utilisateurs n'a pas investi un temps suffisant à la compréhension de Windows, que cela soit volontaire ou par contrainte du milieu.

De fait, bien peu de ces gens savent utiliser la combinaison ALT-TAB pour passer d'une application à l'autre, par exemple. Pire, nous avons tous en mémoire des exemples d'applications lancées 10, 20 fois, voire plus, sur le même poste.
Un simple clic sur le Gestionnaire de Programme de Windows 3.1 ou sur une autre application fait disparaître celle qui avait la focalisation.

L'utilisateur de Windows réfléchi comme un nouveau-né (ce qui se comprend puisqu'il aborde une nouvelle réalité): tout objet quittant son champ de vision n'existe plus dans la réalité.
Pour cette raison, nombreux sont les développeurs prudents qui préfèrent adopter le modèle MDI en format maximisé. Bien que généralement en totale contradiction avec la norme et avec les besoins techniques de l'application, ce choix se justifie ici par la simplification de l'interface : un seul espace, celui de le fenêtre mère de l'application, contient toutes les fenêtres filles. La mère occupant tout l'espace écran, les chances de cliquer sur une autre application deviennent proches de zéro. En tout état de cause, MDI ou SDI, une application doit absolument comporter un mécanisme interdisant plusieurs instances (sauf si cette possibilité est justifiée par un besoin fonctionnel qui a été clairement décidé avec le client).

N'oubliez pas qu'un contrôle mono-instance doit pouvoir aujourd'hui fonctionner aussi bien sous Windows 3.1 que sous Windows 95 ou NT. De ce fait, même en DELPHI 1 il ne faut plus utiliser le pointeur PrevInstance disponible en 16 bits qui renvoie toujours NIL sous Windows 32 bits (chaque application ayant son propre environnement et non plus un environnement commun permettant de voir les autres instances). Il existe toutefois une technique portable fonctionnant pour toutes les combinaisons (compilateurs 16/32 et environnements 16/32). Elle consiste a chercher la fenêtre principale de l'application (FINDWINDOW) et à annuler l'exécution en cours après avoir redonné l'avant-plan à la première si elle existe déjà.
Attention, l'activation par un simple SHOW n'est pas suffisante, il faut exploiter le mode RESTORE pour être certain que l'instance appelée passe réellement en avant-plan.

Pour conclure le conseil de cette rubrique sera le suivant : Si votre application est utilisée par une population assez grande et mal formée, préférez le modèle MDI, si

votre application contient beaucoup de liens de subordination fonctionnelle entre plusieurs fenêtres, préférer le modèle SDI. Bien entendu de très nombreuses applications mélanges ces deux impératifs. A vous de faire un choix intelligent...

C. Traitements génériques et variables globales

Les diverses fiches qui constituent une application ont souvent besoin de partager du code, des types, des constantes ou des variables globales. On constate que les développeurs placent souvent ces globales au sein même des objets TFORM, ou, au mieux, dans la fiche principale, parfois dans le corps de plusieurs fiches. Le code est ainsi émaillé de références croisées ce qui comporte des risques certains notamment pour la maintenance.

S'il est possible de faire référence à une propriété ou un champ d'un objet fiche depuis un autre objet, cette technique doit être proscrite sauf lorsqu'elle repose sur un fondement méthodologique, comme la délégation par exemple. Il y a de nombreuses bonnes raison à cela, et s'il n'est pas question ici d'entrer dans la théorie, on retiendra de la programmation Objet le principe d'Encapsulation qui impose qu'un objet ne doit pas avoir à connaître la structure d'un autre objet en dehors des propriétés et méthodes publiques, sauf cas exceptionnels bien cernés.

Certaines structures de données sont parfois volumineuses et, sous Windows, la taille du segment de données étant très petite, une bonne habitude consiste à « dynamiser » ces variables. Comme une application DELPHI bien écrite ne comporte généralement que des fenêtres dynamiques, il semble logique de placer ces variables au sein des objets TFORM. Cette vision simpliste a les travers que nous avons énoncés plus haut. Les variables globales de grandes taille doivent être gérées de façon dynamique, mais selon un processus propre et de façon isolée. Elles ne doivent en aucun cas dépendre de l'existence d'une fiche ou d'une autre, ce qui, énoncé ainsi, démontre bien qu'il ne s'agit alors plus de Globales mais de Locales partagées...

Toute application qui doit gérer des objets (au sens le plus large) partagés, se doit de posséder d'une unité de code (UNIT) dédiée à cette mission. Nous vous conseillons de créer une unité portant le nom COMMON. Les types, les variables, les constantes, les objets, les procédures et fonctions devant être partagés se trouveront déclarés dans la partie INTERFACE de cette unité. La partie IMPLEMENTATION ne contenant que le code des éventuels procédures, fonctions et objets, ce qui implique que, dans certains projets, cette section pourra être totalement vide.

Toute fiche devant accéder aux données partagées fera référence à l'unité commune dans la clause USES de sa partie IMPLEMENTATION. Toutefois, ce type de déclaration se justifie par le seul fait qu'à cet endroit DELPHI ne gère par les références croisées. Certaines Globales peuvent toutefois être utilisées au sein du code de certaines fiches (des TYPES par exemple). Dès lors, et seulement pour l'unité COMMON qui ne doit poser, par sa structure, aucun problème de référence croisée, il est possible d'en faire la déclaration dans la clause USES de la partie INTERFACE de l'unité en ayant besoin.

1. L'initialisation des Globales

En dehors des constantes typées, le seul moyen d'être sûr du contenu d'une variable, par défaut, est de l'initialiser par code. Où et quand initialiser les Globales de l'unité COMMON ? Certains développeurs place ce code dans le gestionnaire de l'événement ONCREATE de la fiche principale. Que dire de cette pratique sinon qu'encore une fois elle consiste à « localiser » des Globales, ce qui est à proscrire. La structure des

unités en Object Pascal offre une solution bien plus élégante : la section INITIALIZATION.
Ce mot clé remplace le mot BEGIN de l'unité elle-même. Elle se termine par le END finale de l'unité. Le code qui s'y trouve est exécuté au chargement de l'exécutable, ce qui en assure le traitement précoce avant même qu'une seule fiche (normalement) n'existe. Le code d'initialisation sera ainsi placé dans cette section qui lui est destinée.

2. Le traitement des structures dynamiques

Lorsque que des structures globales sont consommatrices de mémoire dans le segment de données (c'est à dire, en principe, dès qu'une Globale dépasse quelques octets...) il convient de les rendre dynamiques. Le code qui les créera en mémoire trouve naturellement sa place dans la partie INITIALIZATION décrite plus haut. Cette technique est rendue obligatoire pour les Globales, même de faible volume, fondée sur le modèle TOBJECT puisque celui-ci est dynamique par essence. Le code source de la VCL de DELPHI démontre parfaitement ce mécanisme lors de la création d'objets systématiquement présents comme APPLICATION, SCREEN, PRINTER ou CLIPBOARD.
Ainsi, l'initialisation à « -1 » d'une variable LONGINT ou la création d'un objet de type TLIST global seront des actions codées dans la partie INITIALIZATION. Toutefois, un problème se pose, celui de la destruction des données dynamiques.
DELPHI 2 offre une solution rationnelle par l'ajout du mot clé FINALIZATION ayant la fonction inverse de INITIALIZATION. Ce code est exécuté quand l'exécutable se termine. Les parties d'initialisation et de terminaison de toutes les unités en possédant sont exécutés dans l'ordre inverse ce qui assure une gestion rationnelle de ces codes particuliers. Ce que fait le mot clé FINALIZATION de DELPHI doit être simulé par programmation en DELPHI 1 qui offre la procédure ADDEXITPROC à laquelle on passe en paramètre le nom de la procédure assurant la fonction de terminaison. Cette dernière centralisera le code de déallocation des blocs mémoires réservés lors de la phase d'initialisation.

D. Astuce de l'instruction WITH

L'instruction WITH permet au compilateur de générer un code plus efficace, il est donc conseillé de ce servir de cette instruction lorsque plusieurs affectations qui se suivent concernent le même objet. L'une des particularités de WITH, dans le code machine généré à sa place, est qu'un pointeur vers l'objet déférencé est alloué durant sa portée. Ainsi, WITH autorise la syntaxe suivante, un peu étrange lorsqu'on la voit la première fois :

```
WITH TMYFORM.CREATE(SELF) DO TRY SHOWMODAL ; FINALLY RELEASE ; END ;
```

L'instruction SHOWMODAL ci-dessus porte sur la variable de type TMYFORM créée dans l'instruction WITH. Mais il n'y a pas de variable ... On utilise ici la zone de stockage pointeur temporaire allouée par WITH. Une telle construction doit obligatoirement être utilisée conjointement à un bloc de protection de ressource TRY...FINALLY. Entre ces deux instructions on a tout loisir de placer le code qu'on désire. Pour l'appel de fiches, l'exécution d'une requête SQL ou tout autre processus nécessitant la création temporaire d'une variable objet il est toujours préférable d'utiliser cette astuce de l'instruction WITH qui évite la déclaration d'une variable et rend le code plus efficace.
Voici un autre exemple démontrant une requête n'utilisant aucune variable TQUERY:

```
WITH TQUERY.CREATE(SELF) DO TRY DATABASENAME :='DBDEMOS' ;
SQL.ADD('SELECT MAX (TOTALAMOUTPAID) AS TOTAL FROM ORDERS') ;
```

```
ACTIVE :=TRUE ;
MaVariableTotal :=FIELDBYNAME('TOTAL').AsFloat ;
ACTIVE :=FALSE ;
FINALLY FREE ; END ;
```

A noter : Il est clair que l'imbrication de nombreux niveaux de WITH est nuisible à la lecture du code source, tout comme le sont les structures IF THEN ELSE qui se suivent. Face à un choix entre écriture « académique » et lisibilité des sources on choisira toujours cette dernière.

E. Utilisation des PROPERTIES dans les fiches

Le modèle objet de DELPHI défini les propriétés d'un objet comme des variables fantômes remplacées à la compilation par des appels aux champs internes ou aux procédures et fonctions déclarées dans les sections READ et WRITE. Cette méthode, très utilisée dans la conception des composants, peut être mise à profit dans bien d'autres cadres, notamment au sein des objets TFORM sous-classés que sont les fiches que vous manipulez dans vos applications.

Il ne faut jamais perdre de vue que la conception d'une fiche dans DELPHI n'est ni plus ni moins qu'une écriture guidée d'un sous-classement de l'objet TFORM (ou de l'un de ses descendants avec DELPHI 2). Dès lors, il n'y a aucune raison de ne pas utiliser le mot clé PROPERTY pour définir des propriétés ayant des comportements complexes. Notre conseil est ici d'utiliser au maximum cette technique pour simplifier l'interface des fiches lorsque celles-ci doivent communiquer avec d'autres fiches.

Prenons un exemple simple ; dans un projet vous disposez d'une fiche « liste des clients » et vous ajoutez une fiche « détail d'un client ». Vous souhaitez naturellement pouvoir appeler une fiche détail en double-cliquant dans la liste des clients. Plusieurs techniques de synchronisation existent, suivant si cette dernière doit être persistante ou non. Nous supposerons une synchronisation non persistante dans cet exemple (donc à l'entrée uniquement). La méthode qu'on voit le plus généralement employée consiste, dans la fiche liste et en réponse au double-clic de la grille de données, à créer une fiche détail puis à la synchroniser par code en appelant par exemple la méthode Findkey d'une de ses tables. Une telle approche n'est pas conforme au conseil d'isolation donné dans une rubrique précédente. Toutefois, dans ce cas précis (quoi doit être généralisé à de nombreuses circonstances équivalentes) il n'est pas toujours possible d'opter pour la solution de la fonction appelante tel que nous le décrivions. Dans un tel cas, et dans d'autres ou une fiche doit effectuer des traitements pilotés par une autre fiche (ou objet, ou code quelconque) il convient d'exploiter les capacités des propriétés définies par PROPERTY.

Pour continuer notre exemple, nous résoudrons ici le problème en créant, dans la fiche détail, une propriété PROPERTY CODECLIENT du type du code client (supposons un LONGINT). En lecture cette propriété fera référence à un champs privé du TFORM détail, en écriture à une procédure privée dont la tâche sera d'effectuer la synchronisation.

Ainsi, la propriété (dans la partie PUBLIC de l'objet TFORM de la fiche détail) ressemblera à:
PROPERTY CODECLIENT :LONGINT READ FCODECLIENT WRITE SETCODECLIENT ;
La procédure SETCODECLIENT prendra un LONGINT en paramètre qui servira à synchroniser la fiche. Ainsi, depuis la fiche Liste des clients, il suffira d'écrire :
`MaficheDetail.CodeClient :=26 ;` pour que la fiche détail soit immédiatement synchronisée sur ce code client sans avoir, du côté de l'appelant, à connaître le moindre morceau de la logique de cette action ni aucun champ de la fiche détail.

Nous ne répéterons pas les avantages d'une telle isolation et insisterons ici sur le fait que, demain, vous pourrez modifier le code effectuant la synchronisation (peut-être par une méthode plus rapide ou plus sophistiquée) sans que cela n'influence la moindre autre fiche de votre application.
Notre conseil sera ici d'insister sur la nécessité d'exploiter cette technique lorsque celle de la fonction appelante n'est pas appropriée afin de conserver le même degré d'isolation dans toute l'application.

F. Les Traitements en tâche de fond

Certaines application nécessitent la création de tâches de fond. Parmi ceux-ci on rencontre souvent l'affichage d'une horloge (exemple stupide mais générique). Pour gérer de telles tâche il existe plusieurs méthodes qui dépendent beaucoup de l'OS choisi et du compilateur natif qu'on utilise. Malgré tout, on constate que la majorité des développeurs utilise un TIMER comme déclencheur. Si cette méthode est acceptable dans certains cas, il s'avère qu'il existe, aussi bien sous Windows 3.1 que sous Windows 95 / NT, d'autres techniques plus efficaces.

Sous Windows 3.1, et avec DELPHI 1, la meilleure d'entre elles consiste à programmer l'événement ONIDLE de APPLICATION. Aucune ressource TIMER n'est utilisée, et on a l'assurance que la tâche n'est appelée que lorsque l'application ne fait rien d'autre qu'attendre. En cas de traitement long il n'y a pas de messages TIMER qui viennent encombrer la mémoire non plus. La fréquence de cet événement est assez grande.
Sous Windows 95 et NT, et avec DELPHI 2, nous conseillons la création d'une tâche THREAD dont la priorité sera IDLE, ce qui recréé un équivalent de la technique 16 bits tout en tirant avantage de la technologie 32 bits.

On peut alors se demander quand utiliser des timers ? ... Uniquement lorsque vous avez besoin de mesurer un laps de temps à peu près constant. Les timers peuvent être utiliser pour distinguer des événements entre eux, selon un seuil programmé.
Les timers sont en fait réellement utiles lorsqu'il faut qu'un certain temps minimum espace deux événements. Avec ONIDLE votre tâche peut être appelée 10 fois ou 1000 fois par seconde, vous ne maîtrisez pas la temporisation. En utilisant un timer programmé sur 10000 ms vous aurez la certitude que le gestionnaire d'événement ne sera pas appelé plus de 6 fois par minute. On peut donc accepter l'utilisation d'un timer lorsque le gestionnaire d'événement appelé effectue des tâches qui doivent être espacées dans le temps par un intervalle minimum. N'oubliez pas que Windows garanti un espace minimum et non un espace maximum.

En dehors des horloges affichées, d'autres traitement (plus essentiels) peuvent être placés en tâche de fond. Les impressions lourdes, et les requêtes sur données en font partie. Si vous utilisez un générateur de document de type ReportSmith ou un outil comme QUICKREPORT, vous ne pourrez pas maîtriser l'impression en tâche de fond. Si vous optez pour une programmation intégrale (ce qui se justifie dans certains cas) vous pourrez concevoir une séquence interruptible que vous placerez dans la gestion de l'événement (ou de la tâche) IDLE. Pour les requêtes sur données, tout dépendra de la méthode retenue et de l'OS.

Sous Windows 95 / NT, toutes les techniques sont possibles, notamment la création d'un THREAD dans lequel une requête SQL est envoyée au BDE. Cette technique permet d'interroger des données sans bloquer l'application, ce qui est très intéressant pour les traitements statistiques longs.
En environnement 16 bits une telle possibilité n'est pas offerte, mais vous pouvez fort bien la simuler. Si un traitement de données est très long, vous avez la possibilité,

plutôt que d'utiliser une requête SQL, d'effectuer le traitement vous-mêmes en balayant les tables.
Bien entendu cela est moins efficace, mais ici, ce qui est intéressant c'est que cette tâche ne soit pas bloquante, peu importe qu'elle dure 10 minutes de plus.
Vous pourrez ainsi concevoir une routine interruptible que vous placerez dans un gestionnaire de ONIDLE.

G. La dynamique des données

Lorsqu'une application utilise plusieurs fiches, ce qui est donc le cas le plus souvent, et que ces fiches font références à des données communes, il convient de rafraîchir ces dernières afin qu'un changement intervenu dans une fiche soit immédiatement visible dans une autre.

Prenons un exemple fort simple : Au milieu de la saisie d'une commande l'utilisateur décide de créer un nouvel article en appelant la fiche de maintenance de la table des articles. Lorsqu'il revient à la commande en cours, il s'attend naturellement à ce que l'article fraîchement créé soit accessible dans le DBLOOKUP servant au choix de l'article. Si vous n'avez rien programmé en ce sens, l'article créé ne sera pas visible !

Comme il serait très vite fatigant de programmer les événements ONSHOW ou autre de toutes les fiches de l'application pour y écrire des suites de « MaTable.Refresh ; », et ce, pour chaque table sans en oublier une, nous proposons ici une méthode plus rapide.
Elle consiste à programmer un seul événement, ONACTIVEFORMCHANGE de l'objet SCREEN, et d'y placer un code générique qui s'occupera de rafraîchir tous les objets TTABLE automatiquement. Au passage on en profitera pour traiter les TQUERY qui nécessite une fermeture suivie d'une ouverture pour obtenir le même effet.

Ce code balaye tous les composants de la TFORM qui devient active (ACTIVEMDICHILD pour le modèle MDI) et teste leur type par une instruction du type « IF .... IS TTABLE THEN ... ».
En réponse positive à ce test il conviendra de transtyper le composant en un TTABLE pour atteindre sa méthode REFRESH. Pour les TQUERY on enchaînera CLOSE et OPEN.

Cette solution à l'avantage de la simplicité, de la compacité, de l'élégance le tout en étant totalement indépendante du nombre de fiches de l'application et de la structure de ces dernières dans lesquelles, cerise sur le gâteau, il n'y a rien à programmer...
Ouf ! Autant d'avantages pour si peu de code, que demander de plus...
... En fait il y a un petit problème (c'était trop beau !) : effectuer un REFRESH sur un TDATASET arrête les modes EDIT et INSERT. Dans la séquence proposée, on testera la propriété STATE du TDATASET et on effectuera le REFRESH que si STATE=DSBROWSE.

H. Localisation des interface des fiches

La plupart des projets possèdent des fiches servant à la saisie d'informations ponctuelles (mot de passe par exemple) ou à la sélection de données (liste des clients par exemple). Très souvent les développeurs arrêtent leur réflexion à la conception de telles fiches sans penser à leur utilisation. Dès lors, il n'est pas rare de rencontrer dans une application la répétition du même code d'appel d'une même fiche. La séquence est généralement du type (dans le meilleur des cas):

```
Procedure ButtonPassWordClick1(Sender :Tobject) ;
Var F :TFInputPassWord ;
```

```
begin
F := TFInputPassword.Create(self) ;
F.Edit1.text:=" ;
If F.Showmodal=MrOK then motdepasse :=F.Edit1.text.... else ....
F.Release ;
end,
```

Ainsi, si plusieurs modules ont besoin de saisir un mot de passe (ou d'offrir une sélection d'un client, d'un article...), on retrouve le même code, avec parfois des variantes, un peu partout. Une telle pratique est à proscrire pour de nombreuses raisons évidentes.

L'un des plus importantes est que chaque module appelant doit avoir connaissance de la classe de l'appelé, de sa structure, du nom de ses champs et propriétés. Toute amélioration de la fiche appelée se traduira par une maintenance lourde et des risques de bugs pour avoir oublié de modifier l'un des appels.

L'une des règles de la programmation qui se trouve renforcée dans le modèle Objet est que l'Appelé ne doit pas connaître l'Appelant. On peut étendre ce prédicat à son inverse, ce qui, nous l'avons prouvé, a un sens bien concret notamment en ce qui concerne la maintenabilité de l'application.

De fait, notre conseil ici est que toute fiche doit contenir dans l'unité qui la déclare une procédure ou une fonction isolée (hors de l'objet fiche) déclarée dans la partie INTERFACE de l'unité dont la fonction première sera de résoudre les problèmes d'appel à la fiche sous-jacente. Seule cette procédure ou fonction sera utilisée par le projet. Elle seule connaît certains détails de la fiche appelée, et comme elle se trouve déclarée dans la même unité de code, sa modification, dans le cas de la modification de la fiche elle-même, sera à la fois intuitive, directe et centralisée.
En fait, ce que nous décrivons ici est l'un des principes des méthodes de classe. Il est donc tout à fait licite de mettre en œuvre ce mécanisme en déclarant une méthode de classe dans la fiche elle-même. Ce jeu déclaratif réclame une très bonne connaissance du modèle Objet de DELPHI, toutefois sa mise en œuvre est extrêmement simple ce qui force a adopter ce modèle pour les fonctions appelantes.

L'exemple qui suit va démontrer cette simplicité.
Supposons un projet où il existe une fiche (TFORM) appelée TFInputText possédant en tout et pour tout qu'un champ EDIT (EDIT1) et un bouton de validation renvoyant un code modal MROK. Nous voulons utiliser cette fiche à chaque fois qu'un morceau de texte devra être saisi. Le champ EDIT devra être initialisé à l'entrée et la fonction appelante devra retourner le texte tapé par l'utilisateur.

Pour mettre en œuvre cette fonction, nous allons, dans TFinputText, déclarer une méthode de classe dans la partie PUBLIC de l'objet TFORM :

```
PUBLIC
CLASS FUNCTION ASKTEXT(CONST S :STRING) :STRING ;

Dans la partie IMPLEMENTATION nous écrirons :

CLASS FUNCTION TFInputText.AskText(Const S :String) :String ;
Begin
Result :=" ;
With Create(Application) do
try
  Edit1.text:=s ;
```

```
  Showmodal ;
  Result :=edit1.text;
Finally release ; end ;
end ;
```

Le code ci-dessus utilise l'astuce de notation de l'instruction WITH (étudiée dans le présent document) évitant la création d'une variable temporaire. En dehors de cela, le code est très simple : il créée une instance de la classe, initialise le champ EDIT, appelle la fiche en mode modal, renvoie le résultat de la fonction puis supprime l'instance de la fiche. On peut bien entendu faire évoluer cette séquence vers quelque chose de plus sophistiqué selon les besoins.

NDLA : A l'heure actuelle mon avis reste partagé entre l'utilisation d'une véritable fonction autonome et celle d'une méthode de classe. Cette dernière solution me séduit par sa « beauté » très Objet, toutefois il faut connaître le nom de la classe pour appeler la fonction, nom qui n'est pas toujours très significatif et alors même qu'on pourrait vouloir changer la Classe appelée. La fonction autonome à l'avantage d'encapsuler la complexité et de transformer la stratégie Objet sous-jacente en un simple mot ressemblant à toutes les fonctions de service de l'environnement. Cette banalisation me semble intéressante aussi. A défaut pour l'instant d'un plus grand retour sur mes propositions, chacun aura le choix entre les deux mises en œuvre de la Fonction Appelante présentées ici.

1. Le cas général

La forme de la procédure d'appel n'est pas figée, elle peut dépendre de la fonction même de la fiche. Toutefois, et dans les cas qui nous intéressent on peut dégager certaines formes très génériques. Plutôt qu'un discours théorique voici quelques exemples d'en-têtes de procédures ou fonctions qui vous en diront plus :

Pour une saisie de mot de passe : FUNCTION GETPASSWORD :STRING ; Le mot de passe est retourné directement par la fonction, il est vide si l'utilisateur abandonne sa saisie (OK / CANCEL).

Pour une fiche de sélection de bons de commande: FUNCTION GETORDER(VAR CODE :LONGINT) :BOOLEAN ; Ici, la fonction renvoie TRUE si l'utilisateur a confirmé sa sélection ou FALSE dans le cas contraire. Le code de la commande sélectionnée est renvoyé dans le paramètre VAR. Une telle fonction doit débuter par l'initialisation du paramètre variable qui ne doit jamais, quelque soit le choix de l'utilisateur, contenir du « garbage ».

Pour les fonctions de sélection, il peut être intéressant de proposer un positionnement par défaut dans la fiche de sélection. La décision de mettre en œuvre cette fonction est du ressort du développeur qui estimera l'intérêt selon le contexte de l'application. Si cette fonction est ajoutée elle consiste en l'utilisation du paramètre VAR pour synchroniser la fenêtre de choix sur l'élément dont le code est passé. Cette méthode implique que l'appelant initialise convenablement le paramètre variable. Pour cette raison les fonctions de ce type devront se signaler par un mot clé spécial contenu dans leur nom. S'il est raisonnable de conseiller le préfixe GET pour toutes les procédures et fonctions dont état ici, nous n'avons pas d'avis fixé quant au mot clé d'avertissement. Puisque le paramètre d'appel doit être initialisé, on peut proposer GETINIT comme préfixe.

2. Les fiches de sélection de données

Si les paragraphes qui précèdent donnent une orientation pour la mise en œuvre des fiches de sélection, il est bon de figer les choses avec plus de rigueur.

Lorsqu'une fiche est utilisée pour la sélection de données, elle se doit de proposer une fonction dite « fonction appelante » dont le préfixe est GET ou GETINIT. Il ne peut s'agir que d'une fonction dont le résultat est obligatoire Booléen, dans le même esprit que les méthodes EXECUTE des dialogues standard de la VCL. Le premier paramètre est de type variable, il est utilisé en entré pour la synchronisation dans le cas d'un GETINIT, et, dans tous les cas, en sortie pour renvoyer la clé de l'enregistrement sélection (si le résultat de la fonction est TRUE). Si l'utilisateur annule son choix, la fonction retourne FALSE et le paramètre VAR est vidé (chaîne vide ou -1 pour les clés incrémentales par exemple).
Voici quelques cas particuliers :

a) Les clés composées

Il est clair qu'il est souvent nécessaire de retourner non pas une seule valeur mais une suite de valeurs. Bon nombre de tables possèdent des clés primaires composées. Dans ce cas il convient de remplacer le paramètre VAR par un paramètre de type TSTRINGS. Cet objet sera créé par l'appelant qui aura la charge de le supprimer. L'objet réellement créé sera de type TSTRINGLIST. La procédure utilise l'ancêtre TSTRINGS afin d'être compatible directement avec les suites de chaînes des TMEMO, TLISTBOX, TCOMBOBOX et autres objets possédant des listes de chaînes. Le nombre de valeurs retournées, s'il peut être variable dans certains cas, sera lu, au retour de la fonction, par la propriété COUNT du paramètre TSTRINGS.

b) La sélection étendue

Le modèle de fonction appelante jusqu'ici proposé à l'avantage d'être simple et de couvrir la majorité des besoins. Toutefois, il est possible que, dans certains cas particuliers, il faille gérer le retour de fonction avec plus de précision ou proposer des paramètres supplémentaires. Par exemple, un fiche de sélection peut proposer la création d'une nouvelle fiche. Une telle fonction doit être paramétrable. Toutefois il est impératif de respecter l'esprit même du « standard » que représente la technique de la fonction appelante.
Dès lors il faudra créer une première fonction appelante possédant, en second paramètre, un champ Booléen autorisant, ou non, la création. Ce paramètre s'appellera, par convention, ALLOWCREATE. Cette fonction appelante utilisera le préfixe CREATE en plus des autres préfixes déjà définis. On pourra ainsi avoir des fonctions GETCREATE, GETINITCREATE. Nonobstant, on créera tout de même une fonction de base de type GET ou GETINIT qui ne fera qu'appeler la fonction CREATE avec le paramètre ALLOWCREATE à FALSE. Il va sans dire que l'affichage du bouton permettant la création (et de tous les éléments visuels relatifs à cette fonction) doit être gérer convenablement selon la valeur de ALLOWCREATE.

Une fonction peut aussi renvoyer une réponse plus riche de sens qu'un simple Booléen. Par exemple, une fiche de sélection peut autoriser la création (cas précédant) mais ne pas être à même d'effectuer elle-même une telle création. Dans ce cas, il faut que l'appelant puisse savoir si l'utilisateur a sélectionné un enregistrement, a annulé son choix ou bien a fait une demande de création.

Il serait en fait préférable qu'une telle fiche gouverne elle-même la création. Dans le cas de son annulation elle se placerait à nouveau en sélection, dans le cas de sa validation est retournerait la clé exactement comme si l'enregistrement avait été seulement sélectionné. Cette possibilité pose un problème lié à la nature même de l'enregistrement à créer. Dans certains cas il s'agit juste de créer une clé primaire. Si elle est automatique il n'y a même rien à demander de plus à l'utilisateur.
Si elle ne l'est pas, un simple champ TEDIT est nécessaire. Dans la plupart des cas on ne peut malheureusement pas se contenter de créer uniquement la clé primaire. Certaines informations hors clé sont obligatoires (le nom et le prénom d'un client alors

que seul le code client fait partie de la clé par exemple). Ces informations obligatoires imposent alors la création une « mini fiche » de saisie.
Cette dernière peut devenir assez vite complexe si des business rules particulières s'appliquent, si des tables de références sont utilisées, etc.
Dans un tel cas, la fiche de création peut ressembler énormément à la fiche de saisie / modification « normale ». Il serait stupide d'avoir à la dupliquer. Autant dire qu'une telle redondance de code est plus que prohibée !

La re-exploitation de la fiche de saisie en mode MODAL peut poser un problème si l'application est en mode MDI. En effet, la fiche a été créée avec le style « fsMDIChild » et on ne peut pas ouvrir une telle fiche avec SHOWMODAL. Nous testons en ce moment des mécanismes permettant une telle adaptation automatique (ce qui n'est pas simple du tout). Lorsqu'une solution aura été validée, elle sera intégrée au Plan Qualité.

Pour en revenir à la fonction appelante, il existe donc des cas où la création est possible mais où la fiche de sélection ne peut pas gérer elle-même cette création. Il faut que l'appelant puisse détecter cette situation puisque faute d'une solution totalement intégrée (Cf. paragraphe précédent) il aura la charge de cette création. On admettra ainsi que le résultat de la fonction appelante ne soit pas Booléen mais de type énuméré. Le type lui-même sera déclaré dans l'unité des Globales de la façon suivante :
TYPE TENHANCEDCREATE = (tec_cancel, tec_Select, tec_Create, tec_user1, tec_user2) ;

Les trois valeurs correspondent respectivement àl'annulation de la demande, à la validation de la sélection et à la demande de création. Les deux premiers cas sont équivalents au résultat Booléen FALSE / TRUE du mode non étendu. La valeur de CANCEL est en première position pour que d'un point vue de la représentation interne elle soit équivalente à FALSE, et que toute initialisation à zéro corresponde, par défaut, à une annulation.

Les valeurs « tec_user1 » et « tec_user2 » permettent de gérer deux modes de sorties supplémentaires (en dehors de ceux de base et du mode Création). Cela permet d'avoir une seule déclaration de type valable tout au long d'une application. On pourra fixer des valeurs plus explicites, en relation avec l'application tout autant qu'étendre le nombre de ces valeurs. La raison oblige à conseiller de restreindre ce nombre mais vous restez maître de cette décision dans votre application du moment que votre choix est motivé et qu'il suit la présente convention.

Un fonction appelante retournant un type énuméré à la place d'un type Booléen se signalera par le préfixe ENHANCED ajouté aux autres préfixes.
On pourra ainsi trouver des fonctions GETCREATEENHANCED, GETINITCREATEENHANCED par exemple.

Toutefois, il sera à la charge du développeur de proposer systématiquement une fonction GET ou GETINIT de base. Celle-ci se chargera d'appeler la fonction étendue et d'interpréter la valeur de retour en la limitant à la signification Sélection / Non Sélection.

c) Francisation

Si vous désirez à tout prix franciser les préfixes, utilisez :

GET	CHOIX
GETINIT	CHOIXINIT

GETCREATE	CHOIXCREATION
GETINITCREATE	CHOIXINITCREATION
GETENHANCED	CHOIXENTENDU
etc...	

I. Externalisation des traitements

Nous avons vu (*Traitements génériques et variables globales*, page 18), comment gérer les globales et certains traitements par l'intermédiaire d'une unité commune appelée COMMON. Dans le même esprit, nous allons voir ici qu'on peut pousser assez loin la séparation entre les traitements et la gestion de l'interface.

Fonctionnellement, une fiche (TFORM), n'a d'intérêt que celui de permettre de coder une interface utilisateur propre très rapidement. Est-il donc logique d'intégrer des traitements dans l'unité de gestion d'une fiche ? Comment faire la différence entre un traitement qui doit être effectué cette dernière et un autre qui mériterait d'être externalisé ? Où placer le code ainsi extrait des fiches ?

La localisation de ces traitements particuliers tombe sous le sens, ils doivent se trouver dans une unité commune. COMMON semble ainsi être une bonne place. En fait ce choix est plus délicat qu'il n'y parait, mais, pour simplifier, on admettra que les petits projets utiliserons l'unité COMMON comme zone de stockage des traitements externalisés. Il faudra tout de même bien les identifier dans la partie interface.

Pour les autres projets il conviendra de créer une nouvelle unité que nous nommerons EXCODE (EXternal CODE).

Attention, le code placé dans l'unité de gestion d'une fiche a l'avantage de n'être chargé que lorsqu'il existe une instance de l'objet fiche alors que le code placé dans une unité commune augmentera la taille minimale utilisée par l'application. En ce qui concerne l'unité COMMON cela se justifie pleinement, pour l'unité EXCODE il faudra trancher si la taille de cette dernière devenait trop importante. En général on ne tient plus compte aujourd'hui de la taille des logiciels et 100, voire 500 Ko, de plus ou moins ne gênent personne. Pour certains projets il faudra tout de même garder présent à l'esprit les conséquences éventuelles d'un tel choix.

Nous savons maintenant où placer le code délocalisé, reste à savoir quel code mérite ce traitement et quel avantage tirer de cette technique.

Stricto sensu tout code qui ne gère pas explicitement de l'interface devrait, dans la logique ici proposée, se trouver dans EXCODE. Comme nous l'avons vu il faudra parfois être moins catégorique. Le meilleure compromis consiste à externaliser uniquement le code dont on peut pressentir une éventuelle réutilisation. Cette méthode permet un tri efficace mais n'est pas absolue, il faut ainsi en admettre les limites.

Prenons un exemple concret :
Dans une application donnée on trouve une fiche permettant de manipuler des entités « fournisseurs », or, il se trouve que chaque fournisseur possède une clé d'accès (primaire) calculée à la création de tout nouveau enregistrement. La clé utilise les 4 premiers caractères du nom de la société du fournisseur plus le mois et l'année de création de la fiche sur 4 chiffres. Si le nom de la société fait moins de 4 caractères, des espaces complètent celui-ci. Comme cette clé peut ne pas être unique (deux enregistrements fournisseurs créés le même mois dont le nom de société commence par les 4 mêmes lettres), un caractère supplémentaire est ajouté en fin de clé. Il s'agit du zéro. En cas de doublon ce dernier chiffre sera incrémenté. Ainsi, la société

DUFOUR dont l'enregistrement est créé le 10 avril 97 aura pour clé DUFO04970, la société KTH saisie le 15 mai 1997 aura pour clé KTH 0597 et la société DUFOIX créée dans le système le 25 avril 1997 aura pour clé DUFO04971 (le « 1 » final gère le doublon avec le premier exemple).

Gérer la constitution d'une telle clé réclame un peu de code, ne serait-ce que pour chercher les doublons et les gérer par incrémentation du caractère final.

Placer le code dans la fiche de gestion des fournisseurs semble, de prime abord, assez raisonnable puisqu'il n'y a pas d'autres fiches permettant la création d'un enregistrement fournisseur. Toutefois un tel cas n'est pas à écarter. Les besoins de l'utilisateur pourront évoluer, il sera peut-être aussi décidé d'importer des enregistrements depuis un autre système (ou d'anciennes tables), etc.
Lorsqu'un tel besoin se fera sentir il sera peut-être difficile d'extraire le code de son contexte, ce qui demandera de toute façon du travail alors que le placer ailleurs d'amblé évite tous les problèmes (ce code resservira-t-il ? ce code sera-t-il facile à extraire de son contexte ? Où ce code sera-t-il alors déplacé ? D'ailleurs, sera-t-il dupliqué ou réellement déplacé ? etc.).

La solution consistera ici à créer une fonction à laquelle on passera en paramètre le nom de société du fournisseur et qui retournera la clé. La date de création ne paraît pas indispensable puisque la fonction peut fort bien interroger l'horloge. Mais que ce passe-t-il s'il faut un jour recréer les clés de certaines fiches ? Si le modèle conceptuel n'a pas prévu de conserver la date de création des enregistrements il n'y a plus à se poser la question, si cette date est conservée, il faudra alors prévoir de la passer en paramètre à la fonction.
La fonction sera localisée dans l'unité EXCODE.

Au travers de cet exemple nous avons pu dégager une certaine logique qu'il conviendra d'appliquer convenablement, c'est à dire en faisant grand usage de son libre arbitre qui, pour s'exercer, nécessite une bonne compréhension du mécanisme. La présentation un peu longue d'un contexte particulier, comme souvent dans le présent document, permet de vivre de l'intérieur le cheminement des pensées, il est donc essentiel d'oublier le côté anecdotique pour en extraire la logique sous-jacente.

IV. Techniques particulières

La structure du présent document est encore balbutiante et, il faut l'avouer, loin d'être parfaite. Dès lors, comme il fallait bien classer ce qui va suivre quelque part, un chapitre au nom aussi flou que « techniques particulières » semblaient s'imposer...

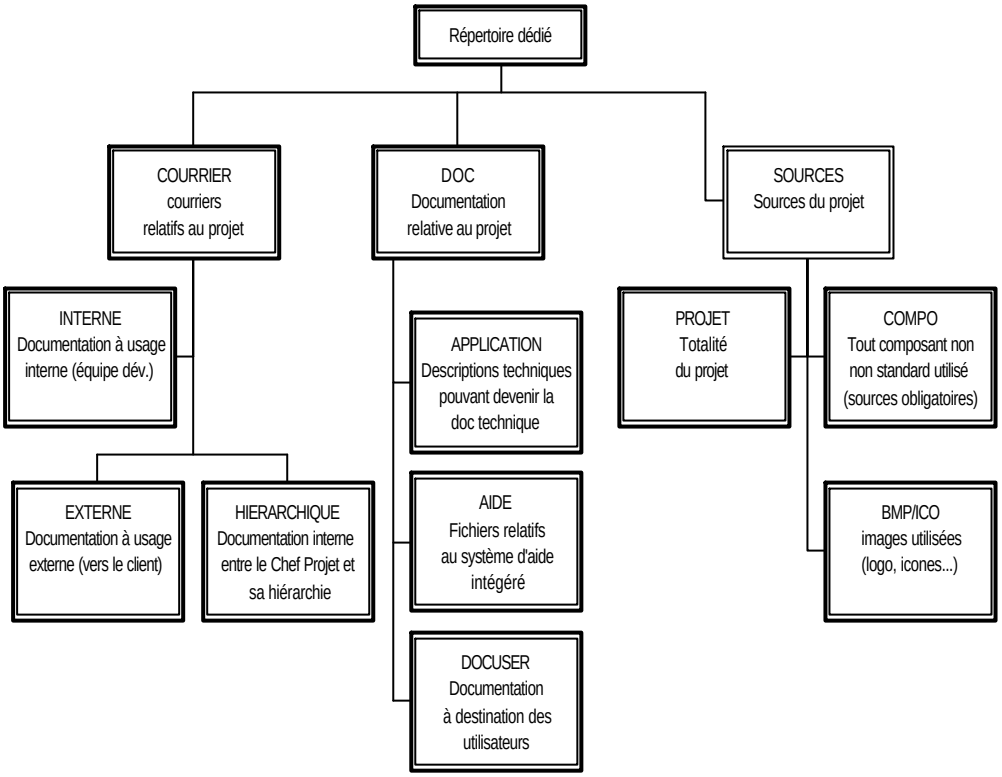
A. L'organisation du dossier

Si les détails techniques d'implémentation ont leur importance, la tenue du dossier, notamment son organisation, peut être un élément décisif lors de futures maintenances, qu'elles soient correctives ou évolutives.

Ainsi, il est souhaitable d'adopter une certaine rigueur dans le stockage de tous les éléments afférents au dossier. Si nous n'interviendrons pas ici sur le stockage des éléments physiques (cahier des charges, analyse, courriers en provenance de l'utilisateur...), nous conseillerons ici un modèle de classement des informations générées par le développement.

Bien évidemment, il s'agit toujours ici de conseil que nous vous enjoignons de suivre, et non d'une démarche académique qui, alors, pourrait être ridicule.
Voici un exemple d'arborescence de classement d'un dossier DELPHI :

Organisation disque d'un dossier



B. L'administration de l'application

La plupart des applications nécessitent la mise en place de certaines maintenances réservées à son administration. Pour exemple citons celle des utilisateurs notamment. Certains développeurs font le choix de créer un module exécutable séparé regroupant l'ensemble de ces fonctions. Dans certains cas cela est justifié.

Dans tous les autres il s'agit là d'une complication non justifiée et d'une perte de temps qui l'est encore moins (création d'un nouveau cadre d'application, etc). De plus, la séparation de ces fonctions est peu pratique : sur un poste donné on ne sera jamais certain d'avoir accès à l'exécutable d'administration ce qui complique la maintenance opérationnelle.

Plus loin, la désynchronisation physique des deux exécutables (application et administration) ajoute une contrainte au suivi du projet (cohérence, changement de certaines constantes, ...). Pire, il n'y a pas de moyen simple d'empêcher l'administrateur d'utiliser une version périmée de l'outils d'administration.

Pour toutes ces raisons, le conseil de cette rubrique sera d'intégrer le module d'administration directement au menu principal de l'application.
Bien entendu il ne s'agit pas de laisser ces outils en libre-service !
La technique la plus simple qui offre, au minimum, les mêmes garanties que celles d'un exécutable séparé, consiste à gérer les choses de la façon suivante :

Mise en place

1. Créez une entrée ADMINISTRATION dans le menu principal de l'application.

2. Ajoutez sous celle-ci les items de ce menu.
3. Ajoutez un item « Activer / Désactiver » auquel vous associerez un accélérateur (par exemple MAJ-CTRL-F12)
4. Revenez sur l'entrée principale de ce menu et placez sa propriété VISIBLE à FALSE.

Mise en œuvre
Windows est un environnement subtil, et « non visible » (VISIBLE=FALSE)ne signifie pas « non actif », ce qui se gère avec un autre paramètre (ENABLED=FALSE). De fait, les accélérateurs connectés à des entrées de menu non visibles mais actives restent fonctionnels.

Lorsque l'utilisateur active l'accélérateur « Activer/Désactiver » votre application réagira de cette façon : si la propriété VISIBLE du menu administration est TRUE, il suffira de la basculer à FALSE (en ayant éventuellement pris soin de fermer automatiquement toutes les fenêtres relatives restant ouvertes), si la propriété VISIBLE est à FALSE, l'application affichera un dialogue « mot de passe ». Si le mot de passe administrateur est correct la propriété VISIBLE du menu sera basculée à TRUE, sinon, rien de se passera.

Le risque qu'un utilisateur, même sans le vouloir, puisse activer le dialogue « mot de passe » est, statistiquement le même que celui qu'il découvre l'exécutable d'administration sur une disquette qui traîne ou sur un répertoire du réseau. Discuter du risque relatif de ces deux probabilités semble proche du célèbre débat sur le sexe des anges. Dans tous les cas l'utilisateur sera arrêté par un mot de passe administration ce qui revient donc strictement au même, in fine. La sécurité n'est donc ni meilleure ni moins bonne côté utilisateur, par contre celle de l'application ainsi que sa maintenabilité s'en trouvent renforcées.

Vous vous apercevrez vite que ce menu « administration » intégré à l'application vous incitera a ajouter ponctuellement certaines informations, certains paramètres que vous n'auriez jamais eu l'envie d'aller créer dans une seconde application dédiée. Ce qui, incidemment favorise une meilleure approche de l'administration de l'application.

Parmi ces petites choses qui peuvent trouver place dans le menu d'administration notons la gestion centralisée des erreurs d'exécution et des *Logs* (technique décrite dans le présent document).

C. La gestion centralisée des *Logs* et erreurs d'exécution

Si une telle gestion n'est que rarement réclamée elle n'en a pas moins d'importance pour autant. La maintenance d'une application peut se trouver simplifiée s'il existe un moyen de tracer les erreurs d'exécution ainsi que les connexions et déconnexions des utilisateurs. La majorité des applications sont sensibles à une fin brutale : certaines données ne sont pas mises à jour (déstabilisation des informations), certains fichiers peuvent être endommagés, etc.

Les utilisateurs (encore eux...) n'ont que rarement conscience de ces problèmes techniques et, malgré les conseils, certains sont de « dangereux récidivistes » qui ne connaissent d'autres façons de mettre fin à une cession de travail qu'en éteignant l'ordinateur. Sous Windows il faut aussi penser que d'autres applications peuvent être lancées durant l'exécution de la vôtre. Ces applications peuvent « planter » le système (avec blocage ou retour au niveau console) sans pour autant que votre application n'ait eut la moindre de chance de se terminer proprement.

Dans un environnement réseau il devient vite impossible de savoir ce qu'il s'est passé. Par contre, en cas de problème dans votre application (erreurs à l'ouverture de certaines fiches, perte de données, etc) au bout du fil un seul coupable, vous (le développeur), un seul responsable, votre programme.

Il est donc important de pouvoir s'assurer qu'il n'y a pas eu de dysfonctionnement au sein de votre application (trace des erreurs d'exécution) afin de détecter un éventuel bug, et qu'il n'y a pas eu de sortie suspecte d'un ou plusieurs utilisateurs (trace des Logs).
Les deux aspects se gèrent différemment, mais on utilisera une table centralisée pour leur mise en œuvre. Il ne peut s'agir d'un fichier ASCII car il vous faudrait gérer le partage en réseau sur ce dernier ce qui n'est pas forcément simple. Une table fera donc l'affaire. Elle fera partie de celles de l'application et sera gérée selon les mêmes méthodes (sur le serveur en modèle C/S par exemple).

1. La gestion des Logs

Une fois l'utilisateur connecté à votre application (code utilisateur et mot de passe validés) vous ouvrirez la table des Logs et inscrirez Date, Heure, Code utilisateur et un code de LogIn. Vous pourrez aussi ajouter à ces informations le code BIOS de la machine, le nom réseau, ainsi que tout autre détail vous permettant de localiser la machine et l'utilisateur de façon séparée (n'oubliez pas, en effet, qu'un problème peut-être lié à une installation donnée et qu'un même utilisateur peut changer parfois de machine, donnant ainsi au bug un caractère aléatoire si vous ne pouvez pas détecter la machine isolément).

Pour la sortie, le mieux est de vous brancher sur l'événement ONDESTROY de la fenêtre mère. Vous ouvrirez le fichier de Logs et inscrirez les mêmes informations en utilisant, cette fois, le code de LogOut.

Une simple interrogation SQL (de type GROUP BY) sur cette table vous permettra de savoir, pour chaque machine ou pour chaque utilisateur si le nombre des entrées correspond à celui des sorties. Une analyse rapide et simple qui permettra, en cas de problème, d'éliminer ou de confirmer certaines hypothèses.

2. Le suivi des erreurs d'exécution

Certaines erreurs ne peuvent malheureusement pas être interceptées lorsque, notamment, Windows se bloque ou retourne en mode DOS. Ces problèmes peuvent toutefois être détectés par la gestion des Logs puisque, par force, il manquera une sortie sur la station concernée (d'où la complémentarité de ces deux procédés présentés conjointement). Pour toutes les autres erreurs non gérées par votre application (TRY EXCEPT) il peut être essentiel de savoir ce qu'il s'est passé. Rares sont les utilisateurs capables de noter convenablement un message d'erreur. Certains, même, ayant un sens aigu de la culpabilité, iront même jusqu'à nier le problème, de peur qu'on le leur reproche. Ainsi, dans certains cas, votre application pourra souffrir d'un réel problème dont vous ne serez pas tenu au courant, problème pouvant avoir des conséquences à long terme.

DELPHI offre une méthode assez simple pour centraliser les erreurs, l'événement ONEXCEPTION de l'objet APPLICATION. En programmant un gestionnaire pour cet événement, il vous sera possible d'ouvrir la table des Logs et d'y inscrire le message ainsi que l'heure, la date, et les informations concernant la machine et l'utilisateur. La colonne contenant les codes LOGIN et LOGOUT sera utilisée avec un troisième code EXCEPTION, ce qui autorisera le filtrage des données pour leur affichage ou pour toute requête SQL. Régulièrement, lors de vos visites sur le site du client, vous pourrez inspecter la table des Logs et prendre connaissance des éventuels problèmes.

Le menu ADMINISTRATION (voir autre rubrique du présent document) pourra parfaitement contenir une fiche de visualisation de cette table. Elle peut être très simple et comporter un simple DBGRID et DBNAVIGATOR. Elle peut aussi être agrémentée des informations statistiques évoquées ici (couples LOGIN/LOGOUT par exemple) et comporter un bouton de RESET.

Concernant ce dernier point, chacun gérera en son âme et conscience. Faut-il désactiver la fonction RESET si le fichier *Log* contient une erreur ou s'il n'y a pas parité des LOGIN et LOGOUT ? Faut-il tout de même autoriser le RESET mais en garder une trace (avec les informations statistiques) sur un fichier ASCII ou une autre table inaccessible ? La réponse dépendra certainement de l'environnement, de la taille du réseau, des exigences du clients et du temps disponible pour coder ces options généralement non incluses dans le prix de vente du logiciel.

D. Cryptage

Si vous devez crypter des chaînes de caractères utilisez les fonctions suivantes (la clé étant fournie aux procédure sous la forme d'un WORD) :

```
const
  C1 = 52845;
  C2 = 22719;

function Encrypt(const S: String; Key: Word): String;
var
  I: byte;
begin
  Result[0] := S[0];
  for I := 1 to Length(S) do begin
    Result[I] := char(byte(S[I]) xor (Key shr 8));
    Key := (byte(Result[I]) + Key) * C1 + C2;
  end;
end;

function Decrypt(const S: String; Key: Word): String;
var
  I: byte;
begin
  Result[0] := S[0];
  for I := 1 to Length(S) do begin
    Result[I] := char(byte(S[I]) xor (Key shr 8));
    Key := (byte(S[I]) + Key) * C1 + C2;
  end;
end;
```

E. L'utilisateur Application

La majorité des applications en réseau (ou client / serveur) nécessitent une gestion des utilisateurs, souvent pour les droits d'accès, fréquemment pour stocker qui a modifié / créé quoi et quand.

Dans ce dernier cas, qui nous intéresse ici, il est important de prévoir un identifiant spécifique pour l'application elle-même, voire une plage d'identifiants réservés à l'application.

En effet, l'ensemble des traitements agissant sur les données n'est pas toujours intégralement sous le contrôle de l'utilisateur. Certaines mises à jour automatiques ou intégrations de données (transferts inter plate-formes par exemple) peuvent être déclenchés par l'application en fonction de son état.

Dès lors que des opérations de ce type ne sont pas explicitement lancées par un utilisateur, il convient de prévoir un code utilisateur pour l'application puisque celle-ci se comporte, fonctionnellement, comme un utilisateur qui prend la décision d'agir sur les données. On peut aussi opter pour un code utilisateur constitué du code de l'utilisateur « humain » auquel est un ajouté le code utilisateur de l'application ou du process.

F. Gérer les accès à l'application

Lorsqu'une application doit gérer des droits d'accès, en réseau ou en client / serveur, il est préférable (sauf cas particuliers) de ne pas utiliser directement les options de la base de données pour ce faire.
En effet, une telle gestion implique de manipuler sans cesse la base pour prendre en compte les départs et arrivées des utilisateurs (utilisateurs n'ayant plus accès à l'application, nouveaux utilisateurs, mutation de droits de certains utilisateurs). Au-delà de la lourdeur de cette gestion, chaque utilisateur dispose d'un mot de passe donnant réellement accès aux données de la base. Il peut ainsi utiliser ce mot de passe depuis un outil interactif. Il possède ainsi une vrai clé.

Il est plus intelligent de ne pas distribuer de clés réelles en utilisant une indirection. Le système fonctionne alors comme cela :

La base possède un fichier des utilisateurs protégé (mot de passe Paradox, Grant SQL,...). Seule l'application connaît le mot de passe qui n'est connu de personne.
Lorsqu'un utilisateur entre dans l'application il fournit son identifiant qui n'est autre que la clé d'accès à la table des utilisateurs. Il fournit aussi sont mot de passe.
L'application cherche l'utilisateur dans la table et vérifie à la fois son existence et la validité de son mot de passe.

Une fois l'identification passée avec succès, l'application note le groupe auquel l'utilisateur appartient (information stockée par le superviseur dans la table des utilisateurs).
Le code du groupe renvoie à une table des groupes qui contient au minimum le véritable mot de passe pour accéder à la base de données. Ce mot de passe correspond à une utilisateur imaginaire qui représente en fait les droits de tout un groupe.
Enfin, l'application ouvre la base en lui fournissant, automatiquement et de façon transparente, le véritable mot de passe.

On peut fournir à l'utilisateur un utilitaire lui permettant de changer à volonté son mot de passe qui n'est qu'un champ dans la table des utilisateurs sans aucune signification réelle.
En dehors de l'application, l'identifiant et le mot de passe d'un utilisateur ne lui permettent pas d'accès à la base de données. La protection est ainsi totale et cette indirection simplifie à l'extrême la gestion des droits de la base elle-même qui ne connaît que quelques utilisateurs dont la définition claire ne varie pas dans le temps (ou très peu).

Si d'aventure il s'avère nécessaire qu'un utilisateur accède à la base par d'autres outils que votre application, il faut soit revoir la stratégie (et gérer les droits directement avec

le SGBD-R) soit se contenter de donner le mot de passe du groupe (ce qui n'offre que certains droits et qui peut être changer régulièrement).

G. Prévoir les traitements génériques

L'expérience prouve qu'il s'avère souvent nécessaire d'ajouter ou de modifier des comportements concernant des fiches d'une application. Ces comportements peuvent intervenir, soit au moment de l'instanciation d'une classe fiche (descendant de Tform), soit au moment de l'appel du destructeur de l'instance (donc, soit en début, soit en fin de vie d'une instance de la classe).

Delphi, dans son modèle objet, et au travers de la VCL, propose, pour les fiches, deux événements importants qui sont : « OnCreate » et « OnDestroy ». Ces événements se situent dans une logique de délégation et il ne faut pas les confondre avec le constructeur et le destructeur de la classe (même si on peut supposer ici que les événements en questions sont générés depuis ces méthodes particulières de l'objet Tform).

Cette dualité délégation / sous-classement offre l'avantage de pouvoir choisir l'approche la mieux appropriée pour atteindre son but.

Pour mieux fixer la problématique, prenons un exemple de comportement de début de vie d'instance, et un autre de fin de vie (ce ne sont que des exemples qui pourraient, ponctuellement, être résolus autrement):
On peut vouloir ajouter à une application un paramètre global permettant d'afficher ou non les bulles d'aide, ajout qui pourrait se compléter de la nécessité, pour chaque fiche, au moment de son apparition, de prendre en compte les droits d'accès de l'utilisateur en cours, notamment en plaçant les ensembles de données en mode « read only » ou non.

On voit ici que traiter ce problème sur chaque fiche peut devenir une tâche assez lourde, soit parce qu'il y a déjà beaucoup de fiches dans l'application, soit parce qu'il faudra ajouter une contrainte supplémentaire à la « check list » indiquant les opérations minimales à coder sur chaque nouvelle fiche.

Que dire alors si les comportements en questions sont supprimés ou modifiés grandement dans le futur ? Chaque fiche devra être revérifée. Et chacune de ces interventions viendra alourdir le cycle de maintenance corrective tout autant que celui de la maintenance évolutive. Sans parler des inévitables bugs ponctuels que de telles manipulation risquent d'engendrer, ni même de l'accroissement de la taille du logiciel créé par la décentralisation de ces traitements à portée, pourtant, générique.

Pour l'instant nous nous sommes limité à un exemple d'ajout, en cours de conception d'une application, de nouvelles contraintes modifiant la liste des opérations minimales à coder dans la partie gérant la création de nouvelles instances de la classe.
Nous aborderons plus tard le problème des comportement devant intervenir sur la fin de vie des instances.

Pour résoudre le problème, la première idée est de reprendre chaque fiche en mode conception et d'ajouter ou de compléter son événement « OnCreate » afin, d'une part, de lire un paramètre global (voir *Traitements génériques et variables globales*, page 18) indiquant si les bulles d'aide doivent ou non être affichées et d'en répercuter l'effet, et, d'autre part, de lire le code d'accès de l'utilisateur courant pour modifier, pour chaque ensemble de données, sa propriété gérant les droits en écriture.

Concernant les bulles d'aide, il suffira de lire une variable globale de type « GLOB_DISPLAYHINT », de type booléen, et d'en affecter le résultat à la propriété « SHOWHINT » de l'objet TFORM courant.

Toutefois, on voit bien que, pour chaque fiche de l'application, il faudra répéter ce même code à l'identique, ce qui n'est pas forcément très subtil...

Pour les ensembles de données, chaque fiche représente un cas particulier. Telle fiche possédera un objet « Table1 », telle autre un couple « tClient, tFactures » et une autre, enfin, nécessitera peut-être de mettre à jour une quinzaine d'objet ayant tous des noms différents.

Que de travail ! ...Surtout qu'il faudra le répéter sur chaque fiche, sans rien oublier. L'oubli d'une seule table dans la liste ci-dessus pourra causer des effets inattendus (on appel çà des « bugs » aussi...).

Cette méthode n'est pas la bonne, nous en arrêterons la description ici.

La solution va, en réalité, dépendre de l'outil utilisé, à savoir Delphi 16 ou 32 bits. Pour ce dernier environnement il suffit de créer un TFORM de base pour toute application et d'en faire descendre toute fiche. De ce fait, lorsqu'il faudra ajouter des traitements génériques il suffira de modifier l'objet TFORM « maître » et de recompiler l'application. C'est là toute la magie des objets mise au service de l'héritage des TFORM.

Cette solution est de loin la meilleure mais elle a un inconvénient majeur : si Delphi 1, au niveau du langage, supporte parfaitement qu'on fasse dériver une fiche d'un descendant de TFORM, le designer de fiche ne sait pas travailler sur de telles classes dérivées. Si on modifie l'entête d'un objet TFORM pour indiquer une autre classe parente, il n'est pas possible de la modifier librement en mode conception.
On évitera ainsi cette approche en Delphi 1.

Par contre, ce qui marche en Delphi 1 fonctionne généralement bien en Delphi 32 bits, il existe alors une solution générique qui s'adapte aux deux types d'environnement. Elle consiste à créer dans l'unité COMMON (voir *Traitements génériques et variables globales*, page 18) une procédure pour gérer les traitements de début de vie et une autre pour ceux de fin de vie d'une instance de TFORM. Il s'agit de traitements génériques qui devront s'adapter à tous les cas de figure, nous verrons comment dans quelques lignes.

La seule contrainte, mais qui reste fort légère et qui, de surcroît est systématique, consistera à appeler l'une et l'autre de ces procédures dans le « OnCreate » et le « OnDestroy » de toutes les fiches de l'application.

Si ces événements existent déjà, on considérera toujours que ces appels sont effectués en premier.

Comment gérer la « personnalisation » des traitements ?
Les procédures ici étudiées, que nous appellerons INITCHECKPOINT et EXITCHECKPOINT, ne prennent qu'un seul paramètre de type TFORM. Leurs entêtes respectifs sont ainsi :

```
Procedure InitCheckPoint(Sender : Tform) ;
Procedure ExitCheckPoint(Sender : Tform) ;
```

Grâce au paramètre SENDER, mais aussi aux propriétés et méthodes de base de la classe TFORM, il va être possible de globaliser la plupart des traitements. Et si les

contraintes de développement imposaient à un moment donné d'appliquer un traitement réellement spécifique à certaines fiches, toujours grâce au SENDER, il serait possible (en utilisant l'opérateur IS) de connaître la classe de l'appelant (ce qui permettrait de traiter les cas particuliers).
L'appel des procédures CHECKPOINT se fera ainsi en passant comme paramètre l'identificateur SELF qui référencera l'instance en cours de la fiche appelante. Voici un exemple d'appel de INITCHECKPOINT dans une fiche TFORM1 :

```
Procedure TFORM1.FormCreate(Sender: TObject);
Begin
InitCheckPoint(Self) ;
... autres traitements éventuels du OnCreate de la fiche ...
End ;
```

Reprenons notre exemple de traitement de début de vie d'instance.
S'agissant des bulles d'aide, la procédure INITCHECKPOINT positionnera la propriété SHOWHINT de l'appelant en faisant référence à une variable globale de l'unité COMMON (qui sera, selon toute évidence, lue et sauvegardée depuis et sur un fichier de type INI).

Concernant les droits d'accès aux tables, second point de notre exemple, nous supposerons aussi qu'une variable de même nature existe et qu'elle contient une indication claire quant aux droits de l'utilisateur en cours.
Le code de INITCHECKPOINT pourrait alors être le suivant :

```
Procedure InitCheckPoint(Sender : Tform) ;
Var i : Integer ;
Begin
Sender.ShowHint := GLOBAL_SHOWHINT ;
For i :=0 to Sender.ComponentCount -1 do
        Try
                If (Sender.Components[i] is TDataSet)
                and (Sender.Components[i].Tag<>K_NOINITCHECKPOINT) then
                (Sender.Components[i] as TDataSet).ReadOnly := GLOBAL_READONLY ;
        Except End ;
End ;
```

Explications :
La variable GLOBAL_SHOWHINT est de type BOOLEAN, elle se trouve dans l'unité COMMON du projet et indique l'état de la visibilité des bulles d'aide.

La boucle FOR balaye tous les composants appartenant au SENDER (donc la fiche faisant l'appel). Comme le OWNER des TDATASET est systématiquement le TFORM lui-même, tous les composants gérant des données seront trouvés dans la liste COMPONENTS de l'appelant. Le fait de tester le type TDATASET permet d'englober en un seul traitement tous les descendants de cette classe, donc le TTABLE, le TQUERY et le TSTOREDPROC. Il serait possible de ne tester que l'un de ces types ou bien d'appliquer un traitement différent en les testant tous séparément.

La constante K_NOINITCHECKPOINT contient une valeur LONGINT arbitraire (je conseille une valeur limite comme MAXLONGINT définie par DELPHI). Tout composant de type dérivé TDATASET qui possédera un TAG ayant cette valeur échappera aux traitements de la procédure. Cela est très utile pour gérer les cas d'exception qui, de ce fait (par leur TAG) seront clairement identifiés.

Chacun a, bien entendu, la possibilité de sophistiquer à souhait en déclarant une étendue de valeurs qui permettrait de gérer des cas exceptionnels de nature différente. Il est certain que cet aspect n'est pas essentiel car, justement, il sera très facile de le prendre en compte à tout moment dans la procédure INITCHECKPOINT, d'où l'intérêt énorme de cette technique.

La procédure EXITCHECKPOINT sera, de toute évidence, gérée de la même façon. L'exemple d'une telle procédure ayant été promis, le voici :
Il est généralement intéressant de proposer à l'utilisateur (ou, au minimum, à l'administrateur de l'application) la possibilité de décider si la fermeture des fiches entraîne l'annulation ou la validation des opérations non validées sur les tables.
A noter que certaines applications possèdent des systèmes de saisie compliqués auxquels la présente solution ne convient guère. Mais dans tous les autres cas (nombreux) il existe une solution efficace qui sera codée dans la procédure EXITCHECKPOINT de la façon suivante :

```
Procedure ExitCheckPoint(Sender : Tform) ;
Var i : Integer ;
Begin
For i :=0 to Sender.ComponentCount -1 do
Try
If (Sender.Components[i] is TDataSet)
and (Sender.Components[i].Tag<>K_NOEXITCHECKPOINT)
        then if (Sender.Components[i] as TDataSet).State in [dsEDIT,dsINSERT]
                then
                if GLOBAL_CANCEL
                then (Sender.Components[i] as TDataSet).Cancel
                else (Sender.Components[i] as TDataSet).POST ;
Except End ;
End ;
```

Explications :
La constante K_NOEXITCHECKPOINT joue un rôle identique que celui de K_NOINITCHECKPOINT.
La variable GLOBAL_CANCEL est un booléen qui contient VRAI si la fermeture des fiches entraîne une annulation des saisies en cours et FAUX si cette fermeture entraîne la validation des saisies.
Ici aussi on peut sophistiquer les traitements à souhait selon les exigences de l'application.

<<Fin provisoire>>